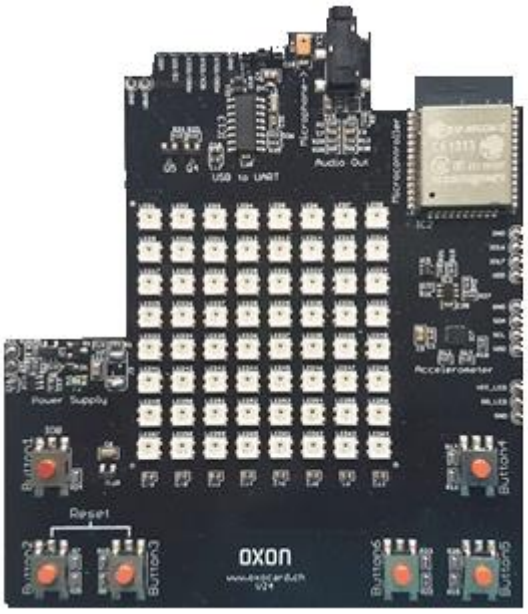
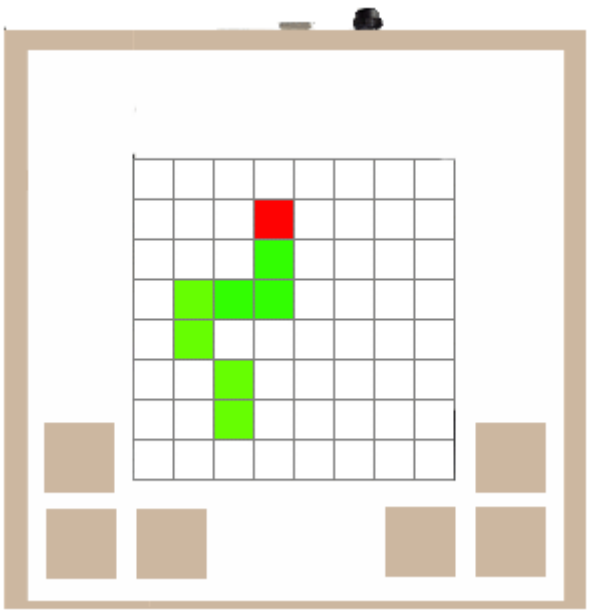
TigerJython4Kids

www.tigerjython4kids.ch

ROBOTIK mit OXOCARD

Jarka Arnold
Aegidius Plüss





Internet of Things (IoT)

www.tigerjython4kids.ch

INHALT

ROBOTIK MIT OXOCARD

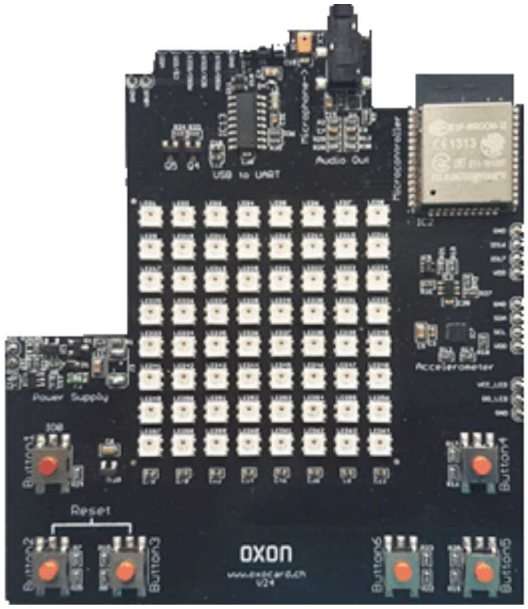
ARBEITSBLÄTTER:

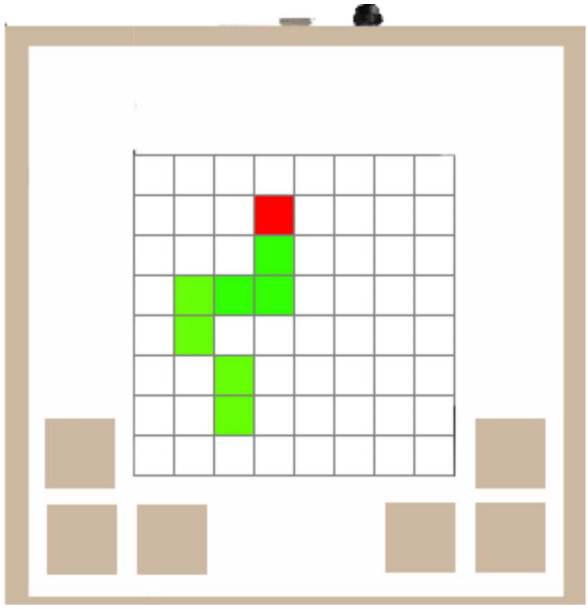
ANHANG:

Version 1.0, August 2018

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

Die Oxocard ist ein in der Schweiz entwickelter programmierbarer Computer, der in einer Kartonbox verpackt ist (Bezugsquelle: www.oxocard.ch). Er besteht aus einer ca. 10 x 10 cm grossen Platine mit einem **ESP32 Microcontroller**, Flash Speicher, ColorLEDs (Neopixels), mit welchen du einfache Bilder und Nachrichten anzeigen kannst, 6 Buttons und einer USB-Schnittstelle.





Die Oxocard ist auch in der Lage, mit dem **Beschleunigungssensor** ihre Neigung und Bewegungen (z.B. Schütteln) zu erkennen und kann über **WLAN** mit anderen Geräten kommunizieren. Am integrierten **Audioanschluss** kannst du einen Kopfhörer oder Lautsprecher anschliesst, um Töne, Tonfolgen und kurze Melodien abzuspielen. Weitere Aktoren und Sensoren. kannst du über die Pins anschliessen. Dies bedingt aber etwas Lötarbeit.

Für die Entwicklung der Oxocardprogramme stehen dir zwei Möglichkeiten zur Verfügung:

- Entwicklungsumgebung TigerJython
- Online-Editor

TigerJython ist eine schülergerechte Entwicklungsumgebung mit einer ausgezeichneten Korrekturhilfe, die auch die Programmentwicklung für die Oxocard vollständig unterstützt. Wie du deine Programme mit TigerJython editieren und auf die Oxocard herunterladen kannst, findest du unter dem Menüpunkt "Loslegen".

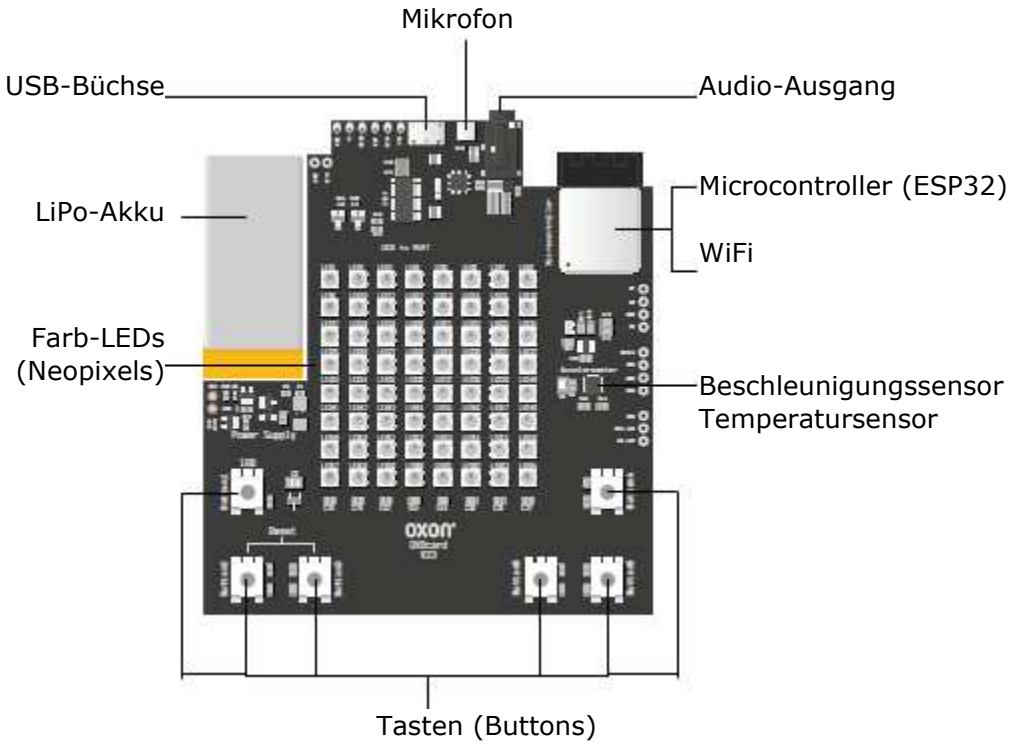
Beim **Online-Editor** sind keine lokalen Installationen notwendig. Du entwickelst deine Oxocardprogramme in einem Browserfenster und kannst sie über das WLAN auf die Oxocard downloaden. Eine Anleitung dazu findest du unter dem Menüpunkt Online-Editor.

In den ersten Kapiteln des Lernprogramms kannst du die wichtigsten Programmstrukturen kennen lernen oder sie repetieren, wenn du bereits über Grundkenntnisse in Python verfügst. In den weiteren Kapiteln findest du viele weiterführende Musterbeispiele zur Verwendung der LEDs, des Beschleunigungssensors, der WLAN-Kommunikation, sowie Programmcodes einiger Games.

SIMULATIONSMODUS

Falls du keine Oxocard hast oder wenn du dein Programm zuerst auf dem PC testen möchtest, so kannst du den Simulationsmodus verwenden, bei dem dir allerdings nur ein Teil der Hardware-Optionen zur Verfügung steht.

OXOCARD HARDWARE



1. LOSLEGEN

DU LERNST HIER...

wie du eine Oxocard für das Programmieren mit Python vorbereiten musst und wie du deine Programme editieren und auf der Oxocard ausführen kannst.

PROGRAMMENTWICKLUNG

Die Programme werden mit ***TigerJython*** auf einem Computer entwickelt, dann auf die Oxocard hinuntergeladen und dort mit ***MicroPython*** (einer reduzierten Python-Version) ausgeführt. Der Programmdownload erfolgt über ein USB-Kabel, das gleichzeitig für die Stromversorgung der Oxocard sorgt. Eine Anleitung zur Installation von *TigerJython* findest du unter Einrichten. Wir empfehlen dir, jeweils die neuste *TigerJython*-Version zu verwenden.

USB-VERBINDUNG

Verbinde die Oxocard mit einem USB-Kabel mit dem Computer (Mac oder Windows).

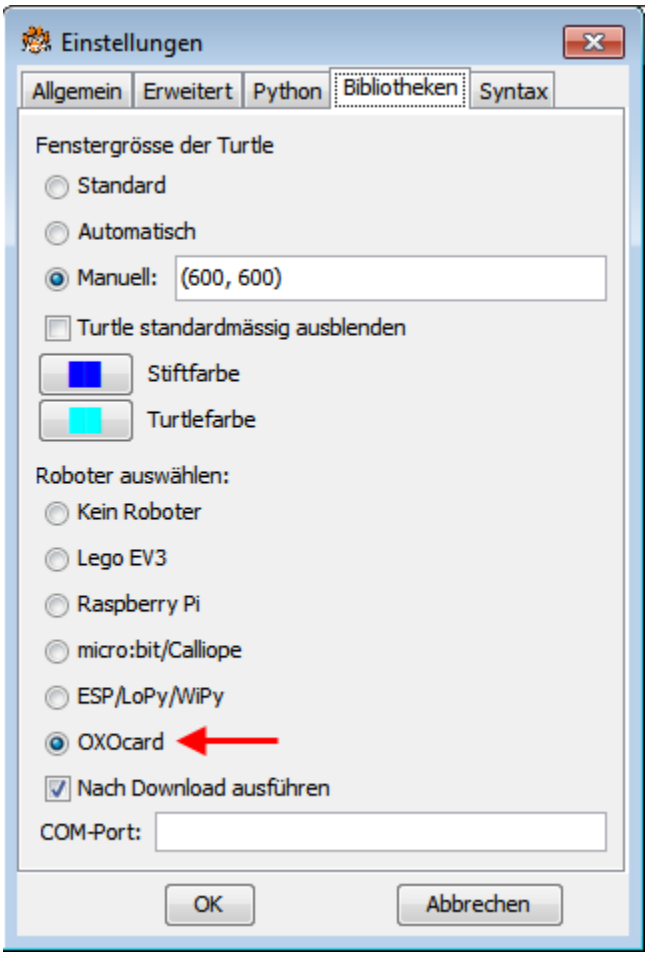
- Windows 10 erkennt das USB-Gerät automatisch. Es ist keine Treiber-Installation nötig
- Bei Windows 7, 8 und Mac musst du einen USB-Treiber installieren. Lade die Treiberdatei für MacOS oder Windows hinunter, packe sie aus und führe sie per Mausklick aus. Dazu brauchst du allerdings Administrator-Rechte.

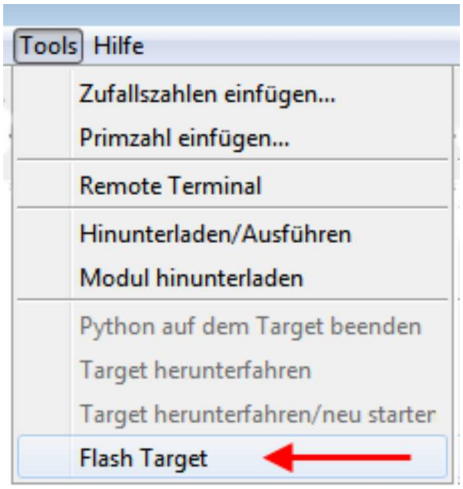
Du kannst prüfen, ob die Karte richtig erkannt wird
- unter Windows: im Geräte-Manager unter Anschlüsse(COM&LPT)
- unter MacOS: im Terminal mit dem Befehl ls /dev/tty.*

FIRMWARE INSTALLIEREN

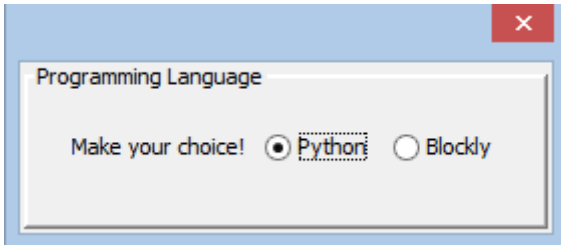
Vor der ersten Verwendung musst du eine Firmware, welche insbesondere den MicroPython-Interpreter und einige speziell für die Oxocard geschriebene Module enthält, auf die Oxocard hinunterladen.

Mit TigerJython
Wähle unter *Einstellungen/Bibliotheken* die Option *Oxocard*. Das Eingabefeld COM-Port kannst du leer lassen, da der COM-Port normalerweise automatisch gefunden wird. Diese Einstellung bleibt gespeichert und ist auch für das Hinunterladen der Programme erforderlich. Wähle im Menü unter *Tools* die Option ***Flash Target***.





Wähle in der dann erscheinenden Dialogbox die **Option "Python"** und die Python-Firmware wird installiert und von uns entwickelte Bibliotheksmodule werden auf die Oxocard hinuntergeladen.



Willst du die Oxocard später wieder auf die Fabrikeinstellungen zurückstellen, so wählst du die Option "Blockly".

Mit OxoFlash

Mit einem Windows- oder Mac-Computer kannst du jederzeit auch ohne TigerJython die Oxocard neu flashen. Dazu lädst du die Applikation *OxoFlash* herunter und führst sie aus:.
Für Windows (http://www.tigerjython4kids.ch/download/OxoFlash.msi)
Für MacOS (http://www.tigerjython4kids.ch/download/OxoFlash.dmg)
(Für MacOS musst du zuerst das ebenfalls vorhandene Programm *unlock* mit rechtem Mausklick im Terminal ausführen)

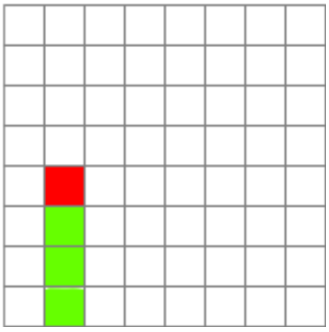
Das Programm setzt voraus, dass eine USB-Verbindung zur Oxocard gemäss vorhergehendem Abschnitt eingerichtet ist. Es erscheint wiederum eine Auswahlbox, in der du entweder Python oder Blockly auswählen kannst.

MUSTERBEISPIEL

Zum Einstieg schreibst du ein Programm, welches auf dem Display, den wir LedGrid nennen, eine einfache Schlangenfigur anzeigt. Starte TigerJython und tippe im Editorfenster das unten stehende Programm ein.

```
from oxosnake import *

makeSnake()
```

Kontrolliere, ob die Oxocard am Computer angeschlossen ist und klicke auf die Schaltfläche *Hinunterladen/Ausführen*.



Wenn auf dem Display die nebenstehende Figur erscheint, funktioniert deine Oxocard einwandfrei.

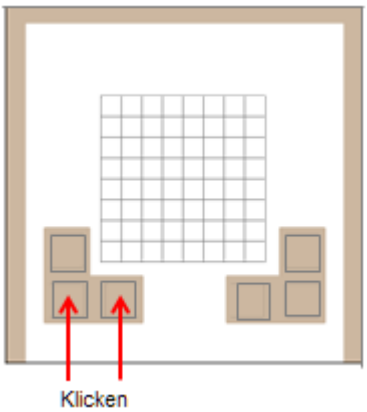
Zum Programmcode:
Mit der ersten Programmzeile importierst du das Modul *snake*.
Mit der zweiten Zeile wird ein Snake-Objekt erzeugt und angezeigt. Im nächsten Kapitel lernst du, wie du die Schlange bewegen kannst.

Die vier zuletzt hinuntergeladenen Programme bleiben auf der Oxocard gespeichert. Bei einem Reset der Oxocard oder beim Anschluss einer Stromversorgung über das USB-Kabel startet das zuletzt hinuntergeladene Programm.

Neben dem TigerJython-Fenster erscheint jeweils bei der Programmausführung ein zweites Fenster, das sogenannte **Terminal-Fenster** (manchmal auch REPL/Console). Hier werden Mitteilungen und Fehlermeldungen angezeigt und du kannst hier auch jederzeit eine Programmausführung mit **Ctrl+C** abbrechen. Willst du das Programm neu starten, so drückst du die beiden Reset-Tasten (oder Ctrl+D).

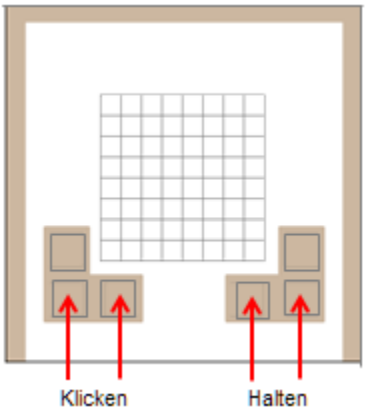
OXOCARD RESETTEN/EINSCHALTEN

Die Oxocard wird über eine spezielle Tastenkombination neu gestartet. Man drückt dazu gleichzeitig kurz die beiden linken Tasten.



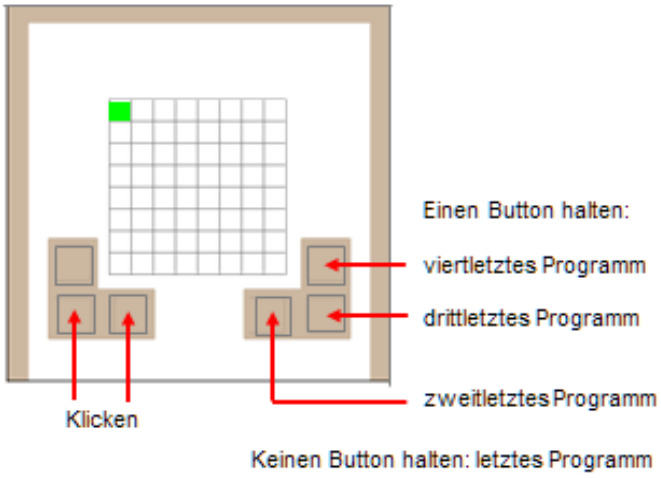
OXOCARD AUSSCHALTEN (SLEEP MODUS)

Die Oxocard hat keinen Ein-/Ausschalter, sondern wird über eine spezielle Tastenkombination heruntergefahren. Du hältst dazu die beiden Tasten rechts unten gedrückt und klickst die Reset-Tasten kurz. Du hältst die beiden rechten Tasten weiter, so lange bis ein rotes Kreuz aufleuchtet und langsam verschwindet. Jetzt ist die Oxocard im "Schlafmodus" und hat nur noch einen sehr kleinen Stromverbrauch.



LETZTE 4 PROGRAMME STARTEN

Nach dem Download und bei einem Reset wird immer das zuletzt hinuntergeladene Programm gestartet. Wenn du aber beim Resetten einen der Buttons auf der rechten Seite gedrückt hältst, so wird das zweitletzte, drittletzte oder viertletzte hinuntergeladene Programm gestartet. Der Programmstart wird durch ein kurzes Aufblicken der grünen LED oben links angezeigt.

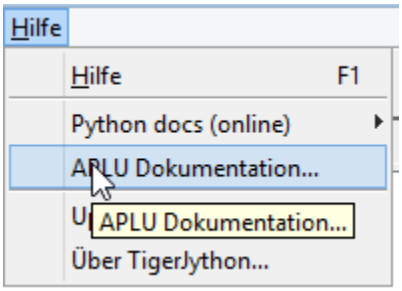


OXOCARD AUFLADEN

Die Oxocard wird jedesmal aufgeladen, wenn du eine Stromversorgung an der USB-Buchse anschliesst, indem du diese mit einem Computer, einem Standard-USB-Ladegerät oder einer USB-Powerbank verbindest.

OXOCARD API DOKUMENTATION

Du findest eine Beschreibung der speziell für die Oxocard entwickelten Module unter dem Menü-Eintrag Hilfe | APLU Dokumentation



SIMULATIONSMODUS

Um ein Programm im Simulationsmodus auszuführen, klickst du den **grünen Startbutton**.

Zur Zeit werden im Simulationsmodus alle Befehle von *oxosnake*, *oxocard*, *oxobutton* und *oxoaccelerometer* unterstützt. Es erscheint dann ein Simulationsfenster, das du mit gedrückter Maustaste verschieben kannst. Beim Start des nächsten Programms wird das vorhergehende Fenster automatisch geschlossen.

MERKE DIR...

Du schreibst ein Programm für die Oxocard im TigerJython-Editor. Um das Programm auf der Oxocard auszuführen, klickst du auf die Schaltfläche *Hinunterladen/Ausführen*. Die Programmausführung im Simulationsmodus wird mit Klick auf den grünen Pfeil gestartet.

Simulation Download auf Oxocard

Vergiss nicht, die Karte in den Sleep-Modus zu versetzen, wenn du sie nicht mehr brauchst.

ONLINE-EDITOR VERWENDEN

DU LERNST HIER...

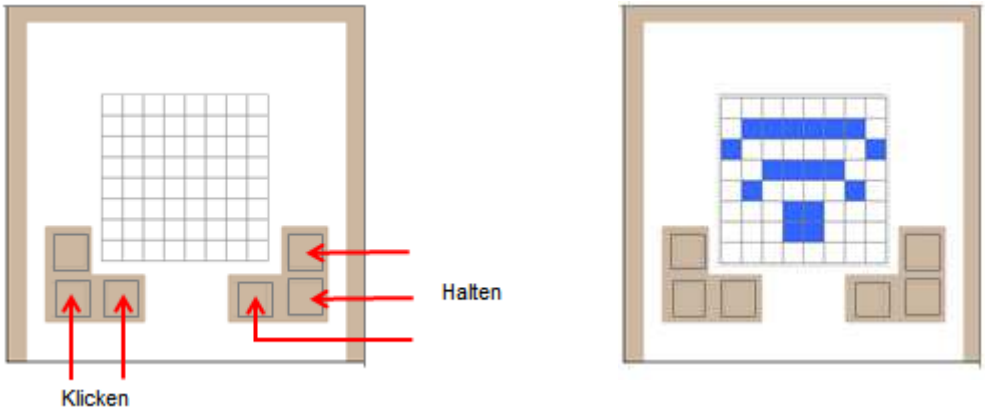
wie du für die Python-Programme mit einem webbasierten Editor entwickeln kannst. Dabei sind keine lokalen Installationen erforderlich und du brauchst auch kein USB-Kabel. Die Programme werden in einem Browser-Fenster editiert, auf dem Webserver bereitgestellt, über das WLAN auf die Oxocard heruntergeladen und dort ausgeführt.

Der Oxoeditor ist eine Alternative zum TigerJython-Editor, eignet sich aber eher für das Editieren von einfacheren Oxocard Programmen und bietet nicht so komfortable Benutzeroberfläche und Korrekturhilfe wie die lokal installierte TigerJython-Entwicklungsumgebung.

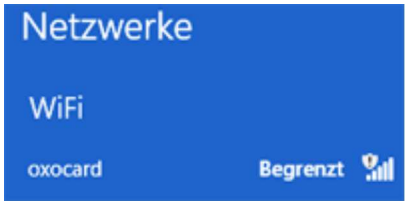
OXOCARD KONFIGURIEREN

Damit du Programme für die Oxocard mit dem Online-Editor entwickeln kannst, muss sich die Oxocard im Bereich eines WiFi Hotspots (Accesspoint) befinden. Vor der ersten Verwendung musst du sie konfigurieren, damit sie sich mit diesem Hotspot verbinden kann. Gehe wie folgt vor:

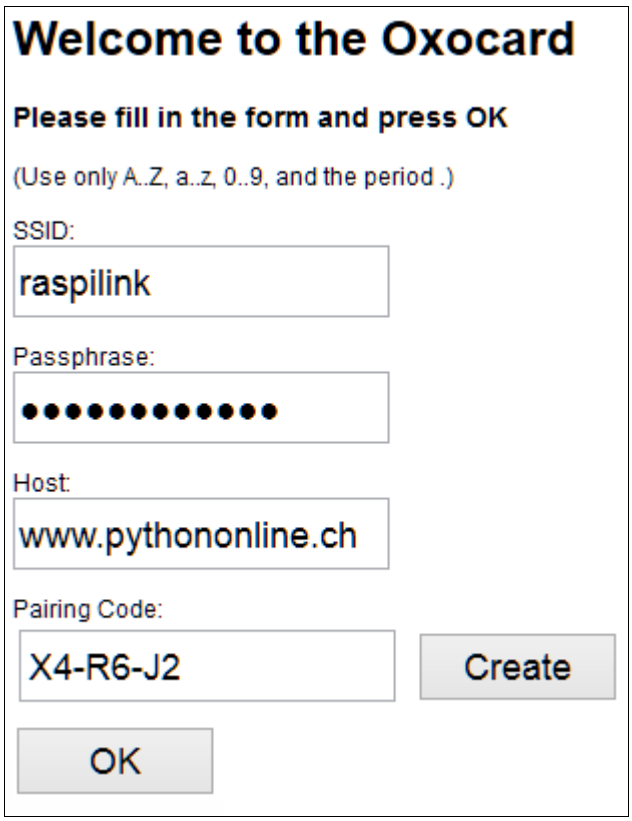
1. Halte die **drei** rechten Buttons und drücke gleichzeitig kurz die unteren zwei linken Buttons. Dann lasse die drei rechten Buttons wieder los. Die Oxocard started neu und aktiviert einen Hotspot mit der SSID *oxocard*. Es erscheint ein blinkendes, blaues WiFi-Symbol.



2. Wähle auf deinem PC das WiFi-Netz mit dem Namen *oxocard.* Du brauchst kein Passwort (leer lassen).



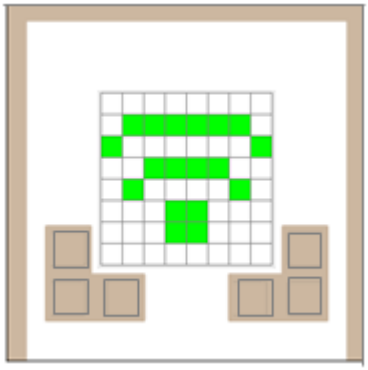
3. Starte einen Webbrowser auf dem PC und gib die Webadresse **192.168.4.1** ein. Die Oxocard meldet sich mit einer Eingabemaske.



SSID und Passphrase sind Angaben von deinen Hotspot, die du auch sonst brauchst. Als Host wählst du unseren Webserver *www.pythononline.ch*.

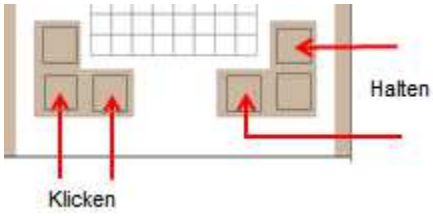
Zur Identifikation muss die Oxocard einen "Paarungscode" erhalten. Damit dieser eindeutig ist, wird er von der Karte selbst erzeugt, wenn du den Button *Create* klickst. Schreibe diesen Code auf, da du ihn später im Online-Editor brauchst.

Sobald du die Eingabe mit OK bestätigst, wird das WiFi-Symbol auf der Oxocard grün. Die Konfiguration bleibt auf der Oxocard gespeichert.



Falls du etwas daran ändern willst, gehst wieder gleich vor. Dabei brauchst du nur das einzugeben, was du ändern willst (beispielsweise kannst du den Paarungscode unverändert lassen).

Falls du den Paarungscode vergessen hast, so kannst du die beiden diagonal liegenden Buttons auf der rechten Seite halten und die beiden Buttons unten links klicken. Der Code wird auf der Anzeige als Lauftext ausgeschrieben.



4. Wechsle auf dem Computer in den WiFi-Einstellungen zu deiner üblichen Netzwerkverbindung, damit du wieder mit dem Internet verbunden bist.

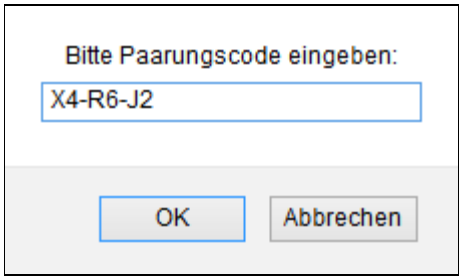
PROGRAMM EDITIEREN UND BEREITSTELLEN

Du kannst irgendein Gerät (PC, Tablett, Handy) verwenden, das mit dem Internet verbunden ist. Starte einen Webbrowser und wähle die Adresse:

www.pythononline.ch/oxocard

Du wirst mit dem Online-Editor verbunden und in deinem Browser erscheint ein Editorfenster.

Du musst als erstes dem Editor mitteilen, mit welcher Oxocard du programmieren willst. Als Identifikation verwendes du ihren Paarungscode. Um ihn anzugeben, klickst du auf den Button *Eingeben*, gibst im Dialog den Paarungscode ein und klickst OK.



Der Paarungscode wird in deinem Browser (als Cookie) gespeichert. Falls du eine andere Oxocard verwenden willst, kannst ihn jederzeit durch Klicken auf *Ändern* neu eingeben.

Du kannst nun im Editor ein Programm eintippen, mit *Kopieren/Einfügen* ein bestehendes Programm einfügen oder ein auf deinem Gerät gespeichertes Programm mit *Öffnen* in den Editor übernehmen

Als Test tippst du ein::

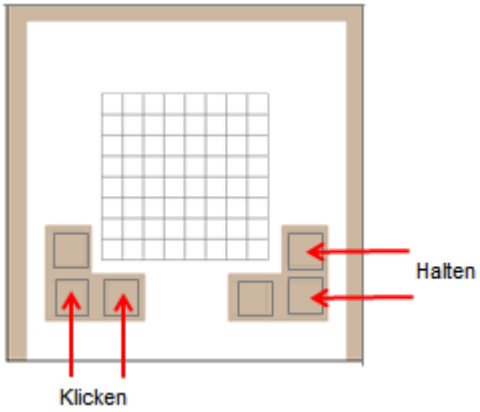
```
from oxocard import *
bigTextScroll("Hallo Anna", RED)
```

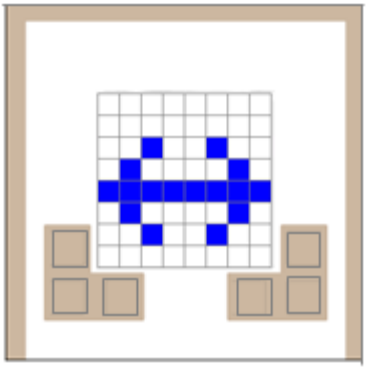
Klicke auf ***Bereitstellen***. Das Programm wird dabei auf dem Host zur Ausführung auf der Oxocard bereitgestellt.

