Jarka Arnold, Tobias Kohn, Aegidius Plüss

PYTHON
exemplarisch

Eine Einführung ins Programmieren
mit der Lernumgebung TigerJython

www.python-exemplarisch.ch

Inhalt

Version 1.0, Oktober 2015

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

1. LERNUMGEBUNG

TIGERJYTHON4KIDS

Die Lernumgebung besteht aus einer einfachen Entwicklungsumgebung *TigerJython* und einem Online-Lernprogramm mit vielen lauffähigen Musterbeispielen und Aufgaben zum Üben. *TigerJython4Kids* eignet sich hervorragend für das Erlernen der wichtigsten Programmierkonzepte und kann bereits in der Volksschule eingesetzt werden. Ein weiterführendes Lernprogramm, das nicht nur für Programmiereinsteiger bestimmt ist, findet man unter www.tigerjython.ch.

TIGERJYTHON IDE

Die Distribution vom *TigerJython* umfasst eine einzige JAR-Datei, die kostenlos heruntergeladen werden kann. Sie enthält alle für die Programmierung notwendigen Komponenten, ausser dem JRE (Java Runtime Environment).

TigerJython funktioniert problemlos unter Windows, Mac und Linux.

INSTALLATION

Lade die Datei *tigerjython2.jar* herunter und speichere sie in einem beliebigen Verzeichnis auf der Festplatte. Im gleichen Verzeichnis kannst du auch alle deine Programme speichern.

Download tigerjython2.jar

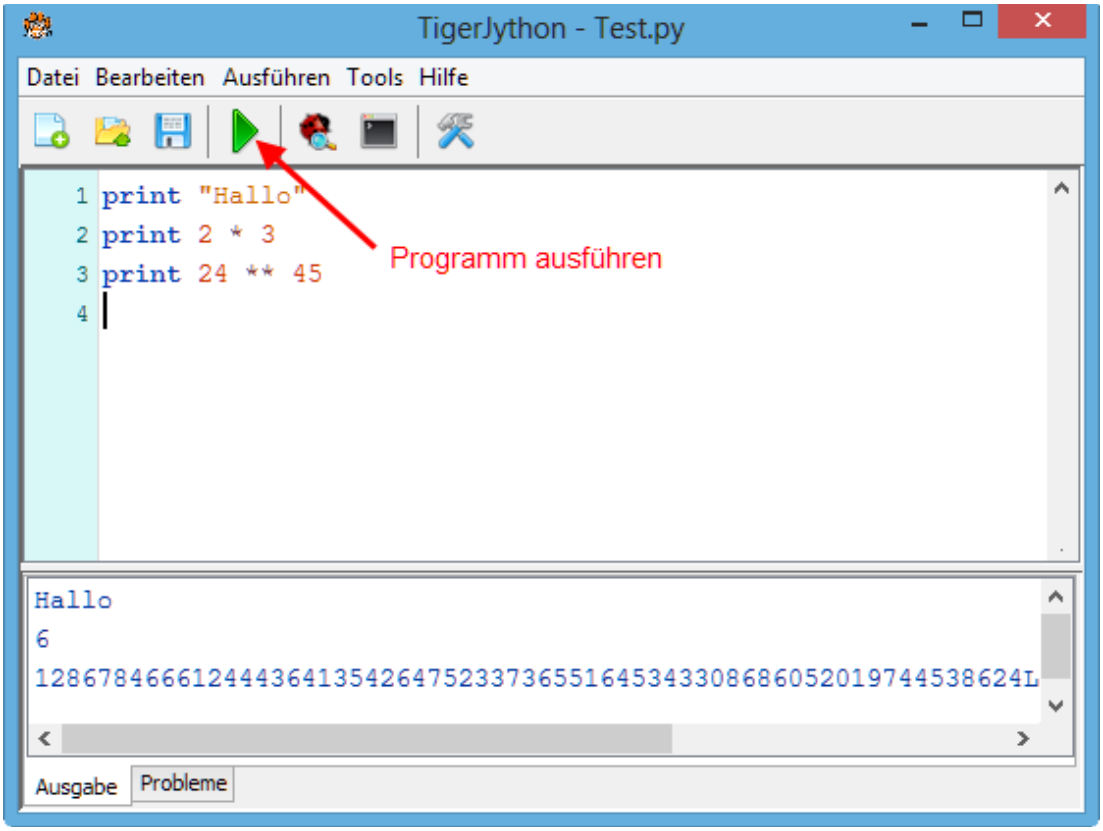
Du kannst einen Link (Verknüpfung) auf diese Datei erstellen, damit du diese vom Desktop starten kannst. Unter Linux musst du dieser Datei das Recht *Ausführen* (executable) geben. Falls du dem Link eine passende Desktop-Ikone zuordnen möchtest, so kannst du diese für Windows von hier und für Mac/Linux von hier downloaden.

ERSTE SCHRITTE

Starte den TigerJython-Editor mit Doppelklick auf ***tigerjython2.jar*** oder auf den Link zu dieser Datei.

ERSTE SCHRITTE

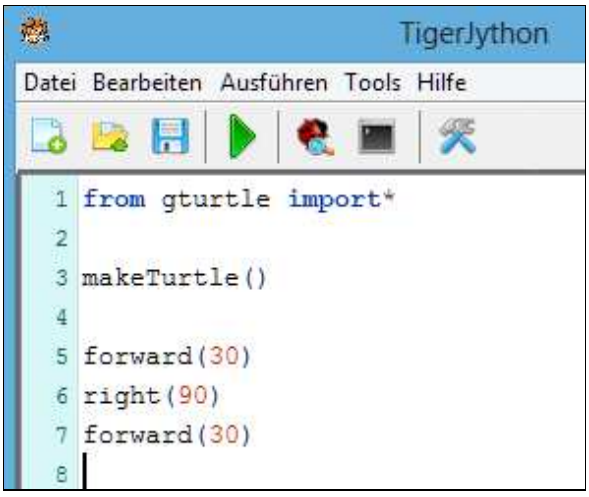
Starte den TigerJython-Editor mit Klick auf ***tigerjython2.jar*** oder auf den Link zu dieser Datei.



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*. Teste die Funktionalität, in dem du einige print-Befehle eingibst und auf die grüne Schaltfläche *Programm ausführen* klickst. Im Unterschied zu den meisten anderen Programmiersprachen rechnet Python mit beliebig langen Zahlen.

PROGRAMM EDITIEREN

Schreibe ein einfaches Turtlegrafik Programm.

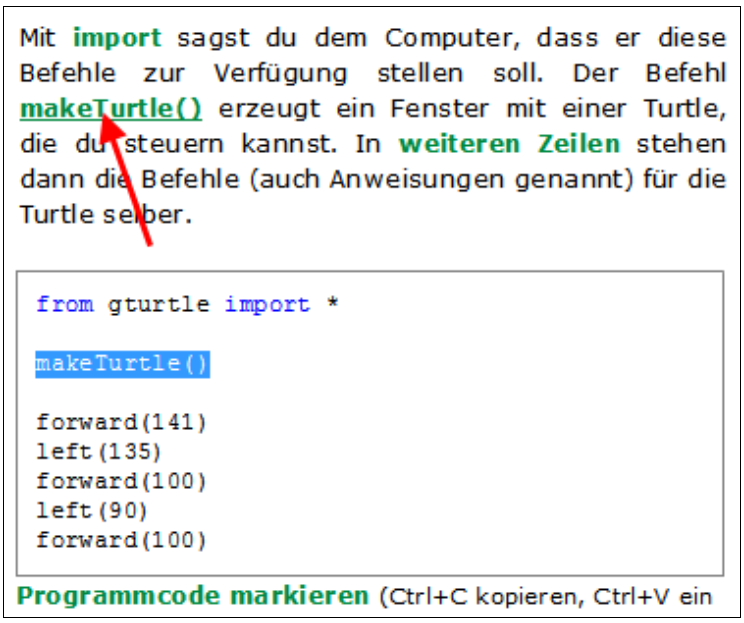


Beim Editieren kannst du die üblichen Tastenkürzel verwenden:

Ctrl+C	Kopieren
Ctrl+V	Einfügen
Ctrl+X	Ausschneiden
Ctrl+A	Alles markieren
Ctrl+Z	Rückgängig
Ctrl+S	Speichern
Ctrl+N	Neues Dokument
Ctrl+O	Öffnen
Ctrl+Y	Wiederholen
Ctrl+F	Suchen
Ctrl+H	Suchen und Ersetzen
Ctrl+Q	Markierte Zeilen auskommentieren Kommentarzeichen wegnehmen
Ctrl+D	Zeile löschen
Shift+ Cursor	Markieren

Die Programmbeispiele im Lehrgang sind so aufbereitet, dass du sie sehr einfach als Programmvorlagen verwenden kannst.

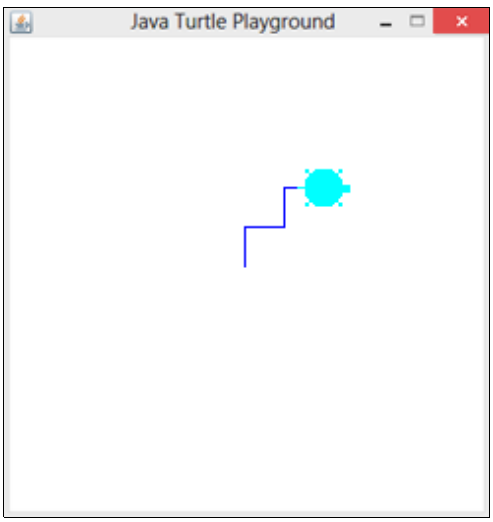
Mit Klick auf *Programmcode markieren* kannst du das ganze Programm auswählen. Mit der Maus kannst du aber auch nur einen Programmteil auswählen. Mit Ctrl+C kopierst du den markierten Programmcode in die Zwischenablage und mit Ctrl+V fügst du ihn in deinem TigerJython-Editorfenster ein.



Unser **Markierungstrick** hilft dir, die Erklärungen zum Programm zu verfolgen.

Mit Klick auf grün geschriebene Wörter wird die entsprechende Stelle im Programm markiert.

PROGRAMM AUSFÜHREN



Mit Klick auf den grünen Pfeil führst du das Programm aus.

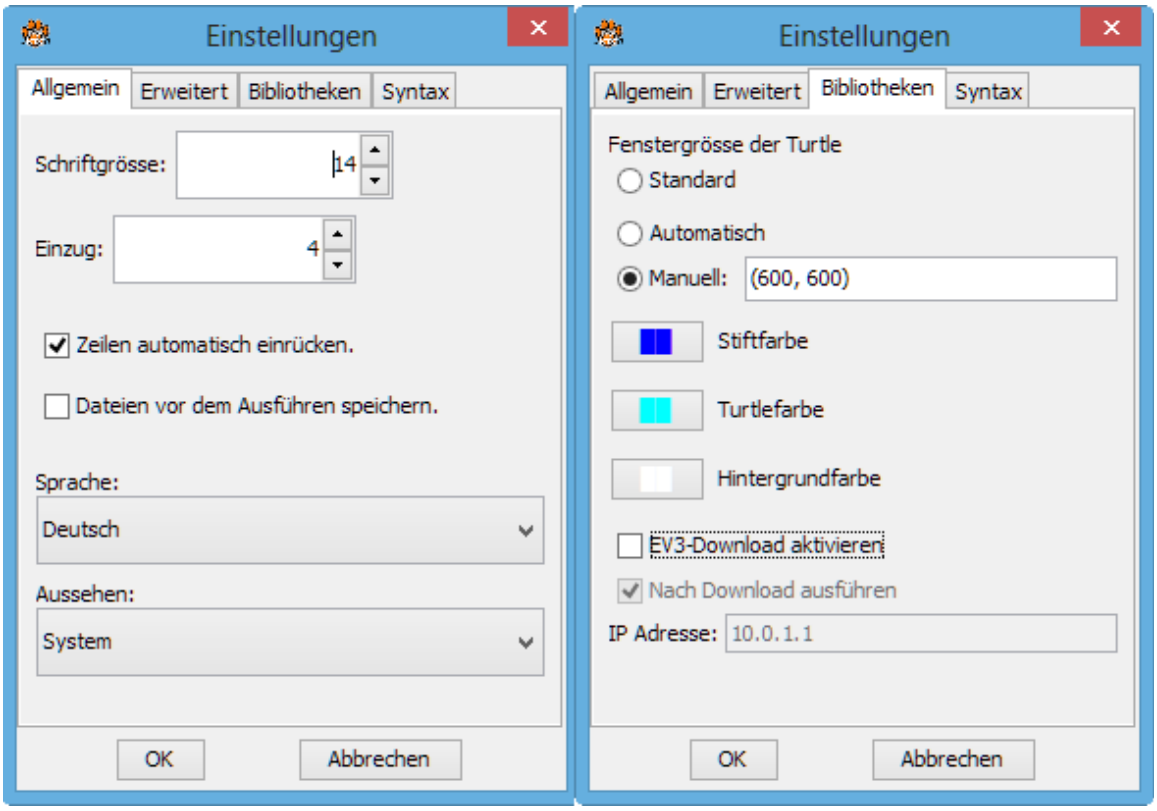
Die Grafik wird in einem neuen Grafikfenster angezeigt.

Falls das Programm nicht korrekt ist, erscheinen im Fenster *Probleme* die Fehlermeldungen.

EINSTELLUNGEN

Unter Einstellungen kannst du einige Anpassungen vornehmen:

- Grösse des Turtlefensters
  Optimale Grösse für Beispiele aus dem Kapitel Turtlegrafik: (600, 600)
- Hintergrundfarbe des Turtlefensters, Stift- und Turtlefarbe
- Schriftgrösse, Einzug und Schriftfarben des Editors
- Sprache (Deutsch, Englisch, Französisch)
- Zusätzliche Tools für EV3-Robotik aktivieren usw.



DOKUMENTATION

In TigerJython sind zusätzliche Module wie z. Bsp. Turtlegrafik integriert. Mit Klick auf *APLU Dokumentation* im Register *Hilfe* kannst du die Dokumentationen zu diesen Bibliotheken ansehen.

BEISPIELPROGRAMME

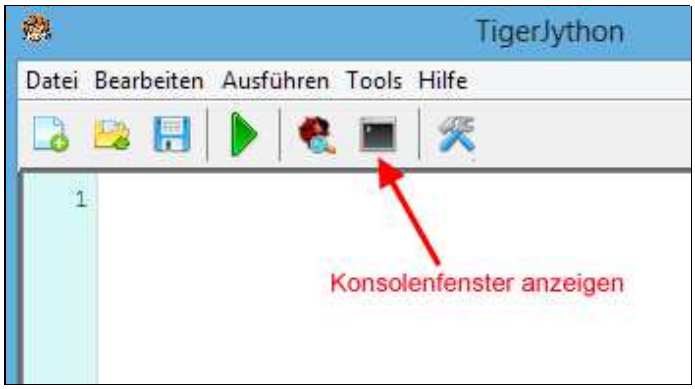
Wir schlagen dir vor, das Lehrmittel kapitelweise durchzuarbeiten und dabei die Beispielprogramme mit dem Link *Programmcode markieren* mit *Ctrl+C* und *Ctrl+V* einzeln

in den Tigejython-Editor zu übernehmen, unter einem geeigneten Namen zu speichern und dann auszuführen.

Du kannst aber auch **alle Programme** von hier **downloaden**.

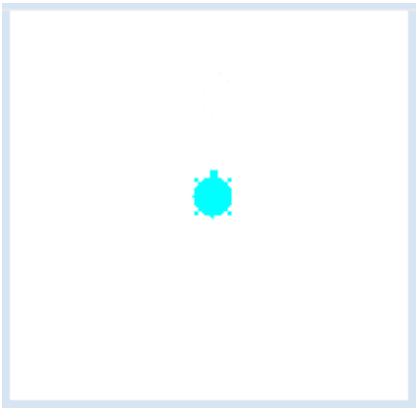
KONSOLENFENSTER

In Python kannst du einzelne Anweisungen auch sofort ausführen. Dieses Vorgehen eignet sich besonders gut, um etwas auszutesten. Dazu musst du das Konsolen-Symbol anklicken, wodurch das Konsolenfenster geöffnet wird.



Willst du beispielsweise einige Turtle-Befehle ausprobieren, so importierst du zuerst das Modul *gturtle* und erstellst mit dem Befehl *makeTurtle()* ein Turtlefenster mit einer Turtle.

```
>>> from gturtle import *
>>> makeTurtle()
```

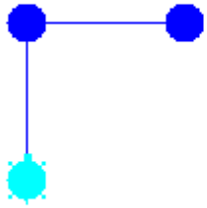


Danach stehen dir alle Befehle aus der Turtlegrafik direkt zur Verfügung. Beispielsweise:

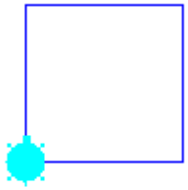
forward(100)	kurz: fd(100)	100 Schritte (Pixel) vorwärts bewegen
back(50)	kurz: bk(50)	50 Schritte rückwärts bewegen
left(90)	kurz: lt(90)	90° nach links drehen
right(90)	kurz: rt(90)	90° nach rechts drehen
clearScreen()	kurz: cs()	löscht alle Spuren und setzt die Turtle in die Mitte

Beispiel:

```
>>> fd(100)
>>> dot(20)
>>> rt(90)
>>> fd(100)
>>> dot(20)
>>> home()
```



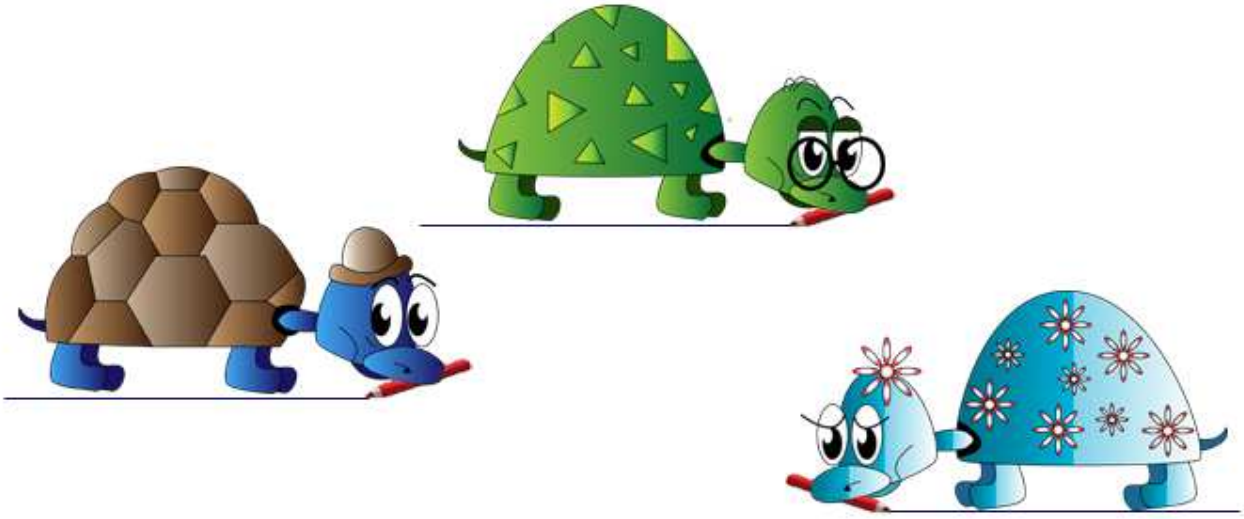
Mit dem Schlüsselwort *repeat* kannst du eine oder mehrere Anweisungen wiederholt ausführen. Wenn du mehrere Befehle als Befehlsblock wiederholen willst, so musst du sie gleich weit einrücken.



```
>>> repeat 4:
        fd(100)
        rt(90)
```

Es macht dir sicher Spass, weitere Turtlebefehle auszuprobieren. Eine Übersicht der Befehle findest du im Kapitel Turtlegrafik unter Dokumentation. In diesem Kapitel lernst du auch systematisch, wie du ganze Programme schreiben kannst.

TURTLEGRAFIK



Lernziele

- Du kannst ein einfaches Programm schreiben und mit der Turtle Figuren auf den Bildschirm zeichnen.
- Du kannst die Farbe der Linien und Flächen und die Linienbreite einstellen.
- Du weisst, wie die Turtle Anweisungen mehrmals wiederholen kann.
- Du weisst, wie man Programmteile nur unter bestimmten Bedingungen ausführen kann.
- Du kannst eigene Befehle mit Parametern definieren.
- Du kennst die Bedeutung von Variablen und kannst diese in deinen Programmen verwenden.
- Du weisst, was Rekursionen sind und kannst einfache rekursive Programme schreiben.
- Du kannst Turtle-Objekte erzeugen und mehrere Turtles gleichzeitig verwenden.

2.1 TURTLE BEWEGEN

EINFÜHRUNG

Programmieren heisst, einer Maschine Befehle zu erteilen und sie damit zu steuern. Die erste solche Maschine, die du steuerst, ist eine kleine Schildkröte auf dem Bildschirm: Die Turtle. Was kann diese Turtle und was musst du wissen, um sie zu steuern?

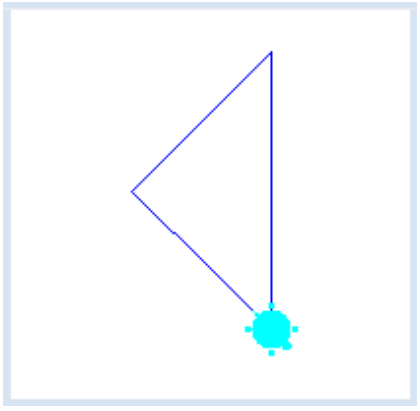
Turtlebefehle werden grundsätzlich Englisch geschrieben und enden immer mit einem Klammerpaar. Dieses enthält weitere Angaben zum Befehl. Selbst wenn keine weiteren Angaben nötig sind, muss ein leeres Klammerpaar vorhanden sein. Die Klein/Grossschreibung muss exakt eingehalten werden.

