Simple is better than complicated
Mit gutem Beispiel vorangehen
Üben, üben, üben...

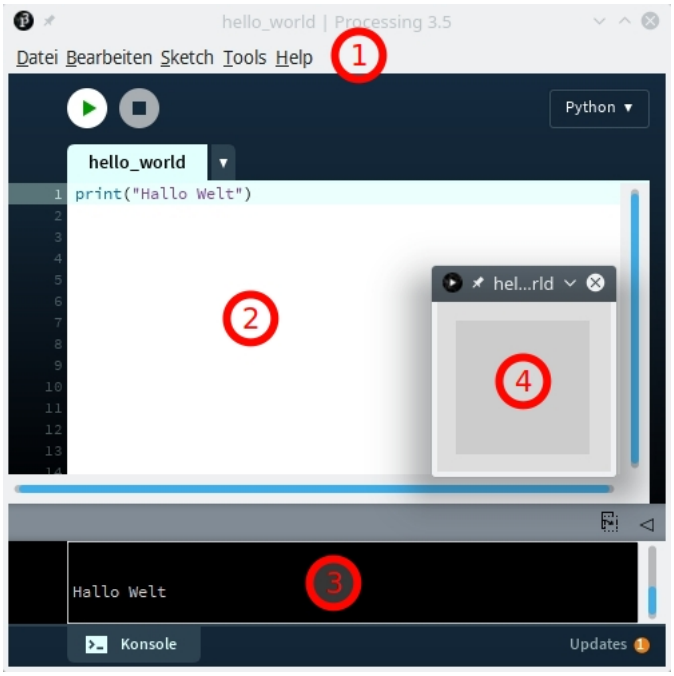
Diese Einführung in die Programmierung mit Hilfe von Processing.py ist für das geleitete Selbststudium gedacht.
An Hand von Beispielen werden die wichtigsten Grundkonzepte der Programmierung nach und nach eingeführt.

Die Aufgaben sind mit aufsteigendem Schwierigkeitsgrad von  (grundlegend, bzw. einfach) über  (weiterführend) bis  (herausfordernd) gekennzeichnet.

Hallo Welt

Umgebung von Processing.py
Befehl ausführen
Text in der Konsole ausgeben
Mehrere Befehle der Reihe nach ausführen
Pixel mit Koordinaten ansprechen

Processing.py ist eine spezialisierte, absichtlich sehr reduzierte Umgebung für die graphische Programmierung mit der Programmiersprache Python. Ein Programm in Processing.py wird darum auch *Sketch* (Skizze) genannt.



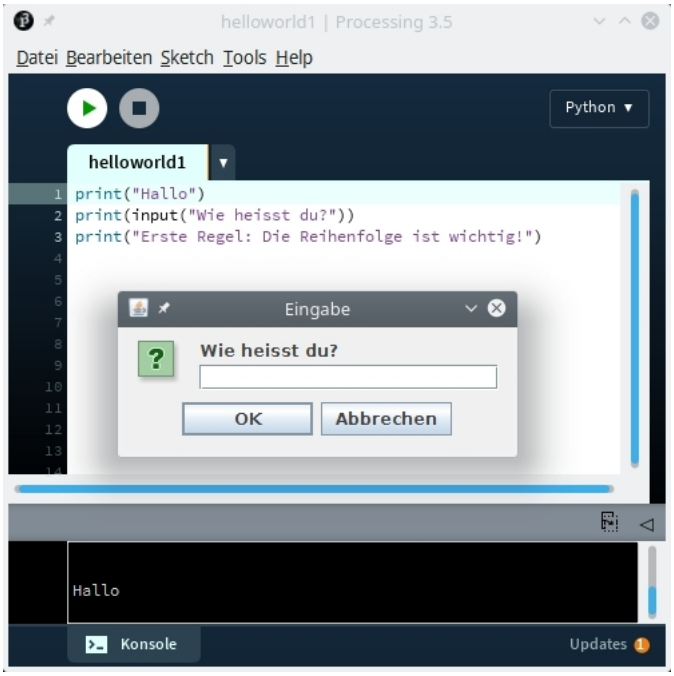
1. Die *Menuleiste* enthält Untermenus:
   Datei für das Erstellen, Öffnen, Speichern etc. von Sketches
   Bearbeiten für das Bearbeiten von Sketches
   Sketch für das Ausführen und Stoppen von Sketches
   Tools mit Werkzeugen
   Help mit Hilfeseiten
2. Der *Programmbereich* zeigt den Inhalt des aktuellen Sketches.
3. In der *Konsole* werden Ausgaben und Hinweise angezeigt.
4. Im *Graphikfenster* werden die Resultate der Graphikfenster dargestellt.

Der Programmbereich enthält zeilenweise die *Befehle*, die ausgeführt werden sollen. Hier ist es genau ein Befehl

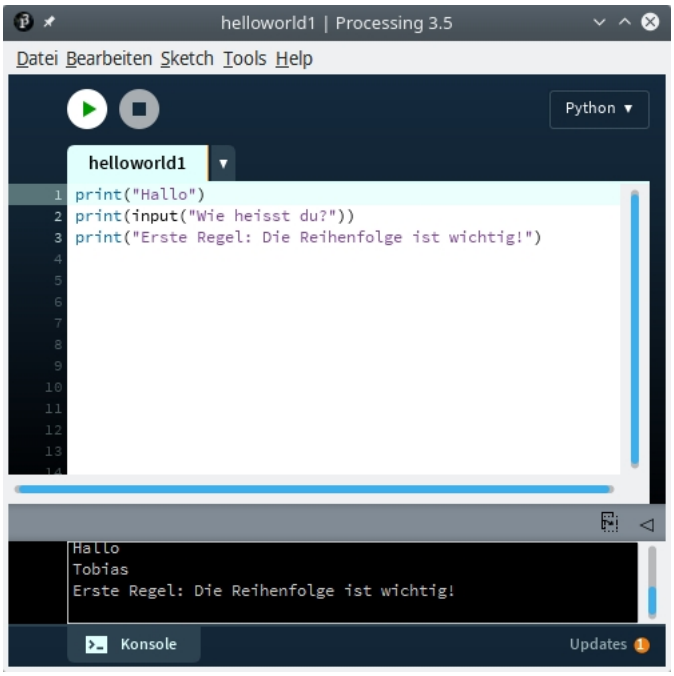
```
print("Hallo Welt")
```

Ein Befehl besteht immer aus einem *Namen* (hier **`print`**) gefolgt von einem Klammerpaar `()`, das möglicherweise *Parameter* enthält (hier den Text `"Hallo Welt"`)

Die *Befehle* eines Programms werden in der Reihenfolge ausgeführt, wie sie hingeschrieben werden.

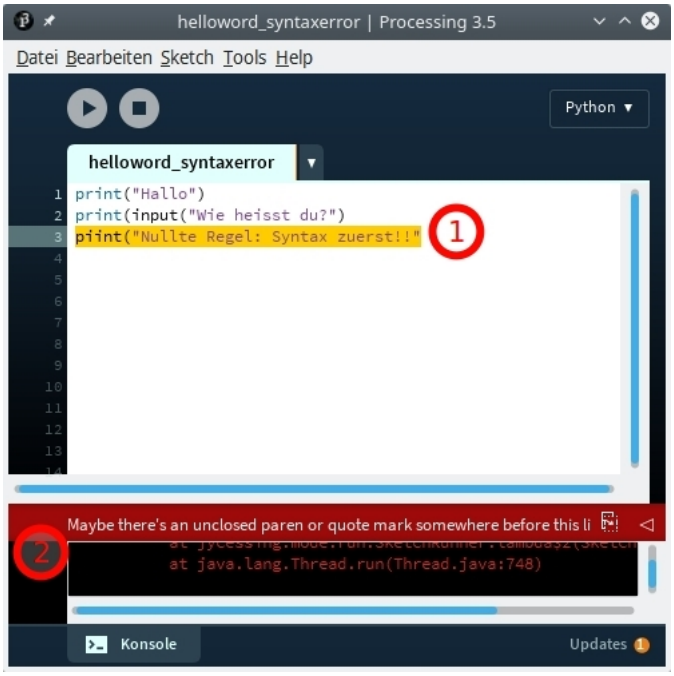


Der zweite Befehl ist zusammengesetzt, der Parameter ist wieder ein Befehl **`input`**, welcher einen Dialog zur Eingabe eines Textes öffnet.



Danach wird erst die dritte Zeile ausgeführt.

Programmiersprachen, und hier bildet Processing.py leider *keine* Ausnahme, sind leider sehr stur, was die Syntax betrifft. Jede vergessene Klammer, jedes vergessene Anführungszeichen, jeder Schreibfehler führen dazu, dass das Programm nicht ausgeführt werden kann.



Processing.py versucht bei Syntaxfehlern zu helfen, indem

1. die Stelle gelb markiert wird, wo das erste Problem erkannt wurde, und
2. in der Konsole ein Text angezeigt wird, der auf dessen Ursache hindeutet.

Häufig befindet sich die Fehlerquelle in der Zeile *vor* der markierten.

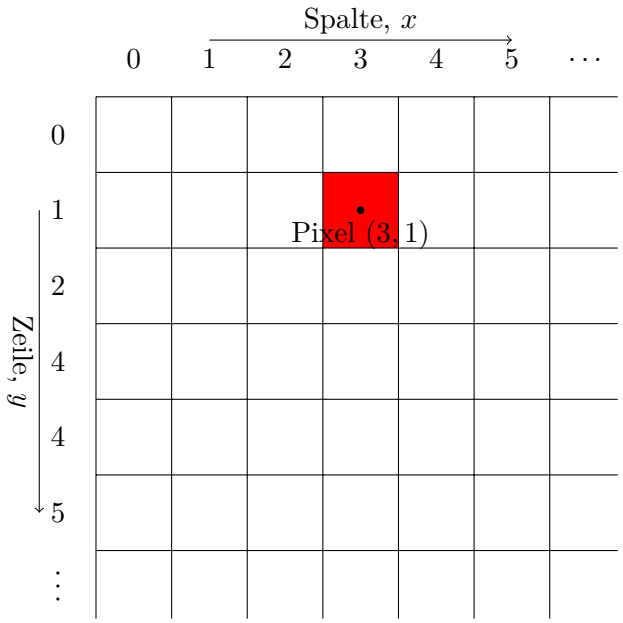
Die Hinweise sind auf Englisch gehalten und betreffen nicht immer präzise das eigentliche Problem.

Arbeitsauftrag Bringe das Beispielprogramm zum Laufen, indem du die Syntaxfehler korrigierst.