Um den Programmcode **auf dem PC** zu **speichern**, klickst du auf *Speichern*. Das Programm wird als Python-Datei herunteruntergeladen und in der Regel im Download-Verzeichnis gespeichert (abhängig von den Einstellungen auf deinem PC).

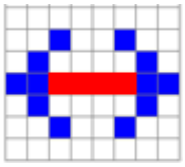
PROGRAMM AUF DIE OXOCARD DOWNLOADEN UND AUSFÜHREN

Halte die beiden rechten Buttons und klicke die linken unteren Buttons. Es erscheint ein blaues Download-Symbol und nach kurzer Wartezeit startet das Programm.





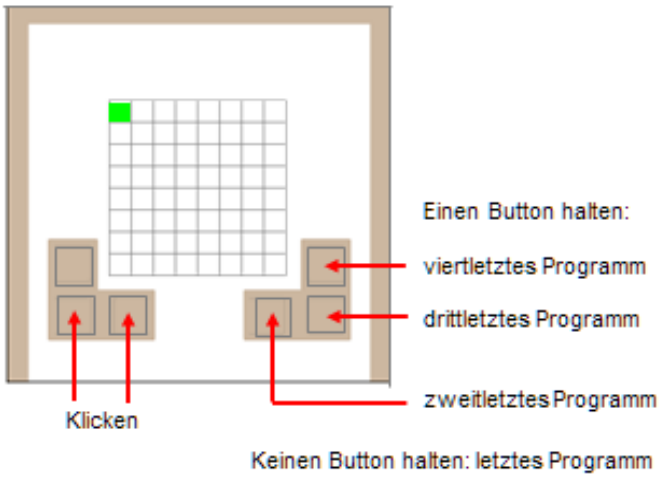
Bemerkung:
Falls sich die Oxocard nicht am Hotspot anmelden kann, so wird eine Fehlermeldung ausgeschrieben. Falls es noch kein Programm mit dem richtigen Paarungscode gibt oder die Verbindung zum Server misslingt, so erscheint nebenstehendes Fehlersymbol.



EINES DER LETZTEN 4 PROGRAMME STARTEN

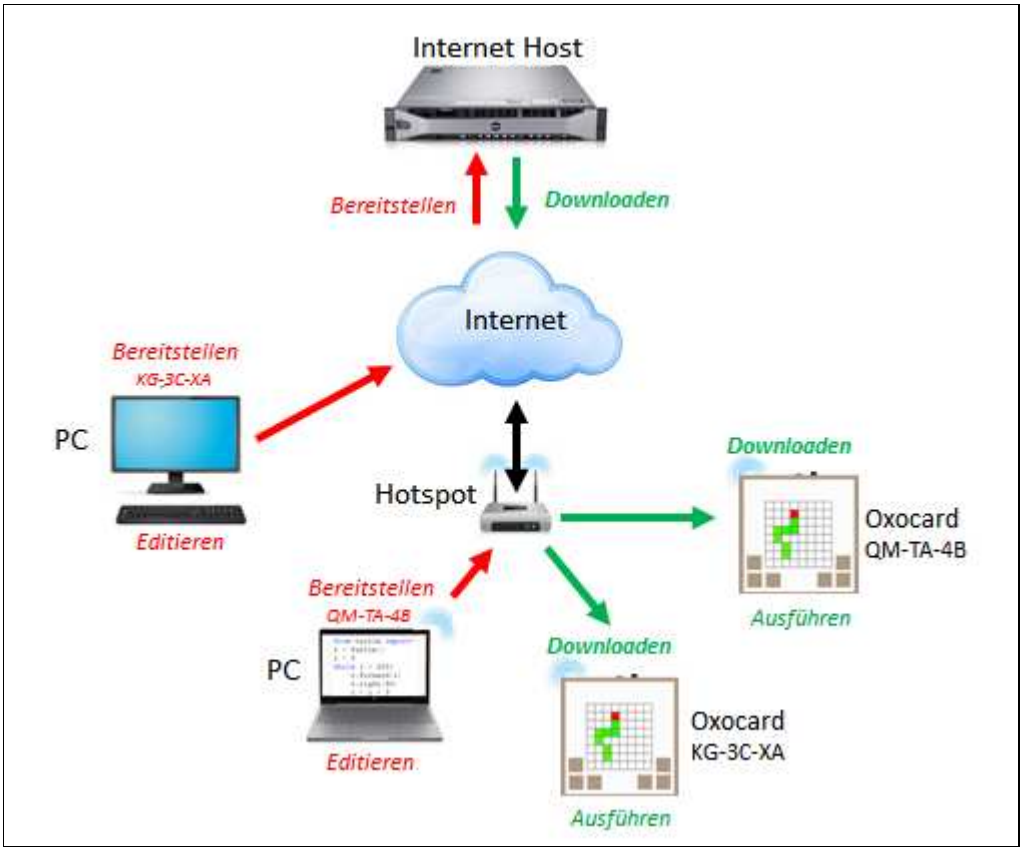
Die vier zuletzt gestarteten Programme bleiben auf der Oxocard gespeichert und können auch ohne Verbindung zum Internet wieder gestartet werden.

Bei einem Reset der Oxocard mit den beiden linken Button wird immer das zuletzt ausgeführte Programm gestartet. Willst du das zweitletzte, drittletzte oder viertletzte Programm starten, musst du beim Resetten gleichzeitig einen der rechten Buttons gedrückt halten. Jeder Programmstart wird durch ein kurzes Aufblicken der grünen LED oben links eingeleitet.



ZUSATZINFORMATIONEN:

Schematischer Ablauf des Online-Editors

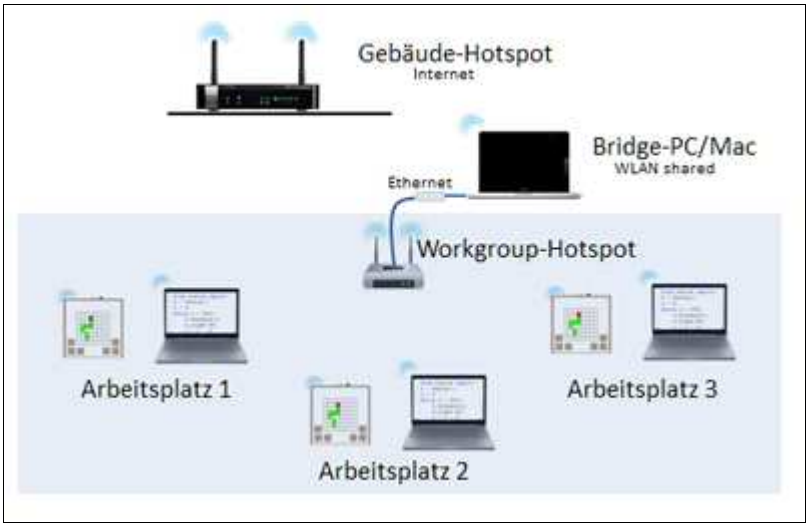


Netzinfrastruktur für eine Workgroup

In Ausbildungsinstitutionen ist es oft nicht erlaubt, sich mit SSID/Passwort direkt mit dem Internet zu verbinden. Es gibt zwei Möglichkeiten für die Workgroup-PCs/Oxocards eine Netzinfrastruktur aufzubauen, um die notwendige Verbindung zum Internet zu realisieren:

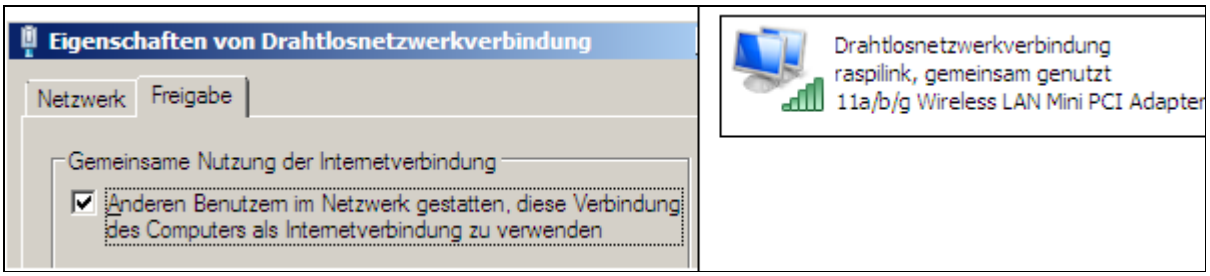
- Verwendung eines WiFi-Routers mit SIM-Karte und einem Daten-Flat-Abonnement
- Verwendung eines Bridge-PC/Mac, der sich auf dem Gebäudenetz mit den notwendingen Anmeldeoptionen verbindet und seine WiFi-Verbindung für den Ethernet-Port freigibt (falls kein Ethernet-Port eingebaut ist, über einen USB-Ethernet-Adapter).

Schematisch:



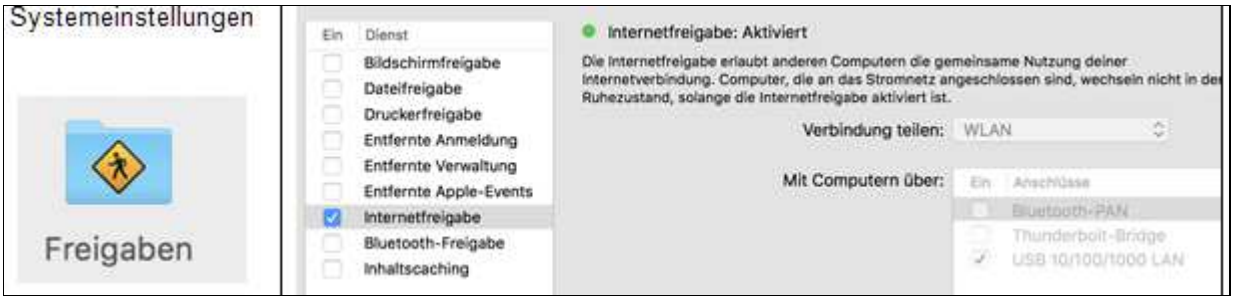
Einstellungen auf dem dem Bridge-PC:

Windows: Unter Netzwerk-Adapteroptionen für den WLAN-Adapter unter Eigenschaften



Bemerkung: Nicht alle eingebauten WLAN-Adapter unterstützen die "Gemeinsame Nutzung" (Internet Connection Sharing). Tippt man in einem Kommandofenster *netsh wlan show driver* ein, so sollte unter *"Unterstütze gehostete Netzwerke"* der Eintrag *Ja* stehen. Ist das nicht der Fall, so kann ein externer USB-WLAN-Adapter das Problem lösen.

MacOS:



2. SNAKE-OBJEKT

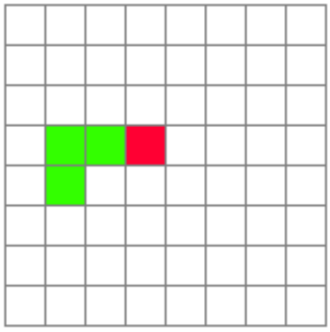
DU LERNST HIER...

dass ein Programm aus einer Folge von Programmzeilen besteht, die der Reihe nach (als Sequenz) abgearbeitet werden. Dabei verwendest du eine kleine Python-Schlange (***Snake)*** aus leuchtenden LEDs, die du mit den Befehlen *forward(), left()* und *right()* auf dem LedGrid bewegen kannst.

MUSTERBEISPIELE

Damit du die Snake-Befehle verwenden kannst, importierst du das Modul ***oxosnake***, welches nach dem Konzept der objektorientierten Programmierung (**OOP**) aufgebaut ist. Mit dem Befehl makeSnake() erzeugst du ein softwaremässiges Schlangenobjekt, das wie eine natürliche Schlange **Eigenschaften** hat (Länge, Position des Kopfes, Blickrichtung, Farbe des Kopfes, Farbe der Schwanzelemente usw.). Die Schlage besitzt aber auch **Fähigkeiten**. So kann sie sich beispielsweise mit dem Befehl forward() um einen Schritt vorwärts bewegen. In der Parameterklammer kannst du auch die Anzahl Schritte eingeben. Teile der Schlange, die ausserhalb des 8x8 pixel grossen sichtbaren Teils des LedGrids liegen, bleiben unsichtbar, werden aber nicht gelöscht.

Mit right(90) dreht die Schlange um 90 Grad nach rechts und mit **left(90)** um 90° nach links. Dies macht sich aber erst beim nächsten *forward()*-Befehl bemerkbar. Du kannst auch andere Drehwinkel wählen, diese werden aber jeweils auf ein Vielfaches von 45° gerundet.



Die Befehle werden grundsätzlich Englisch geschrieben und enden immer mit einer Parameterklammer. Diese kann Werte für den Befehl enthalten. Die Gross-/Kleinschreibung musst du exakt einhalten.

```
from oxosnake import *

makeSnake()

forward(2)
right(90)
forward(2)
```

Du kannst das Programm eintippen oder aus der Vorlage kopieren. Dazu klickst du auf *Programmcode markieren* und kopierst das Programm mit *Ctrl+C* in den Zwischenspeicher und fügst es mit *Ctrl+V* in das TigerJython-Fenster ein.

Um das Programm auf die Oxocard herunterzuladen und dort auszuführen, klickst du in der Taskleiste auf den schwarzen Button *(Hinunterladen/Ausführen*).

LEDs DIMMEN

Die LEDs leuchten sehr stark. Insbesondere wenn du die Oxocard nicht in der Kartonbox eingebaut hast, ist es vorteilhaft, die Helligkeit zu reduzieren. Du kannst auch in deinem Programm beim Erzeugen des Snake-Objekts die Helligkeit festlegen. Mit mit makeSnake(dim = 20) reduzierst du die Helligkeit um den Faktor 20.

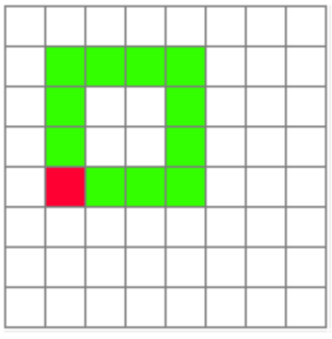
```
from oxosnake import *

makeSnake(dim = 20)

forward(2)
right(90)
forward(2)
```

MERKE DIR...

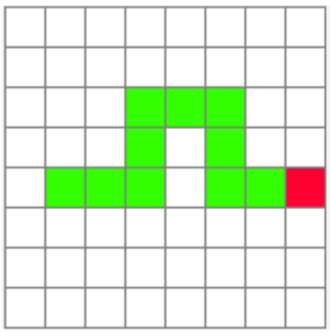
>Du musst unterscheiden zwischen dem Schreiben (Editieren) des Programms und seiner Ausführung. Das Programm wird auf deinem Computer editiert und dann auf die Oxocard heruntergeladen und dort mit MicroPython ausgeführt. Mit *makeSnake()* erzeugst du ein Snake-Objekt, welches du mit Befehlen aus dem Modul *snake* bewegen kannst.

ZUM SELBST LÖSEN

1. Die Schlage soll sich auf einem Quadrat mit der Seitenlänge 4 bewegen.



2. Die Schlange soll sich auf der nebenstehenden Spur bewegen. Mit dem Befehl *penDown()* kannst du bewirken, dass der Kopf der Schlange tatsächlich eine Spur hinterlässt. Du siehst diese aber nur, falls die Schlage sie nicht verdeckt. Die Zeile *penDown()* muss in deinem Programm vor den Bewegungsbefehlen stehen.



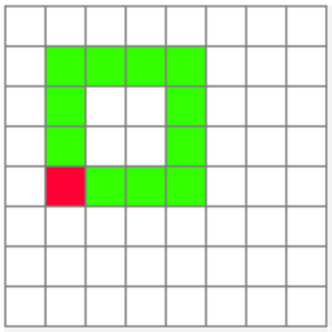
3. WIEDERHOLUNG MIT REPEAT

DU LERNST HIER...

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

MUSTERBEISPIELE

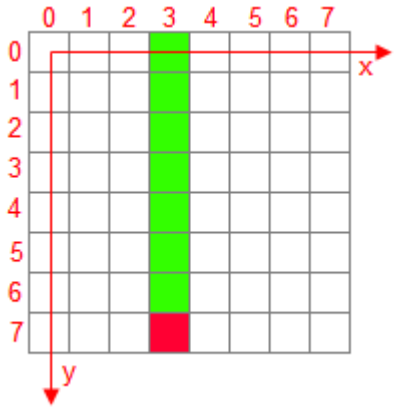
Die Schlange soll ein Quadrat mit einer Seitenlänge von 3 Schritten durchlaufen. Statt viermal die zwei Zeilen *forward(3) right(90)* hinzuschreiben, schreibst du sie nur einmal hin und sagst dem Computer mit dem Schlüsselwort repeat, dass er diese beiden Befehle 4 Mal wiederholen soll.



```
from oxosnake import *

makeSnake()
repeat 4:
    forward(3)
    right(90)
```

Wie du siehst, muss man die Programmzeilen, die wiederholt werden, gleichviel einrücken. Es ist zwar gleichgültig, um wieviel, aber wir wollen uns konsequent an 4 Leerschläge halten, damit alle Programme gleich aussehen. Zum Einrücken kannst du auch die Tabulator-Taste verwenden.Man spricht bei der Wiederholstruktur auch vom Durchlaufen einer Schleife.

Im folgenden Beispiel soll die Schlange in der untersten Zeile beginnen und sich dann endlos von unten nach oben und wieder zurück bewegen. Du verwendest den Befehl setPos(x, y), um die Schlange im x-y-Koordinatensystem auf den Punkt *(x, y)* zu setzen. Mit dem Befehl setSpeed(90) kannst du die Geschwindigkeit erhöhen (maximaler Speed = 100).



Der Ursprung des Koordinatensystems ist oben links, x-Achse nach rechts, y-Achse nach unten.

```
from oxosnake import *

makeSnake()
setPos(3, 7)
setSpeed(90)
repeat:
    forward(8)
    right(180)
```

Schreibst du repeat ohne einen Wert, so wird die Schleife endlos wiederholt. Du kannst das Programm mit Ctrl+C im Terminalfenster abbrechen oder einfach ein neues Programm herunterladen.

Deine Schlange erhält mit den Befehlen *setPos()* und *setSpeed()* neue Eigenschaften. Diese könntest du ihr bereits beim Erzeugen mit dem Befehl *makeSnake()* mitgehen:

```
from oxosnake import *

makeSnake(speed = 90, pos = (3, 7))

repeat:
    forward(8)
    right(180)
```

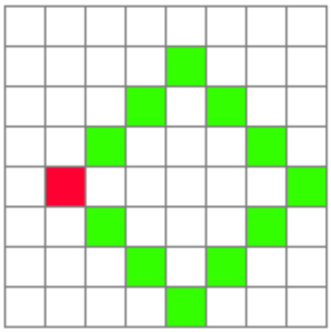
MERKE DIR...

Die Wiederholung wird mit *repeat n*: eingeleitet, wobei n die Anzahl der Wiederholungen ist. Der Doppelpunkt ist wichtig. Die Befehle im nachfolgenden Programmblock sind alle gleichweit eingerückt.

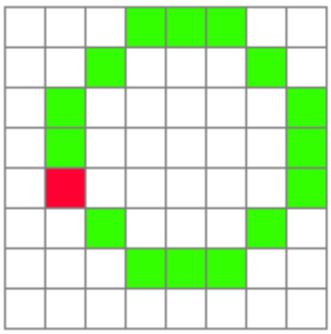
Mit *repeat:* ohne Wiederholungszahl wird die Schleife endlos wiederholt (bis du Ctrl+C im Terminalfenster klickst, ein neues Programm hinunterlädst, die Oxocard resettest oder sie in den Sleep-Modus setzt).

ZUM SELBST LÖSEN

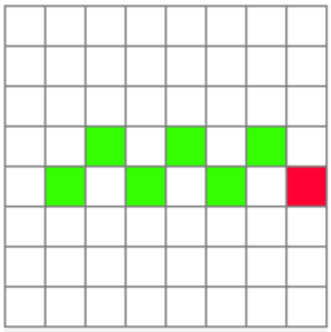
1. Verwende eine repeat-Schleife und lasse die Schlange genau einmal und dann auch endlos die nebenstehende Figur durchlaufen.



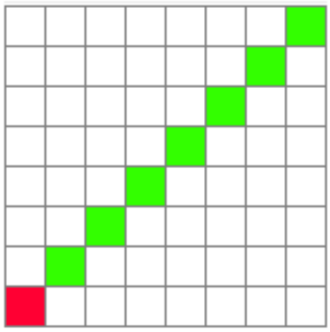
2. Die Schlange bewegt sich auf den Seiten eines 8-Ecks. Wenn du herausfindest, um welchen Winkel sie nach je zwei Schritten drehen muss, ist die nebenstehende Figur unter Verwendung einer *repeat*-Schleife ganz einfach. Lass die Schlage genau einmal und dann endlos das 8-Eck durchlaufen.



3. Kannst du auch diese Zick-Zack-Bewegung programmieren?



4. Die Schlange soll sich wiederholt diagonal von unten nach oben und zurück bewegen. Verwende eine endlose Schleife und teste, ob du das Programm mit der Eingabe von Ctrl+C im Terminalfenster beenden kannst.



.

4. FUNKTIONEN

DU LERNST HIER...

wie du mit benannten Programmblöcken, in Python ***Funktionen*** genannt, deine Programme strukturieren kannst. Die Verwendung von Funktionen ist von grosser Wichtigkeit, denn du vermeidest dadurch, dass du gleichen Code mehrmals im Programm hinschreiben musst (Codeduplikation) und du kannst damit Probleme in kleinere, leichter zu lösende Teilprobleme zerlegen.

MUSTERBEISPIELE

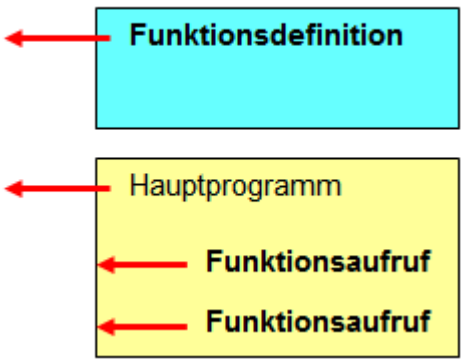
Im letzten Kapitel hast du gelernt. wie die Schlage ein Quadrat zeichnen kann. Dies ist eine typische, in sich geschlossene Aufgabe und wir wollen den dazu benötigten Code in einer Funktion *square* zusammenfassen. Dazu definierst du die Funktion mit dem Schlüsselwort *def* wie folgt:

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

Der Computer wird beim Lesen dieser Programmzeilen noch gar **nichts machen**, sondern sich lediglich merken, was er mit der Funktion *square* **tun soll**.

Die Funktionsdefinition besteht also aus einer Kopfzeile mit *def* und dem Funktionsnamen gefolgt von einer Parameterklammer und einem Doppelpunkt.

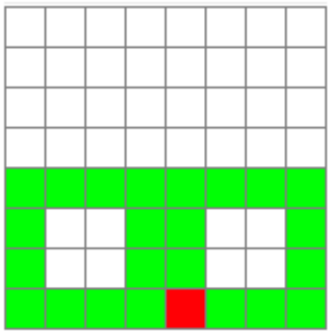
```
from oxosnake import *

def square():
    repeat 4:
        forward(3)
        right(90)

makeSnake()
setSpeed(90)
setPos(0, 7)
square()
setPos(4, 7)
square()
```



Die Befehle im Funktionskörper bilden einen Programmblock und sie müssen daher eingerückt sein.

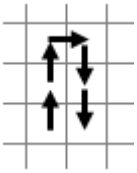
Hast du die Funktion einmal definiert, so kannst du sie in deinem Programm verwenden, indem du sie mit ihrem Namen aufrufst. Du kannst eine Funktion auch mehrmals aufrufen. Dein Programm zeichnet zwei Quadrate an verschiedenen Positionen.

Es wäre praktisch, wenn du beim Aufruf von *square()* auch sagen könntest, wo sich die linke untere Ecke der Quadrate befindet, damit du nicht im Hauptprogramm zweimal *setPos()* aufrufen musst. Dies ist kein Problem, da du in der Definition der Funktion die Koordinaten x und y als **Parameter** mit **Parameternamen** x und y einfügen kannst: *square(x, y)*. Beim **Aufruf** der Funktion mit *square(0, 7)* werden dann die Werte x, y der Funktion übergeben und an Stelle der Parameternamen verwendet.

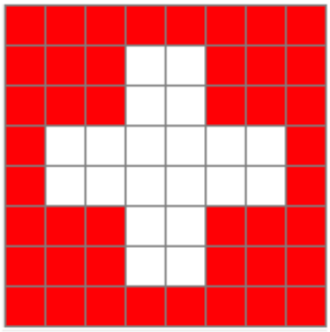


```
from oxosnake import *

def square(x, y):
    setPos(x, y)
    repeat 4:
        forward(3)
        right(90)

makeSnake()
setSpeed(90)
square(0, 7)
square(4, 7)
```

Im nächsten Beispiel willst du ein Schweizerkreuz zeichnen. Du teilst das Programm in Teilaufgaben ein und definiert zuerst eine Funktion shape(), die eine Teilfigur zeichnet. Diese rufst du 4 Mal auf um die Gesamtfigur zu zeichnen.



Die Schlange soll dabei eine weisse Spur auf einem roten Hintergrund zeichnen. Der Befehl penDown() bewirkt, dass die Schlange eine Spur hinterlässt.



Die Farbe der Spur wird mit setPenColor(WHITE) gesetzt, der Hintergrund wird mit setBgColor(RED) auf rot gesetzt.

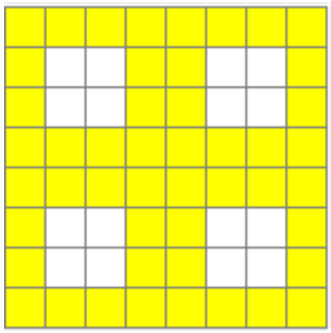
```
from oxosnake import *

def shape():
    forward(2)
    right(90)
    forward()
    right(90)
    forward(2)

makeSnake(speed = 70, pos = (3, 3))
setPenColor(WHITE)
penDown()

repeat 4:
   shape()
   left(90)
hide()
setBgColor(RED)
```

MERKE DIR...

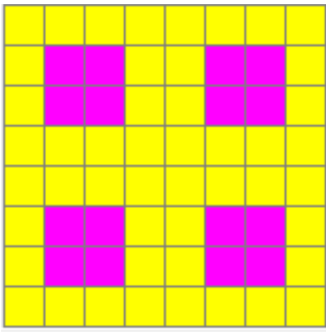
Um ein etwas komplizierteres Problem zu lösen, teilst du dieses in Teilprobleme ein und löst ein Teilproblem um das andere. Man nennt dies **Modularisierung** oder **strukturierte Programmierung**. ***Funktionen*** sind fundamentale Hilfsmittel für die strukturierte Programmierung, denn sie ermöglichen es, ein Teilprogramm mehrmals aufzurufen. In der Funktionsdefinition kannst du auch **Parameter** einfügen und die **Parameternamen** im Funktionskörper verwenden. Beim **Aufruf** der Funktion werden die Werte in der Parameterklammer der Funktion übergeben und an Stelle der Parameternamen eingesetzt.Bit dem Befehl *setPenColor(color)* änderst du die Stiftfarbe, *setBgColor(color)* ändert den Hintergrund. Am einfachsten ist es, die vordefinierten Farben RED, GREEN, BLUE, WHITE, YELLOW, CYAN, MAGENTA und BLACK zu verwenden.

ZUM SELBST LÖSEN

1a. Mit dem Befehl *setPenColor(YELLOW)* kannst du die Stiftfarbe auf gelb setzen. Verwende die Funktion *square(x, y)*, um nebenstehende Figur zu zeichnen.

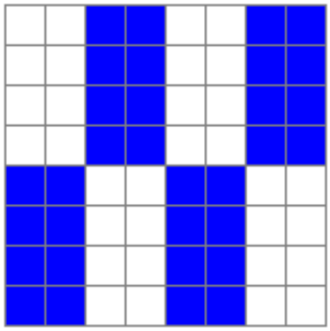


1b. Zeichne die nebenstehende Figur, in dem du zuerst mit dem Befehl *setBgColor(MAGNETA)* die Hintergrundfarbe änderst und dann mit der gelben Stiftfarbe Quadrate zeichnest.



1c. Die Farben RED, GREEN, BLUE, WHITE, YELLOW, CYAN und MAGENTA sind vordefiniert. Du kannst aber mit *setPenColor(r, g, b)* jede beliebige RGB-Farbe wählen, wo r, g, b die rote, grüne und blaue Farbkomponente ist (eine Zahl zwischen 0 und 255). Da die LEDs sehr hell leuchten, sind oft Zahlen im Bereich 0 bis 50 vorteilhafter. Zeichne unter Verwendung von *square(x, y)* einige selbsterfundene farbige Bilder.

2. Die Funktion *rectangle()* soll ein Rechteck mit der Höhe 4 und der Breite 1 zeichnen. Am Ende soll die Schlange wieder am Startpunkt sein und nach oben schauen. Definiere diese Funktion und verwende sie, um die nebenstehende Figur zu zeichnen.



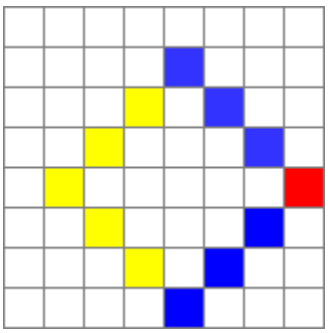
5. IF-ELSE-STRUKTUR (SELEKTION)

DU LERNST HIER...

wie du mit Hilfe der if-else-Struktur bewirken kannst, dass bestimmte Programmblöcke nur unter gewissen Bedingungen ausgeführt werden.

MUSTERBEISPIELE

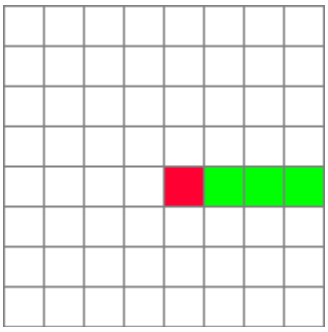
Die Selektion wird mit dem Schlüsselwort ***if*** eingeleitet, gefolgt von einer Bedingung. Die Anweisungen nach ***if*** werden nur dann ausgeführt, wenn die Bedingung wahr ist, sonst werden die Anweisungen nach ***else*** ausgeführt. In der ***if-***Bedingung werden üblicherweise die Vergleichsoperatoren >, >=, < , <= , == , != verwendet. Die Anweisungen im *if*- und *else*-Block müssen eingerückt sein.



In folgenden Beispiel bewegt sich die Schlange auf einer quadratischen Bahn. Die Schlange hat einen gelben Schwanz, wenn sich ihr Kopf in der linken Hälfte der Oxocard befindet, sonst ist der Schwanz blau.

Du fragst nach jedem Vorwärtsschritt die aktuelle x-Koordinate mit dem Befehl getX() ab. Ist die Koordinate kleiner als 4, wird die Schwanzfarbe mit dem Befehl setTailColor() auf gelb gesetzt, sonst wird sie auf blau gesetzt.

```
from oxosnake import *

makeSnake(speed = 90, heading = 45)
repeat:
    repeat 3:
        forward()
        if getX() < 4:
            setTailColor(YELLOW)
        else:
            setTailColor(BLUE)
    right(90)
```



Der *else*-Block kann auch wegfallen. In diesem Fall führt das Programm den *if-Block aus,* falls die Bedingung erfüllt ist und fährt dann mit der Zeile unter dem if-Block weiter. Ist die Bedingung nicht erfüllt, wird der if-Block einfach übersprungen.