Die Turtle kann sich innerhalb ihres Fensters bewegen und dabei eine Spur zeichnen. Bevor die Turtle aber loslegt, musst du den Computer anweisen, dir eine solche Turtle zu erzeugen. Das machst du mit dem Befehl *makeTurtle().* Um die Turtle zu bewegen verwendest du die drei Befehle *forward(distanz)*, *left(winkel)* und *right (winkel)*.

PROGRAMMIERKONZEPTE: *Quellprogramm editieren, Programm ausführen, Programmsequenz*

DEIN ERSTES PROGRAMM

So sieht dein erstes Programm mit der Turtle aus. Klicke auf *Programmcode markieren*, kopiere und füge es in den TigerJython-Editor ein. Führe es aus, indem du auf den grünen Start-Knopf klickst. Die Turtle zeichnet ein rechtwinkliges Dreieck.



Die Turtlebefehle sind in einer Datei (einem sogenannten Modul) »gturtle« gespeichert.

Mit import sagst du dem Computer, dass er diese Befehle zur Verfügung stellen soll. Der Befehl **makeTurtle()** erzeugt ein Fenster mit einer Turtle, die du steuern kannst. In weiteren Zeilen stehen dann die Befehle (auch Anweisungen genannt) für die Turtle selber.

```
from gturtle import *

makeTurtle()

forward(141)
left(135)
forward(100)
left(90)
forward(100)
```

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

MEMO

Am Anfang jedes Turtle-Programms musst du zuerst das Turtle-Modul laden und eine neue Turtle erzeugen:

```
from gturtle import *
makeTurtle()
```

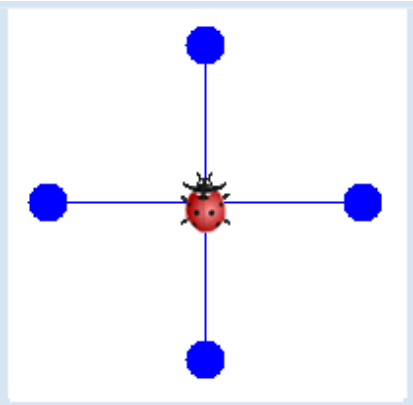
Danach kannst du der Turtle beliebig viele Befehle geben. Die drei Befehle, die die Turtle sicher versteht sind:

forward(s)	s (in pixel) vorwärts bewegen.
left(w)	Um den Winkel w (in Grad) nach links drehen.
right(w)	Um den Winkel w nach rechts drehen.

DEINE PERSÖNLICHE TURTLE

Du kannst bei makeTurtle() eine Datei angeben, welche als Turtlebild verwendet wird und so deinem Programm eine persönliche Note geben. Hier verwendest du die Datei *beetle.gif* aus dem Verzeichnis *sprites* der Tigerjython-Distribution. Beachte, dass du den Dateinamen in Anführungszeichen setzen musst.

Die Turtle zeichnet ein Kreuz mit gefüllten Kreisen in den Endpunkten.



```
from gturtle import *

makeTurtle("sprites/beetle.gif")

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)

forward(100)
dot(20)
back(100)
right(90)
```

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

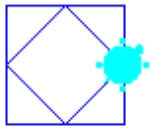
MEMO

Willst du ein anderes Symbol für die Turtle verwenden wie in obigem Beispiel, so musst du mit irgendeinem Bildeditor ein Bild erstellen. Übliche Turtlebilder haben eine Grösse von 32x32 Pixel auf einem transparenten Hintergrund im gif oder png-Format. Die Bilddatei sollte sich im Unterverzeichnis *sprites* des Verzeichnisses gespeichert sein, in dem sich dein Programm befindet [mehr...]

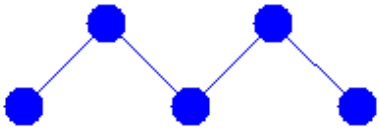
Im Programm verwendest du den neuen Befehl back(), mit dem sich die Turtle rückwärts bewegt, sowie dot(), mit dem die Turtle einen gefüllten Kreis zeichnet, dessen Radius (in Pixel) du angeben kannst.

AUFGABEN

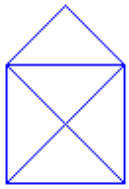
1. Zeichne mit der Turtle zwei Quadrate ineinander.



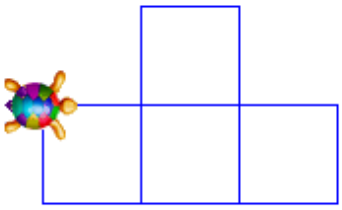
2. Verwende zusätzlich den Befehl dot(), um folgende Figur zu zeichnen:



3. Das »Haus vom Nikolaus« ist ein Zeichenspiel für Kinder. Ziel ist es, das besagte Haus in einem Linienzug aus genau 8 Strecken zu zeichnen, ohne dabei eine Strecke zweimal zu durchlaufen. Zeichne das Haus vom Nikolaus mithilfe der Turtle.



4*. Erstelle mit einem Bildeditor ein eigenes Turtlebild und zeichne damit nebenstehendes Bild. Die Seitenlänge der Quadrate ist 100. Es ist gleichgültig, wo die Turtle mit der Zeichnung beginnt/endet.



2.2 FARBEN VERWENDEN

EINFÜHRUNG

Um ihre Spur zu zeichnen, hat die Turtle einen Farbstift (engl. pen). Für diesen Farbstift kennt die Turtle weitere Anweisungen. Solange der Farbstift »unten« ist, zeichnet die Turtle eine Spur. Mit *penUp()* nimmt sie den Farbstift nach oben und bewegt sich nun, ohne eine Spur zu zeichnen. Mit *penDown()* wird der Farbstift wieder nach unten auf die Zeichenfläche gebracht, so dass eine Spur gezeichnet wird.

Über die Anweisung *setPenColor(farbe)* kannst du die Farbe des Stifts auswählen. Wichtig ist, dass du den Farbnamen in Gänsefüsschen setzt. Wie immer beim Programmieren kennt die Turtle nur englische Farbnamen. Die folgende Liste ist zwar nicht vollständig, aber doch ein erster Anhaltspunkt: *yellow, gold, orange, red, maroon, violet, magenta, purple, navy, blue, skyblue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white.*

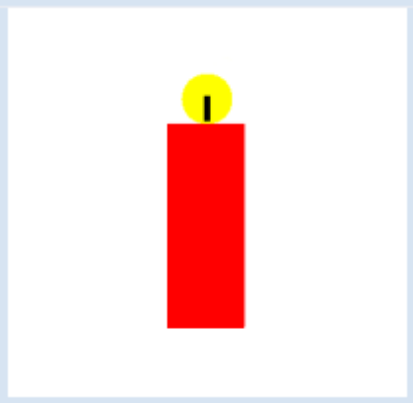
PROGRAMMIERKONZEPTE: *Programmgesteuertes Zeichnen*

FARBE UND BREITE DES STIFTS

Mit einer breiten roten Linie zeichnet die Turtle eine Kerze. Die Linienbreite in Pixel kannst du mit dem Befehl setLineWidth() einstellen.

Die gelbe Flamme kannst du mit dem Befehle dot() zeichnen. Dazwischen ist ein Programmteil, in dem sich die Turtle zwar bewegt, aber keine Linie zeichnet, weil der Stift mit penUp() gehoben wurde. Nach penDown() zeichnet die Turtle wieder.

hideTurtle() macht die Turtle unsichtbar.



```
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()
```

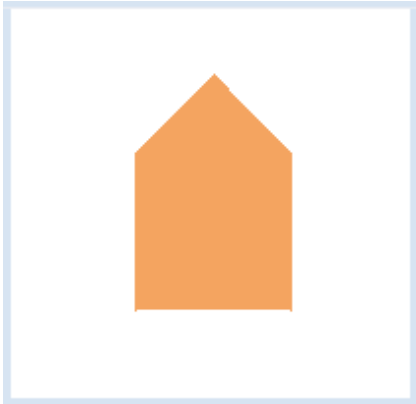
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Der Zeichenstift (pen) der Turtle kann mit setPenColor(farbe) die Farbe wechseln. Bei penUp() hört die Turtle auf, etwas zu zeichnen, bei penDown() zeichnet sie wieder weiter. Die Breite der gezeichneten Linie kannst du mit der Anweisung setLineWidth(breite) steuern.

Die Turtle kennt die sogenannten ***X11-Farben***. Das sind einige dutzend Farbnamen, die du im Internet unter http://cng.seas.rochester.edu/CNG/docs/x11color.html finden kannst. Alle diese Farben kannst du mit setPenColor(farbe) wählen.

GEFÜLLTE FLÄCHEN



Du kannst mit der Turtle fast beliebige Flächen mit Farben füllen. Mit dem Befehl startPath() sagst du der Turtle, dass du die Absicht hast, eine Fläche zu füllen. Die Turtle merkt sich dabei die momentane Lage als Startpunkt eines Linienzugs. Du umfährst dann mit der Turtle die Fläche und befiehlst ihr zuletzt mit fillPath(), denn erreichten Endpunkt des Linienzugs mit dem Startpunkt zu verbinden und die so entstandene Fläche zu füllen. Die Füllfarbe kannst du mit setFillColor(farbe) einstellen.

Texte einer Zeile nach einer einleitenden Raute # sind Kommentare. Sie werden bei der Programmaus- führung nicht beachtet. Du kannst dir zum Beispiel notieren, unter welchen Programmnamen die Datei gespeichert ist oder erklärenden Text zum Programmcode schreiben.

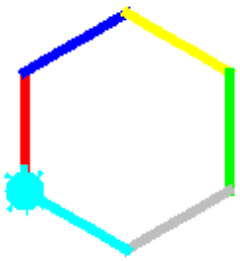
```
from gturtle import *

makeTurtle()

setPenColor("sandybrown")
setFillColor("sandybrown")
startPath()
forward(100)
right(45)
forward(72)
right(90)
forward(72)
right(45)
forward(100)
fillPath()
hideTurtle()
```

MEMO

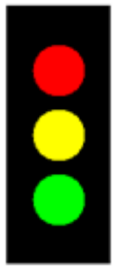
Wenn du eine Fläche, die mit einem Linienzug beschränkt ist, füllen willst, beginnst du das Zeichnen mit dem Befehl startPath(). Mit fillPath() wird der Startpunkt und der Endpunkt verbunden und die geschlossene Fläche gefüllt. Kommentare werden mit dem Zeichnen # eingeleitet.

AUFGABEN

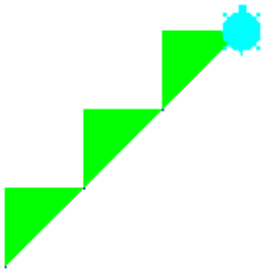
1. Zeichne mit der Turtle ein regelmässiges Sechseck und wähle für jede Seite eine andere Farbe.



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



3. Die Turtle soll das nebenstehende Bild zeichnen.



2.3 WIEDERHOLUNG

EINFÜHRUNG

Computer sind besonders gut darin, die gleichen Anweisungen (also auch Turtle-Befehle) immer wieder zu wiederholen. Um ein Quadrat zu zeichnen, musst du also nicht viermal die Befehle *forward(100)* und *left(90)* eingeben. Es genügt auch, der Turtle zu sagen, sie soll diese zwei Anweisungen viermal wiederholen.

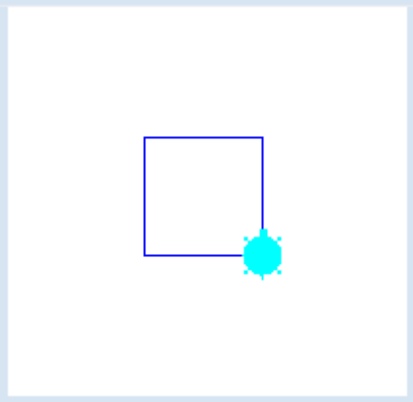
Mit der Anweisung *repeat* sagst du der Turtle, sie soll einige Befehle eine bestimmte Anzahl Mal wiederholen. Damit der Computer weiss, dass diese Befehle zusammengehören (einen Programmblock bilden), müssen diese gleich weit eingerückt sein. Wir verwenden für Einrückungen grundsätzlich drei Leerschläge.

PROGRAMMIERKONZEPTE: *Einfache Wiederholstruktur statt Codeduplikation*

REPEAT - STRUKTUR

Um ein Quadrat zu zeichnen, muss die Turtle vier mal geradeaus gehen und sich je um 90° drehen. Würdest du das alles untereinander schreiben, dann würde das Programm ziemlich lange.

Mit der Anweisung repeat 4: sagst du der Turtle, sie soll die eingerückten Zeilen viermal wiederholen. Achtung: Vergiss den Doppelpunkt nicht!



```
from gturtle import *

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

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

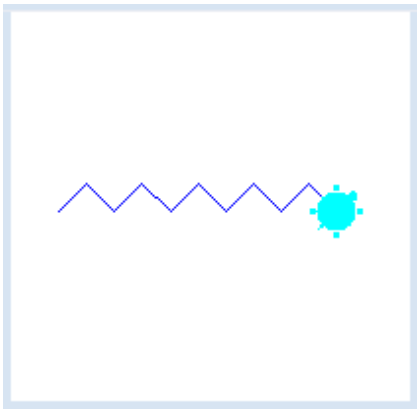
MEMO

Mit *repeat Anzahl:* sagst du dem Computer, er soll eine oder mehrere Anweisungen »Anzahl« Mal wiederholen, bevor er weitere Anweisungen ausführt. Alles, was wiederholt werden soll, muss unter repeat stehen und eingerückt sein.

```
repeat anzahl:
    Anweisungen, die
    wiederholt werden
    sollen
```

TÖNE WIEDERHOLEN

Ein typisches Beispiel für eine Wiederholung ist das Dah-Dih-Dah-Dih eines Feuerwehr-Autos. Mit der Turtle kannst du eine solche Tonfolge leicht erzeugen und gleichzeitig spasseshalber die Turtle eine Zickzack-Kurve zeichnen lassen. Einen reinen Ton erzeugst du mit playTone(), wobei du die Tönhöhe als Frequenzangabe (in Hertz) und die Dauer des Tons (in Millisekunden) angibst.



```
from gturtle import *

makeTurtle()

setPos(-200, 0)
right(45)

repeat 5:
    playTone(392, 400)
    forward(50)
    right(90)
    playTone(523, 400)
    forward(50)
    left(90)
```

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

MEMO

Mit setPos(x, y) kannst du die Turtle auf eine bestimmte Position im Turtlefenster setzen, ohne dass sie dabei eine Spur zeichnet. Die beiden Zahlen x und y sind die Koordinaten, wobei der Nullpunkt in der Mitte des Fensters liegt. (Der Koordinatenbereich hängt von der Grösse des Fensters ab.)

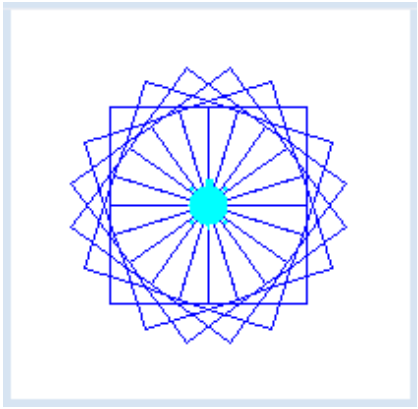
Du kannst bei playTone() die Tonhöhen auch mit einem Buchstaben gemäss der Notenskala angeben, also mit c, d, usw. oder in der eingestrichenen Oktave mit c', d', usw. (oder mit zwei oder drei Apostrophen). Du muss die Tonhöhen aber in Anführungszeichen setzen. Wenn du willst, kannst du auch noch eine Angabe des zu verwendenden Musikinstruments machen (vorhanden sind: piano, guitar, harp, trumpet, xylophone, organ, violin, panflute, bird, seashore). Versuch es mal mit

Unterer Ton: playTone("g'", 400, instrument = "trumpet")
Oberer Ton: playTone("c''", 400, instrument = "trumpet")

VERSCHACHTELTES REPEAT

Ein Quadrat gelingt einfach mit einer vierfachen Wiederholung. Nun sollst du 20 Quadrate zeichnen und die Quadrate etwas gegeneinander verdrehen. Dazu musst du zwei repeat-Anweisungen ineinander schachteln. Im inneren Programmblock zeichnet die Turtle ein Quadrat und dreht anschliessend um 18 Grad nach rechts. Die äussere repeat-Anweisung wiederholt dies 20 mal. Beachte dabei die korrekte Einrückung bei den Anweisungen, die wiederholt werden sollen.

Falls du die Turtle mit hideTurtle() versteckst, erfolgt das Zeichnen schneller.



```
from gturtle import *

makeTurtle()

# hideTurtle()
repeat 20:
    repeat 4:
        forward(80)
        left(90)
    right(18)
```

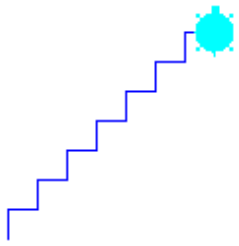
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

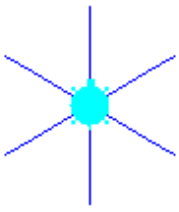
Die repeat-Befehle lassen sich verschachteln. Eine korrekte Einrückung bei den Anweisungen, die wiederholt werden sollen, ist wichtig.

AUFGABEN

1. Zeichne eine Treppe mit 7 Stufen.

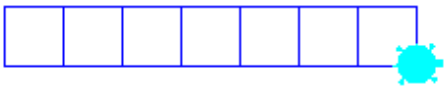


2. Zeichne einen Stern. Verwende dabei den Befehl *back()*.



3. Einen "richtigen" Stern kannst du mit den Drehwinkeln 140° und 80° zeichnen.

4. Zeichne folgende Figur mit zwei verschachtelten *repeat*-Anweisungen. Im inneren repeat-Block wird ein Quadrat gezeichnet.



5. Mit dem Befehl *leftArc(30, 180)* zeichnet die Turtle einen Linksbogen mit dem Radius 30 und Sektorwinkel *180°,* also einen Halbkreis.

Zeichne mit den Befehlen *leftArc(radius, angle)* und *rightArc(radius, angle)* die nebenstehende Wellen-Figur.

6. Zeichne mit Hilfe von zwei Viertelkreisen einen Vogel.

7..Zeichne 5 Treppenstufen und spiele dazu Töne mit den Frequenzen 264, 297, 330, 352 und 396 ab.

2.4 FUNKTIONEN

EINFÜHRUNG

In einem grösseren Bild kommen Figuren wie Dreiecke und Quadrate mehrmals vor. Die Turtle weiss aber nicht, was ein Dreieck oder ein Quadrat ist. Du musst also jedes Mal der Turtle mit einem vollständigen Programmcode erklären, wie sie die Figuren zeichnet. Geht das nicht auch einfacher?

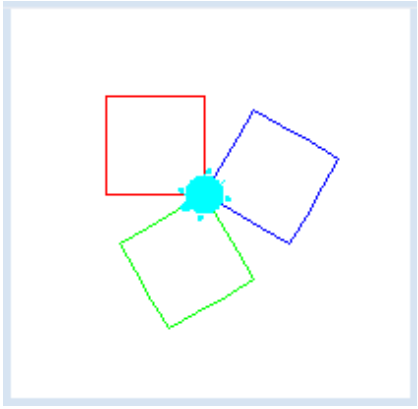
Es geht einfacher! Du kannst der Turtle nämlich neue Befehle beibringen, z.B.ein Quadrat oder Dreieck zu zeichnen, und musst ihr dann nur sagen, sie soll diesen Befehl ausführen, also ein Quadrat oder ein Dreieck zeichnen. Um einen neuen Befehl zu definieren, wählst du einen beliebigen Bezeichner, beispielsweide *square*, und schreibst *def square():* Danach notierst du alle Anweisungen, die zum neuen Befehl gehören. Damit der Computer weiss, was zum neuen Befehl gehört, müssen diese Anweisungen eingerückt sein.

PROGRAMMIERKONZEPTE: *Modulares Programmieren, Funktionsdefinition, Funktionsaufruf*

EIGENE BEFEHLE DEFINIEREN

In diesem Programm definierst du mit ***def*** den neuen Befehl square(). Die Turtle weiss danach, wie sie ein Quadrat zeichnen kann, sie hat aber noch keines gezeichnet.

Mit dem Befehl *square()* zeichnet die Turtle an jeder aktuellen Position ein Quadrat mit der Seitenlänge 100. In unserem Beispiel ist es ein rotes, ein blaues und ein grünes Quadrat.



```
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, Ctrl+V einfügen)

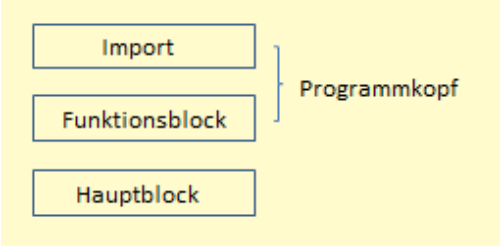
MEMO

Mit ***def bezeichner():*** definierst du einen neuen Befehl. Wähle einen Namen, der die Tätigkeit wiederspiegelt. Alle Anweisungen, die zum neuen Befehl gehören, müssen eingerückt sein.

```
def bezeichner():
    Anweisungen
```

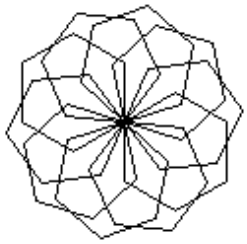
Vergiss die Klammern und den Doppelpunkt nach dem Bezeichner nicht! In Python nennt man neue Befehle auch *Funktionen*. Wenn du die Funktion *quadrat()* verwendest, sagt man auch, die Funktion werde "aufgerufen".

Wir gewöhnen uns daran, die Funktionsdefinitionen im Programmkopf anzuordnen, da diese vor ihrem Aufruf definiert sein müssen.

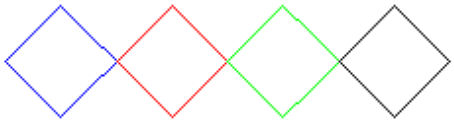


AUFGABEN

1. Definiere einen Befehl *hexagon()*, mit dem du ein Sechseck zeichnen kann. Verwende diesen Befehl, um die nebenstehende Figur zu zeichnen.