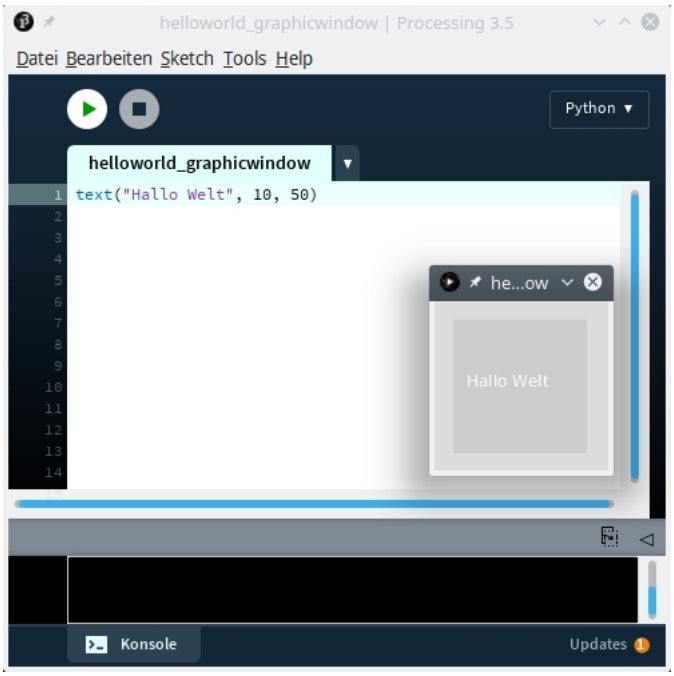
Da die Hauptausrichtung von Processing.py auf Graphik zielt, bringen wir unser "'Hallo Welt"'-Beispiel zunächst ins Graphikfenster.
Dazu muss man verstehen, dass dieses aus einem rechteckigen Feld von einzelnen Pixeln besteht, die *Zeilen-* und *Spalten*-weise angeordnet sind. Ein Pixel kann man sich als einen Punkt des Bildschirms vorstellen, der in einer gewählten Farbe leuchtet. Somit hat jedes dieser Pixel eine eindeutige Adresse

$$(x, y)$$

die aus der Nummer x seiner Spalte und der Nummer y seiner Spalte besteht.



Anstelle des `print`-Befehls von vorhin verwenden wir deshalb nun den `text`-Befehl mit 3 Parametern: dem Text, der Spalten und der Zeilen-Nummer, wo der Text dargestellt werden soll.

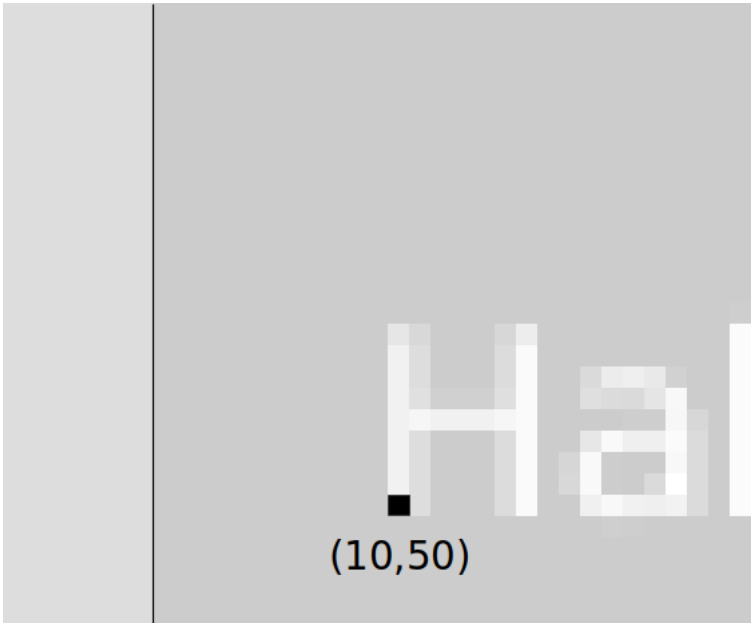


Standardmässig verwendet Processing.py ein Graphikfenster mit 100 Spalten und 100 Zeilen. Man spricht von einem *Default*, der jederzeit geändert werden kann.

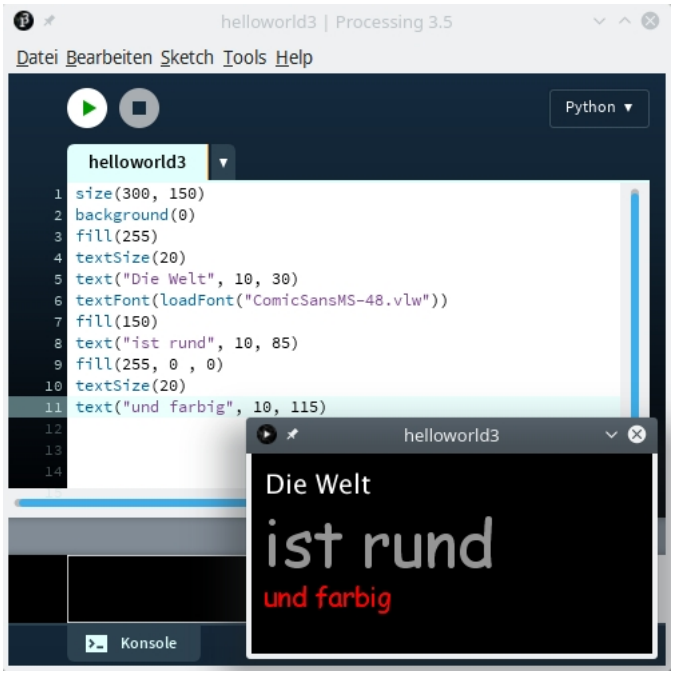
Graphikattribute

Farben mischen
Textattribute festlegen

Der im Graphikfenster dargestellte Text komplizierter ist, als es auf den ersten Blick erscheint. Der linke Rand des Pixelrasters und das Pixel $(10, 50)$, wo der Text beginnt, ist nachträglich markiert worden. Es handelt sich also um relativ komplizierte Graphikelement, bestehend aus Linien und Kurven.



Die Attribute dieser Graphikobjekte und des gesamten Graphikrasters können mit vordefinierten Befehlen geändert werden.




```
size(300,150)
```

Stellt die Grösse des Pixelrasters auf 300 Spalten und 150 Zeilen ein

```
  background(0)
```

Legt die Hintergrundfarbe als Schwarz fest.

> Grautöne können als eine ganze Zahl von 0 (Schwarz) bis 255 (Weiss) angegeben werden.

```
fill(255)
```

Legt die Füllfarbe als Weiss fest.

```
textSize(20)
```

Stellt die Textgrösse auf 25 Pixel ein

```
textFont(loadFont("ComicSansMS-48.vlw"))
```

Lädt den Font `ComicSansMS-48` und stellt diesen als verwendete Schriftart ein.
(Beachte: Es wird wieder ein zusammengesetzter Befehl verwendet!)

> Bevor ein Font verwendet kann, muss er mit dem Menupunkt `Tools|Schrift erstellen...` aus einer installierten Systemschrift erzeugt werden.

```
fill(255, 0, 0)
```

Legt die Füllfarbe als Rot fest.

> Mischfarben können durch drei ganze Zahlen von 0 bis 255 angegeben werden, welche je den Rot-, Grün- und Blauanteil der Farbe bestimmen. Es handelt sich dabei um eine *additive Farbmischung*.

Arbeitsauftrag Schreibe Texte in den Grundfarben Rot, Grün, Blau, Gelb, Magenta (Violett) und Türkis (Zyan).
Experimentiere mit weiteren Mischfarben.

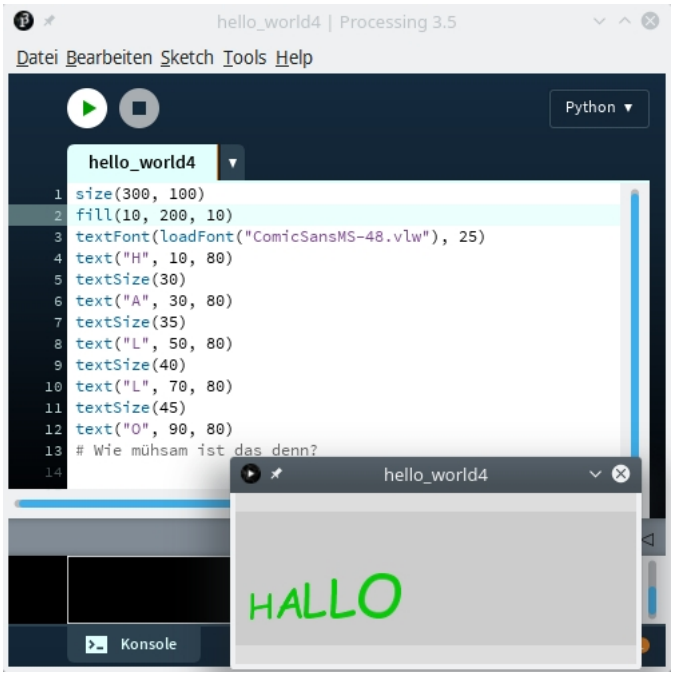
`hello_world_3a`

Arbeitsauftrag Arbeite das Tutorial zu den Farben und unterschiedlichen Farbmodellen in Processing.py durch: `https://py.processing.org/tutorials/color/`

Wiederholungen

Das DRY-Prinzip
Das Programm nimmt uns repetitive Aufgaben ab
Werte in Variablen speichern

Angenommen wir wollen mit den Textgrössen weiterspielen und unseren Text "'Hallo Welt"' von links nach rechts immer grösser wachsen lassen. Naheliegend ist ein folgender erster Versuch mit den bisher bekannten Mitteln:




```
# Wie mühsam ist das denn?
```

Alles was in einer Zeile nach dem Raute-Symbol `#` steht gilt als *Kommentar* und wird von Processing.py ignoriert.

Kommentare sind extrem wichtig, weil sie die Lesbarkeit eines Programms für den menschlichen Leser erhöhen.
Ohne Kommentare versteht man normalerweise den Aufbau auch eines selbstgeschriebenen Programms nach kurzer Zeit nicht mehr ohne grossen Aufwand!

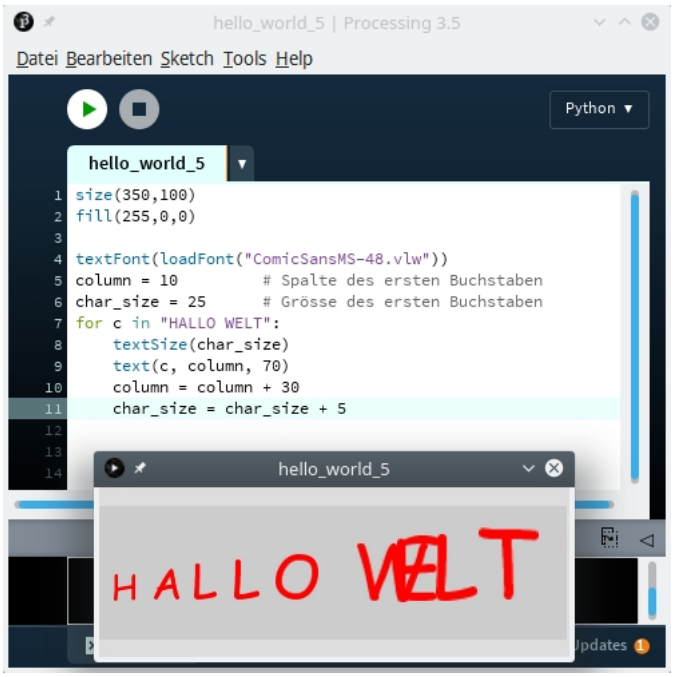
Arbeitsauftrag Beende das Programm mit dem ganzen Text "'Hallo Welt"' so, dass die Schriftgrösse möglichst genau linear von links nach rechts zunimmt.