In deinem Beispiel ändert die Schlange die Bewegungsrichtung wenn sie am rechten oder linken Rand ankommt. Beachte, dass im Vergleichsoperator das Gleichheitszeichen verdoppelt wird und dass man Bedingungen mit *or* verbinden kann. Eine solche Bedingung ist wahr ist, wenn die eine oder die andere Teilbedingung (oder auch beide) wahr sind.

```
from oxosnake import *

makeSnake(speed = 90)
right(90)
repeat:
    if getX() == 7 or getX() == 0:
       left(180)
    forward()
```

MERKE DIR...

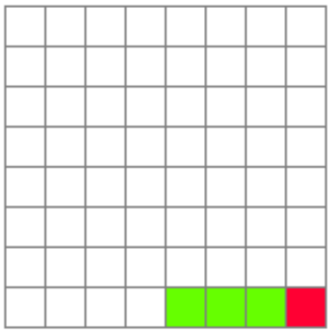
Mit der Programmstruktur *Selektion* kannst du bewirkt, dass bestimmte Anweisungen nur unter gewissen Bedingungen ausgeführt werden. Sie funktioniert gleich wie in der Umgangssprache: "Falls *eine Bedingung erfüllt ist*, dann mache *dies*, sonst mache *jenes*.

Man kann Bedingungen A und B mit *or* oder *and* verbinden:

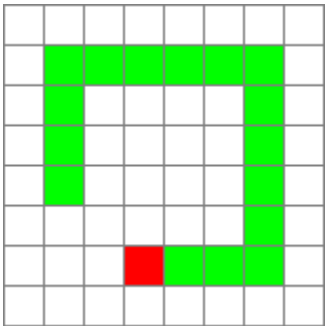
A	B	A or B	A and B
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

ZUM SELBST LÖSEN

1. Eine Schlange startet in der linken unteren Ecke und bewegt sich von links nach rechts. Wenn sie am rechten Rand angekommen ist, wird sie wieder an den linken Rand zurückversetzt und bewegt sich wiederholt von links nach rechts.



2. Die Schlange startet an der Position (1, 6) und bewegt sich in einer endlosen Schleife auf einem Quadrat mit der Seitenlänge 5. Nach jeder zurückgelegten Seite fügst du mit dem Befehl *growTail()* ein Schwanzelement hinzu, bis die Schlange so lang ist, dass der Kopf das letzte Schanzelement berührt. Das kannst du abfragen mit
   if intersect():
   Dann soll die Schwanzfarbe auf BLUE wechseln und das Programm abbrechen.



6. VARIABLEN

DU LERNST HIER...

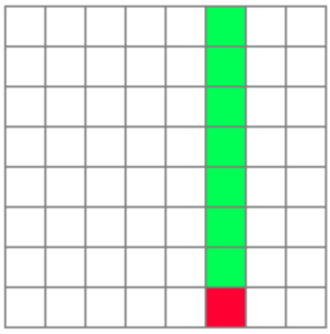
was Variablen sind und wie man sie verwendet.

MUSTERBEISPIELE

Variablen sind in der Informatik Speicherplätze für Werte, die man im Laufe einer Programmausführung wieder ändern kann. Eine Variable wird über ihren Namen angesprochen, kurz gesagt: **Eine Variable hat einen Namen und einen Wert**.

Im Beispiel wird der Anfangswert der Variablen x zufällig im Bereich der ganzen Zahlen zwischen 0 und 7 festgelegt. Du verwendest dazu die Funktion randint(a, b) aus dem Modul random, die bei jedem Aufruf eine ganzzahlige Zufallszahl im Bereich a und b liefert (Grenzen eingeschlossen).

Im ersten Beispiel wird die Schlange ganz oben auf eine zufällige Spalte gesetzt. Sie bewegt sich dann mit voller Geschwindigkeit nach unten, bis sie auf der untersten Zeile angekommen ist. Dann beginnt der Prozess endlos von neuem. Dazu speicherst du die Zufallszahl, die *randint()* zurückgibt, in der Variablen x und übergibst dann ihren Wert an *setPos(x, 0)*. Beim Erzeugen der Schlangen mit *makeSnake()* legst du die Bewegungsrichtung, die Geschwindigkeit und mit size = 8 die Länge der Schlange fest.

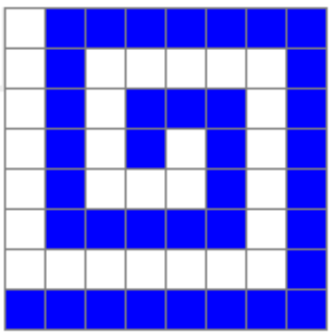


```
from oxosnake import *
from random import randint

makeSnake(heading = 180, size = 8, speed = 90)
repeat:
    x = randint(0, 7)
    setPos(x, 0)
    forward(8)
```

Im nächsten Beispiel zeichnet dieSchlange eine quadratische Spirale. Sie startet in der Mitte, bewegt sich einen Schritt vorwärts und zeichnet ein spur. Dreht danach 90° rechts und vergrössert den Wert der Variablen size um 1. In der Informatik schreibt man:

size = size + 1 oder abgekürzt auch *size += 1*

Dieser Befehl ist also nicht etwa eine mathematische Gleichung, sondern bewirkt eine neue **Wertzuweisung** von *size*.



Die Schlange zeichnet dann immer grössere Strecken bis die Variable *size* den Wert 10 erreicht und die Schlange samt dem Schwanz aus dem LedGrid verschwindet.

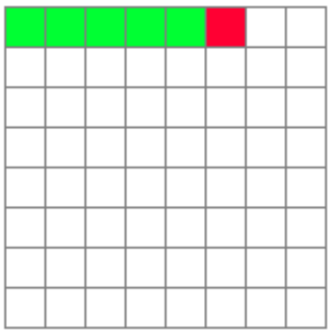
```
from oxosnake import *

makeSnake(speed = 90, pos = (3, 3))
penDown()

size = 1
repeat:
    forward(size)
    right(90)
    size = size + 1
    if size == 10:
        break
```

MERKE DIR...

Variablen verwendet man für Werte, die man speichern, wieder lesen und meist auch verändern will. Mit dem Befehl v = v + 5 wird der aktuelle Wert von v um 5 vergrössert.

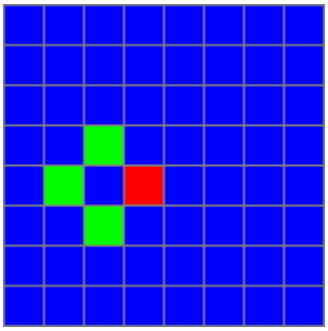
ZUM SELBST LÖSEN

1. Erzeuge eine Schlange mit heading = 90 an der Position (0, 0). In einer endlosen Schleife wird jeweils eine Zufallszahl n zwischen 0 und 7 erzeugt. Die Schlange bewegt sich n Schritte vorwärts. Mit sleep(1) kannst du an der Endposition 1 Sekunde lang warten. Danach kehrt die Schlange wieder zum Ausgangspunkt und zeichnet eine weitere zufällig lange Strecke.

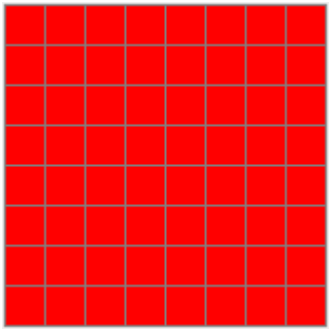


2. Die Schlange soll 20 kleine quadratische Figuren, deren Positionen mit zufällig gewählten Koordinaten bestimmt sind, zeichnen. Die Schlange bewegt sich dabei aber immer schneller. Dazu setzt du ihre Geschwindigkeit v zu Beginn auf 40 und erhöhst sie nach jedem Schleifendurchgang um 5.

   Am besten wählst du mit makeSnake(heading = 45) bereits beim Erzeugen der Schlange die passende Blickrichtung.



3. Mit *setBgColor((255, 0, 0))* leuchten alle LEDs rot mit maximaler Helligkeit. Mit dem Befehl *dim(dimFaktor)* kannst du die Helligkeit dimmen. Führe dieses Dimmen in 6 Stufen durch. Setzte zuerst dimFaktor = 1 und verdopple diesen Wert jeweils nach 0.5 Sekunden.



7. WHILE & FOR

DU LERNST HIER...

wie du die Wiederholstrukturen mit den Schlüsselwörter *while* und *for* verwendest. Die *while-Schleife* ist eine der wichtigsten Programmstrukturen überhaupt. Sie kann allgemein für jede Art von Wiederholungen verwendet werden und kommt in praktisch allen Programmiersprachen vor.

Mit *repeat* konntest du bisher einfache Wiederholungen programmieren, ohne Variablen zu verwenden. Da du jetzt den Variablenbegriff kennst, kannst du auf *repeat* verzichten, was zu empfehlen ist, da *repeat* nicht zu den Python-Schlüsselwörtern gehört.

WHILE-SCHLEIFE

Eine *while-Schleife* wird mit dem Schlüsselwort while eingeleitet, gefolgt von einer Bedingung und einem Doppelpunkt. So lange die Bedingung erfüllt ist, werden die Befehle im nachfolgenden Programmblock wiederholt. In der Bedingung werden in der Regel die Vergleichsoperatoren >, >=, <, <=, ==, != verwendet. Die Anweisungen im *while*-Block müssen eingerückt sein.

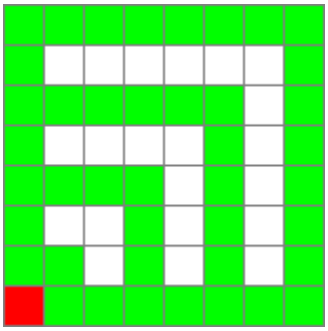
In der dir bekannten Funktion *square(s)* zum Zeichnen eines Quadrats ersetzt du das repeat durch eine while-Schleife, indem du einen Variable i einführst.

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

```
def square(s):
    i = 0
    while i < 4:
        forward(s)
        right(90)
        i += 1
```

Du musst i zuerst auf 0 initialisieren und testest dann in der Schleifenbedingung, ob i kleiner als 4 ist. Falls dies zutrifft, durchläuft du den Schleifenkörper und erhöhst die Variable um 1. Da i in jedem Schleifendurchlauf um 1 erhöht wird, nennt man i auch **Schleifenzähler**. Die Schleifenbedingung heisst auch **Laufbedingung**, da die Schleife so lange durchlaufen wird, solange sie wahr ist.

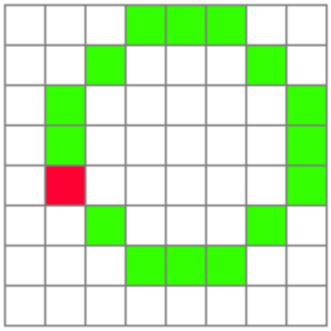
In deinem Beispiel zeichnet die Schlange in einer while-Schleife immer grössere Quadrate. Die Quadratseite s ist zu Beginn 1 und wird nach jedem Schleifendurchgang um 2 vergrössert, so lange s kleiner als 8 ist.



```
from oxosnake import *

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

makeSnake(speed = 90, pos = (0, 7))
s = 1
while s < 8:
    square(s)
    s = s + 2
```

In der Robotik werden häufig sogenannte "endlose while-Schleifen" verwendet. Diese werden mit while True: eingeleitet. Da *True* immer wahr ist, wiederholen sich die Schleifendurchgänge endlos. Bisher hast du das gleiche mit *repeat:* erreicht.

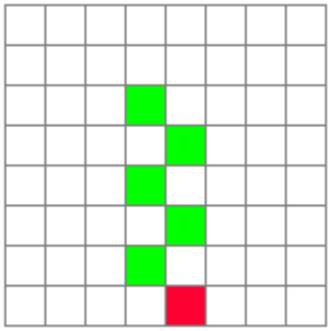
Mit dem folgende Programm zeichnet die Schlange endlos ein 8-Eck.



```
from oxosnake import *

makeSnake(speed = 90)
while True:
    forward(2)
    right(45)
```

Du kannst auch mehrere while-Schleifen ineinander verschachteln. In der inneren Schleife bewegt sich die Schlange wellenartig von ober nach unter, so lange die y-Koordinate kleiner als 11 ist, bis also auch der Schwanz verschwunden ist. Die äussere Schleife wird endlos wiederholt. Der Startpunkt auf der obersten Zeile wird zufällig gewählt.

Achte auf die korrekte Einrückung!



```
from oxosnake import *
from random import randint

makeSnake(heading = 135, speed = 90)
while True:
    x = randint(1, 6)
    setPos(x, 0)
    while getY() < 11:
        forward()
        right(90)
        forward()
        left(90)
```

FOR-SCHLEIFE

Oft benötigst du in einer Wiederholschleife eine ganzzahlige Variable, die bei jedem Durchgang um eins grösser wird. Du kannst dies zwar mit einer *while*-Schleife lösen, einfacher geht es aber mit einer *for*-Schleife, bei der der Schleifenzähler automatisch verändert wird.

Die Funktion *range(n)* liefert dir alle Zahlen von 0 bis n-1 und mit der for-Schleife werden alle diese Werte durchlaufen.

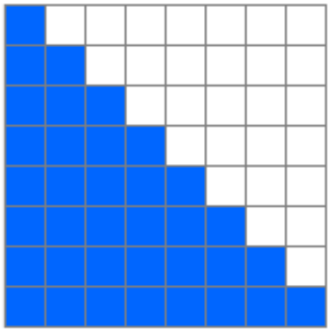
for i in range(n): durchläuft Zahlen *i* von 0 bis *n-1*

Da range(a, b) alle Zahlen von a bis b-1 liefert, kannst du mit

for i in range(a, b): die Zahlen i von a bis b-1durchlaufen

Beachte, dass der letzte Wert b - 1 und nicht etwa b ist. Nach der Zeile mit *for* muss ein Doppelpunkt stehen und der Schleifenkörper ist wie bei einer while-Schleife eingerückt.

Im Beispiel zeichnet die Schlage reihenweise Zeilen, die immer um 1 länger werden. Dazu durchläuft die Zeilennummer i die Werte von 0 bis 7, also die Werte in range(8). Im Schleifenkörper verwendest du die Variable i, um die Anzahl Vorwärtsschritte festzulegen.



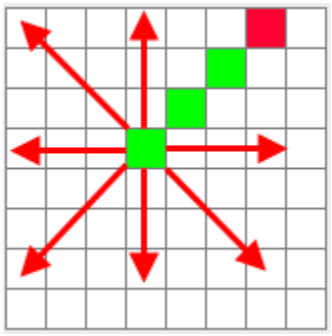
```
from oxosnake import *

makeSnake(speed = 90, pos = (0, 0), heading = 90)
hide()
penDown()
for i in range(8):
    setPos(0, i)
    forward(i)
```

MERKE DIR...

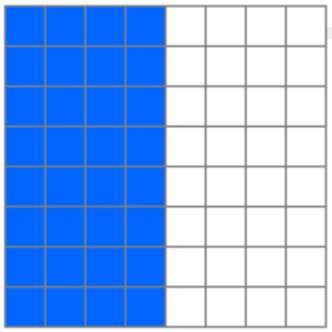
Um bestimmte Programmblöcke mehrmals auszuführen, kannst du im TigerJython die ***repeat-***, **while**- oder ***for***-Schleife verwenden. Die while-Schleife ist sehr allgemein und immer einsetzbar

ZUM SELBST LÖSEN

1. Die Schlange startet an der Position (3, 3) und bewegt sich mit heading = 0 drei Schritte vorwärts. Dann springt sie wieder an die Ausgangsposition, vergrössert heading um 45 und bewegt sich wieder drei Schritte vorwärts, so lange heading kleiner als 360 ist. Verwende eine while-Schleife mit einer geeigneten Laufbedingung.

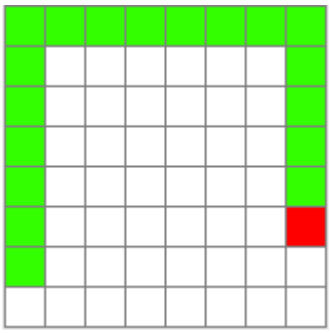


2. Verwende eine for-Schleife, um das nebenstehende Bild zu erzeugen.



3. Erzeuge mit *makeSnake(pos = (0, 7), size = 2)* eine kleine Schlange an der Position (0, 0). Die Schlange bewegt sich auf dem äussersten Quadrat. Jedes Mal, wenn sie eine Quadratseite zurückgelegt hat, wird ihr Schwanz um 1 Teil länger.

   Verwende den Befehl *growTail()* und lasse die Schlange so lange laufen, bis ihr Schwanz die Länge 20 hat.



8. SOUND

DU LERNST HIER...

mit Oxocard Töne, Tonfolgen und kurze Melodien abzuspielen. Für die Speicherung der Tonfolgen verwendest du ***Listen***.

MUSTERBEISPIELE

Eine Tonfolge abspielen
Um mit der Oxocard ein Tonfolge abzuspielen, brauchst du einen Kopfhörer oder einen aktiven Lautsprecher (mit einem Verstärker), den du direkt an die integrierte Audiobuchse einschliessen kannst.

Zuerst musst du das Modul music importieren. Dann kannst du mit dem Befehl playTone(f, 500) einen Ton mit der Frequenz f während 500 Millisekunden abspielen. Frequenzen einiger Töne findest du im Overlay-Fenster:

Um mehrere Töne nacheinander abzuspielen, erstells du eine Liste *song* mit den zugehörigen Frequenzen. Die Elemente einer Liste musst du in eckigen Klammern und mit Komma getrennt schreiben (Frequenz 0 legt eine Pause ein).

Zum Abspielen durchläufst du die Liste mit einer for-Schleife und entnimmst ihr die Frequenzen, die du *playTone()* übergibst. Hier ein Beispiel des Beginns eines Kinderlieds.

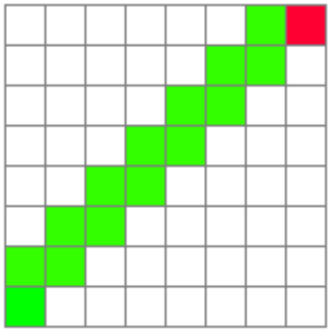
```
from music import playTone

song = [262, 294, 330, 349, 392, 392, 392, 0, 440, 440, 440, 440, 392, 0,
        349, 349, 349, 349, 330, 0, 330, 0, 294, 294, 294, 294, 262]

for f in song:
   playTone(f, 500)
```

Statt Frequenzen kannst du auch eine musikalische Notation (r ist eine Pause) verwenden. Die Tonhöhen werden dabei durch einen Buchstaben und (eventuell ein Kreuz für einen Halbton nach oben) angegeben. Du kannst auch mit einem anschliessenden Doppelpunkt und einer Zahl sagen, dass dieser Ton länger ausgehalten wird. Folgende Töne der chromatischen Tonleiter sind verfügbar:

```
'C', 'C#', 'D', 'D#', 'E', 'F', 'F#', 'G', 'G#', 'A', 'A#', 'H',
'c', ' c#', 'd', 'd#', 'e', 'f', 'f#', 'g', 'g#', 'a', 'a#', 'h',
'c2', 'c2#', 'd2', 'd2#', 'e2', 'f2', 'f2#', 'g2', 'g2#', 'a2', 'a2#', 'h2',
'c3'
```

Programme mit Sound ergänzen

Es ist lustig, deine Programme mit Sound zu versehen. Hier bewegt sich die Schlange die Treppe hinauf und spielt dabei immer höhere Töne.



```
from oxosnake import *
from music import playTone

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

makeSnake()
setPos(0, 7)
scale = ['c','d','e','f','g','a','h','c2']

for tone in scale:
    playTone(tone, 300)
    step()
```

Eingebaute Melodien abspielen

Einige Melodien sind im Modul *music* eingebaut. Mit folgendem Programm kannst du sie abspielen:

```
from music import *
from oxocard import *

songs = [ENTERTAINER, RINGTONE, BIRTHDAY, JUMP_UP, JUMP_DOWN, DADADADUM,
         POWER_UP, POWER_DOWN, PUNCHLINE, WEDDING, BOOGYWOODY, PRELUDE]

n = 1
for song in songs:
    display(n)
    n += 1
    for f in song:
        playTone(f, 150)
    sleep(3
```

Du kannst auch einfacher der Funktion ***playSong()*** eine Liste mit Tönen übergeben, die sie dann abspielt:

playSong(ENTERTAINER)

oder falls du nach die Tondauer angeben willst:

playSong(ENTERTAINER, 300) (standardmässig ist sie 150 ms).

```
from music import *
playSong(ENTERTAINER, 300)
```

MERKE DIR...

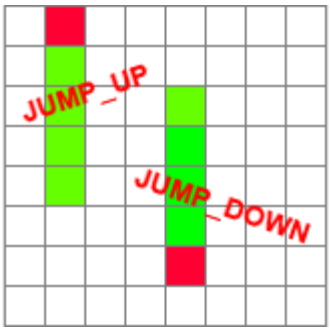
Mit *playSound(frequency, duration)* kannst du einen Sound mit gegebener Frequenz (in Hertz) und Dauer (in Millisekunden) abspielen. Die Funktion kehrt erst zurück, wenn der Sound fertig gespielt ist, du kannst also nicht mehrere Töne miteinander erzeugen.

ZUM SELBST LÖSEN

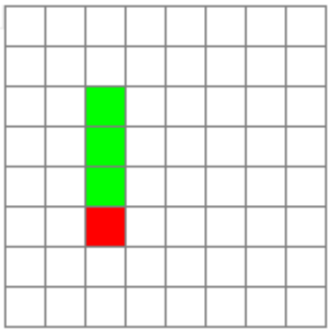
1. Spiele den folgenden Song wiederholt ab und bestimme selbst die passende Geschwindigkeit.

   song = ['d', 'd#', 'e', 'c2:2', 'e', 'c2:3', 'e', 'c2:3', 'c2', 'd2', 'd2#', 'e2', 'c2', 'd2', 'e2:2', 'h', 'd2:2', 'c2:4']

2. Die (schnelle) Schlange startet zuerst an einer zufälligen Position in der untersten Reihe, bewegt sich nach oben und spielt dabei die Melodie JUMP_UP. Danach startet sie an einer zufälligen Position in der obersten Reihe, bewegt sich nach unten und spielt dabei die Melodie JUMP_DOWN. Diese Bewegungen wiederholt sie in einer endlosen Schleife.



3. Die (schnelle) Schlage startet an einer zufälligen Position in der obersten Reihe und spielt während der Bewegung nach unten eine Tonfolge mit abnehmender Frequenz ab. Dies soll sich endlos wiederholen.



4. Suche eine passende Vertonung zum folgenden Programm:

```
from oxosnake import *

makeSnake(speed = 90)
right(90)
repeat:
    if getX() == 7 or getX() == 0:
       left(180)
    forward()
```

9. LED-DISPLAY

DU LERNST HIER...

wie du auf dem LED-Display einzelne Punkte, Figuren, Zahlen und Texte anzeigst.

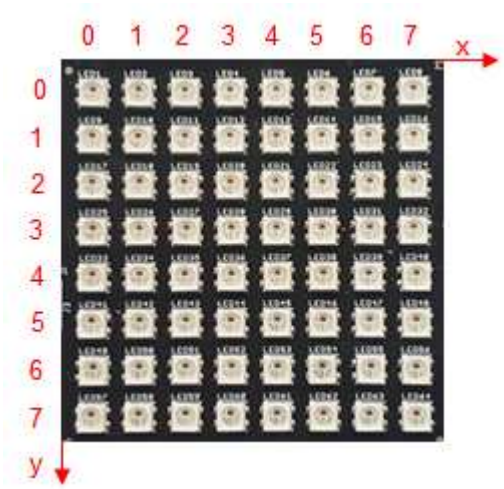
KOORDINATENSYSTEM, DIMMEN

Das Modul *oxocard* verwendet die Klasse *OxoGrid*, die das Display als Softwareobjekt modelliert.

Die 64 Farb-LEDs (Neopixels) des Displays sind in einer 8x8 Matrix angeordnet und über ihre x- und y-Koordinaten einzeln ansprechbar. Beispielsweise kannst du mit

dot(x, y, color)

die LED an der Position x, y mit der gegebenen Farbe einschalten. color ist die RGB-Farbe und wird als Farbtupel (r, g, b) mit den Farbwerten für rot, grün und blau im Bereich 0..255 angegeben.



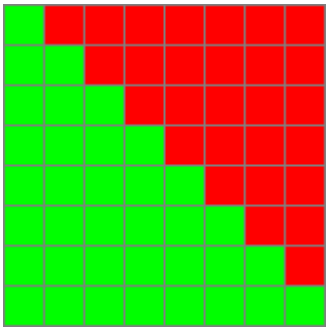
Du kannst alle LEDs mit dem Befehl *clear()* ausschalten oder mit dem Befehl *clear(color)* alle LEDs auf die angegebene Farbe setzen.

Die LEDs leuchten sehr stark, was störend sein kann, wenn du die Oxocard nicht in der Kartonbox eingebaut hast. Mit dem Befehl *dim(20)* kannst du die Helligkeit generell um den Faktor 20 reduzieren. Damit kannst du bis auf diesen Befehl alle Programme unverändert für eine eingebaute oder nicht-eingebaute Karte verwenden.

MUSTERBEISPIELE

Einzelne LEDs einschalten

Dein Programm soll alle LEDs der Reihe nach einschalten. Ist die x-Koordinate grösser als die y- Koordinate, ist die LED- rot, sonst grün. Damit man das Zeichnen besser verfolgen kann, wird mit dem Befehl sleep(0.1) die Programmausführung in der Wiederholschleife um 0.1 Sekunden angehalten.

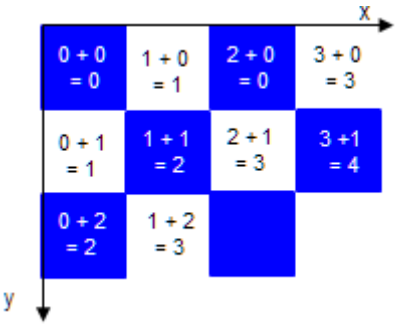


```
from oxocard import *

for y in range(8):
```

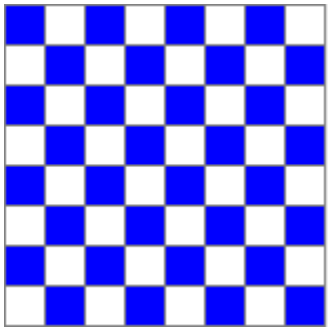
```
    for x in range(8):
        if x > y:
            dot(x, y, RED)
        else:
            dot(x, y, GREEN)
        sleep(0.1)
```

Schachbrett zeichnen

Mit nur wenigen Programmzeilen kannst du ein Schachbrettmuster zeichnen, indem du überlegst, dass für die weissen Zellen die Summe s = x + y der Koordinaten gerade und für die blauen Zellen ungerade ist.



In Python liefert s % 2 den Rest, den es bei der (ganzahligen) Division durch 2 gibt. Für gerade Zahlen ist also s % 2 = 0.

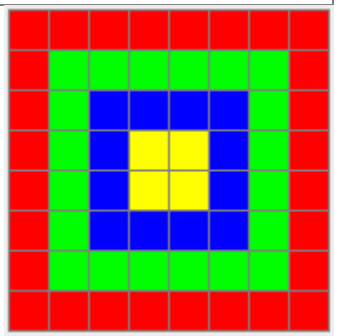


Du durchläufst mit zwei ineinandergeschachtelten for-Schleifen zeilenweise das Display und setzt die Farbe entsprechend auf weiss oder blau.

```
from oxocard import *

for y in range(8):
    for x in range(8):
        if (x + y) % 2 == 0 :
            dot(x, y, BLUE)
        else:
            dot(x, y, WHITE)
```

Figuren zeichnen

Mit der Funktion rectangle(ulx, uly, w, h, color) kannst du ein Rechteck mit gegebener oberen linken Ecke *ulx*, *uly* mit Breite w und Höhe h und der gegebenen Farbe zeichnen. In deinem Beispiel zeichnest du vier konzentrisch angeordnete Quadrate in verschiedenen Farben.



```
from oxocard import *

rectangle(0, 0, 8, 8, RED)
rectangle(1, 1, 6, 6, GREEN)
rectangle(2, 2, 4, 4, BLUE)
rectangle(3, 3, 2, 2, YELLOW)
```

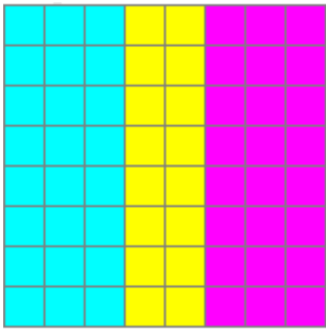
Aus der Dokumentation kannst du entnehmen, dass das Modul *oxocard* auch Befehle für das Zeichnen von Linien, Kreisen und Pfeilen enthält.

MERKE DIR...

Wenn du das Modul *oxocard* importierst, stehen dir Befehle zum Anzeigen von Punkten, Rechtecken, Kreisen, Linien, Pfeilen, Buchstaben und Zahlen zur Verfügung. Wenn du mehrere Zeichenoperationen durchführst, so werden diese übereinander gezeichnet. Um das Display zu löschen, rufst du *clear()* auf. Mit *clear(color)* kannst du auch alle Pixel auf die Farbe *color* setzen.

ZUM SELBST LÖSEN

1. Ergänze den folgenden Programmcode, so dass die LEDs wie in der Vorlage in den Farben CYAN, YELLOW und MAGENTA leuchten.

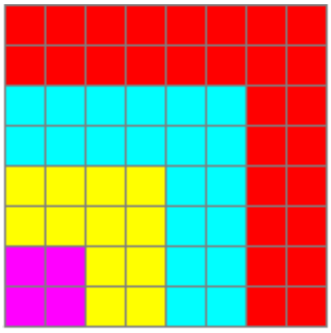


```
from oxocard import *

for y in range(8):
    for x in range(8):
        if

        elif

        else:
            dot(x, y, MAGENTA)
        sleep(0.1)
```

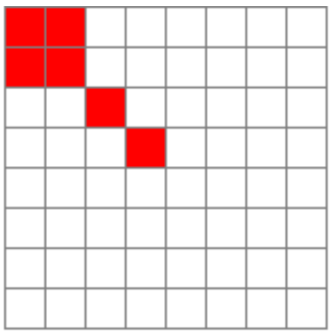
2. Verwende den Befehl

   fillRectangle(x, y, collor),

   wo (x, y) die Koordinaten des linken unteren Eckpunktes sind, um die nebenstehende Figur zu zeichnen.



3. Mit dem Befehl *arrow(x, y, dir, length, color)* kannst du einen Pfeil zeichnen. (x, y) ist der Anfangspunkt. Die Richtung *dir* hat die Werte 0: Ost, 1: Nord-Ost, 2: Nord, 3: Nord-West, 4: West, 5: Süd-West, 6: Süd, 7: Süd-Ost). Lasse einen Pfeil mit Anfangspunkt (3,3) in 45-Schritten im Uhrzeigersinn rotieren (endlos).



10. BILDER, ANIMATIONEN

DU LERNST HIER...

wie du auf dem LED-Display ganze Bilder darstellen und sie animieren kannst.

MUSTERBEISPIELE

Eigene Images erstellen

Statt einzelne LEDs anzusteuern, kannst du die Farben aller 64 LEDs in einer Datenstruktur angeben, in welcher die Farben zeilenweise aufgelistet werden. Dabei wird der Farbwert als hexadezimale Zahl angegeben (durch Voranstellen von *0x*).

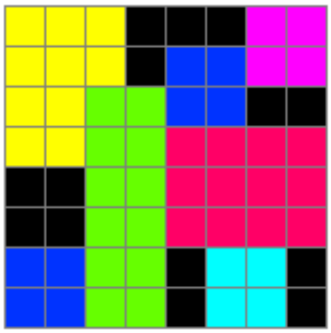
Die Hexzahlen werden durch Ziffern 0..9 und den Buchstaben a, b, c, d, e, f (manchmal auch gross geschrieben) angegeben, wobei folgende Wertigkeiten gelten:

Ziffer	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
Wert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Bei mehrstelligen Hexzahlen hat jede Stelle von rechts nach links aufsteigend die Wertigkeit $16^0 = 1$, $16^1 = 16$, $16^2 = 256$, usw. Die Hexzahl 0xe7 entspricht als der Dezimalzahl 14 * 16 + 7 = 231.