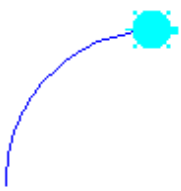
2a. Definiere einen Befehl für ein Quadrat, das auf der Spitze steht und zeichne damit die nebenstehende Figur.



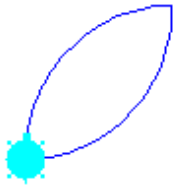
2b*. Du kannst gefüllte Quadrate zeichnen, indem du die Befehle *startPath()* und *fillPath()* verwendest.

3a. Du erlebst in dieser Aufgabe, wie du unter Verwendung von Funktionen eine Aufgabe schrittweise lösen kannst [mehr...].

Definiere eine Funktion *arc()*, mit dem die Turtle einen Bogen zeichnet und sich dabei insgesamt um 90 Grad nach rechts dreht. Mit *speed(-1)* kannst du die Turtlegeschwindigkeit auf den maximalen Wert setzen.

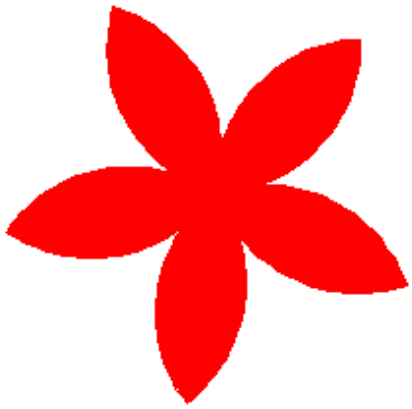


3b. Ergänze das Programm mit der Funktion *petal(),* welche zwei Bogen zeichnet. Die Turtle sollte am Ende aber wieder in Startrichtung stehen.

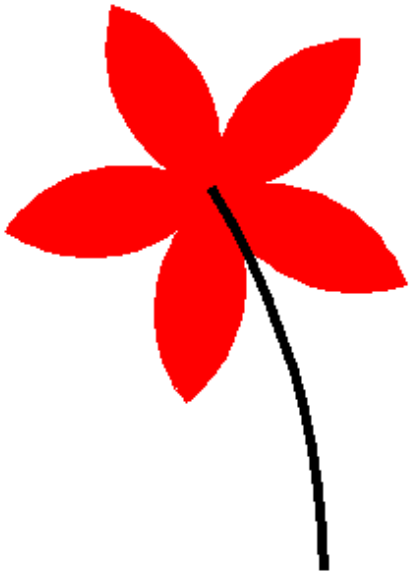


3c. Ergänze das Programm so, dass *petal()* ein rot gefülltes Blatt (ohne sichtbare Umrandungslinie) zeichnet.

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



3e*. Ergänze die Blume mit einem Stiel.



2.5 PARAMETER

EINFÜHRUNG

Beim Befehl *forward()* gibst du in Klammern an, um welche Strecke die Turtle vorwärts gehen soll. Dieser Wert in den Klammern gibt an, wie weit vorwärts gegangen wird. Er präzisiert den Befehl und heisst ein Parameter: Hier ist es eine Zahl, die bei jeder Verwendung von *forward()* anders sein kann. Im vorhergehenden Kapitel hast du einen eigenen Befehl *square()* definiert. Im Unterschied zu *forward()* ist die Seitenlänge dieses Quadrats aber immer 100 Pixel. Dabei wäre es doch in vielen Fällen praktisch, die Seitenlänge des Quadrats anpassen zu können. Wie geht das?

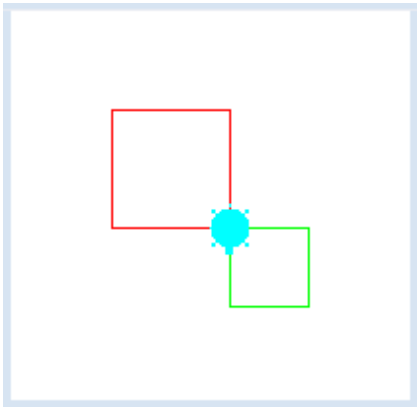
PROGRAMMIERKONZEPTE: *Parameter, Parameterübergabe*

BEFEHLE MIT PARAMETER

Auch in diesem Programm definieren wir ein Quadrat. An Stelle der leeren Parameterklammer bei der Definition der Funktion *square(),* setzen wir den Parameternamen *sidelength* ein und verwenden diesen beim Aufruf von forward(sidelength).

Du kannst dadurch *square* mehrmals verwenden und bei jeder Verwendung eine Zahl für *seite* angeben.

Mit square(80) zeichnet die Turtle ein Quadrat mit der Seitenlänge von 80 Pixeln, mit square(50) eines mit der Seitenlänge von 50 Pixeln.



```
from gturtle import *

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

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

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

MEMO

Parameter sind Platzhalter für Werte, die jedes Mal anders sein können. Du gibst den Parameter bei der Definition eines Befehls hinter den Befehlsnamen in einem Klammerpaar an.

```
def befehlsname(parameter):
    Anweisungen, die
    parameter verwenden
```

```
def befehlsname(parameter):
    Anweisungen, die
    parameter verwenden
```

Der Parametername ist frei wählbar, sollte aber seine Bedeutung wiederspiegeln. Bei der Verwendung des Befehls gibst du wieder in Klammern den Wert an, den der Parameter haben soll.

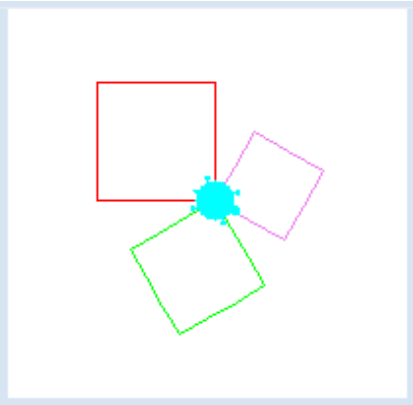
```
befehlsname(123)
```

Hier wird der Parameter im ganzen Befehl durch 123 ersetzt.

MEHRERE PARAMETER

Befehle können mehrere Parameter besitzen. Beim Quadrat kannst du zum Beispiel mit def square(sidelength, color) als Parameter seite und farbe wählen.

Du kannst dann *quadrat* viel flexibler verwenden.Mit square(100, "red") zeichnet die Turtle ein rotes Quadrat mit der Seitenlänge von 100 Pixeln, mit square(80, "green") ein grünes mit der Seitenlänge von 80 Pixeln.



```
from gturtle import *

def square(sidelength, color):
    setPenColor(color)
    repeat 4:
        forward(sidelength)
        left(90)

makeTurtle()
square(100, "red")
left(120)
square(80, "green")
left(120)
square(60, "violet")
```

MEMO

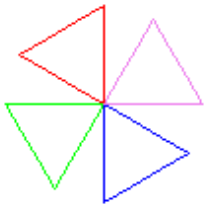
Befehle können mehrere Parameter besitzen. Diese werden in der Parameterklammer getrennt mit Komma eingegeben.

```
def befehlsname(parameter1, parameter2....):
   Anweisungen, die parameter1
   und parameter2 verwenden
```

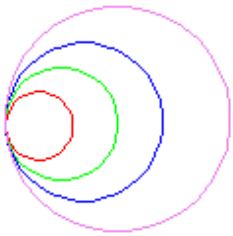
Die Reihenfolge der Parameter in der Parameterklammer bei der Definition des Befehls muss mit der Reihenfolge der Werte beim Aufruf des Befehls übereinstimmen.

AUFGABEN

1. Definiere einen Befehl *triangle(color)*, mit welchem die Turtle farbige Dreiecke zeichnen kann. Zeichne 4 Dreiecke in den Farben red, green, blue und violet



2. Definiere einen Befehl *colorCircle(radius, color)*, mit welchem die Turtle einen farbigen Kreis zeichnet. Du kannst dabei den Befehl *rightArc(radius, angle)* verwenden. Zeichne die nebenstehende Figur.



3. Das folgende Programm zeichnet leider 3 gleich grosse Fünfecke, aber nicht wie gewünscht verschieden grosse. Warum nicht? Korrigiere es.

```
from gturtle import *

def pentagon(sidelength, color):
   setPenColor(color)
   repeat 5:
      forward(90)
      left(72)

makeTurtle()
pentagon(100, "red")
left(120)
pentagon(80, "green")
left(120)
pentagon(60, "violet")
```

4. Du sagst der Turtle mit dem Befehl segment(), sich um eine bestimmte Strecke s vorwärts zu bewegen und sich um einen bestimmten Winkel w nach rechts zu drehen:

```
def segment(s, w):
   forward(s)
   right(w)
```

Schreibe ein Programm, das diesen Befehl 92 mal mit s = 300 und w = 151 ausführt. Mit *setPos(x, y)* kannst du die Turtle zu Beginn geeignet im Fenster positionieren.

5*. Die Turtle soll zwei, drei oder vier segment-Bewegungen ausführen. Schau dir die schönen Grafiken in folgenden Fällen an:

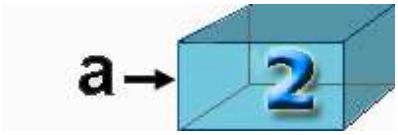
Anzahl Segmente	Werte	Anzahl Wiederholungen
2	forward(77) right(140.86) forward(310) right(112)	37
3	forward(15.4) right(140.86) forward(62) right(112) forwad(57.2) right(130)	46

2.6 VARIABLEN

EINFÜHRUNG

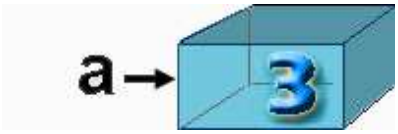
Im vorhergehenden Kapitel hast du Quadrate gezeichnet, deren Seitenlänge im Programm fest eingebaut waren. Manchmal möchtest du aber die Seitenlänge mit einem Eingabedialog einlesen. Dazu muss das Programm die eingegebene Zahl als **Variable** speichern. Du kannst eine Variable als einen Behälter (Container) auffassen, auf dessen Inhalt du mit einem Namen zugreifst. Kurz gesagt, hat eine Variable **einen Namen und einen Wert**. Den Namen der Variablen darfst du frei wählen. Nicht erlaubt sind Schüsselwörter und Namen mit Sonderzeichen.

Mit der Schreibweise a = 2 erstellst du den Behälter, auf den du mit dem Namen a zugreifst und legst die Zahl 2 hinein. In Zukunft sagen wir, dass du damit eine Variable a **definierst** und ihr einen Wert **zuweist**.



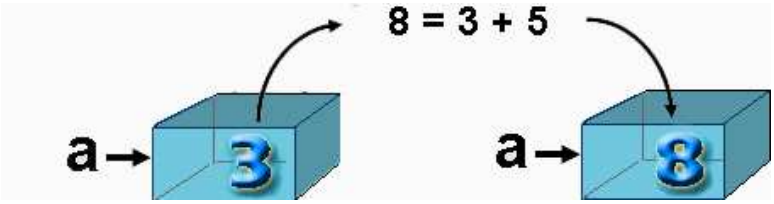
a = 2 : Variablendefinition (Zuweisung)

Du kannst in den Behälter nur **ein einziges** Objekt legen. Wenn du später unter dem Namen a die Zahl 3 speichern willst, so schreibst du a = 3 [mehr...].



a = 3 : neue Zuweisung

Was geschieht, wenn du nun **a = a + 5** schreibst? Du nimmst die Zahl, die sich gegenwärtig im Behälter befindet, auf den du mit a zugreifst, also die Zahl 3 und addierst dazu 5. Das Resultat 8 speicherst du wieder unter dem Namen a.

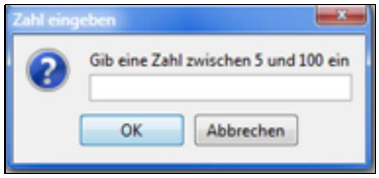


Das Gleichheitszeichen hat also in Computerprogrammen nicht dieselbe Bedeutung wie in der Mathematik. Es ist keine Gleichung, sondern eine Variablendefinition oder eine Zuweisung.

PROGRAMMIERKONZEPTE: *Variablendefinition, Zuweisung*

VARIABLENWERT EINLESEN UND VERÄNDERN

Im Programm kannst du mit Hilfe einer Dialogbox der Variablen x einen Wert zwischen 10 und 100 zuweisen. Diesen Wert veränderst du nachfolgend in der Wiederholstruktur und zeichnest dadurch eine Spirale.




```
from gturtle import *

makeTurtle()
```

```
repeat 10:
    forward(x)
    left(120)
    x = x + 20
```

MEMO

Mit *Variablen* kannst du Werte speichern, die du im Laufe des Programms lesen und verändern kannst. Jede Variable hat einen Namen und einen Wert. Mit dem Gleichheitszeichen definierst du eine Variable und weist ihr einen Wert zu [mehr...].

UNTERSCHEIDE ZWISCHEN VARIABLEN UND PARAMETER

Du solltest zwischen einer Variablen und einem Parameter unterscheiden. Parameter sind nur innerhalb einer Funktion gültig und transportieren Daten in eine Funktion, während Variablen überall möglich sind. Beim Aufruf erhält der Parameter einen Wert und kann im Innern der Funktion wie eine Variable verwendet werden. Um den Unterschied klar zu machen, verwendest du in deinem Programm in der Funktion *square()* den Parameter sidelength. Mit einem Eingabedialog liest du eine Zahl ein und speichert sie in der *Variablen* s. Beim Aufruf von square() übergibst du dem Parameter Seite den Variablenwert von s.


```
from gturtle import *

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

makeTurtle()
s = inputInt("Enter the side length")
square(s)
```

MEMO

Du musst zwischen der Variable *s* und dem Parameter *seite* unterscheiden. Parameter sind in der Funktionsdefinition Platzhalter und können beim Aufruf wie Variablen aufgefasst werden, die nur im Inneren der Funktion bekannt sind. Ruft man die Funktion mit einer Variablen auf, so wird der Variablenwert in der Funktion benutzt. *square(sidelength)* zeichnet somit ein Quadrat mit der Seitenlänge *sidelength*.

GLEICHE NAMEN FÜR VERSCHIEDENE DINGE

Wie du weisst, sollen Parameter- und Variablennamen ausdrücken, auf was sie sich beziehen. Sie sind aber eigentlich frei wählbar. Deshalb ist es üblich, bei gleichen Bezügen den gleichen Namen für Parameter und Variablen zu wählen.

Es ergeben sich dadurch im Programm keine Namenskonflikte, du musst aber zum Verständnis des Programms den Unterschied im Auge behalten.


```
from gturtle import *

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

makeTurtle()
sidelength = inputInt("Enter the side length")
square(sidelength)
```

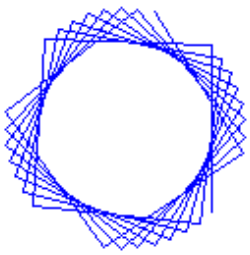
MEMO

Auch wenn du für gewisse Parameter und Variablen den gleichen Namen verwendest, solltest du Parameter und Variablen begrifflich auseinander halten.

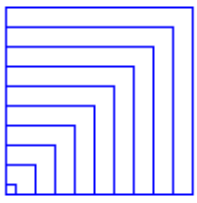
AUFGABEN

1. Nach Eingabe der Anzahl Ecken in einer Dialogbox soll die Turtle ein regelmässiges n-Eck zeichnen. Beispielsweise wird nach der Eingabe 8 ein 8-Eck gezeichnet. Den passenden Drehwinkel soll das Programm berechnen. Spiele dazu Turtle und überlege dir, wie weit du dich drehen musst, um die nächste Seite zu zeichnen. Erinnere dich an das Zeichnen eines gleichseitigen Dreiecks.

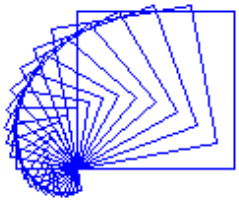
2. Nach der Eingabe eines Winkels in einer Dialogbox zeichnet die Turtle 30 Strecken der Länge 100, wobei sie nach jeder Strecke um den gegebenen Winkel nach links dreht. Experimentiere mit verschiedenen Winkeln und zeichne schöne Figuren. Mit *hideTurtle()* kannst du das Zeichnen beschleunigen.



3. Die Turtle soll 10 Quadrate zeichnen. Definiere zuerst einen Befehl *quadrat* mit dem Parameter *seite*. Die Seitenlänge des ersten Quadrats ist 8. Bei jedem nächsten Quadrat ist die Seitenlänge um 10 grösser.



4. Du kannst in einer Dialogbox die Seitenlänge des grössten Quadrats eingeben. Die Turtle zeichnet dann 20 Quadrate. Nach jedem Quadrat wird die Seitenlänge um den **Faktor** 0.9 kleiner und die Turtle dreht um den Winkel 10° nach links.



2.7 SELEKTION

EINFÜHRUNG

Was du im täglichen Leben unternimmst, hängt oft von gewissen Bedingungen ab. So entscheidest du dich je nach Wetter, wie du in die Schule fährst. Du sagst: "*Falls es regnet, fahre ich mit dem Tram, sonst mit dem Fahrrad*". Auch der Ablauf eines Programms kann von Bedingungen abhängig sein. Solche Programmverzweigungen auf Grund von bestimmten Bedingungen gehören zu den Grundstrukturen jeder Programmiersprache. Die Anweisungen nach *if* werden nur dann ausgeführt, wenn die Bedingung wahr ist, sonst werden die Anweisungen nach *else* ausgeführt.

PROGRAMMIERKONZEPTE: *Bedingung, Programmverzweigung, Selektion, If-Else-Struktur*

EINGABE ÜBERPRÜFEN

Nach der Eingabe der Seitenlänge in einer Dialogbox soll das Quadrat nur dann gezeichnet werden, wenn es im Fenster vollständig Platz hat.

Dazu prüfen wir den Wert von s. Wenn s kleiner als 300 ist, wird ein Quadrat mit der Seitenlänge s gezeichnet, sonst erscheint im unteren Teil des Tigerjython-Fensters eine Meldung. Diese Prüfung erfolgt in der Programmiersprache mit der *if*-Anweisung


```
from gturtle import *

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

makeTurtle()
s = inputInt("Enter the side length")
if s < 300:
    square(s)
else:
    print "The side length is too big"
```

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

MEMO

Die Anweisungen nach ***if*** werden nur dann ausgeführt, wenn die Bedingung wahr ist, sonst werden die Anweisungen nach ***else*** ausgeführt. Manchmal kommt es vor, das nur etwas gemacht wird, wenn die Bedingung erfüllt ist; sonst aber nichts passiert. Für diese Fälle kannst du den *else*-Block weglassen. Achte auf die Doppelpunkte nach der *if*-Bedingung und nach *else*, sowie auf die korrekte Einrückung der beiden Programmblöcke.

MEHRFACHE AUSWAHL

Wir wollen farbige Quadrate zeichnen. In einer Dialogbox kannst du die gewünschte Farbe mit einer Zahl eingegeben. In einer if-Struktur wird diese Zahl überprüft und die entsprechende Füllfarbe gesetzt. Wir testen zuerst auf den Wert 1, dann mit elif auf 2 und dann auf 3. Falls eine andere Zahl eingegeben wird, setzen wir mit else die Farbe auf schwarz.

Mit dem Befehl fill(10, 10) wird die geschlossene Fläche um den gegebene Punkt mit der Füllfarbe gefüllt. Da sich die Turtle nach dem Zeichnen des Quadrats wieder in der Fenstermitte (0, 0) befindet, wählen wir mit (10, 10) einen Punkt, der sicher im Inneren des Quadrats liegt.

```
from gturtle import *

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

makeTurtle()
n = inputInt("Enter a number: 1:red 2:green 3:yellow")
if n == 1:
    setFillColor("red")
elif n == 2:
    setFillColor("green")
elif n == 3:
    setFillColor("yellow")
else:
    setFillColor("black")

square()
fill(10, 10)
```

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

MEMO

Es können mehrere Bedingungen nacheinander überprüft werden. Falls die Bedingung bei *if* nicht erfüllt ist, wird die Bedingung bei *elif* überprüft. *elif* ist eine Abkürzung von *else if*. Falls keine der *elif*-Bedingungen erfüllt ist werden die Anweisungen nach *else* ausgeführt.

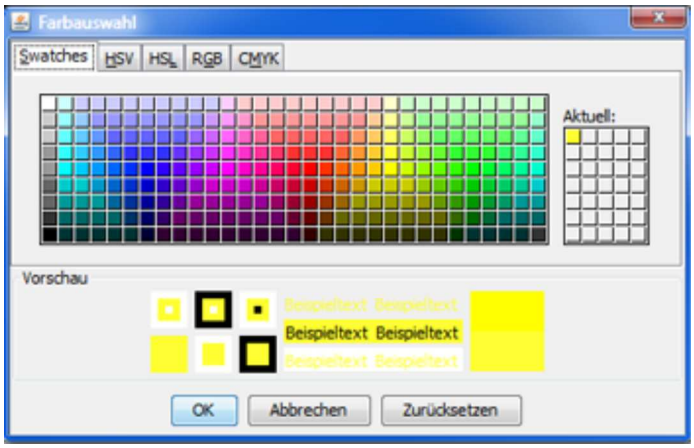
Es fällt dir sicher auf, dass das Prüfen auf Gleichheit in Python mit einem doppelten Gleichheitszeichen erfolgt. Dies ist etwas gewöhnungsbedürftig, aber nötig, da das einfache Gleichheitszeichen für Zuweisungen verwendet wird.

Beachte die Notation für Vergleichsoperatoren: >, >= , < , <= , == , != .

Mit dem Befehl *fill(x, y)* können geschlossene Figuren mit der Füllfarbe gefüllt werden. Der Punkt (x, y) muss sich aber im Inneren der Figur befinden.

FARBAUSWAHL, BOOLESCHE VARIABLEN

Das nachträgliche Füllen mit *fill()* setzt voraus, dass das Innere der Figur nicht bereits mit einer anderen Figur belegt ist. Du kennst von früher die startPath()/fillPath()-Kombination, mit der du auch neue Figuren, die über schon vorhandenen Figuren liegen, korrekt füllen kannst. Du verwendest im Programm durch Aufruf von askColor() ein elegantes Dialogfeld, mit welchem du die Farbe des Sterns auswählst.



Den Stern zeichnest du mit der Funktion star(), die neben der Grösse des Sterns auch einen Parameter hat, dessen Wert *True* oder *False* sein kann, und der bestimmt, ob der Stern gefüllt werden soll oder nicht.