`hello_world_3a`

Selbstverständlich kann eine solche repetitive Lösung nicht gewünscht sein. Im Gegenteil, eine der wichtigsten Maximen in der Programmierung lautet: Don't Repeat Yourself, auch das DRY-Prinzip genannt.

Alle Wiederholungen sind dem Computer zu überlassen. Das von uns geschriebene Programm soll weitestgehend DRY sein!

Da ein Text eine Abfolge von einzelnen Buchstagen ist, bietet Processing.py einen einfachen Weg an, dieselben Befehle für jeden einzelnen Buchstaben darin auszuführen:




```
column = 10                    # Spalte des ersten Buchstaben
char_size = 25                 # Grösse des ersten Buchstaben
```

Die Position und die Grösse der Buchstaben ändern sich und müssen daher in Variablen gespeichert werden!

Variablen sind Namen für Werte. Diese Namen können aus Buchstaben, Ziffern und das Unterstreichungszeichen `_` bestehen (aber nicht mit einer Ziffer beginnen).
Beispiele: `c`, `character`, `x_1`

Bei der Wertzuweisung `x = 10` ist die *Reihenfolge* entscheidend! Links steht der Variablenname, rechts der Wert, der der Variable zugewiesen wird

```
for c in "HALLO WELT":
```

Der Variable `c` werden der Reihe nach alle Buchstaben der Zeichenkette `HALLO WELT` zugewiesen.

```
    textSize(s)
    text(c, column, 70)
    column = column + 30
    char_size = char_size + 5
```

Alle danach folgenden und gleich weit eingerückten Befehle werden mit allen einzelnen Buchstaben der Zeichenkette als Wert für die Variable `c` wiederholt.

Durch das gemeinsame Einrücken werden mehrere Befehle zu einem *Block* zusammengefasst.

Hat man einer Variablen einmal einen Wert zugewiesen, kann ihr Name überall an Stelle dieses Wertes eingesetzt werden.

Variablen werden *verwendet*, indem ihr Name anstelle ihres Wertes eingesetzt wird

Die Wertzuweisung `x = x + 30` erhöht den Wert der Variablen `x` um 30 und ist nicht als Gleichung (wie in der Mathematik) zu lesen.

> Bei einer Wertzuweisung wird *zuerst* der Ausdruck rechts ausgewertet und *anschliessend* das Ergebnis der Variable links zugewiesen.

Das Ergebnis ist natürlich noch nicht befriedigend, weil die horizontalen Abstände von der Breite des jeweiligen Zeichens abhängen sollte!

Arbeitsauftrag Verbessere das Programm so, dass die horizontalen Abstände zwischen den einzelnen Zeichen an deren Breite angepasst wird.

`hello_world_5b`

Hinweis Der Befehl `textWidth(c)` ergibt die Breite des Textes in der Variable `c`.

Leider wächst die Zeichengrösse jetzt nicht mehr linear, weil sie von einem Buchstaben zum nächsten um denselben Wert (5) zunimmt.

Arbeitsauftrag Verbessere die letzte Version des Programms noch einmal so, dass die Zeichengrösse wieder linear zunimmt.

`hello_world_5c`

Arbeitsauftrag Eine kleine Herausforderung: Lasse das "HALLO WELT" anschliessend wieder rückwärts kleiner werden.

`hello_world_5d`

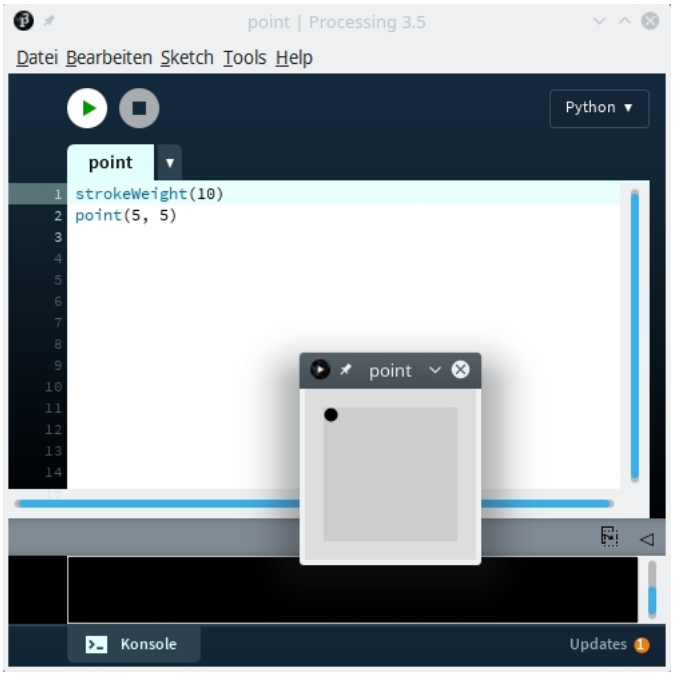
Hinweis Verwende diesmal eine Variable für den Text, um dem DRY-Prinzip zu genügen.

Hinweis Der Befehl `reverse(text)` ergibt die Buchstaben der Zeichenkette `text` in umgekehrter Reihenfolge.

Punkte

Punkte setzen
Listen verwenden
Bereiche sind spezielle Listen

Die Hauptstärke von Processing.py liegt in der Programmierung von Graphiken. Wir wenden uns daher nun den elementaren Elementen solcher Computergraphiken zu.

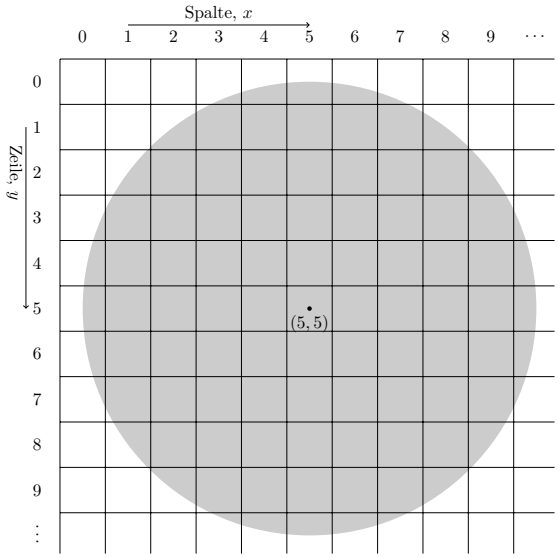



```
strokeWeight(10)
```

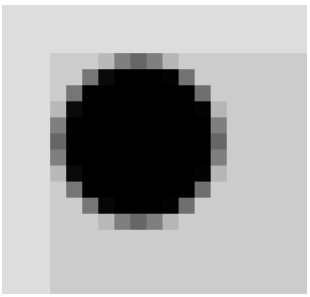
Stellt die Strichdicke auf 10 Pixel ein.

```
point(5, 5)
```

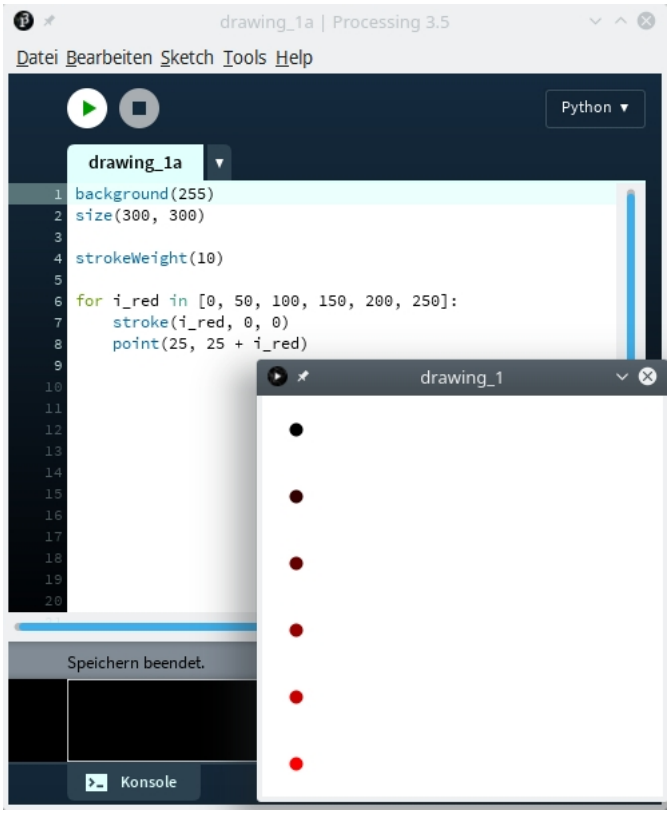
Zeichnet einen Punkt als ausgefüllter Kreis mit Zentrum bei Spalte 5 und Zeile 5 und der voreingestellten Strichdicke als Durchmesser.



Vergrössern wir hier die Ausgabe, um die einzelnen Pixel sehen zu können, sehen wir, dass die Darstellung schon recht komplex ist. Der Punkt wird als angenäherter Kreis dargestellt, wobei einiger Aufwand betrieben wird, den Rand möglichst zu glätten.



Als nächstens zeichnen wir eine Reihe von Punkten mit zunehmendem Rotanteil in der Farbe.




```
for i_red in [0, 50, 100, 150, 200, 250]:
```

Die **for**-Wiederholung funktioniert nicht nur mit Zeichenketten, sondern auch mit Listen.

Eine Liste in Python wird mit eckigen Klammern um die durch Kommas getrennte Aufzählung der Elemente geschrieben.

