TigerJython4Kids

www.tigerjython4kids.ch

Teil 1: Turtlegrafik

Jarka Arnold
Aegidius Plüss

Eine exemplarische Einführung ins Programmieren
mit Python und der Lernumgebung TigerJython

www.tigerjython4kids.ch

Inhalt

Version 2.1, August 2019

Autoren: Jarka Arnold, Aegidius Plüss
Kontakt: help@tigerjython.com

VORWORT DER AUTOREN

Der Computer gehört zu den wichtigsten Errungenschaften unserer modernen Gesellschaft. Er ist in der heutigen Welt allgegenwärtig, manchmal als Desktop, Notebook, Tablet oder Smartphone direkt sichtbar, öfters aber in vielen technischen Systemen verbaut, von der Waschmaschine, über das Auto bis zum Haushaltroboter. Der moderne Mensch ist also ständig mit Computern und computerisierten Geräten und Maschinen in Berührung und muss daher wissen, wie man damit umgeht. Diese haben oft ihre Tücken und gehorchen einer speziellen Logik. Kennt man die Grundkonzepte der Informatik, so versteht man diese Logik viel besser und akzeptiert den Computer als hilfreichen Partner im persönlichen und beruflichen Leben.

Trotz aller Begeisterung, ja Ehrfurcht vor den Möglichkeiten der heutigen Computertechnik bleibt ein Computer eine Ansammlung von einfachen elektronischen Bauteilen, wie Transistoren, Widerständen und Kondensatoren, die geschickt zusammengebaut und miniaturisiert sind. Ihre Intelligenz und Vielseitigkeit erhalten sie erst durch Programme, die von Menschen konzipiert und gemacht wurden. Erst Programme machen den Computer zu einem brauchbaren Gerät und geben ihm ein scheinbar intelligentes Verhalten. Indem man weiss, wie man einen Computer programmiert, wird man auch fähig sein, Möglichkeiten und Grenzen heutiger und zukünftiger Computertechnologien richtig einzuschätzen.

Der Lehrgang ist in Anlehnung an die Programmiersprache LOGO spielerisch aufgebaut und wendet sich an Programmieranfänger ohne informatische Vorkenntnisse ausser etwas Übung auf dem Niveau einer einfachen Textverarbeitung. Wir beschränken uns hier bewusst auf die zwei Themen Turtlegrafik und Robotik, verwenden dazu aber im Gegensatz zu vielen anderen Lernsystemen keine spezielle, erziehungsorientierte Programmiersprache, sondern die bekannte universelle und professionelle Programmiersprache Python. Wie sich zeigen wird, ist ein Einstieg ins Programmieren mit Python nicht komplizierter als mit speziell für Kinder entwickelten Sprachen und Lernumgebungen. Der grosse Vorteil ist aber, dass die hier präsentierten Denkweisen und Verfahren so allgemein gültig sind, dass sie sich unmittelbar auf einen anderen Computereinsatz oder eine andere Programmiersprache übertragen lassen.

Genau dies ist unsere Absicht und unsere Zielsetzung: Dieser Informatiklehrgang soll nicht auf kurzlebigen Trends und speziellen didaktischen Lernsystemen aufbauen, sondern eine allgemeine Denkschulung sein, die nachhaltig ist.

Albert Einstein sagte einmal im Zusammenhang mit der komplizierten Materie der Relativitätstheorie: "Man sollte alles so einfach wie möglich machen, aber nicht einfacher". Wir befolgen in diesem Lehrgang diesen Rat und machen die Programmierung so einfach wie möglich, werden aber nie etwas fachlich Falsches sagen. Wir wissen auch, dass für Anfänger und Fortgeschrittene das Erstellen von korrekten und funktionstüchtigen Programmen mit einer echten intellektuellen Herausforderung verbunden ist.

Deine Anstrengungen sollen aber dadurch belohnt werden, dass du selbst konzipierte und selbst geschriebene Programme erstellst, die dir Freude machen und die du auch deinen Mitmenschen, seinen es Jugendliche oder Erwachsene vorführen kannst. In diesem Sinn wünschen wir dir also: *Viel Spass beim Programmieren!*

Jarka Arnold, Aegidius Plüss

EINRICHTEN

DU LERNST HIER...

wie du auf deinem Computer TigerJython installieren und erste Programme ausführen kannst. TigerJython ist eine einfache Entwicklungsumgebung für das Programmieren mit Python. Sie enthält alle für die Programmierung notwendigen Komponenten, ist gratis und funktioniert problemlos unter Windows, MacOS und Linux.

EINRICHTUNG

Die Distributionen verwenden ein plattformspezifisches eingebettetes Java Runtime Environment (JRE). Es ist daher keine Vorinstallation des JRE notwendig. Lade die passende Distribution herunter und installiere TigerJython in einem beliebigen Verzeichnis. Die Installationsdateien sind mit einem offiziellen Zertifikat signiert und stellen keine Gefahr für deinen Computer dar.

Download
TigerJython für Windows (64 bit)

Download
TigerJython für Windows (32 bit)

Download
TigerJython für MacOS (V 10.8 up)

Download
TigerJython für Linux (64 bit)

Download
TigerJython für Linux (32 bit)

Bei der Installation wird die eigentliche Programmdatei ***tigerjython2.jar*** in ein Unterverzeichnis *bin* des Installationsverzeichnisses kopiert. Für einen **Update** von TigerJython kannst du nur *tigerjython2.jar* downloaden und ersetzen.

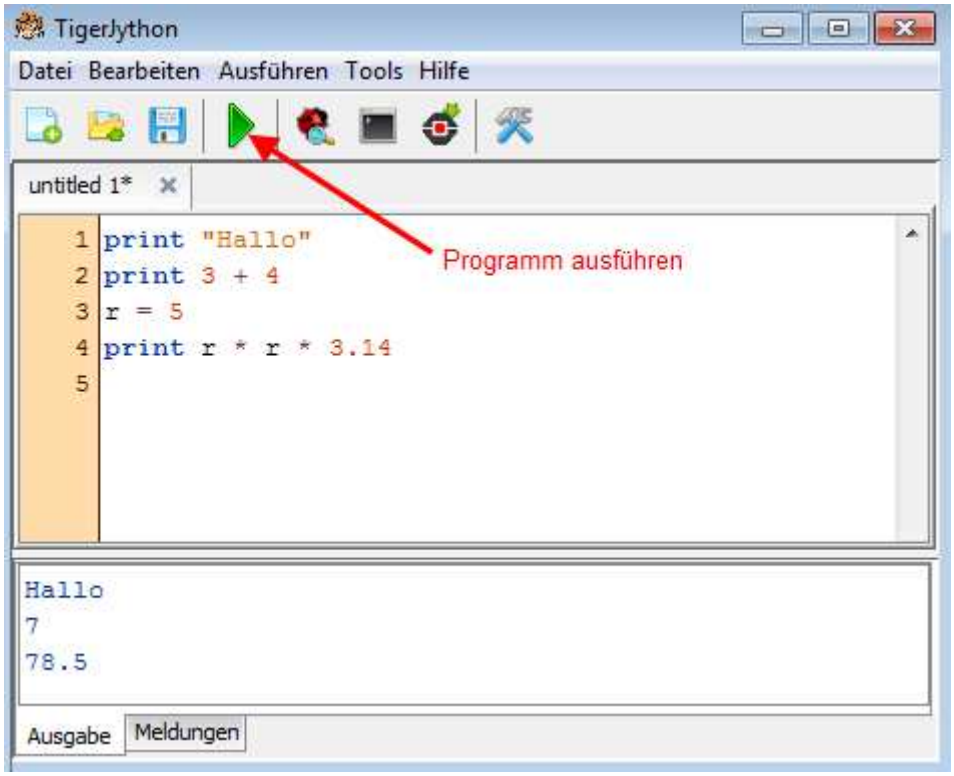
Download tigerjython2.jar

Beachte, dass TigerJython mit Doppelklick auf die Applikation mit der TigerJython-Ikone und nicht mit Doppelklick auf *tigerjython2.jar* gestartet wird.

Anmerkung:
In einem Computerpool, in dem das JRE Version 1.7 oder 1.8 bereits installiert ist, genügt es, *tigerjython2.jar* herunterzuladen und auszuführen.

MUSTERBEISPIEL

Der wichtigste Teil des Applikationsfensters ist das Editorfeld, wo du mit Tastatur und Maus Programme eingibst.



Teste die Funktionalität, in dem du einige print-Befehle eingibst und auf die grüne Schaltfläche *Programm ausführen* klickst. Die Bedienung des Editors ist einfach. Zur Verfügung stehen dir die Schaltflächen *Neues Dokument, Öffnen, Speichern, Programm ausführen, Debugger ein-/ausschalten, Konsole anzeigen* und *Einstellungen*.

Beim Editieren kannst du die üblichen Tastenkürzel verwenden:

Ctrl + C	Kopieren
Ctrl + V	Einfügen
Ctrl + X	Ausschneiden
Ctrl + A	Alles Markieren
Ctrl + Z	Rückgängig

ZUM SELBST LÖSEN

Ein Programm ist eine Anleitung an den Computer, was er **schrittweise** zu tun hat. Dies musst du ihm in einer genau festgelegten Sprache mitteilen, die keinerlei Schreibfehler zulässt. Er weisst alle Schreibfehler als **Syntaxfehler** zurück, selbst wenn er eigentlich schon herausfinden könnte, welchen Fehler du gemacht hast. Auch die Einhaltung der **Gross-Kleinschreibung** ist ganz wesentlich!

Die Programmiersprache Python hat verglichen mit einer Umgangssprache einen winzigen Wortschatz aus rund **30 Keywords**. Es ist also für dich vom Wortschatz her ein Vielfaches einfacher, ein Computerprogramm als einen Text in einer Fremdsprache zu schreiben.

Eines der Keywords ist ***print***. Mit diesem kannst du den Computer anweisen, etwas in das Ausgabefenster zu schreiben. Versuche es einmal mit einem ersten Programm, das die Summe der Zahlen a = 3 und b = 7 zu berechnen. Du schreibst also im Editor

```
a = 3
b = 7
print a + b
```

und drückst dann den grünen Run-Button. Im Ausgabenfenster siehst du die Zahl 10. Damit du das Programm später wieder verwenden kannst, solltest du es unter einem beliebigen Namen speichern. Wähle dazu im Menü "Datei | Speichern" und gib einen aussagekräftigen Namen (ohne Erweiterung) ein, beispielsweise *erstes*. Das Programm wird dann unter dem Namen *erstes.py* auf der Festplatte gespeichert und kann immer wieder verwendet werden.

Schreibe nun auch ein paar andere Programme, beispielsweise um die Zahlen a und b zu multiplizieren oder zu dividieren. Im letzten Fall siehst du, dass eine Dezimalzahl ausgeschrieben wird, die den exakten Wert nur **approximiert**, der Computer also in bestimmten Fällen nicht exakt rechnet. Falls du Text ausschreiben willst, musst du diesen in Anführungszeichen (einfache oder doppelte) setzen. Du kannst mit *print* auch gleichzeitig mehrere Ausgaben machen, indem du sie mit Komma trennst, also zum Beispiel *print a, b*.

DOKUMENTATION UND HILFE

Unter dem Menüpunkt Hilfe stehen zusätzliche Information zur Verfügung, insbesondere eine Dokumentation der Zusatzmodule und der Bildbibliothek. In einer zukünftigen Version von TigerJython wird die Dokumentation auch beim Drücken von F1 angezeigt, wenn sich der Mauscursor innerhalb des Funktionsnamens befindet und sich die Funktion im aktuellen Namensraum befindet.

AUTO-VERVOLLSTÄNDIGUNG

Ist man bei der Schreibweise von Funktionen nicht ganz sicher, so kann man nach der Eingabe von einigen Anfangsbuchstaben die Tastenkombination *Ctrl+Space* drücken. Es öffnet sich dann ein Fenster mit Funktionsvorschlägen aus dem aktuellen Namensraum auf der Basis der standardmässig bekannten oder importierten Module.

DOWNLOAD DER BEISPIELPROGRAMME

Wir schlagen dir vor, den Lehrgang kapitelweise durchzuarbeiten und dabei die Beispielprogramme mit dem Link *Programmcode markieren , Ctrl+C* und *Ctrl+V* einzeln in den TigerJython-Editor zu übernehmen, unter einem geeigneten Namen zu speichern und dann auszuführen.

Du kannst aber auch alle Programmbeispiele aus diesem Lehrgang hier downloaden.

LOSLEGEN!

MIT PROGRAMMIEREN LOSLEGEN...

Zur Verfügung stehen dir drei Lernprogramme, die du direkt von der Startseite mit Klick auf die Tabs "Turtlegrafik", "Robotik" oder "Datenbanken" starten kannst.

Mit **Turtlegrafik** lernst du anschaulich die wichtigsten Programmierkonzepte kennen. Es sind keine Programmiervorkenntnisse erforderlich. Im Lernprogramm findest du zahlreiche lauffähige Musterbeispiele und Aufgaben zum Selbstlösen.

KONZEPT DES LEHRGANGS

Alle Kapitel folgen einer einheitlichen Struktur:

DU LERNST HIER...

Es handelt sich um eine kurze Kapitelübersicht und dient vor allem der Motivation.

MUSTERBEISPIEL

Hier werden an einem exemplarisch ausgewählten Programm die neuen Begriffe und Verfahren eingeführt. Der vollständig lauffähige Programmcode kann mit einem Klick markiert und über die Zwischenablage in den TigerJython-Editor kopiert werden. Um die Erklärungen zum Programmcode besser folgen zu können, sind wichtige Begriffe mit dem Programmcode verlinkt.

ZUM SELBST LÖSEN

Es geht hier um Anleitungen zur Eigentätigkeit entsprechend dem Konzept: "Learning by Doing". Einige Aufgaben sind mit einen Stern versehen. Diese haben einen etwas höheren Schwierigkeitsgrad.

MERKE DIR...

sind wichtige Anmerkungen und Zusatzbemerkungen, es handelt sich aber nicht um eine Zusammenfassung des Stoffs.

ZUSATZSTOFF

enthält interessante Ergänzungen von etwas weniger grossen Wichtigkeit.

VERLINKUNG MIT OVERLAY-FENSTERN

Der Online-Lehrgang verwirklicht ein neuartiges Prinzip der Verlinkung zu weiteren Informationen mit Hilfe von Links, die im Text integriert sind und mit einem Mausklick oder Touch aktiviert werden. Dabei öffnet sich ein Overlayfenster mit fachlichen oder didaktischen Informationen und Hinweisen. Sie sind als Unterrichtshilfen gedacht und wenden sich vor allem an Lehrkräfte. Es kann sich auch um Begründungen für das von uns gewählte methodische Vorgehen handeln.

ZIELGRUPPE UND ZEITAUFWAND

Auf Grund unserer eigenen Schulerfahrung gehen wir davon aus, dass sich der Lehrgang ab dem mittleren Volksschulalter einsetzen lässt. Er ist aber interessant genug. dass er sich auch für einen später einsetzenden Anfängerunterricht eignet. In weiterführenden Schulen empfehlen wir, unser umfangreicheres Lehrmittel http://www.tigerjython.ch zu verwenden.

Für die einzelnen Kapitel gehen von einem Zeitaufwand von 1 bis höchstens 2 Lektionen aus. Bei einer Beschränkung auf die Turtlegrafik liegt der Gesamtaufwand also im Bereich von 14-20 Lektionen, also von etwa einer Semesterwochenstunde.

Im 2. Teil "Robotik" und 3. Teil "Datenbanken" gehen wir davon aus, dass die grundlegenden Konzepte aus dem 1. Teil "Turtlegrafik" bekannt sind. Es ist allerdings denkbar, in einem Informatikkurs oder einer Projektwoche direkt mit der Robotik einzusetzen, was aber mit einem zusätzlichen Aufwand für die Lehrperson verbunden ist, da sie die fehlenden Kenntnisse mit didaktischem Geschick laufend "nachreichen" muss.

LÖSUNGEN DER AUFGABEN

Sind Sie an einer Ausbildungsinstitution tätig, so können Sie die Lösungen der Aufgaben mit einer Email-Anfrage an help@tigerjython.com erhalten. Die Anfrage muss die folgenden überprüfbaren Angaben enthalten: Name, Adresse, Ausbildungsinstitution, Email-Adresse. Sie verpflichten sich dabei, die Lösungen nur für den persönlichen Gebrauch zu verwenden und nicht weiter zu geben.

URHEBERRECHTE

1. TURTLE BEWEGEN

DU LERNST HIER...

dass ein Programm aus einer Folge von Programmzeilen besteht, die der Reihe nach (als Sequenz) abgearbeitet werden. Dabei verwendest du ein Grafikfenster mit einem Turtlebild , das sich mit Befehlen ähnlich wie ein kleiner Roboter steuern lässt.

Turtlebefehle werden grundsätzlich Englisch geschrieben und enden immer mit einem Klammerpaar, das man **Parameterklammer** nennt. Dieses kann weitere Angaben für den Befehl enthalten. Selbst wenn keine Angaben nötig sind, muss in Python ein leeres Klammerpaar vorhanden sein. Wie du bereits weisst, ist es wichtig, die Gross-/Kleinschreibung exakt einzuhalten. Bei der Bewegung hinterlässt die Turtle eine Spur. Es ist so, als ob sie einen auf der Zeichenfläche aufliegenden Zeichenstift(pen) mit sich tragen würde. Damit kann sie schöne Figuren zeichnen.

MUSTERBEISPIEL

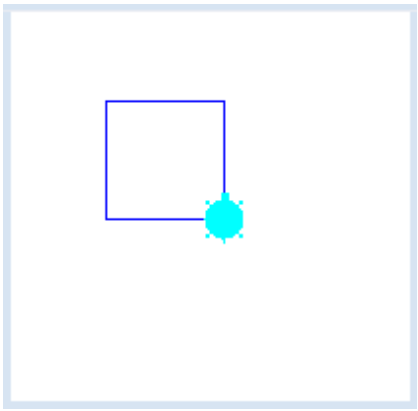
Bevor die Turtle loslegen kann, musst du den Computer anweisen, die Turtlebefehle aus einem Modul bereit zu stellen. Dazu schreibst du

```
from gturtle import *
```

und befiehlst dem Computer mit

```
makeTurtle()
```

ein Turtlefenster mit einer Turtle in Fenstermitte mit Blickrichtung nach oben (home position) zu öffnen. Du kannst diese zwei Zeilen bereits eintippen und den Run-Button drücken und es erscheint tatsächlich das Turtlefenster. Mit forward(100) bewegt sich die Turtle um 100 Schritte vorwärts, mit left(90) dreht sie um 90 Grad nach links. Mit diesen Befehlen kannst du ihr mit folgendem Programm befehlen, ein Quadrat zu zeichnen.



```
from gturtle import *

makeTurtle()
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Du kannst das Programm entweder eintippen oder aus der Vorlage kopieren. Dazu klickst du auf den Link *Programmcode markieren* und kopierst das Programm mit *Ctrl-C* in den Zwischenspeicher. Dann klickst du mit der Maus in das TigerJython-Editorfeld und fügst das Programm mit *Ctrl-V* ein. Dieses Verfahren funktioniert natürlich nur, wenn du das Lehrmittel online verwendest.

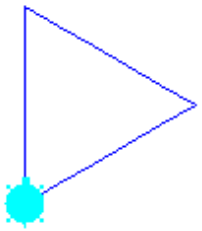
MERKE DIR…

Du musst unterscheiden zwischen dem Schreiben (Editieren) des Programms und seiner Ausführung beim Klicken des Run-Buttons.

ZUM SELBST LÖSEN

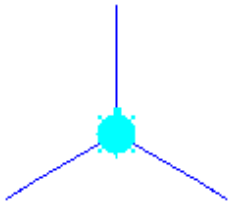
1. Mit dem Befehlt *right(90)* sagst du der Turtle, dass sie 90° nach rechts drehen soll.

   Versuche die Turtle anzuleiten, ein gleichseitiges Dreieck mit der Seitenlänge 100 zu zeichnen. Findest du den richtigen Drehwinkel?



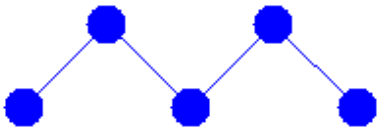
2. Die Turtle kann sich auch rückwärts bewegen. Soll sie beispielsweise 100 Schritte rückwärts gehen, so verwendest du den Befehl *back(100)*:

```
from gturtle import *

makeTurtle()
forward(100)
back(100)
```



   Zeichne die nebenstehende Figur.

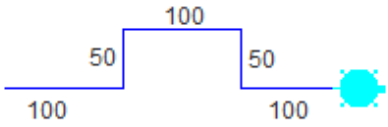
3. Mit dem Befehl *dot(20)* zeichnet die Turtle dort, wo sie sich gerade befindet, einen gefüllten Kreise mit dem Durchmesser 20.



```
from gturtle import *
makeTurtle()
forward(100)
dot(20)
```

   Kannst du die nebenstehende Figur zeichnen?

4. Zeichne die nebenstehende Figur. Du kannst die Befehle auch mit 2 Buchstaben abkürzen: *fd(100), lt(90), rt(90).*



2. FARBEN VERWENDEN

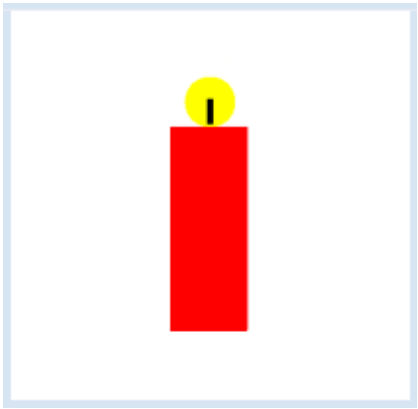
DU LERNST HIER...

einige weitere Turtlebefehle kennen. Damit kannst du schöne farbige Bilder zeichnen. Die Turtlegrafik verwendet die sogenannten *X11-Farben*; das sind einige dutzend englisch geschriebene Farbnamen. Die folgende Liste ist zwar nicht vollständig, gibt dir aber doch einen ersten Anhaltspunkt: *yellow, gold, orange, red, maroon, violet, magenta, purple, navy, blue, skyblue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white.*

MUSTERBEISPIEL

Du willst mit der Turtle das Bild einer brennenden Kerze zeichnen. Dazu brauchst du zuerst eine gute Idee, wie du das bewerkstelligen willst. Dabei spielt auch deine Phantasie und dein Erfindergeist eine wichtige Rolle. Es gibt mehrere Lösungsideen, eine davon lautet so:

Du zeichnest zuerst mit einem dicken roten Stift die Kerze selbst. Dann fährst du mit abgehobenem Stift ein wenig weiter und zeichnest einen gelb-gefüllten Kreis. Um den Docht zu malen, fährst du mit einem schwarzen, dünnen Stift leicht zurück.



Als neue Befehle brauchst du penUp() und penDown() um den Stift hochzuheben und ihn wieder abzusenken, sowie setLineWidth() für die Stiftdicke und setPenColor() für die Stiftfarbe. Zuletzt versteckst du noch mit hideTurtle() das Turtlebild.