```
from gturtle import *

makeTurtle()

def star(size, filled):
    if filled:
        startPath()
    repeat 9:
        forward(size)
        left(175)
        forward(size)
        left(225)
    if filled:
        fillPath()

clear("black")
repeat 5:
    color = askColor("Color selection", "yellow")
    if color == None:
        break
    setPenColor(color)
    setFillColor(color)
    setRandomPos(400, 400)
    back(100)
    star(100, True)
```

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

MEMO

Die Funktion askColor() hat einen Parameter für den Text in der Titelzeile und für die Farbe, die als Standardwert ausgewählt ist. Die Funktion gibt beim Klicken des OK-Buttons die ausgewählte Farbe und beim Klicken des Abbrechen-Buttons den speziellen Wert *None* zurück. Du kannst mit einer if-Anweisung auf diesen Wert testen und die repeat-Schleife mit *break* abbrechen.

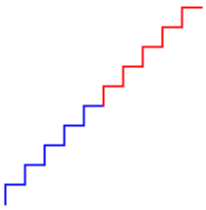
Eine Variable oder einen Parameter, der die Werte *True* oder *False* annehmen kann, nennt man eine boolesche Variable bzw. einen booleschen Parameter [mehr...]. Du kannst direkt mit if filled: auf den Wert testen, es ist also nicht nötig (und wenig elegant) *if filled == True:* zu schreiben.

AUFGABEN

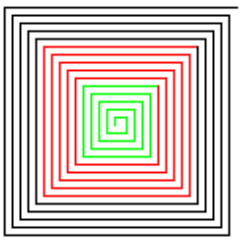
1. In einer Dialogbox fragst du den Benutzer, wie gross die Seitenlänge eines Quadrats sein soll. Falls sie kleiner ist als 50, wird ein rotes Quadrat mit dieser Seitenlänge gezeichnet, sonst ein grünes.

2. Die Turtle soll mir *repeat 10* eine Treppe mit 10 Stufen zeichnen, wobei die ersten 5 Stufen blau und die restlichen rot sind (Abbildung a).

(a)



(b)



Die Turtle soll eine Spirale zeichnen und dabei zuerst die grüne, dann die roten und am Schluss die schwarze Farbe verwenden (Abbildung b).

2.8 WHILE-SCHLEIFEN

EINFÜHRUNG

Du hast bereits den Befehl *repeat* kennengelernt, mit dem du einen Programmblock mehrmals wiederholen konnst. *repeat* kannst du so allerdings nur im TigerJython verwenden. Die *while*-Struktur ist aber überall einsetzbar.

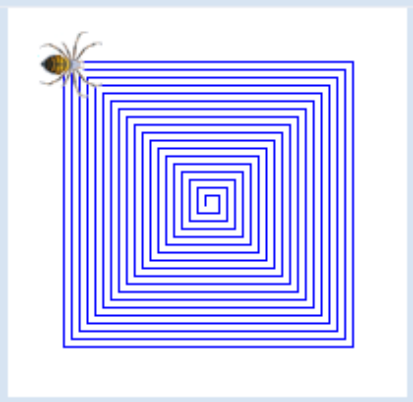
Die *while-Schleife* wird mit dem Schlüsselwort *while* eingeleitet, gefolgt von einer Schleifenbedingung. Die Anweisungen im Schleifenblock werden wiederholt, so lange die Bedingung erfüllt ist. Nach Ende der Wiederholungen wird das Programm mit der nächsten Anweisung nach dem Schleifenblock fortgesetzt.

PROGRAMMIERKONZEPTE: *Iteration, While-Struktur, Verknüpfte Bedingungen, Schleifenabbruch*

SPINNENNETZ

Mit Hilfe einer while-Schleife soll die Turtle eine reckteckige Spirale zeichnen. Dazu verwenden wir eine Variable a, die den Startwert 5 erhält und bei jedem Schleifendurchlauf um 2 vergrössert wird. Solange die Bedingung a < 200 wahr ist, werden die Anweisungen im Schleifenblock ausgeführt.

Zur Erhöhung des Spassfaktors nimmst du anstelle der Turtle eine Spinne.



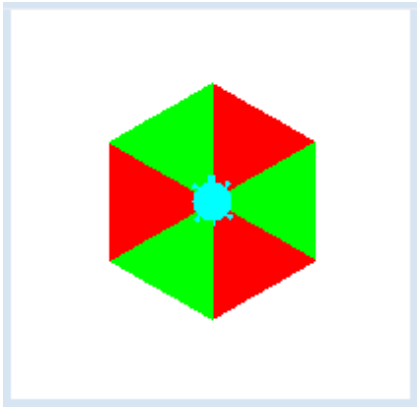
```
from gturtle import *

makeTurtle("sprites/spider.png")

a = 5
while a < 200:
    forward(a)
    right(90)
    a = a + 2
```

MEMO

Eine while-Schleife dient zur Wiederholung eines Programmblocks. Die Bedingung muss wahr sein, damit der Programmblock ausgeführt wird. Darum spricht man auch von einer*"Ausführungsbedingung"* oder *"Laufbedingung*". Fehlt im Schleifenblock die Wertänderung, bleibt die Laufbedingung immer wahr und das Programm bleibt endlos in der Schleife "hängen".

In unserer Lernumgebung kannst du ein hängendes Programm mit dem Stoppknopf oder mit Schliessen des Turtlefensters abbrechen. Im allgemeinen sind endlose Schleifen ohne Abbruch-Möglichkeit aber gefährlich, da im Extremfall ein Neustart des Computers nötig ist.

BEDINGUNGEN MIT OR VERKNÜPFEN



Die Turtle soll mit einer while-Schleife die nebenstehende Figur zeichnen. Wie du siehst, zeichnet sie abwechslungsweise rote und grüne Dreiecke.

Für den Farbwechsel kannst du folgenden Trick verwenden: Du testest die Schleifenvariable darauf, ob sie 0, 2 oder 4 ist und wählst die Stiftfarbe rot.

Mit dem Befehl fillToPoint(0. 0) kannst du eine Figur während des Zeichnens füllen. Dabei wird am Punkt (0, 0) sozusagen ein Gummiband befestigt, dessen anderes Ende die Turtle mitzieht. Dabei werden alle Punkte, über die sich das Gummiband bewegt, fortlaufend eingefärbt.

```
from gturtle import *

def triangle():
    repeat 3:
        forward(100)
        right(120)

makeTurtle()
i = 0
while i < 6:
    if i == 0 or i == 2 or i == 4:
         setPenColor("red")
    else:
         setPenColor("green")

    fillToPoint(0, 0)
    triangle()
    right(60)
    i = i + 1
```

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

MEMO

Bei Verwendung von mehreren Programmstrukturen musst du auf die korrekte Einrückung der einzelnen Schleifenblöcke achten.

Wie du siehst, kannst du zwei oder mehr Bedingungen mit *or* verknüpfen. Eine so verknüpfte Bedingung ist dann wahr, wenn die eine oder auch die andere Bedingung erfüllt ist (also auch, wenn beide Bedingungen erfüllt sind).

Mit dem Befehl *fillToPoint(x, y)* kannst du Figuren mit der Stiftfarbe während des Zeichnens füllen, im Gegensatz zum Befehl *fill()*, mit dem du bereits gezeichnete geschlossene Figuren füllen kannst.

BEDINGUNGEN MIT AND VERKNÜPFEN

Die Turtle soll mit einer while-Schleife 10 zusammengebaute Häuser zeichnen. Die Häuser sind von 1 bis 10 nummeriert. Die Häuser mit den Nummern 4 bis 7 sind gross, die übrigen klein.

In der *while*-Schleife wird die Hausnummer *nr* benutzt, um die Grösse der Häuser zu bestimmen. Wenn *nr* grösser als 3 und kleiner als 8 ist, sind die Häuser gross.

Zum Färben verwenden wir den Befehl fillToHorizontal(0). Dadurch wird die Fläche zwischen der gezeichneten Figur und der waagrechten Linie y = 0 fortlaufend gefüllt.

```
from gturtle import *

makeTurtle()
setPos(-200, 30)
right(30)
fillToHorizontal(0)
setPenColor("sienna")

nr = 1
while nr <= 10:
    if nr > 3 and nr < 8:
        forward(60)
        right(120)
        forward(60)
        left(120)
    else:
        forward(30)
        right(120)
        forward(30)
        left(120)

    nr += 1
```

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

MEMO

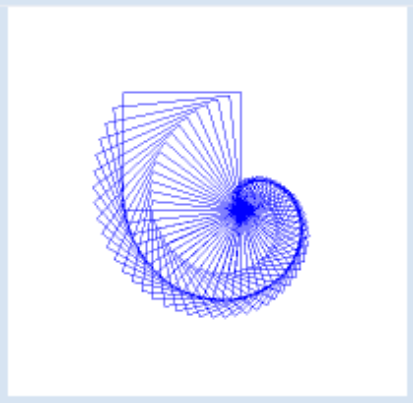
Zwei Bedingungen kannst du mit and verknüpfen. Eine solche Verknüpfung ist nur dann wahr, wenn beide Bedingungen erfüllt sind.

Mit dem Befehl fillToHorizontal(y) kannst du Figuren mit der Stiftfarbe während des Zeichnens füllen. Gefüllt wird die Fläche zwischen der gezeichneten Figur und der horizontalen Linie durch y.

nr += 1 kannst du lesen als: *nr* wird erhöht um 1. Es ist eine abgekürzte Scheibweise für die Zuweisung *nr = nr + 1*.

SCHLEIFEN MIT BREAK VERLASSEN

Eine Schleife, deren Bedingung immer wahr ist, wird endlos durchlaufen. Du kannst allerdings das Verlassen einer Schleife zu irgendeinem Zeitpunkt mit dem Schlüsselwort break erzwingen. Dein Programm zeichnet gedrehte Quadrate mit zunehmender Seitenlänge, bis die Seitenlänge 120 beträgt.



```
from gturtle import *

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

makeTurtle()
hideTurtle()

i = 0
while 1 == 1:
    if i > 120:
        break
    square(i)
    right(6)
    i += 2
print "i =", i
```

MEMO

Statt while 1 == 1: kannst du eleganter *while True*: einsetzen, da *True* immer wahr ist. (Im Gegensatz zum Wert *False*, der immer falsch ist.) Die Schleife wird in Zweierschritten durchlaufen. Statt i = i + 2 verwendest du die abgekürzte Schreibweise i += 2 (i wird erhöht um 2). Mit dem print-Befehl kannst du etwas in die TigerJython-Console im unteren Bereich des Editors schreiben. Für Text verwendest du Anführungszeichen und trennst Zahlen mit einem Komma ab. Es wird automatisch ein Leerzeichen zwischen dem Text und der Zahl eingefügt. Verstehst du, warum i = 122 ausgegeben wird? Selten gebraucht wird das Schlüsselwort ***continue***, das bewirkt, dass der restliche Teil des Schleifenkörpers übersprungen wird, die Schleife danach aber weitergeführt wird.

EINGABE-VALIDIERUNG

Gibst du dem Benutzer die Möglichkeit mit einem Eingabedialog einen Wert in einem bestimmten Bereich einzugeben, so kannst du dich nicht darauf verlassen, dass er sich an deine Vorgaben hält. Ein "robustes" Programm überprüft die Eingabe und fängt eine fehlerhafte Eingabe mit einer Rückmeldung ab. Diese

```
from gturtle import *

makeTurtle()

n = 0
while n < 1 or n > 3:
    n = inputInt("Enter 1, 2 or 3")
if n == 1:
    setPenColor("red")
```

Eingabeprüfung kannst du am besten mit einer while-Schleife durchführen, die solange durchlaufen wird, bis sich der Eingabewert an die Vorgaben hält.

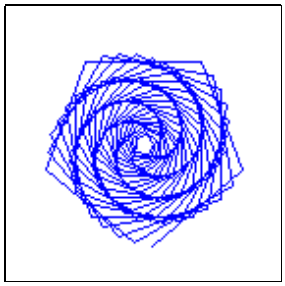
```
elif n == 2:
    setPenColor("green")
else:
    setPenColor("yellow")
dot(200)
```

In deinem Programm wählt der Benutzer mit den Zahlen 1, 2, oder 3 die Farben rot, grün oder gelb des gezeichneten Kreises aus.

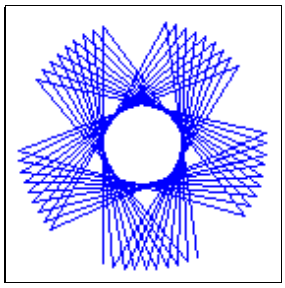
AUFGABEN

1. Die Turtle bewegt sich um eine Strecke 5 vorwärts, dreht sich um 70° nach rechts und vergrössert die Streckenlänge um 0.5. Diese Schritte wiederholt sie, solange die Streckenlänge kleiner als 150 ist.

   Versuche es auch mit dem Drehwinkel 89°!

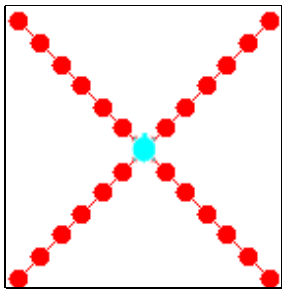


2. Wie du sicher gemerkt hast, beträgt die Drehung an der Spitze bei einem 5-er Stern 144°. Verändere diesen Drehwinkel ganz wenig, z. B. 143° und vergrössere die Anzahl der Wiederholungen. Dann erhältst du eine neue Figur.



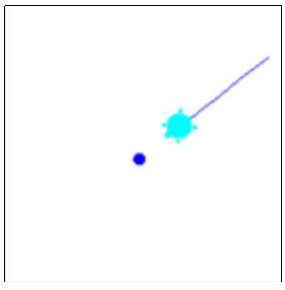
3. Die Turtle zeichnet ein Diagonalmuster mit gefüllten roten Kreisen, die die Distanz 40 haben. Der Abstand der Kreise vom der Mitte ist kleiner als 400.

   Verwende den Befehl *dot(25)*, um die Kreise zu zeichnen.



4. Die Turtle befindet sich an der Position (250, 200). Sie will sich mit Schritten der Länge 10 auf einer Geraden an die Homeposition laufen. Sie bewegt sich dabei so lange, bis der Abstand zur Home-Position kleiner als 1 ist.

   Verwende die Befehle *towards()* und *heading(degrees)* aus der Dokumentation.



5*. Du möchtest die Turtle präziser bei Home positionieren und wählst als Abstandskriterium einen 10-mal kleineren Wert. Es kann sein, dass die Turtle jetzt nicht mehr anhält. Begründe das Verhalten.

2.9 REKURSIONEN

EINFÜHRUNG

Aus deiner frühen Kindheit kennst du möglicherweise die etwas unheimliche Geschichte vom Mann mit dem hohlen Zahn:

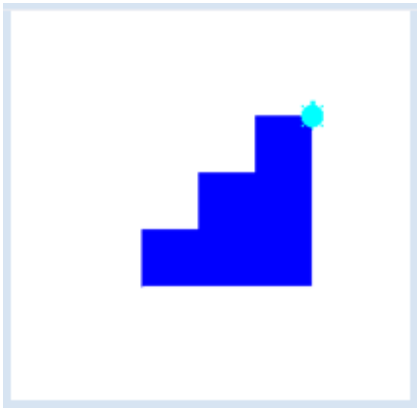
Äs isch ämal ä Ma gsi	Es war einmal ein Mann,
dä het ä hohle Zahn gha	der hatte einen hohlen Zahn,
u i däm hohle Zahn isch äs Schachteli gsi	in diesem hohlen Zahn befand sich eine Schachtel
u i däm Schachteli isch äs Briefli gsi	in der Schachtel war ein Brief
u i däm Briefli isch gstande:	in diesem Brief stand:
Äs isch ämal ä Ma gsi...	Es war einmal ein Mann...

Strukturen, die in ihrer Definition wieder sich selbst verwenden, nennt man ***rekursiv***. Du kennst sicher die russische Matrjoschka: Eine Matrjoschka enthält in sich eine (etwas kleinere) Matrjoschka, diese enthält in sich eine Matrjoschka, diese enthält...

PROGRAMMIERKONZEPTE: *Rekursion, Rekursionsverankerung, Indirekte Rekursion*

REKURSIVER TREPPENBAU

Man stellt dir die Aufgabe eine Treppe mit 3 Stufen zu bauen. Statt zu sagen, dass du dreimal eine Stufe legen musst, könntest du aber auch sagen, dass eine Treppe aus 3 Stufen aus einer Stufe und einer Treppe aus 2 Stufen besteht, dann wiederum dass eine Treppe aus 2 Stufen aus einer Stufe und einer Treppe aus 1 Stufe besteht und dass eine Treppe aus 1 Stufe aus einer Stufe und einer Treppe aus 0 Stufen besteht, eine Treppe aus 0 Stufen aber nichts ist.



Diese Bauanleitung nennt man rekursiv, da in der Definition der Treppe mit 3 oder allgemein n Stufen die Treppe mit 2 oder allgemein n-1 Stufen verwendet wird. Als Programmcode:

```
def treppe(n):
    stufe()
    treppe(n-1)
```

Du wirst gleich die Turtle damit beauftragen, mit stufe() einen Treppenblock zu zeichnen, um dann mit dem Aufruf treppe(3) eine 3 stufige Treppe zu bauen. Aber warte noch, es fehlt doch die Angabe, dass beim Bau einer Treppe aus 0 Stufen gar nichts geschehen soll. Dies kannst du aber leicht mit einer if-Bedingung einbauen:

```
if n == 0:
   return
```

Die Anweisung *return* besagt, dass sie aufhören und zur vorherigen Arbeit zurückkehren kann. Dein Programm sieht nun so aus:

```
from gturtle import *

def stairs(n):
    if  n == 0:
        return
    step()
    stairs(n - 1)

def step():
    forward(50)
    right(90)
    forward(50)
    left(90)

makeTurtle()
fillToHorizontal(0)
stairs(3)
```

MEMO

Unter Rekursionen versteht man ein fundamentales Lösungsverfahren in der Mathematik und Informatik, bei dem ein Problem derart gelöst wird, dass man es auf das gleiche, aber etwas vereinfachte Problem zurückführt. Wenn also f ein Funktion ist, die das Problem löst, so wird bei einer (direkten) Rekursion in der Definition von f wieder f verwendet. Auf den ersten Blick scheint es seltsam, dass man ein Problem derart lösen will, dass man die Lösung bereits voraussetzt. Dabei übersieht man aber einen wesentlichen Punkt: Es wird nicht genau dasselbe Problem zur Lösung verwendet, sondern eines, das **der Lösung näher liegt**. Dazu verwendet man einen meist ganzzahligen Ordnungsparameter n, den man f übergibt.

```
def f(n):
   ...
```

Bei der Wiederverwendung von f im Definitionsteil wird der Parameter verkleinert:

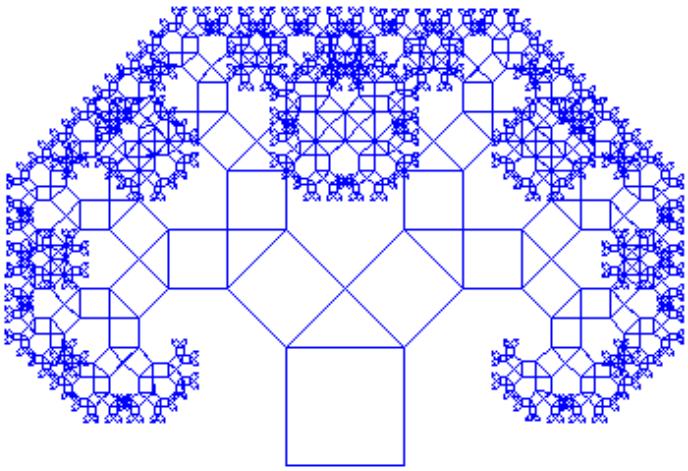
```
def f(n):
   ...
   f(n-1)
   ...
```

Eine so definierte Funktion würde sich aber **endlos** selbst aufrufen. Um dies zu verhindern, braucht man eine **Abbruchbedingung**, die man **Rekursionsverankerung** (oder Rekursionsbasis) nennt.

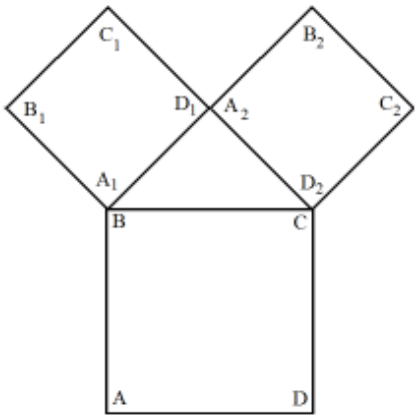
```
def f(n):
   if n == 0:
      return
   ...
   f(n - 1)
   ...
```

Mit dem Schüsselwort *return* wird die weitere Verarbeitung der Funktion abgebrochen. Man sagt auch, die Funktion kehre zurück.

DER PYTHAGORASBAUM



Mit rekursiven Algorithmen kannst du wunderbare Grafiken erzeugen. Du gehst von folgender Anleitung aus:



- Zeichne ausgehend von A ein Quadrat ABCD mit Basisseite AD
- Füge ein rechtwinkliges, gleichschenkliges BD_1C Dreieck an der Seite BC an
- Zeichne den Baum erneut ausgehend von den Quadraten A_1D_1 und A_2D_2 als Basisseiten

Es ist bekannt, dass die Umsetzung in ein rekursives Programm ungewohnt ist. Darum erhältst du hier eine ausführliche Anleitung, wie du vorgehen musst.

- Definiere einen Befehl *square(s),* mit dem die Turtle ein Quadrat mit der Seitenlänge *s* zeichnet und wieder in die Anfangsposition mit Anfangsblickrichtung zurückkehrt
- Definiere den Befehl *tree(s)*, welcher einen Baum ausgehend von einem Quadrat der Seitenlänge *s* zeichnet. In der Definition darfst du *tree()* wieder verwenden. Wichtig: **Nach dem Zeichnen des Baums ist die Turtle wieder in der Anfangsposition mit Anfangsblickrichtung**. Du überlegst schrittweise, als ob du die Turtle wärst (das neu Hinzugefügte ist grau unterlegt).
- Du zeichnest zuerst vom Punkt A aus ein Quadrat mit der Seitenlänge *s*:

```
def tree(s):
    square(s)
```

- Du fährst zur Ecke B des Quadrats, drehst 45 Grad nach links und betrachtest dies als Startpunkt eines neuen Baums mit verkleinertem Parameter s1. Es gilt nach dem Satz von Pythagoras:

$$s1 = \frac{s}{\sqrt{2}}$$

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
```

- Da du ja voraussetzt, dass du nach dem Zeichnen des Baums wieder am Startpunkt mit der Startblickrichtung landest, befindest du dich wieder in B und schaust in Richtung B1. Du drehst dich um 90 Grad nach rechts und fährst die Strecke s1 vorwärts. Jetzt bist du im Punkt D1 und hast die Blickrichtung zu B2. Von hier aus zeichnest du den Baum erneut.

```
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
```

- Jetzt musst du nur noch an den Anfangsort A mit der Anfangsblickrichtung zurückkehren. Dazu bewegst du dich um s1 rückwärts, drehst dich um 45 Grad nach links und fährst um s rückwärts.

```
    forward(s1)
    tree(s1)
def tree(s):
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
    forward(s1)
    tree(s1)
    back(s1)
    left(45)
    back(s)
```

```
from gturtle import *
import math

def tree(s):
    if s < 2:
        return
    square(s)
    forward(s)
    s1 = s / math.sqrt(2)
    left(45)
    tree(s1)
    right(90)
    forward(s1)
    tree(s1)
    back(s1)
    left(45)
    back(s)

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

makeTurtle()
ht()
setPos(-50, -200)
tree(100)
```

MEMO

Bei vielen rekursiv definierten Figuren ist es wichtig, dass die Turtle wieder an ihren Anfangsort mit der Anfangsblickrichtung zurückkehrt.