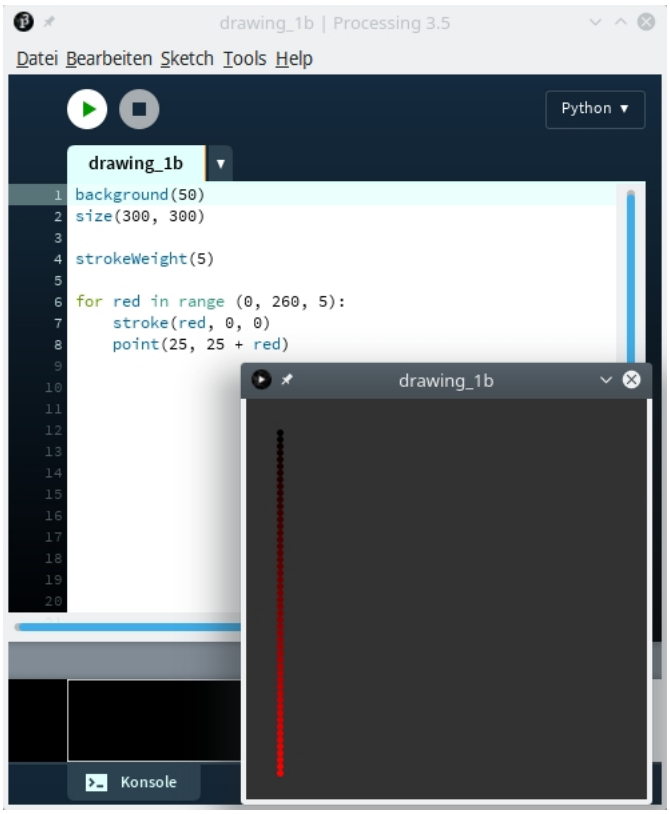
`[1, 2, 3]` ist die Liste der Zahlen 1, 2 und 3; `["Eins", "Zwei", "Drei"]` ist die Liste der Wörter "'Eins"', "'Zwei"' und "'Drei"'.

```
    stroke(i_red, 0, 0)
```

Stellt die Zeichenfarbe mit dem Rotanteil `i_red` und Grün-, bzw Blauanteil jeweils 0 ein.

```
    point(25, 25 + i_red)
```

Zeichnet einen Punkt in Spalte 25 und der Zeile $25 + r$, wenn r der aktuelle Rotanteil ist.
Für längere Listen, zum Beispiel wenn wir eine feinere Farbabstufung wollen, wird die statische Schreibweise mit den eckigen Klammern mühsam.




```
for i_red in range(0, 260, 5):
```

Die Liste der Rot-Anteile wird hier mit dem **range**-Befehl angegeben. Die Wiederholung wird mit dem Rot-Anteil im Bereich von 0 bis ausschliesslich 260 in 5er-Schritten durchgeführt.

`range(start, end, schritt)` ergibt eine Liste der Zahlen von `start` bis vor `end` mit Schrittweite `schritt`.

`range(1, 10, 3)` ergibt also die Liste `[1, 4, 7]`.

Lässt man den dritten Parameter weg, ist die Schrittweite 1

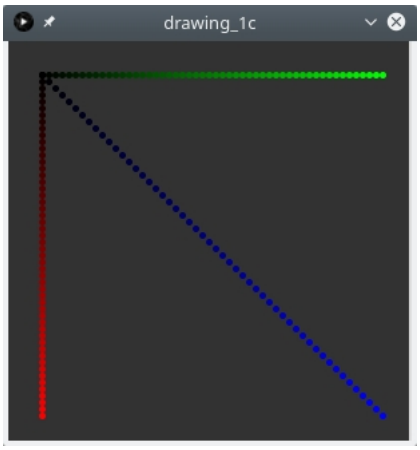
`range(1, 5)` ergibt also die Liste `[1, 2, 3, 4]`

Verwendet man nur einen Parameter, so wird dieser als Endwert interpretiert und der Startwert 0 verwendet.

`range(6)` ergibt also die Liste `[0, 1, 2, 3, 4, 5]`

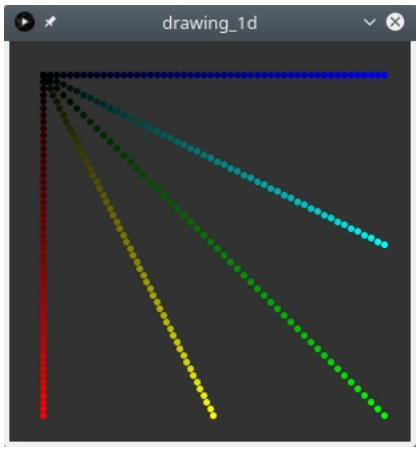
Arbeitsauftrag Erweitere das letzte Programm so, dass auch Punkte mit zunehmenden Grün- und Blau-Anteilen horizontal, bzw. diagonal gezeichnet werden.

`drawing_1c`



Arbeitsauftrag Füge noch Mischfarben dazwischen ein.

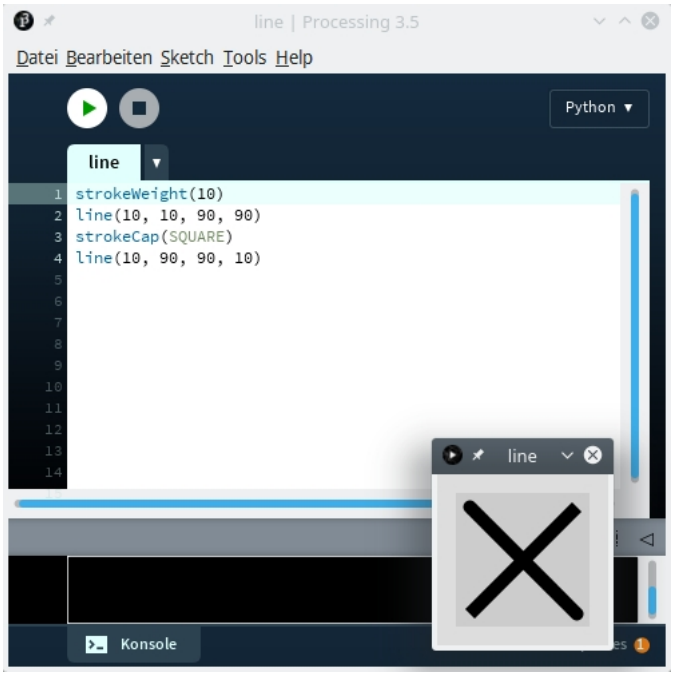
`drawing_1d`



Linien und Rechtecke

Gerade Linien ziehen
Rechtecke zeichnen
Wiederholungen wiederholen

Eines der wichtigsten Graphikelemente ist die gerade Linie (Strecke) zwischen zwei Punkten.



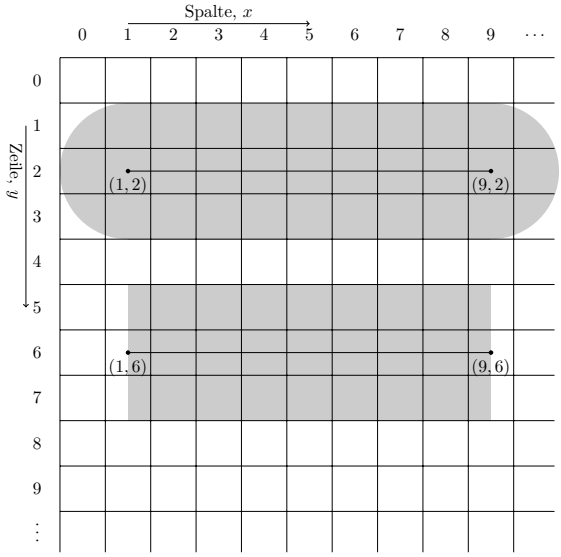
Zu beachten ist dabei, dass es für das Zeichnen der Linienende mehrere Möglichkeiten gibt. Standardmässig werden sie als Halbkreise gezeichnet.

```
line(10, 10, 90, 90)
```

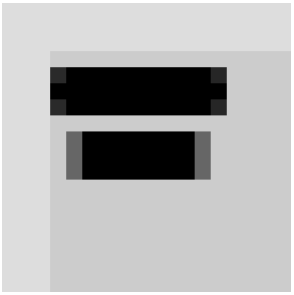
Zeichnet eine Line vom Punkt (10, 10) zum Punkt (90, 90) mit abgerundeten Ecken.

```
strokeCap(SQUARE)
line(10, 90, 90, 10)
```

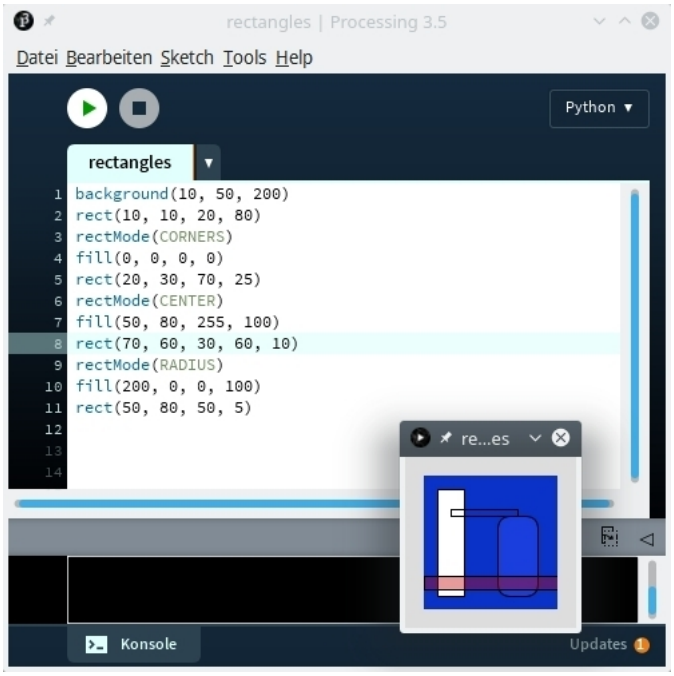
Die zweite Linie wird mit rechtwinklig abgegrenzten Enden gezeichnet.
Die beiden Möglichkeiten im Pixelraster dargestellt sehen so aus:



Wiederum lohnt es sich die Pixel vergrössert anzusehen.



Ein weiteres graphisches Grundelement sind Rechtecke, deren Seiten parallel zu den Spalten und Zeilen des Pixelrasters sind.




```
rect(10, 10, 20, 80)
```

Zeichnet ein Rechteck mit linker oberer Ecke beim Pixel $(10, 10)$, Breite von 20 und Höhe von 80 Pixel.

```
fill(0, 0, 0, 0)
```

Stellt eine vollständig transparente Füllfarbe ein.

```
rectMode(CORNERS)
rect(20, 30, 70, 25)
```

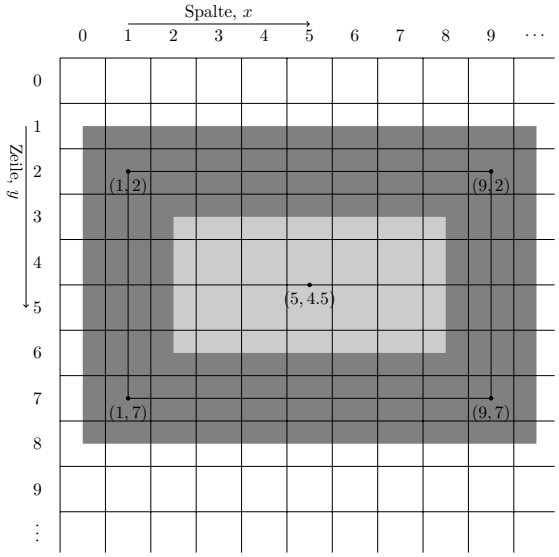
Wählt den Rechtecks-Modus `CORNERS` und zeichnet ein Rechteck mit einer Ecke beim Pixel $(20, 30)$ und diagonal entgegengesetzter Ecke bei $(70, 20)$.