```
from gturtle import *

makeTurtle()
setLineWidth(60)
setPenColor("red")
forward(100)
penUp()
forward(50)
penDown()
setPenColor("yellow")
dot(40)
setLineWidth(5)
setPenColor("black")
back(15)
hideTurtle()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

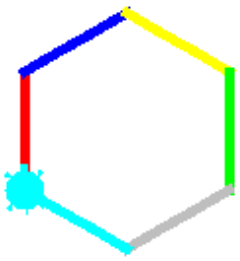
MERKE DIR…

Das Zeichen # wird in Python für Kommentare verwendet. Du kannst damit Anmerkungen in den Programmtext einfügen, die für die Programmausführung keine Rolle spielen.

Auf der Webseite http://cng.seas.rochester.edu/CNG/docs/x11color.html findest du noch viele andere Farben, die du als Stift- und Füllfarben verwenden darfst. Bei den Farbnamen spielt die Gross-/Kleinschreibung keine Rolle, sie müssen aber in Anführungszeichen stehen.

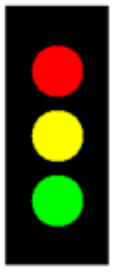
ZUM SELBST LÖSEN

1. Ergänze das untenstehende Programm so, dass die Turtle ein regelmässiges Sechseck zeichnet und wähle für jede Seite eine andere Farbe.



```
from gturtle import *
makeTurtle()
setPenColor("red")
forward(80)
right(60)
```

2. Zeichne eine Verkehrsampel. Das schwarze Rechteck kannst du mit der Stiftbreite 80 zeichnen, die Kreise mit *dot(40)*.



3. Zeichne mit einem dicken roten Stift ein rotes Kreuz.

4. Zeichne eine deutsche Flagge oder eine einfache andere Landesflagge.

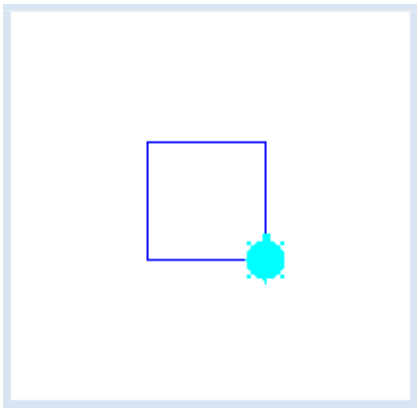
3. WIEDERHOLUNG

DU LERNST HIER...

dass du eine oder mehrere Programmzeilen zu einem Programmblock zusammenfassen und ihn dann eine bestimmte Anzahl mal wiederholt durchlaufen kannst. Dadurch ersparst du dir viel Schreibarbeit und das Programm wird übersichtlicher.

MUSTERBEISPIEL

Um ein Quadrat zu zeichnen, muss die Turtle vier Mal die Befehle forward(100) und left(90) ausführen. Du kannst dies in TigerJython elegant mit repeat programmieren.



```
from gturtle import *

makeTurtle()
repeat 4:
  forward(100)
  left(90)
```

Programmcode markieren (Ctrl+C kopieren)

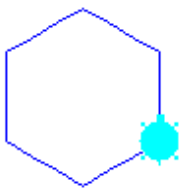
Die Wiederholung wird mit repeat 4: eingeleitet. Dabei ist der Doppelpunkt sehr wichtig. Vergisst du ihn, so ergibt sich bei der Programmausführung eine Fehlermeldung:

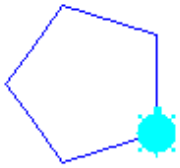
```
repeat 4
Am Ende der Zeile fehlt ein Doppelpunkt ':'.
```

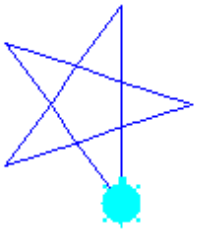
Die Befehle im nachfolgenden **Programmblock** musst du alle **gleichweit einrücken**. Du verwendest dazu immer 4 Leerschläge, du kannst aber auch die Tabulator-Taste brauchen, um sie zu erzeugen. Man spricht bei der Wiederholstruktur auch vom **Durchlaufen einer Schleife**.

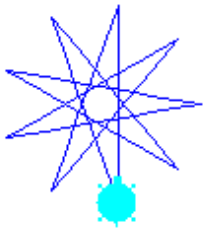
ZUM SELBST LÖSEN

1. Experimentiere mit dem Programm aus dem Musterbeispiel. Ändere die Anzahl Wiederholungen und den Drehwinkel so, dass die Turtle die folgenden Figuren zeichnet.

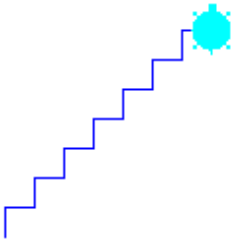
a) 

b) 

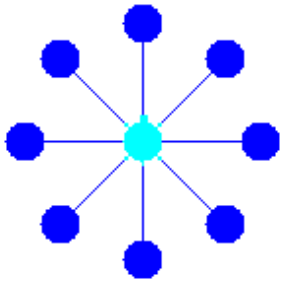
c) 

d) 

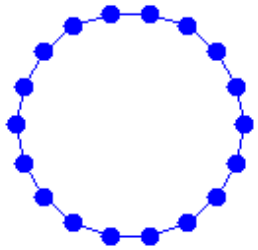
2. Zeichne eine Treppe mit 7 Stufen.



3. Zeichne die nebenstehende Figur. Dazu brauchst du auch die Befehle *back()* und *dot()*.



4. Zeichne eine Perlenkette, die aus 18 Perlen (dots) besteht. Zwischen den Perlen muss die Turtle jeweils einige Schritte vorwärts gehen und um einen kleinen Winkel (z.B. 20°) nach links drehen.

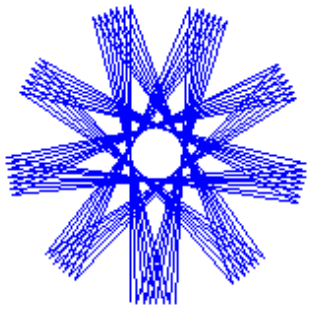


5. Nach einer Idee von Joshua Goldstein ergeben sich hübsche Bilder, wenn die Turtle wiederholt forward-right-Befehlspaare ausführt. Zeichne die Grafiken mit

   a) *forward(300)* , *right(151)* und 92 Wiederholungen

   b) *forward(200)*, *right(159.72)* und 63 Wiederholungen. Du kannst du Turtle verstecken und sie zu Beginn noch mit *back()* nach unten versetzen, damit die Grafik im Fenster besser eingepasst ist.

   c) Suche über eine Internet-Suchmaschine mit den Stichworten *goldstein turtle* den Originalartikel von J. Goldstein und erstelle einige weitere von dort inspirierten Bilder (auch mit mehreren *forward-right*-Paaren).



ZUSATZSTOFF: VERSCHACHTELTE SCHLEIFEN

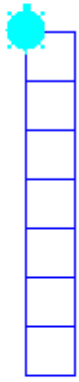
Richtig spannend und anspruchsvoll wird es, wenn du zwei repeat-Strukturen ineinander verschachtelst. Du musst dann immer denken, dass zuerst die "innere", weiter eingerückte Wiederholstruktur durchlaufen wird, bevor sich die "äussere", weniger eingerückte Struktur wiederholt. Versuche zu verstehen, was hier ausgeschrieben wird.

```
repeat 2:
    print "aussen-vorher"
    repeat 3:
        print "innen"
    print "aussen-nachher"
```

Den zeitlichen Ablauf kannst du an folgender Aufnahme erkennen, wo der Pfeil zeigt, welche Zeile in einem bestimmten Moment ausgeführt wird.

```
1 repeat 2:
2     print "aussen-vor"
3     repeat 3:
4         print "innen"
5     print "aussen-nach"
6
```

Mit diesem Kenntnissen bist du jetzt in der Lage, die folgende Leiter mit zwei ineinander geschachtelten repeat-Strukturen zu zeichnen. Die Idee ist die Folgende: Die innere Schleife zeichnet ein einzelnes Quadrat und die Turtle befindet sich nachher wieder in der linken unteren Ecke des Quadrats. Sie wird dann in der äusseren Schleife vorgeschoben und das Quadrat wird erneut gezeichnet.



```
from gturtle import *

makeTurtle()
repeat 7:
    repeat 4:
        forward(30)
        right(90)
    forward(30)
```

Programmcode markieren (Ctrl+C kopieren Ctrl+V einfügen)

ZUM SELBST LÖSEN

6. Versuche zuerst auf einem Blatt Papier herauszufinden, was das folgende Programm zeichnet. Lass es dann laufen, um deine Vermutung zu bestätigen.

```
from gturtle import *
makeTurtle()

repeat 5:
    repeat 4:
        forward(100)
        right(90)
    left(36)
```

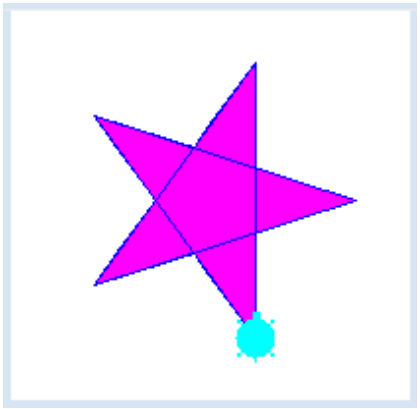
4. FIGUREN FÜLLEN

DU LERNST HIER...

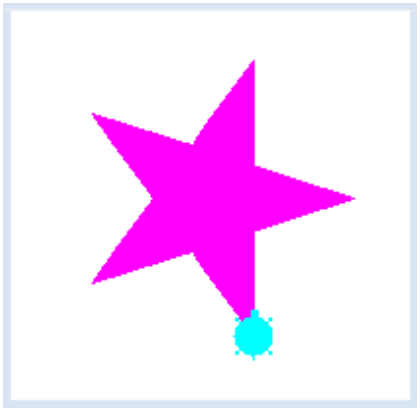
dass sich die Turtle an ihre Bewegung erinnern kann, um so eine von ihr gezeichnete geschlossene Figur mit einer Farbe zu füllen.

MUSTERBEISPIEL

Um eine Figur auszufüllen, sagst du zuerst der Turtle mit startPath(), dass sie sich ausgehend vom aktuellen Ort die nachfolgend gezeichnete Figur merken soll. Mit dem Befehl fillPath() wird der jetzige Ort mit dem Startort verbunden und die eingeschlossene Fläche ausgefärbt. Mit setFillColor() kannst du die Füllfarbe angeben (sagst du nichts, so ist sie standardmässig blau).



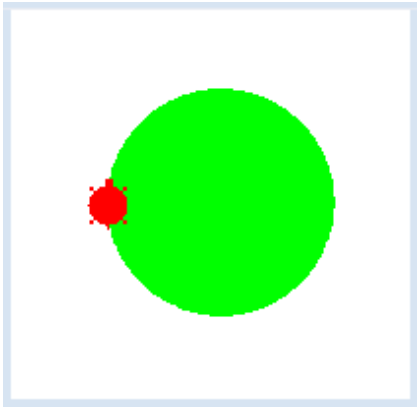
```
from gturtle import *

makeTurtle()

setFillColor("magenta")
#setPenColor("magenta")
startPath()
repeat 5:
    forward(160)
    left(144)
fillPath()
```



Willst du keine sichtbaren Umrisslinien, so musst du die auskommentierte Zeile aktivieren, d.h. das Zeichen # löschen. Damit ist die Umrissfarbe gleich der Füllfarbe.

KREIS ALS VIELECK

Du kannst mit der Turtle einen gefüllten Kreis als Vieleck mit sehr vielen Seiten zeichnen. Willst du keine sichtbaren Umrisslinien, wählst du wie vorhin entweder die gleiche Stift- und Füllfarbe oder hebst den Stift ab. Mit setColor() änderst du zudem noch die Farbe der Turtle.

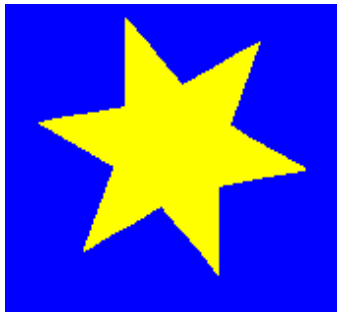


```
from gturtle import *

makeTurtle()

setColor("red")
setFillColor("green")
setPenColor("green")
startPath()
repeat 120:
    forward(3)
```

```
    right(3)
fillPath()
```

ZUM SELBST LÖSEN

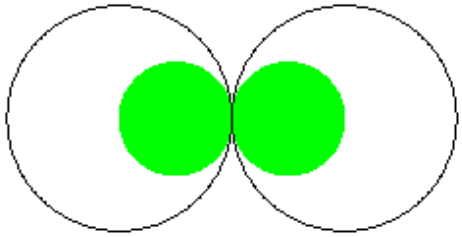
1. Um den nebenstehenden 6er-Stern zu zeichnen, dreht die Turtle abwechslungsweise 140 und 80 Grad. Du kannst das ganze Turtlefenster mit clean("blue") blau anmalen.



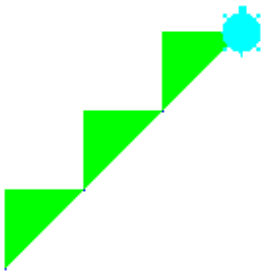
```
from gturtle import *
makeTurtle()
clean("blue")
setFillColor("yellow")
....
```

2. Zeichne zwei gefüllte Halbkreise. Wenn du die Turtle mit *hideTurtle()* versteckst, wird das Bild viel schneller gezeichnet.

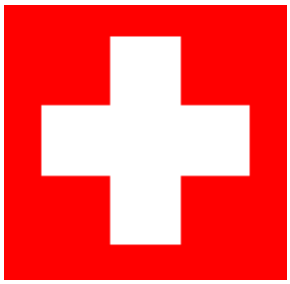
3. Diese lustige Figur besteht aus gefüllten und nicht gefüllten Kreisen. Zeichne sie wie im Musterbeipiel mit Vielecken.



4 Zeichne das nebenstehende Bild.



5. Um ein Schweizerkreuz zu zeichnen, verwendest du zwei ineinander geschachtelte *repeat*-Schleifen. Den Bildschirm malst du vorher rot an. Ergänze das folgende Programmskelett.



```
from gturtle import *
makeTurtle()
clean("red")
setPenColor("yellow")
setFillColor("yellow")
startPath()
repeat 4:
    repeat 3:
        ....
```

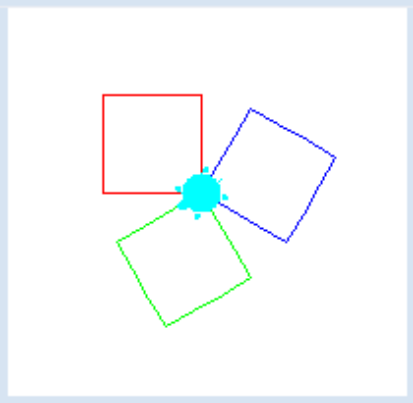
5. FUNKTIONEN

DU LERNST HIER...

wie man Programme strukturiert, indem man eigene Befehle, in Python Funktionen genannt, definiert. Eine Funktion ist eigentlich ein benannter Programmblock, den man mit seinem Namen ausführen lassen kann.

MUSTERBEISPIEL

In den vorhergehenden Kapiteln hast du vielfach als Teilaufgabe ein Quadrat gezeichnet. Es ist daher sinnvoll, dieser Teilaufgabe einen Namen *square* zu geben, damit du den Programmcode immer wieder unter diese Namen verwenden kannst.



Die Funktionsdefinition beginnt die mit dem Keyword def. Dann folgt der Name, eine sogenannte Parameterklammer und ein Doppelpunkt. Nach diesem Funktionskopf folgt der Funktionskörper, der aus irgendwelchem Programmcode besteht.

Die Funktionsdefinition def square(): legt allerdings nur fest, WIE ein Quadrat zu zeichnen ist. Das Quadrat wird aber erst weiter unten im Programm tatsächlich gezeichnet, nämlich beim Funktionsaufruf. Da Funktionsdefinitionen vor dem Aufruf stehen müssen, setzen wir alle Funktionsdefinitionen gemeinsam in den obersten Teil des Programms.

In deinem Programm rufst du *square()* dreimal auf.

```
from gturtle import *

def square():
    repeat 4:
        forward(100)
        left(90)

makeTurtle()
setPenColor("red")
square()
right(120)
setPenColor("blue")
square()
right(120)
setPenColor("green")
square()
```

Programmcode markieren (Ctrl+C kopieren)

MERKE DIR…

Die Funktionsnamen kannst du ziemlich frei und nach deinem eigenen Geschmack wählen, du musst dich aber an ein paar Einschränkungen halten. Erlaubte **Bezeichner** sind Wörter

- ohne Umlaute und Spezialzeichen, ausser $ und _ (also mit den Buchstaben a..z, A..Z, den Zahlen 0..9, sowie $ und _).
- Sie dürfen nicht mit einer Zahl beginnen und
- dürfen keine Keywords sein.

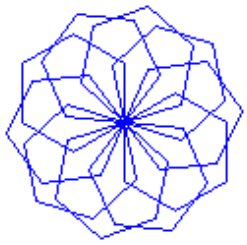
Erlaubt sind also: *quadrat*, *zeichneKreis, mySub1, print_result*
Nicht erlaubt sind: *1wochentag, im.dezember, führeaus, komm her*

Wir halten uns zudem an die Regel, dass Bezeichner im *CamelCase* geschrieben werden. Dabei werden Wortgruppen aneinander gefügt und ein neues Wort beginnt immer mit einem Grossbuchstaben, also beispielsweise *fillSquare, myFirstFunction, RobotClass*. Dadurch werden die Programme auch für andere Personen besser lesbar.

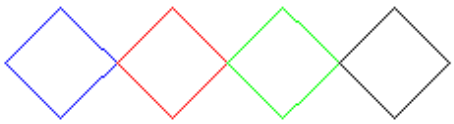
Schliesslich ist es ein weit verbreiterter Standard, Funktionsbezeichner immer mit einem Kleinbuchstaben (oder notfalls dem _) zu beginnen, also *drawFigure* und nicht *DrawFigure*. Da Funktionsbezeichner möglichst vielsagend sein sollten, solltest du eher Verbformen als Substantive einsetzen, also *drawFigure* eher als *figure*, zudem eignen sich englischsprachliche Bezeichner besser, da so deine Programme universell lesbar sind.

ZUM SELBST LÖSEN

1. Definiere einen Befehl *sechseck()*, mit dem die Turtle ein Sechseck zeichnet. Verwende diesen Befehl, um die nebenstehende Figur zu erstellen.



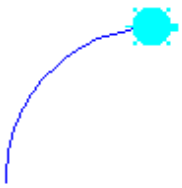
2a. Definiere einen Befehl für ein Quadrat, das auf der Spitze steht und zeichne damit die nebenstehende Figur.



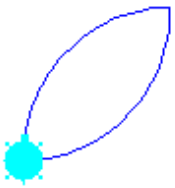
2b. Zeichne gefüllte Quadrate.

3a. Du erlebst in der folgenden Aufgabe, wie du unter Verwendung von Funktionen ein Problem schrittweise lösen kannst.

Definiere eine Funktion *bogen()*, mit der die Turtle einen Bogen zeichnet und sich dabei insgesamt um 90 Grad nach rechts dreht. Mit *speed(-1)* bewegt sich die Turtle schneller.



3b. Ergänze das Programm mit der Funktion *blumenblatt()*, welche zwei Bogen zeichnet. Die Turtle sollte am Ende aber wieder in Startrichtung stehen.



3c. Ergänze das Programm so, dass *blumenblatt()* ein rot gefülltes Blatt (ohne sichtbare Umrandungslinie) zeichnet.

3d. Erweitere das Programm mit der Funktion *blume()* so, dass eine 5-blättrige Blume entsteht. Damit die Blume noch schneller erscheint, kannst du mit *hideTurtle()* die Turtle bereits am Anfang unsichtbar machen.

3e*Mit *setRandomPos(600, 400)* kannst du die Turtle an eine zufällige Position im x-Bereich zwischen -300 und 300 und y-Bereich zwischen -200 und 200 setzen. Zeichne damit 10 Blumen an zufälligen Positionen. (Die Grösse des Turtlefensters hängt von der Wahl in den Einstellungen von TigerJython ab.)

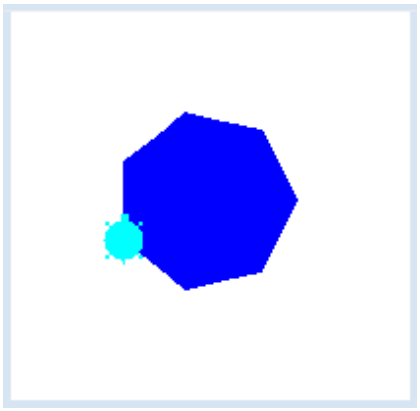
6. VARIABLEN

DU LERNST HIER...

den wichtigen Begriff der Variablen kennen. Darunter versteht man in der Informatik Namen, die als Platzhalter für Werte dienen, die sich im Laufe der Programmausführung ändern können.

MUSTERBEISPIELE

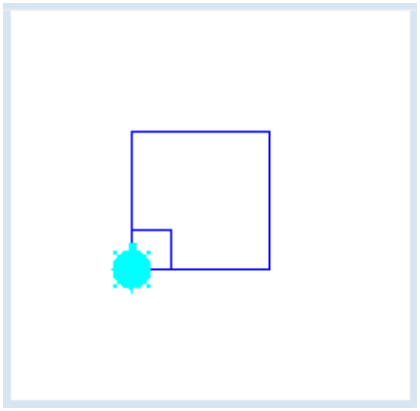
Damit du das Programm flexibel verändern kannst, willst du die Anzahl der Ecken eines n-Ecks mit einer Variablen *n* festlegen. Dazu machst du eine **Zuweisung**, beispielsweise n = 7 für ein Siebeneck. Dabei wird der Name *n* mit der Zahl 7 verbunden und *n* kann dann im Programm mehrmals gelesen, aber auch verändert werden. Um die Grafik noch etwas attraktiver zu machen, zeichnest du ein gefülltes n-Eck, indem du am Anfang mit fillToPoint() sagst, dass alle nachfolgend gezeichneten Linien zum Anfangspunkt, hier der Homeposition, gefüllt werden.



```
from gturtle import *

makeTurtle()
fillToPoint()
n = 7
repeat n:
    forward(100)
    right(360 / n)
```

Du kannst jetzt leicht auch beliebige andere n-Ecks zeichnen. Mache dies für ein paar andere Werte von *n*. Im Gegensatz zur Mathematik kann der **Variablenname**, auch **Variablenbezeichner** genannt, aus einem ganzen Wort bestehen. Bei der Wahl des Namen hältst du dich an die gleichen Regeln wie bei der Wahl von Funktionsnamen.