Je zwei Hexziffern entsprechen einer Farbkomponente in der Reihenfolge rot, grün, blau (von links nach rechts): 0xrrggbb, beispielsweise entspricht 0xe71aff dem RGB-Farbwert red = 0xe7 = 241, green = 0x1a = 26, blue = 0xff = 255, also (241, 26, 255).

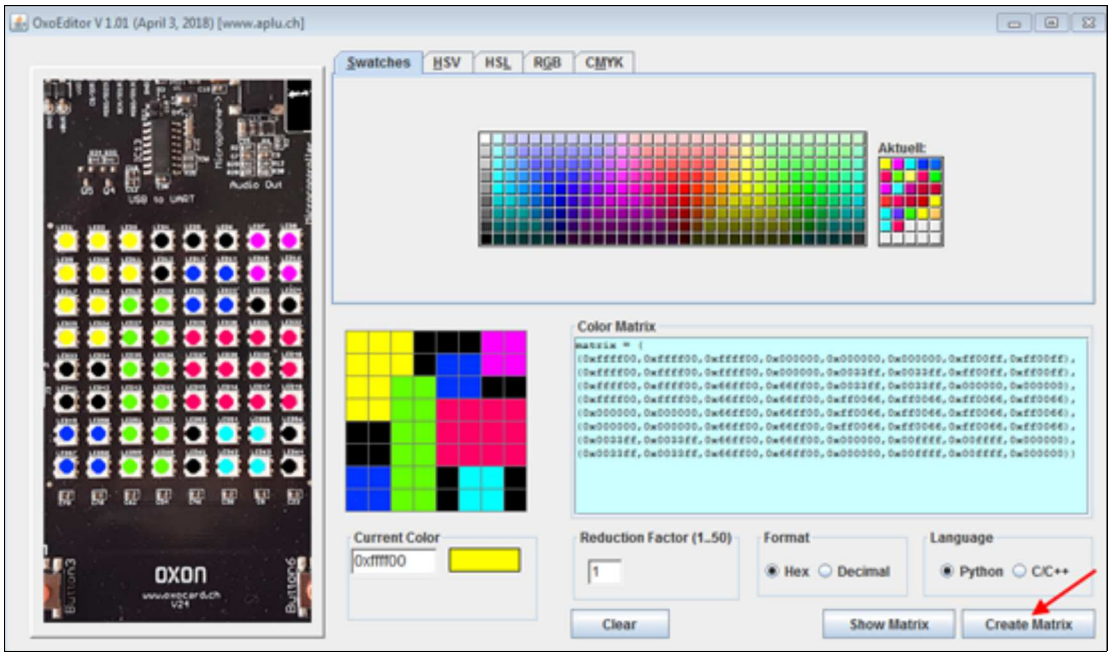
(Eine ähnliche Farbbezeichnung siehst du auch in Grafikprogrammen, wie z.B. Photoshop.) Die Farbwerte jeder Zeile wird in einem Tupel mit 8 Hexzahlen kodiert, und für das ganze 8x8 pixel grosse Bild in einem übergeordneten Tupel zusammengefasst. Dies ergibt eine übersichtliche **Datenstruktur**, die man eine **Matrix** nennt.



Mit dem Befehl image(matrix) wird das ganze Bild, d.h. alle Pixel miteinander, angezeigt. Beispielsweise nebenstehendes Bild mit dem Programm:

```
from oxocard import *

matrix = (
(0xffff00,0xffff00,0xffff00,0x000000,0x000000,0x000000,0xff00ff,0xff00ff),
(0xffff00,0xffff00,0xffff00,0x000000,0x0033ff,0x0033ff,0xff00ff,0xff00ff),
(0xffff00,0xffff00,0x66ff00,0x66ff00,0x0033ff,0x0033ff,0x000000,0x000000),
(0xffff00,0xffff00,0x66ff00,0x66ff00,0xff0066,0xff0066,0xff0066,0xff0066),
(0x000000,0x000000,0x66ff00,0x66ff00,0xff0066,0xff0066,0xff0066,0xff0066),
(0x000000,0x000000,0x66ff00,0x66ff00,0xff0066,0xff0066,0xff0066,0xff0066),
(0x0033ff,0x0033ff,0x66ff00,0x66ff00,0x000000,0x00ffff,0x00ffff,0x000000),
(0x0033ff,0x0033ff,0x66ff00,0x66ff00,0x000000,0x00ffff,0x00ffff,0x000000))

image(matrix)
```

Ein praktisches Hilfsmittel bei der Erstellung einer Farbmatrix ist unsere Applikation Oxoeditor. Lade die Datei oxoeditor.zip herunter und speichere die darin enthaltene jar-Datei in einem beliebigen Verzeichnis. Starte sie dann mit einem Doppelklick (Auf deinem Rechner mit das Java Runtime Environment installiert sein).



Mit der Maus wählst du zuerst die gewünschte Farbe und markierst die LEDs, die mit dieser Farbe leuchten sollen. Mit dem Klick auf ***Create Matrix*** wird die Matrix erzeugt. Diese fügst du mit Ctrl+C und Ctrl+V in dein Programm ein. Damit die LEDs nicht zu stark leuchten, kannst du einen ***Reduction Factor*** eingeben (z.B. 5). Dadurch werden alle Farben um den Faktor 5 schwächer. Mit Klick auf die Schaltfläche ***Clear*** wird das ganze Bild gelöscht.

Automatisch und manuell rendern

Grundsätzlich werden alle Zeichnungsoperation in einem Bildbuffer ausgeführt. Damit sie auch tatsächlich auf dem Display sichtbar werden, muss der Bildbuffer auf dem Display ***gerendert*** werden. Das Rendering wird mit dem Befehl *repaint()* durchgeführt. Standardmässig ist das automatische Rendering eingeschaltet. und das *repaint()* wird bei jeder Zeichnungsoperation automatisch aufgerufen.

Oft möchtest du mehrere Zeichnungsoperationen zusammenfassen, bevor du sie als Ganzes sichtbar machst. Dazu schaltest du mit *enableRepaint(False)* das automatische Rendering aus und führst es an der gewünschten Stelle mit einem Aufruf von *repaint()* aus.

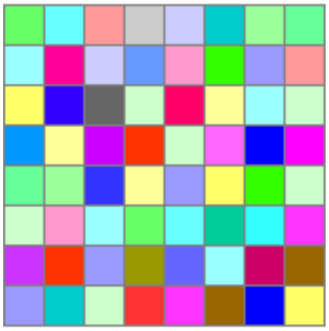
Zur Demonstration werden mit *dot()* einzelne Zeilen gezeichnet. Mit automatischem Rendering sieht man den Aufbau der einzelnen Zeilen, bei manuellem Rendering werden die Zeilen als Ganzes sichtbar.

```
from oxocard import *

while True:
    for y in range(8):
        clear()
        for x in range(8):
            dot(x, y, RED)
```

```
from oxocard import *

enableRepaint(False)
while True:
    for y in range(8):
        clear()
        for x in range(8):
            dot(x, y, RED)
        repaint()
```

Als Anwendung erstellst du ein lustiges Farbspiel und wählst für jede LED eine zufällige Farbe. Dabei soll aber die Farbänderung nicht für jede LED einzeln sichtbar sein, sondern nur für das ganze Bild.

Du deaktivierst mit dem Befehl enableRepaint(False) das automatische Rendern des Displays. Erst nachdem alle LEDs ihre neue Farbe erhalten haben, aktualisierst du das ganze Display mit repaint().

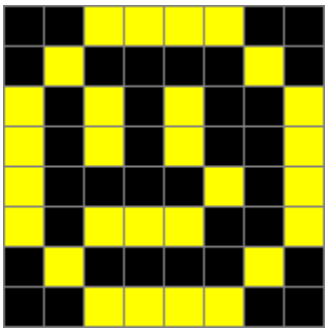


Farben kannst du durch Angabe des RGB-Farbtupels angeben, also beispielsweise color = (50, 10, 80). Einfacher ist es eine der Standardfarben RED, BLUE, YELLOW, GREEN, CYAN, MAGENTA, WHITE oder BLACK zu verwenden.

Um die Animation noch etwas variantenreicher zu machen, wartest du mit *sleep()* eine zufällige Zeit zwischen 10 und 100 ms, bist du in der Endlosschleife ein neues Bild zeichnest.

```
from oxocard import *
from random import randint

enableRepaint(False)
while True:
    for i in range(8):
        for k in range(8):
            r = randint(0, 50)
            g = randint(0, 50)
            b = randint(0, 50)
            dot(i, k, (r, g, b))
    repaint()
    sleep(0.001 * randint(10, 100))
```

Bilder animieren



Eine Animation besteht aus Einzelbildern, die sich nur wenig voneinander unterscheiden und Bild um Bild zeitlich nacheinander dargestellt werden. Die Bilder erstellst du am einfachsten mit dem Oxoeditor.

In deinem Programm animierst du ein Smiley mit den zwei Bildern.

```
from oxocard import *

matrix1 = (
(0x000000,0x000000,0xffff00,0xffff00,0xffff00,0xffff00,0x000000,0x000000),
(0x000000,0xffff00,0x000000,0x000000,0x000000,0x000000,0xffff00,0x000000),
(0xffff00,0x000000,0x000000,0xffff00,0x000000,0xffff00,0x000000,0xffff00),
(0xffff00,0x000000,0x000000,0xffff00,0x000000,0xffff00,0x000000,0xffff00),
(0xffff00,0x000000,0xffff00,0x000000,0x000000,0x000000,0x000000,0xffff00),
(0xffff00,0x000000,0x000000,0xffff00,0xffff00,0xffff00,0x000000,0xffff00),
(0x000000,0xffff00,0x000000,0x000000,0x000000,0x000000,0xffff00,0x000000),
(0x000000,0x000000,0xffff00,0xffff00,0xffff00,0xffff00,0x000000,0x000000))
matrix2 = (
(0x000000,0x000000,0xffff00,0xffff00,0xffff00,0xffff00,0x000000,0x000000),
(0x000000,0xffff00,0x000000,0x000000,0x000000,0x000000,0xffff00,0x000000),
(0xffff00,0x000000,0xffff00,0x000000,0xffff00,0x000000,0x000000,0xffff00),
(0xffff00,0x000000,0xffff00,0x000000,0xffff00,0x000000,0x000000,0xffff00),
(0xffff00,0x000000,0x000000,0x000000,0x000000,0xffff00,0x000000,0xffff00),
(0xffff00,0x000000,0xffff00,0xffff00,0xffff00,0x000000,0x000000,0xffff00),
(0x000000,0xffff00,0x000000,0x000000,0x000000,0x000000,0xffff00,0x000000),
(0x000000,0x000000,0xffff00,0xffff00,0xffff00,0xffff00,0x000000,0x000000))

while True:
    image(matrix1)
    sleep(0.2)
    image(matrix2)
    sleep(0.2)
```

MERKE DIR...

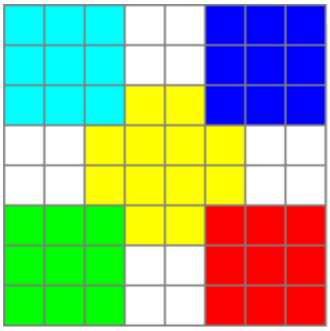
Bilder werden mit einem Tupel definiert, das Tupels mit den Farbwerten der einzelnen Zeilen enthält. Diese werden meist in 6-ziffriger Hexdarstellung angegeben, wobei immer 2 Hexziffern den einzelnen Farbkomponenten r, g, b entsprechen.

Um bestimmte Zeichenoperationen miteinander anzuzeigen, muss das automatische Rendering mit *enableRepaint(False)* abgeschaltet werden. Nachdem alle Zeichenoperationen abgeschlossen sind, wird *repaint()* aufgerufen.

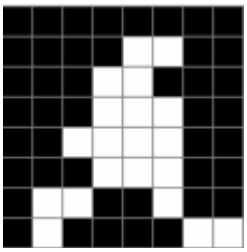
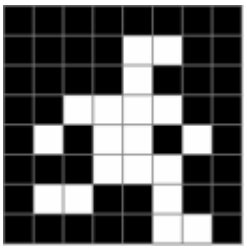
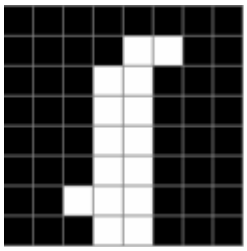
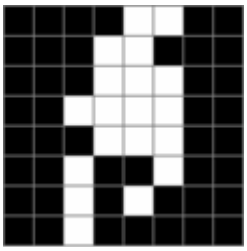
ZUM SELBST LÖSEN

1. Zeichne unter Verwendung des Oxoeditors das nebenstehende Farbmuster.

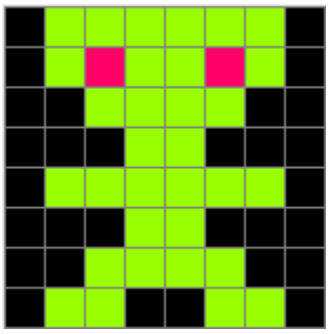
   Entwerfe selbst weitere schöne Bilder.



2. Erstelle eine Animimation einer nach rechts laufenden Figur mit den untenstehenden Bildern, die sich alle 0.08 s abwechseln.

3. Erstelle mit dem gleichen Verfahren weitere lustige Animationen, zum Beispiel mit diesem Alien-Bild.



11. STRINGS UND SCROLLTEXT

DU LERNST HIER...

wie du mit Strings umgehst und wie du sie als Lauftext auf dem Display darstellen kannst. Du lernst auch, wie du Textinformationen im Terminal-Fenster (REPL/Console) ausschreiben kannst.

STRINGS

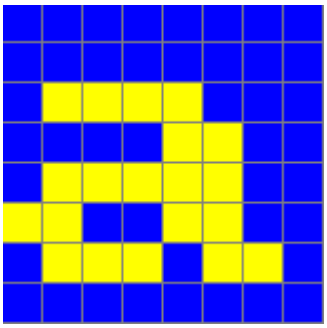
Wörter und Sätze sind aneinander gefügte Zeichen und werden als String bezeichnet. Zur Definition einer Stringvariablen verwendet man ein einfaches oder doppeltes Anführungszeichenpaar (Gänsefüsschen), also beispielsweise *vorname = 'Maya'* oder *vorname = "Maya"*. Wir verwenden meist das doppelte Anführungszeichen.

MUSTERBEISPIELE

Im ersten Beispiel willst du den Namen "Maya" auf dem Display Buchstaben um Buchstaben ausschreiben. Du verwendest *dazu insertBigChar(ch, textColor, bgColor)*, wobei *textColor* und *bgColor* auch weggelassen werden können. Beachte, dass du keine Umlaute und Accents verwenden darfst.

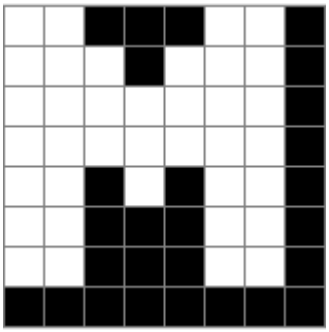
Um auf einzelne Buchstaben des Strings zuzugreifen, verwendest du einen Stellenindex, der bei 0 beginnt. *vorname[2]* ist also der String, der nur den Buchstaben "y" enthält.

```
from oxocard import *

vorname = "Maya"
for i in range(len(vorname)):
    insertBigChar(vorname[i], YELLOW, BLUE)
    sleep(0.5)
```



Eleganter durchläufst du den String mit einer verallgemeinerten for-Schleife:

```
from oxocard import *


vorname = "Maya"
for ch in vorname:
    insertBigChar(ch)
    sleep(0.5)
```



Du kannst Strings mit dem +-Zeichen aneinander fügen. Definierst du beispielsweise den Familienname mit name = "Berger", so erhältst du mit anschrift = vorname + name den String "MayaBerger". Willst du zwischen Vorname und Namen noch eine Leerzeichen, so schreibst du

anschrift = vorname + " " + name

Das Aneinanderfügen von Strings nennt man auch **Stringkonkatenation**.

```
from oxocard import *

vorname = "Maya"
name = "Berger"
anschrift = vorname + " " + name
for ch in anschrift:
    insertBigChar(ch)
    sleep(0.5)
```

Lauftexte und Ausschreiben mit einem einfachen Font

Statt einzelner Buchstaben kannst du auch den String als Lauftext (Scrolltext) ausschreiben, was besser lesbar ist.

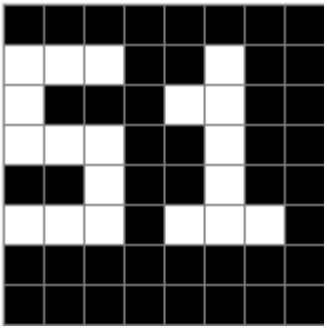
```
from oxocard import *


vorname = "Maya"
name = "Berger"
anschrift = vorname + " " + name
bigTextScroll(anschrift)
```

Es gibt auch Zeichen aus einem kleineren Font, das sich aber nur zum Ausschreiben von einfachen Texten und Zahlen eignet, da es nicht alle Zeichen, insbesondere keine Kleinbuchstaben enthält.

```
vorname = "MAYA"
name = "BERGER"
anschrift = vorname + " " + name
smallTextScroll(anschrift)
```

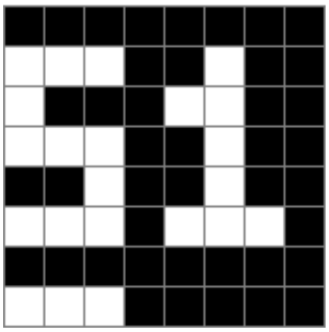
Für zweiziffrige Zahlen eignet sich das Ausschreiben mit *display().*

```
from oxocard import *

for n in range(100):
    display(n)
    sleep(0.1)
```



Rufst du *display(n)* mit einer negativen Zahl auf, so erscheint automatisch ein Vorzeichen unter der ersten Ziffer.

```
from oxocard import *

for n in range(-1, -100, -1):
    display(n)
    sleep(0.1)
```



Du verwendest in diesem Beispiel auch eine allgemeinere Form des *range()* Befehls, bei dem du den Anfangswert, den Endwert und das Inkrement angibst (der Endwert ist nicht mehr enthalten).

Ausschreiben in das Terminal-Fenster (REPL/Console)

Es ist oft zweckmässig, bestimmte Informationen zur Laufzeit auf dem Entwicklungssystem anzuzeigen Du verwendest dazu den *print()*-Befehl, der auch mehrere durch Kommas getrennte Parameter erlaubt. Nach jeder ausgeschriebenen Zeile wird automatisch ein Zeilenumbruch eingefügt.

```
from oxocard import *
from random import randint

print("Programm startet. Ich wuerfle...")
n = randint(1, 6)
print("Wuerfelzahl:", n)
print("Programm beendet")
```

```
----------------
Programm startet. Ich wuerfle...
Wuerfelzahl: 6
Programm beendet
```

Beachte, dass du auch hier keine Umlaute und Accents verwenden kannst.

MERKE DIR...

Strings sind eine spezielle Datenstruktur zu Speicherung von Texten. Du definierst sie mit einfachen oder doppelten Anführungszeichen. Beachte, dass es einen grossen Unterschied zwischen *a = "7"* und *m = 7* gibt. Das eine ist ein String und das andere eine Ganzzahl. Du kannst also sehr wohl *n = m + 1* schreiben, aber *b = a + 1* führt zu einer Fehlermeldung, da Strings und Zahlen nicht addiert werden können. Für die Umwandlung einer Zahl in einen String verwendest du *s = str("123")*, für die Umwandlung eines Strings in eine Zahl *z = int("123")*

ZUM SELBST LÖSEN

1. Schreibe einige lustige Texte, die endlos als Scrolltext ausgeschrieben werden.

2. Programmiere eine Uhr, die mit *display()* die Sekunden ausschreibt. Nach einer Minute soll wieder 0 angezeigt werden. Überprüfe, wie genau deine Uhr läuft und optimiere die Genauigkeit.

3. Schreibe ein Programm, das die Quadratzahlen von 1 bis 9 auf dem Display und gleichzeitig in der Console ausschreibt.

4. Mit x in range(100, -1, -1) durchläufst du alle Zahlen x von 100 bis und inklusive 0. Schreibe den Kehrwert von x in der Console aus. Was beobachtest du bei x = 0?

12. BUTTONS

DU LERNST HIER...

wie man die Oxocard-Tasten (Buttons) verwendet, um interaktive Programme zu entwickeln.

Oxocard-Buttons

Die Oxocard verfügt über 6 Buttons, die über die vordefinierten Konstanten BUTTON_L1, BUTTON_L2, BUTTON_L3, BUTTON_R1, BUTTON_R2 und BUTTON_R3 ansprechbar sind.

Es gibt zwei Möglichkeiten, die Button-Werte zu erfassen:

- **Mit Pollen**
  Es wird ständig abgefragt, ob ein Button gedrückt ist oder gedrückt wurde. Wenn dies der Fall ist, wird ein bestimmter Programmblock ausgeführt.
- **Mit Events**
  Ein Buttonklick wird als Ereignis (Event) aufgefasst. Dabei wird automatisch eine Funktion (Callbackfunktion) aufgerufen, in welcher definiert ist, wie das Programm reagieren soll.

MUSTERBEISPIELE

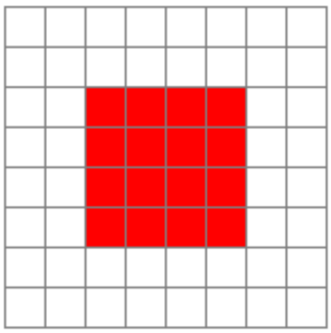
Pollen, um den aktuellen Status (gedrückt/losgelassen) zu erfassen

Mit button = Button(BUTTON_R2) erzeugst du ein Objekt der Klasse Button, mit dem du den rechten unteren Button ansprechen kannst. Dazu musst du zuerst das Modul *button* importieren. Die Klasse stellt dir verschiedene Funktionen zur Verfügung, die man auch **Methoden** nennt, da sie zu einer Klasse gehören.

Die Methode button.isPressed()

gibt *True* zurück, wenn der Button im Moment des Aufrufs gedrückt ist.

Dein Programm wartet in einer Endlosschleife, bis du den Button R2 drückst. Dann wird alle 0.1 Sekunden ein rotes Quadrat erscheinen und wieder verschwinden, und zwar so lange du den Button gedrückt hältst.

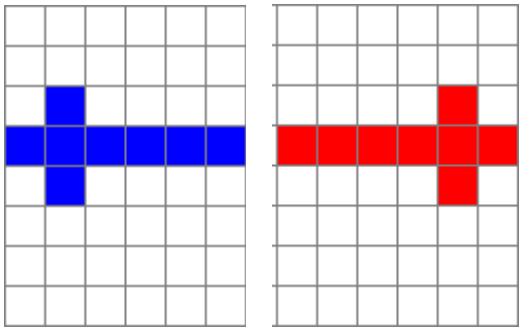


```
from oxocard import *
from oxobutton import *

button = Button(BUTTON_R2)
while True:
    if button.isPressed():
        fillRectangle(2, 2, 4, 4, RED)
        sleep(0.2)
        clear()
    sleep(0.2)
```

Pollen, um auf Klicken eines Buttons zu reagieren

So wie du es von den Mausklicks kennst, möchtest du hier mit einem Button-Klick eine Aktion ausführen. Du kannst aber ein kurzes Klicken nicht mit *isPressed()* erfassen, da dein Programm nicht mit Sicherheit gerade dann diese Funktion aufruft, wenn der Button gedrückt ist. Die Methode wasPressed() "erinnert" sich aber, ob der Button seit dem letzten Aufruf gedrückt und wieder losgelassen wurde.

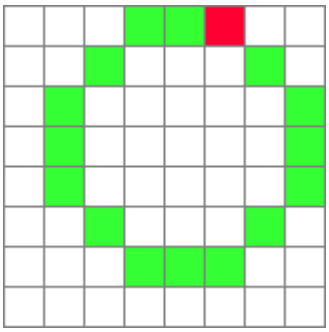
In deinem Beispiel verwendest du den linken und den rechten unteren Button, um einen Pfeil nach links bzw. nach rechts zu zeichnen. Das sleep(0.1) ist wichtig, damit du nicht unnötig viele Rechnerressourcen verschwendest, wenn das Programm nichts anderes machen muss, als zu überprüfen, ob ein Button gedrückt wurde. Wenn du die Wartezeit auf 1 s erhöhst, kannst du demonstrieren, dass der Klick tatsächlich auch dann erkannt wird, wenn das Programm gerade schläft.



```
from oxocard import *
from oxobutton import *

button1 = Button(BUTTON_R2)
button2 = Button(BUTTON_L2)

while True:
    if button1.wasPressed():
        clear()
        arrow(3, 3, 0, 5, RED)
    if button2.wasPressed():
        clear()
        arrow(4, 3, 4, 5, BLUE)
    sleep(0.1)
```

Buttons werden häufig dazu verwendet, um ein laufendes Programm zu beenden. Statt einer endlosen while-Schleife, lässt du die Schleife nur solange laufen, bis ein bestimmter Button geklickt wurde.

In deinem Beispiel bewegt sich die Schlange so lange auf dem Kreis, bis du den Button R2 klickst. Dann versteckst du sie mit hide().



```
from oxosnake import *
from oxobutton import *

makeSnake(speed = 100)
button = Button(BUTTON_R2)

while not button.wasPressed():
    forward(2)
    right(45)
    sleep(0.1)
hide()
```

Buttonklicks als Events erfassen

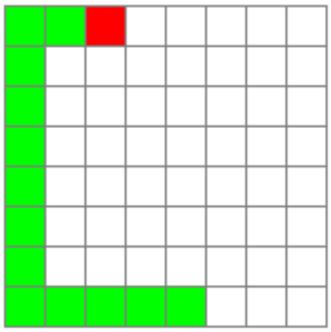
Die Ereignissteuerung verlangt eine spezielle Programmiertechnik: Du definierst dazu in einer Funktion (**Callbackfunktion**), was beim Auftreten des Events (hier ein Buttonklick) geschehen soll. Die Callbackfunktion wird nie explizit von deinem Programm aufgerufen, sondern direkt vom System, wenn der Button geklickt wurde. Du teilst dem System den Namen deiner Callbackfunktion beim Erzeugen des Buttons mit, z.B. wenn die Callbackfunktion *onEvent* heisst:

Button(BUTTON_KONSTANTE, onEvent)

Man nennt dies auch "**Registrieren des Callbacks**".

Die Verwendung von Events ist insbesondere dann vorteilhaft, wenn du in ein laufendes Programm eingreifen willst. Im nächsten Beispiel bewegt sich die Schlange endlos auf einem Quadrat. Ohne die Bewegung zu unterbrechen, kannst du mit dem rechten Button die Länge der Schlange vergrössern.

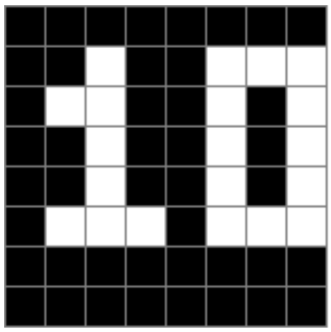
Im Hauptprogramm musst du die Callbackfunktionen beim Erzeugen des Button R2 registrieren.



```
from oxosnake import *
from oxobutton import *

def onClick(pin):
    growTail()

makeSnake(pos = (0, 7))
Button(BUTTON_R2, onClick)
```

```
while True:
    forward(7)
    right(90)
```

Du möchtest einen Klickzähler so realisieren, dass auf dem Display die Anzahl der Tastenklicks der Taste BUTTON_R2 angezeigt wird, Du speicherst in der Variablen *count* die Anzahl der Tastenklicks. Zuerst initialisierst du *count* auf 0 und stellst den aktuellen Wert ständig mit *display()* dar.



In der Callbackfunktion möchtest du *count* um eins erhöhen. In einer Funktion kannst du aber eine im Hauptprogramm definierte Variable nur dann verändern, wenn du sie als global bezeichnest.

```
from oxocard import *
from oxobutton import *

def onClick(pin):
    global count
    count += 1

count = 0
Button(BUTTON_R2, onClick)
while True:
    display(count)
    sleep(0.1)
```

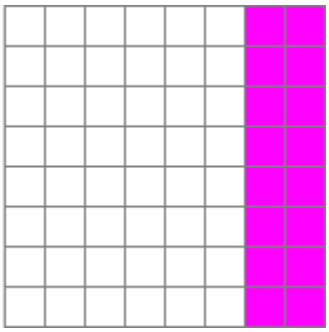
MERKE DIR...

Beim Pollen eines Buttons überprüfst du ständig mit der Methode *button.isPressed()*,ob der Button gerade gedrückt ist oder mit *button.wasPressed()*, ob der Button seit dem letzten Aufruf dieser Methode gedrückt wurde.

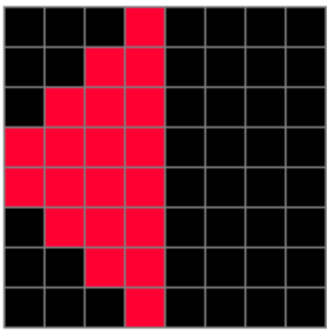
Bei der Eventsteuerung definierst du in einer Callbackfunktion, wie auf ein Buttonklick zu reagieren ist. Die Callbackfunktion muss beim Erzeugen des Buttons registriert werden.

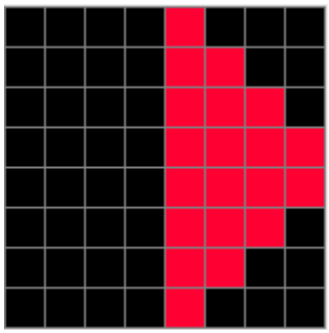
ZUM SELBST LÖSEN

1.a Wenn du den rechten Button klickst, sollen LEDs in den zwei Spalten rechts leuchten. Beim Klick des linken Buttons sollen die zwei linken Spalten leuchten.
Verwende den Befehl *rectangle(x, y, w, h, color)*, um die Streifen zu zeichnen.

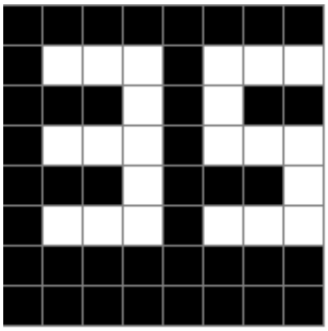


1.b Erstelle mit dem *Oxoeditor* zwei

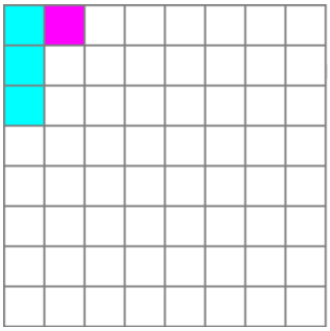




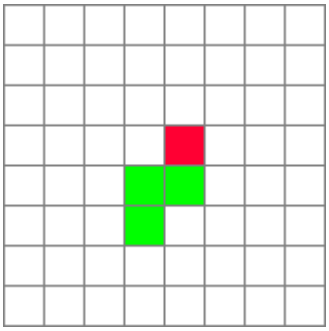
2. Im oben gezeigten Beispiel reagiert das Programm auf Buttonklick und zeigt bei jedem Klick eine um 1 grössere Zahl an.
   a) Löse die gleiche Aufgabe mit Pollen
   b) Dein Programm soll zusätzlich beim Klicken des linken Buttons eine um 1 kleinere Zahl anzeigen. Löse das Problem mit Callbacks oder mit Pollen und verhindere, dass negative Zahlen oder Zahlen über 99 vorkommen.