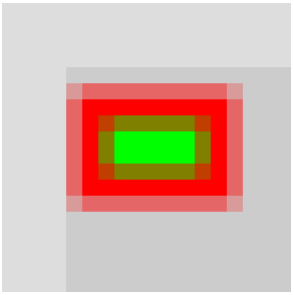
```
rectMode(CENTER)
rect(70, 60, 30, 60, 10)
```

Wählt den Rechtecks-Modus `CENTER` und zeichnet ein Rechteck mit dem Mittelpunkt beim Pixel $(70, 60)$ sowie einer Breite von 30 und Höhe von 60 Pixeln. Der fünfte Parameter gibt den Radius für die gekrümmten Ecken an.

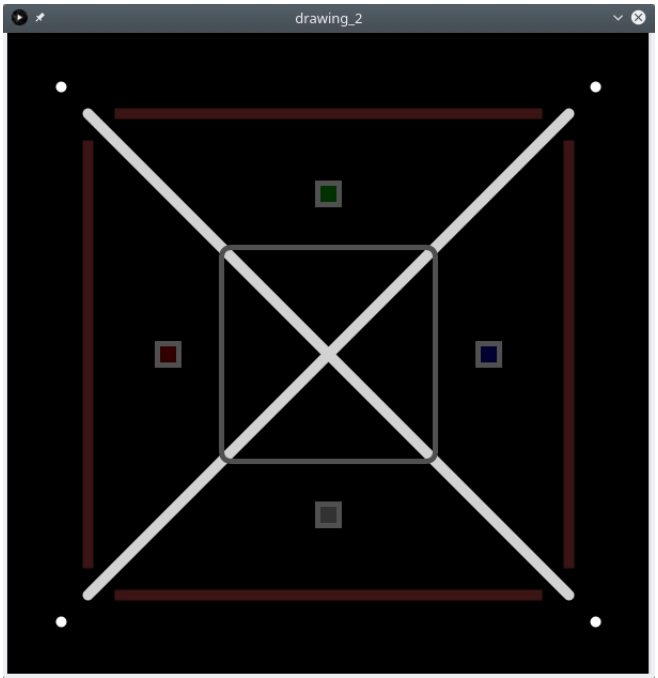
```
rectMode(RADIUS)
rect(50, 80, 50, 5)
```

Wählt den Rechtecks-Modus `RADIUS` und zeichnet ein Rechteck mit dem Mittelpunkt beim Pixel $(50, 80)$ sowie der halben Breite von 50 und halben Höhe von 5 Pixeln.



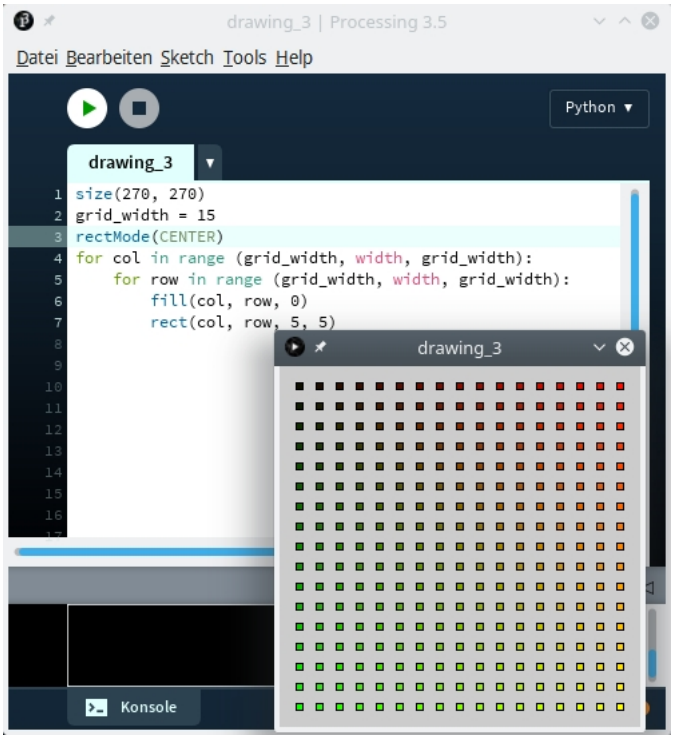


Arbeitsauftrag Erstelle ein Processing.py-Programm, das die folgende Graphik nachbaut `drawing_2`



Arbeitsauftrag Erstelle ein Processing.py-Programm, das eine Peace-Fahne zeichnet. `peace`

Wollen wir eine Graphik aufbauen, die sich in zwei Richtungen wiederholt, müssen wir zwei **for**-Anweisungen ineinander verschachteln.




```
for col in range (grid_width, width, grid_width):
```

Jeder Durchlauf dieser Wiederholung erstellt eine Spalte.

```
    for row in range (grid_width, width, grid_width):
```

In der ersten Wiederholung enthalten ist eine zweite Wiederholung. Jeder Durchlauf davon zeichnet ein Rechteck in der jeweiligen Zeile.

```
        fill(col, row, 0)
        rect(col, row, 5, 5)
```

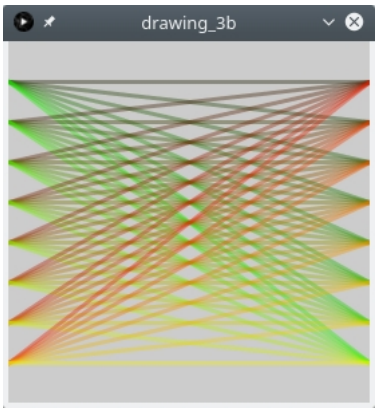
Zeichnet das Rechteck mit der Farbe, deren Rot-, bzw Grün-Anteil der Zeile und Spalte entspricht. Beachte, dass die beiden Zeilen doppelt eingerückt sind, also zu der zweiten, inneren Wiederholung gehören!

Die Einrückstiefe gibt an, zu welchem Block die Anweisung gehört.

Arbeitsauftrag

`drawing_3b`

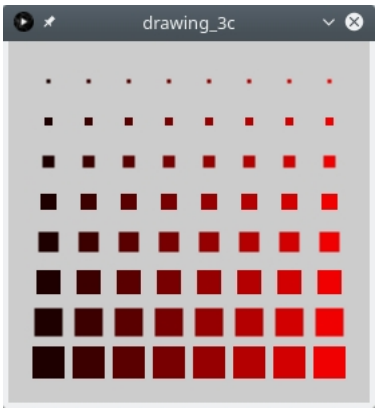
Schreibe ein Processing.py-Programm, das ein Muster aus farbigen Gitterlinien zeichnet, wie unten gezeigt. (Von jedem Punkt am linken Rand geht eine Linie zu allen Punkten am rechten Rand!)



Arbeitsauftrag

`drawing_3c`

Schreibe ein Processing.py-Programm, das ein Gitter aus Quadraten zeichnet, die in der Grösse von oben nach unten zunehmen und deren Rotanteil von links nach rechts zunimmt.



Arbeitsauftrag

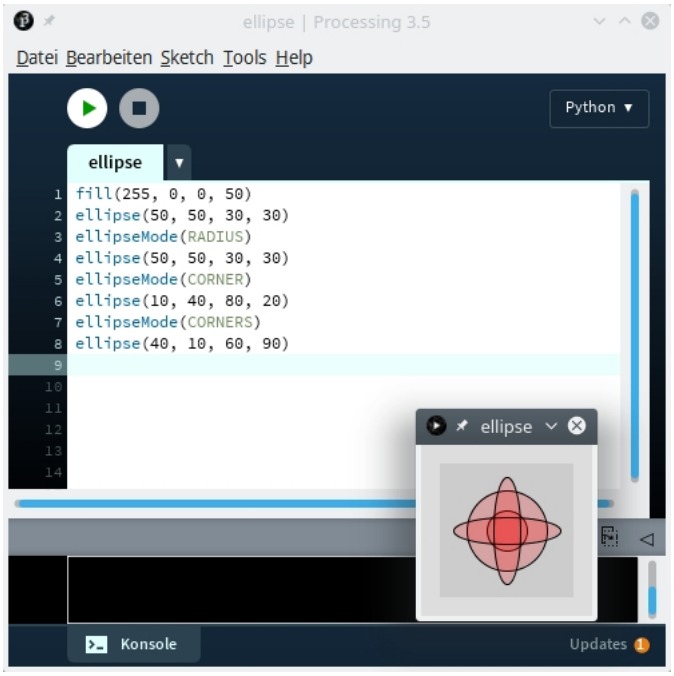
`drawing_3d`

Schreibe ein Processing.py-Programm, das ein Gitter aus Quadraten zeichnet, wie unten gezeigt. Jedes der Quadrate besteht aus übereinander gelegten Quadraten mit abnehmender Grösse und zunehmendem Blau-Anteil in der Farbe.

Kreise und Bögen

Kreise sind Ellipsen
Parametrisieren hilft Probleme zu lösen

Kreise werden in Processing.py als Ellipsen mit gleich grossen Durchmessern gezeichnet.




```
ellipse(50, 50, 30, 30)
```

Zeichnet einen Kreis als Ellipse mit Zentrum $(50, 50)$ sowie x- und y-Durchmesser je 30 Pixel.

```
ellipseMode(RADIUS)
ellipse(50, 50, 30, 30)
```

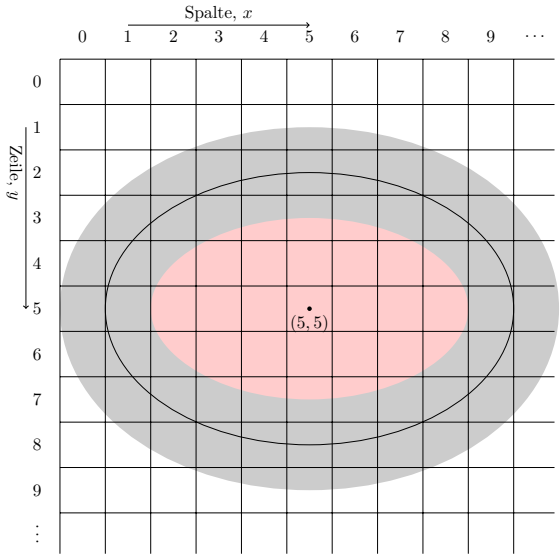
Für das Zeichnen von Ellipsen gibt es die gleichen Modus wie bei den Rechtecken. Hier wird mit dem Modue `RADIUS` ein Kreis als Ellipse mit Zentrum $(50, 50)$ sowie x- und y-Radius 30 Pixel gezeichnet.