AUFGABEN

1. Wo liegt der wesentliche Unterschied der beiden Programme? Untersuche insbesondere Ort und Richtung der Turtle nach Programmende. Warum nennt man figA eine "*Last Line Recursion*" und figB eine "*First Line Recursion*"?

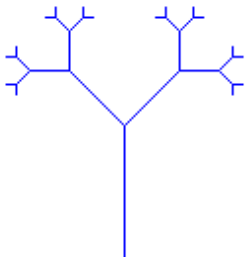
```
from gturtle import *

def  figA(s):
   if s > 200:
      return
   forward(s)
   right(90)
```

```
from gturtle import *

def  figB(s):
   if s > 200:
      return
   figB(s + 10)
   forward(s)
```

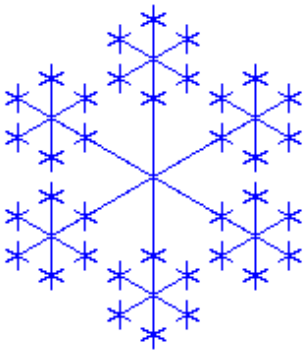
```
    figA(s + 10)

makeTurtle()
figA(100)
```

```
    right(90)

makeTurtle()
figB(100)
```

2. Ein bekannter Graf ist der vollständige binäre Baum. Er sieht für eine bestimmte Rekursionstiefe wie nebenstehend gezeigt aus. Schreibe eine rekursive Funktion tree(s), die den Baum mit der "Stammlänge" s zeichnet.

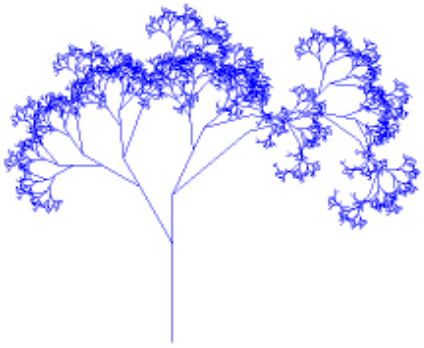


3. Zeichne die nebenstehende Sternfigur.
Definiere dazu die rekursive Funktion star(s), welche die Turtle einen Stern mit der "Dimension" s zeichnen lässt (s ist die Distanz vom Zentrum der Sternfigur zum Zentrum in der nächsten Generation). s wird von Generation zu Generation auf 1/3 reduziert. Rufe stern(180) auf und verankere die Rekursion, dass sie bei s < 20 abbricht. Wenn du hideTurtle() verwendest, wird die Turtle viel schneller zeichnen.



4*. Du wirst einen Baum zeichnen, der schon fast wie ein echter Baum aussieht. Definiere dazu die rekursive Funktion treeFractal(s), mit der "Stammlänge" s, die wie folgt aufgebaut ist:

- Verankere die Rekursion bei einer Stammlänge kleiner als 5.
- Speichere dir zuerst mit getX() und getY() die aktuelle x- und y-Koordinaten der Turtle, sowie mit heading() ihre Blickrichtung, damit du einfach zurückkehren kannst
- Jetzt fährst um s/3 nach vorne, drehst dich um 30 Grad nach links und zeichnest den Baum mit der Stammlänge 2*s/3
- Du drehst dich um 30 Grad nach rechts, fährst s/6 nach vorn und zeichnest den Baum mit der Stammlänge s/2
- Du drehst dich um weitere 25 Grad nach rechts, fährst um s/3 nach vorn, drehst 25° nach rechts und zeichnest den Baum noch einmal mit der Stammlänge s/2
- Du kehrst mit setPos() und heading() wieder in die Anfangslage mit der Anfangsposition zurück



2.10 EREIGNISSTEUERUNG

EINFÜHRUNG

Bisher bestand ein Programm aus einem einzigen Ablaufstrang, in dem eine Anweisung um die andere mit möglichen Verzweigungen und Wiederholungen ausgeführt wird. Klickst du eine Maustaste, so weisst du aber nicht, wo sich dein Programm eben gerade befindet. Um das Klicken im Programm zu erfassen, muss daher ein **neues Programmierkonzept** verwendet werden, die **Ereignissteuerung**. Hier das Prinzip:

Du definierst eine Funktion mit irgendeinem Namen, z.B. *onMouseHit()*, die im Programm nirgends explizit aufgerufen wird. Nun verlangst du vom Computer, dass **er** diese Funktion aufrufen soll, wenn die Maustaste geklickt wird. Du sagst also im Programm: *Wann immer die Maus geklickt wird, führe onMouseHit() aus*.

PROGRAMMIERKONZEPTE: *Ereignisgesteuertes Programm, Mausevent*

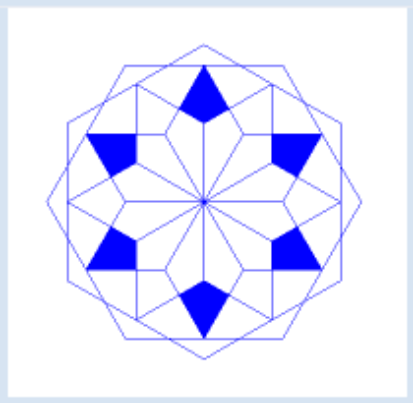
MAUSEVENTS

Das neue Konzept ist in Python sehr einfach umzusetzen. Im ersten ereignisgesteuerten Programm soll im Hauptteil die Turtle zuerst eine lustige Figur zeichnen. Nachher willst du diese noch verschönern, indem du bestimmte Bereiche mit einem Mausklick einfärbst.

Du schreibst dazu die Funktion onMouseHit(x, y), die dir die x- und y-Koordinate des Mausklicks liefert, und führst darin mit fill(x, y) ein Floodfill (Füllen eines geschlossenen Bereichs) aus.

Das Wichtige dabei ist aber, das du dem **System mitteilst**, dass es die Funktion *onMouseHit()* aufrufen soll, wenn die Maustaste gedrückt wird. Dazu verwendest du beim Aufruf von *makeTurtle()* den Parameter mit dem festgelegten Namen mouseHit und übergibst ihm den Namen deiner Funktion.

Damit die Zeichnung rasch erstellt wird, kannst du die Turtle mit *hideTurtle()* verstecken.



```
from gturtle import *

def onMouseHit(x, y):
    fill(x, y)

makeTurtle(mouseHit = onMouseHit)
hideTurtle()
addStatusBar(30)
setStatusText("Click to fill a region!")

repeat 12:
    repeat 6:
        forward(80)
        right(60)
    left(30)
```

MEMO

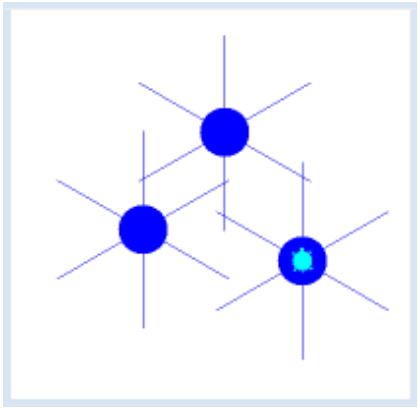
Programmtechnisch wird das Konzept der ereignisgesteuerten Programmierung bei unserer Turtle so umgesetzt, dass du eine Funktion schreibst, die beim Auftreten des Ereignisses aufgerufen werden soll. Du teilst dies dem System mit, indem du diesen Funktionsnamen als Parameter von *makeTurtle()* übergibst. Dabei verwendest du die Schreibweise *parameter_name = parameter_wert*.

Du kannst die Füllfarbe mit *setFillColor()* deinen Wünschen anpassen.

Wichtige Informationen für den Benutzer kannst du in einer Statuszeile ausschreiben, die mit addStatusBar(n) unten am Turtlefenster erscheint. Die Zahl *n* gibt die Höhe dieser Textzeile an (in Pixel).

ZEICHNEN PER MAUSKLICK

Die Turtle soll an der Stelle des Mausklicks einen Strahlenstern zeichnen. Du schreibst dazu die Funktion onMouseHit(x, y), mit der du die Turtle anweist, wie sie den Stern zeichnen soll. Damit *onMouseHit()* beim Mausklick aufgerufen wird, übergibst du in *makeTurtle()* dem Parameternamen mouseHit diesen Funktionsnamen *onMouseHit*.



```
from gturtle import *

def onMouseHit(x, y):
    setPos(x, y)
    repeat 6:
        dot(40)
        forward(60)
        back(60)
        right(60)

makeTurtle(mouseHit = onMouseHit)
speed(-1)
```

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

MEMO

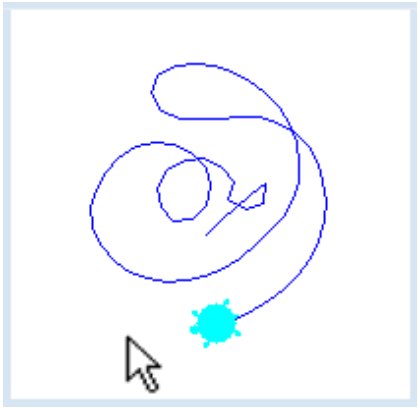
Das Programm hat einen Schönheitsfehler: Wenn du bereits wieder klickst, während die Turtle noch am Zeichnen eines Sterns ist, so zeichnet sie diesen Stern nicht fertig, sondern beginnt mit dem Zeichnen des neuen Sterns. Dabei führt sie aber die Befehle des alten Sterns auch noch weiter, wodurch der neue Stern falsch gezeichnet wird.

Dieses Falschverhalten ist offenbar darauf zurück zu führen, dass bei jedem Klick die Funktion *onMouseHit()* aufgerufen und ausgeführt wird, auch wenn die vorhergehende Ausführung noch nicht beendet ist. Um dies zu verhindern, verwendest du an Stelle des Parameters mit dem Namen *mouseHit* den Parameter mit dem Namen *mouseHitX*.

TURTLE VERFOLGT MAUS

Du möchtest, dass die Turtle ständig der Maus folgt. Dazu kannst du den Mausklick nicht gebrauchen, sondern musst die **Mausbewegung** als Ereignis betrachten. *makeTurtle()* kennt den Parameter *mouseMoved*, dem du eine Funktion übergeben kannst, die bei jeder Verschiebung der Maus aufgerufen wird.

Die Funktion onMouseMoved(x, y) erhält als Parameterwerte die aktuellen Cursorkoordinaten.



```
from gturtle import *

def onMouseMoved(x, y):
    setHeading(towards(x, y))
    forward(10)

makeTurtle(mouseMoved = onMouseMoved)
speed(-1)
```

MEMO

Neben *mouseHit* und *mouseHitX* stehen dir für das Erfassen von Mausevents in *makeTurtle()* weitere Parameter zur Verfügung.

mousePressed	Maustaste wird gedrückt
mouseReleased	Maustaste wird losgelassen
mouseClicked	Maustaste wird gedrückt und losgelassen
mouseDragged	Maus wird mit gedrückter Taste bewegt
mouseMoved	Maus wird bewegt
mouseEntered	Maus tritt in das Turtlefenster ein
mouseExited	Maus tritt aus dem Turtlefenster aus

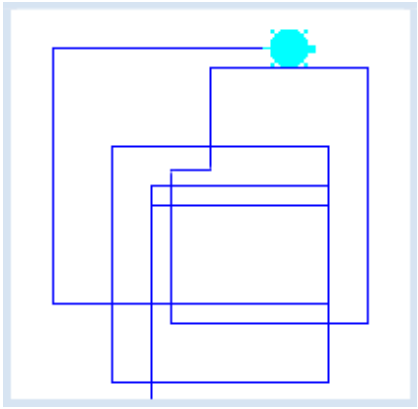
Du kannst auch mehrere Parameter gleichzeitig verwenden, also beispielsweise die zwei Funktionen *onMousePressed()* sowie *onMouseDragged()* angeben:

```
makeTurtle(mousePressed = onMousePressed, mouseDragged = onMouseDragged)
```

Mit *isLeftMouseButton()* bzw. *isRightMouseButton()* findest du heraus, welche Maustaste gedrückt wurde.

Es gibt bei diesen Events einen wichtigen Unterschied zu *mouseHit*: Die Bewegung der Turtle ist während der Ausführung der Funktion nicht sichtbar. Du solltest also entweder die Turtle mit *speed(-1)* auf hohe Geschwindigkeit setzen, mit *hideTurtle()* verstecken oder den Code für die Bewegung im Hauptteil des Programms durchführen.

TASTATUREVENTS



Auch das Drücken einer Taste auf der Tastatur löst einen Event aus.Um ihn "abzufangen", übergibst du wie vorhin mit Mausevents in *makeTurtle()* dem System den Namen der Funktion, hier *onKeyPressed()*, die es aufrufen soll, wenn das Ereignis eintritt. Dazu verwendest du den vordefinierten Parameternamen keyPressed. Beim Aufruf erhält deine Funktion eine Zahl *key*, aus der du die Taste bestimmen kannst, die gedrückt wurde. (Die Werte kannst du durch einige Versuche selbst herausfinden.) In deinem Programm bewegt sich die Turtle ständig um 10 Schritte vorwärts. Du kannst ihre Richtung mit den Cursortasten verändern. Damit die Turtle das Fenster nicht verlässt, verwendest du den Wrap-Modus.

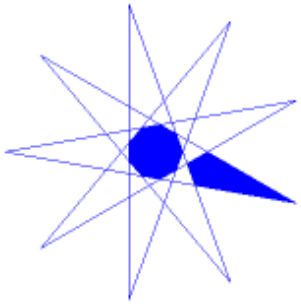
```
from gturtle import *

LEFT = 37
RIGHT = 39
UP = 38
DOWN = 40

def onKeyPressed(key):
    if key == LEFT:
        setHeading(-90)
    elif key == RIGHT:
        setHeading(90)
    elif key == UP:
        setHeading(0)
    elif key == DOWN:
        setHeading(180)

makeTurtle(keyPressed = onKeyPressed)
wrap()
while True:
    forward(10)
```

MEMO

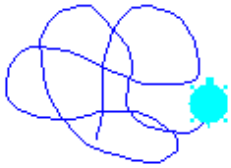
Das Hauptprogramm ist in einer Endlosschleife damit beschäftigt, die Turtle vorwärtszuschieben. Bei einem Tastaturevent wird es kurz unterbrochen und die Callbackfunktion gestartet, die sehr schnell zu Ende läuft. Das Hauptprogramm läuft dann weiter, wobei sich aber (eventuell) die Bewegungsrichtung der Turtle geändert hat.

AUFGABEN

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



2. Du kannst die Turtle dazu verwenden, ein Programm zum Freihandzeichen zu erstellen. Dabei wird der Zeichenstift mit mit dem Press-Event gesetzt mit dem Drag-Event bewegt.



3. Mit gedrückter linken Maustaste zeichnest du eine beliebige Figur. Mit einem Klick auf die rechte Maustaste kannst du ein Gebiet ausfärben.

4. Ergänze das Programm mit der Tastatursteuerung so, dass beim Drücken der Leertaste (Key Code 32) eine geschlossene Fläche, in der sich die Turtle gerade befindet, gefüllt wird. Zeichne damit einen gefüllten Buchstaben.

ZUSATZSTOFF

DEIN PERSÖNLICHES MAUSBILD

Du kannst das Bild des Mauscursors deinen eigenen Vorstellungen anpassen und damit dem Programm ein spezielles Aussehen geben. Dazu verwendest du den Befehl *setCursor()* und übergibst einen der Werte aus der untenstehenden Tabelle. Du kannst sogar ein eigenes Mausbild verwenden, wenn du setCustomCursor() den Pfad deiner Bilddatei übergibst. Übliche Mausikonen sind 32x32 Pixel gross und haben einem transparenten Hintergrund. Sie sollten im gif oder png-Format gespeichert sein.

Das oben gezeigte Verfolgungsprogramm kannst du jetzt mit deiner eigenen Mausfigur verschönern, wobei du auch mit *moveTo()* dafür sorgst, dass die Turtle sich immer bis zur Maus bewegt.

```
from gturtle import *

def onMouseMoved(x, y):
      moveTo(x, y)

makeTurtle(mouseMoved = onMouseMoved)
setCustomCursor("sprites/cutemouse.gif")
speed(-1)
```

MEMO

Mögliche Parameter von *setCursor()*:

Parameter	**Ikone**
Cursor.DEFAULT_CURSOR	Standard-Ikone
Cursor.CROSSHAIR_CURSOR	Fadenkreuz
Cursor.MOVE_CURSOR	Verschiebungs-Cursor (Kreuzpfeile)
Cursor.TEXT_CURSOR	Text-Cursor (vertikaler Strich)
Cursor.WAIT_CURSOR	Geduld-Cursor

Das Verzeichnis sprites in der Pfadangabe von *setCustomCursor()* ist im Verzeichnis, in dem sich dein Programm befindet

2.11 TURTLEOBJEKTE

EINFÜHRUNG

In der Natur ist eine Schildkröte ein Individuum mit seiner ganz spezifischen Identität. In einem Zoogehege könntest du jeder Schildkröte einen eigenen Namen geben, z.B. Pepe oder Maya. Schildkröten habe aber auch Gemeinsamkeiten: Sie sind Lebewesen aus der Tierklasse der Schildkröten. Solche Beschreibungen haben sich derart gut bewährt, dass sie in der Informatik als grundlegendes Konzept eingeführt wurden, das sich **Objektorientierte Programmierung** (**OOP**) nennt. Mit der Turtlegrafik ist es für dich spielerisch leicht, die Grundprinzipien der OOP kennen zu lernen.

PROGRAMMIERKONZEPTE: *Klasse, Objekt, Objektorientierte Programmierung, Konstruktor, Klone*

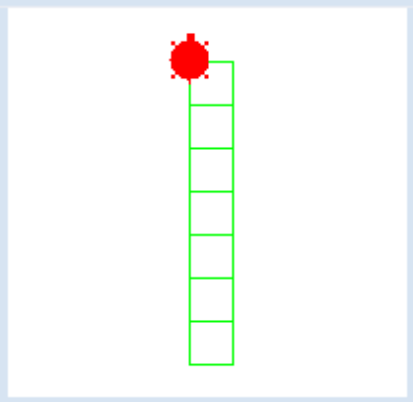
TURTLEOBJEKT ERZEUGEN

Bei der bisher verwendeten Turtle handelt es sich um ein anonymes Objekt, für das wir keinen Namen brauchten. Wenn du mehrere Turtles gleichzeitig verwenden willst, musst du aber jeder Turtle mit einem Namen eine eigene Identität geben. Den Namen kannst du als Variablennamen verwenden.

Mit der Anweisung maya = Turtle() erzeugst du eine Turtle mit dem Namen *maya*.
Mit der Anweisung *pepe* = *Turtle()* erzeugst du eine Turtle mit dem Namen *pepe*.

Du kannst die benannten Turtles mit den dir bekannten Befehlen steuern, aber du musst natürlich nun immer sagen, welche der Turtles du meinst. Dazu stellst du dem Befehl den Turtlenamen durch einen Punkt getrennt voran, zum Beispiel maya.forward(100).

Im ersten Beispiel zeichnet *maya* eine Leiter. Die Zeile *makeTurtle()* brauchst du nicht mehr, da du ja die Turtle selbst erzeugst.



```
from gturtle import *

maya = Turtle()
maya.setColor("red")
maya.setPenColor("green")
maya.setPos(0, -200)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
    maya.forward(50)
```

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

MEMO

Gleichartige Objekte werden in Klassen zusammengefasst. Ein Objekt einer Klasse wird erzeugt, indem man den Klassennamen mit einer Parameterklammer verwendet. Wir nennen dies den **Konstruktor** der Klasse. Funktionen, die zu einer bestimmten Klasse gehören, nennen wir in Zukunft **Methoden**.