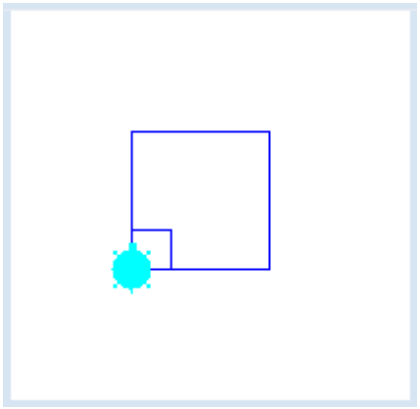
Der Wert der Variablen kann sich im Laufe des Programms durch eine erneute Zuweisung auch ändern. Beispielsweise für die Seitenlänge size eines Quadrats:



```
from gturtle import *

makeTurtle()
size = 20
repeat 4:
   forward(size)
   right(90)
size = 70
repeat 4:
   forward(size)
   right(90)
```

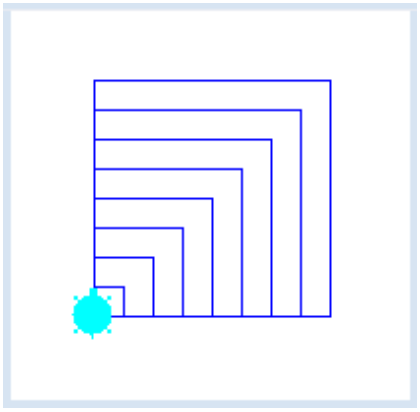
Du kannst bei der erneuten Zuweisung sogar den alten Wert verwenden, also size = size + 50 schreiben. Dabei musst du dir vorstellen, dass schrittweise Folgendes geschieht:

- der alte Wert 20 von size wird in ein Rechenwerk übertragen
- die Zahl 50 wird dazugezählt
- der neue Wert 70 wird wieder als size abgelegt.



```
from gturtle import *

makeTurtle()
size = 20
repeat 4:
   forward(size)
   right(90)

size = size + 50
repeat 4:
   forward(size)
   right(90)
```

Diese Schreibweise ist sehr elegant. Beispielsweise kannst du damit die Turtle mit einem kurzen Programm 10 ineinander geschachtelte Quadrate zeichnen lassen:



```
from gturtle import *

makeTurtle()
size = 20
repeat 10:
    repeat 4:
        forward(size)
        right(90)
    size = size + 20
```

MERKE DIR...

Eine Variable entsteht dann, wenn du ihr mit dem Gleichheitszeichen einen Wert zuweist. Du kannst ihren Wert jederzeit durch eine neue Zuweisung ändern und dabei sogar ihren eigenen (alten) Wert gebrauchen. Du darfst die dabei verwendete Schreibweise nicht mit einer mathematischen Gleichung verwechseln. Die Programmanweisung:

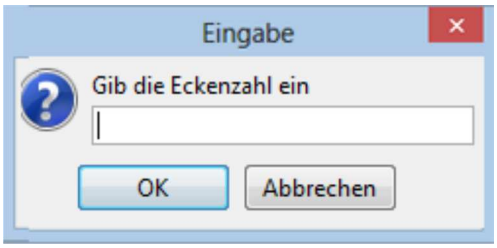
n = n + 1

hat **nichts** mit einer mathematischen Gleichung zu tun, also mit der Aufgabe, *n* so zu bestimmen, dass sich links und rechts von Gleichheitszeichen derselbe Wert ergibt.

Fürt *n = n + 1* gibt es noch die Kurzschreibweise *n += 1* , die genau das Gleiche macht.

ZUM SELBST LÖSEN

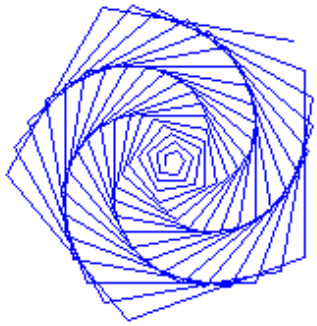
1. Variablen werden auch zum Einlesen vor Werten verwendet. Mit der Anweisung



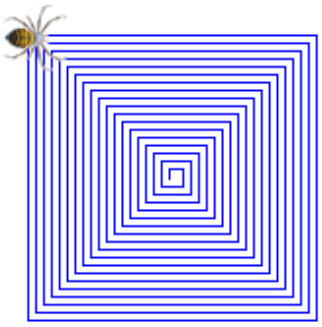

```
n = inputInt("Gib die Eckenzahl ein")
```

   öffnet sich ein Eingabedialog, wo du eine Zahl eingeben kannst. Beim Drücken des OK-Buttons schliesst sich das Dialogfenster und der eingegebene Wert wird der Variablen *n* zugewiesen. Schreibe damit ein Programm, bei dem der Benutzer die Eckenzahl des gefüllten n-Ecks interaktiv eingeben kann.

2. Mit Variablen kannst du mit wenig Aufwand schöne Grafiken erstellen. Das nebenstehende Bild entsteht, indem du die Turtle 250 mal um eine Strecke *s* vorwärts laufen und dann 70 Grad drehen lässt, wobei du bei jedem Durchgang die Strecke um 1 erhöhst. Die anfänglich Streckenlänge ist 5. Mit *hideTurtle()* wird die Zeichnung schnell gezeichnet. Schreibe das Programm und mache einige selbst erfundene Varianten davon, um andere ähnliche Figuren zu erhalten.



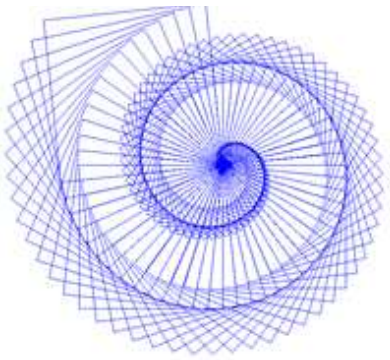
3. Durch Angabe eines Spritebildes in *makeTurtle()* kannst du das Bild der Turtle beliebig verändern. Die in TigerJython enthaltenen Bilder und ihre Namen findest du in der Menüoption *Hilfe* | *APLU Dokumentation* unter *Bildbibliothek*. Du musst den Bildnamen noch *sprites* vorstellen und das ganze in Gänsefüsschen setzen, also für das Spinnenbild schreiben:



```
makeTurtle("sprites/spider.png")
```

   Lass die Spinne das nebenstehende Netz spinnen. Wähle noch andere Spritebilder, z.B. *beetle.gif*.

4. Die folgende hübsche Figur entsteht, indem die Turtle 120 Quadrate mit der Seitenlänge *s* zeichnet. Dabei wird bei jedem Durchlauf die Seitenlänge um 2 erhöht und die Turtle um 6 Grad gedreht. Schreibe das Programm.



7. PARAMETER

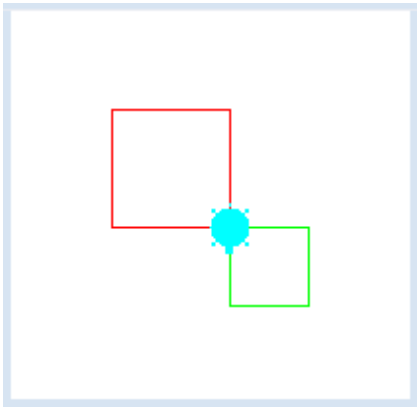
DU LERNST HIER...

wie du eine Funktion definierst, der du Parameterwerte übergeben kannst. Du kennst dies bereits von vielen Turtlebefehlen. Wenn du beispielsweise *forward(100)* schreibst, erhält die Funktion *forward()* den Wert 100 und bewegt anschliessend die Turtle gemäss dieser Schrittweite vorwärts.

MUSTERBEISPIELE

Im Kapitel 5 hast du eine Funktion *square()* definiert, die ein Quadrat mit fixer Seitenlänge 100 zeichnet. Man sagt anschaulich, dass die Seitenlänge 100 im Programm "*fest verdrahtet*" sei.

Die Funktion kann viel flexibler eingesetzt werden, wenn du die Seitenlänge beim Funktionsaufruf angeben kannst, also z.B. *square(50)* oder *square(70)* schreiben kannst. Dazu musst du die Funktionsdefinition von square(s) mit einem Parameter versehen, dessen Name du in die Parameterklammer schreibst. Den Parameter kannst du im Innern der Funktion (im Funktionskörper) wie eine gewöhnliche Variable verwenden. Im Programm zeichnet die Turtle zwei Quadrate mit den Seitenlängen 80 und 50.

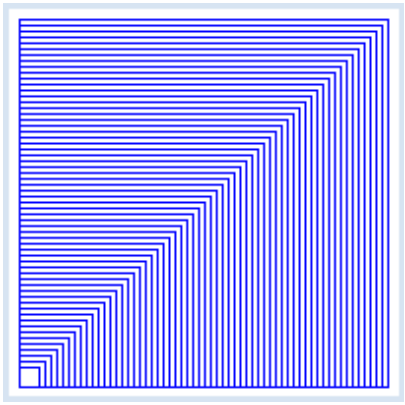


```
from gturtle import *

def square(s):
    repeat(4):
        forward(s)
        left(90)

makeTurtle()
setPenColor("red")
square(80)
left(180)
setPenColor("green")
square(50)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Erst richtig einleuchtend ist die Verwendung des Parameters *s*, wenn du, wie im folgenden Programm, sehr viele, sagen wir 100 Quadrate mit verschiedenen Seitenlängen zeichnen willst.

Damit die Turtle schnell arbeitet, versteckst du sie. Zudem setzt du die Turtle am Anfang mit setPos(-200, -200) nach links unten, damit das Bild schön zentriert ist.



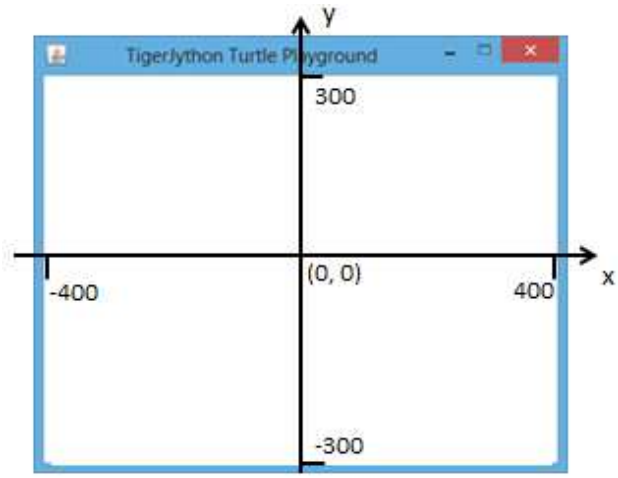
```
from gturtle import *

def square(s):
    repeat 4:
        forward(s)
        right(90)

makeTurtle()
hideTurtle()
setPos(-200, -200)
side = 400
repeat 100:
    square(side)
    side -= 4
```

Programmcode markieren (Ctrl+C kopieren)

MERKE DIR…

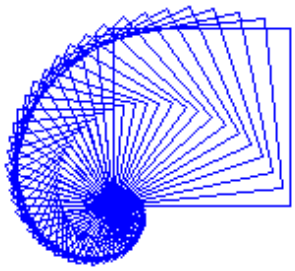
Wie du im Programm siehst, rufst du die Funktion *square()* mit einer Variablen *side* auf. Dadurch erhält die Funktion für den Parameter *s* den Wert von *side*.

Die x-y-Achsen des Koordinatensystems sind wie in der Mathematik üblich positioniert mit dem Urprung in der Mitte, der positiven x-Achse nach rechts und der positiven y-Achse nach oben.



ZUM SELBST LÖSEN

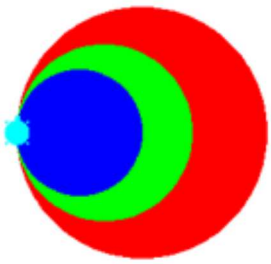
1. Das nebenstehende Bild entsteht, indem die Länge der Quadratseite ausgehend von 180 bei jedem nachfolgenden Quadrat um den Faktor 0.9 verkleinert wird. Es werden 100 Quadrate gezeichnet. Schreibe das Programm mit der Funktion *square(s)*



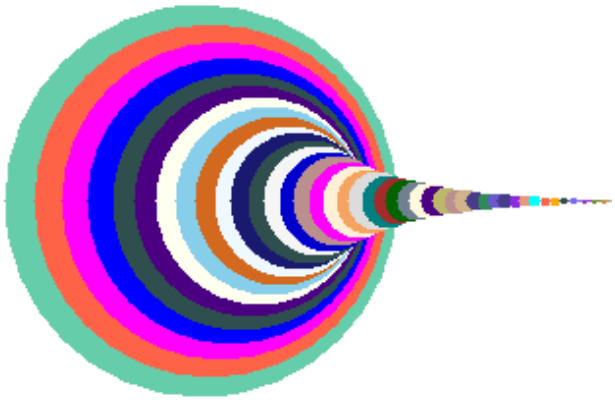
2. Die Funktion *star(s)* zeichnet einen gefüllten 5-zackigen Stern mit der Strichlänge *s*. Schreibe das Programm so, dass etwa nebenstehendes Bild erscheint.

3. Auch Farbnamen können als Parameter verwendet werden. Definiere eine Funktion, *dreieck(color)*, welche ein gefülltes Dreieck mit der gegebenen Farbe zeichnet. Erstelle damit das nebenstehende Bild.

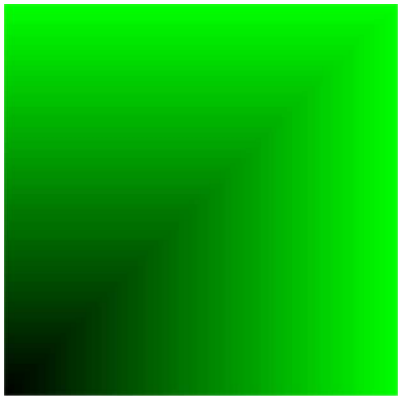
4. Du kannst einer Funktion auch beliebig viele Parameter übergeben, die du mit einem Komma trennst. Definiere unter Verwendung des Befehls *rightCircle(radius)* den neuen Befehl *fillRightCircle(radius, color)*, mit dem die Turtle einen gefüllten Kreis mit gegebenen Radius und gegebener Füllfarbe zeichnet und erstelle damit die nebenstehende Zeichnung. Wie du siehst, kann man mit eigenen Funktionen die Befehlsliste der Turtle ergänzen.



5. Erweitere die Befehlsliste der Turtle mit dem Befehl *colorDot(d)*, welcher ausgehend von den Befehlen *dot(d)* und *getRandomX11Color()* einen gefüllten Kreis mit dem Durchmesser d zeichnet, der mit einer zufälligen Farbe gefüllt ist. Erstelle damit einige lustige Bilder, beispielsweise in der Art des nebenstehenden.



6*. Ergänze den Programmcode aus dem Musterbeispiel 2 mit Farben. Du zeichnest 100 gefüllte Quadrate, wobei das grösste Quadrat eine hellgrüne Farbe und jedes weitere Quadrat eine etwas dunklere Farbe hat. Die Farben erhältst du mit dem Befehl *makeColor(r, g, b)* wobei r, g, b die roten, grünen und blauen Farbkomponenten sind. Diese können entweder als eine ganze Zahl zwischen 0 und 255 oder als eine Dezimalzahl zwischen 0 und 1 angegeben werden. Hier wählst du die rote und blaue Komponente 0 und änderst die grüne in der Wiederholschleife.



8. ZUFALLSZAHLEN

DU LERNST HIER...

wie du Zufallszahlen erzeugen kannst. Da der Zufall im täglichen Leben eine ausserordentlich grosse Rolle spielt, kannst du mit Zufallszahlen solche Abläufe auf dem Computer nachbilden (simulieren). Wie bei einem Würfelwurf sind Zufallszahlen meist gleichverteilt, d.h. treten mit geicher Häufigkeit (Wahrscheinlichkeit) auf.

MUSTERBEISPIEL

Du zeichnest 50 Sterne und legst ihre Position im Turtlefenster mit Zufallszahlen fest. Die Funktion ***randint(a, b)*** liefert bei jedem Aufruf eine ganzzahlige Zufallszahl im Bereich a, b. Beispielsweise erhältst du zufällige Würfelzahlen mit *randint(1, 6)*.

Falls du mit einem rechteckigen Turtlefenster mit den Koordinaten 800x600 arbeitest, kannst du die x-Koordinaten der Sterne im Bereich -370, 370 und die y-Koordinaten im Bereich -270, 270 zufällig wählen.

Die Funktion *star()* zeichnet einen einzelnen Stern.

```
from gturtle import *
from random import randint

def star():
    startPath()
    repeat 5:
        forward(50)
        left(144)
    fillPath()

makeTurtle()
hideTurtle()

repeat(50):
    x = randint(-270, 270)
    y = randint(-270, 270)
    setPos(x, y)
    star()
```

Es ist schön, auchdie Farben zufällig zu wählen. Die Funktion *makeColor(r, g, b)* erzeugt eine Farbe wobei r die rote, g die grüne und b die blaue Farbkomponente sind, die im Bereich 0 bis 255 liegen müssen.

makeColor(randint(0, 255), randint(0, 255), randint(0, 255)) erzeugt also eine zufällige Farbe.

```
from gturtle import *
from random import randint

def star():
    startPath()
    repeat 5:
        forward(50)
        left(144)
    fillPath()

makeTurtle()
hideTurtle()
clear("darkblue")

repeat(50):
    c = makeColor(randint(0, 255), randint(0, 255), randint(0, 255))
    setPenColor(c)
    setFillColor(c)
    x = randint (-270, 270)
    y = randint (-270, 270)
    setPos(x, y)
    star()
```

Programmcode markieren (Ctrl+C kopieren)

MERKE DIR...

from random import randint importiert die Funktion *randint()* aus dem Modul random. *randint(a, b)* erzeugt eine ganzzahlige Zufallszahl zwischen a und b, wobei a die kleinste und b die grösste Zahl ist.

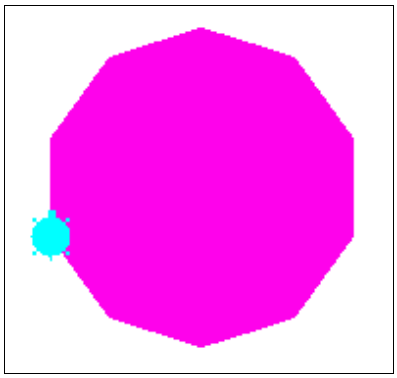
ZUM SELBST LÖSEN

1. Zeichne 100 gefüllte Kreise (dots). Die Position der Kreise, ihr Durchmesser und ihre Farbe sind zufällig.

2. Random walk
   Die Turtle startet an der Homeposition (0, 0) und bewegt sich wiederholt (z. B. 100 mal) um 40 Schritte in einer zufällig gewählten Richtung vorwärts.

   Um die Bewegungsrichtung festzulegen, kannst du Funktion
   heading(angle)
   verwenden, wobei *angle* eine Zufallszahl zwischen 0 und 360 ist.

3. Zufällige n-Ecke
   In deinem Programm wird bei jedem Lauf eine zufällige ganze Zahl n zwischen 3 und 12 erzeugt. Die Turtle zeichnet danach ein entprechendes regelmässiges n-Eck und füllt es mit einer zufälligen Farbe.



9. IF-ELSE

DU LERNST HIER...

wie man vorgehen muss, damit Programmblöcke nur unter gewissen Bedingungen ausgeführt werden. Du lernst auch, wie man Bedingungen *negiert* und wie man sie mit *Und*-, sowie *Oder*-Operatoren kombiniert.

MUSTERBEISPIEL

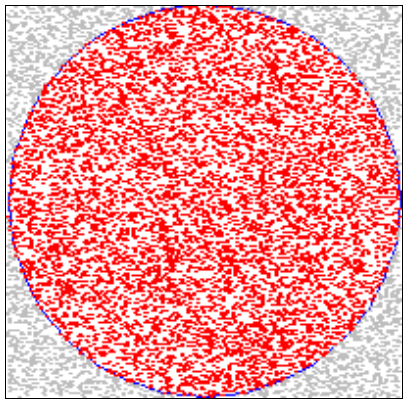
In deinem Programm lässt du die Turtle 10000 mal an einen zufälligen Ort innerhalb des Quadrats mit den Seitenlängen 400 springen und zeichnest dort unter der Bedingung, dass sich die Turtle innerhalb des Kreises mit dem Radius 200 befindet, einen roten Punkt. Sonst zeichnest du einen etwas kleineren grauen Punkt.

Befindet sich die Turtle innerhalb des Kreises, so gilt nach dem Satz von Pythagoras, dass ihr Abstandsquadrat *rsquare* zum Ursprung kleiner als 200 * 200 = 40000 ist.

Umgangssprachlich würdest du sagen:

"Falls das Radiusquadrat kleiner als 40000 ist, zeichne einen roten Punkt, sonst zeichne einen grauen Punkt."

Dies drückst du mit dem Keyword if bedingung: aus und rückst den nachfolgenden Programmblock ein. Dabei darfst du den Doppelpunkt nicht vergessen! Falls die Bedingung nicht erfüllt ist, wird der Programmblock ausgeführt, der unter der Zeile mit else: (wieder mit Doppelpunkt) steht.