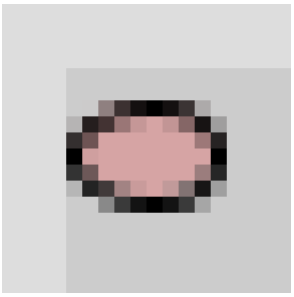
```
ellipseMode(CORNER)
ellipse(10, 40, 80, 20)
```

Es wird eine Ellipse mit oberer linker Ecke bei $(10, 40)$ sowie einer Breite von 80 und Höhe von 20 Pixeln gezeichnet.

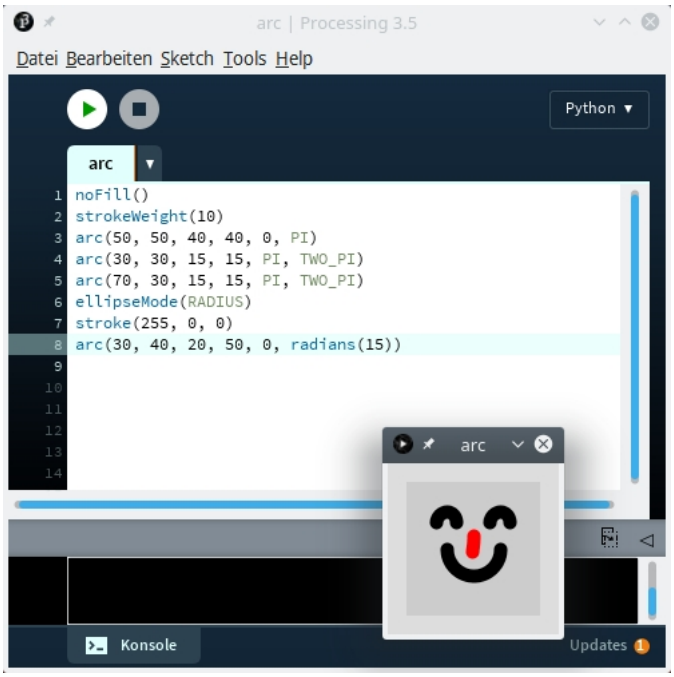
```
ellipseMode(CORNERS)
ellipse(40, 10, 60, 90)
```

Mit dem Modue `CORNERS` wird zuletzt eine Ellipse mit einer Ecke bei $(40, 10)$ und gegenüberliegender Ecke bei $(60, 90)$ gezeichnet.





Genau gleich werden Kreisbögen entlang von Ellipsen gezeichnet.




```
arc(50, 50, 40, 40, 0, PI)
```

Der `arc`-Befehl hat dementsprechend auch dieselben vier Parameter wie der `ellipse`-Befehl und interpretiert sie je nach dem eingestellten Modul für das Zeichnen von Ellipsen. Der 5. und 6. Parameter sind der Start- bzw. Stoppwinkel für den Bogen. Diese Winkel dürfen leider nicht in Graden angegeben werden, sondern im Bogenmass (*radians*).

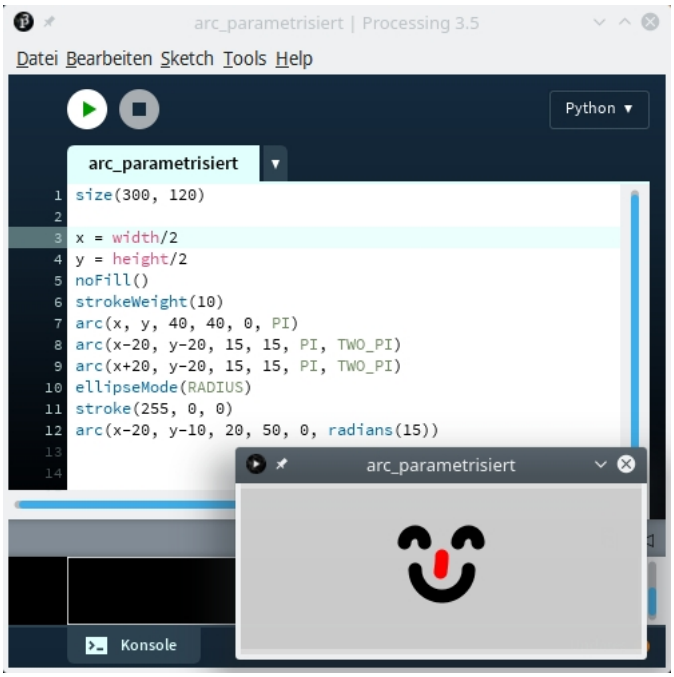
Winkel müssen in Processing.py immer im Bogenmass angegeben werden, wobei für Umrechnung $180^{\circ} = \pi$ gilt.

```
arc(30, 30, 15, 15, PI, TWO_PI)
```

Für die Winkel 45°, 90°, 180° und 360° sind in Processing.py die Konstanten `QUARTER_PY` `HALF_PY`, `PI` respektive `TWO_PI` vordefiniert.

```
arc(30, 40, 20, 50, 0, radians(15))
```

Für andere Winkel kann die Umrechnungsfunktion `radians` verwendet werden, die ihr Argument als Winkel im Gradmass interpretiert und das entsprechende Bogenmass ergibt.
Das obige Beispielprogramm, das ein Smiley-Gesicht aus Ellipsenbögen zeichnet, hat den Nachteil, dass es nur für die gewählte Grösse des Graphikfensters funktioniert. Soll das Gesicht an andere Grössen angepasst werden können, so dass es weiterhin in der Mitte des Fensters steht, müssen wir die verwendeten Graphikbefehle *parametrisieren*.




```
x = width/2
y = height/2
```

Processing.py stellt die Variablen `width` und `height` zur Verfügung, damit wir unser Programm an die Grösse des Pixelarrays anpassen können.

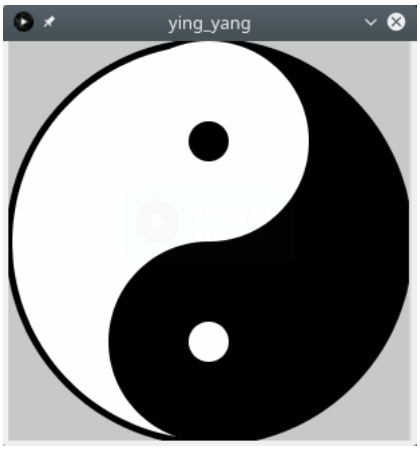
```
arc(x, y, 40, 40, 0, PI)
```

Zeichnet den Bogen in die Mitte des Fensters.

Arbeitsauftrag Führe das letzte Beispiel fort, indem du es weiter parametrisierst, um auch die Grösse des Gesichts an die Fenstergrösse anzupassen.
`arc_parametrisiert_2`

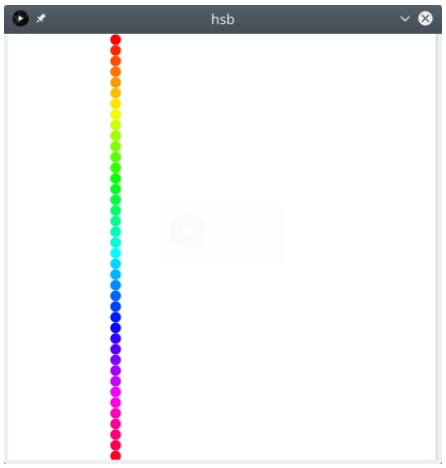
Arbeitsauftrag Zeichne mit einem Processing.py-Programm ein Ying-Yang Symbol wie unten gezeichnet.
Hinweis: Verwende dazu sich teilweise Bögen und Kreise in der richtigen Reihenfolge!

`ying_yang`



Arbeitsauftrag Erstelle mit einem Processing.py-Programm eine Reihe von Punkten in den Farben des Regenbogens, wie unten gezeigt.
Hinweis: Verwende dafür das **hsb**-Farbmodell (siehe `https://py.processing.org/tutorials/color/`)

`hsb`



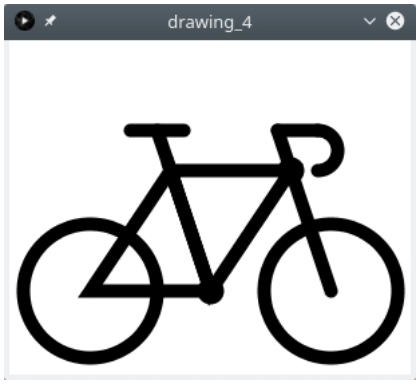
Arbeitsauftrag Zeichne mit einem Processing.py-Programm einen Regenbogen wie unten gezeigt.

`rainbow`

Arbeitsauftrag

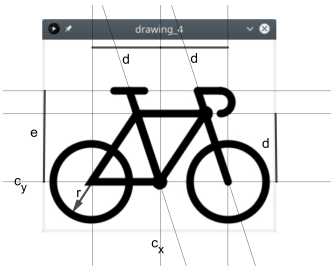
`drawing_4`

Schreibe ein Processing.py-Programm, das ein Velo-Pictogramm wie gezeigt erstellt.



Hinweis

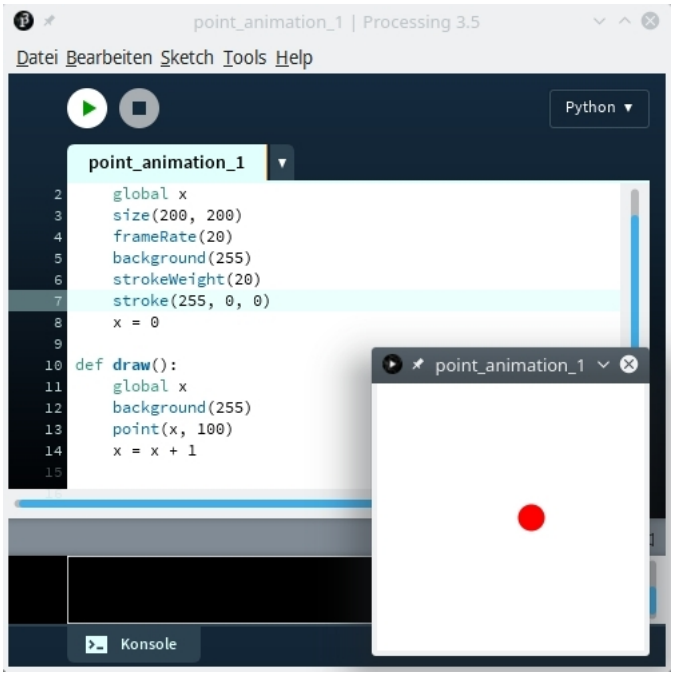
Parametrisiere das Programm ausgehend von der Position (c_x, c_y) der Tretkurbel mit geeigneten (und geeignet benannten) Variablen für die Positionen weiterer wichtiger Punkte (Sattelmitte, Radzentren, Ecken der beiden Dreiecke des Rahmens, ...).
Berechne diese aus den Abständen d und e sowie der gemeinsamen Neigung des Sattelrohrs und der Gabel in der untenstehenden Skizze.