3. Die Schlange soll sich endlos auf dem äusseren Quadrat bewegen. Mit einem rechten Buttonklick willst du die Kopffarbe auf MAGENTA und die Schwanzfarbe auf CYAN und mit einem linken Buttonklick die Kopffarbe auf GREEN und die Schwanzfarbe auf YELLOW setzen.



4. Programmiere ein einfaches Snake-Game: Die Schlange bewegt sich in einer endlosen while-Schleife schrittweise vorwärts. Du kannst mit einem Klick auf den rechten bzw. linken Buttonum 90° nach rechts bzw. nach links drehen. Das Spiel wird beendet, sobald die Schlange über den Rand hinausläuft. Nach jeweils 5 s wird die Geschwindigkeit der Schlange vergrössert. Ziel ist es, eine möglichst hohe Geschwindigkeit zu erreichen. Zeige die erreichte Geschwindigkeit am Ende des Spiels auf dem Display dar.



13. BESCHLEUNIGUNGSSENSOR

DU LERNST HIER...

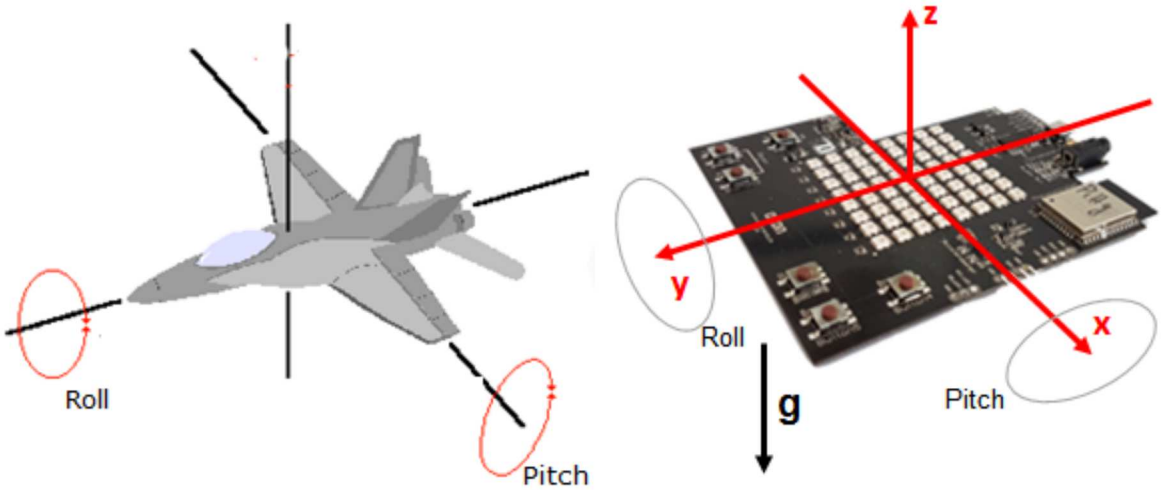
wie du mit dem Beschleunigungssensor Lageänderungen und Bewegungen der Oxocard erfassen kannst.

SENSORWERTE

Der Sensor misst die konstante Erdbeschleunigung von rund 10 m/s^2 und kann damit Lageänderungen der Oxocard erfassen, aber auch Beschleunigungen, die auf Grund von Bewegungsänderungen der Oxocard auftreten, beispielsweise weil man ihr einen Stoss gibt.

Der Sensor (Accelerometer) ist auf der Oxocard gut sichtbar. Ähnliche Sensoren sind auch in den meisten Smartphones eingebaut.

Wie bei Flugzeugen, kann man die Neigung der Oxocard gegenüber der Erdoberfläche bestimmen. Wenn man die Karte nach vorne oder nach hinten neigt, ändert der Pitch-Winkel bei einer Seitwärtsneigung der Roll-Winkel.



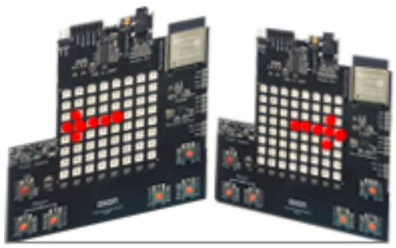
MUSTERBEISPIELE

Pitch und Roll

Den Sensor werden wir wieder als ein Objekt auffassen, das Eigenschaften und Fähigkeiten hat. Es ist in der Klasse *Accel* modelliert. Du erzeugst das Objekt mit mit acc = Accel.create() und verwendest die Methoden getRoll() bzw. getPitch(), um die momentanen Werte von Roll und Pitch zu erhalten. Kippe die Oxocard ausgehend von der horizontalen Lage nach vorne und hinten beobachte dabei die Sensorwerte, die im Terminalfenster ausgeschrieben werden.

```
from oxocard import *
from oxoaccelerometer import *

acc = Accelerometer.create()

while True:
    roll = acc.getRoll()
    pitch = acc.getPitch()
    print("pitch:", pitch, "roll:", roll)
    sleep(0.3)
```

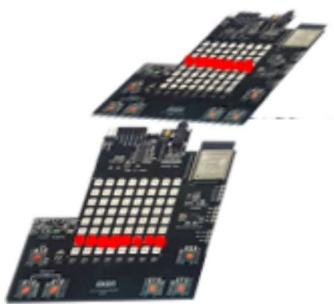
Als Anwendung schreibst du eine Anwendung, bei der die Rollposition auf dem Display sichtbar ist. Drehst du die Oxocard nach rechts, erscheint auf dem Display ein Rechtspfeil, drehst du sie nach links, erscheint ein Linkspfeil. Um ein Flackern zu vermeiden, wenn der Pfeil gelöscht und wieder neu dargestellt wird, schaltest du mit *enableRepaint(False)* das automatische Rendering ab und stellst das vollständig aufgebaute Bild mit *repaint()* dar.



```
from oxocard import *
from oxoaccelerometer import *

acc = Accelerometer.create()
enableRepaint(False)

while True:
    roll = acc.getRoll()
    clear()
    if roll >= 0:
        arrow(3, 3, 0, 5, RED)
    if roll <= 0:
        arrow(4, 3, 4, 5, RED)
    repaint()
    print("roll = ", roll)
    sleep(0.3)
```

Hier bewegt sich die ganze LED-Reihe beim Kippen der Oxocard nach vorne oder nach hinten.

Zuerst wird eine Linie in der Mitte des Displays gezeichnet. Ist der Pitch-Winkel negativ, wird die y-Koordinate um 1 vergrössert, ist er positiv, wird sie um 1 verkleinert.



```
from oxocard import *
from oxoaccelerometer import *

acc = Accelerometer.create()
enableRepaint(False)
y = 3
while True:
    pitch = acc.getPitch()
    if pitch > 0 and y < 7:
        y += 1
    if pitch < 0 and y > 0:
        y -= 1
    clear()
    line(0, y, 0, 8, RED)
    repaint()
    sleep(0.1)
```

Pollen und Ausschreiben der Sensorwerte

Die Methoden *getX()*, *getY()* und *get(Z)* liefern die Beschleunigungskomponenten in den drei Koordinatenrichtungen. Falls der Sensor nur gekippt ist, aber sonst ins Ruhe ist, entspricht dies den Komponenten der Erdbeschleunigung je im Bereich -9.81 und 9.81 m/s2. Mit *getValues()* kannst du auch ein Tupel mit allen drei Werten gleichzeitig holen. Für viele Sensortests schreibst du die periodisch gemessenen Werte einfach in die Konsole. Man sagt auch, dass man den Sensor "polle".

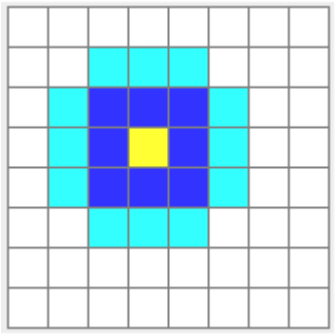
```
from oxocard import *
from oxoaccelerometer import *

acc = Accelerometer.create()

while True:
    accX = acc.getX()
    accY = acc.getY()
    accZ = acc.getZ()
    print(accX, accY, accZ)
   #print("(acc_x, acc_y, acc_z) = (5.2f, %5.2f, %5.2f)" %(accX, accY, accZ))
    sleep(0.1)
```

Es ist üblich, beim Ausschreiben der Werte auch anzugeben, um welchen Wert es sich handelt. Am besten machst du das mit einer Formatierungsangabe (auskommentierte Zeile). Mit *%5.2f* legst du fest, dass die Werte als Float auf 2 Nachkommastellen gerundet und einer Feldbreite von 5 Zeichen ausgeschrieben werden. Dies hat auch den Vorteil, dass die Werte exakt untereinander erscheinen.

Wasserwaage mit einer Libelle

Du kannst mit der Oxocard ein Messgerät bauen, das wie eine Wasserwaage-Libelle funktioniert. Dazu zeichnest du einen Punkt entsprechend der momentanen Werte der Beschleunigungskomponenten in x- und y-Richtung.

Gleichzeitig willst du auch eine kreisartige Markierung zeichnen. Da du bei jeder Messung das Bild löschen und neu aufbauen musst, handelt es sich um eine Animation und du solltest mit *enableRepaint(False)* das automatische Rendern abschalten und mit *repaint()* selbst rendern.

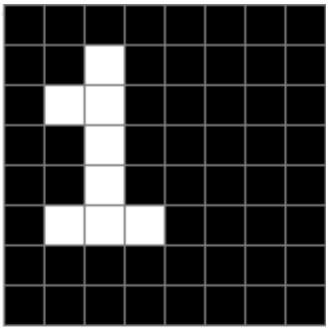


```
from oxocard import *
from oxoaccelerometer import *


acc = Accelerometer.create()

x = 3
y = 3
enableRepaint(False)
while True:
    accX = acc.getX()
    accY = acc.getY()
    x = max(min(int(3 - accX), 5), 1)
    y = max(min(int(3 + accY), 5), 1)
    clear()
    circle(3, 3, 2, CYAN)
    fillRectangle(2, 2, 3, 3, BLUE)
    dot(x, y, YELLOW)
    repaint()
    sleep(0.1)
```

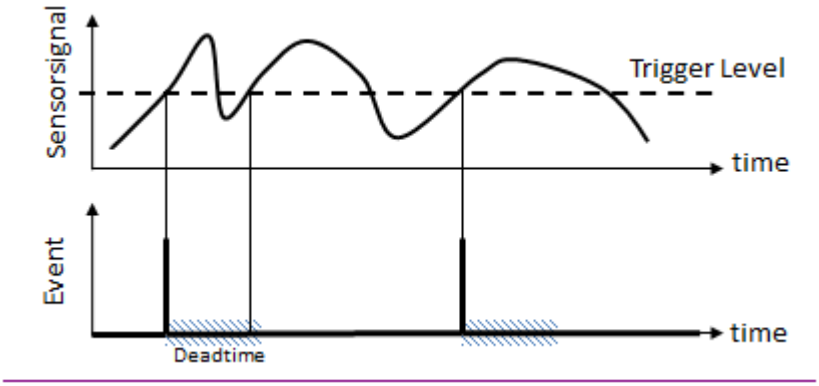
Mit Events schnelle Beschleunigungsänderungen erfassen

Um schnelle Beschleunigungsänderungen zu erfassen, ist das Pollen des Sensors nicht geeignet, da diese weniger als 1 Millisekunde dauern können und das Pollintervall nicht genügend klein gewählt werden kann. Darum müssen solche Ereignisse mit dem Eventmodell erfasst werden, wobei der Sensor einen Event auslöst, wenn die Beschleunigung einen bestimmten Wert (Triggerpegel) überschreitet. Dabei wird eine Callbackfunktion aufgerufen, in welcher du festlegst, wie das Programm reagieren soll.

Hier wird ein Event mit leichtem Klopfen (Tap) auf die Oxocard ausgelöst. In der Callbackfunktion onTap() erhöhst du lediglich eine Zählvariable n um 1. Im Hauptprogramm wird der Wert auf dem Display angezeigt. Wie du bereits weisst, muss die Callbackfunktion registriert werden. Dies erfolgt beim Erzeugen des Sensors. Dabei kannst du auch *trigger_level* und *rearm_time* festlegen. Kleinere Werte beim trigger_level machen den Sensor empfindlicher.



rearm_time ist die Zeit in Sekunden, während der der Sensor inaktiviert wird, bis er wieder Events registriert. Eine solche "Totzeit" ist oft nötig, damit der Sensor nicht mehrere Ereignisse kurz hintereinander auslöst.




```
from oxocard import *
from oxoaccelerometer import *

def onTap():
    global n
    n += 1

Accelerometer.create(onTap, trigger_level = 10, rearm_time = 0.2)

n = 0
print("Tap detector started")
while True:
    display(n)
    sleep(0.1)
```

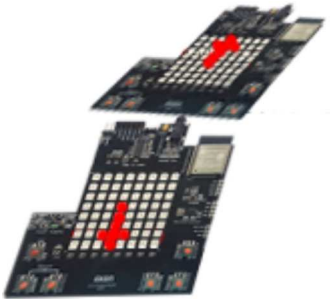
MERKE DIR...

Mit dem Beschleunigungssensor kannst du die Lage und Bewegungsänderungen der Oxocard erfassen. Meist werden die Sensorwerte mit Pollen, d.h. in einer while-True Schleife regelmässig abgefragt.

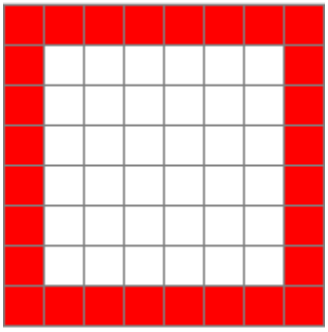
Schnelle Änderungen müssen mit dem Eventmodell erfasst werden. Ein Event wird ausgelöst, wenn der Sensorwert einen Triggerpegel überschreitet. In einer Callbackfunktion definierst du, wie das Programm reagieren soll.

ZUM SELBST LÖSEN

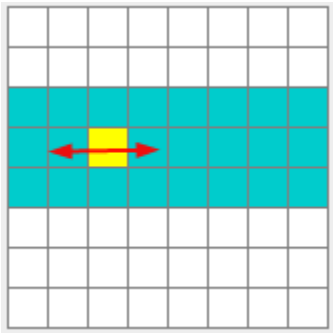
1. Schreibe ein Programm so, dass beim Kippen der Oxocard nach vorne und nach Hinten die passenden Pfeile angezeigt werden.



2. Durch Kippen der Oxocard kannst du alle LEDs am Rand der Reihe nach einschalten.



3. Konstruiere eine Wasserwaage. Dabei soll sich auf einem Balken eine Markierung je nach Kipplage nach links oder rechts bewegen.



14. WEBSERVER, IOT

DU LERNST HIER...

wie du einen Webserver und sogar einen Accesspoint auf der Oxocard einrichten kannst, um über das WLAN mit einem Smartphone oder PC mit der Oxocard zu kommunizieren. In den Musterbeispielen lernst du auch, wie man ferngesteuerte Geräte programmiert.

Du erhältst damit Einblick in die modernen Gebiete der Rechnerkommunikation und des "Internet der Dinge" (IOT, Internet of Things).

Um die Einzelheiten zu verstehen, benötigst du einige Kenntnisse von HTTP, HTML (und wenig JavaScript), die du dir am besten mit einem Web-Tutorial aneignest.

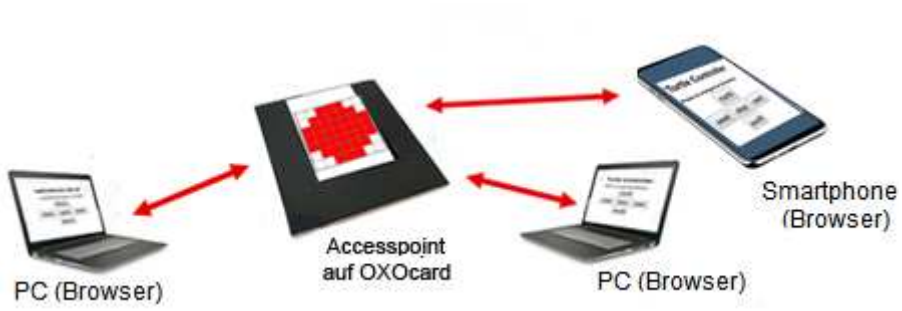
ZWEI VERBINDUNGSMÖGLICHKEITEN

Damit zwei Rechner miteinander Informationen austauschen können, müssen sie über einen Datenkanal miteinander verbunden sein. Bei der Oxocard wird dabei das WLAN verwendet und als Verbindungsprotokoll TCP/IP eingesetzt. Die Verbindung erfolgt über ein spezielles Gerät, das man Accesspoint, Hotspot oder WLAN-Router nennt. (Kennst du diese Begriffe nicht, so orientiere dich im Internet. Du verwendest dieselben Verfahren wie mit deinem Smartphone.)

Für die TCP/IP-Kommunikation mit der Oxocard gibt es zwei Szenarien:

Verwendung eines vorhandenen Accesspoints
Die Oxocard loggt sich auf einem bestehenden Accesspoint ein. Ein anderes Gerät, beispielsweise dein PC oder Smartphone, das mit ihr kommunizieren will, loggt sich ebenfalls auf demselben Accesspoint ein (oder ist über das Internet mit diesem Accesspoint verbunden).

Die Oxocard als Accesspoint
Die Oxocard ist selbst ein Accesspoint mit einer eigenen SSID und die anderen Geräte loggen sich über WLAN auf diesem Accesspoint ein.



MUSTERBEISPIELE

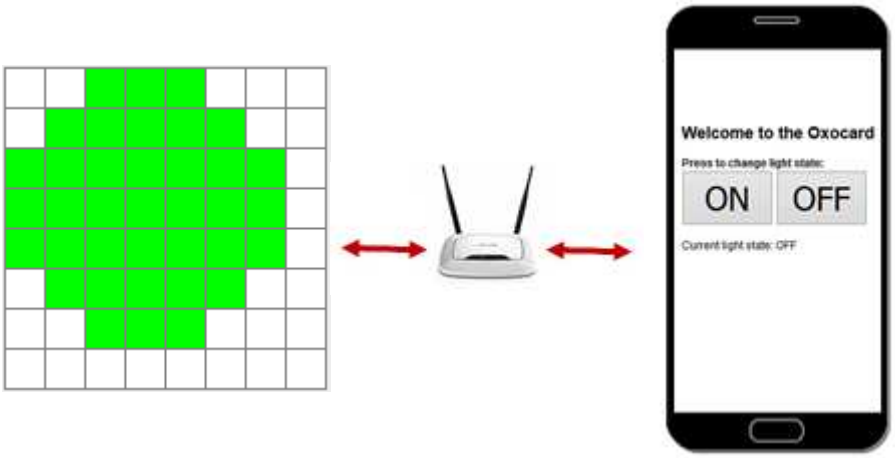
Das Modul ***tcpcom*** einhält mehrere Klassen, um die Programmierung von Webapplikationen stark zu vereinfachen. Mit der Klasse *Wlan* kann man sich auf einem bestehenden Accesspoint einloggen oder einen eigenen Accesspoint starten. Die Klasse *HTTPServer* enthält einen einfachen Webserver. Erzeugt man mit *HTTPServer(requestHandler = onRequest)* ein Serverobjekt, so wird der Server auch gleich auf Port 80 gestartet.

Dieser wartet auf auf diesem Port auf einen Client. Trifft ein HTTP-Request eines Clients ein, so wird die Callbackfunktion *onRequest()* aufgerufen, wo du festlegst, wie der Server reagieren soll und welchen HTTP-Response er dem Client zurück sendet. Als Client kannst du den Browser eines Smartphones oder eines beliebigen Computers verwenden.

A. Vorhandenen Accesspoint verwenden

Im ersten Beispiel realisierst du eine Fernsteuerung (Remote control). Dabei kannst du mit einem Webbrowser einen grünen Farbkreis ein- oder ausgeschalten. Stattdessen könnte es sich aber auch um irgend ein anderes Gerät handeln, dass du ein- oder ausschalten willst (Motor, Lampe, Raumheizung, usw.).

Zuerst loggt sich die Oxocard mit *Wlan.connect()* auf einem Accesspoint mit einer SSID und einem (leeren) Passwort ein. (Diese Zeile musst du entsprechend den Angaben deines Accesspoints anpassen).



Nachfolgend wird der Webserver automatisch beim Erzeugen des HTTPservers gestartet. Er funktioniert eventgesteuert mit der Callbackfunktion onRequest(), die bei einem *eingehenden GET-Request* aufgerufen wird. Sie hat drei Parameter clientIP, filename, params. *clientIP* liefert die IP-Adresse des Clients, *filename* ist der Name der im GET-Request angeforderten Datei. *params* liefert die Parameter des GET-Requests als Tupel in der Form (key, value).

In der Callbackfunktion wird hier lediglich eine Variable *state* auf den Wert "ON" oder "OFF" gesetzt, auf die in der Endlosschleife des Hauptprogramm getestet wird. Verändert sich ihr Wert, so wird der Farbkreis ein- oder ausgeschaltet (*state* muss daher als *global* bezeichnet werden).

Für den HTTP-Response verwendest du den vordefinierten String *html*, wobei der aktuelle Wert des Zustands mit der Formatzeile

*Current light state: %s
*

eingebaut wird. Der return-Wert von *onRequest()* wird automatisch als HTTP-Response an den Client übertragen.

```
from tcpcom import *
from oxocard import *

html = """<!DOCTYPE html>
<html>
  <head> <title>Oxocard Light</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
     <h1>Welcome to the Oxocard</h1>
     <b>Press to change light state:</b>
     <form method="get">
       <input type="submit" style="font-size: 50px;
           height: 90px; width: 150px" name="light" value="ON" />
       <input type="submit" style="font-size: 50px;
           height: 90px; width: 150px" name="light" value="OFF" />
     </form>
     <br>
    Current light state: %s<br>
  </body>
</html>
"""

def onRequest(clientIP, filename, params):
    global state
    if len(params) > 0:
        state = params[0][1]
    return html%(state)

state = "OFF"
oldState = ""
dot(0, 0, BLUE)
sleep(0.2)
clear()
if not Wlan.connect("ssid", "password"):
    print("Connection failed")
else:
    print("Successfully connected to AP. IP:", Wlan.getMyIPAddress())
    HTTPServer(requestHandler = onRequest)
    clear()
```

```
    while True:
        if oldState != state:
            if state == "ON":
                fillCircle(3, 3, 3, GREEN)
            if state == "OFF":
                clear()
            oldState = state
        sleep(0.1)
```

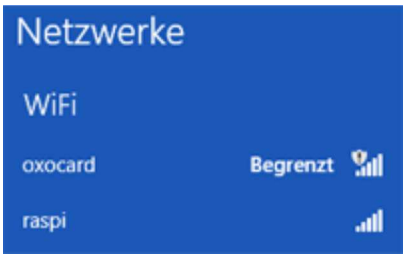
Nach dem Programmstart wird im Console-Fenster die IP-Adresse des Webservers angezeigt, die du im Browser auf dem Smartphone oder auf PC eingeben musst. (Etwas schöner wäre es, diese Adresse auf dem LED-Display der Oxocard als Scrolltext auszuschreiben).

Das Tag <meta name="viewport"> bewirkt, dass die Textgrösse der Fenstergrösse des Webbrowsers angepasst wird. Damit wird die Webseite auf Smartphones vergrössert dargestellt.

B. Oxocard als Accesspoint

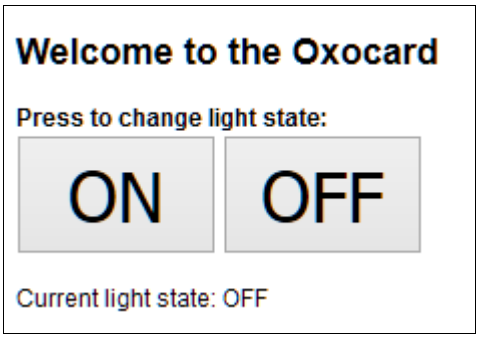
Auf der Oxocard kann auch ein Accesspoint aktiviert werden. Du kannst dann mit einem PC oder Smartphone im selben Raum direkt auf die Oxocard zugreifen. (Allerdings beeinflussen sich mehrere Accesspoints im gleichen Raum gegenseitig, was andere WLAN-Nutzungen stören kann)

Die Funktion Wlan.activateAP(ssid = "oxocard", password = "") aktiviert den Accesspoint. Du kannst selbstverständlich die SSID und das Passwort ändern.

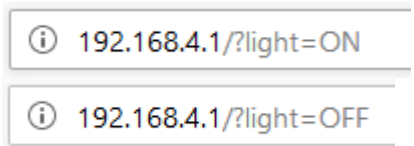


Für Clients, die sich im Zugriffsbereich des Accesspoints befinden, ist die SSID *oxocard* in den WLAN-Einstellungen sichtbar und die Clients können sich (mit leerem Passwort) einloggen.

Um auf den Webserver der Oxocard zuzugreifen, musst du einen Browser starten und als URL die fixe IP-Adresse **192.168.4.1** eingeben.



Beobachte in der Statuszeile des Browsers, wie die Werte ON/OFF als Request-Parameter gesendet werden.



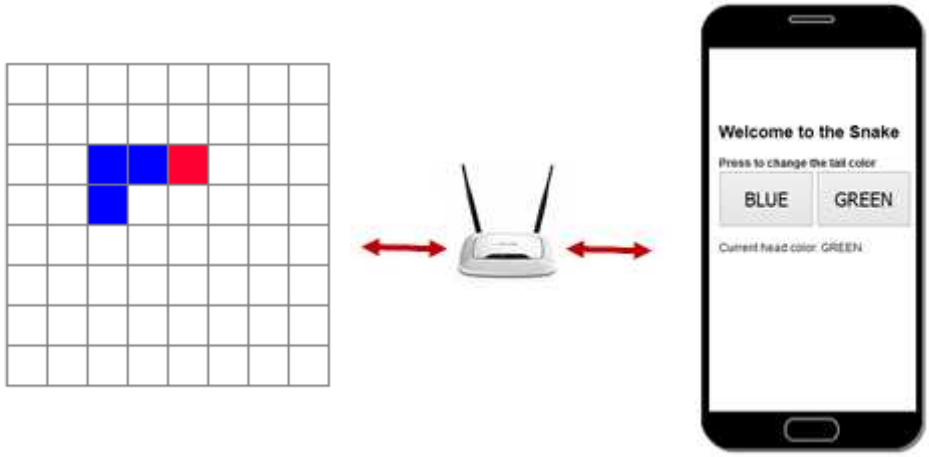

```
from tcpcom import *
from oxocard import *

html = """<!DOCTYPE html>
<html>
  <head> <title>Oxocard Light</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
```

```
  </head>
  <body>
     <h2>Welcome to the Oxocard</h2>
     <b>Press to change light state:</b>
     <form method="get">
       <input type="submit" style="font-size: 50px;
           height: 100px; width: 150px" name="light" value="ON" />
       <input type="submit" style="font-size: 50px;
           height: 100px; width: 150px" name="light" value="OFF" />
     </form>
     <br>
    Current light state: %s<br>
  </body>
</html>
"""

def onRequest(clientIP, filename, params):
    global state
    if len(params) > 0:
        state = params[0][1]
    return html%(state)

state = "ON"
oldState = ""
Wlan.activateAP(ssid = "oxocard", password = "")
HTTPServer(requestHandler = onRequest)
while True:
    if oldState != state:
        if state == "ON":
            fillCircle(3, 3, 3, GREEN)
        if state == "OFF":
            clear()
        oldState = state
    sleep(0.1)
```

WEITERE ANWENDUNGEN

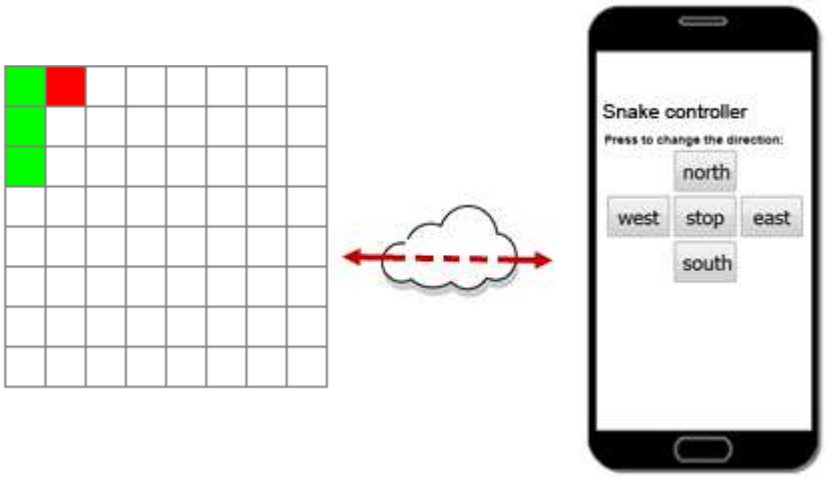
Bei den folgenden Programmen kannst du selbst entscheiden, ob du einen vorhandenen Accesspoint oder den Accesspoint der Oxocard verwenden möchtest, musst aber den Programmcode entsprechend anpassen.

Im folgenden Beispiel willst du ferngesteuert die Farbe der Schlange ändern. Im Hauptprogramm bewegst du die Schlange endlos auf einem Quadrat. Beim Klick auf die Schaltflächen "BLUE" bzw. "GREEN" sendet der Client einen GET-Request, wodurch die Callbackfunktion onRequest() aufgerufen wird. Hier legst du mit *setTailColor()* die Farbe des Schwanzes fest, sendest aber auch noch die so gesetzte Farbe im HTTP-Response an den Client zurück, der sie anzeigt.




```
from tcpcom import *
from oxosnake import *

html = """<!DOCTYPE html>
<html>
  <head> <title>Oxocard Snake</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
     <h1>Welcome to the Snake</h1>
     <b>Press to change the tail color</b>
     <form method="get">
       <input type="submit" style="font-size: 35px;
           height: 90px; width: 150px" name="color" value="BLUE" />
       <input type="submit" style="font-size: 35px;
           height: 90px; width: 150px" name="color" value="GREEN" />
     </form>
     <br>
    Current color: %s<br>
  </body>
</html>
"""

def onRequest(clientIP, filename, params):
    state = "GREEN"
    if len(params) > 0:
        state = params[0][1]
        if state == "BLUE":
            setTailColor(CYAN)
        if state == "GREEN":
            setTailColor(GREEN)
    return html%(state)

makeSnake()
if not Wlan.connect("ssid", "pasword"):
    print("Connection failed")
else:
    print("Successfully connected to AP. IP:", Wlan.getMyIPAddress())
    HTTPServer(requestHandler = onRequest)
    while True:
        forward(4)
        right(90)
```

Im nächsten Beispiel wird die Snake mit 5 Schaltflächen von einem Smartphone oder PC aus ferngesteuert.



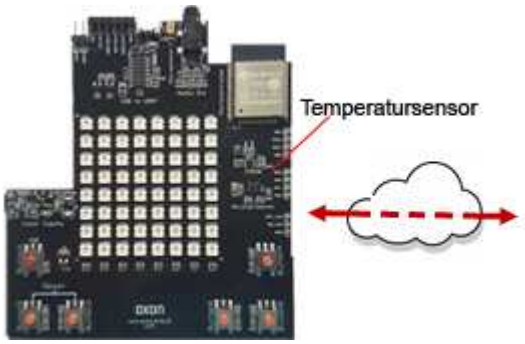
Der HTML-Code ist etwas umfangreicher, da für die Anordnung der Schaltflächen eine Tabelle verwendet wird. Beim Klick auf einen Button bewirkt der an die Oxocard gesendete HTTP-Request wiederum den Aufruf der Callbackfunktion *onRequest()*. Der Wert von *params[0][1]* liefert den Namen des gedrückten Buttons (d.h. "north", "west", "south". "east" oder "stop") und dieser wird in der Variablen *state* gespeichert. Im Hautprogramm wird in einer endlosen *while*-Schleife (ausser im Fall von state = 'stop') *forward()* aufgerufen und und die Bewegungsrichtung entsprechend geändert. (Das *sleep(0.1)* verhindert, dass im Fall von *state* = *'stop'* in der Schleife unnötig viel Rechenzeit vergeudet wird.)