```
from gturtle import *

makeTurtle()
hideTurtle()
openDot(400)

repeat 10000:
    setRandomPos(400, 400)
    rsquare = getX() * getX() + getY() * getY()
    if rsquare < 40000:
        setPenColor("red")
        dot(4)
    else:
        setPenColor("gray")
        dot(3)
```

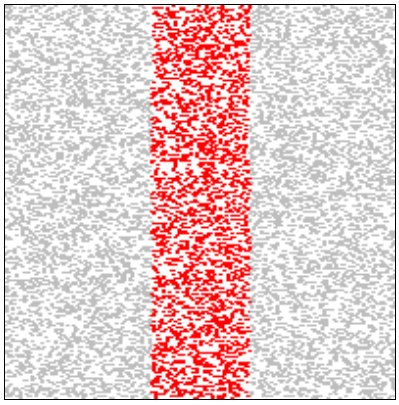
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MERKE DIR...

Statt zu sagen, dass eine Bedingung erfüllt oder nicht erfüllt ist, sagt man auch, die Bedingung sei **wahr** oder **falsch.** Sollen die grauen Punkte nicht gezeichnet werden, kannst du den else-Teil auch weglassen. Versuche es!

BEDINGUNGEN MIT *UND* VERKNÜPFEN

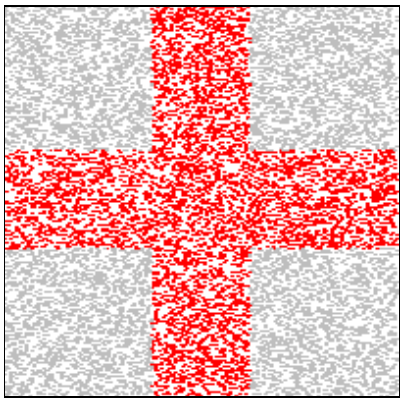
Wie auch in der Umgangssprache üblich, kannst du zwei Bedingungen mit und (and) verknüpfen. Die so kombinierte Bedingung ist nur dann wahr, wenn beide Teilbedingungen wahr sind. Hier zeichnest du dann rote Punkte, wenn die x-Koordinate der Turtle *grösser als -50* ***und*** *kleiner als 50* ist. Es entsteht offenbar ein vertikaler roter Streifen.



```
from gturtle import *

makeTurtle()
hideTurtle()

repeat 10000:
    setRandomPos(400, 400)
    x = getX()
    y = getY()
    if (x > -50 and x < 50):
       setPenColor("red")
       dot(4)
    else:
       setPenColor("gray")
       dot(3)
```

BEDINGUNGEN MIT *ODER* VERKNÜPFEN

Ein horizontaler roter Streifen genügt der Bedingung *y > -50* and *y < 50*. Punkte, die entweder im vertikalen oder horizontalen oder in beiden Gebieten liegen, erfüllen beide Bedingungen. Umgangssprachlich sagst du, dass sie im *vertikalen oder (auch) horizontalen Streifen* liegen. Du verknüpfst die beiden Bedingungen mit oder (or).

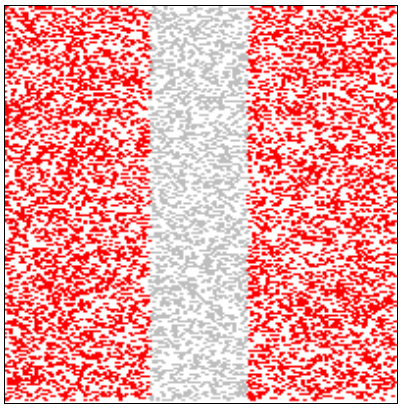


```
from gturtle import *

makeTurtle()
hideTurtle()

repeat 10000:
    setRandomPos(400, 400)
    x = getX()
    y = getY()
    if (x > -50 and x < 50) or (y > -50 and y < 50):
       setPenColor("red")
```

```
        dot(4)
    else:
        setPenColor("gray")
        dot(3)
```

EINE BEDINGUNG NEGIEREN

Mit dem Keyword *not* kannst du eine Bedingung negieren, das heisst, ihren Wahrheitswert umkehren. Punkte, die nicht im vertikalen Streifen liegen, genügen der Bedingung not (x > -50 and x < 50), wie das folgende Programm zeigt:



```
from gturtle import *

makeTurtle()
hideTurtle()

repeat 10000:
    setRandomPos(400, 400)
    x = getX()
    y = getY()
    if not (x > -50 and x < 50):
        setPenColor("red")
        dot(4)
    else:
        setPenColor("gray")
        dot(3)
```

MERKE DIR...

Die korrekte Klammerung bei Bedingungen ist sehr wichtig. Da *not* stärker bindet als *and* und *or*, musst du hier eine Klammer setzen. Am schwächsten bindet *or*. Den grauen Streifen erhältst du logischerweise auch, wenn du verlangst, dass *x <= -50* oder *x >= 50* ist. Man kann also Bedingungen verschieden formulieren. Versuche es!Für Zahlen gibt es folgende Vergleichsoperatoren:

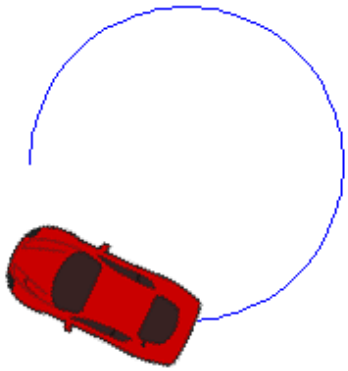
<	kleiner
<=	kleiner oder gleich
==	gleich
>=	grösser oder gleich
>	grösser
!=	verschieden

Du musst dich insbesondere an die **Verdoppelung des Gleichheitszeichens** bei der Gleichheitsbedingung gewöhnen. Diese ist nötig, damit der Computer zwischen der Zuweisung und der Gleichheitsbedingung unterscheiden kann.

MEHRFACH-AUSWAHL

Um mehr als zwei Fälle zu unterscheiden, musst du im *else*-Teil wieder eine *if*-Bedingung einfügen. In Python kann man kürzer elif verwenden und schreibt:

```
if bed1:
    ...
elif bed2:
    ...
else:
    ...
```

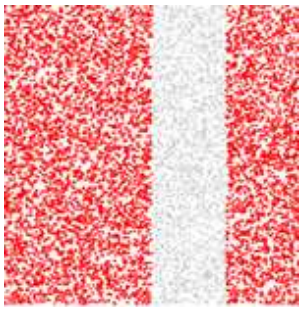


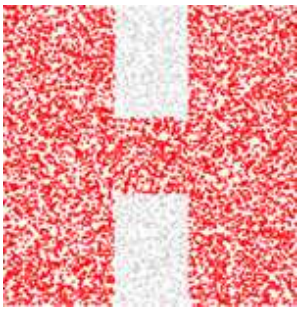
Du siehst dies an folgendem Musterbeispiel, bei dem der Benutzer eines von 3 Spritebildern auswählen kann. Zudem werden illegale Eingaben mit einer Fehlermeldung "abgefangen".

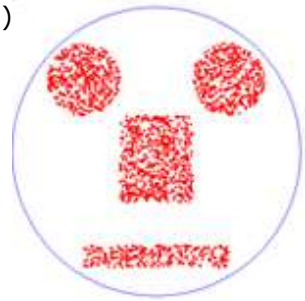
```
from gturtle import *

n = inputInt("Welches Auto (1, 2, 3)")
if n == 1:
    makeTurtle("sprites/car0.png")
elif n == 2:
    makeTurtle("sprites/car1.png")
elif n == 3:
    makeTurtle("sprites/car2.png")
else:
    msgDlg("Illegale Eingabe")

if n > 0 and n < 4:
    rightCircle(100)
```

ZUM SELBST LÖSEN

1. Erstelle ungefähr folgende Zeichnungen:

a) 

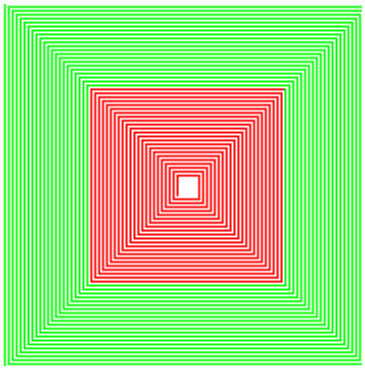
b) 

c*) 

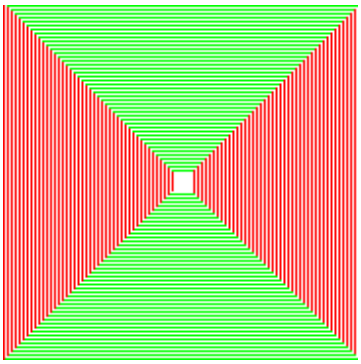
2. Du hast gelernt, dass man mit folgendem Programm eine Spirale zeichnen kann.

```
from gturtle import *
makeTurtle()
hideTurtle()
s = 10
repeat 200:
    forward(s)
    right(90)
    s = s + 1
```

a) Zeichne mit dieser Vorlage die nebenstehende Figur.



b) Die Modulo-Operation *a%b* liefert den Rest, wenn du *a* ganzzahlig durch *b* dividierst, beispielsweise liefert *8%3* den Rest 2. Damit kannst du herausfinden, ob die natürliche Zahl *s* gerade oder ungerade ist: *s%2* liefert nämlich 0, falls *s* gerade ist. Zeichne die nebenstehende Figur.



3. Oft möchtest du dem Benutzer die Möglichkeit geben, eine Farbe selbst auszuwählen. Elegant geht dies mit dem Farbwahl-Dialog *askColor(title, farbe)*, wo *title* der Text in der Titelzeile und *farbe* eine Farbvorwahl ist. Klickt der Benutzer den *OK*-Button, so wird die gewählte Farbe zurückgegeben. Drückt er den Abbrechen- oder den *Close*-Button der Titelzeile, so hat der Rückgabewert den speziellen Wert *None*.

   Mit dem folgenden Programm-Gerüst siehst du, wie man grundsätzlich ein Programm mit einem solchen Eingabedialog konzipiert. Wie du siehst, wird in einer endlosen *repeat*-Schleife der Rückgabewert auf *None* getestet und die Schleife mit dem speziellen Keyword *break* verlassen. Mit *dispose()* wird zudem das Turtlefenster geschlossen.

```
from gturtle import *

makeTurtle()

repeat:
    color = askColor("Farbauswahl", "yellow")
    if color == None:
        break
dispose()
```

Ersetze die Kommentarzeile mit dem # durch eigenen Code, der beispielsweise einen farbig gefüllten Stern zeichnet.

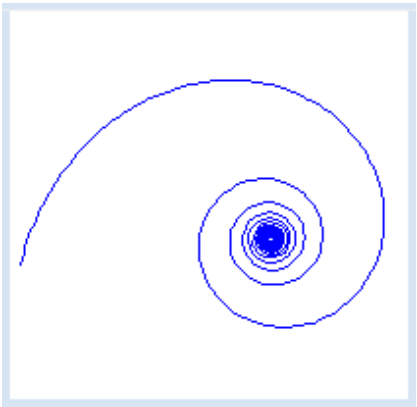
10. WHILE & FOR

DU LERNST HIER...

wie du die Wiederholstrukturen mit den Keywords *while* und *for* verwendest. Es handelt sich um die wichtigsten Programmstrukturen überhaupt.

MUSTERBEISPIEL ZUR WHILE-SCHLEIFE

Die weitaus am häufigsten eingesetzte Wiederholstruktur kann umgangssprachlich so formuliert werden: "*Solange die folgende Bedingung erfüllt ist, führe den nachfolgenden Programmblock aus...*" Die Wiederholschleife beginnt mit dem Keyword while und einer Bedingung, **gefolgt von einem Doppelpunkt**. Der nachfolgende Programmblock muss eingerückt werden.



```
from gturtle import *

makeTurtle()
hideTurtle()
setPos(-300, -200)

a = 1
while a < 100:
    forward(8)
    right(a)
    a = 1.01 * a
```

Programmcode markieren (Ctrl+C kopieren)

MERKE DIR...

Die Bedingung a < 100 nennt man auch *Laufbedingung*, da sie wahr sein muss, damit der Block durchlaufen wird. Die übliche Art, die Schleife zu verlassen, besteht darin, im Schleifenkörper dafür zu sorgen, dass die Laufbedingung falsch wird. Hier erhöhst du *a* bei jedem Durchgang um den Faktor 1.01 und brichst ab, wenn *a* nicht mehr kleiner als 100 ist.

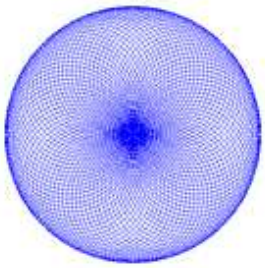
ZUM SELBST LÖSEN

1. Bisher hast du für eine bestimmte Zahl von Wiederholungen die repeat-Schleife verwendet. Neu kannst du auch eine *while*-Schleife mit einem Schleifenzähler *i* verwenden.

```
from gturtle import *

makeTurtle()

i = 0
while i < 4:
```

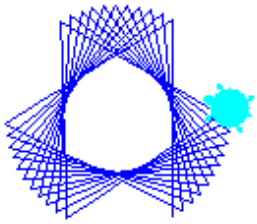
```
forward(100)
right(90)
i = i + 1
```

Die nebenstehende Figur entsteht, indem du mit einer *while*-Schleife 90 mal mit *rightCircle(100)* einen Kreis zeichnest und die Turtle um 4 Grad nach links drehst. Schreibe das Programm und spiele etwas mit dem Drehwinkel.



2. Die Turtle soll mit normaler Geschwindigkeit wiederhollt 100 Schritte vorwärts gehen und 122 Grad links drehen. Nach 30 Sekunden soll sie sich zur Ruhe setzen.

   Anleitung: Mit *time.time()* liefert dir das System die aktuelle Uhrzeit (in Sekunden seit 1.1.1970). Dazu musst du mit *import time* das entsprechende Modul importieren. Willst du ein Zeitintervall bestimmen, so musst du die Anfangszeit in einer Variablen *startTime* speichern und die Differenz *time.time() - startTime* bilden.



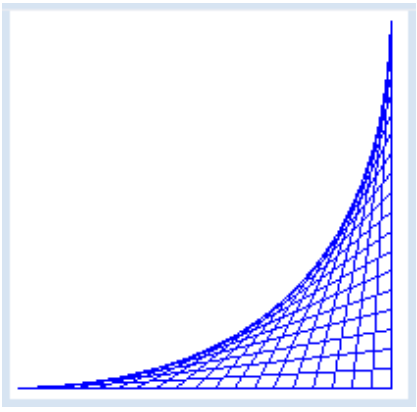
MUSTERBEISPIEL ZUR FOR-SCHLEIFE

Oft bist in der Situation, dass du eine Wiederholschleife eine bestimmte Anzahl mal durchlaufen musst und dabei eine ganzzahlige Variable brauchst, die bei jedem Durchgang um eins grösser wird. Du kannst dies zwar mit einer *repeat*- oder *while*-Schleife machen, es gibt aber noch eine elegantere Möglichkeit mit einer *for*-Schleife, wo der Schleifenzähler automatisch verändert wird.

Willst du die *n* Zahlen *i* von 0 bis *n-1* durchlaufen, so schreibst du dazu:

```
for i in range(n):
    (Schleifenkörper)
```

Mit der Funktion line(x1, y1, x2, y2) zeichnet die Turtle eine Linie vom Punkt (x1, y1) zum Punkt (x2, y2). Nachfolgend zeichnest du damit eine hübsche Linienschar mit 20 Linien.



```
from gturtle import *

def line(x1, y1, x2, y2):
    setPos(x1, y1)
    moveTo(x2, y2)

makeTurtle()
hideTurtle()

for i in range(21):
    line(10 * i, 0, 200, 10 * i)
```

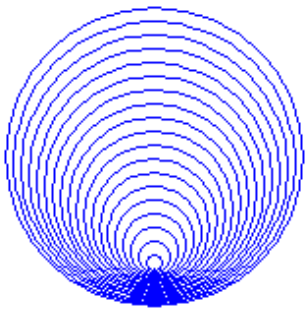
Programmcode markieren (Ctrl+C kopieren)

MERKE DIR...

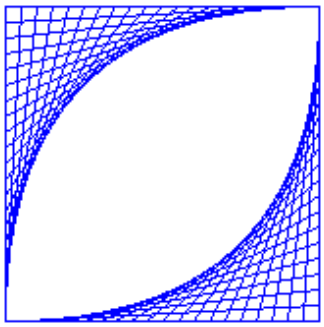
Beachte, dass mit for i in range(21): alle 21 Zahlen 0, 1, 2,..20 durchlaufen werden. Der Parameter *n* in *range(n)* gibt also nicht etwa den Endwert an. Vergiss auch den Doppelpunkt nicht.

ZUM SELBST LÖSEN

3. Die nebenstehende Figur erzeugst du, indem du 20 Kreise mit *openDot(d)* zeichnest und die Turtle bei jedem Schleifendurchgang um 5 Schritte vorwärts laufen lässt. Verwende dazu eine for-Schleife.



4. Zeichne die nebenstehende Figur.



ZUSATZSTOFF: INEINANDER GESCHACHTELTE FOR-SCHLEIFEN

Schleifen können auch ineinander geschachtelt sein. Willst du beispielsweise ein Gitter (eine Tabelle oder Matrix) mit 8 horizontalen und 8 vertikalen Zellen durchlaufen, so kannst du dies mit einem Zeilenindex *i* und einem Spaltenindex *k* tun, die beide von 0 bis 7 laufen.

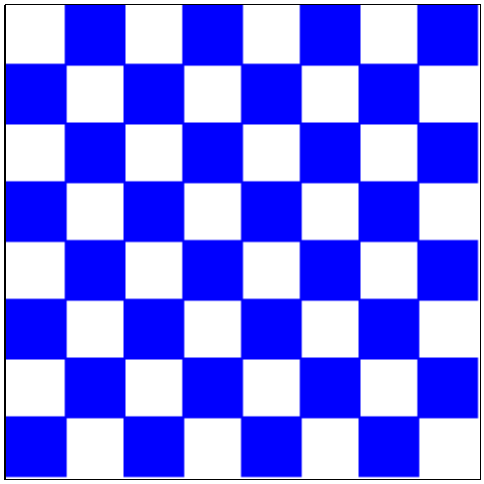
	k = 0	1	2	3	4	5	6	7
i = 0	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2	...							
3								
4								
5								
6								
7	7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

Mit

```
for i in range(8):
    for k in range(8):
        print i, k
```

durchläufst du bei festem *i* in der inneren Schleife mit *k* die Zeile und nimmst dir dann die nächste Zeile *i* vor.

Als lustige Übung zeichnest du mit diesem Verfahren ein Schachbrett. Du siehst leicht ein, dass du immer dann ein gefülltes Quadrat zeichnen musst, wenn die Summe des Zeilen- und Spaltenindex eine gerade Zahl ist.



```
from gturtle import *

def field(x, y):
    setPos(x, y)
    startPath()
    repeat 4:
        forward(30)
        left(90)
    fillPath()

makeTurtle()
hideTurtle()
for i in range(8):
    for k in range(8):
        if (i + k) % 2 == 0:
            field(30 * i, 30 * k)
```

Programmcode markieren (Ctrl+C kopieren Ctrl+V einfügen)

11. EREIGNISSE

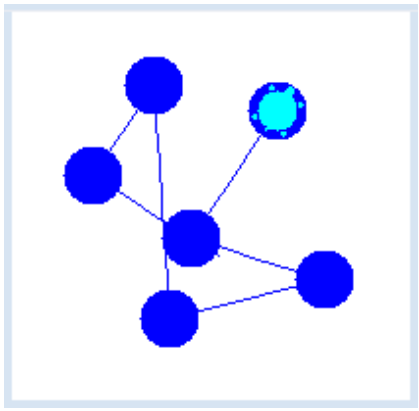
DU LERNST HIER...

dass die ereignisgesteuerte Programmierung ein neues Programmierkonzept ist, das aber den alltäglichen Erfahrungen sehr nahe kommt. Statt dass du mit einem Programm der Turtle in zeitlicher Abfolge Befehle erteilst, könntest du die Turtle so steuern, dass sie bei bestimmten **Ereignissen** ihren **Zustand ändert**.

MUSTERBEISPIEL ZU CALLBACKS

Nach dem Start des Programms verharrt die Turtle in der Homeposition, bis du ihr mit einem Mausklick befiehlst, an die Stelle des Mauscursors zu springen. Die Programmiertechnik, um auf einen Mausevent zu reagieren, ist die Folgende:

Du schreibst eine Funktion drawDot() (der Name ist frei wählbar). Dort legst du fest, was die Turtle beim Mausklick tun soll. Hier bewegt sie sich die Stelle des Mausklicks und zeichnet mit *dot(30)* einen gefüllten Kreis. Deine Funktion wird als benannte Parameter von makeTurtle() registriert. So teilst du dem System mit, dass es diese Funktion bei jedem Mausklick aufrufen soll. Dabei werden ihr die Koordinaten x, y des Mauscursors übergeben.



```
from gturtle import *

def drawDot(x,y):
    moveTo(x, y)
    dot(30)

makeTurtle(mouseHit = drawDot)
```

Programmcode markieren (Ctrl+C kopieren)

MERKE DIR...

Die neue Programmiertechnik ist daran zu erkennen, dass die Funktion drawDot(x, y) nirgends von deinem Programm aufgerufen wird. Es ist vielmehr das System, dass sie beim Auftreten eines Ereignisses aufruft. Eine solche Funktion nennen wir **Callbackfunktion** oder kurz **Callback**.

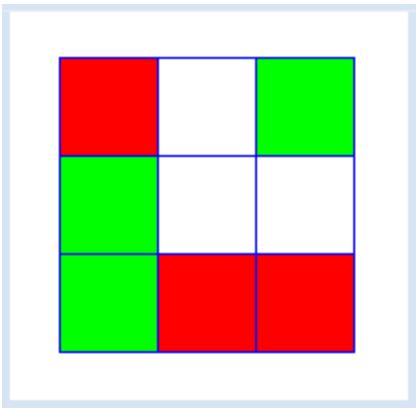
DEIN ERSTES GAME

Tic-Tac-Toe ist ein bekanntes Spiel, bei dem zwei Spieler abwechslungsweise Kreuze oder Kringel in die Felder eines 3x3-Gitters setzen mit dem Ziel, als erster drei eigene Zeichen in einer horizontalen, vertikalen Linie oder einer der Diagonalen zu haben.

Der Einfachheit halber verwendest du im Programm keine Zeichen, sondern färbst bei einem

Mausklick die Felder je nach Spieler rot oder grün. Dabei verwendest du den Füllbefehl fill(), mit dem die Turtle das geschlossene Gebiet ausfüllt, in dem sie sich gerade befindet.

Damit die Füllfarbe nach jedem Klick umschaltet, führst du eine Variable player ein, die zwischen dem Wert 1 und 2 umschaltet. *player* ist im Hauptprogramm definiert und ihr Wert wird zuerst auf 1 gesetzt (man sagt auch, *"auf 1 initialisiert"*). Verwendet wird die Variable aber in der Callbackfunktion fillSquare(x, y). Damit du die Variable in der Funktion verändern kannst, musst du sie als global bezeichnen.