Animation

Animation heisst Belebung
Funktionen benennen Programmteile
Mit Bedingungen kann der Programmablauf gesteuert werden
Machen wir einen Vergleich
Logik hilft weiter

Processing.py erlaubt es nicht nur, statische Graphiken darzustellen, sondern enthält auch vordefinierte Strukturen, um Graphiken zu animieren.




```
def setup():
```

Es wird ein Block von darauf folgenden, gleich weit eingerückten Anweisungen mit `setup` benannt, damit er über seinen Namen aufgerufen werden kann. Man sagt, es wird eine *Funktion definiert.*

Eine Funktion mit Namen `fun` wird mit der `def`-Anweisung definiert.
Diese besteht aus der Klausel `def fun():` gefolgt von einem Block, der die Anweisungen enthält, die beim *Aufruf* der Funktion ausgeführt werden sollen.
Das Klammerpaar nach dem Funktionsnamen ist zwingend.

Processing.py ruft die Funktion mit dem Namen `setup` zu Beginn genau einmal auf.

Eine Funktion mit Namen `fun` kann durch den Ausdruck `fun()` aufgerufen werden. Dabei werden die Anweisungen ihres Blocks der Reihenfolge nach ausgeführt.

```
    global x
```

Die Variable `x` soll *global* verwendet werden, das heisst nicht nur innerhalb der Funktion `setup`.

Variablenzuweisungen gelten nur innerhalb des Blocks, in dem sie stehen, ausser der Variablenname wird als global deklariert.

```
    frameRate(20)
```

Die Animation soll mit einer Geschwindigkeit von 20 Hertz ablaufen, d.h die unten stehende Funktion `draw` wird 20 Mal pro Sekunde aufgerufen.

```
def draw():
```

Eine zweite Funktion mit Namen `draw` wird definiert. Diese wird durch Processing.py anschliessend endlos wiederholt ausgeführt (bis das Programm abgebrochen wird).

```
    x = x + 1
```

Die globale, in der Funktion `setup` defnierte, Variable `x` wird um 1 erhöht.

Das Gleichheitszeichen in Python wird *Zuweisungsoperator* genannt und darf nicht mit dem mathematischen Gleichheitszeichen verwechselt werden!
In Python wird zuerst der Ausdruch auf der rechten Seite (`x+1`) ausgwertet und *danach* das Ergebnis der Variable auf der linken Seite zugewiesen.

Weil das Vergrössern / Verkleinern des Werts in einer Variable sehr häufig vorkommt, kann dasselbe Ergebnis kürzer mit der Anweisung

```
    x += 1
```

erzielt werden.

Die Anweisung `x += 1` ist eine Abkürzung für `x = x + 1` Analog gibt es auch die Kurzschreibweisen der Art `x -= 2`, `x *= -1`, `x /= 3`

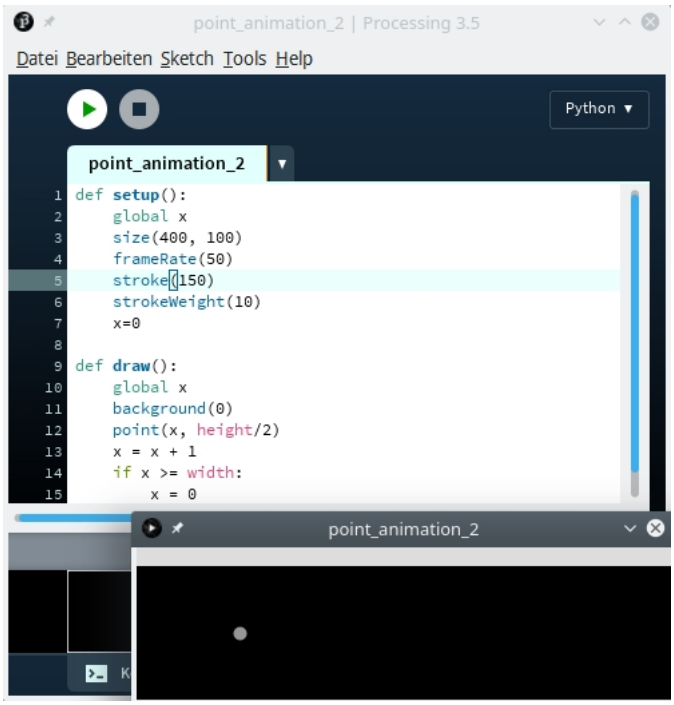
Zusammen können die beiden Funktionen `setup` und `draw` verwendet werden, um eine Animation einzurichten und danach abzuspielen.
Dies geschieht dadurch, dass zuerst `setup` ein Mal und danach `draw` (endlos) wiederholt durch das System aufgerufen und damit ausgeführt wird

Arbeitsauftrag Erweitere die obige Animation so, dass mehrere (z.B. 4) verschieden farbige Kreise auf unterschiedlichen Zeilen (gleichmässig verteilt über die Höhe des Pixelrasters) mit verschiedenen Geschwindigkeiten bewegt werden.
`point_animation_1a`

Arbeitsauftrag Baue die letzte Aufgabe so aus, dass z. B. 10 oder mehr Kreise animiert werden. Weil das Programm durch die Verwendung von vielen Variablen schnell unübersichtlich wird, lohnt es sich hier eine Liste von x-Koordinaten zu brauchen und mit einer Wiederholung zu arbeiten (**for**-Anweisung, innerhalb welcher die Farbe und Position der einzelnen Punkte berechnet wird).
`point_animation_1b`

Arbeitsauftrag Ändere die Animation so ab, dass vier Punkte auch in unterschiedlichen Richtungen (rechts nach links, oben nach unten, unten nach oben) bewegt werden.
`point_animiation_1c`

Alle bisherigen Animationen haben natürlich den Nachteil, dass sie nicht mehr besonders interessant sind, sobald die Punkte das Pixelraster verlassen haben. Sie sind nicht mehr zu sehen, wandern aber ausserhalb des Rasters immer weiter.
Wir wollen dies ändern, indem wir die Punkte jeweils wieder zurücksetzen, wenn sie das Pixelraster verlassen haben.




```
if x >= width:
    x = 0
```

Falls der Wert in der Variable `x` grösser oder gleich der Breite des Pixelrasters ist, setze den Wert wieder auf 0. Dadurch wird der Kreis, wenn er den rechten Rand des Pixelrasters erreicht hat, wieder zurück an dessen linken Rand gesetzt.

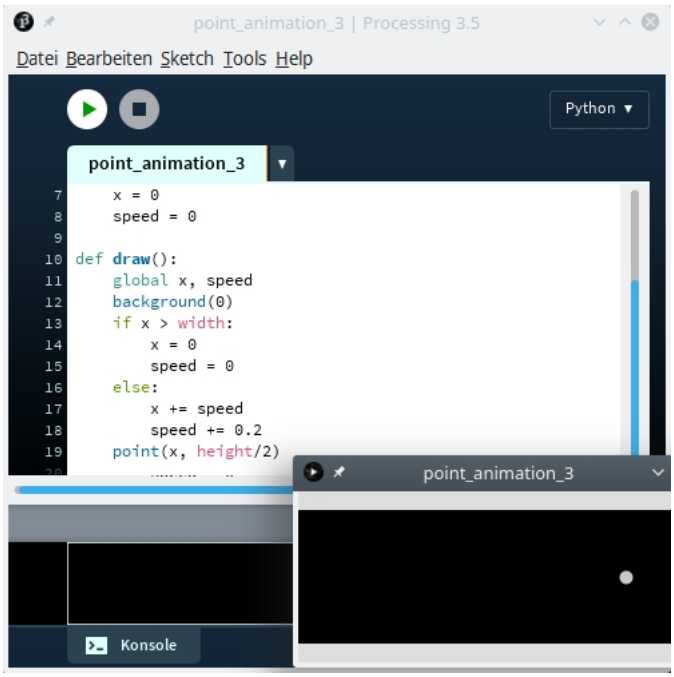
Die `if`-Anweisung besteht aus einer Klausel `if <bedingung>:` gefolgt von einem eingerückten Block von Anweisungen.
Dieser Block wird nur dann durchgeführt, wenn die `<bedingung>` zutrifft (*wahr ist*).

Bedingungen können wir zunächst durch den Vergleich zweier Werte formulieren.

Processing.py kennt 6 Vergleichsoperatoren: `==`, `!=`, `>`, `<`, `>=` und `<=`, welche der Reihe nach als *gleich*, *ungleich / verschieden*, *grösser als*, *kleiner als*, *grösser oder gleich wie* und *kleiner oder gleich wie* gelesen werden.

Es ist wichtig, den Unterschied zwischen der Zuweisung `x = 6` und dem Vergleich `x == 6` (mit doppeltem =) zu beachten!

Um das sich bewegende Rechteck zu beschleunigen, können wir die Zunahme der Spalte in jedem Schritt vergrössern.




```
    global x, speed
```

Es werden zwei globale Variablen verwendet, die eine (`x`) für die x-Position des Punkts und die zweite (`speed`) für dessen Geschwindigkeit (in Pixel/Schritt).

```
    speed = 0
```

Die Anfangsgeschwindigkeit ist 0 Pixel/Schritt.

```
    if x > width:
        x = 0
        speed = 0
    else:
        x += speed
        speed += 0.2
```