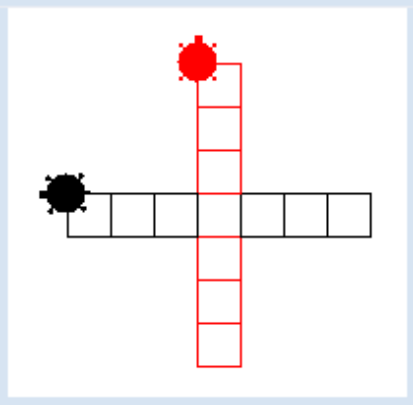
MEHRERE TURTLEOBJEKTE ERZEUGEN

Es liegt ja nun auf der Hand, dass du auf die beschriebene Art im gleichen Programm mehrere Turtles verwenden kannst. Willst du *maya* und *pepe* erzeugen, so schreibst du
maya = Turtle() und pepe = Turtle()

Allerdings befinden sich diese dann jeweils in ihrem eigenen Turtlefenster. Du kannst sie aber ins gleiche Turtlegehege setzen, indem du auch das Gehege als ein Objekt der Klasse *TurtleFrame* erzeugst:

tf = TurtleFrame()

und dieses bei der Erzeugung der Turtles angibst.
Während maya die gleiche Leiter wie vorhin baut, soll der schwarze pepe gleichzeitig eine horizontale schwarze Leiter bauen.



```
from gturtle import *

tf = TurtleFrame()

maya = Turtle(tf)
maya.setColor("red")
maya.setPenColor("red")
maya.setPos(0, -200)

pepe = Turtle(tf)
pepe.setColor("black")
pepe.setPenColor("black")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    repeat 4:
        maya.forward(50)
        maya.right(90)
        pepe.forward(50)
        pepe.left(90)
    maya.forward(50)
    pepe.forward(50)
```

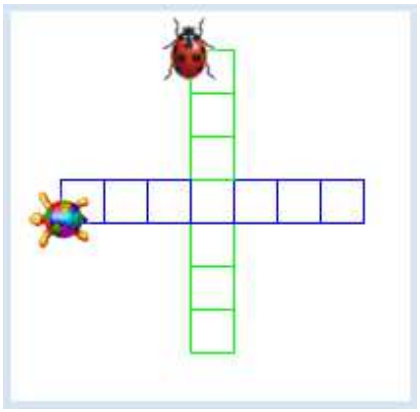
MEMO

Wenn du mehrere Turtles in das gleiche Fenster setzen willst, musst du ein *TurtleFrame* erzeugen und dieses als Konstruktorparameter der Turtle angeben. Die Turtles kollidieren nicht miteinander, sondern bewegen sich sozusagen übereinander, wobei die eben sich bewegende Turtle immer über allen anderen zu liegen kommt.

TURTLEPARAMETER

Beide Turtles zeichnen eine gleichartige Leiter. Dafür wird derselbe Code verwendet. Daher ist es eleganter, dazu eine Funktion step() zu definieren, der man mitteilt, welche Turtle die Zeichnungen ausführen soll. Dazu wird die jeweilige turtle als Parameter an die Funktion step() übergeben.

Als Parameterbezeichner kannst du irgendeinen Namen verwenden, beispielsweise lediglich t, kurz für irgendeine Turtle. Du übergibst dann beim Aufruf das eine Mal maya und das andere Mal pepe.



```
from gturtle import *

def step(t):
    repeat 4:
        t.forward(50)
        t.right(90)
    t.forward(50)

tf = TurtleFrame()

maya = Turtle(tf, "sprites/beetle.gif")
maya.setPenColor("green")
maya.setPos(0, -150)
pepe = Turtle(tf, "sprites/cuteturtle.gif")
pepe.setPos(200, 0)
pepe.left(90)

repeat 7:
    step(maya)
    step(pepe)
```

MEMO

Du kannst für jede Turtle ein eigenes Bild verwenden, wenn du beim Erzeugen der Turtle den Pfad auf die Bilddatei angibst. Hier verwendest du die beiden Bilddateien beetle.gif und cuteturtle.gif, die sich in der Distribution von TigerJython befinden.

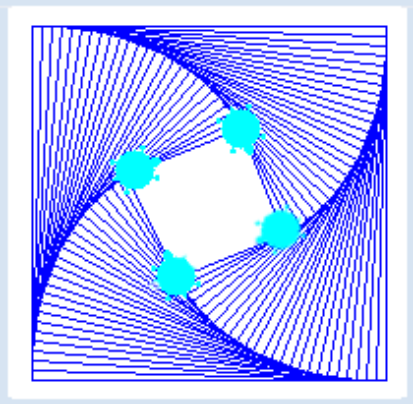
KÄFERPROBLEME MIT GEKLONTER TURTLE

Beim berühmten Käferproblem [mehr...] starten n Käfer in den Ecken eines regulären n-Ecks und verfolgen sich gegenseitig mit konstanter Geschwindigkeit. Dabei wird die Lage der Käfer in gleichen Zeitschritten fixiert und jeder Käfer in die Richtung zum Käfer an der nächsten Polygonecke gedreht. Nachher bewegen sich alle Käfer geradlinig um eine immer gleiche Schrittweite vorwärts.

Du kannst dieses Problem sehr elegant lösen, indem du zuerst mit der namenlosen (globalen) Turtle das Polygon zeichnest und an jeder Ecke ein geklontes Turtleobjekt erstellst. Du wählst hier ein Viereck und erstellst mit clone() die Turtleklons t1, t2, t3, t4. Ein Klone ist ein neues Turtle-Objekt mit identischen Eigenschaften.

Nachher stellst du in einer Endlosschleife mit setHeading() ihre Blickrichtung ein und schiebst sie um die Schrittweite 5 vorwärts. Die Zeichnung wird besonders schön, wenn du noch die Verbindungsgeraden zwischen den jeweils sich verfolgenden Turtles einzeichnest.

Am einfachsten definierst du dazu die Funktion drawLine(a, b), mit welcher die Turtle a mit moveTo() eine Spur zur Turtle b zeichnet und wieder zurück springt.



```
from gturtle import *

s = 360

makeTurtle()
setPos(-s/2, -s/2)

def drawLine(a, b):
    ax = a.getX()
    ay = a.getY()
    ah = a.heading()
    a.moveTo(b.getX(), b.getY())
    a.setPos(ax, ay)
    a.heading(ah)

# generate Turtle clone
t1 = clone()
t1.speed(-1)
forward(s)
right(90)
t2 = clone()
t2.speed(-1)
forward(s)
right(90)
t3 = clone()
t3.speed(-1)
forward(s)
right(90)
t4 = clone()
t4.speed(-1)
forward(s)
right(90)
hideTurtle()

repeat:
    t1.setHeading(t1.towards(t2))
    t2.setHeading(t2.towards(t3))
    t3.setHeading(t3.towards(t4))
    t4.setHeading(t4.towards(t1))

    drawLine(t1, t2)
    drawLine(t2, t3)
    drawLine(t3, t4)
    drawLine(t4, t1)

    t1.forward(5)
    t2.forward(5)
    t3.forward(5)
    t4.forward(5)
```

MEMO

Erzeugst du mit clone() aus der globalen Turtel eine neue Turtle, so hat diese die gleiche Position, die gleiche Blickrichtung und die gleiche Farbe (bei Verwendung von benutzerdefinierten Turtlebildern hat sie das gleiche Turtlebild) [mehr...] .

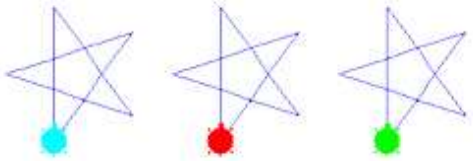
Die Funktion *drawLine()* kann vereinfacht werden, wenn man Position und Blickrichtung der Turtle mit *pushState()* abspeichert und den Zustand mit *popState()* wieder zurückholt:

```
def drawLine(a, b):
    a.pushState()
    a.moveTo(b.getX(), b.getY())
    a.popState()
```

Die Verfolgungskurve lässt sich mathematisch berechnen (siehe).

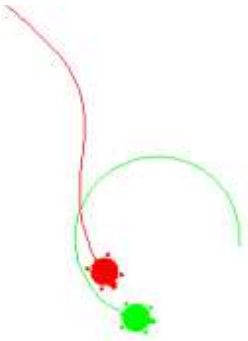
AUFGABEN

1. Drei Turtles sollen abwechlungsweise Zack-um-Zack einen fünfzackigen Stern zeichnen. Die Turtles haben die Farben cyan (Standardfarbe), rot und grün. Die Turtlefarbe kann als zusätzlicher Parameter des Konstruktors angegeben werden.



2. Eine grüne Mutterturtle bewegt sich mit grüner Stiftfarbe ständig auf einem Kreis. Eine rote Kindturtle ist zuerst weit von der Mutter entfernt und bewegt sich dann mit roter Stiftfarbe in Richtung zur Mutter.

   (Das Kind *child* kann mit *direction = child.towards(mother.getX(), mother.getY))* die Richtung zur Mutter bestimmen.)

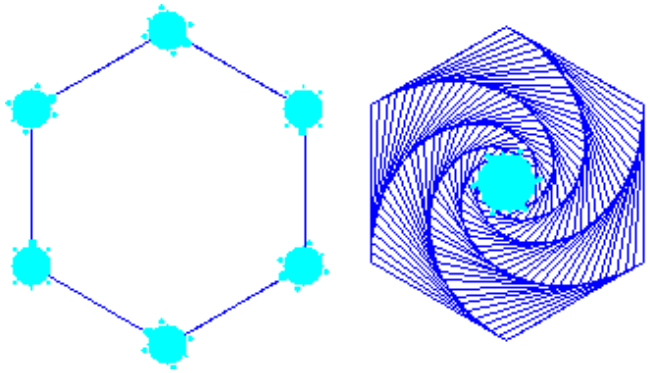


3. 

   Die Turtle laura zeichnet (nicht gefüllte) Quadrate. Nach jedem gezeichneten Quadrat springt eine zweite Turtle hinein und färbt es grün.

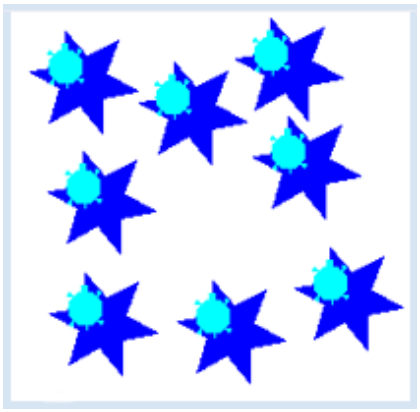
   Verwende für die beiden Turtles verschiedene Turtlebilder. Im *tigerjython2.jar* stehen die Bilder beetle.gif, beetle1.gif, beetle2.gif und spider.png zur Verfügung. Du kannst aber euch eigene Bilder verwenden. Du musst diese im Unterverzeichnis *sprites* des Verzeichnisses speichern, in dem sich dein Programm befindet.

4. Erstelle ähnlich wie im Beispiel "Käferprobleme" eine Verfolgungsgrafik für 6 Turtles, die in den Ecken eines regelmässigen 6-Eck die Verfolgung starten.



ZUSATZSTOFF

TURTLES MIT MAUSKLICK ERZEUGEN



Dein Programm erzeugt bei jedem Mausklick an der Stelle des Mauscursors eine neue Turtle, die unabhängig von den bereits vorhandenen Turtles einen Stern zeichnet. Dabei erlebst du die volle Tragweite und die Eleganz der Objektorientierten Programmierung sowie der Ereignissteuerung.

Um den Mausklick zu erfassen, definierst du eine Funktion drawStar(). Damit diese beim Drücken der linken Maustaste vom System aufgerufen wird, verwendest du im Konstruktor von TurtleFrame den benannten Parameter mouseHit und übergibst ihm den Namen dieser Funktion.

```
from gturtle import *

def drawStar(x, y):
    t = Turtle(tf)
    t.setPos(x, y)
    t.fillToPoint(x, y)
    for i in range(6):
        t.forward(40)
        t.right(140)
        t.forward(40)
        t.left(80)

tf = TurtleFrame(mouseHit = drawStar)
```

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

MEMO

In der OOP werden Objekte mit gleichen Fähigkeiten und gleichen Eigenschaften in Klassen zusammengefasst. Mit dem Konstruktor erzeugt man einzelne Objekte (Instanzen).

Um einen Mausklick zu verarbeiten, schreibst du eine Funktion mit beliebigem Namen (aber zwei Parametern x und y) und übergibt diesen Funktionsnamen im Konstruktor von *TurtleFrame* dem benannten Parameter *mouseHit*.

x und y liefern die Koordinaten des Mausklicks.

2.12 UMGANG MIT LISTEN

EINFÜHRUNG

Eine Liste kannst du dir wie eine offene Perlenkette vorstellen, in der du beliebig viele Elemente aneinander reihst. In Python kannst du in Listen beliebige Daten speichern. Zum Beispiel sind Musikstücke Listen von Tönen, Texte sind Listen von Buchstaben und Vielecke sind Punklisten. Eine der wichtigsten Operationen ist das Durchlaufen der Liste, beispielsweise um das Musikstück abzuspielen, den Text auszuschreiben oder das Vieleck zu zeichnen.

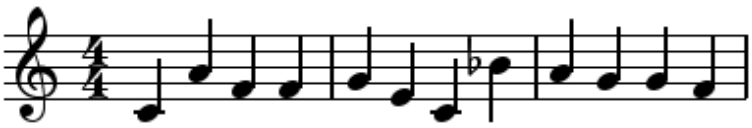
PROGRAMMIERKONZEPTE: *Listen, for-in-range-Struktur, Sound*

LISTE MIT TONFREQUENZEN

Die Turtle kann Töne mit bestimmten Frequenzen und einer bestimmten Dauer abspielen. Dazu rufst du die Funktion *playTone(frequency, duration)* auf. Der erste Parameter bestimmt die Tonfrequenz in Hz, der zweite die Abspieldauer in Millisekunden. Hier die Tonfrequenzen einiger Noten der Tonleiter:

Ton	Frequenz
c''	524
h'	494
b'	466
a'	440
g'	392
f'	349
e'	330
d'	294
c'	262

Im folgenden Beispiel enthält die Liste *song* Tonfrequenzen eines dir gut bekannten Liedes. Mit for f in song kannst du die Liste durchlaufen und die Töne mit playTone() abspielen.



```
from gturtle import *

song = [262, 440, 349, 349, 392, 330, 262, 466, 440, 392, 392, 349]
for f in song:
    playTone(f, 250)
```

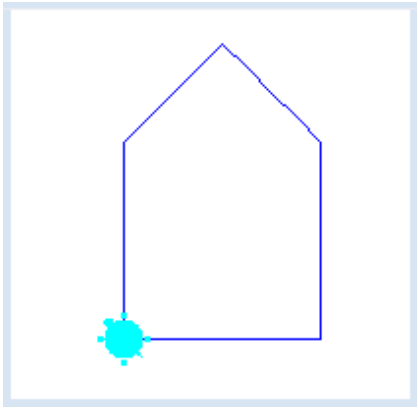
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

In Python wird eine Liste mit eckigen Klammern erstellt. Die einzelnen Elemente der Liste sind durch Kommas getrennt. Mit *for e in list* kannst du alle Elemente e der Liste der Reihe nach durchlaufen.

LISTE MIT STRECKENLÄNGEN

Mit Strecken der Länge 100 und 71, die entweder im Winkel 45° oder 90° zu einander stehen, zeichnest du ein Haus. Ausgehend von der linken unteren Ecke durchläufst du also folgende Liste: [100, 71, 0, 71, 10, 0, 100], wobei die Zahl 71 die gerundete Länge der Diagonale eines Quadrats mit der Seitenlänge 50 ist. Nach jeder gezeichneten Strecke dreht die Turtle 45° nach rechts.



```
from gturtle import *

makeTurtle()

for s in [100, 71, 0, 71, 100, 0, 100]:
    forward(s)
    right(45)
```

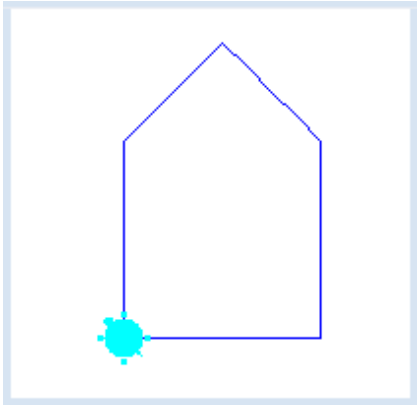
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Listen eignen sich hervorragend zur Speicherung von zusammengehörenden, gleichartigen Daten. In anderen Programmiersprachen verwendet man dazu oft einen Array.

LISTEN IN LISTEN

Ein Haus kannst du auch zeichnen, indem du die Eckpunkte angibst und mit der Funktion *moveTo()* verbindest. Da du die x-y-Koordinaten eines Eckpunkts am besten auch als Liste auffasst, entsteht auf diese Art eine Liste, deren Elemente wieder Listen sind.



```
from gturtle import *

makeTurtle()

house = [[0, 100], [50, 150], [100, 100], [100, 0], [0, 0]]

for p in house:
    moveTo(p)
```

MEMO

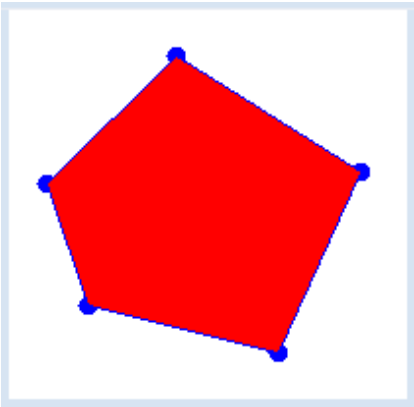
Wie du siehst, gibt es von der Funktion *moveTo()* mehrere Versionen, nämlich eine, der du zwei Parameter *x* und *y* übergeben kannst und eine, der du eine Punktliste übergibst. Die Verwendung von Punktlisten ist eleganter, weil sich zusammengehörende x-y-Werte in einer gemeinsamen Datenstruktur befinden. [mehr...]

LISTEN ERZEUGEN UND ELEMENTE HINZUFÜGEN

Mit li = [] definierst du eine leere Liste, sozusagen wie eine Perlenkette, die noch keine Perle enthält. Danach kannst du mit li.append(x) ein Element *x* zur Liste *li* hinzufügen. Dieses wird damit immer am Ende angefügt. Du kannst die Liste jederzeit mit einem print-Befehl anzeigen, wobei du einfach *print li* schreibst.

In deinem Beispiel erzeugst du zuerst eine leere Liste corner. Mit einem linkem Mausklick fügst du einen Punkt mit den Koordinaten der Mausposition zu *corner* und zeichnest gleichzeitig an dieser Stelle einen kleinen Kreis. Mit *print corner* schreibst du den Inhalt der Liste im Ausgabefenster aus.

Mit einem rechten Mausklick zeichnest du ein Polygon mit den Listenpunkten und füllst es rot aus.



```
from gturtle import *

def onMousePressed(x, y):
    if isLeftMouseButton():
        pt = [x, y]
        setPos(pt)
        dot(10)
        corner.append(pt)
        print corner
    if isRightMouseButton():
        setFillColor("red")
        startPath()
        for p in corner:
            moveTo(p)
        fillPath()

makeTurtle(mousePressed = onMousePressed)
hideTurtle()
corner=[]
```

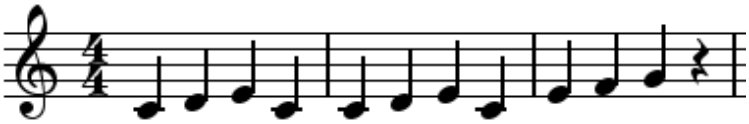
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Du brauchst also beim Schreiben des Programms nicht zu wissen, wie lang die Liste zu Laufzeit des Programms dann tatsächlich wird. Aus diesem Grund sagt man, dass eine Liste eine dynamische Datenstruktur sei. [Wichtige Listenoperationen].

AUF ELEMENTE EINER LISTE ZUGREIFEN

Auf einzelne Elemente einer Liste kannst du mit einem ***Index*** in einer eckigen Klammer zugreifen, wobei die Zählung bei 0 beginnt. Speicherst du also die Frequenzen der C-Dur-Tonleiter in der Liste d, liefert d[0] gibt das erste Element der Liste s, d.h. die Frequenz 262.



```
from gturtle import *

d = [262, 294, 330, 349, 392, 440, 494, 524]
song = [d[0], d[1], d[2], d[0], d[0], d[1], d[2], d[0], d[2], d[3], d[4], 0]
for f in song:
    playTone(f, 200)
```

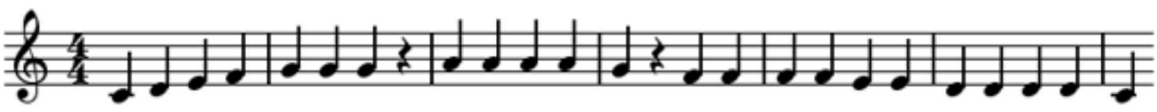
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Auf einzelne Elemente kannst du mit einem ***Index*** zugreifen. *d[2]* liefert zum Beispiel das dritte Element 330. Um es zu ändern, kannst du eine Zuweisung verwenden, also zum Beispiel *d[2] = 698*. Bei der Verwendung des Index musst du streng darauf achten, dass es das Element mit diesem Index auch tatsächlich gibt. Ist n die Anzahl der Elemente der Liste, so muss der Index immer im Bereich 0...(n - 1) liegen. Negative Indices und solche die grösser oder gleich der Länge der Liste sind, führen zu einem bösen Laufzeitfehler.

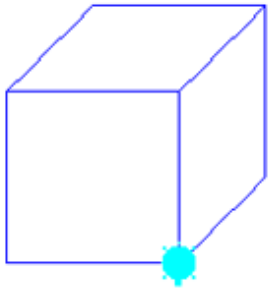
AUFGABEN

1. Erstelle eine Liste mit Tonfrequenzen des Liedes "Ali mini Enteli" und spiele sie ab.

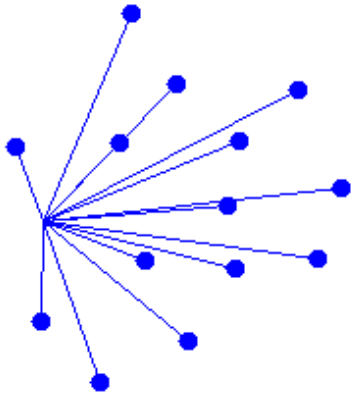


Pausen kannst du mit dem Element 0 erzeugen.