```
from gturtle import *

def square():
    repeat 4:
        forward(50)
        right(90)

def drawGrid():
    for x in range(3):
        for y in range(3):
            setPos(50 * x, 50 * y)
            square()

def fillSquare(x, y):
    global player
    if player == 1:
        setFillColor("red")
        player = 2
    elif player == 2:
        setFillColor("green")
        player = 1
    setPos(x, y)
    fill()

player = 1
makeTurtle(mouseHit = fillSquare)
hideTurtle()
drawGrid()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

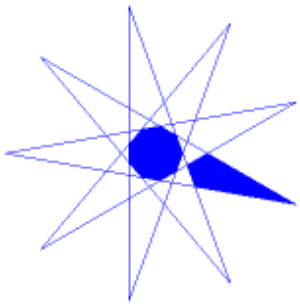
MERKE DIR...

Da *player* sowohl im Hauptprogramm wie in der Funktion fillSquare(x, y) verwendet wird, nennt man sie eine **globale Variable**. Eine Variable die hingegen nur in einer Funktion verwendet wird, nennt man eine **lokale Variable**.

Du wirst als Eigentätigkeit das Programm noch etwas verbessern.

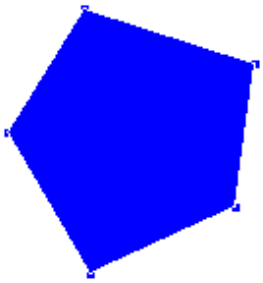
ZUM SELBST LÖSEN

1. Zeichne mit einer Wiederholstruktur den nebenstehenden Stern und fülle ihn mit Mausklicks nach deinem Geschmack aus.



2. Bei jedem Mausklick soll ein gefüllter Stern entstehen. Du musst die Turtle verstecken, damit der Stern fertig gezeichnet wird, bevor du das nächste Mal klickst.

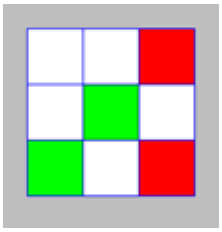
3. Schreibe ein Programm, um mit 5 Mausklicks ein gefülltes 5-Eck zu zeichnen. Bei jedem Klick erscheint ein kleiner Markierungspunkt und beim fünften Klick das gefüllte Polygon. Du benötigst dazu eine Zählvariable *n*, die du im Callback erhöhst. Das Füllen machst du mit *startPath()/fillPath()*.



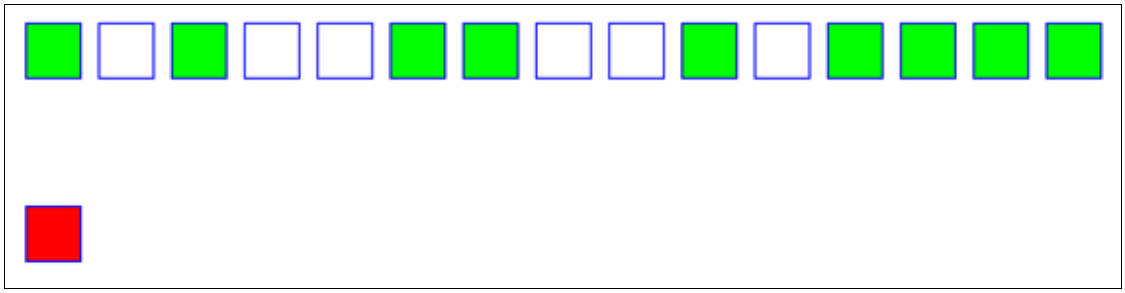
4. Verbessere das Tic-Tac-Toe-Spiel:

   a) Färbe den Hintergrund grau und die Felder weiss

   b) Bestimme bei jedem Mausklick mit *getPixelColorStr()* die Farbe, auf die du gerade klickst. Führe das Füllen nur dann durch, falls du auf einen weissen Hintergrund klickst.



5*. Erstelle ein **Nim-Spiel** mit 15 grünen Steinen. Jeder Spieler kann mit einem Mausklick maximal 3 Steine entfernen und muss dann auf eine rote Schaltfläche, sozusagen den *OK*-Button, klicken. Dann kommt der andere Spieler zum Zug. Wer den letzten Stein entfernt, hat verloren. Wie in der Aufgabe 4 verwendest du die Hintergrundfarbe mit *getPixelColorStr()* . (Das Spiel wird üblicherweise mit Zündhölzern gespielt.)



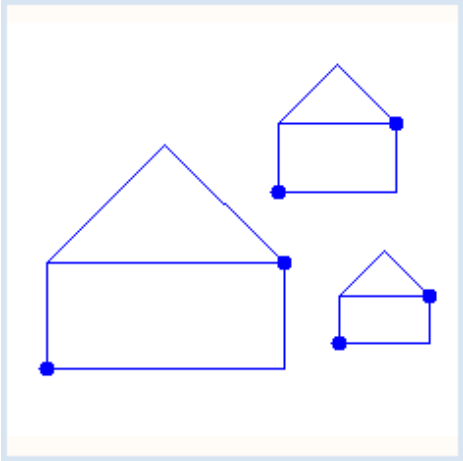
12. FUNKTIONEN MIT RÜCKGABEWERTEN

DU LERNST HIER...

wie eine Funktion mit dem Schlüsselwort *return* Informationen zurückgeben kann. Auch einige Turtlebefehle sind Funktionen, die Werte zurückgeben, wie beispielsweise die Funktion *getX(),* welche die aktuelle x-Koordinate der Turtle zurück liefert.

MUSTERBEISPIEL

Du willst mit der Maus Häuser zeichnen und zwar so, dass du die Maus mit gedrückter Maustaste von der linken unteren Ecke bis zur rechten oberen Ecke des Hauses bewegst. Sobald du die Maustaste loslässt, wird das Haus gezeichnet.



Die Funktion getDimension() bestimmt aus den Mauskoordinaten die Breite, Höhe und die Dachgrösse und gibt diese Werte mit *return* zurück. Mit dem Aufruf von b, h, r = getDimension() in der Funktion *drawHouse()* stehen diese Werte für das Zeichnen des Hauses zur Verfügung.

```
from gturtle import *
from math import sqrt

def onMousePressed(x,y):
    global x0, y0
    x0 = x
    y0 = y
    setPos(x0, y0)
    dot(8)

def onMouseReleased(x,y):
    global x1, y1
    x1 = x
    y1 = y
    setPos(x1, y1)
    dot(8)
    drawHouse()

def getDimension():
    base = x1 - x0
    height = y1 - y0
    roof = base / sqrt(2)
    return base, height, roof

def drawHouse():
    b, h, r = getDimension()
    setPos(x0, y0)
    repeat 2:
        forward(h)
        right(90)
        forward(b)
        right(90)
```

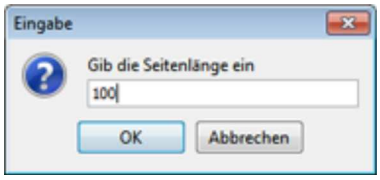
```
    forward(h)
    right(45)
    forward(r)
    right(90)
    forward(r)
    n = 0
    heading(0)

makeTurtle(mousePressed = onMousePressed, mouseReleased = onMouseReleased)
hideTurtle()
n = 0
```

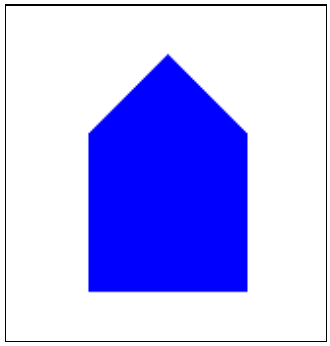
MERKE DIR...

Eine Funktion kann einen oder mehrere Werte zurückgeben. Dabei ist die Reihenfolge in der return Anweisung wichtig. Genau in dieser Reihenfolge holst du die Werte mit *b, h, r = getDimension()* zurück. Für die Berechnung der Dachlänge nach dem Satz von Pythagoras brauchst du die Wurzel aus 2. Dazu verwendest du die Funktion *sqrt()* aus dem Modul *math*. Damit die Werte x0, y0, x1, y1 in den Calbackfunktionen *onMousePressed()* und *onMouseReleased()* geändert werden können, müssen sie als global definiert sein.

ZUM SELBST LÖSEN

1. Dein Programm soll nach der Eingabe der Seitenlänge s ein Haus zeichnen, das aus einem Quadrat mit der Seite s und einem mit 45° geneigten Dach besteht. Für die Berechnung der Dachgrösse verwendest du eine Funktion mit Rückgabewert. Eine Dialogbox für die Eingabe der Seitenlänge erhältst du mit:
   s = inputInt("Gib die Seitenlänge ein")

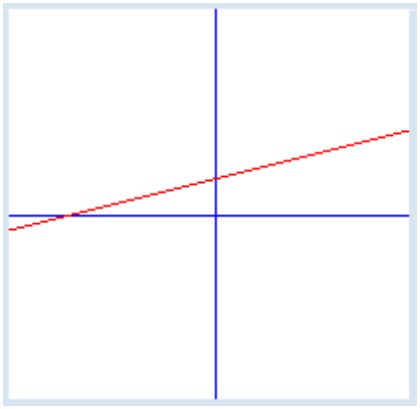




2. Du sollst die Werte der Funktion

$$y = \frac{1}{4}\ x + 50$$

   im Bereich *x = -250* bis *250* mit einer Schrittweite von 5 berechnen und den Graf der Funktion zeichnen. Verwende für die Berechnung der Werte eine Funktion mit Rückgabewert. Zeichne eine x- und eine y-Achse und verwende die Funktion *moveTo(x, y)*, um die Funktion zu zeichnen.



   Schreibe die Funktionswerte mit *print x, y* im Ausgabefenster aus. So kannst du sehr einfach den Schnittpunkt der Geraden mit der x-Achse bestimmen.

```
-205 -1.25
-200 0.0
-195 1.25
-190 2.5
-185 3.75
-180 5.0
-175 6.25
```

13. LISTEN & TUPELS

DU LERNST HIER...

wie du beliebig viele Daten in einer Liste oder einem Tupel ablegen kannst und wie du darauf zugreifst. Statt Werte in verschiedenen Variable *a, b, c* einzeln abzuspeichern, stellt Python Datentypen zur Verfügung, die als Behälter für beliebig viele Daten verwendet werden. Wichtigste zusammengesetzte (strukturierte) Datentypen sind die Liste und das Tupel. In Python werden die einzelnen Werte der Liste in eckige Klammern und die Werte des Tupels in runde Klammern geschrieben und mit Kommas getrennt. Beispielsweise schreibst du das Tupel mit den Zahlen 10, 20, also beispielsweise die x, y-Koordinaten eines Punkts so:

pt = (10, 20)

oder die Liste mit den Tönen eines Songs

song = ['c', 'd', 'e', 'f', 'g', 0, 'g', 0, 'a', 'a', 'a', 'a', 'g']

Der zusätzliche Leerschlag nach dem Komma ist fakultativ. Zur besseren Lesbarkeit setzen wir ihn aber immer.

Auf einen einzelnen Wert der Liste oder des Tupels greifst du mit einem Index in einer eckigen Klammer zu, der immer bei 0 beginnt, beispielsweise liefert *song[2]* den Ton 'e' oder *x = pt[0]*, die x-Koordinate 10 des Punkts. Der Unterschied zwischen Listen und Tupels besteht darin, dass Listen verändert werden können, nachdem sie erzeugt wurden, Tupel hingegen nicht.

MUSTERBEISPIEL

Um den Song abzuspielen, verwendest du playTone(note, duration), wo *duration* die zeitliche Länge des Tons in Millisekunden ist (note = 0 legt eine Pause ein). Du musst also die Liste durchlaufen und die Werte aus der Liste holen. Am einfachsten machst du dies mit einer for-Schleife. Dazu gibt es zwei Möglichkeiten.

Du durchläufst die Liste mit einem Index *i.* Ist *n* die Anzahl der Listenelemente, (man sagt auch die **Länge der Liste)** so muss der Index i von 0 bis *n-1* laufen.. Statt von Hand zu zählen, kannst du die Länge n der Liste mit der eingebauten Funktion *n = len(li)* bestimmen. Dies hat den Vorteil, dass die Listenlänge immer noch stimmt, wenn du die Länge der Liste veränderst. Zum Abspielen schreibst du daher:

```
from gturtle import *

song = ['c', 'd', 'e', 'f', 'g', 0, 'g', 0, 'a', 'a', 'a', 'a', 'g']

n = len(song)
for i in range(n):
    playTone(song[i], 400)
```

Oft setzt man *len(song)* direkt ein und schreibt:

```
for i in range(len(song)):
    ...
```

Es gibt eine zweite, kürzere Art, die Liste mit einer for-Schleife zu durchlaufen, bei der du keinen Index brauchst. Mit der Schreibweise

```
for element in liste:
```

kannst du nämlich ein Element um das andere aus der Liste holen. Das Abspielprogramm schreibst du also eleganter so:

```
from gturtle import *

song = ['c', 'd', 'e', 'f', 'g', 0, 'g', 0, 'a', 'a', 'a', 'a', 'g']

for tone in song:
    playTone(tone, 400)
```

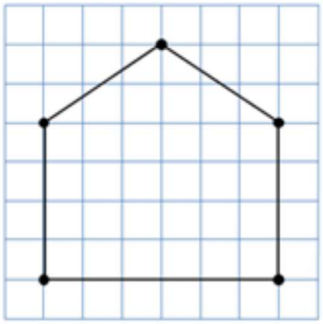
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

ZUM SELBST LÖSEN

1. Ergänze die Notenliste so, dass das ganze Lied gemäss folgender Notennotation gespielt wird. Kennst du es?

2a. Zeichne auf ein Häuschenpapier eine Figur, z.B. die Umrisse eines Hauses. Lese die Koordinaten der Eckpunkt *(x, y)* ab und schreibe sie in eine Liste, zum Beispiel
haus = [0, 0, 0, 200, 150, 300, 300, 200, 300, 0, 0, 0]
Durchlaufe die Liste mit irgendeiner Wiederholstruktur, um die Figur mit der Turtle zu zeichnen.

.



2b. Schreibe die Koordinaten als Punkttupel (x, y) und setze sie in eine Liste, also beispielsweise *haus = [(0, 0], (0, 200) , (150, 300), (300, 200), (300, 0), (0, 0)]. Z*eichne das Haus noch einmal. Beachte, dass du allen Turtlebefehlen statt x-, y-Koordinaten direkt ein Punkttupel oder eine Punktliste übergeben kannst.

DIE LISTE ALS OBJEKT

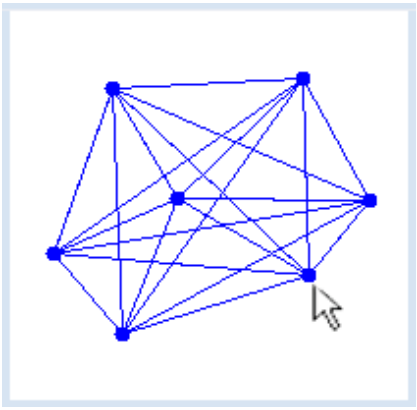
Bisher hast du die Listen im Programmcode definiert. Du kannst aber eine Liste auch erst zu Laufzeit des Programms erstellen, denn ihre Länge ist flexibel (man sagt auch "dynamisch"). Dabei ist es wichtig, dass du die Liste wie ein **Objekt** als etwas Eigenständiges betrachtest.

In deinem Programm erzeugst du mit einem Mausklick einen neuen Punkt pt = (x, y) und zeichnest jeweils alle Verbindungslinien zu den bestehenden Punkten. Dazu musst du die bestehenden Punkte in einer Liste *points* speichern.

Dem Objekt *points* kannst du **Befehle erteilen**, beispielsweise *"Füge das folgende Element hinten an.* Dazu brauchst du den **Punktoperator**. Befehle sind **Funktionen**, bei Objekten spricht man auch von **Methoden**.

Um den Punkt pt hinten in der Liste anzufügen, schreibst du points.append(pt). Zuerst musst du mit points = [] eine leere Liste erzeugen.

Um die Verbindungslinien zu zeichnen, schreibst du eine Funktion drawLines(pt) und rufst sie im Mauscallback auf. In der for-Schleife verbindest du den neuen Punkt mit allen anderen.



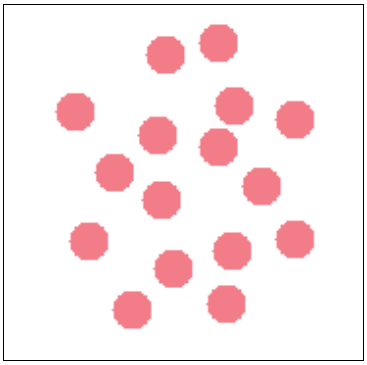
```
from gturtle import *

def drawLines(pt):
    for p in points:
        setPos(pt)
        moveTo(p)

def onMouseHit(x, y):
    pt = (x, y)
    setPos(pt)
    dot(8)
    drawLines(pt)
    points.append(pt)

makeTurtle(mouseHit = onMouseHit)
#hideTurtle()
points = []
```

MERKE DIR...

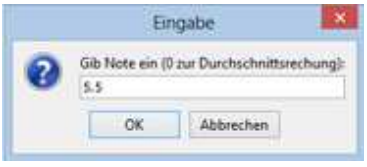
Mit points.append(pt) fügst du ein neues Element hinten an der Liste an.
for p in points: durchläufst du die Liste Element um Element.

ZUM SELBST LÖSEN

3. Als Vorlage nimmst du das vorhergehende Beispiel. Bei jedem Mausklick wird wie oben ein neuer Punkt erzeugt und dann werden alle Punkte mit der gleichen zufälligen Farbe übermalt.



4. In einer Schleife kann der Benutzer mit *inputFloat()* beliebig viele Zeugnisnoten zwischen 1 und 6 eingeben, die in eine Notenliste eingefügt werden. Bei der Eingabe 0 wird der Durchschnitt mit *msgDlg()* ausgeschrieben.



.

14. STRINGS (ZEICHENKETTEN)

DU LERNST HIER...

mit Strings umzugeben. Die Zeichen eines Texts (Buchstaben, Satzzeichen, Leerschläge, Zeilenumbrüche, usw.) werden im Computer als eine Zahl zwischen 0 und 127 im **ASCII-Code** dargestellt. Will man auch die Umlaute und Akzente berücksichtigen, so braucht es bereits die Zahlen 0..255, für eine Erweiterung auf Sprachen mit ganz anderen Zeichen (kyrillisch, chinesisch, usw.) braucht man den **Unicode** mit Zahlen 0..65535.

MUSTERBEISPIEL

Wörter und Sätze sind aneinander gefügte Zeichen und werden als String bezeichnet.

Zur Definition verwendet man ein einfaches oder doppeltes Anführungszeichenpaar (Gänsefüsschen), also beispielsweise *name = 'Maya'* oder *name = "Maya"*. Wir verwenden meist das doppelte Anführungszeichen.

Ein String lässt sich wie eine Liste auffassen, deren Elemente einzelne Zeichen sind. Wie bei Listen kannst du ein einzelnes Zeichen mit einem Index herausholen. Im Programm wird der String auf zwei verschiedene Arten mit einer for-Schleife "durchlaufen". Was findest du eleganter?


```
from gturtle import *

makeTurtle()
right(90)
name = "Maya Bircher"
for c in name:
    forward(20)
    label(c)
right(90)
for i in range(len(name)):
    forward(20)
    label(name[i])
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

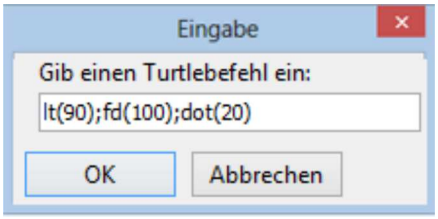
MERKE DIR...

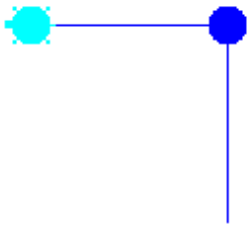
Wie du bereits weisst, kannst du zwei Strings mit dem Additionsoperator + zusammenfügen (konkatenieren). Dabei musst du aber darauf achten, dass links und rechts vom Pluszeichen auch wirklich Strings stehen. Willst du also eine Zahl *z* hinzufügen, so musst du diese zuerst mit *str(z)* in einen String umwandeln. Es gibt auch einen leeren String *leer = ""*. Unterscheide ihn gut von einem String mit einem Leerzeichen *luecke = " "*.

Im Gegensatz zu Listen kannst du ein einzelnes Zeichen in einem String nicht ändern, da Strings wie Tupels unveränderlich sind. Um beispielsweise in "*Maya*" den ersten Buchstaben zu ändern, musst du den ganzen String neu zuweisen: *name = "Kaya".*

PROGRAMMCODE INTERAKTIV AUSFÜHREN

Ein Programm kannst du auch als einen String auffassen, der vom Laufzeitsystem, dem Python-Interpreter, gelesen und als Programm ausgeführt wird. Darum ist es in Python möglich, Programmzeilen in einem Eingabedialog einzulesen und direkt auszuführen. Dazu verwendest du die Funktion *exec().*





```
from gturtle import *

makeTurtle()
repeat:
    s = inputString("Gib einen Turtlebefehl ein:")
    exec(s)
```

Programmcode markieren (Ctrl+C kopieren)

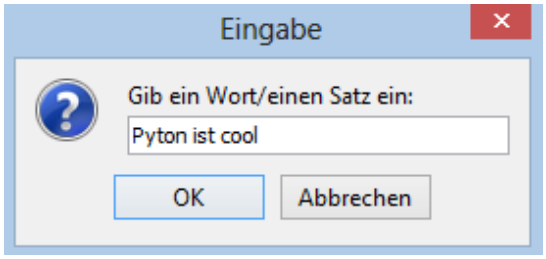
MERKE DIR...

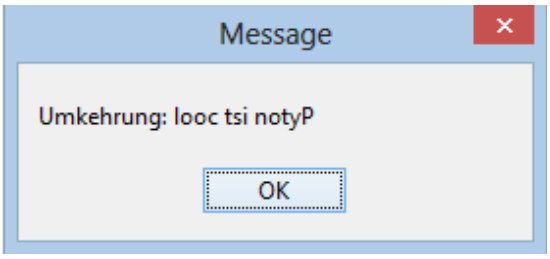
Du kannst auch mehrere Befehle mit Strichpunkten getrennt miteinander eingeben. Dein Programm benimmt sich aber leider sehr schlecht, wenn du eine fehlerhafte Eingabe machst, da *exec()* dann eine sogenannte Exception auslöst (man sagt auch: *"eine Exception wirft"*). Dies führt zum Abbruch des Programms, also zu einem **Crash**! Es ist aber leicht, dieses böse Fehlverhalten zu korrigieren, wenn du die Exception in einem **try-except-Block** "fängst".