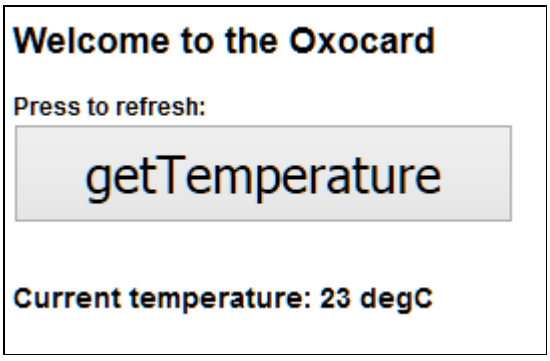
```
from tcpcom import *
from oxosnake import *

html = """<!DOCTYPE html>
<html>
  <head> <title>Oxocard Snake</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
    <h2>Welcome to the Snake</h2>
    <b>Press to change the tail color</b>
    <form method="get">
    <table>
    <tr>
      <td> </td>
      <td><input type="submit" style="font-size:24px; height:50px; width:75px"
          name="btn" value="north"/></td>
      <td> </td>
    </tr>
    <tr>
      <td><input type="submit" style="font-size:24px; height:50px; width:75px"
          name="btn" value="west"/></td>
      <td><input type="submit" style="font-size:24px; height:50px; width:75px"
          name="btn" value="stop"/></td>
      <td><input type="submit" style="font-size:24px; height:50px; width:75px"
          name="btn" value="east"/></td>
    </tr>
    <tr>
      <td> </td>
```

```
      <td><input type="submit" style="font-size:24px; height:50px; width:75px"
          name="btn" value="south"/></td>
      <td> </td>
    </tr>
    </table>
    </form><br>
    Current state: %s<br>
  </body>
</html>
"""

def onRequest(clientIP, filename, params):
    global state
    if len(params) > 0:
        state = params[0][1]
    return html%(state)

makeSnake()
state = "stop"
Wlan.activateAP(ssid = "oxocard", password = "")
HTTPServer(requestHandler = onRequest)
while True:
    if state == 'north':
        setHeading(0)
    elif state == 'west':
        setHeading(-90)
    elif state == 'south':
        setHeading(180)
    elif state == 'east':
        setHeading(90)
    if state != 'stop':
        forward()
    sleep(0.1)
```

Temperatur messen und auf Smartphone oder PC anzeigen

Oft müssen Messungen an weit entfernten Orten durchgeführt und die Werte in Echtzeit an eine Auswertestelle übertragen werden. Zur Demonstration dieses Szenarios wird mit dem Temperatursensor, der im Beschleunigungssensor der Oxocard eingebaut ist, die aktuelle Temperatur gemessen und über WLAN an ein Notebook oder Smartphone übertragen und dort im Browser angezeigt. Die Temperatur erhältst du durch Aufruf der Methode getTemperature() des Accel-Objekts. Sie entspricht aber der Temperatur im Innern des Accelerometer-Chips und steigt an, je länger die Oxocard in Betrieb ist. Um realistischere Werte für die Umgebungstemperatur zu erhalten, musst du die Oxocard aus dem Gehäuse nehmen.

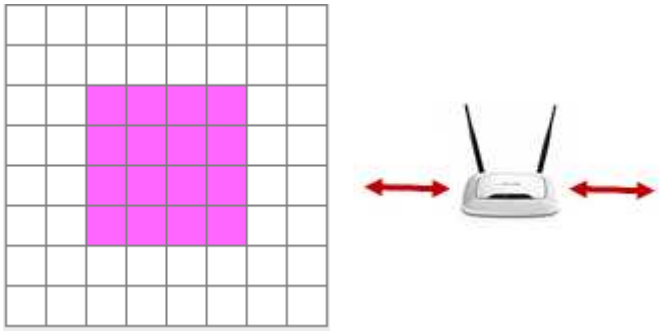


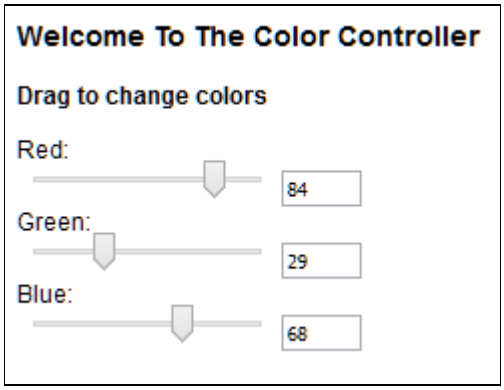


```
from tcpcom import *
from oxocard import *
from oxoaccelerometer import *

html = """<!DOCTYPE html>
<html>
  <head> <title>Oxocard Temperature </title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
     <h2>Welcome to the Oxocard</h2>
      <b>Press to refresh:</b>
     <form method="get">
       <input type="submit" style="font-size: 25px;
       height: 50px; width:250px" name="temp" value="getTemperature"/>
     </form>
     <br>
     <h3>Current temperature: %d degC</h3>
  </body>
</html>
"""

def onRequest(clientIP, filename, params):
    return html%(temp)

Wlan.activateAP(ssid = "oxocard", password = "")
HTTPServer(requestHandler = onRequest)
acc = Accelerometer.create()

while True:
    temp = acc.getTemperature()
    display(temp)
    sleep(1)
```

Oxocardfarben mit Schieberegler einstellen

Der HTML-Code lässt sich beliebig erweitern und du kannst natürlich auch JavaScript und CSS verwenden. In diesem Beispiel willst du mit 3 Schiebereglern die Farbe eines Quadrats einstellen. Der JavaScript-Code sorgt für ein interaktives Verhalten der Schieberegler, so wie du es von einer PC-Applikation gewohnt bist: Während des "Ziehens" wird der aktuelle Wert ständig angezeigt und beim "Loslassen" wird der HTTP-Request (als Submit) automatisch an den Webserver auf der Oxocard gesendet.





```
from tcpcom import *
from oxocard import *

html = """<!DOCTYPE html>
<html>
  <head>
    <title>Oxocard Color Controller</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <script>
      function showText(color, value)
      {
        switch(color)
        {
          case 0:
            document.forms[0].redtext.value = value; break;
          case 1:
            document.forms[0].greentext.value = value; break;
          case 2:
            document.forms[0].bluetext.value = value; break;
        }
      }

      function submitValues(color, value)
      {
        document.forms[0].submit();
      }
    </script>
  </head>
  <body>
     <h3>Welcome To The Color Controller</h3>
     <b>Drag to change colors </b>
     <p>
     <form method="get">
        Red:<br>
        <input type="range" min="0" max="100" value="%s"
        oninput="showText(0, value)" onchange="submitValues(0, value)">
        <input size="5" name="redtext" value="%s"><br>

        Green:<br>
        <input type="range" min="0" max="100" value="%s"
        oninput="showText(1, value)" onchange="submitValues(1, value)">
        <input size="5" name="greentext" value="%s"><br>

        Blue:<br>
        <input type="range" min="0" max="100" value="%s"
        oninput="showText(2, value)" onchange="submitValues(2, value)">
        <input size="5" name="bluetext" value="%s">
     </form></p>
  </body>
</html>
"""

def onRequest(clientIP, filename, params):
    global r, g, b
    if len(params) > 0:
        r = int(params[0][1])
        g = int(params[1][1])
        b = int(params[2][1])
```

```
        print(r, g, b)
    return html%(r, r, g, g, b, b)

r = g = b = 100
oldR = oldG = oldB = 0

if not Wlan.connect("SSID", pw):
    print("Connection to AP failed")
else:
    print("Successfully connected to AP. IP:", Wlan.getMyIPAddress())
    HTTPServer(requestHandler = onRequest)
    while True:
        if r != oldR or g != oldG or b != oldB:
            fillRectangle(2, 2, 4, 4, (r, g, b))
            oldR = r
            oldG = g
            oldB = b
        sleep(1)
```

MERKE DIR...

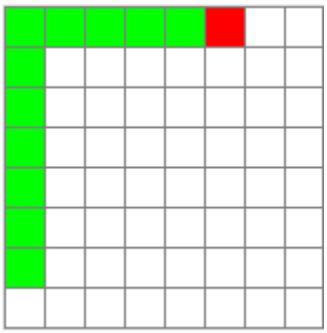
Die Oxocard kann sowohl als Accesspoint und als Webserver funktionieren. Der Webserver wartet auf einen HTTP-Request, der von einem Webbrowser eines Clients (PC, Smartphone, usw.) gesendet wird. Der Server verarbeitet den Request und und sendet einen HTTP-Response an den Client zurück.

Mit dem Modul *tcpcom* wird das Eintreffen eines HTTP-Requests beim Server als **Ereignis** (Event) aufgefasst und automatisch eine Callbackfunktion aufgerufen.

ZUM SELBST LÖSEN

1. Schreibe ein Programm, mit welchem du ferngesteuert mit einem Smartphone alle LEDs auf der Oxocard miteinander auf Gelb oder Grün stellen kannst.

2. Die Schlange bewegt sich auf einem Quadrat. Mit Click auf den Button "grow" wird der Schwanz um 1 Teil grösser, mit Klick auf den Button" shorten" wird sie um 1 Teil kleiner. Verwende die Befehle *growTail()* und *shortenTail()*.



3. Mit einem Schieberegler willst du die Helligkeit eines gefüllten Kreises fernsteuern. Verwende den unten stehenden Vorschlag für den HTML-Code, der einen Submit-Button verwendet. Der aktuelle Helligkeitswert wird im HTTP-Response an den Client zurück übertragen (durch zweimalige Verwendung des


```
html = """<!DOCTYPE html>
<html>
  <head> <title>Oxocard Light Controller</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  </head>
  <body>
     <h2>Welcome To The Light Controller</h2>
     <b>Drag to change intensity and press SUBMIT</b>
     <p>
     <form method="get">
       <input type="range" min="0" max="100" name="bar" value="%d">
       <input type="submit" name="submit" value="SUBMIT">
     </form></p>
     <br>
     Current intensity: %d
  </body>
</html>
"""
```

15. RECHNERKOMMUNIKATION MIT MQTT

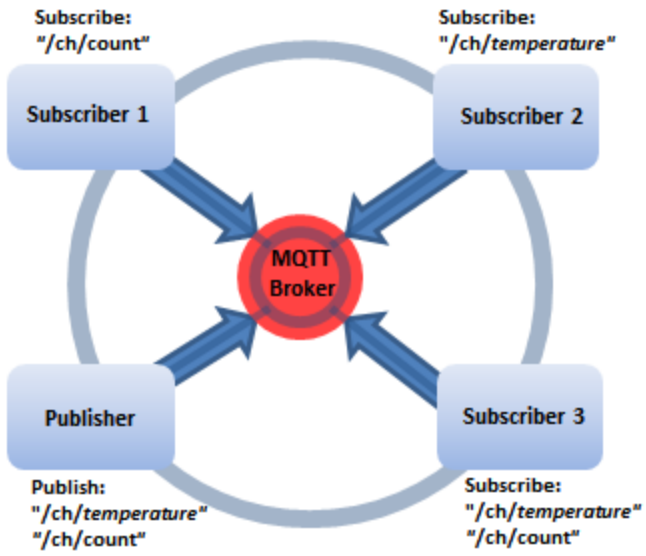
DU LERNST HIER...

wie du über das Internet mehrere Systeme (Oxocard, Raspberry Pi, Arduino, PC) miteinander verbindest, um Kurzinformationen auszutauschen. Das MQTT-Protokoll (Message Queuing Telemetry Transfer) wurde erfunden, um den Datenaustausch schlank zu halten. Es verwendet TCP/IP als Transport-Protokoll. Im Zentrum, als quasi als Nabe eines Rades, befindet sich ein MQTT-Server, der "Broker" genannt wird. Geräte, die untereinander Informationen austauschen möchten sind sozusagen über Speichen mit dem Broker verbunden. Sie werden MQTT-Clients genannt.

MUSTERBEISPIELE

Ein Client (Publisher) kann zu einem bestimmten Thema (Topic) Informationen (Payload) publizierten. Andere Clients (Subscriber) können auf dieses Thema ein Abonnement abschliessen. Der Publisher sendet seine Informationen an den Broker, der sie an alle abonnierten Subscriber weiter leitet. Es gibt keine automatische Rückmeldung an den Publisher, ob und wer die Information tatsächlich gelesen hat.

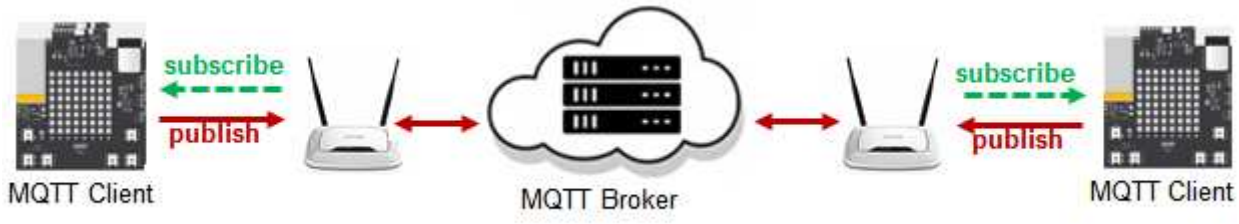
In einem typischen Musterbeispiel ist der Publisher ein Microcontroller, der Messdaten sammelt, beispielsweise die Lufttemperatur oder andere Zustandsparameter eines Systems, z.B. einen Zählerstand- Er publiziert die Informationen alle Minuten und diese können von beliebig vielen Subscriber abholt werden. (Da ein Publisher auch gleichzeitig Subscriber sein kann, können Informationen auch ausgetauscht werden.)



Zuerst schreibst du einen Publisher, der die Anzahl Tastenbetätigungen als Zählerstand publiziert. Du verwendest dazu einen MQTT-Broker kennen, der seine Dienste gratis zur Verfügung stellt. Du kannst wählen zwischen den folgenden Servern:

test.mosquitto.org, m2m.eclipse.org, broker.hivemq.com

Du nimmst die Verbindung zu einem dieser Server über das Internet auf, wobei du einen WLAN-Accesspoint mit einer dir bekannten SSID/Passwort verwendet, beispielsweise über einen Tethering-Dienst auf deinem Smartphone.



Mit Wlan.connect() loggst du dich dort ein. Nachfolgend erstellst du mit client = MQTTClient(broker) ein Client-Objekt und stellst mit client.connect() die Verbindung zum Broker her. Jedes Mal, wenn du die linke untere Taste drückst, publizierst du mit client.publish() den neuen Wert. Auf dem LedGrid zeigst du einige Informationen als Scrolltext dar.

```
from mqttclient import MQTTClient
from oxobutton import *
from oxocard import *

broker = "broker.hivemq.com"
topic = "/ch/count"
btn1 = Button(BUTTON_R2) # button at right bottom
btn2 = Button(BUTTON_L2) # button at left bottom

bigTextScroll("Connect AP.")
Wlan.connect("mySSID", "myPassword")
bigTextScroll("Connect broker.")
client = MQTTClient(broker)
client.connect()
count = 0
bigTextScroll("Press left button.")
while not btn1.wasPressed():
    if btn2.wasPressed():
        count += 1
        if count == 100:
            count = 0
        display(count)
        client.publish(topic, str(count))
client.disconnect()
bigTextScroll("Disconnected.")
```

Um die Informationen abzuholen, schreibst du einen Subscriber, der wiederum über einen Accesspoint auf das Internet kommt (es kann der gleiche sein). Auch hier erzeugst du ein Client-Objekt und registrierst eine Callbackfunktion onMessageReceived(), die vom System automatisch aufgerufen wird, wenn eine Mitteilung zum abonnierten Topic eintrifft. Dazu musst du mit client.subscribe() das Topic abonnieren. Auch hier zeigst du auf dem LedGrid einige Informationen, insbesondere der erhaltene Zählerwert dar.

```
from mqttclient import MQTTClient
from oxobutton import *
from time import sleep
```

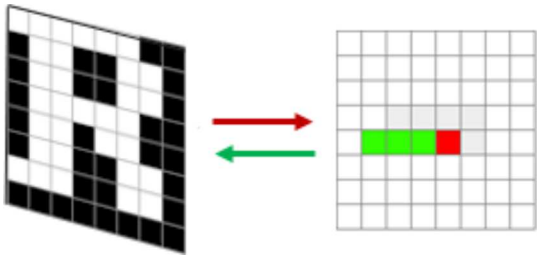
```
from oxocard import *

def onMessageReceived(topic, payload):
    display(payload.decode())

broker = "broker.hivemq.com"
topic = "/ch/count"
button = Button(BUTTON_R2) # button at right bottom

bigTextScroll("Connect AP.")
Wlan.connect("mySSID", "myPassword")
bigTextScroll("Connect broker.")
client = MQTTClient(broker)
client.set_callback(onMessageReceived)
client.connect()
bigTextScroll("Subscribing.")
client.subscribe(topic)
while not button.wasPressed():
    sleep(0.5)
client.disconnect()
bigTextScroll("Disconnected.")
```

Weil eingehende Mitteilungen automatisch die Funktion *onMessageReceived()* aufrufen, kannst du dich im Hauptprogramm darauf beschränken, in einer Schleife zu prüfen, ob du das Programm mit einem Buttonklick abbrechen willst.

MIT CALLBACK-FUNKTIONEN UMGEHEN

Das Programmieren mit Callbacks muss gelernt sein, da es etwas ungewöhnlich ist, dass das laufende Programm irgendwann und an irgendeiner Stelle durch das System unterbrochen wird und nachfolgend die Callbackfunktion zur Ausführung gelangt. Als **Leitplanke** gilt, dass die **Callbackfunktion möglichst kurz** sein soll, d.h. möglichst **rasch zurückkehren** muss. Sie darf also in der Regel keine *sleep()* und keine Wiederholschleifen enthalten. In vielen Fällen genügt es, wenn du **in der Callbackfunktion eine globale Variable neu zuweist**, deren neuer Wert du im Hauptprogramm verwendest. Nach der Rückkehr aus dem Callback fährt das Programm genau dort weiter, wo es unterbrochen wurde.

Im folgenden Beispiel bewegt sich auf einer Oxocard die Schlange endlos noch rechts (mit Rücksprung auf die linke Seite) oder endlos nach links (mit Rücksprung auf die rechte Seite). Sie "hört" auf Befehle "RIGHT", "LEFT" oder "STOP", die von einer anderen Oxocard gesendet werden, je nachdem ob man diese nach rechts oder nach links neigt, bzw horizontal hältst.



Publisher Subscriber

Zuerst betrachtest du das Steuerungsprogramm, das auf einem MQTT-Broker das Topic */ch/roll* publiziert. In der Endlosschleife wird vom Beschleunigungssensor der Roll-Winkel geholt und je nach Bereich die Zustandsvariable *state* auf einen der drei Werte gesetzt. Aus Effizienzgründen willst du aber nur Fall, wo sich der Zustand ändert, den neuen Zustand publizieren und ihren ersten Buchstabe auf dem LedGrid ausschreiben. Der Trick, um dies zu erreichen, ist die Verwendung einer Variablen *oldState*, die den vorhergehenden Zustand abspeichert. Nur wenn *state* und *oldState* verschieden sind, wird der neue Zustand übertragen und *oldState* auf *state* gesetzt.

```
from mqttclient import MQTTClient
from oxocard import *
from oxoaccelerometer import *

acc = Accelerometer.create()

#broker = "m2m.eclipse.org"
#broker = "test.mosquitto.org"
broker = "broker.hivemq.com"
topic = "/ch/roll"

insertBigChar("V", RED)
Wlan.connect("mySSID", "myPassword")
client = MQTTClient(broker)
client.connect()
oldState = ""
state = "STOP"

while True:
    roll = acc.getRoll()
    if roll > -10 and roll < 10:
        state = "STOP"
    elif roll > 20:
        state = "RIGHT"
    elif roll < -20:
        state = "LEFT"
    if state != oldState:
        client.publish(topic, state)
        insertBigChar(state[0])
        oldState = state
    sleep(0.01)
```

Du kannst das Programm testen, indem du den MQTT-Subscriber auf dem PC laufen lässt.
Nun zur Oxocard mit der laufenden Schlange: Du musst zuerst wieder den MQTT-Client erzeugen und ebenfalls das Topic */ch/roll* abonnieren. Am besten ist es, wenn du dir vorstellst, dass das Programm in drei Zuständen sein kann: Im Zustand "RIGHT", wenn sich die Schlange nach rechts, im Zustand "LEFT", wenn sich die Schlage nach links bewegt und im Zustand "STOP", wenn die Schlage still steht.

Der Zustandswechsel wird dir als Message vom Broker mitgeteilt. Dabei wird vom System her die Callbackfunktion *onMessageReceived()* aufgerufen und wir haben es so eingerichtet, dass die *payload* gerade einer der drei Zustandswerte ist. Darum genügt es, im Callback der Variablen *state* den Wert von *payload* zuzuweisen.

```
def onMessageReceived(topic, payload):
    global state
    state = payload
```

(Wir haben uns also extrem gut an die Regel gehalten, dass Callbacks möglichst kurz sein sollen!)
In der Schleife des Hauptprogramms musst du noch mit if-Bedingungen dafür sorgen, dass die Schlange wieder auf die andere Seite hüpft, wenn sie am Ende der Zeile angekommen ist.

```
from oxosnake import *
from mqttclient import MQTTClient
from oxobutton import *

def onMessageReceived(topic, payload):
    # State receiver
    global state
    state = payload

#broker = "m2m.eclipse.org"
#broker = "test.mosquitto.org"
broker = "broker.hivemq.com"
topic = "/ch/roll"
state = "RIGHT"

makeSnake(pos = (4, 4), heading = 90, speed = 70)
Wlan.connect("mySSID", "myPassword")
client = MQTTClient(broker)
client.registerCallback(onMessageReceived)
client.connect()
client.subscribe(topic)
while True:
    # State dispatcher
    if state == "LEFT":
        setHeading(-90)
        forward()
    elif state == "RIGHT":
        setHeading(90)
        forward()
    elif state == "STOP":
        sleep(0.1)

    # Snake round-robin
    if getX() == 11:
        setX(0)
    if getX() == -4:
        setX(7)
```

MERKE DIR...

Mit MQTT kannst du sehr einfach kurze Informationen an mehrere Empfänger versenden. Dazu benötigst du einen MQTT-Broker, der für den Datenaustausch zwischen MQTT-Clients zuständig ist. Diese können Informationen zu einem Topic publizieren oder sich auf ein bestimmte Topics abonnieren. Für den Datenaustausch werden nur wenige Bytes verwendet, er ist daher effizient und kostengünstig.

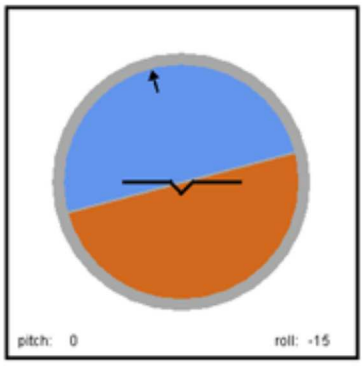
Zu Testzwecken kannst du auch einen Publisher oder Subscriber verwenden, der unter TigerJython auf dem PC läuft. Du kannst die Programme von hier [LINK MACHEN] downloaden.

ZUM SELBST LÖSEN

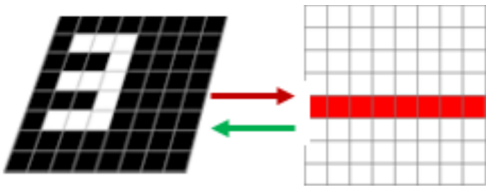
1. Verbessere die beiden oberen Programmbeispiele so, dass es eine Fehleranzeige gibt, falls das Einloggen auf dem Accesspoint oder das Einloggen auf dem Broker nicht gelingt. Anleitung: Beide *connect()*-Methoden geben bei Erfolg True und bei Misserfolg False zurück. Du kannst im Fehlerfall eine vielsagendes Bild anzeigen.

2a. Eine Oxocard erfasst den momentanen Roll-Winkel und publiziert diesen alle 0.1 Sekunden unter irgendeinem Topic, z.B. /ch/roll. Eine zweite Oxocard stellt diesen Wert auf dem Display dar. Um den Datentransfer zu optimieren, kannst du den Roll-Winkel nur dann übermitteln, falls er sich geändert hat.

2b*. Stelle den Roll-Winkel auch auf einem PC dar, entweder nur im Ausgabefenster oder sogar als Grafik.



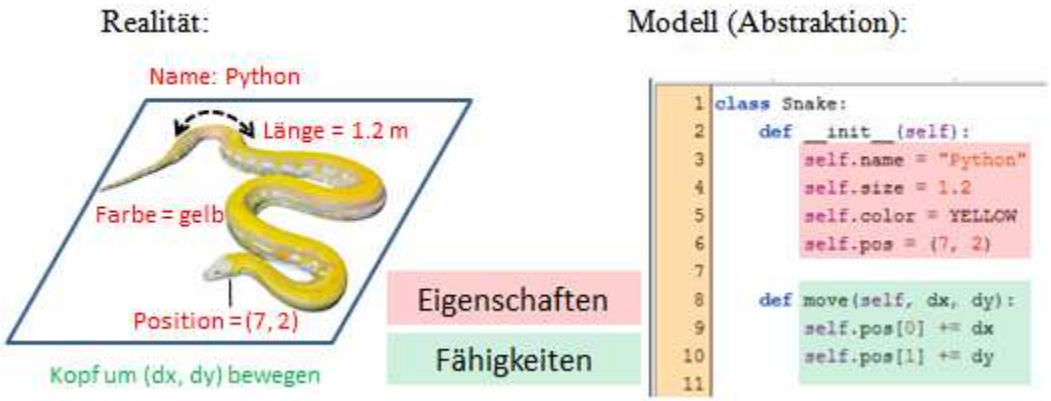
3. Eine Oxocard misst den Pitch-Winkel *pitch* und sendet den Wert *value = int(pitch / 10)*
an eine zweite Oxocard, die den Wert mit einer horizontalen Linie im Bereich y = 0 bis y = 7 darstellt. Die Linie ganz unten soll einem value von 0 und die Linie ganz oben einem value von 7 entsprechen.



16. OBJEKTORIENTIERTE PROGRAMMIERUNG

DU LERNST HIER...

wie du mit dem Konzept der objektorientierten Programmierung (OOP) die Wirklichkeit einfach und anschaulich als Software modellieren kannst. Dabei werden die Eigenschaften und Fähigkeiten von realen Objekten als Variable und Funktionen einer bestimmten Klasse aufgefasst. (Man nennt solche Variablen auch *Instanzvariablen* und die Funktionen auch *Methoden*.)



MUSTERBEISPIELE

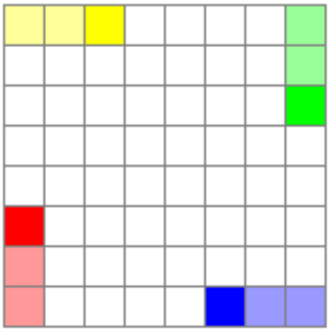
Die Klasse Snake ist bereits im Modul *snake* definiert und steht dir mit

```
from snake import Snake
```

zur Verfügung. Du erzeugst eine einzelne Schlange unter Verwendung der speziellen Methode *Snake()*, die ausnahmsweise grossgeschrieben ist (entspricht dem Klassennamen). Man nennt sie **Konstruktor** der Klasse, weil man bei ihrem Aufruf ein **Objekt** (auch **Instanz** genannt) erzeugt (konstruiert). Der Konstruktor hat genau die gleichen optionalen Parameter wie *makeSnake()*, du kannst der Schlage also damit ihre "Anfangseigenschaften" geben, beispielsweise

```
roxy = Snake()
kitty = Snake(heading = 90, pos = (1, 6))
roby = Snake(headColor = (255, 255, 0), tailColor = (30, 30, 0))
```

In deinem Programm erzeugst du 4 Schlangenobjekte und bewegst sie auf Quadraten. Um in der Funktion *step()* die Methoden eines bestimmten Objekts (einer Instanz) aufzurufen, verwendest du den **Punktoperator**. Um die Animation zu beschleunigen, schaltest du das automatische Rendering mit *enableRepaint(False)* ab und renderst erst, nachdem alle Schlagen eine Quadratseite abgelaufen sind.



```
# Snake16.py

from oxosnake import *

pythy1 = Snake(headColor = (255, 0, 0), tailColor = (120, 0, 0),
               pos = (0, 5), size = 3, speed = 90)
pythy2 = Snake(headColor = (255, 255, 0), tailColor = (120, 120, 0),
               pos = (2, 1), heading = 90, size = 3, speed = 90)
pythy3 = Snake(headColor = (0, 255, 0), tailColor = (0, 120, 0),
               pos = (6, 3), heading = 180, size = 3, speed = 90)
pythy4 = Snake(headColor = (0, 0, 255), tailColor = (0, 0, 120),
               pos = (4, 7), heading = 270, size = 3, speed = 90)

def step(snake):
    snake.forward(2)
    snake.right(45)

enableRepaint(False)
while True:
    step(pythy1)
    step(pythy2)
    step(pythy3)
    step(pythy4)
    repaint()
```

Schalte durch Entfernen (Auskommentieren) der Zeile *enableRepaint(False)* das automatische Rendering wieder ein und gebe eine Erklärung für das Programmverhalten.

Klassenableitung

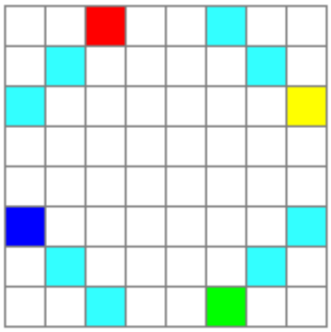
Durch die Definition von Klassen werden Eigenschaften und Fähigkeiten in einer Programmstruktur zusammengefasst. Man spricht dabei von einer **Kapselung** (Encapsulation). In der OOP kannst du aber auch durch **Klassenableitung** (Inheritance) bestehende Klassen **erweitern** und **verändern**. Damit ist es möglich, Eigenschaften und Fähigkeiten **hinzuzubauen**, ohne dass man den Code der bestehenden Klasse verändern muss.

Im vorhergehenden Programm sind ja fast alle Schlagen gleichartig und haben alle eine zusätzliche Funktion *step()*. Du definierst daher deine eigene Klasse *PySnake*, welche einen einfacheren Konstruktor und eine zusätzliche Methode *step()* enthält.

Die Klassendefinition wird mit dem neuen Schlüsselwort *class* eingeleitet, gefolgt vom Klassennamen *PySnake*. In einer runden Klammer gibst du die Klasse *Snake* an, von der die neue Klasse **abgeleitet** ist. *PySnake* "erbt" damit alle Eigenschaften und Fähigkeiten der Klasse *Snake*, was heisst, dass einem Objekt von *PySnake* automatisch auch alle Eigenschaften und Fähigkeiten von *Snake* zur Verfügung stehen. Man sagt anschaulich, dass eine *PySnake* auch eine *Snake* ist (in Analogie zum Tierreich eine Untergattung).

Der Konstruktor der Klasse *PySnake* muss in Python mit *__init__()* definiert werden. Eigentlich wäre die Bezeichnung *PySnake()* naheliegender, da ja der Konstruktor aufgerufen wird, wenn man mit *pythy = PySnake()* ein Objekt erzeugt. Sowohl der Konstruktor und alle Methoden der Klasse müssen einen zusätzlichen Parameter *self* haben, den man meist aber gar nicht braucht.

Im Konstruktor von *PySnake()* initialisierst du die Klasse *Snake*, aus der *PySnake* abgeleitet ist. (Man nennt *Snake* auch die **Basisklasse** oder Superklasse von *PySnake*.) Merke dir einfach, dass du zur Initialisierung *Snake.__init__(self, ...)* schreiben musst. Schliesslich folgt die Definition der Methode *step()* mit dem zusätzlichen Parameter *self*.