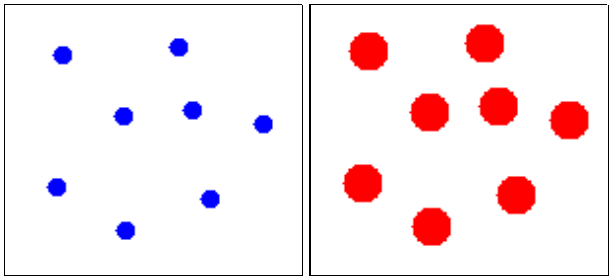
2. Die Turtle soll das nebenstehende Schrägbild eines Würfels mit Hilfe einer Eckpunktliste zeichnen. Als zusätzliche Bedingung soll sie aber jede Kante nur einmal durchlaufen.



3. Mit dem linken Mausklick kannst du beliebig viele Punkte zeichnen, die in einer Liste gespeichert werden. Klickst du mit der rechten Maustaste, so werden vom Punkt des Mausklicks aus Verbindungslinien zu allen gespeicherten Punkten gezeichnet.



4. Mit der linken Maustaste werden beliebig viele blaue Punkte gezeichnet. Anschliessend werden mit einem einzigen rechten Mausklick alle Punkte rot und etwas grösser gezeichnet.



5*. Man kann die Frequenzen der Töne wohltemperierten Tonleiter wie folgt berechnen. Wenn du von einem Grundton (z. Bsp. c' mit f = 262 Hz) ausgeht, erhältst du die Frequenz des darauf folgenden Halbtons durch Multiplikation mit dem Faktor r = 1.05946 und daher die Frequenz des nächsten Ganztons durch Multiplikation mit dem Faktor r * r ≈ 1,122.

(Der Multiplikationsfaktor r ergibt sich aus der Überlegung, dass die Oktave mit dem Frequenzverhältnis 2 aus zwölf Halbtönen besteht, also r^12 = 2 und somit r 1.05946 ist). Die Frequenzen in der Tabelle im ersten Beispiel sind gerundet.

a) Erstelle eine Liste mit Tonfrequenzen der Halbton-Tonleiter beginnend bei c' und spiele sie ab.
b) Erstelle eine Liste mit Tonfrequenzen der c-Dur-Tonleiter und spiele sie ab.

2.13 FOR-IN-RANGE

EINFÜHRUNG

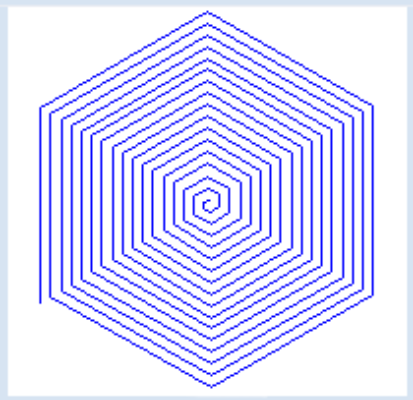
In vielen Programmiersprachen sind *for*-Schleifen eine Alternative zu den while-Schleifen und werden insbesondere dann verwendet, wenn bei jedem Schleifendurchgang eine Schleifenvariable um den gleichen Wert verändert wird. Besonders sinnvoll sind *for-Schleifen* im Zusammenspiel mit der Funktion *range()*.

PROGRAMMIERKONZEPTE: *for-Schleifen, range()*

EINFACHE FOR-IN-RANGE-SCHLEIFE

Die Funktion *range(n)*, wo n eine natürliche Zahl ist, liefert eine Liste mit aufeinander folgenden ganzen Zahlen die bei 0 beginnen und bei n-1 enden.
range(4) liefert also die Liste [0, 1, 2, 3].

In deinem Beispiel liefert *range(200)* eine Liste [0, 1, 2, 3, ...199]. Wie du bereits gelernt hast, kannst du also mit for i in range(200) diese Liste durchlaufen. Die Befehle im Schleifenblock werden daher 200 mal wiederholt. Die Turtle bewegt sich dabei jeweils i Schritte vorwärts und dreht 60° links.



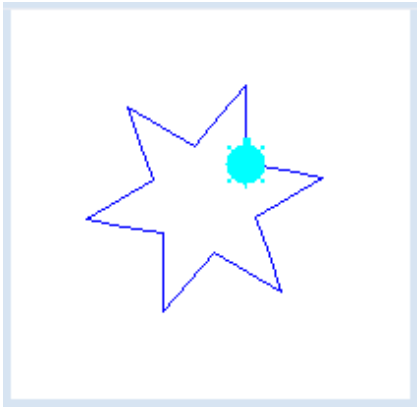
```
from gturtle import *

makeTurtle()
hideTurtle()

for i in range(200):
    forward(i)
    left(60)
```

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

MEMO

Die Funktion range(n) ist zwar hilfreich, führt aber oft zu fehlerhaften Programmen, da die Zahlen nicht bis und mit n laufen, sondern nur bis n - 1. Zudem ist der erste Wert nicht 1, sondern 0. Du musst dich daran gewöhnen, dass in der Informatik das Zählen meist bei 0 und nicht bei 1 beginnt. Für *n* darfst du keine negativen Zahlen und keine Dezimalzahlen verwenden. Du kannst n auch als Anzahl der Zahlen auffassen.

MIT FOR-SCHLEIFE ZÄHLEN



Die Schleife *for i in range(n)* wird oft als sogenannte Zählschleife eingesetzt, ohne dass i im Schleifenblock überhaupt verwendet wird. Die Zahl *n* gibt an, wie oft die Befehle im Schleifenblock wiederholt werden.

In deinem Beispiel werden mit der Schleife
for i in range(6)
die Befehle im Schleifenblock 6-mal wiederholt und die Turtle zeichnet einen 6-er Stern.

```
from gturtle import *

makeTurtle()
for i in range(6):
    forward(40)
    left(140)
    forward(40)
    right(80)
```

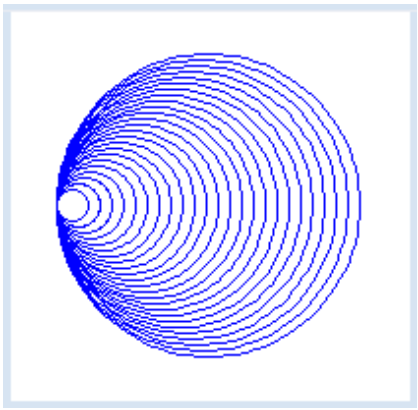
Programmcode markieren (Ctrl+C kopieren, Ctrl+V einfügen)

MEMO

Die for-Schleife *for i in range(n)* ohne Verwendung von *i* entspricht der Funktion *repeat n*, die du als Wiederholungsstruktur in den vorhergehenden Kapiteln oft verwendet hast. Die *repeat*-Schleife benötigt keine Zählvariable und ist daher für Programmiereinsteiger einfacher. Du kannst sie aber nur mit *TigerJython* verwenden.

RANGE MIT DREI PARAMETER

Die Funktion *range()* kann auch 2 oder 3 Parameter haben. Der erste Parameter ist der *Startwert* der Liste, der zweite der *Stoppwert* und der dritte die *Wertänderung* von einem Element zum nächsten. Fehlt der dritte Parameter, ist die Wertänderung 1. Wichtig und etwas gewöhnungsbedürftig ist, dass der Stoppwert selbst in der Liste nicht enthalten ist, d.h. dass die Zahlen bei aufsteigenden Werten immer kleiner als der Stoppwert bleiben.



Beispiele:
range(4, 10) liefert die Liste [4, 5, 6, 7, 8, 9]
range(2, 20, 3) liefert die Liste [2, 5, 8, 11, 14, 17]

In deinem Beispiel verwendest du die Funktion *range()* mit drei Parametern, um eine Kreisschar zu zeichnen Der erste Kreis hat den Radius 10 (*Startwert*), der Radius des letzten Kreises ist kleiner als 200 (*Stoppwert*) und jeder folgende Kreis ist um 5 grösser als der vorhergehende (*Wertänderung*).

```
from gturtle import *

makeTurtle()
hideTurtle()
setPos(-200, 0)

for x in range(10, 200, 5):
    rightArc(x, 360)
```

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

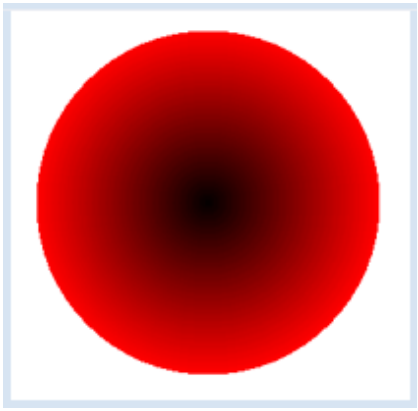
MEMO

range(start, stop, step) liefert mit positivem step eine Liste mit ganzen Zahlen, die bei start beginnen und jeweils um step vergrössert werden. Der letzte Wert ist immer streng kleiner als stop.

RÜCKWÄRTS ZÄHLEN

In range(start, stop, step) kann die Wertänderung step auch negativ sein. Damit kannst du eine for-Schleife rückwärts durchlaufen.

In deinem Beispiel zeichnest du zuerst den grössten gefüllten Kreis und dann immer kleinere, dunklere Kreise. Du verwendest dabei die Funktion *makeColor(r, g, b)*, die eine RGB -Farbe mit den Komponenten *r*, *g* und *b* erzeugt, wobei *g* und *b* immer null sind.



```
from gturtle import *

makeTurtle()
hideTurtle()

for i in range(255, 0, -5):
    c = makeColor(i, 0, 0)
    setPenColor(c)
    dot(i)
```

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

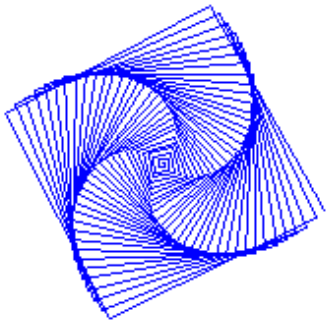
MEMO

In for i in range(start, stop, step) kannst du für *start, stop* und *step* auch negative, ganze Zahlen einsetzen. Ist *step* negativ, so wird jeder nachfolgende i-Wert verkleinert.

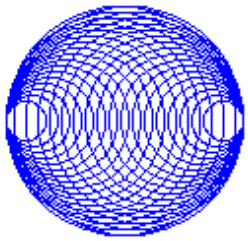
Anstelle einer for-Schleife kannst du aber auch immer die *while*-Schleife einsetzen. Diese benötigt zwar zwei Programmzeilen mehr, kann aber auch dann verwendet werden, wenn der *Startwert*, *der Stoppwert* oder die *Wertänderung* Dezimalzahlen sind.

AUFGABEN

1. Erstelle die nebenstehende Zeichnung, indem du die Zeilen
forward(i)
right(89)
200 mal wiederholst. Verwende dazu eine for-in-range-Struktur.

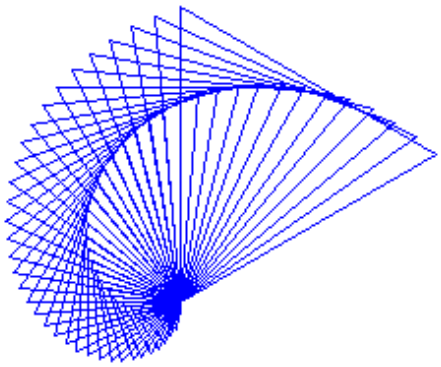


2. Zeichne die nebenstehende Figur.
Die Kreise kannst du mit
rightArc(radius, angle) oder
leftArc(radius, angle)
zeichnen.



3. Zeichne mit for-in-range gefüllte Kreise im Abstand 50, wobei die x-Koordinate des ersten Kreises -250 und des letzten 250 ist.

4. Die Turtle zeichnet ein gleichseitiges Dreieck mit der Seitenlänge 250, dreht dann um 5° nach links und zeichnet das nächsten Dreieck mit einer um 4 verkleinerten Seitenlänge usw., solange die Seitenlänge grösser als 5 ist.



2.14 ZUFALLSZAHLEN

EINFÜHRUNG

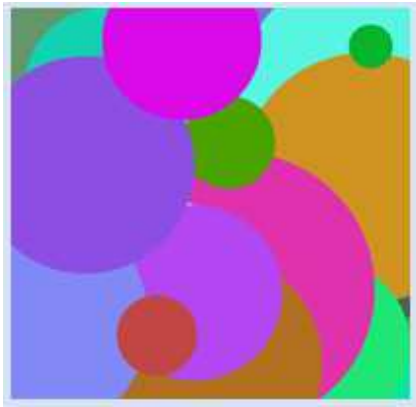
Um die Gesetzmässigkeiten des Zufalls mit dem Computer zu untersuchen, führt man **Experimente** durch, bei denen der Programmverlauf durch Zufallszahlen gesteuert wird. Diese werden mit einer Funktion erzeugt, die beim wiederholten Aufruf zufällige Zahlen aus einem bestimmten Zahlenbereich zurückgibt.

PROGRAMMIERKONZEPTE: *Zufallszahlen, Zufallsexperimente*

ZUFÄLLIGES MALEN

Ganzzahlige Zufallszahlen im Zahlenbereich a bis b kannst du durch wiederholten Aufruf von ***randint(a, b)*** erzeugen, wobei a die kleinste und b die grösste der auftretenden Zufallszahlen ist.

Du zeichnest farbig gefüllte Kreise mit zufälligem Durchmesser im Bereich 50 und 400 an zufälligen Mittelpunkten im Bereich -300 und 300. Die Farbe wird ebenfalls zufällig erzeugt, wobei die rote, grüne und blaue Farbkomponente je eine Zufallszahl zwischen 0 und 255 ist.



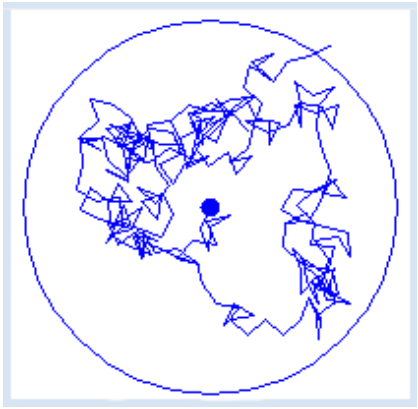
```
from gturtle import *
from random import randint

makeTurtle()
hideTurtle()

for i in range(50):
    x = randint(-300, 300)
    y = randint(-300, 300)
    setPos(x, y)
    r = randint(0, 255)
    g = randint(0, 255)
    b = randint(0, 255)
    c = makeColor(r, g, b)
    setPenColor(c)
    d = randint(50, 400)
    dot(d)
```

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

MEMO

Durch wiederholten Aufruf von randint(a, b) erhältst du ganzzahlige Zufallszahlen, wobei a die kleinste und b die grösste Zahl ist. Im Gegensatz zu *range()* ist hier die grösste Zahl b mit eingeschlossen. Hier verwendest du from random import randint, damit du nicht bei jedem Aufruf von *randint()* den Modulnamen *random* voranstellen musst.

RANDOM WALK



Die Turtle befindet sich in einem kreisförmigen Gehege mit dem Radius 200. Sie startet an der Homeposition (0, 0) und bewegt sich wiederholt um 20 Schritte in beliebiger Richtung vorwärts Es zeigt sich, dass sie sich im Mittel trotz der Zufallsbewegung vom Startpunkt entfernt, also irgendwann einmal den Rand des Geheges erreicht.

In deinem Programm schreibst du in der Statusbar laufend die Anzahl der Schritte und die Distanz vom Startpunkt aus. Damit das Zufallsexperiment etwas schneller abläuft, wird die Turtle mit *hideTurtlet()* unsichtbar gemacht.

```
from gturtle import *
import random

makeTurtle()
hideTurtle()
addStatusBar(20)
dot(10)
openDot(400)
n = 0
r = 0
while r < 200:
    z = random.random()
    angle = z * 360
    heading(angle)
    forward(20)
    r = distance(0, 0)
    setStatusText("Number of steps: " + str(n) + ". Distance: " + str(r))
    n += 1
```

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

MEMO

Du musst mit addStatusBar(20) die Statusbar mit der Höhe 20 Pixel anzeigen lassen, bevor du mit setStatusText() einen Text hineinschreiben kannst.

Die Funktion *z = random.random()* liefert eine dezimale Zufallszahl z mit einem Wert zwischen 0 und 1 (1 kommt allerdings nie vor). Damit kannst du Zufallszahlen im Bereich 0..360 erzeugen, indem du z mit 360 multiplizierst.

Mit distance(0, 0) kannst du sehr einfach die Distanz der Turtle vom Startpunkt ermitteln. Führst du die Simulation oftmals durch, so siehst du schwankende Schrittzahlen, um den Gehegerand zu erreichen. In Aufgabe 3 untersuchst du, wie viele Schritte **im Mittel** dazu nötig sind.

BERECHNUNG VON π MIT ZUFALLSPUNKTEN

Um den Flächeninhalt S einer beliebigen ebenen Fläche zu bestimmen, bettest du sie in ein Quadrat oder Rechteck mit bekanntem Flächeninhalt U ein und lässt zufällige Punkte darauf fallen. Es ist einleuchtend, dass der prozentuale Anteil der Punkte, die auf S fallen,

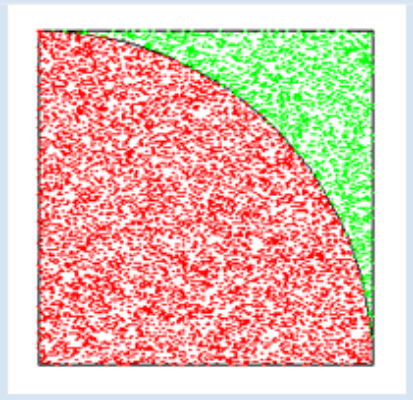
gleich dem Verhältnis S/U ist. Hat beispielsweise S einen Drittel der Gesamtfläche U, so fallen im Mittel auch nur 1/3 aller Punkte auf S.

Mit einem solchen "Zufallsregen" (auch Monte-Carlo-Verfahren genannt) kannst du elegant die Zahl π berechnen. Du bettest dazu in ein Quadrat mit der Seitenlänge r = 1 (also mit dem Flächeninhalt U = 1) einen Viertelkreis mit dem Flächeninhalt

$$S = \frac{1}{4} * r^2 * \pi = \frac{\pi}{4}$$

ein und wirfst n Zufallspunkte auf das Quadrat. Du zählst dabei die Anzahl k der Punkte, die auf den Viertelkreis fallen. Das prozentuale Anteil k / n der Punkte die auf den Viertelkreis fallen, ist S / U = S, also

$$\frac{\pi}{4} = \frac{k}{n} \quad \text{oder} \quad \pi = \frac{4 * k}{n}$$



Die Zahl PI ist 3.1492

In deinem Programm erstellst du auch eine grafische Darstellung, um das Verfahren anschaulich zu demonstrieren. Um zu bestimmen, ob die Punkte in den Viertelkreis fallen, verwendest du den Satz von Pythogoras

```
from gturtle import *
import random

from gturtle import *

def init():
    setPenColor("black")
    for i in range(4):
        forward(200)
        right(90)
    moveTo(0, 200)
    right(90)
    rightArc(200, 90)

makeTurtle()
hideTurtle()
addStatusBar(20)
n = 10000
k = 0
init()

for i in range(n):
    zx = random.random()
    zy = random.random()
    if zx * zx + zy * zy <= 1:
        k = k + 1
        setPenColor("red")
    else:
        setPenColor("green")
    setPos(zx * 200, zy * 200)
    dot(2)
pi = 4 * k/n
setStatusText("PI  =  " + str(pi))
```

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

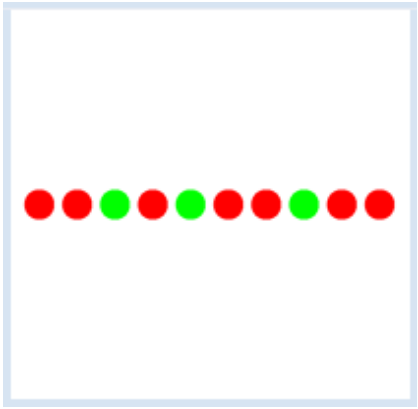
MEMO

Mit 1000 Regentropfen wird π bereits auf 2 Stellen genau. Je grösser *n*, umso genauer wird das Ergebnis.

BÜCHSENSPIEL

Auf einem Jahrmarkt versuchen 10 Personen mit einem Ball 10 gegenüberliegende Büchsen zu treffen, so dass diese umfallen. Alle Personen werfen gleichzeitig und wählen zufällig eine der gegenüberliegenden Büchsen. Es wird angenommen, dass jeder Wurf ein Treffer ist.

Einige der Büchsen bleiben allerdings stehen, da mehrere Personen dieselbe Büchse anvisiert haben. Du möchtest mit dem Computer herausfinden, wie viele Büchsen im Mittel stehen bleiben.



Dazu simulierst du das Spiel mit Zufallszahlen und lässt den Computer 100 Mal 10 Bälle werfen. Nach jeder Runde zählst du die Zahl der getroffenen und stehengebliebenen Büchsen und berechnest am Schluss den Mittelwert. Grafisch kannst du Büchsen mit Kreisen darstellen. Zu Beginn jeder Runde sind alle grün, die getroffenen werden rot markiert.