```
from gturtle import *

makeTurtle()
repeat:
    s = inputString("Gib einen Turtlebefehl ein")
    try:
        exec(s)
    except:
        msgDlg("Illegale Eingabe")
```

ZUM SELBST LÖSEN

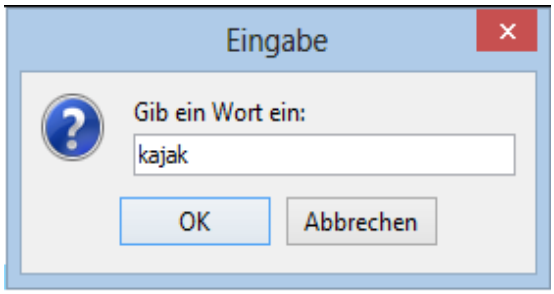
1a. Ein String, den du mit einem Eingabedialog *inputString("Gib ein Wort/einen Satz ein")* einliest, soll von von hinten gelesen ausgeschrieben werden.

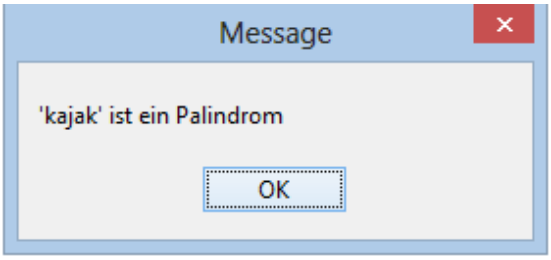




Das Programm soll solange laufen, bis man den *Abbrechen*-Button klickt.

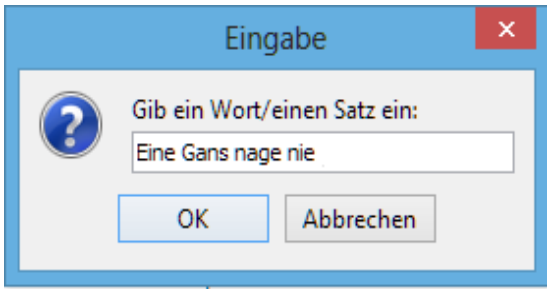
1b. Modifiziere das Programm, dass es herausfindet, ob das Eingabewort ein Palindrom ist, also von hinten gelesen gleich bleibt. Du kannst Strings mit == oder != miteinander vergleichen.

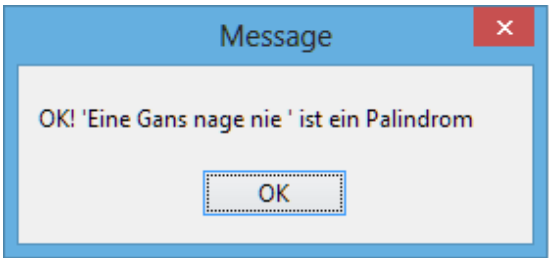




1c*.Bei einem Palindrom beachtet man in der Regel die Gross-/Kleinschreibung, sowie die Leerzeichen nicht. Verbessere das Programm sinngemäss.

Anleitung: Wie eine Liste, ist auch ein String ein Objekt. Mit der String-Funktion *lower()* wird der String zurückgegeben, der aus kleigeschriebenen Buchstaben besteht. Die String-Funktion *replace(a, b)* gibt einen String zurück, in dem alle Vorkommnisse von *a* durch *b* ersetzt sind. Der ursprüngliche String wird dabei nicht verändert!





2a. Verbessere das Programm zur interaktiven Ausführung von Turtlebefehlen so, dass bei Eingabe des Befehls "exit" in beliebiger Gross-/Kleinschreibung das Programm beendet wird und das Turtlefenster schliesst. (Zum Schliessen des Fensters wird *dispose()* aufgerufen.)

2b. Mache das Programm noch robuster, indem du mit der Stringfunktion *strip()* Leerzeichen vor und nach dem Eingabebefehl entfernst.

15. COMPUTERANIMATION

DU LERNST HIER...

wie du animierte Computergrafiken ähnlich wie in einem Animationsfilm erstellst. Die Bilder kannst du mit der Turtlezeichnen oder als Bilddateien laden.

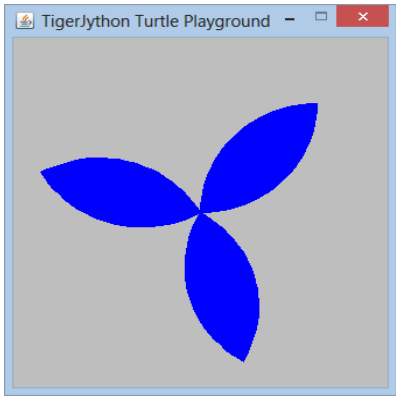
MUSTERBEISPIEL

Eine Animation besteht aus Einzelbildern, die sich nur wenig ändern und Schritt um Schritt zeitlich nacheinander dargestellt werden. Da das menschliche Auge nur rund 25 Bilder pro Sekunde erfassen kann, ergibt sich wie beim Film eine fliessende, ruckelfreie Bewegung, wenn die Bildfolge genügend schnell gezeigt wird.

Als Beispiel zeichnest du mit der Funktion *propeller(a)* einen 3-blättrigen Propeller. wobei *a* der Richtungswinkel zum ersten Propellerblatt ist. Für ein ruckelfreie Computeranimation ist es wichtig, dass das Zeichnen in einem **Bildspeicher** (Bildbuffer) erfolgt und das Bild erst nachher als Ganzes auf dem Bildschirm dargestellt ("gerendert") wird. Diese Art des Zeichnens nennt man auch **Doppelbufferung**.

Den Animationsablauf kannst du so beschreiben:

Propellerrichtung auf 0 Grad initialisieren
Wiederhole:
    Propeller im Bildbuffer zeichnen
    Propeller auf dem Bildschirm darstellen
    1 / 25 Sekunde warten
    Bild im Bildbuffer loeschen
    Propellerrichtung erhoehen



Mit enableRepaint() stellst du das Grafiksystem auf Doppelbufferung um.

Von dann an werden die Zeichnungsoperation nur noch im Bildbuffer ausgeführt und sind erst dann auf dem Bildschirm ersichtlich, wenn du repaint() aufrufst.

Mit deinen Vorkenntnissen aus der Turtlegrafik verstehst du Programm ohne Probleme und kannst dich an der Animation erfreuen.

```
from gturtle import *

def propeller(a):
    setHeading(a)
    repeat 3:
        fillToPoint()
        rightArc(100, 90)
        right(90)
        rightArc(100, 90)
        left(30)

makeTurtle()
hideTurtle()
# kein automatisches Rendering
enableRepaint(False)

dt = 40  # Zeitinkrement (ms)
```

```
a = 0     # Winkelinitialisierung
da = 10   # Winkelinkrement (Grad)

# Animationsschleife
while True:
    propeller(a)
    repaint()      # Rendern
    delay(dt)      # Warten
    clear("gray") # Löschen
    a += da        # Neue Lage
```

MERKE DIR...

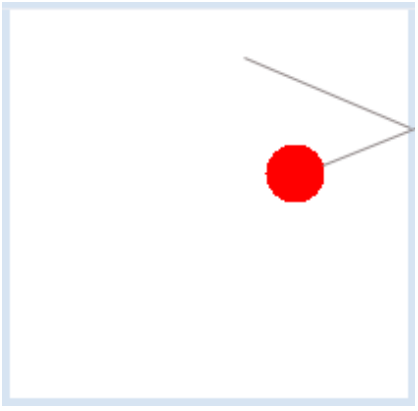
Zum besseren Verständnis des Programms ist es manchmal zweckmässig, an geeigneten Stellen Kommentare einzufügen. Falls die Animation ruckelt, ist der Computer zu wenig leistungsstark oder überlastet. Abhilfe schafft manchmal, wenn du TigerJython neu startest.

ZUM SELBST LÖSEN

1. Ergänze die Propelleranimation mit dem Hintergrundbild eines Flugzeugs. Dazu definierst du am einfachsten eine Funktion *flugzeug(x, y)*, die das Bild aus der Bildbibliothek an der Stelle x, y zeichnet. Du rufst die Funktion periodisch nach der Zeile *clear()* in der Animationsschleife auf.

```
def flugzeug(x, y):
    heading(0)
    setPos(x, y)
    drawImage("sprites/airplane.png")
```

2. Eine Billardkugel bewegt sich im Turtlefensters so, dass sie an den Rändern jeweils unter der Berücksichtigung der Regel *Einfallswinkel = Ausfallswinkel* reflektiert wird.



Anleitung: Mit *heading()* kannst du die aktuelle Turtlerichtung zurückzuholen und sie mit *setHeading(winkel)* neu setzen.

ANIMATION MIT SPRITES

Mit der Turtlegrafik kannst du auch Bilder aus Bilddateien (*Sprites*) animiert darstellen.Die Funktion drawImage(datei) zeichnet das Bild aus der Bilddatei an der aktuellen Lage und mit der aktuellen Blickrichtung der Turtle. Bewegst du nun in der Animationsschleife die Turtle und änderst zudem das Bild, so hast du schon mit wenigen Zeilen Programmcode eine lustige Animation erschaffen.

```
from gturtle import *

makeTurtle()
hideTurtle()
setPos(200, 0)
left(90)
wrap()
enableRepaint(False)

i = 0
while True:
    clear()
    drawImage("sprites/pony_" + str(i) + ".gif")
    repaint()
    delay(60)
    forward(5)
    i += 1
    if i == 7:
        i = 0
```

MERKE DIR...

Die Dateinamen der sieben Spritebilder heissen *"sprites/pony_0.gif"*, *"sprites/pony_1.gif"*, usw. Du kannst sie elegant mit einer String-Konkatenation *"sprites/pony_" + str(i) + ".gif"* erzeugen, wo i von 0 bis 6 läuft. Mit wrap() sorgst du dafür, dass das Pony wieder am rechten Fensterrand erscheint.

ZUM SELBST LÖSEN

3. Ergänze die Ponyanimation mit einem Hintergrundbild. Dieses kannst du hinzufügen mit *drawBkImage("sprites/catbg.png")*. Du kannst auch ein eigenes Hintergrundbild, verwenden, wenn du es in das Unterverzeichnis *sprites* des Verzeichnis kopierst, in dem sich dein Python-Programm befindet. Verwendest du eigene Spritebilder, so musst du darauf achten, dass sie einen transparenten Hintergrund haben.

16. OBJEKTORIENTIERTE PROGRAMMIERUNG (OOP)

DU LERNST HIER...

dass die objektorientierte Programmierung (OOP) ein Programmierkonzept ist, das in vielen Fällen hervorragend geeignet ist, um die reale Welt als Software zu modellieren. So wie du beispielsweise eine lebende Schildkröte als ein bestimmtes Lebewesen einer Tierklasse auffasst, ist eine Turtle in der Turtlegrafik ein bestimmtes Objekt der Softwareklasse *Turtle*.

TURTLE-OBJEKTE

Damit die Turtle ein Zuhause hat, erzeugst du zuerst ein Objekt der Klasse *TurtleFrame* mit dem Namen *tf*. Dazu rufst du die Funktion *TurtleFrame()* auf, die gleich heisst wie die Klasse selbst. Dabei erscheint auf dem Bildschirm ein Turtlefenster.

tf = TurtleFrame()

Man nennt diese Funktion den **Konstruktor** der Klasse *TurtleFrame*. Analog erzeugst du mit dem Konstruktor der Klasse *Turtle* ein Turtleobjekt *tia*, verwendest aber *tf* als Parameter, damit *tia* ins Turtlefenster gesetzt wird: tia = Turtle(tf). Ganz wie in der realen Welt besitzt ein Softwareobjekt bestimmte *Eigenschaften* und *Fähigkeiten*. Beispielsweise ein Turtleobjekt:

Eigenschaften
Turtlefarbe
Blickrichtung
Ort

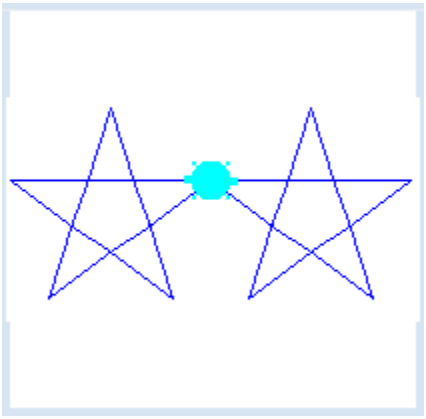
Fähigkeiten
forward(n)
left(w)
right(w)

Mit dem Namen des Objekts kannst du auf die Eigenschaften und Fähigkeiten des Objekts zugreifen, indem du den **Punktoperator** verwendest. Um beispielsweise *tia* vorwärts laufen zu lassen, rufst du die Funktion *tia.forward(100)* auf. Statt *Funktion* sagt man bei Objekten auch oft **Methode**. Für das Turtleobjekt stehen dir alle Methoden zur Verfügung, die du bereits aus der Turtlegrafik kennst.

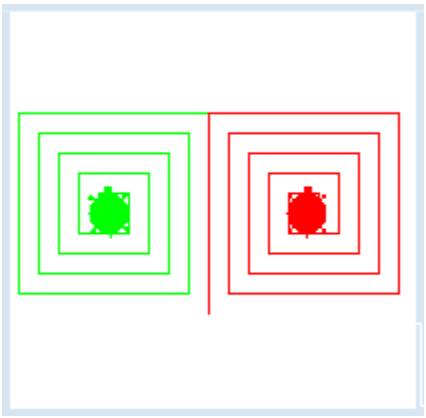
MUSTERBEISPIELE

Im ersten Beispiel lässt du zwei Turtles *tia* und *joe* quasi miteinander einen Stern zeichnen.

```
from gturtle import *

tf = TurtleFrame()
tia = Turtle(tf)
joe = Turtle(tf)
tia.right(90)
joe.left(90)
repeat 5:
    tia.forward(100)
    joe.forward(100)
    tia.right(144)
    joe.left(144)
```

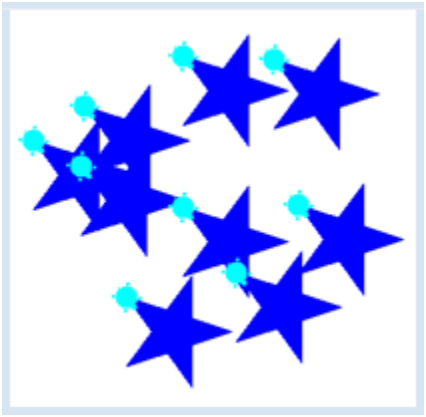


Um die Turtlefarbe zu setzen, verwendest du einen zusätzlichen Parameter im Konstruktor der Turtle, um die Strichfarbe zu setzen, verwendest du die Methode *setPenColor()*. Im folgenden Programm zeichnen die grüne *tia* und der rote *joe* gleichzeitig eine grüne und eine rote Spirale.



```
from gturtle import *

tf = TurtleFrame()
tia = Turtle(tf, "green")
tia.setPenColor("green")
joe = Turtle(tf, "red")
joe.setPenColor("red")

s = 200
while s > 5:
    tia.forward(s)
    joe.forward(s)
    tia.left(90)
    joe.right(90)
    s -= 10
```

Richtig lustig wird es, wenn du das *TurtleFrame* mit vielen Turtles bevölkerst, die du mit einem Mausklick erzeugst und die nach ihrer Entstehung einen gefüllten Stern zeichnen.



In diesem Beispiel werden die grossen Vorteile der OOP offensichtlich, denn jedes neue Turtleobjekt weiss von sich aus, wie es sich benehmen muss.

```
from gturtle import *

def onMouseHit(x, y):
    t = Turtle(tf)
    t.setPos(x, y)
    star(t)

def star(t):
    t.startPath()
    repeat 4:
        t.forward(100)
        t.left(144)
    t.fillPath()

tf = TurtleFrame(mouseHit = onMouseHit)
```

ANIMATIONEN MIT MEHREREN AKTOREN

Brauchst du für eine Animation mehrere Aktoren, so kannst du mehrere Turtleobjekte erzeugen und jedem davon ein eigenes Animationsbild zuordnen. Du bewegst im folgenden Programm das rosa Pony von rechts nach links und das blaue Pony gleichzeitig von links nach rechts. Damit das blaue Pony in der richtigen Lage erscheint, musst du sein Bild vertikal spiegeln. Das machst du mit zwei zusätzlichen booleschen Parametern in *drawImage(),* die man wie folgt verwenden kann:

drawImage(img, False, False)	keine Spiegelung
drawImage(img, True, False)	horizontale Spiegelung
drawImage(img, False, True)	vertikale Spiegelung
drawImage(img, True, True)	horizontale und vertikale Spiegelung

```
from gturtle import *

tf = TurtleFrame()
tia = Turtle(tf)
tia.ht()
tia.setPos(250, 0)
tia.left(90)
tia.wrap()
img_tia = [0] * 8
for i in range(8):
    img_tia[i] = "sprites/pony_" + str(i) + ".gif"

joe = Turtle(tf)
joe.ht()
joe.setPos(-250, 0)
joe.right(90)
joe.wrap()
img_joe = [0] * 8
for i in range(8):
    img_joe[i] = "sprites/pony2_" + str(i) + ".gif"