Im Hauptprogramm erzeugst du vier Objekte *pythy1..pythy4* und und rufst in einer Endlosschleife ihre Methode *step()* auf.

```
# Snake16a.py

from oxosnake import *

class PySnake(Snake):
    def __init__(self, color, position, direction):
        Snake.__init__(self, headColor = color,
                       tailColor = reduceBrightness(color, 2),
                       pos = position, size = 3,
                       heading = direction,
                       speed = 90)

    def step(self):
        self.forward(2)
        self.right(45)

pythy1 = PySnake(RED, (0, 5), 0)
pythy2 = PySnake(YELLOW, (2, 1), 90)
pythy3 = PySnake(GREEN, (6, 3), 180)
pythy4 = PySnake(BLUE, (4, 7), 270)

enableRepaint(False)
while True:
    pythy1.step()
    pythy2.step()
    pythy3.step()
    pythy4.step()
    repaint()
```

Das Programm ist durch die Definition einer eigenen Klasse nicht nur eleganter geworden, sondern gibt dir die Möglichkeit, das neue Objekt *PySnake* wie ein durch das System vordefiniertes Objekt zu verwenden. Dazu fügst du die Klassendefinition in eine neue Datei mit dem Namen *pysnake.py* ein und lädst diese mit der Option *Modul herunterladen* auf die Oxocard.

```
from oxosnake import *

class PySnake(Snake):
    def __init__(self, color, position, direction):
        Snake.__init__(self, headColor = color,
                       tailColor = reduceBrightness(color, 10),
                       pos = position, size = 3,
                       heading = direction,
                       speed = 90)
```

```
    def step(self):
        self.forward(2)
        self.right(45)
```

In irgendeinem Programm kannst du nun die neue *PySnake* verwenden, indem du das Modul *pysnake* importierst.

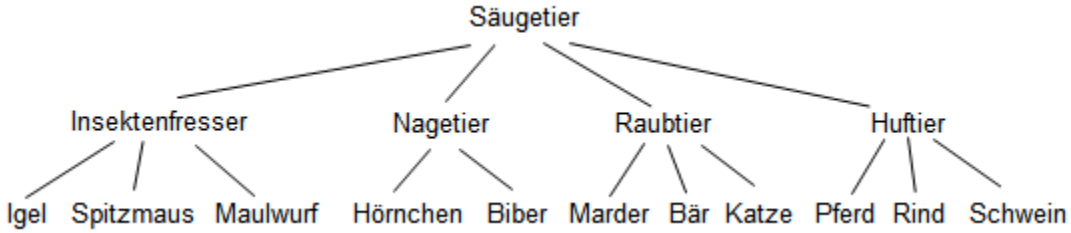
```
# Snake16b.py

from pysnake import *

pythy1 = PySnake(RED, (0, 5), 0)
pythy2 = PySnake(YELLOW, (2, 1), 90)
pythy3 = PySnake(GREEN, (6, 3), 180)
pythy4 = PySnake(BLUE, (4, 7), 270)

enableRepaint(False)
while True:
    pythy1.step()
    pythy2.step()
    pythy3.step()
    pythy4.step()
    repaint()
```

Du hast jetzt dein erstes Bibliotheksmodul geschrieben und dabei eine weitere wichtige Strukturierungsmöglichkeit von Programmen kennen gelernt.

MERKE DIR...

Die OOP stellt dir ein Programmierkonzept zur Verfügung, mit dem du die Realität anschaulich und übersichtlich modellieren kannst. Dabei werden Eigenschaften und Fähigkeiten als Instanzvariable und Methoden in der Klassendefinition abgebildet. Man erzeugt ein Objekt einer Klasse durch Aufruf ihres Konstruktors und führt seine Methoden mit dem Punktoperator aus.

Klassen können verändert und ergänzt werden, indem man sie ableitet. Es entsteht dadurch eine ähnliche Abhängigkeit wie beispielsweise bei der Klassifikation im Tierreich.



ZUM SELBST LÖSEN

1a. Bei jedem Klick auf einen von dir gewählten Button soll eine neue Schlage mit speed = 100 und der Länge 1 (nur der Kopf) mit zufälliger Farbe an zufälliger horizontaler Position auf der untersten Zeile entstehen. Du fügst sie in eine Liste *family*, die zuerst leer ist. In der Endlosschleife bewegst du alle Familienmitglieder nach oben, bis sie auf y = -1 sind und setzt sie dann wieder auf die unterste Zeile. Eine zufällige Farbe aus den 7 Grundfarben erhältst du mit *getRandomColor()*. Verwende *enableRepaint(False)*, um die Animation zu beschleunigen.

1b. Mache dasselbe mit Schlangen der Länge 2, welche ein Schwanzfarbe haben, die der um den Faktor 20 abgedunkelten Kopffarbe entspricht. Verwende dazu die Funktion *reduce(color, 20)*, welche die abgedunkelte Farbe von *color* zurückgibt.

2. Leite aus *Snake* eine Klasse *DancingSnake* ab, deren Konstruktor wie oben bei *PySnake* aufgebaut ist, aber nur die Anfangsposition der Schlage als Parameter verwendet. Die Schlangen haben alle *speed = 100*. Mit *color = getRandomColor()* erhält der Kopf jeder neuen Schlage eine Zufallsfarbe und mit *reduce(color, 20)* der zugehörige Schwanz die gleiche, aber dunklere Farbe. Die DancingSnake hat zudem eine Methode *dance()*, mit der sie auf einem Achteck mit 1 Schritt Seitenlänge eine volle Runde dreht.

   Erstelle 3 (oder mehr) Instanzen der *DancingSnake* an verschiedenen (eventuell zufälligen) Positionen und lasse sie endlos tanzen.

17. INTERNET-RESSOURCEN

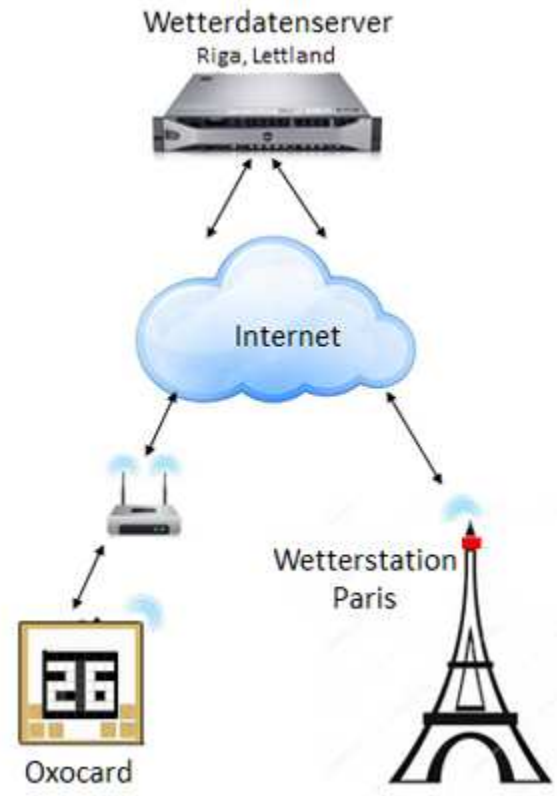
DU LERNST HIER...

wie du Daten und Informationen vom Internet beziehen und direkt weiter verarbeiten kannst. Dazu muss sich deine Oxocard mit WiFi über einen Hotspot mit dem Internet verbinden. Genau gleich wie du mit einem Webbrowser auf deinem Smartphone oder PC eine Verbindung zu irgendeinem Webserver erstellst, verbindet sich dann deine Oxocard als Webclient mit dem Webserver und macht eine Browserabfrage (einen HTTP-Request).

MUSTERBEISPIELE

Das Modul *weather* kapselt den Kode für eine Browserabfrage von Wetterdaten auf dem Wetterdatenserver **http://openweathermap.org**. Gehe mit einem Browser auf diesen Weblink, um dich genauer zu informieren. Der Wetterdatenserver sammelt die Wetterdaten in über 200'000 Städten der ganzen Welt mit einer Aktualisierungszeit von unter 2 Stunden.

Für eine Wetterdaten-Anfrage mit der Oxocard benötigst du lediglich eine einzige Funktion *weather.request(ssid, password, city, key),* mit der die Oxocard eine Verbindung zum Host erstellt und die Daten für die gegebene Stadt anfordert. Die Funktion gibt die erhaltenen Informationen in einem Dictionary zurück. Damit steht dir augenblicklich ein Satz von aktuellen Wetterinformationen zur Verfügung, aus denen du eine beliebige Auswahl treffen kannst.



Key	Value
"status"	"OK" oder Fehlerstring, der den Fehler beschreibt, z.B. "City not found"
"temp"	Temperatur in °C (float)
"pressure"	Luftdruck in hPa (mbar) (float)
"humidity"	Luftfeuchtigkeit in % (int)
"temp_min"	Tagesminimum der Temperatur in °C (float)
"temp_max"	Tagesmaximum der Temperatur in °C (float)

"description"	Wetterlage in Worten (deutsch) (string)
"sunrise"	Sonnenaufgang in Universal Time (UTC) (string)
"sunset"	Sonnenuntergang in Universal Time (UTC) (string)
"datetime"	Datum - Uhrzeit (UTC) der Wettererfassung (string)

Im folgenden Programm holst du rund alle 30 Sekunden die Temperatur und die Wetterbeschreibung von Paris und schreibst die Daten auf dem Pixeldisplay der Oxocard aus. (Die Zugangsdaten *myssid/mypassword* musst du an deinen Hotspot anpassen.)

```
import weather
from oxocard import *
from time import sleep

city = "Paris, FR"
bigTextScroll(city, GREEN)
while True:
    info = weather.request("myssid", "mypassword", city)
    if info["status"] == "OK":
        bigTextScroll(info["description"])
        sleep(2)
        display(round(info["temp"]))
        sleep(30)
        display("::")
        sleep(2)
    else:
        bigTextScroll(info["status"], RED)
        break
```

MERKE DIR...

Die Beschaffung von Informationen und Ressourcen aus dem Internet ist im digitalen Zeitalter von grosser Wichtigkeit. Ein Mikrosystem kann diese Daten autonom und ohne Eingriff eines Menschen abholen und weiter verarbeiten.

Falls du in *weather.request()* den letzten Parameter *key* weglässt, wird ein interner Authorisierungsschlüssel verwendet, der bis zu 60 Abfragen/min ermöglicht. Um unabhängig von anderen Usern zu sein, kannst du auf dem Wetterdatenserver *http://openweathermap.org* einen eigenen kostenlosen Key beantragen und diesen als Parameter verwenden.

ZUM SELBST LÖSEN

1a. Aus einer Liste von 10 verschiedenen Städten wird der Reihe nach auf dem Pixeldisplay der Name der Stadt und die aktuelle Temperatur angezeigt. Die Anzeige soll endlos laufen.

1b. Mit den Buttons BUTTON_R2 und BUTTON_R3 kann man die Liste vor und rückwärts durchlaufen und es wird immer der Städtename und die Temperatur angezeigt.

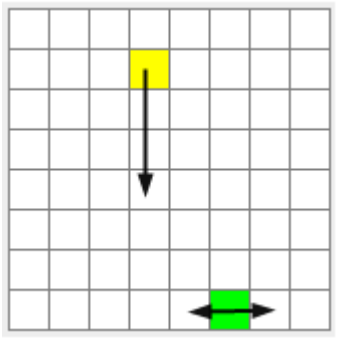
2. Aus einer Liste von 10 Schweizer Städten wird diejenige Stadt mit der aktuell kleinsten und grössten Temperatur auf dem Pixeldisplay (zusammen mit ihrer Temperatur) angezeigt.

ARBEITSBLATT 1: "FANG DAS EI" MIT OXOCARD

SPIELBESCHREIBUNG

Auf dem 8x8 LED-Display bewegt sich ein gelbes Pixel auf einer zufälligen Spalte von oben nach unten, analog einem Ei, das von einem Tisch hinunter fällt.

Auf der untersten Zeile kannst du durch Drücken des linken oder rechten Buttons einen Pixel hin und her bewegen, analog einem Korb, mit dem du das Ei auffangen möchtest, bevor es am Boden zerschlägt. Bist du mit dem Korbpixel auf der untersten Zeile dort, wo das Eipixel ankommt, so hast du es "gefangen" und kriegst einen Punkt gutgeschrieben. Das Spiel wird 10 mal wiederholt und es geht darum, dass du möglichst viele Eier bzw. Punkte sammelst.



Nach dem Prinzip der Modularisierung teilst du Aufgabe in mehrere Teilaufgaben ein, die du schrittweise löst.

TEILAUFGABE 1: AUF FIXER SPALTE FALLENDES EIPIXEL

Als erstes schreibst du ein Programm, dass bei jedem Aufruf der Funktion *moveEgg()* das Eipixel auf der fixen Spalte mit x = 3 Zeile um Zeile nach unten bewegt. Du musst dabei bedenken, dass es sich um eine Animation handelt, die du nach einem bekannten Prinzip anpackst: In jedem Animationsschritt löschst du alles vorangehende, also hier nur das Ei , berechnest die nächste Position und zeichnest das Ei neu. (Die Koordinaten des Eies sind *xE, yE*.) Wenn das Ei unten angekommen ist, so setzt du es wieder ganz nach oben. Du gehst von folgendem Programmgerüst aus, und fügt die fehlenden Anweisungen ein:

```
from oxogrid import *


period = 0.2 # determines rate of fall

def moveEgg():
    global yE
    if
        ...
    else:
        ...
    dot(xE, yE, YELLOW)

xE = 3
yE = -1

# Animation loop
while True:
    clear()
    moveEgg()
    sleep(period)
```

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

Beachte, dass du die Variable *yE* in der Funktion *moveEgg()* als global bezeichnen musst, da sie in der Funktion verändert wird. Da *moveEgg()* das Ei als erstes um 1 Zeile nach unten setzt, muss du zu Beginn *yE* auf -1 setzen.

TEILAUFGABE 2: AUF ZUFÄLLIGER SPALTE FALLENDES EIPIXEL

Als nächstes erweiterst du das Programm so, dass bei jedem neuen Ei die Falllinie zufällig gewählt wird. Verwende den Aufruf *random(a, b)* aus dem Modul *random*, der eine Zufallszahl zwischen a und b (a und b eingeschlossen) liefert. Am besten testest du in der Animation-Loop, ob das Ei unten angekommen ist, und wählst dann für das nächste Ei eine zufällige x-Koordinate:

```
while True:
    clear()
    moveEgg()
    if yE == 7: # egg at bottom
        ...
    sleep(period)
```

TEILAUFGABE 3: VERSCHIEBBARES KORBPIXEL

Schreibe im gleichen Stil ein anderes Programm, mit welchem du ein grünes Korbpixel auf der untersten Zeile hin und her schieben kannst, wenn du den linken oder rechten Button klickst. Erstelle dazu zwei Buttonobjekte *btn1* und *btn2*, für den linken und rechten Button:

```
btn1 = Button(BUTTON_L2)
btn2 = Button(BUTTON_R2)
```

Dazu musst du das Modul button importieren:

```
from button import *
```

In der Funktion *moveBasket()* testest du mit

```
btn1.wasPressed() bzw. btn2.wasPressed()
```

ob einer der Buttons gedrückt wurde und änderst die Korbposition *xB* entsprechend. Gehe von folgendem Programmgerüst aus.

```
from oxogrid import *
from oxobutton import *

period = 0.2 # determines rate of fall

def moveBasket():
    global xB
    if btn1.wasPressed():
        ...
    if btn2.wasPressed():
        ...
```

```
    dot(xB, 7, GREEN)

btn1 = Button(BUTTON_R2)
btn2 = Button(BUTTON_L2)
xB = 3

while True:
    clear()
    moveBasket()
    sleep(period)
```

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

Verbessere dein Programm noch so, dass man den Korb nicht über den linken oder rechten Rand hinausbewegen kann.

TEILAUFGABE 4: FLACKERN VERMEIDEN

Es ist dir sicher aufgefallen, dass das Korbpixel flackert, was sehr störend ist. Das ist ja auch verständlich, da du immer wieder den Display mit *clear()* löschst und dieser leere Display auch sichtbar ist. Es gibt einen klassischen Trick, wie du das Flackern bei Animationen verhindern kannst: Du sagst dem System mit *enableRepaint(False)*, dass es nicht jede Grafikoperation neu darstellen soll. Du fasst dann das Löschen und neu Zeichnen als eine einzige Aktion zusammen und sagst nachher mit *repaint()*, dass diese angezeigt wird.

Ergänze deinen Code mit diesem Trick, damit das Flackern verschwindet.

TEILAUFGABE 5: ZUSAMMENFÜGEN VON EI UND KORB

Füge nun die Codeteile so zusammen, so dass sich sowohl das Eipixel wie der Korb (allerdings noch unabhängig voneinander) bewegen. Das Hauptprogramm enthält die Schleife:

```
while True:
    clear()
    moveEgg()
    moveBasket()
    repaint()
    if yE == 7: # egg at bottom
        xE = randint(0, 7)
    sleep(period)
```

TEILAUFGABE 6: KOLLISIONEN ENTDECKEN

Das Programm soll nun selbst herausfinden, ob das Eipixel und das Korbpixel auf der untersten Zeile zusammentreffen (kollidieren). In diesem Fall erhöhst du eine Variable *hits* um 1, welche die Anzahl gewonnener Punkte speichert und zeigst mit *display()* oder *print()* die Zahl der Spielpunkte an. Die Funktion für den Kollisionstest lautet:

```
def checkHit():
    global hits
    if xE == xB:  # yes!
        hits += 1
```

```
            display(hits, YELLOW)
            #print("Nb Hits :", hits)
        else:
            display('-', RED)
        repaint()
        sleep(1)
```

Du brauchst also nur die Animations-Loop zu ergänzen

TEILAUFGABE 7: ZAHL DER EIER BEGRENZEN

Es ist naheliegend, dass du die Anzahl der verfügbaren Eier und damit die Fangversuche begrenzt. Führe dazu eine Variable *nbEggs* ein, die zuerst auf einen bestimmten Wert, z.B. 10 eingestellt ist. Bei jedem Hit verringerst du sie um 1 und stoppst das Programm, wenn du keine Eier mehr hast.

Deine Animations-Loop heisst nun:

```
while nbEggs > 0:
    clear()
    ...
display(hits, GREEN)
repaint()
```

Bei Programmende schreibst du noch die Anzahl Hits, d.h. den Score des Spiels in grüner Farbe aus.

SPIEL FERTIGSTELLEN UND PERSONALISIEREN

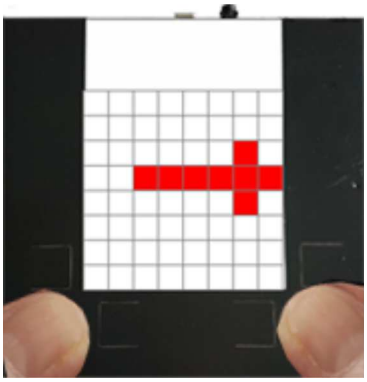
Es ist nun deiner Fantasie überlassen, dem Spiel durch eigene Varianten und Verbesserungen einen persönlichen Touch zu geben. Viel Spass!

ARBEITSBLATT 2: "MEMORY" MIT OXOCARD

SPIELBESCHREIBUNG

Das Ziel des Spiels ist es, sich an eine möglichst lange Folge von Links-Rechts-Pfeilen zu erinnern. Du kannst damit dein Gedächtnis trainieren!

Das Programm zeigt bei eine zufällige Folge von Links- bzw. Rechtspfeilen an. Danach musst du diese Folge durch Drücken des linken bzw. rechten Buttons wiedergeben. Ist die Wiedergabe richtig, wird der Vorgang mit einer um ein Element längeren Folge wiederholt, sonst ist das Spiel fertig.



TEILAUFGABE 1: ZUFÄLLIGE FOLGE MIT ZAHLEN 0 UND 1 ERZEUGEN

Als erstes schreibst du ein Programm, das beim Aufruf der Funktion *createSeq(n)* eine zufällige Folge mit n Elementen der Zahlen 0 oder 1 erzeugt und diese in einer Liste *seq* speichert. Teste das Programm mit verschiedenen n und schreibe die Liste mit *print(seq)* aus.

```
from oxocard import *
from random import randint

def createSeq(n):
    for i in range(n):
    x = randint(0, 1)
    seq.append(x)

seq = []
n = 3
createSeq(n)
print(seq)
```

TEILAUFGABE 2: ZUFÄLLIGE FOLGE VON ← UND → ERZEUGEN

Definiere eine Funktionen *right(),* die während 0.2 s einen Rechtspfeil angezeigt und danach wieder löscht. Auch nach dem Löschen soll das Programm 0.2 s warten. Definiere eine zweite Funktion *left()*, die auf die gleiche Art einen Linkspfeil anzeigt.

Ergänze die Funktion *createSeq()* so, dass bei 0 ein Rechtspfeil und bei 1 ein Linksspfeil angezeigt wird. Teste das Programm mit verschiedenen n.

TEILAUFGABE 3: WIEDERGABE DER FOLGE MIT BUTTONS

Im nächsten Entwicklungsschritt baust du die Interaktion mit dem Spieler ein. Beim Drücken des linken bzw. rechten Buttons wird in die Liste *seq2* eine 0 oder 1 eingetragen, je nachdem, ob der Spieler den linken oder rechten Button drückt.

```
n = 3
seq2 = []

while len(seq2) < n:
    if button_a.was_pressed():
        seq2.append(0)
    if button_b.was_pressed():
        seq2.append(1)
    sleep(0.1)
```

Das *sleep(0.1)* verhindert, dass das Programm in der Wiederholschleife nicht unnötig viel Leistung vergeudet (und damit schwer abzubrechen ist). Ergänze das Programm mit den Funktionen *right()* und *left()*, so dass beim Drücken der Buttons die passenden Pfeile angezeigt werden.

TEILAUFGABE 4: BEIDE FOLGEN VERGLEICHEN

Nach der Benutzereingabe vergleichst du die beiden Listen *seq* und *seq2.* Sind sie gleich, so ist die Wiedergabe richtig. Als Rückmeldung kannst du zum Beispiel alle Leds grün schalten bzw. das Zeichen '--' einblenden.

```
if seq == seq2:
    clear(GREEN)
    print("Ok")
else:
    display('-', RED)
    ...
```

TEILAUFGABE 5: SCHWIERIGKEITSGRAD STEIGERN

Bisher hat die gezeigte Sequenz immer noch die fixe Länge 3. Nun erweiterst du das Programm, dass das Spiel mit n = 2 beginnt und bei einer richtigen Wiedergabe jeweils n um 1 erhöht. Für die Wiederholung verwendest du eine *while True*-Schleife.

```
n = 2

while True:
    seq = []
    seq2 = []
    createSeq(n)
    while len(seq2) < n:
        .....
        if seq == seq2:
           ....
           n += 1
        .....
```

TEILAUFGABE 6: SPIEL BEENDEN

Um das Spielende zu erfassen, musst du aus der Endlosschleife eine while-Schleife machen, die so lange läuft, bis die boolesche Variable *gameOver True* ist.

```
....
n = 2
gameOver = False

while not gameOver:
    seq = []
    seq2 = []
    createSeq(n)

    while len(seq2) < n:
        if button_a.was_pressed():
            .....

    if seq == seq2:

        ....
        n += 1
    else:
        ....
        gameOver = True
```

Am Schluss kannst du als Game Score noch die Länge der letzten korrekt wiedergegebenen Folge anzeigen, beispielsweise mit einem scrollenden Text.

Ein neues Spiel kannst du jederzeit durch Drücken der Reset-Tasten starten.

TEILAUFGABE 7: DAS SPIEL NACH EIGENEN IDEEN ERWEITERN

Du hast sicher viele Ideen, wie du das Spiel individuell gestalten oder verbessern kannst.

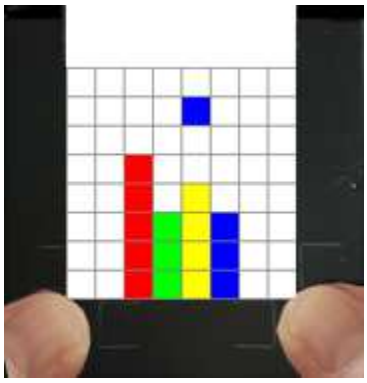
ERGÄNZUNG: TÖNE MEMORISIEREN

Du kannst das Memory-Spiel auch mit Tonfolgen programmieren. Schau im Kapitel Sound nach, wie du Töne erzeugen und hörbar machen kannst.

ARBEITSBLATT 3: TETRIS MIT OXOCARD

SPIELBESCHREIBUNG

Das Spiel ist dem berühmten Tetris nachempfunden, aber stark vereinfacht. Du hältst das Board so in der Hand, dass acht Vierreihen vor dir liegen.
Zu Beginn des Spiels erscheinen in der untersten Reihe 4 Farben, die festlegen, mit welchen Farben d. die Spalten zu füllen sind.



Auf der obersten Reihe wird nun ein Stein (Pixel) mit zufälliger Farbe sichtbar, der sich nach unten bewegt. Du kannst ihn mit den Buttons nach links- bzw. nach rechts verschieben (vorausgesetzt, es hat dort noch keine Steine), mit dem Ziel, ihn in der gleichfarbigen Spalte zu platzieren. Fällt ein Stein auf eine falsche Farbe, so wird er selbst und auch der darunter liegende Stein gelöscht (ausser ganz unten).
Insgesamt werden 24 Steine fallen gelassen und die Herausforderung besteht darin, möglichst viele Steine in den richtigen Spalten zu platzieren.

TEILAUFGABE 1: FALLENDE STEINE IN ZUFÄLLIGEN FARBEN

Das folgende Programm zeichnet fallende rote Steine, die in einer zufällige gewählten Spalte x = 2 bis x = 5 ganz oben starten.

```
from oxocard import *
from random import randint

while True:
    x = randint(2, 5) #random start position
    y = 0

    while y < 8:
        dot(x, y, RED)
        sleep(0.2)
        dot(x, y, BLACK)
        y += 1
```

Erweitere das Programm so, dass die neuen Steine eine zufällige Farbe aus den Farben RED, GREEN, YELLOW und BLUE haben. Am besten speicherst du dazu die Farben in einem Tupel colors:

colors = (RED, GREEN, YELLOW, BLUE)

TEILAUFGABE 2: MIT BUTTONS SPALTE WECHSELN

Mit Klick auf den unteren linken Button soll sich der fallende Stein um eine Spalte nach links, beim Klick den unteren rechten Button um eine Spalte nach rechts verschieben. Beachte,

dabei dass sich die Steine nur in den Spalten x = 2 bis x = 6 verschieben dürfen.

Um einen Buttonklick zu verarbeiten, fügst du einen Test

button.was_pressed()

in die Wiederholschleife ein.

TEILAUFGABE 3: FARBEN IN DEN SPALTEN FESTLEGEN

Schreibe eine Funktion *init()*, welche in der Zeile 7 die vier Spaltenfarben festlegt. Wähle die Farben in der Reihenfolge der Liste *colors*.

Sorge auch dafür, dass die Pixels in der Zeile y = 7 nicht durch die fallenden Steine überdeckt werden., indem du sie nur bis in Zeile y = 6 fallen lässt.

TEILAUFGABE 4: BELEGTE ZELLEN ERKENNEN

Jetzt möchtest du die korrekt platzierte Steine stehen lassen. Dazu musst du vorausblickend kontrollieren, ob du den Stein auf die nächste Zeile herunterfallen lässt. Es gibt verschiedene Fälle: Ist der Platz frei, so gibt es kein Problem und du kannst das Pixel herunterfallen lassen. Ist aber dort bereits ein Stein und hat dieser die gleiche Farbe, so beendest du das Herunterfallen, sonst musst du gemäss der Spielvorgabe den darunter liegenden Stein und den Stein selbst löschen.

Du kannst die Farbe c eines Pixels an der Stelle x. y zurückholen, indem du
c = getColor(x, y) aufrufst.

TEILAUFGABE 5: LINKS-/RECHTSVERSCHIEBUNG EINSCHRÄNKEN

Du kannst natürlich einen Stein nicht nach links oder rechts bewegen, wenn dort bereits eine Steinsäule vorhanden ist. Verhindere also die Verschiebung des fallenden Steins, wenn sich links oder rechts davon bereits ein Stein befindet.

TEILAUFGABE 6: SPIELENDE / ERGEBNIS ANZEIGEN

Nun bist du fast fertig: Begrenze noch die Anzahl der Spielsteine auf 24 und beende das Spiel, nachdem alle gespielt sind Die erreichte Punktzahl (der Score) ist gleich der Totalzahl der gesetzten Steine. Schreibe den Score auf dem Display aus..

TEILAUFGABE 7: EIGENE IDEEN EINBAUEN

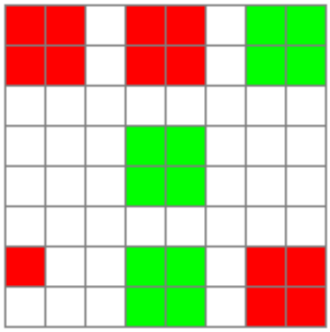
Ergänze oder modifiziere das Spiel nach deinen eigenen Ideen, um ihm einen persönlichen Touch zu geben.

ARBEITSBLATT 4: GAMEN MIT TIC-TAC-TOE

SPIELBESCHREIBUNG

Zwei Spieler können nacheinander in einem 3x3 Gitter in leere Zellen ihre Marke setzen, z.B. Kringel und Kreuze oder rote und grüne Steine. Wer zuerst 3 Marken auf einer Zeile, Spalte oder Diagonale setzten kann, hat gewonnen.

Auf der Oxocard sind die Steine als rote oder grüne Quadrate sichtbar. Mit 4 Cursortasten kann der Spieler, der am Zug ist, einen hell-farbigen Cursor auf eine Zelle verschieben und mit der OK-Taste dort seinen Stein setzen.



TEILAUFGABE 1: DEN SPIELZUSTAND BESCHREIBEN

Du kannst das Display wie ein Spielbrett (Game Board) auffassen. In jedem Moment des Spiels befindet sich dieses in einem bestimmten **Zustand**. Es ist sehr wichtig, dass du den Zustand in einer angepassten Datenstruktur beschreibst. Du verwendest hier eine Liste *state* mit 9 Elementen, wobei der Index zeilenweise den Zellen entspricht. Du legst fest, dass der Wert -1 eine leere Zelle, eine 0 eine Marke des Spielers 0 und eine 1 eine Marke des Spielers 1 angibt. Beispielsweise wird der oben gezeichnete Spielzustand beschrieben durch:

state = [0, 0, 1, -1, 1, -1, -1, 1, -1]

Der Cursor befindet sich in der Zelle mit dem Index 6. Diese Information gehört ebenfalls zum Spielzustand, sowie auch, dass gerade Spieler 0 (mit den roten Steinen) am Zug ist, da der Cursor hellrot erscheint.

Für die Bewegung des Cursors eignet sich der Zellenindex schlecht. Besser verwendest du für den Cursorzustand eine Zellenangabe *location* mit zwei Werten im Bereich 0 bis 3. (Diese Werte entsprechen noch nicht direkt seiner Lage auf dem Display.) Es ist wichtig, dass du gleich zwei Transformationsfunktionen *toLocation(index)* und *toIndex(location)* schreibst, damit du von der lästigen Umrechnerei zwischen dem Index und der Location ein für alle Mal befreit bist. (Man pflegt solche Funktionen mit *to..* einzuleiten.)

Zuerst willst du ein paar fest gewählte Spielzustände darstellen. Du brauchst in folgender Vorlage nur noch in der Funktion *drawBoard()* ein paar Zeilen einzusetzen.

```
from oxocard import *

def toLocation(index):
    locX = index % 3
    locY = index // 3
    return (locX, locY)

def toIndex(location):
    return 3 * location[1] + location[0]

def drawBoard():
```

```
    for index in range(9):
        x, y = toPoint(index)
        # todo

enableRepaint(False)
state = [0, 0, 0, 0, 0, 0, 0, 0, 0]
drawBoard()
repaint()
sleep(3)
state = [1, 1, 1, 1, 1, 1, 1, 1, 1]
drawBoard()
repaint()
sleep(3)
state = [1, -1, 0, 0, 1, -1, 0, -1, 1]
drawBoard()
repaint()
```

Damit immer gleich das ganze Spielbrett auf einmal neu erscheint, verwendest du *enableRepaint(False).*

TEILAUFGABE 2: CURSORBEWEGUNG EINBAUEN

Als nächstens behandelst du die tastengesteueten Cursorbewegungen. Die aktuelle Lage des Cursors speicherst du wie eine Location in einer Liste, die zuerst auf [0, 0] initialisiert ist. In einer Endlosschleife des Hauptprogramms überprüfst du, ob einer der Buttons gedrückt wurde und veränderst die Cursorlocation entsprechend. Für die Darstellung auf dem Display schreibst du am besten eine Funktion *drawCursor()*, die im folgenden Programmskelett bereits ausprogrammiert ist.