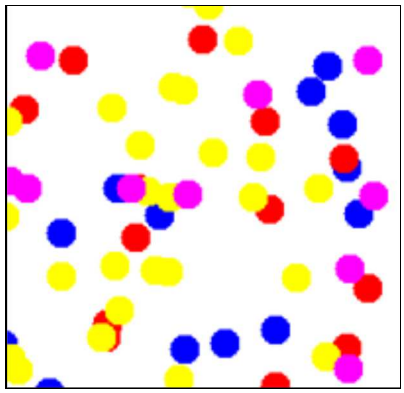
```
from gturtle import *
import random

def init():
    for i in range(1, 11):
        setPos(-280 + 50 * i, 0)
        setPenColor("green")
        dot(40)

makeTurtle()
hideTurtle()
n = 100
sum = 0

for k in range(n):
    init()
    notHit = 0
    for i in range(1, 11):
        zz = random.randint(1, 10)
        setPos(-280 + 50 * zz, 0)
        setPenColor("red")
        dot(40)
    for i in range(1, 11):
        setPos(-280 + 50 * i, 0)
        if getPixelColorStr() == "green":
            notHit += 1
    print notHit
    delay(200)
    sum += notHit

addStatusBar(20)
setStatusText("Mean not taken can:  " + str(sum/n))
```

MEMO

Statt ein Experiment in der Realität auszuführen, kannst du es mit dem Computer simulieren und beliebig oft wiederholen. Die Computersimulation ist meist wesentlich weniger aufwendig und damit auch kostengünstiger als das reale Experiment. Zudem lassen sich Versuchseinstellungen viel einfacher abändern und anpassen.

AUFGABEN

1. Zeichne Konfetti in verschiedenen Farben an zufällig gewählten Positionen.



2. Gehe von der Liste mit den Tonfrequenzen der C-Dur-Tonleiter aus:

   d = [262, 294, 330, 349, 392, 440, 494, 524]

   Spiele ein "Zufallslied", bei dem diese Töne zufällig mit einer zufälligen Dauer zwischen 100 ms und 1000 ms endlos abgespielt werden.

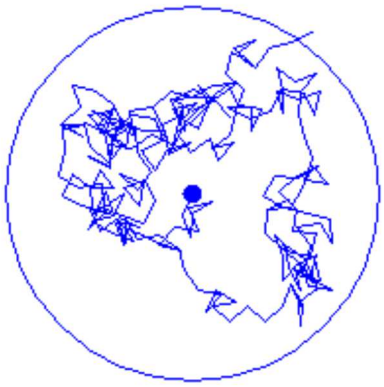
3. Bestimme mit der Monte-Carlo-Methode die Fläche der nebenstehenden grauen Figur. Verwende das unten stehende Programm, um die Figur mit grauer Farbe zu erzeugen und lasse die Zufallspunkte auf ein Quadrat, in welches diese Fläche eingebettet ist, fallen. Wie viel % der Quadratfläche beträgt die graue Fläche?

   Mit *getPixelColorStr(),* kannst du die Pixelfarbe an der aktuellen Turtleposition bestimmen. Wenn diese "gray" ist, so liegt der Punkt in der grauen Fläche und kann als Treffer erfasst werden.

```
from gturtle import *

makeTurtle()

setPenColor("gray")
setFillColor("gray")
setPos(0, 50)
startPath()
rightArc(100, 180)
rightArc(50, 180)
leftArc(50, 180)
leftArc(25, 360)
fillPath()
```

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

4. Bestimme wie viele Schritte im Mittel nötig sind, damit die Turtle bei einem *Random Walk* mit einer Schrittlänge von 20 und zufälliger Bewegungsrichtung vom Startpunkt die Entfernung 200 hat. Führe dazu die Simulation 100 mal aus.



5*. Versuche eine Gesetzmässigkeit herauszufinden, wie die Entfernung und die Schrittzahl zusammenhängen.

2.15 COMPUTERANIMATION

EINFÜHRUNG

Die Computergrafik kann ähnlich wie in einem Film zur Darstellung von zeitlich veränderlichen Inhalten verwendet werden. Man nennt ein so eingesetztes Programm eine Computeranimation. Um den zeitlichen Ablauf sichtbar zu achten, wird in einer Animationsschleife immer nach einem gleich grossen Zeitschritt ein neues Bild gezeichnet, das sich nur wenig vom vorhergehenden unterscheidet. Werden mehr als 25 Bilder pro Sekunde erzeugt, so ergibt sich für das Auge eine fliessende Bewegung.

PROGRAMMIERKONZEPTE: *Computeranimation, Animationsschleife, Doppelbufferung, Sprite*

FIGUREN BEWEGEN

Das Prinzip der Animation kannst du an einer sich scheinbar bewegenden Figur ausprobieren. Dabei zeichnest du die Figur in kurzen Zeitschritten an leicht verschobenen Positionen. Die vorangehende Figur musst du immer wieder löschen.

Im ersten Beispiel lässt du einen Propeller rotieren. Für die Wiederholung verwendest du eine endlose while-Schleife. In dieser Animationsschleife zeichnest du zuerst den Propeller. Danach wartest du 100 ms. Diese Wartezeit bestimmt, wie häufig der Propeller pro Sekunde neu gezeichnet wird, also in diesem Fall rund 10 Mal. Dann löschst du den Bildschirm mit clear() und drehst die Turtle um 20 Grad, damit der nächste Propeller in leicht gedrehter Lage gezeichnet wird.

```
from gturtle import *

def drawFigure():
    repeat 3:
        fillToPoint(0, 0)
        rightArc(100, 90)
        right(90)
        rightArc(100, 90)
        right(90)
        left(120)

makeTurtle()
hideTurtle()
setPenColor("black")

while True:
    drawFigure()
    delay(100)
    clear()
    right(20)
```

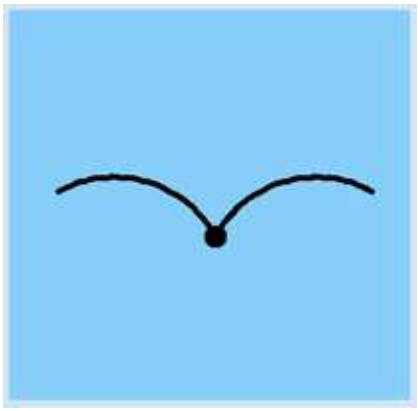
MEMO

In einer Animationsschleife zeichnest du die Figur periodisch an leicht verschobenen Positionen. Um die vorangehende Figur zu löschen, löschst du den ganzen Bildschirm.

BILDSCHIRMFLACKERN VERMEIDEN

Üblicherweise ist jeder Zeichnungsbefehl sofort auf dem Bildschirm sichtbar. Daher erscheint beim Löschen des Bildschirmes mit *clear()* kurz ein leeres Grafikfenster, was zu einem unschönen Bildschirmflackern führen kann. Um dies zu vermeiden, verwendest du die sogenannte Doppelbufferung. Du rufst den Befehl *enableRepaint(False)* auf, der bewirkt, dass die Grafikbefehle in einem unsichtbaren Bildbuffer ausgeführt werden und nicht mehr automatisch im Grafikfenster erscheinen. Erst wenn das neue Bild vollständig im Buffer aufgebaut ist, stellst du mit *repaint()* den ganzen Buffer auf einmal auf dem Bildschirm dar.

In deinem Beispiel zeigst du die Flügelbewegungen eines Vogels. Dieser besteht der Einfachheit halber aus einem runden Körper und zwei Kreisbögen, die links und rechts vom Körper gezeichnet werden. Um die Lage der Flügel einzustellen, drehst du die Turtle vor dem Zeichnen des Flügels in die passende Richtung. Die Funktion *bird(angle)* hat daher einen Parameter *angle*, der die Turtle-Richtung vor dem Zeichnen der Flügel angibt. Dieser Winkel steigt bei der Abwärtsbewegung von 5° bis 55° und fällt bei der Aufwärtsbewegung von 55° bis 5°. Für die Winkeländerung kannst du eine *for*-Schleife mit *step* = 2° verwenden.



Da du enableRepaint(False) verwendest, musst du jedesmal nach dem Zeichnen repaint() aufrufen, damit der Inhalt des Bildbuffers auf dem Bildschirm sichtbar wird. Zum Löschen des Bildschirmes verwendest du den Befehl clear("lightSkyBlue"). Dadurch erhältst du eine hellblaue Hintergrundfarbe.

```
from gturtle import *

def drawBird(angle):
    dot(20)
    right(angle)
    rightArc(100, 90)
    home()
    left(angle)
    leftArc(100, 90)
    home()

def move(start, end, step):
    for a in range(start, end, step):
        clear("lightSkyBlue")
        drawBird(a)
        repaint()
        delay(40)

makeTurtle()
hideTurtle()
setLineWidth(5)
setPenColor("black")
enableRepaint( False)
```

```
while True:
    move(55, 5, -2)
    move(5, 55, 2)
```

MEMO

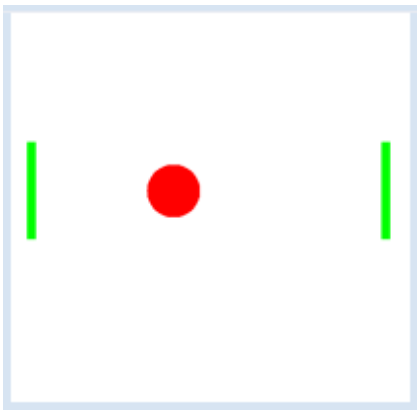
Der Befehl enableRepaint(False) bewirkt, dass die Grafikbefehle nicht automatisch im Turtlefenster sichtbar sind, sondern nur im Bildbuffer ausgeführt werden. Damit das neue Bild angezeigt wird, musst du aber an geeigneter Stelle repaint() aufrufen.

GLEICHMÄSSIGE BEWEGUNG

Die Animationsschleife sollte in möglichst gleichen Zeitticks durchlaufen werden, da sonst die Bewegung ruckelt. Die Operationen in der Animationsschleife können aber unterschiedlich viel Zeit in Anspruch nehmen, selbst wenn der Code immer derselbe ist, der Computer im Hintergrund noch mit anderen Aufgaben beschäftigt ist. Um die verschieden lange Ausführung der Animationsschleife auszugleichen, wird daher folgender Trick angewendet: Man merkt sich in der Variable startTime vor den Zeichnungsoperationen die aktuelle Systemzeit, die du als Dezimalzahl mit time.clock() erhältst.

Nach dem Zeichnen wartest du in einer Warteschleife so lange, bis die Differenz der neuen Systemzeit und der Startzeit die gewünschte Animationsperiode erreicht. (Damit der Computer dabei nicht unnütz viel Rechenzeit verschwendet, hältst du das Programm mit delay(1) kurz an.) Dieser Trick funktioniert natürlich nur, falls die Zeit für das Zeichnen kürzer als die Animationsperiode ist.

In einem Ping-Pong-Spiel willst du den roten Ball möglichst gleichmässig zwischen den grünen Balken hin und her bewegen.



```
from gturtle import *
import time

def wall():
    setPenColor("green")
    setLineWidth(10)
    setPos(-200, -50)
    forward(100)
    setPos(200, -50)
    forward(100)

makeTurtle()
hideTurtle()
enableRepaint(False)

x = -170
v = 10

while True:
    startTime = time.clock()
    clear()
    wall()
    setPos(x, 0)
    setPenColor("red")
    dot(60)
```

```
    repaint()
    x = x + v
    if x > 165 or x < -165:
        v = -v
    while (time.clock() - startTime)  < 0.020:
        delay(1)
```

MEMO

Ein gleichmässig bewegtes Bild erreichst du, indem du in gleichen zeitlichen Abständen die Figur an der neuen Position anzeigst. Dazu taktest du die Animation unter Verwendung der Systemzeit. Auch in diesem Beispiel wendest du das Prinzip der Doppelbufferung an, um das Flackern zu vermeiden.

TURTLE ALS LEITFIGUR FÜR SPRITES VERWENDEN

Statt das Bild mit der Turtle zu zeichnen, kannst du auch ein Computerimage verwenden, das als png-, gif- oder jpg-Datei vorliegt. In der Gameprogrammierung nennt man ein solches Bild ein Sprite und das bewegte Bildschirmobjekt einen Aktor. Um ein Sprite zu verwenden, rufst du getImage(path) auf und gibst dabei den Pfad zur Bilddatei an (als absoluter Dateipfad, als Dateipfad relativ zum Verzeichnis, in dem sich dein Programm befindet, oder als Internet URL).

Mit drawImage() stellst das Bild an der Position der Turtle mit ihrer Blickrichtung dar. Du kannst also sozusagen mit der Turtle als "Leitfigur" das Bild positionieren und drehen, da sich das Sprite zusammen mit der Turtle mitbewegt. Meist versteckst du dabei mit *hideTurtle()* die Leitfigur.

Im Beispiel verwendest du das Bild *car0.png* eines Autos aus der TigerJython-Distribution und bewegst es auf einem Kreis. Alle im Verzeichnis *sprites* verfügbaren Bilder kannst du aus der TigerJython Hilfe (APLU Dokumentation) entnehmen.

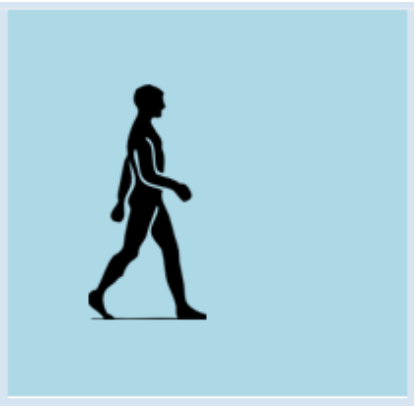
```
from gturtle import *
import time

makeTurtle()

hideTurtle()
setPos(-180, 0)
enableRepaint(False)
img = getImage("sprites/car0.png")

while True:
    startTime = time.clock()
    clear()
    drawImage(img)
    repaint()
    forward(3)
    right(1)
    while (time.clock() - startTime)  < 0.04:
        delay(1)
```

MEMO

Bewegte Figuren, Aktoren genannt, spielen in Computergames eine zentrale Rolle. Mit der Turtle kannst das Spritebild auf dem Bildschirm bewegen und rotieren.

ANIMIERTE AKTOREN

Eine Turtle kann als Leitfigur sogar mehrere verschiedene Spritebilder herumbewegen. Dieses Verfahren kommt dann zum Tragen, wenn du das Spritebild selbst animieren willst. Typisch ist eine laufende Person, die aus mehreren Bewegungsformen besteht. In der Animationsschleife bewegst du einerseits die Turtle vorwärts und änderst laufend das zugehörige, vorher geladene Sprite. Damit die Turtleposition ausserhalb des Fensters automatisch wieder auf der anderen Seite ins Fenster zurückgesetzt wird, verwendest du den Befehl wrap().



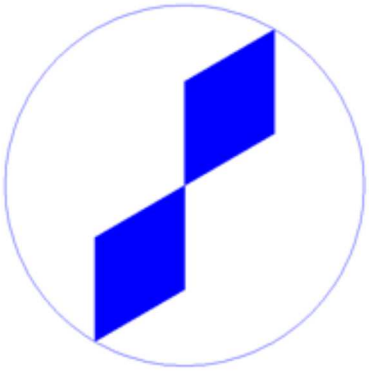
```
from gturtle import *
import time

makeTurtle()
hideTurtle()
wrap()
enableRepaint(False)

img = [0] * 5
for i in range(5):
    img[i] = getImage("sprites/person_" + str(i) + ".png")

right(90)
i = 0
while True:
    startTime = time.clock()
    clear("lightblue")
    drawImage(img[i])
    repaint()
    forward(10)
    i += 1
    if i == 5:
        i = 0
    while (time.clock() - startTime)  < 0.1:
        delay(1)
```

MEMO

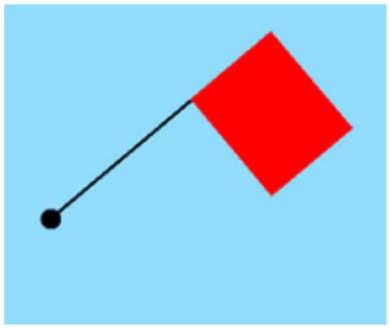
Du kannst eine Computeranimation auf verschiedene Arten programmieren. Am einfachsten ist, es, das Bild der Turtle zu ersetzen, indem du in *makeTurtle()* einen Bildpfad angibst. Du kannst sodann mit Grafikbefehlen ein sich veränderndes Bild zeichnen oder schliesslich die Turtle als Leitfigur dazu verwenden, geladene Bilder zu bewegen. Willst du eigene Spritebilder einbinden, so müssen sie in der gewünschten Pixelgrösse im *png-, gif-* oder *jpg*-Format erstellt sein. Sie sollten einen transparenten Hintergrund haben, damit sie auch in einem Fenster mit farbigem Hintergrund verwendet werden können.

AUFGABEN

1. Zeichne eine Figur, die aus zwei Rauten und einem Kreis besteht und lasse sie wie ein Ventilator um den Punkt (0, 0) rotieren. Verwende *enableRepaint(False)* und *repaint(),* um das Flackern zu beseitigen.

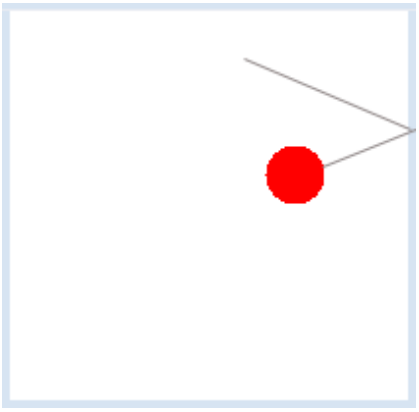


2. Zeichne eine Fahne und schwenke sie um den untersten Punkt der Fahnenstange hinauf und hinunter.

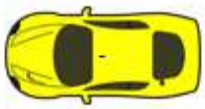


3. Eine Billardkugelbewegt sich im Turtlefensters so, dass sie an den Rändern jeweils unter der Berücksichtigung der Regel *Einfallswinkel = Ausfallswinkel* reflektiert wird.

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



4. Ergänze das Programm mit dem fahrenden Auto so, dass dieses mit den Cursotasten up, down, left und. right gesteuert werden kann. Als Sprite kannst du den gelben Ferrari (*car3.png*) aus der TigerJython-Distribution verwenden.



5. Das reizende Kätzchen soll immer wieder von rechts nach links über das Turtlefenster laufen. Verwende dazu die 8 Spritbilder *cutecat_0.gif,...cutecat_7.gif* aus der TigerJython-Distribution.

   Erstelle ähnliche lustige Animationen mit eigenen Bildern.

Dokumentation Turtlegrafik

Module import: from gturtle import *

Funktion	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
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

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
getPlaygroundWidth()	gibt die Breite m des Turtlefensters zurück (Turtlekoordinaten x = -m/2 ... m/2)
getPlaygroundHeight()	gibt die Höhe n des Turtlefensters zurück (Turtlekoordinaten y = -n/2 ... n/2)

Positionieren

distance(x, y)	gibt die Entfernung der Turtle zum Punkt(x, y) zurück

distance(coords)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
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(x, y)	gibt die Richtung (in Grad) zur Position (x, y) (auch list, tuple, complex)
towards(coord)	dasselbe, aber Koordinatenangabe als Liste, Tupel oder komplexe Zahl
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
clean()	löscht die Zeichnung und versteckt alle Turtles, Turtles bleiben am Ort
clean(color)	löscht die Zeichnung, versteckt alle Turtles, färbt den Hintergrund, Turtles bleiben am Ort
clear()	löscht die Zeichnung, Turtles bleiben am Ort
clear(color)	löscht die Zeichnung und färbt den Hintergrund, Turtles bleiben am Ort
clearScreen(), cs()	löscht die Zeichnung und setzt die Turtle an die Homeposition
dot(diameter)	zeichnet einen mit Stiftfarbe gefüllten Kreis
openDot(diameter)	zeichnet einen nicht gefüllten Kreis
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 (auch list, tuple, complex)

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
getPixelColorStr()	gibt die Farbe des Pixels an der Turtlekoordinate als X11-Farbe zurück
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.)
makeTurtle(closeClicked = onCloseClicked)	registriert die Callbackfunktion onCloseClicked(), die beim Klick des Close-Buttons des Turtlefensters aufgerufen wird. Das Fenster wird mit dispose() geschlossen
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

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, blocking = False)	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)...])	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(text)	schreibt Text an der aktuellen Position
printerPlot(draw)	druckt die mit der Funktion draw erstellte Zeichnung
setFont(Font font)	legt Schriftart fest
setFontSize(size)	legt Schriftgrösse fest
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

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 beendet das Programm nicht, 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 beendet das Programm nicht, 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 beendet das Programm nicht, gibt None zurück
input(prompt)	öffnet einen modalen Dialog mit OK/Abbrechen-Buttons. OK gibt Eingabe als Integer, Float oder String zurück. Abbrechen beendet das Programm
input(prompt, False)	dasselbe, aber Abbrechen beendet das Programm nicht, 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, gibt None zurück

KONTAKT

help@tigerjython.com

Entwicklungsteam:	Jarka Arnold, Pädagogische Hochschule Bern www.java-online.ch Tobias Kohn, Kantonsschule Zürich Oberland www.tobiaskohn.ch Dr. Aegidius Plüss, Universität Bern www.aplu.ch

Über die Autoren

Jarka Arnold
Jarka Arnold besitzt langjährige Erfahrung als Dozentin für Informatik an der Pädagogischen Hochschule Bern. Im Rahmen von mehreren Forschungsprojekten wurden unter ihrer Leitung webbasierte Lernumgebungen für den Programmierunterricht entwickelt, die an vielen Ausbildungsinstitutionen erfolgreich im Einsatz sind (http://www.java-online.ch bzw. http://www.jython.ch).

Tobias Kohn
Tobias Kohn (http://www.tobiaskohn.ch/) Tobias Kohn hat 2008 an der ETH Zürich sein Mathematikstudium abgeschlossen und arbeitet seither als Mathematik- und Informatiklehrer an der Kantonsschule Zürcher Oberland in Wetzikon. Im Herbst 2012 hat er neben seiner Lehrtätigkeit ein Doktoratsstudium an der ETH Zürich aufgenommen und sucht dabei nach Wegen, den Einstieg in die Computerprogrammierung zu vereinfachen.

Aegidius Plüss
Aegidius Plüss (http://www.aplu.ch) ist ehemaliger Professor für Informatik und deren Didaktik an der Universität Bern. Er hat ein Lehrbuch "Java exemplarisch" verfasst und war an zahlreichen Weiterbildungskursen für Informatik-Lehrpersonen beteiligt. Er entwickelt umfangreiche Bibliotheken und Programmierumgebungen für den Informatikunterricht.