tf.enableRepaint(False)
repeat:
    for i in range(8):
        tia.drawImage(img_tia[i])
        joe.drawImage(img_joe[i], False, True)
        tf.repaint()
        tf.delay(100)
        tf.clear()
        tia.forward(5)
        joe.forward(5)
```

MERKE DIR...

Klassen fassen in der objektorientierten Programmierung (OOP) Eigenschaften und Fähigkeiten zusammen. Ein Objekt wird durch Aufruf des Konstruktors erzeugt, der den gleichen Namen wie die Klasse, aber eine Parameterklammer hat. Der Rückgabewert des Konstruktors wird einer Variablen zugewiesen, mit der man mittels des Punktoperators auf das Objekt zugreifen kann.

ZUM SELBST LÖSEN

1. a) Zwei Turtles zeichnen mit dem Befehl *dot()* von unten nach oben abwechslungsweise eine grüne bzw. eine rote Perle und erzeugen zwei Perlenketten.

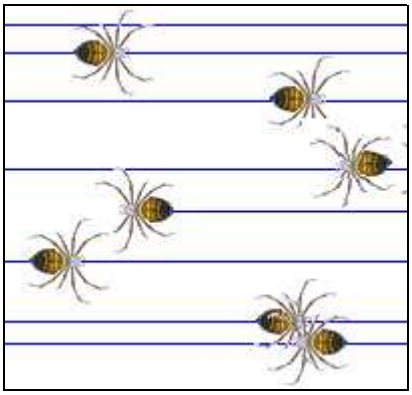
   b) Die Turtles zeichnen nur eine einzige Perlenkette, die abwechslungsweise eine grüne und eine rote Perle hat.

a) b)

2. In einem Turtlefenster soll mit jedem Mausklick ein neues Turtleobjekt erzeugt werden, wobei der Turtle das Spritebild *spider.png* zugeordnet ist:

   ```
   t = Turtle(tf, "sprites/spider.png")
   ```

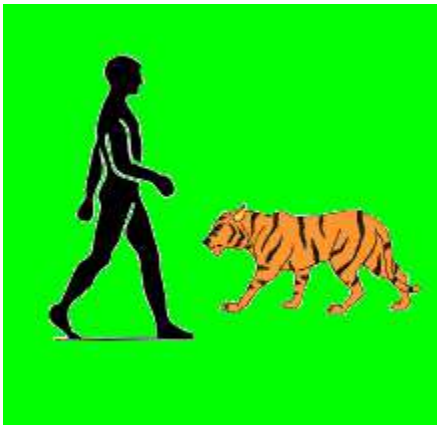
   Jede Turtle dreht zuerst 90° nach rechts und bewegt sich vorwärts. Wenn sie am rechten Rand ankommt (getX() > 300), dreht sie um 180° und läuft wieder zurück. Analog am linken Rand.



3. Erstelle eine endlose Animation mit zwei Ponys, die miteinander laufen.

   Verwende die Bilder pony_0.gif,...,pony_7.gif.

4. Verwende die Sprites
   person_0.png,. .,person_4.png sowie
   tiger_2.gif,...,tiger_6.gif
   und erstelle eine endlose Animation, bei der die Person von links nach rechts und der Tiger von rechts nach links laufen.



Dokumentation Turtlegrafik

Befehl	Aktion
makeTurtle()	erzeugt eine (globale) Turtle im neuen Grafikfenster
makeTurtle(color)	erzeugt eine Turtle mit angegebener Farbe
makeTurtle("sprites/turtle.gif")	erzeugt Turtle mit einem eigenen Turtle-Bild turtle.gif
t = Turtle()	erzeugt ein Turtleobjekt t
tf = TurtleFrame()	erzeugt ein Bildschirmfenster, in dem mehrere Turtles leben
tf = TurtleFrame(title)	dasselbe, aber mit gegebenen Titel
t = Turtle(tf)	erzeugt ein Turtleobjekt t im TurtleFrame tf
clone()	erzeugt ein Turtleklon (gleiche Farbe, Position, Blickrichtung)
isDisposed()	gibt True zurück, falls das Turtlefenster geschlossen ist
putSleep()	hält den Programmablauf an, bis wakeUp() aufgerufen wird
wakeUp()	führt angehaltenen Programmablauf weiter
enableRepaint(False)	schaltet das automatische Bildschirmrendering aus
repaint()	rendert den Bildschirm (nach dem Ausschalten des automatischen Rendering)
savePlayground()	speichert den Playground in einem internen Bildbuffer (zurückholen mit clear())
savePlayground(fileName, format)	speichert den Playground als Bilddatei (format: "png" oder "gif"). Im Fehlerfall wird False zurückgegeben
Options.setFramePosition(x, y)	setzt die obere linke Ecke des Turtlefensters an die gegebene Bildschirmposition (muss vor makeTurtle() aufgerufen werden)
Options.setPlaygroundSize(width, height)	setzt die Grösse des Turtlefensters unabhängig von den Einstellungen in TigerJython (muss vor makeTurtle() aufgerufen werden)

Bewegen

back(distance), bk(distance)	bewegt Turtle rückwärts
forward(distance), fd(distance)	bewegt Turtle vorwärts
hideTurtle(), ht()	macht Turtle unsichtbar (Turtle zeichnet schneller)
home()	setzt Turtle in die Mitte des Fensters mit Richtung nach oben
left(angle), lt(angle)	dreht Turtle nach links
penDown(), pd()	setzt Zeichenstift ab (Spur sichtbar)
penErase(), pe()	setzt die Stiftfarbe auf die Hintergrundfarbe
leftArc(radius, angle)	bewegt Turtle auf einem Bogen mit dem Sektor-Winkel *angle* nach links
leftCircle(radius)	bewegt Turtle auf einem Kreis nach links
penUp(), pu()	hebt den Zeichenstift (Spur unsichtbar)
penWidth(width)	setzt die Dicke des Stifts in Pixel
right(angle), rt(angle)	dreht Turtle nach rechts

rightArc(radius, angle)	bewegt Turtle auf einem Bogen mit dem Sektor-Winkel *angle* nach rechts
rightCircle(radius)	bewegt Turtle auf einem Kreis nach rechts
setCustomCursor(cursorImage)	wählt die Bilddatei des Mauscursors
setCustomCursor(cursorImage, Point(x, y))	wählt die Bilddatei des Mauscursors unter Angabe der Mausposition innerhalb des Bildes
setLineWidth(width)	setzt die Dicke des Stifts in Pixel
showTurtle(), st()	zeigt Turtle
speed(speed)	setzt Turtlegeschwindigkeit
delay(time)	hält das Programm während der Zeit time (in Millisekunden) an
wrap()	setzt Turtlepositionen ausserhalb des Fensters ins Fenster zurück
clip()	turtles ausserhalb des Fensters sind nicht sichtbar
savePlayground(fileName, format)	speichert den Playground als Bilddatei (format: "png" oder "gif"). Im Fehlerfall wird False zurckgegeben
setPlaygroundSize(width, height)	setzt die Grösse des Turtlefensters unabhängig von den Einstellungen in TigerJython (muss vor makeTurtle() aufgerufen werden)
setFramePosition(x, y)	setzt die obere linke Ecke des Turtlefensters an die gegebene Bildschirmposition
setFramePositionCenter()	setzt das Turtlefensters in Bildschirmmitte

Positionieren

direction(x, y)	gibt den Winkel (in Grad) für die Drehung zur Position (x, y) zurück
direction(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
direction(turtle)	gibt den Winkel (in Grad) für die Drehung zu einer anderen Turtle zurück
distance(x, y)	gibt die Entfernung der Turtle zum Punkt(x, y) zurück
distance(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
distance(turtle)	gibt die Entfernung der Turtle zu einer anderen Turtle zurück
getPos()	gibt die Turtleposition zurück als Punkt
getX()	gibt die aktuelle x-Koordinate der Turtle zurück
getY()	gibt die aktuelle y-Koordinate der Turtle zurück
heading()	gibt die Richtung der Turtle zurück
heading(degrees)	setzt die Richtung der Turtle (0 ist gegen oben, im Uhrzeigersinn)
moveTo(x, y)	bewegt Turtle auf die Position (x, y)
moveTo(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
setHeading(degrees), setH(degrees)	setzt die Richtung der Turtle (0 gegen oben, im Uhrzeigersinn)
setRandomHeading()	setzt die Richtung zufällig zwischen 0 und 360°
setPos(x, y)	setzt Turtle auf die Position (x, y)

setPos(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
setX(x)	setzt Turtle auf x-Koordinate
setY(y)	setzt Turtle auf y-Koordinate
setRandomPos(width, height)	setzt Turtle auf einen zufälligen Punkt im Bereich 0..width, 0..height
setScreenPos(x, y)	setzt Turtle auf gegebene Pixelkoordinaten (x, y)
setScreenPos(Point(x, y))	setzt Turtle auf gegebene Pixelkoordinaten
towards(x, y)	gibt die Richtung (in Grad) zur Position (x, y) (auch list, tuple, complex)
towards(turtle2)	gibt die Richtung (in Grad) zu einer anderen Turtle turtle2 zurück
toTurtlePos(x, y)	gibt die Turtlekoordinaten zu den gegebenen Pixelkoordinaten (x, y) als Liste zurück
toTurtlePos(Point(x, y))	gibt die Turtlekoordinaten zu den gegebenen Pixelkoordinaten als Liste zurück
pushState()	speichert den Turtlezustand in einem Stapelspeicher
popState()	holt den zuletzt gespeicherten Zustand vom Stapelspeicher
clearStates()	löscht den Stapelspeicher

Farben

askColor(title, defaultColor)	zeigt ein Fenster zur Farbwahl und gibt die gewählte Farbe zurück, (None, wenn Abbrechen gedrückt wurde)title: Fenstertitel an, defaultColor: Farbvorschlag
clear()	löscht die Zeichnung und versteckt alle Turtles (sie bleiben am Ort). Falls der mit savePlayground() erzeugte Bildbuffer nicht leer ist, wird dieser angezeigt
clear(color)	löscht die Zeichnung, versteckt alle Turtles (sie bleiben am Ort) und färbt den Hintergrund
class ="doc"clean()	löscht die Turtlespuren (Turtles bleiben am Ort sichtbar). Der mit savePlayground() erzeugte Bildbuffer wird gelöscht
clean(color)	löscht die Zeichnung und färbt den Hintergrund (Turtles bleiben am Ort sichtbar).
clearScreen(), cs()	löscht die Turtlespuren und setzt die Turtle an Homeposition
dot(diameter)	zeichnet einen mit Stiftfarbe gefüllten Kreis
openDot(diameter)	zeichnet einen nicht gefüllten Kreis
spray(density, spread, size)	zeichnet eine zufällige Punktwolke an der Turtleposition mit geg. Anzahl Punkten, Ausdehnung und Punktgrösse (ohne size: size = 1)
fill()	füllt die geschlossene Figur, in der sich die Turtle befindet mit der Füllfarbe
fill(x , y)	füllt eine geschlossene Figur um den inneren Punkt (x, y) mit der Füllfarbe
fill(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
fillToPoint()	füllt fortlaufend die gezeichnete Figur von der aktuellen Turtleposition mit der Stiftfarbe
fillToPoint(x , y)	füllt fortlaufend die gezeichnete Figur vom Punkt (x, y) mit der Stiftfarbe

fillToPoint(coords	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
fillToHorizontal(y)	füllt fortlaufend die Fläche zwischen der Figur und der horizontalen Linie in Höhe y mit der Stiftfarbe
fillToVertical(x)	füllt fortlaufend die Fläche zwischen der Figur und der vertikalen Linie am Wert xmit der Stiftfarbe
fillOff()	beendet den fortlaufenden Füllmodus
getColor()	gibt die Turtlefarbe zurück
getColorStr()	gibt die X11-Turtlefarbe zurück
getFillColor()	gibt die Füllfarbe zurück
getFillColorStr()	gibt die X11-Füllfarbe zurück
getPixelColor()	gibt die Farbe des Pixels an der Turtlekoordinate zurück (None, falls ausserhalb des Turtlefensters)
getPixelColorStr()	gibt die Farbe des Pixels an der Turtlekoordinate als X11-Farbe zurück (leerer String, falls kein X11-Farbname existriert; None, falls ausserhalb des Turtlefensters)
getRandomX11Color()	gibt eine zufällige X11-Farbe zurück
makeColor()	gibt eine Farbreferenz von value zurück. Werte-Beispiele: ("7FFED4"), ("Aqua-Marine"), (0x7FFED4), (8388564), (0.5, 1.0, 0.83), (128, 255, 212), ("rainbow", n) mit n = 0..1, Lichtspektrum
setColor(color)	legt Turtlefarbe fest
setPenColor(color)	legt Stiftfarbe fest
setPenWidth(width)	setzt die Dicke des Stiftes in Pixel
setFillColor(color)	legt Füllfarbe fest
startPath()	startet die Aufzeichnung der Turtlebewegung zum nachträglichen Füllen
fillPath()	verbindet die aktuelle Turtleposition mit dem Startpunkt und füllt die geschlossene Figur mit der Füllfarbe
stampTurtle()	erzeugt ein Turtlebild an der aktuellen Turtleposition
stampTurtle(color)	erzeugt ein Turtlebild mit angegebener Farbe an der aktuellen Turtleposition

Callbacks

makeTurtle(mouseNNN = onMouseNNN) auch mehrere, durch Komma getrennt	registriert die Callbackfunktion onMouseNNN(x, y), die beim Mausevent aufgerufen wird. Werte für NNN: Pressed, Released, Clicked, Dragged, Moved, Entered, Exited, SingleClicked, DoubleClicked, Hit: Aufruf im eigenen Thread, HitX: Dasselbe, aber nachfolgende Events ignoriert, bis Callback zurückkehrt
isLeftMouseButton(), isRightMouseButton()	gibt True zurück, falls beim Event die linke bzw. rechte Maustaste verwendet wurde
makeTurtle(keyNNN = onKeyNNN)	registriert die Callbackfunktion onKeyNNN(keyCode), die beim Drücken einer Tastaturtaste aufgerufen wird. Werte für NNN: Pressed, Hit: Aufruf im eigenen Thread, HitX: Dasselbe, aber nachfolgende Events ignoriert, bis Callback zurückkehrt. keyCode ist ein für die Taste eindeutiger integer Code

getKeyModifiers()	liefert nach einem Tastaturevent einen Code für Spezialtasten (Shift, Ctrl, usw., auch kombiniert)
makeTurtle(closeClicked = onCloseClicked)	registriert die Callbackfunktion onCloseClicked(), die beim Klick des Close-Buttons des Turtlefensters aufgerufen wird. Das Fenster muss mit dispose() geschlossen werden
makeTurtle(turtleHit=onTurtleHit)	registriert die Callbackfunktion onTurtleHit(x, y), die aufgerufen wird, wenn auf das Turtlebild geklickt wird
t = Turtle(turtleHit = onTurtleHit)	registriert die Callbackfunktion onTurtleHit(t, x, y), die aufgerufen wird, wenn auf das Bild der Turtle t geklickt wird
showSimulationBar(NNN = onNNN)	zeigt ein Dialogfenster mit den Button 'Step', 'Run'/'Pause', 'Reset' und einem Slider zum Einstellen der Simulationsperiode. Folgende Callbacks NNN können registiert werden: start, pause, step, reset, change(Parameter: simulationPeriod), loop, exit (close button gedrückt). loop wird in jedem Simulationszyklus vom Simulationsthread aufgerufen
showSimulationBar(ulx, uly, initPeriod, NNN = onNNN)	wie oben, aber mit Positionierung des Dialogfensters (obere linke Ecke) und Anfangswert der Simulationsperiode (default: 100)
hideSimulationBar()	schliesst das Dialogfenster und gibt alle Ressourcen frei

Tastatur

getKey()	holt den letzten Tastendruck ab und liefert String zurück (leer, falls illegale Taste)
getKeyCode()	holt den letzten Tastendruck ab und liefert Code zurück
getKeyWait()	wartet bis Taste gedrückt und liefert String zurück (leer, falls illegale Taste)
getKeyCodeWait()	wartet bis Taste gedrückt und liefert Code zurück
kbhit()	liefert True, falls ein Tastendruck noch nicht mit getKey() od. getKeyCode() abgeholt ist

Texte, Bilder und Sound

addStatusBar(20)	fügt eine Statusbar mit der Höhe 20 Pixel hinzu
beep()	erzeugt einen Ton
playTone(freq)	spielt Ton mit gegebener Frequenz (in Hz) 1000 ms (blockierende Funktion)
playTone(freq, block = False)	das selbe, aber nicht blockierende Funktion (um mehrere Töne gleichzeitig abzuspielen)
playTone(freq, duration)	spielt Ton mit gegebener Frequenz und Dauer
playTone([f1, f2, f3 ...])	spielt hintereinander mehrere Töne mit geg. Frequenzen
playTone([(f1, d1),(f2, d2), (f3, d3)...])	spielt hintereinander mehrere Töne mit geg. Frequenzen und Dauer
playTone([("c", 700),("e", 1500)...]	spielt hintereinander mehrere Töne mit geg. Tonbezeichnungen und geg. Dauer. Erlaubt sind: grosse Oktave , ein- , zwei- und dreigestrichene Oktave also im Bereich c, c#, ...h''')
playTone([("c", 700),("e", 1500)...], instrument="piano")	wie vorher, aber mit gewähltem Instrument (piano, quitar, harp, trumpet, organ, panfluete, seashore, violin, xylophone... (gemäss MIDI Spezifikation)).

playTone([("c", 700),("e", 1500...)], instrument="piano", volumen = 10)	wie vorher, aber mit gewählten Lautstärke (0....100)
label(param)	schreibt str(param) an der aktuellen Position aus (linksbündig)
label(param1, param2, ...)	konkateniert str(params) mit Leerstellen getrennt und schreibt den Text aus
label(param1, param2, ..., adjust = 'x')	dasselbe, aber mit x = 'l' linksbündig (default), 'c' zentriert, 'r' rechtsbündig
printerPlot(draw)	druckt die mit der Funktion draw erstellte Zeichnung
setFont(Font font)	legt Schriftart fest. font ist ein Objekt der Klasse Font Beispiel:. Font("Courier New", Font.BOLD, 12). Default: Font("SansSerif", Font.PLAIN, 24)
setFont(name)	legt neue Schriftart mit bestehenden Schriftstil und Schriftgrösse fest
setFont(name, style)	legt neue Schriftart und Schriftstil mit bestehender Schriftgrösse fest style = Font.PLAIN, Font.BOLD, Font.ITALIC
setFont(name, style, size)	legt neue Schriftart, Schriftstil und Schriftgrösse fest
setFontSize(size)	legt neue Schriftgrösse fest (Schriftart und Stil bleiben erhalten)
getTextHeight()	liefert die Höhe des Texts im aktuellen Font (in Pixels)
getTextAscent()	liefert die Höhe des Text-Ascenders im aktuellen Font (in Pixels)
getTextDescent()	liefert die Höhe des Text-Decenders im aktuellen Font (in Pixels)
getTextWidth(text)	liefert die Breite des gegebenen Texts im aktuellen Font (in Pixels)
setStatusText("Press any key!")	schreibt eine Mitteilung in die Statusbar
setTitel("Text")	schreibt den Text in die Titelzeile
img = getImage(path)	lädt ein Bild (im png- gif, jpg-Format) vom lokalen Filesystem oder von einer URL und gibt eine Referenz darauf zurück. Für path = sprites/nnn werden auch Bilder von der TigerJython-Distribution geladen (Help/Bilderbibliothek zeigt die vorhanden Bilder)
drawImage(img)	stelt das Bild img an der Position der Turtle mit ihrer Blickrichtung dar
drawImage(path)	lädt ein Bild (im png-, gif-, jpg-Format) vom lokalen Filesystem oder von einer URL und stellt es an der Position der Turtle mit ihrer Blickrichtung dar. Für path = sprites/nnn werden auch Bilder von der TigerJython-Distribution geladen

Video

rec = VideoRecorder(turtle, filename, resolution)	erstellt einen MP4 (H.264/AVC) Video-Encoder, der das Turtlefenster Turtles, Spuren und Images) aufzeichnet und in der gegebenen Datei abgespeichert. resolution muss ein String mit einer unterstützten Auflösung sein, beispielsweise "640x480". getSupportedResolutions() liefert die aktuell unterstützten Auflösung
rec = VideoRecorder(turtleFrame, filename, resolution)	dasselbe mit einem TurtleFrame
rec = VideoRecorder(turtle, filename, resolution, ulx, uly)	dasselbe mit Angabe der Position der oberen linken Ecke des Turtlefensters bezüglich des Video-Bildes. Standardwerte 0, 0

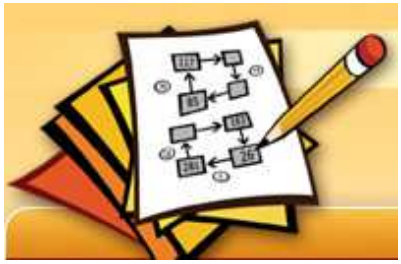
rec = VideoRecorder(turtleFrame, filename, resolution)	dasselbe mit einem TurtleFrame
VideoRecorder. getSupportedResolutions()	liefert einen String mit allen aktuell unterstützten Video-Auflösungen
rec.captureImage()	nimmt ein einzelnes Bild (frame) auf. Die Aufnahme kann in beliebigen zeitlichen Abständen erfolgen. Die Bildabspielrate (frame rate) hängt von der Auflösung ab (normalerweise 25 frames / s)
rec.captureImage(nb)	dasselbe, aber es werden nb gleiche Bilder aufgenommen
rec.finish()	die Aufnahme wird beendet und die Videodatei geschlossen

Dialoge

msgDlg(message)	öffnet einen modalen Dialog mit einem OK-Button und gegebenem Mitteilungstext
msgDlg(message, title = title_text)	dasselbe mit Titelangabe
inputInt(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegebenen Integer zurück (falls kein Integer, wird Dialog neu angezeigt). Abbrechen od. Schliessen beendet das Programm
inputInt(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
inputFloat(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegebenen Float zurück (falls kein Float, wird Dialog neu angezeigt). Abbrechen od. Schliessen beendet das Programm
inputFloat(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
inputString(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt den eingegeben String zurück. Abbrechen od. Schliessen beendet das Programm
inputString(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
input(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt Eingabe als Integer, Float oder String zurück. Abbrechen od. Schliessen beendet das Programm
input(prompt, False)	dasselbe, aber Abbrechen/Schliessen beendet das Programm nicht, sondern gibt None zurück
askYesNo(prompt)	öffnet einen modalen Dialog mit Ja/Nein-Buttons. Ja gibt True, Nein gibt False zurück. Schliessen beendet das Programm
askYesNo(prompt, False)	dasselbe, aber Schliessen beendet das Programm nicht, sondern gibt None zurück

ARBEITSBLÄTTER

In den Arbeitsblättern lernst du, wie du deine Programmierkenntnisse für etwas umfangreichere Problemstellungen anwenden kannst. Sie sind als eine Ergänzung zu den Lehrgängen Turtlegrafik, Robotik und Datenbanken gedacht.



Die zur Lösung nötigen Vorkenntnisse sind unterschiedlich, es wird aber nicht vorausgesetzt, dass der Lehrgang vorgängig vollständig durchgearbeitet wurde. Teilweise müssen auch zusätzliche Programmierkenntnisse selbst erarbeitet werden, die im Lehrgang fehlen.

Die Aufgaben sind meist so offen formuliert, sodass genügend Spielraum für eigene Ideen und Lösungsansätze bleiben. Die Arbeitsblätter werden laufend ergänzt.

ARBEITSBLATT 1: SIEBENSEGMENTANZEIGE

EINLEITUNG

Siebensegmentanzeigen sind auch heute noch wegen ihrer Einfachheit und guten Lesbarkeit weit verbreitet. Sie bestehen aus 7 länglichen Leuchtquellen, Segmente genannt, die gewöhnlich aus Leuchtdioden (LEDs) bestehen. Die Form der Segmente kann leicht variieren, z.B. können die Segment geneigt oder an den Enden schräg abgeschnitten sein. Oft haben 7-Segmentanzeigen auch noch zusätzliche kleine LEDs, die als Dezimalpunkte oder Doppelpunkte. verwendet werden. Einzelne Displays werden hintereinander zu einem 7-Segment-Array angeordnet.

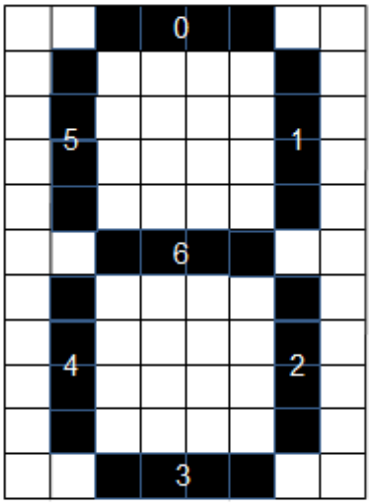
AUFGABENSTELLUNG UND ZIELSETZUNG

7-Segmentanzeigen werden auch oft auf Computerbildschirmen zur Anzeige von Daten und Messwerten, insbesondere für virtuelle Messinstrumente verwendet, da sie leicht lesbar sind. Darum soll in dieser Projektarbeit ein 7-Segment-Array mit einer beliebig wählbaren Zahl von Einzeldisplays softwaremässig modelliert werden.

Der Programmcode soll sich in verschiedenen Anwendungen einsetzen lassen, beispielsweise zur digitalen Anzeige der aktuellen Uhrzeit in Stunden, Minuten und Sekunden. Dabei wird Python als Programmiersprache und die TigerJython-Turtlegrafik als Grafiksystem verwendet.

Standardmässig werden die Segmente mit den Buchstaben A bis G oder mit den Zahlen 0 bis 6 identifiziert. Wir verwenden hier die Nummerierung, da sie der Datenstruktur von Listen und Tupels besser angepasst ist.

Es wird die im nebenstehenden Bild ersichtliche Nummerierung verwendet:



AUFGABE 1

7-Segmentanzeigen werden auch oft auf Computerbildschirmen zur Anzeige von Daten und Messwerten, insbesondere für virtuelle Messinstrumente verwendet, da sie leicht lesbar sind. Darum soll in dieser Projektarbeit ein 7-Segment-Array mit einer beliebig wählbaren Zahl von Einzeldisplays softwaremässig modelliert werden.

VORBEREITUNG ZUR AUFGABE 1

Das in der Abbildung gezeigte Raster stellt quadratische Zellen mit der Seitenlänge s dar, die als Konstante gewählt werden kann. Typisch ist s = 10 (in Turtlekoordinaten).

Die Informationen über die Lage und Ausrichtung der Segmente wird ein einer Liste gespeichert. Die Listenstruktur kann eher "flach" oder stark strukturiert sein. Wir wählen die flache Variante und speichern jedes Segment in 3 Zahlen *x, y, heading*, wo x,y die Koordinaten des Startpunkts und heading die Bewegungsrichtung der Turtle (0 oder 90 Grad) sind. Das folgende Programmskelett zeigt dies für die Segmente 0 bis 3 und als Test werden diese Segmente der Reihe nach gezeigt.

```
from gturtle import *

s = 10

def drawSegment(i):
    setPos(segments[3*i], segments[3*i + 1])
    heading(segments[3*i + 2])
    forward(4*s)

segments = [2*s, 12*s, 90, 7*s, 7*s, 0, 7*s, s, 0]

makeTurtle()
hideTurtle()
setLineWidth(s)
for p in range(3):
   drawSegment(p)
   delay(1000)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Ergänze das Skelett, dass alle Segmente durchlaufen werden.

AUFGABE 2

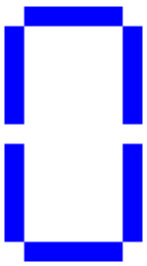
Es ist deine Aufgabe, die Funktion *drawNum(n)* zu schreiben, die für den Parameter n = 0..9 die entsprechende Ziffer anzeigt, d.h. die richtigen Segmente zeichnet.