Baue jetzt das Skelett in dein Programm ein und ergänze die fehlenden Blöcke. Verhindere auch, dass man den Cursor über den Rand des Displays hinausschieben kann.

```
def drawCursor():
    if player == 0:
        color = (255, 0, 0)
    else:
        color = (0, 255, 0)
    dot(3 * cursor[0], 3 * cursor[1], color)

btnUp = Button(BUTTON_L2)
btnDown = Button(BUTTON_L3)
btnLeft = Button(BUTTON_R3)
btnRight = Button(BUTTON_R2)

enableRepaint(False)
state = [1, -1, 0, 0, 1, -1, 0, -1, 1]
cursor = [0, 0]  # x, y
player = 0

while True:
    if btnUp.wasPressed():
        ...
    elif btnDown.wasPressed():
        ...
    elif btnLeft.wasPressed():
```

```
        ...
    elif btnRight.wasPressed():
        ...
    drawBoard()
    drawCursor()
    repaint()
    sleep(0.1)
```

In der Endlosschleife wird das Display ständig neu gezeichnet, auch wenn sich nichts ändert. Damit dies nicht allzu häufig geschieht, ist ein *sleep(0.1s)* eingebaut. Man könnte die Programmlogik auch so ändern, dass nur dann das Display neu gezeichnet wird, falls sich etwas ändert. Trotzdem müsste man in der Schleife mit einem *sleep* verhindern, dass sie leer durchlaufen wird, weil sonst unnötig viel Rechenzeit vergeudet würde.

TEILAUFGABE 3: MIT OK-BUTTON DIE WAHL BESTÄTIGEN

Nachdem ein Spieler den Cursor auf eine Zelle gesetzt hat, soll er mit Klicken auf einen OK-Button die Wahl bestätigen. Falls sich der Cursor in einer leeren Zelle befindet, wird der Stein gesetzt und der Spieler gewechselt, d.h. die Variable *player* neu zugewiesen. Mit dieser Ergänzung hast du das Spiel bereits fertiggestellt . Mit dieser Ergänzung hast du das Spiel bereits fertiggestellt und du kannst jemanden zum Spielen herausfordern. Es ist deiner Fantasie und deinem Erfindergeist überlassen, das Spiel. nach deinen eigenen Ideen etwas individueller zu gestalten (andere Farben, usw.).

ERWEITERUNG 1*: TESTEN, OB DAS SPIEL BEENDET IST

Baue eine Funktion *patternAligned(state)* ein, die True zurückgibt, falls im übergebenen Spielzustand *state* drei Steine der einen oder anderen Farbe auf einer Zeile, Spalte oder Dialogonale liegen. Schreibe eine Erfolgsmeldung aus, falls ein Spieler diesen Zustand erreicht und beende das Spiel. Schreibe auch eine Meldung aus, falls das Spiel bei vollem Spielfeld unentschieden ausgeht.

ERWEITERUNG 2*: ONLINE-GAME MIT 2 OXOCARDs

(Für Kommunikationsprofis) Verwende die Datenkommunikation über MQTT, damit zwei Spieler auf verschiedenen Oxocards miteinander auch weit entfernt voneinander spielen können.

ERWEITERUNG 3*: GEGEN COMPUTER SPIELEN

(Für Strategieprofis). Baue eine Gewinnstrategie ein, damit du mit dem Computer als Spielpartner spielen kannst, d.h. nach deinem Stein setzt der Computer seinen Stein gemäss dieser Strategie, bis einer der beiden das Spiel gewonnen hat oder das Spielfeld voll ist.

Dokumentation OXOcard

Modul import: from oxosnake import *
(Real- und Simulationsmodus)

Globale Konstanten und Funktionen:
RED = (255, 0, 0), GREEN = (0, 255, 0), BLUE = (0, 0, 255), BLACK = (0, 0, 0), WHITE = (255, 255, 255),
YELLOW = (255, 255, 0), CYAN = (0, 255, 255), MAGENTA = (255, 0, 255)
BASE_COLORS = (RED, GREEN, BLUE, WHITE, YELLOW, CYAN, MAGENTA)

Funktion	Aktion
makeSnake()	erzeugt eine Schlange und stellt sie dar. Diese besitzt einen Kopf (head) und einen nachfolgenden Schwanz (tail)**)**, der beliebig lang sein kann
makeSnake(name = "Monty", size = 4, pos = (1. 4), headColor = (255, 0, 0), tailColor = (0, 200, 0), penColor = (0, 0, 255), bgColor = (0, 0, 0), heading = 0, speed = 50, hidden = False, penDown = False, dim = 1)	optionale benannte Parameter und ihre Standardwerte (heading: Winkel in Grad zu Norden im Uhrzeigersinn in 45 Grad Schritten)
forward(steps)	bewegt die Schlange um die Anzahl Schritte vorwärts. Ohne Parameter steps = 1
left(angle)	ändert die Blickrichtung um gegebenen Winkel im Gegenuhrzeigersinn (in 45 Grad Schritten)
right(angle)	ändert die Blickrichtung um gegebenen Winkel im Uhrzeigersinn (in 45 Grad Schritten)
getName()	liefert den Namen der Schlange zurück
setName(name)	setzt den Namen der Schlange
setSpeed(speed)	setzt die Geschwindigkeit
getX()	gibt die x-Koordinate des Kopfs zurück
getY()	gibt die y-Koordinate des Kopfs zurück
getPos(x, y)	gibt die Position des Kopfs als Tupel zurück
setPos(x, y)	setzt den Kopf an die gegebene Position (Parallelverschiebung des Schwanzes)
setPos(liste)	setzt den Kopf an die gegebene Position
setHeading(dir)	setzt die Blickrichtung. Winkel zu Norden im Uhrzeigersinn in 45 Grad Schritten
getHeading()	gibt die Blickrichtung zurück. Winkel zu Norden im Uhrzeigersinn in 45 Grad Schritten
penDown()	setzt den Stift beim Kopf ab, sodass bei nachfolgenden Bewegungen eine Spur gezeichnet wird
penUp()	hebt den Stift ab, sodass keine Spur gezeichnet wird
show()	macht die Schlage ist sichtbar

hide()	macht die Schlage unsichtbar (sie ist immer noch vorhanden und eine Spur wird immer noch gezeichnet)
isHiden()	gibt True zurück, falls die Schlange unsichtbar ist
clean()	löscht alle Spuren, lässt aber die Schlage, wo sie ist
getSize()	liefert die Länge der Schlange (Kopf + Schwanz)
setHeadColor(color) setHeadColor(r, g, b)	setzt die Kopffarbe
setTailColor(color) setTailColor(r, g, b)	setzt die Schwanzfarbe
setPenColor(color) setPenColor(r, g, b)	setzt die Spurfarbe
setBgColor(color) setBgColor(r, g, b)	setzt die Hintergrundfarbe
shortenTail()	schneidet das letzte Schwanzelement ab (falls die Schlange länger als 1 ist)
growTail()	Fügt ein zusätzliches Schwanzelement an (erst sichtbar bei der nächsten Bewegung)
intersect()	gibt *True* zurück, falls sich der Kopf auf einem der Schwanzelemente befindet; sonst wird False zurückgegeben
inPlayground()	gibt *True* zurück, falls sich eines der Schlangenelement noch im sichtbaren Bereich befindet,: sonst wird *False* zurückgegeben
headInPlayground()	gibt True zurück, falls sich der Schlangenkopf noch im sichtbaren Bereich befindet,: sonst wird *False* zurückgegeben
spot(color) spot(r, g, b)	setzt den Pixel des Hintergrunds an der aktuellen Position des Schlangenkopfs auf die gegebene Farbe. Die Schlange liegt "oberhalb", d.h. verdecken die so gesetzten Pixels
getRandomColor()	gibt eine zufällige Farbe aus BASE_COLORS zurück. Bei jedem Aufruf wird eine andere Farbe zurückgegeben, bis alle 7 Farben abgerufen sind
getRandomPos()	gibt eine zufällige Position als Tupel zurück. Bei jedem Aufruf wird eine andere Position zurückgegeben, bis alle 64 Positionen abgerufen sind.
reduceBrightness(color, reduction)	dividiert alle 3 Farbkomponenten um den Faktor r und gibt die neue Farbe zurück. Falls eine Komponente grösser als 0 ist, wird mindestens 1 zurückgegeben, damit die Farbart erhalten bleibt
rgbToInt(color)	gibt das gegebenen RGB-Tupel als Int-Farbwert zurück
intToRGB(color)	gibt den gegebenen Int-Farbwert als RGB-Tupel zurück
dim(dimFactor)	dividiert die 3 Farbkomponenten mit den gegebenen Factor (falls eine Komponente > 0 ist, wird sie nie kleiner als 1, damit die Farbart erhalten bleibt)
enableRepaint(False)	unterdrückt das automatische Rendern des Bildbuffers nach jeder Schlangenbewegung. Es muss mit repaint() selbst gerendert werden. forward(n) mit n > 1 wird als nicht mehr als n Einzelschritte aufgefasst
repaint()	rendert den aktuellen Bildbuffer
dispose()	gibt alle Ressourcen frei und vergrössert dadurch den freien Speicherplatz

Klasse Snake
Es können mehrere Schlangen erzeugt werden. Die zuletzt bewegte Schlange liegt über den anderen und ist vollständig sichtbar.

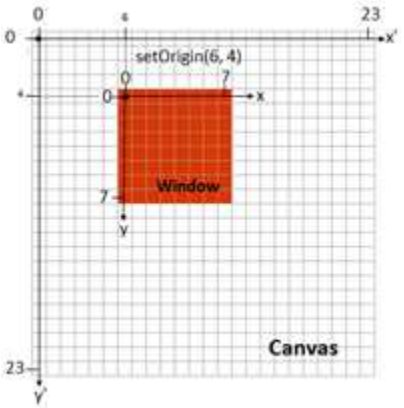
Funktion	Aktion
snake = Snake(name = "Monty", size = 4, pos = (1. 4), headColor = (100, 0, 5), tailColor = (0, 70, 5), penColor = (0, 0, 30), bgColor = (0, 0, 0), heading = 0, speed = 50, hidden = False, penDown = False, dim = 1)	erzeugt ein Snakeobjekt mit den optionalen benannten Paramatern

Alle oben angegebene Funktionene sind auch als Methoden verfügbar. Zusätzlich:

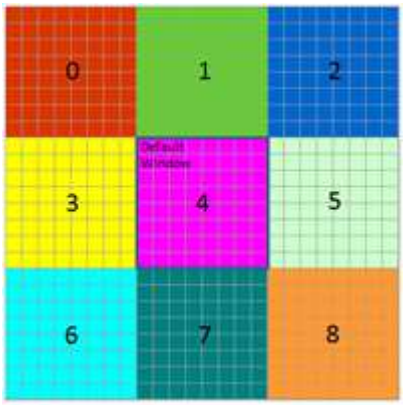
in Touch(nake1, snake2)	gibt True zurück, falls ein Segment der Schlange snake1 und ein Segment der Schlange snake2 übereinander liegen

Modul import: from oxocard import *
(Real- und Simulationsmodus)

Die Oxocard hat einen 8x8 pixel grosse Matrix mit Farb-LEDs (Neopixels). Diese wird durch die Klasse abstrahiert, die einen 24x24 pixel grossen Bildbuffer (Canvas genannt) mit int Farbwerten für die Farbkomponenten r, g, b mit je einem Bereich von 0..255 enthält.



Es wird ein 8x8 pixel grosser Ausschnitt davon (Window genannt) auf den LEDs dargestellt. Canvas und Window haben ein Koordinatensystem, um die einzelnen Pixel zu identifizieren. Der Ursprung liegt bei beiden oben links und die x-Koordinatenachse zeigt nach links und die y-Koordinatenachse nach unten. Standardmässig liegt der Ursprung des x-y-Koordinatensystem beim (x', y') = (8, 8) also ist das Windows im Canvas zentriert. Mit set*Origin(x, y*) kann der Ursprung (Origin) des Windows an die Stelle (x, y) des Canvas verschoben werden. Die Koordinaten für alle Zeichnungsoperationen sind zwar Windowkoordinaten, werden aber unter Berücksichtigung des aktuellen Origin im Canvas durchgeführt. Canvas-Koordinaten ausserhalb 0..23, 0.23 werden ignoriert. Damit stehen im Canvas insgesamt 9 8x8 pixel grosse Bildbereiche (engl *Frames*) zur Verfügung, die sich im Window darstellen lassen. Die Frames haben eine Nummerierung von 0..7.
Grundsätzlich werden alle Zeichnungsoperation im Bildbuffer ausgeführt. Damit sie auch tatsächich auf dem Neopixel-Display sichtbar werden, muss der Bildbuffer auf den LEDs angezeigt werden, was man ***rendern*** nennt. Das Rendering wird mit der Methode *repaint()* durchgeführt.



Standardmässig ist das automatische Rendering eingeschaltet. Dabei wird bei jeder Zeichnungsoperation das *repaint()* automatisch aufgerufen. Oft möchte man bestimmte Zeichnungsoperationen zusammenfassen, bevor man sie als Ganzes sichtbar macht. Dazu schaltet man mit enableRepaint(False) das automatische Rendering aus und führt es an der gewünschten Stelle mit repaint() selbst aus.

Es darf ausserhalb des Canvas gezeichnet werden, wobei solche Pixels unsichtbar bleiben.

Globale Konstanten und Funktionen::
RED = (250, 0, 0), GREEN = (0, 250, 0), BLUE = (0, 0, 250), BLACK = (0, 0, 0), WHITE = (255, 255, 255), YELLOW = (255, 255, 0), CYAN = (0, 255, 255), MAGENTA = (255, 0, 255)
BASE_COLORS = (RED, GREEN, BLUE, WHITE, YELLOW, CYAN, MAGENTA)

Funktion	Aktion
clear(), clear(color)	setzt alle Pixels des Canvas auf die Farbe *color*. Ohne Parameter werden alle Pixels gelöscht (color = BLACK)
clearWindow(), clearWindow(color)	setzt den Bereich des Canvas, über dem das aktuelle Window liegt, auf die Farbe *color*. Ohne Parameter werden die Pixels gelöscht (color = BLACK)
clearFrame(frameNumber) clearWindow(frameNumber, color)	setzt die Pixels des gegebenen Frames auf *color*. Ohne Parameter werden die Pixels gelöscht (color = BLACK)
setOrigin(x, y)	setzt den Ursprung des Windows auf x, y (auch Punktliste/Tupel). x, y sind positive oder negative Integers. Liegen die Pixels ausserhalb des 24x24 pixel-Bereichs, sind die Pixels schwarz
getOrigin()	gibt den Origin als Tupel zurück
def setFrame(frameNumber)	setzt das Window auf das gegebene Frame
dim(dimFactor)	vermindert die Farbwerte aller Pixels um den gegebenen Faktor. Falls eine gegebene Komponente > 0 ist, wird die reduzierte Farbkomponente nicht kleiner als 1, damit die Farbart erhalten bleibt
enableRepaint(False)	deaktiviert das automatische Neuzeichnen (Rendering) des Windows
repaint()	zeichnet das Window neu
dot(x, y, color)	setzt ein einzelnes Pixel des Canvas an der Position x, y auf die gegebene Farbe
line(x, y, dir, length, color)	zeichnet eine Linie beginnend an der Position x, y in der gegebenen Richtung mit gegebener Länge und Farbe (*dir* hat die Werte 0: Ost, 1: Nord-Ost, 2: Nord, 3: Nord-West, 4: West, 5: Süd-West, 6: Süd, 7: Süd-Ost)
arrow(x, y, dir, length, color)	zeichnet einen Pfeil beginnend an der Position x, y in der gegebenen Richtung mit gegebener Länge und Farbe (*dir* hat die Werte 0: Ost, 1: Nord-Ost, 2: Nord, 3: Nord-West, 4: West, 5: Süd-West, 6: Süd, 7: Süd-Ost)
rectangle(ulx, uly, w, h, color)	zeichnet ein Rechteck mit gegebener oberen linken Ecke ulx, uly mit Breite h und Höhe h in der gegebenen Farbe
circle(xcenter, ycenter, r, color)	zeichnet einen approximativen Kreis mit gegebenem Zentrum xcenter, ycenter und Radius r in der gegebenen Farbe
fillCircle(xcenter, ycenter, r, color)	zeichnet einen approximativen gefüllten Kreis mit gegebenem Zentrum xcenter, ycenter und Radius r in der gegebenen Farbe
getColor(x, y)	gibt den Farbwert des Canvas an der Stelle (x, y) zurück. Befindet sich (x, y) ausserhalb des Canvas, wird (0, 0, 0) (BLACK) zurückgegeben
getColorInt(x, y)	gibt den Farbwert des Canvas an der Stelle (x, y) zurück (als Integer). Befindet sich (x, y) ausserhalb des Canvas, wird zurückgegeben
image(matrix)	setzt die Pixels aus der gegebenen Pixelmatrix im Bildbuffer ein. Die Einsetzung beginnt beim aktuellen Origin. matrix ist eine Liste oder ein Tupel in folgendem Format:

	((c00, c01, c02,.. c07), (c10, c11, c12,.. c17), ... (c70, c71, c72,.. c77)) wo c die Farbwerte sind (als RGB-Tupel oder als Farb-Integer
translate(vector)	verschiebt die Pixels des ganzen Canvas um den gegebenen Verschiebungsvektor. Pixels, die von ausserhalb hineingeschoben werden, sind schwarz
rotate(centerx, centery, angle)	dreht die Pixel des Canvas mit dem gegebenen Rotationsmittelpunkt (Windowkoordianten) um den gegeben Winkel (in Grad im Uhrzeigersinn). Pixels, die von ausserhalb hineingeschoben werden, sind schwarz
insertBigChar(char, charColor, bgColor)	stellt ein Zeichen mit Zeichen- und Hintergrundfarbe mit grossem Font dar. Defaults: charColor = (255, 255, 255), bgColor = (0, 0, 0)
display(chars, charColor, bgColor)	stellt bis maximal zwei Zeichen von chars mit Zeichen- und Hintergrundfarbe dar (Zahlen werden in Strings umgewandelt). Wird ein negativer Integer übergeben, erscheint ein Minuszeichen unter der linken Ziffer. Defaults: charColor = (255, 255, 255), bgColor = (0, 0, 0)
bigTextScroll(text, textColor, bgColor, speed)	zeigt mit grossem Font einen nach links scrollenden Text mit gegebener Zeichen- und Hintergrundfarbe an. Dieser wird mit einer einstellbaren Geschwindigkeit im Bereich 1..12 verschoben. Defaults: textColor = (255, 255, 255), bgColor = (0, 0, 0), speed = 6)
smallTextScroll(text, textColor, bgColor, speed)	zeigt mit kleinem Font einen nach links scrollenden Text mit gegebener Zeichen- und Hintergrundfarbe an. Dieser wird mit einer einstellbaren Geschwindigkeit im Bereich 1..12 verschoben. Defaults: textColor = (150, 150, 150), bgColor = (0, 0, 0), speed = 5)
getArray()	gibt den 24x24 pixel Canvas als Tupel mit Zeilentupels der Farbwerte zurück
rgbToInt(color)	gibt das gegebenen RGB-Tupel als Int-Farbwert zurück
intToRGB(color)	gibt den gegebenen Int-Farbwert als RGB-Tupel zurück
reduce(color, r)	dividiert alle 3 Farbkomponenten um den Faktor r und gibt die neue Farbe zurück. Falls eine Komponente grösser als 0 ist, wird mindestens 1 zurückgegeben, damit die Farbart erhalten bleibt
getRandomColor()	gibt eine zufällige Farbe aus BASE_COLORS zurück. Bei jedem Aufruf wird eine andere Farbe zurückgegeben, bis alle 7 Farben abgerufen sind
getRandomPos()	gibt eine zufällige Position als Tupel zurück. Bei jedem Aufruf wird eine andere Position zurückgegeben, bis alle 64 Positionen abgerufen sind.
dispose()	gibt alle Ressourcen frei und vergrössert dadurch den freien Speicherplatz

Modul import: from oxobutton import *
(Real- und Simulationsmodus)

Tasten Identifikationen: BUTTON_L1, BUTTON_L2, BUTTON_L3, BUTTON_R1, BUTTON_R2, BUTTON_R3

Funktion	Aktion

button = Button(buttonID)	erzeugt einen Button mit gegebener ID
button = Button(buttonID, callback)	erzeugt einen Button mit gegebener ID und einer Callbackfunktion
button.isPressed()	liefert True, falls der Button momentan gedrückt ist; sonst False
button.wasPressed()	liefert True, falls der Button seit dem letzten Aufruf geklickt wurde; sonst False

Modul import: from oxoaccelerometer import *
(Real- und Simulationsmodus, im Simulationsmodus muss auch das Modul oxocard importiert werden)

Funktion	Aktion
acc = Accelerometer.create()	erzeugt einen Sensor mit Standardwerten: trigger_level = 10, rearm_time = 0.2, single-click auf allen 3 Achsen und keinem Callback und gibt seine Referenz zurück
acc = Accelerometer.create(callback)	dasselbe unter Angabe einer Callbackfunktion
acc = Accelerometer.create(callback, trigger_level = 10, rearm_time = 0.2, click_config = SINGLE_CLICK_X | SINGLE_CLICK_Y | SINGLE_CLICK_Z)	dasselbe mit benannten Parametern: trigger_level ist der Triggerpegel im Bereich 1..100 (willkürliche Einheiten), der überschritten werden muss, damit ein Click-bzw. Double-Clickevent registriert wird (kleinere Werte machen den Sensor also empfindlicher) (default: 10) rearm_time ist die Zeit in Sekunden, während der der Sensor inaktiviert wird, bis er wieder Clickevents registriert (default: 0.2) Mit click_config kann die Triggerart konfiguriert werden. Or-Kobination von Accel.SINGLE_CLICK_X Accel.SINGLE_CLICK_Y Accel.SINGLE_CLICK_Z Accel.DOUBLE_CLICK_X Accel.DOUBLE_CLICK_Y Accel.DOUBLE_CLICK_Z
acc.get()	gibt ein Tupel mit 3 Float-Werten der Beschleunigungskomponenten (acc_x, acc_y, acc_z) zurück (in m/s^2 im Bereich +-2g)
acc.getX()	gibt die x-Komponente der Beschleunigung zurück (in m/s^2 im Bereich +-2g)
acc.getY()	gibt die y-Komponente der Beschleunigung zurück (in m/s^2 im Bereich +-2g)
acc.getZ()	gibt die z-Komponente der Beschleunigung zurück (in m/s^2 im Bereich +-2g)
acc.getRoll()	gibt den Drehwinkel bei einer Seitwärtsdrehung (in Grad im Bereich -90° bis +90°)
acc.getPitch()	gibt den Drehwinkel beim Kippen nach vorne, bzw. nach hinten (in Grad im Bereich -90° bis +90°)
acc.wasClicked()	gibt True zurück, falls der Sensor seit dem letztem Aufruf einen Clickevent erfahren hat; sonst False
acc.getTemperature()	gibt die Temperatur des Chips zurück (in Grad Celsius, Genauigkeit +-2 degC)
acc.dispose()	gibt alle Ressourcen frei

Modul import: from tcpcom import *
(nur Realmodus)

Klasse Wlan (alle Methoden static):

Funktion	Aktion
Wlan.connect(ssid, pwd, timeout = 10, verbose = False)	verbindet mit einem Accesspoint (Hostspot, WLAN Router) mit gegebenen SSID und Passwort. Gibt True zurück, falls die Verbindung innerhalb des gegebenen Timeout (in sec) gelingt; sonst wird False zurückgegeben. Mit *verbose = True* werden Debuginformationen ausgeschrieben
Wlan.isWlanConnected()	gibt True zurück, falls eine Verbindung zum Accesspoint besteht
Wlan.getIfConfig()	gibt ein Tupel (IP_address, mask, gateway_ip) zurück; None, falls keine Verbindung besteht
Wlan.getMyIPAddress()	gibt die IP-Adresse zurück, die der Accesspoint vergeben hat; leerer String, falls keine Verbindung besteht
Wlan.scan()	gibt ein Tupel mit Netzwerkinfos über alle gefundenen Accesspoints zurück. Format: ssid (string), channel (int), rssi Signalstärke in dB (int)
Wlan.activateAP(ssid = "ESP", password = "aabbaabbaabb", channel = 10, verbose = False)	aktiviert einen Accesspoint mit gegebenen SSID, Passwort und Kanal (1..13). Für password = "" (leer) ist der Zugang passwortfrei

Modul import: from mqttclient import MQTTClient (nur Realmodus)

Klasse MQTTClient (abgeleitet von umqtt.simple.MQTTClient)
(siehe: http://github.com/micropython/micropython-lib/tree/master/umqtt.simple)

Funktion	Aktion
client = MQTTClient(host, port = 0, user = None, password = None)	erzeugt einen MQTT Client unter Angabe der Broker IP- Adresse. Falls nötig, werden die mitgegebenen Authentifizierungsangaben verwendet. Die Verbindung wird noch nicht hergestellt
client.connect(cleanSession = True)	erstellt eine Verbindung zum Broker. Falls cleanSession = *True* ist, werden alle vorher gesendeten Daten des gleichen Clients gelöscht
client.ping()	sendet einen Ping-Request an den Server, damit dieser die Verbindung offen hält
client.publish(topic, payload, retries = 10, retain = False, qos = 0)	sendet zum gegebenen Topic eine Message (payload). Falls die Verbindung nicht mehr offen ist, wird so oft ein neues *connect()* gemacht, bis die Zahl *retries* erreicht ist. Falls *retain = True* wird diese Message als die letzte good/retain Message betrachtet. qos ist der Quality of Service level (nur 0, 1 unterstützt). Gibt True zurück, falls das Publish erfolgreich war; False, falls die Verbindung nicht mehr aufgebaut werden konnte
client.setCallback(onMessageReceived)	registriert die Callbackfunktion *onMessageReceived()*, die beim Erhalt einer Message aufgerufen wird (Funktionsname beliebig)

onMessageReceived(topic, payload)	oben registrierte Callbackfunktion mit den Parametern *topic* und *payload* (type: string)
client.subscribe(topic, qos = 0	abonniert das gegebene Topic mit dem gegegeben qos level (siehe bei *publish()*)
client.disconnect()	schliesst die Verbindung und beendet den internen Subscriber Poll Thread

Modul import: from music import*
(nur Realmodus)

Funktion	Aktion
playTone(frequency, duration)	spielt einen Ton mit gegebener Frequenz (in Hz) und Dauer (in ms) ab. Frequenzbereich: ca. 100 - 1500 Hz. Die Frequenz kann auch in Notennotation (als String) angegeben werden: 'r', 'C', 'C#', 'D'...,'c', c#, d...,'c2','c2#','d2'...,'c3'. 'r' ist eine Pause, 'C' Beginn der Oktave mit 131 Hz, 'c' Beginn der Oktave bei 262 Hz, 'c2' Beginn der Oktave bei 523 Hz, 'c3': 1046 Hz. Hinter der Notenangabe kann ein Doppelpunkt und ein Verlängerungsfaktor für die Dauer angegeben werden. Beispiel einer Notensequenz: ('c2:3', 'g', 'f#', 'g', 'g#:3', 'g', 'r', 'h:2', 'c2') Die Funktion blockiert, bis der Ton fertig gespielt ist
playSong(song, duration = 150)	spielt die Töne in der Notenliste (oder Tupel) *song* mit gegebener Dauer *duration* ab.
beep(n, frequency = 1000)	spielt n kurze Tonsignale mit gegebener Frequenz ab

Vordefinierte Songs:
ENTERTAINER, RINGTONE, BIRTHDAY, JUMP_UP, JUMP_DOWN, DADADADUM, POWER_UP, POWER_DOWN, PUNCHLINE, WEDDING, BOOGYWOODY, PRELUDE

Modul import: from utils import *
(nur Realmodus)

Klasse Vektor

Funktion	Aktion
a = Vector(x, y)	erzeugt ein 2-dimensionales Vektorobjekt mit gegebenen Koordinaten
a = Vector((x, y))	dasselbe mit gegebenem Koordinatentupel (oder Liste).
a = Vector()	dasselbe mit den Koordinaten (0, 0)
c = a.add(b)	gibt einen neuen Summenvektor der beiden Vektoren a und b zurück
c = a.sub(b)	gibt einen neuen Differenzvektor der beiden Vektoren a und b zurück
c = a.mult(k)	gibt einen neuen Vektor mit skalarer Multiplikation zurück
print(a)	schreibt die Komponenten aus
a[0], a[1]	Komponenten in Indexschreibweise (lesend und schreibend)

Klasse I2C_com
(nur Realmodus)

Funktion	Aktion
i2c = I2C_com(i2c_address, scl = Pin(22), sda = Pin(21)	erzeugt ein I2C-Schnittstellenobjekt mit gegebener I2C-Adresse an den gegebenen Pins für SCL und SDA
i2c.readReg(cmd)	liest ein einzelnes Byte an der gegebenen Registeradresse cmd (int) (Rückgabewert: int)
i2c.writeReg(cmd, data)	schreibt ein Datenbyte data (int) an der gegebenen Registeradresse cmd (int)
i2c.readBlock(cmd, len)	liest eine Folge von Bytes beginnend bei der gegebenen Registeradresse cmd (int). Rückgabewert: Liste mit ints
i2c.writeBlock(cmd, data)	schreibt eine Folge von Bytes beginnend bei der gegebenen Registeradresse cmd (int). data ist eine Liste mit ints
i2C_com.dispose()	gibt den I2C-Port wieder frei (für alle Geräte)

from machine import Pin, I2C
i2c = I2C(scl = Pin(21), sda = Pin(22))
devices = i2c.scan()
liefert alle I2C-Geräteadressen, die an den gegebenen Pins für SCL und SDA angeschlossen sind.

Modul import: import weather
(nur Realmodus)

Funktion	Aktion
info = weather.request(ssid, password, city, key)	verbindet über den WiFi-Hostspot mit gegebener SSID und Passwort mit Host **http://openweathermap.org** und macht einen Wetterdaten-Request für den gegebenen Ort. Dabei wird der gegebene Authorisierungsschlüssel verwendet. Dieser kann man man kostenfrei auf diesem Host beziehen. (Wird der Parameter weggelassen, so wird ein Standardschlüssel verwendet, der maximal 60 Anfragen / min für alle Nutzer erlaubt.) Rückgabewert: Dictionary mit den Feldern: "status" "OK", Fehlerbeschreibung (string) "temp" Temperatur in °C (float) "pressure" Luftdruck in hPa (mbar) (float) "humidity" Luftfeuchtigkeit in % (int) "temp_min" Tagesminimum der Temperatur in °C (float) "temp_max" Tagesmaximum der Temperatur in °C (float) "description" Wetterlage in Worten (deutsch) (string) "sunrise" Sonnenaufgang in Universal Time (UTC) (string) "sunset" Sonnenuntergang in Universal Time (UTC) (string) "datetime" Datum - Uhrzeit (UTC) der Wettererfassung (string)

ÜBER DIE AUTOREN

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

Aegidius Plüss war an der Universität Bern Professor für Informatik und deren Didaktik und hat in dieser Tätigkeit viele Informatiklehrkräfte aus- und weitergebildet, die heute aktiv an den Schulen tätig sind. Er gilt als Urgestein in der Informatikausbildung und hat eine grosse Erfahrung mit vielen Programmiersprachen und Computersystemen. Er ist für die textliche Formulierung dieses Lehrgangs und für die didaktischen Libraries in TigerJython verantwortlich.

KONTAKT

help@tigerjython.ch