VORBEREITUNG ZUR AUFGABE 2

Die Information, welche Segmente bei einer bestimmten Zahl n = 0..9 leuchten, wird in der Liste *patterns* als Tupel der Nummern der angezeigten Segmente gespeichert. Wie du leicht nachprüfen kannst, lauten die Tupel für die Zahlen 0..3 wie folgt:

patterns = [(0, 1, 2, 3, 4, 5), (1, 2), (0, 1, 6, 4, 3, 4), (0, 1, 6, 2, 3)]

Es ist deine Aufgabe, das folgende Skelett so zu ergänzen, dass die auskommentierten Zeilen die Ziffern 0..9 ausschreiben. Beachte, dass du in der Wiederholschleife mit *clear()* die Ziffern immer wieder löschen musst.

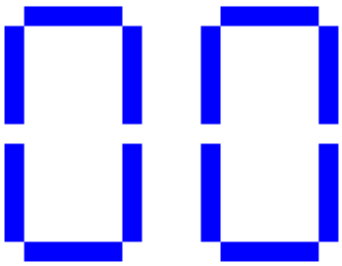


```
from gturtle import *

s = 10

def drawSegment(i):
    setPos(segments[3*i], segments[3*i + 1])
    heading(segments[3*i + 2])
    forward(4*s)

def drawNum(n):
    pattern = patterns[n]
    for segment in pattern:
        drawSegment(segment)

segments = [2*s, 12*s, 90, 7*s, 7*s, 0, 7*s, s, 0]

patterns =  [(0, 1, 2, 3, 4, 5), (1, 2),
             (0, 1, 6, 4, 3, 4), (0, 1, 6, 2, 3)]

makeTurtle()
hideTurtle()
setLineWidth(s)

drawNum(1)
#for n in range(10):
#    clear()
#    drawNum(n)
#    delay(1000)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

AUFGABE 3

Damit du mehrere Ziffern darstellen kannst, führst du in den Funktionen *drawSegment(p, xpos)* und *drawNum(n, xpos)* einen weiteren Parameter *xpos* ein, der die x-Koordinate der Segmente beim Zeichnen verschiebt. Führe dies durch und stelle 2 Ziffern dar, die wie ein Sekundenzähler von 0 bis 99 hinaufzählen.



VORBEREITUNG ZUR AUFGABE 3

Um für Zahlen > 10 die beiden Ziffern zu bestimmen, verwendest du für die Einer den Modulo-Operator % und für die Zehner die Ganzzahldivision mit dem verdoppelten Bruchstrich //. Die Zählschleife lautet:

```
for n in range(100):
    clear()
    if n < 10:
        drawNum(0, 0)
        drawNum(n, 100)
    else:
        drawNum(n // 10, 0)
        drawNum(n % 10, 100)
    delay(1000)
```

AUFGABE 4

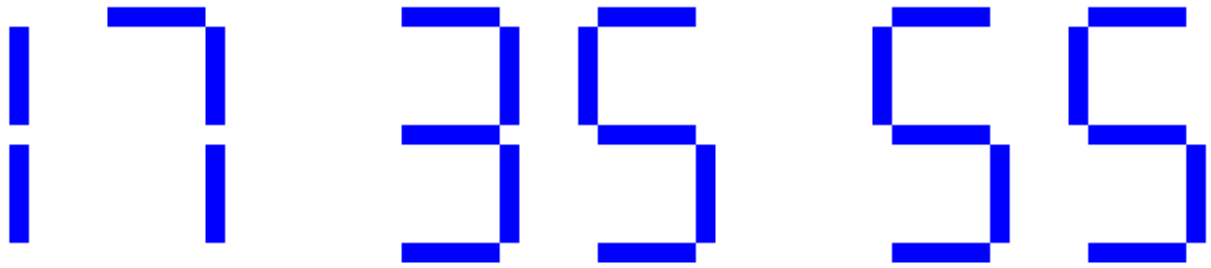
Stelle nun die Aufgabe fertig und zeige die aktuelle Uhrzeit im Format hh mm ss mit 6 Ziffern an.

VORBEREITUNG ZUR AUFGABE 3

Wenn du *import time* verwendest, so kannst du Stunden, Minuten und Sekunden mit folgender Funktion als Strings im Format hhmmss erhalten:

```
t = time.strftime("%H%M%S")
```

Die Ziffern als Zahlen musst du dann mit *int(t[0]), int(t[1])*, ..,*int(t[5])* extrahieren. Du solltest aus Effizienzgründen die Wiederholschleife nur ungefähr alle 50 ms durchlaufen, da ja die Anzeige sowieso nur alle Sekunden ändert. Auch so kann das Bild flackern, da es sich ja um eine Computeranimation handelt. Wie du in der entsprechenden Lektion gelernt hast, solltest du daher mit *enableRepaint(False)* die Doppelbufferung einschalten und die Anzeige mit *repaint()* auf dem Bildschirm rendern.



VERBESSERNGSVORSCHLÄGE/ERWEITERUNGEN

Es gibt mehrere Verbesserungsmöglichkeiten, beispielsweise

- Display mit Rahmen versehen
- Verschiedenfarbige Displays
- Segmente mit abgeschrägten Enden
- Display mit zusätzlichen Dezimalpunkten/Doppelpunkt
- Software-Modellierung mit objekt-orientierter Programmierung (OOP). d.h. ein Display wird als eine Klasseninstanz (Objekt) aufgefasst und seine Eigenschaften und Fähigkeiten sind in in Attributen und Methoden gekapselt

Deinem Erfindergeist und deiner Phantasie sind fast keine Grenzen gesetzt.

ARBEITSBLATT 2: LABYRINTH

EINLEITUNG

Du wirst hier ein kleines Spiel selbst programmieren. Dabei wird die Turtle in ein "Labyrinth" gesetzt, in dem sie ihren Weg zum Ziel suchen muss. Dabei unterstützt du die Turtle, indem du ihren Weg mit schwarzen Feldern blockierst, so dass sie sich abdreht (ein schwarzes Feld erzeugst du über einen Mausklick).

EINSTIEG

Wir geben dir hier erst einmal nur die Grundstruktur für das Spiel an. Die Turtle bewegt sich auf einer Art von "Schachbrett". Mit Klicken kannst du ein Feld schwarz färben.

Bei einem grösseren Programm ist es wichtig, dass du mit Kommentaren erklärst, was die einzelnen Teile machen. Kommentare beginnen jeweils mit #. Python ignoriert alle solchen Kommentare, sie dienen nur dem Leser zum besseren Verständnis.

Nachdem wir zuerst alle Befehl definiert haben, beginnt das Hauptprogramm erst danach. Das machen wir deutlich mit einem Kommentar

```
### ------------- MAIN --------------- ###
```

(eine Kurzform für main program, also Hauptprogramm). Auch das hat für Python keine Bedeutung, hilft uns aber, die Übersicht zu behalten.

```
from gturtle import *

CELLSIZE = 40 #Wähle zwischen: 10, 20, 40, 50

# Zeichnet das Grundgitter:
def drawGrid():
    global CELLSIZE
    hideTurtle()
    setPenColor("gray")
    x = -400
    repeat (800 // CELLSIZE) + 1:
        setPos(x, -300)
        moveTo(x, +300)
        x += CELLSIZE
    y = -300
    repeat (600 // CELLSIZE) + 1:
        setPos(-400, y)
        moveTo(+400, y)
```

```
        y += CELLSIZE
    setPos(0, 0)
    showTurtle()

@onMouseHit
def onClick(x, y):
    turtle_x = getX()
    turtle_y = getY()
    # Zelle schwarz färben
    hideTurtle()
    setPos(x, y)
    if getPixelColorStr() == "white":
        setFillColor("black")
        fill(x, y)
    # Die Turtle wieder dahin zurücksetzen,
    # wo sie am Anfang war.
    setPos(turtle_x, turtle_y)
    showTurtle()

def doStep():
    hideTurtle()
    # Einen Schritt nach vorne machen.
    forward(CELLSIZE)
    # Falls die Turtle auf einem schwarzen Feld landet,
    # setzen wir sie wieder zurück und drehen sie dafür.
    if getPixelColorStr() == "black":
        back(CELLSIZE)
        right(90)
    showTurtle()


### ---------------------- MAIN --------------------------- ###
makeTurtle()
drawGrid()
# An dieser Stelle könntest du ein Feld als Ziel färben.

# Die Turtle auf ein Anfangsfeld setzen:
setPos(-400 + 5 * CELLSIZE // 2, -300 + 5 * CELLSIZE // 2)
penUp()

repeat 1000:
    doStep()
    delay(500)
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Bevor wir die Turtle an eine bestimmte Stelle bewegen, machen wir sie mit *hideTurtle()* unsichtbar und zeigen sie danach wieder mit *showTurtle()*. Das hat zwei Gründe: Zum einen wäre es z. B. merkwürdig, wenn die Turtle bei einem Mausklick kurz zum Mauscursor springt und danach wieder zurückgeht. Zum anderen wäre das Programm zu langsam, wenn die Turtle bei all ihren Bewegungen sichtbar wäre.

AUFGABENSTELLUNG

1. Ergänze das Programm so, dass du mit der Maus schwarze Felder auch wieder wegklicken kannst. Wenn du also auf ein schwarzes Feld klickst, dann wird es wieder weiss.

2. Ergänze das Programm zu einem Spiel, indem du ein Feld rot einfärbst. Wenn die Turtle dieses rote Feld erreicht hat, ist das Spiel fertig und man hat «gewonnen». Im Modul *sys* gibt es übrigens einen Befehl *exit()*, um das Programm sofort zu beenden:

```
from sys import exit
exit() # Programm beenden
```

3. Im Moment kann es passieren, dass die Turtle aus dem Bild herausfällt. Ergänze das Programm also so, dass du alle Felder am Rand zuerst schwarz färbst.

4. Das Spiel wird erst dann interessant, wenn du gewisse Hindernisse oder Punkte einbaust. Du könntest z. B. die Anzahl der schwarzen Blöcke zählen, die man braucht, um das Spiel zu lösen. Je weniger Blöcke, umso höher die Punktzahl. Oder du zählst die Schritte, die die Turtle braucht. Überlege dir selber, wie du die Punktzahl berechnen möchtest und gib am Ende diese Punktzahl mit *msgDlg* aus!

5*. Baue ein Level, in dem du bereits gewisse Wände vorgibst, die der Turtle im Weg stehen. Damit wird das Spiel etwas schwieriger.

6*. Neben den schwarzen Blöcken könntest du noch graue Blöcke einführen, die der Turtle ebenfalls im Weg stehen. Im Unterschied zu den schwarzen Blöcken lassen sich die grauen Blöcke aber nicht mehr entfernen.

7*. Die Turtle könnte beim Gehen eine gelbe Spur hinterlassen und sich so weigern, ein zweites Mal auf ein Feld zu gehen.

8*. Mach das Spiel schwieriger, indem sich die Turtle zufällig nach links oder rechts abdreht. Du kannst die Turtle auch schneller machen, indem du den Wert in *delay()* veränderst. Sie sollte aber nicht zu schnell sein, weil sonst das Erzeugen und Entfernen der Blocks nicht mehr richtig funktioniert.

9*. (Für Profis) Programmiere das Spielfeld so, dass wenn die Turtle rechts hinausläuft, sie von links wieder hineinkommt. Natürlich funktioniert das dann auch in die entgegengesetzte Richtung und genauso für oben/unten.

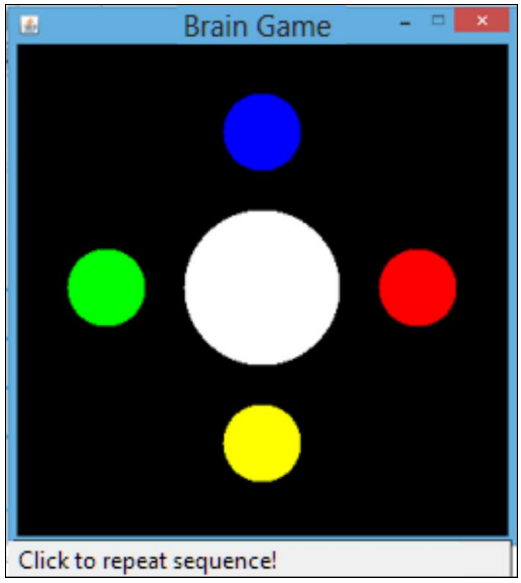
ARBEITSBLATT 3: BRAIN GAME

SPIELBESCHREIBUNG

Das Spielfeld besteht aus einer Lampe, welche in den Farben rot, gelb, grün und blau leuchten kann, und im ausgeschalteten Zustand weiss ist. Es gibt zudem 4 kreisförmige Schaltflächen (Buttons), mit den Farben rot, gelb, grün und blau.

Die Lampe zeigt immer längere Sequenzen mit zufälligen Farben: Zuerst mit einer Farbe, dann mit zwei Farben, usw. Der Spieler muss nach jeder gezeigten Sequenz durch Drücken der Buttons die gezeigte Sequenz wiederholen. Macht er dies richtig, so wird die Länge der Sequenz um 1 erhöht, macht er einen Fehler, so wird das Spiel beendet. Ziel ist es, eine möglichst lange Sequenz zu erreichen.

Das Problem wird durch schrittweisen Ausbau gelöst. Es werden also immer mehr Anforderungen erfüllt.



PHASE 1: DARSTELLUNG DES SPIELFELDES

Vorgehen: Verwende das folgende Programmgerüst.

```
# Ex1.py

from gturtle import *

def showLamp(col):
        pass

def showButton(number):
        pass

def setup():
        pass

# ------------------ main ------------------
Options.setPlaygroundSize(400, 400)
makeTurtle()
ht()
penUp()
clear("black")
setTitle("Brain Game")
addStatusBar(30)
setup()
```

Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

Schreibe die Funktionen gemäss folgender Dokumentation:

Funktion *showLamp(col)*
Zeichnet farbigen Kreis mit Durchmesser 100 in Spielfeldmitte:

col	Farbe
0	rot
1	gelb
2	grün
3	blau
sonst	weiss

Funktion *showButton(number)*
Zeichnet farbigen Kreis mit Durchmesser 50:

col	Farbe	Position
0	rot	rechts
1	gelb	unten
2	grün	links
3	blau	oben
sonst	nichts	

Funktion *setup()*
Erstellt Spielfeld in Ausgangssituation.

PHASE 2: DARSTELLUNG DES SPIELFELDES

Vorgehen:
Füge im Hauptteil die folgenden Zeilen ein, die eine Liste *seq* mit n zufälligen Zahlen im Bereich 0..3 erstellt und die Sequenz mit der Funktion *showSequence()* darstellt.

```
setStatusText("Showing sequence with length: " + str(n) + "...")
seq = []
for i in range(n):
    seq.append(random.randint(0, 3))
showSequence()
```

PHASE 3: MIT DER MAUS DIE LAMPE EIN-/AUSSCHALTEN

Nach der Anzeige der Sequenz mit 3 Farben, soll der Benutzer durch Drücken auf die Farbbuttons die Lampe mit derselben Farbe einschalten können. Beim Loslassen erlischt die Lampe wieder.

Vorgehen:
Verwende die Callbacks: *pressed()* und *released()*, die du mit den Annotationen registrierst. Verwende dazu eine Variable *buttonIndex* im Bereich 0..3 für Klicks auf Buttons und -1 und -2 für Klicks auf weisse und schwarze Flächen.

```
@onMousePressed
def pressed(x,y):
    moveTo(x, y)
    if getPixelColorStr() == "red":
        buttonIndex = 0
    ...
```

```
@onMouseReleased
def released(x,y):
    showLamp(-1)
```

PHASE 4: DIE SEQUENZ ÜBERPRÜFEN

Nach der Anzeige der Sequenz mit 3 Farben beginnt der Benutzer mit der Wiederholung der Farbsequenz durch Drücken der Buttons. Macht er einen Fehler, so bricht das Spiel sofort mit einer IsOver-Meldung ab, macht er es richtig, so wird eine Erfolgsmeldung ausgeschrieben.

Vorgehen:
Der Test auf die richtige Sequenz erfolgt bei jedem Loslassen der Maus, also im Callback *release()*. Dazu muss man mit der Variablen *clickCount* die bereits gemachten Mausklicks zählen und mit

```
if seq[clickCount] == buttonIndex:
```

überprüfen, ob der richtige Klick gemacht wurde. Ist dies der Fall, so erhöht man *clickCount*, andernfalls setzt man ein Flag *isOk = False*. Hat man die ganze Sequenz erfolgreich getestet, so setzt man *isOk = True*. Damit das Hauptprogramm informiert wird, wenn der Test zu Ende ist, verwendet man ein Flag *isUserActive* und fügt eine Warteschleife ein, die solange läuft, bis die Benutzeraktion beendet ist, also:

```
clickCount = 0
isUserActive = True
isOk = False
while isUserActive:
    delay(10)
if isOk:
    setStatusText("Sequence confirmed")
else:
    setStatusText("Sequence failed")
```

Vergiss nicht, im Callback *released()* die Variablen *clickCount, isUserActive* und *isOk* global zu machen.

PHASE 5: DIE SEQUENZ VERLÄNGERN

Bisher hast du mit einer festen Sequenzlänge von 3 gespielt. Jetzt musst du noch gemäss der Spielvorgabe die Sequenzlänge von 1 an erhöhen, falls der Benutzer sie richtig erraten hat. Dies tust du solange, bis er einen Fehler macht.

Du hast das Programm so gut vorbereitet, dass diese Erweiterung einfach zu implementieren ist. Nach dem *setup()* und der Initialisierung von n = 1 fügst du den Rest des Programms in eine endlose while-Schleife und erhöhst darin n, wenn der Benutzer erfolgreich ist. Macht er einen Fehler, so brichst du die Schleife mit *break* ab.

PHASE 6: DAS SPIEL ROBUST MACHEN

Mit der Phase 5 kann man das Spiel bereits spielen, wenn sich der Benutzer an bestimmte Regeln hält und nicht in einem falschen Moment mit der Maus klickt. Darum musst du das Programm noch so verbessern, dass ein Mausklick zum falschen Zeitpunkt zu keiner Katastrophe führt. Dazu führst du ein weiteres Flag *isMouseEnabled* ein, dass nur dann auf *True* steht, wenn der Benutzer die Maus verwenden darf. Ist *isMouseEnabled False*, so kehrst du einfach in den Callbacks ohne weitere Aktion zurück.

```
def pressed(x,y):
    if not isMouseEnabled:
        return
    ...


def released(x,y):
    if not isMouseEnabled:
        return
    ...
```

VERBESSERUNGEN, ERGÄNZUNGEN

Du kannst das Spiel noch nach deinen Ideen erweitern, beispielsweise:

- Wiederholtes Spielen ohne das Programm neu zu starten
- Veränderung des Schwierigkeitsgrades (schnelleres Anzeigen der Farbsequenz, Testen, ob man die Sequenz innerhalb einer bestimmten Zeit bestätigt hat, usw.) Das Einstellen des Schwierigkeitsgrades könnte vor Spielbeginn durch Mausklicks auf den weissen Kreis erfolgen
- Statt einer Farbsequenz kannst du eine Tonsequenz abspielen, die man durch Klicken auf die 4 Buttons wiedergeben muss
- Mehr als 4 Farben/Töne verwenden
- Statt eine Farbsequenz kannst du eine Zahlsequenz mit den Zahlen 1, 2, 3 und 4 anzeigen
- Statt Farben kannst du kleine Bilder verwenden, die mit *drawImage("sprites/bildname")* angezeigt werden
- Sehr einfache Sequenzen (z. B. rot, rot, rot, rot) ausschliessen.

ÜBER DIE AUTOREN

Jarka Arnold war als Dozentin an der Pädagogischen Hochschule Bern für die Informatikausbildung angehender Lehrkräfte für die Sekundarstufe 1 tätig. Sie hat dabei Informatikgrundkonzepte und das Programmieren mit Java, PHP und Python vermittelt. Ihre langjährige Erfahrung in der Aus-und Weiterbildung von Informatiklehrpersonen und viele Musterbeispiele sind in diesen Lehrgang eingeflossen. Sie ist zudem verantwortlich für den Webauftritt dieses Lehrgangs.

Aegidius Plüss war an der Universität Bern Professor für Informatik und deren Didaktik und hat in dieser Tätigkeit viele Informatiklehrkräfte aus- und weitergebildet, die heute aktiv an den Schulen tätig sind. Auf der Grundlage seiner grossen Erfahrungen mit vielen Programmiersprachen, Computersystemen und Elektronik hat er die didaktischen Bibliotheken und die notwendigen Tools für diesen Lehrgang entwickelt.

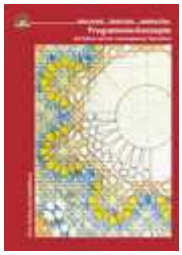
KONTAKT

Die Entwicklergruppe von TigerJython4Kids ist dankbar für jede Art von Rückmeldungen, insbesondere für Fehlermeldungen und Richtigstellungen, Anregungen und Kritik. Wir bieten auch Hilfe und Beratung bei fachlichen oder didaktischen Fragen zu Python und den in TigerJython integrierten Libraries, sowie zur Robotik-Hardware.

Schreiben Sie ein Email an:

help@tigerjython.ch

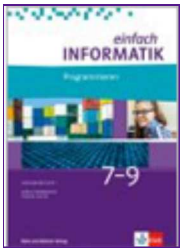
LINKS



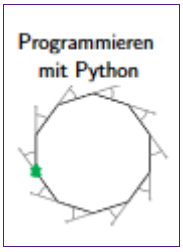
J. Arnold, T. Kohn, A. Plüss: **Programmierkonzepte mit Python**
Umfangreiches Online-Lehrmittel, mit vielen Beispielen aus verschiedenen Gebieten, geeignet für den Einsatz in weiterführenden Schulen und zum Selbststudium.
www.tigerjython.ch

Jarka Arnold, Aegidius Plüss: **Grafik, Robotik und Spiele mit Python**
Online-Lernprogramm mit vielen lauffähigen Programmbeispielen und Aufgaben für den Einsatz im Unterricht auf der Sekundarstufe 1 und 2.
www.jython.ch

Jarka Arnold, Aegidius Plüss: **Python-Online**
Turtlegrafik und Robotik mit Online-Editor
www.python-online.ch



Juraj Hromkovič, Tobias Kohn: **Einfach Informatik 7-9**
Lehrplan-21-kompatibles Lehrmittel für den Informatikunterricht
Verwendet Python und die Entwicklungsumgebung TigerJython
Klett und Balmer Verlag



Urs Hauser, Juraj Hromkovič, Tobias Kohn, Dennis Komm, Giovanni Serafini: **Programmieren mit Python**
Unterrichtsunterlagen für die Programmierung mit TigerJython
http://www.abz.inf.ethz.ch/../TigerJython.pdf

Tobias Kohn: **Python**
Eine Einführung in die Computer-Programmierung.
Ein Skript im PDF-Format.
http://jython.tobiaskohn.ch/PythonScript.pdf (166 Seiten)