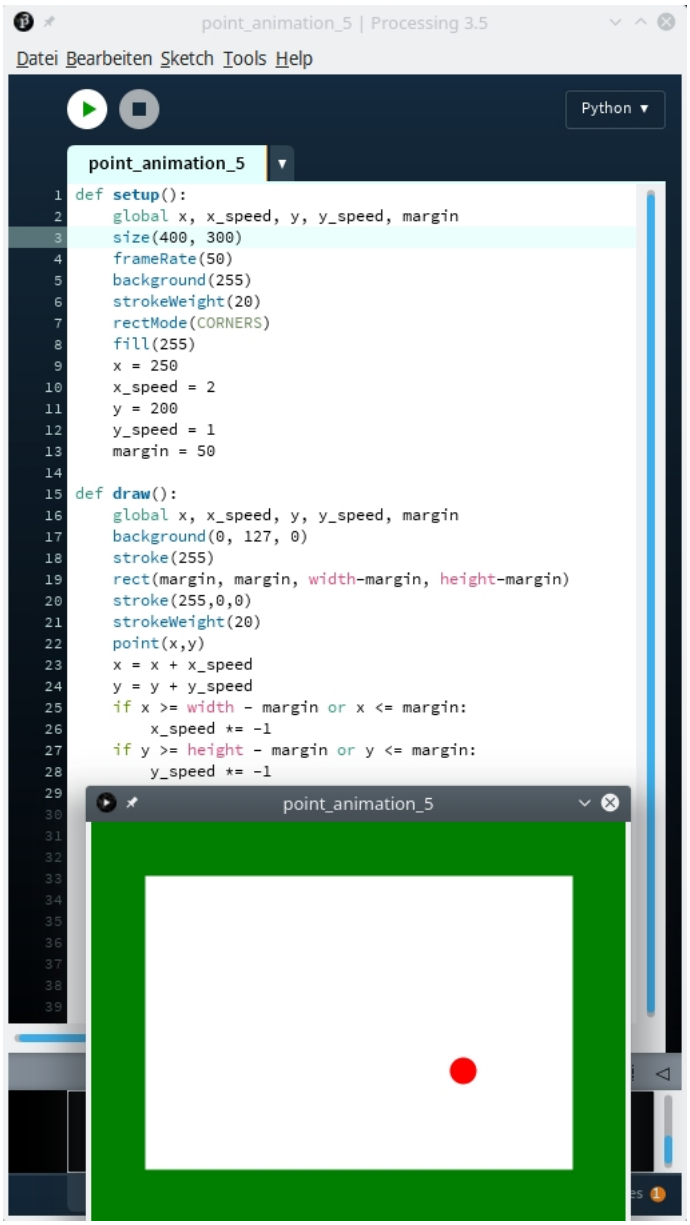
Falls die Bedingung der `if`-Anweisung *nicht* zutrifft, wird die Variable `x` um den Wert der Variable `speed` erhöht und der Wert der Variable `speed` um 0.2. Es wird also in jedem Schritt die Geschwindigkeit des Punkts vergrössert.

Mit einer optionalen `else:`-Klausel in der `if`-Anweisung kann ein weiterer Block definiert werden, der nur dann ausgeführt wird, wenn die Bedingung der `if`-Anweisung nicht zutrifft.

Arbeitsauftrag Ändere das Programm so ab, dass das Rechteck jeweils am Rand des Pixelfelds die Richtung ändert (*Hinweis:* Verwende negative Werte für `speed`, um die Richtung umzukehren.)

`point_animation_4`

Bewegen wir den Punkt in beiden (x- und y-) Richtungen zugleich, bewegt sich der Punkt wie ein Ball auf einem Billardtisch.




```
    if x >= width - margin or x <= margin:
        x *= -1
```

Bedingungen können mit Hilfe der *logischen Operatoren* **or** (für *oder*), **and** (für *und*) und **not** (für *nicht)* zusammengesetzt werden. Diese funktionieren weitgehend so, wie wir es umgangssprachlich gewohnt sind:

not <Bedingung> ist wahr, wenn <Bedingung> falsch ist, und umgekehrt.

<Bedingung 1> **and** <Bedingung 2> ist wahr, wenn <Bedingung 1> und <Bedingung 2> beide wahr sind, ansonsten falsch.

<Bedingung 1> **or** <Bedingung 2> ist wahr, wenn entweder <Bedingung 1> oder <Bedingung 2> oder beide wahr sind, ansonsten falsch (also wenn keine davon wahr ist).
Man spricht von einem *inklusiven* Oder im Gegensatz zum *exklusiven* Entweder-Oder.

Arbeitsauftrag Baue die letzte Simulation so aus, dass in y-Richtung zusätzlich eine Schwerkraft wirkt.

Hinweis: Verwende eine zusätzliche Variable `g` für *Gravitation*, deren Wert in jedem Simulationsschritt zur Geschwindigkeit in y-Richtung addiert wird. Experimentiere mit den Werten für `g`, um ein möglichst realitätsnahes Verhalten zu erzielen.

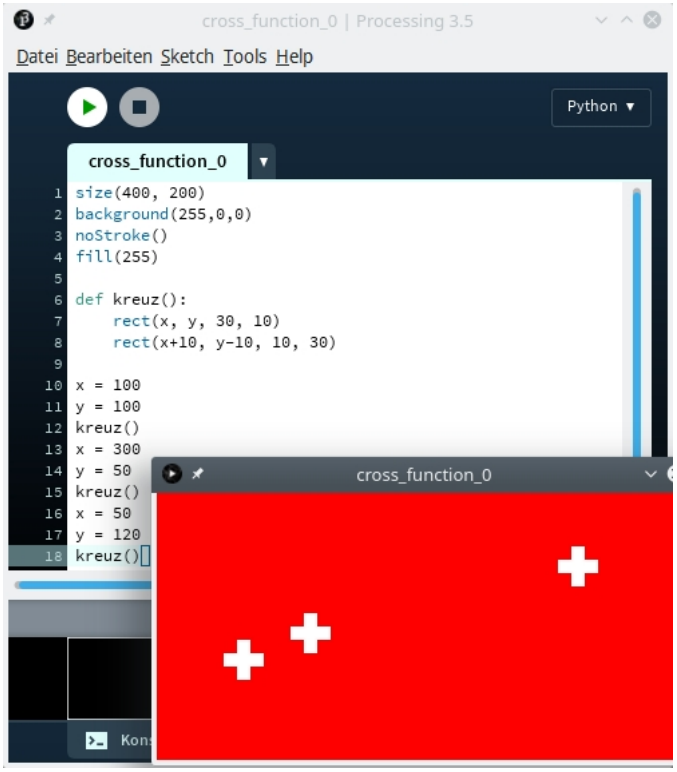
Hinweis: Beachte zudem, dass beim Aufprall am unteren Rand (Boden) ein Bruchteil der Energie absorbiert wird, so dass die Geschwindigkeit in die entgegengesetzte Richtung (aufwärts) nur ein Bruchteil der aktutellen Geschwindigkeit beträgt.

`point_animation_6`

Funktionen

Stay DRY
Funktionen definieren und aufrufen
Parameter in Funktionen verwenden
Wiederholen, wiederholen, wiederholen...

Das DRY-Prinzip fordert, dass wir nie denselben Programm-Code mehrfach hinschreiben. Damit wir dies einhalten können, haben wir die Möglichkeit neben den beiden Funktionen `setup` und `draw`, die einen speziellen Zweck für die Programmierung von Animationen haben, weitere Funktionen nach Bedarf einzuführen.




```
def swiss_cross():
    rect(x, y, 30, 10)
    rect(x+10, y-10, 10, 30)
```

Definiert eine Funktion mit dem Namen `swiss_cross`, welche ein Schweizerkreuz zeichnet, dessen Mittelpunkt bei dem Pixel in Spalte x und Zeile y liegt.

```
x = 100
y = 100
swiss_cross()
```

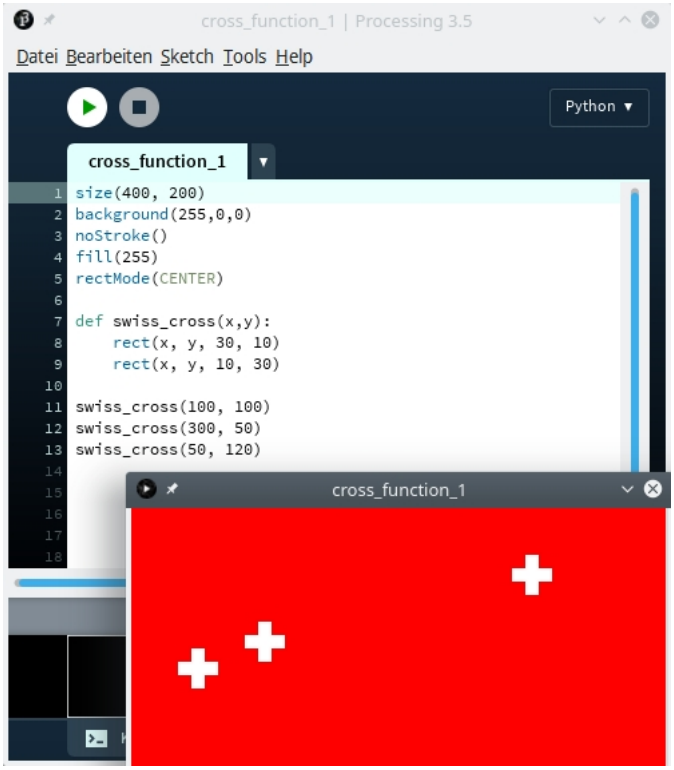
Es werden die Koordinaten des Kreuzmittelpunkts als $x = 100$ und $y = 100$ festgelegt und anschliessend die Funktion `swiss_cross` aufgerufen.

Eine Funktion wird ausgeführt, indem ihr Name gefolgt von einem Klammerpaar `()` hingeschrieben wird.
Dadurch wird an dieser Stelle im Programmablauf, der Code zur Ausführung gebracht, der in der Definition der Funktion steht.
Man sagt auch, die Funktion wird *aufgerufen*.

Arbeitsauftrag Offiziell ist festgelegt: "'Das Schweizerkreuz ist ein im roten Feld aufrechtes, freistehendes weisses Kreuz, dessen unter sich gleiche Arme je einen Sechstel länger als breit sind."' (Wappenschutzgesetz, Artikel 1)
Verbessere das letzte Beispiel so, dass diese Vorgabe erfüllt wird.

`swiss_cross_0a`

Die Art, wie der Mittelpunkt des Kreuzes über zwei globale Variablen festgelegt wird, ist unübersichtlich, umständlich (drei Zeilen!) und schwer nachzuvollziehen. Processing.py erlaubt es daher, die Funktion mit *Parametern* auszustatten.




```
def swiss_cross(x, y):
    rect(x, y, 30, 10)
    rect(x, y, 10, 30)
```

Es wird wiederum eine Funktion `swiss_cross` definiert mit dem Unterschied, dass sie diesmal zwei Parameter `x` und `y` für den Mittelpunkt verwendet, an Stelle von globalen Variablen.

```
swiss_cross(50, 120)
```

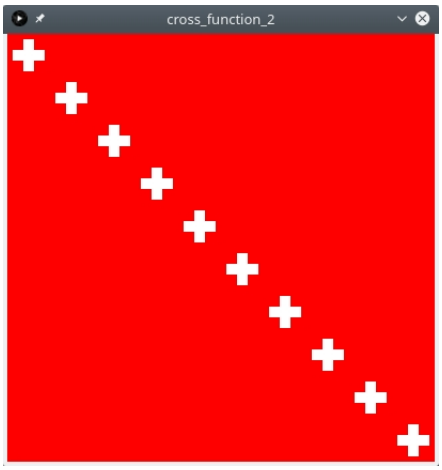
Die Funktion `swiss_cross` wird aufgerufen, wobei der erste Parameter (`x`) den Wert 50 und der zweite (`y`) den Wert 120 erhält.

> Beim Aufruf einer Funktion mit Parametern müssen deren Werte angegeben werden.
> Dies geschieht, indem man die Werte in der Reihenfolge in runde Klammern schreibt, wie die Parameter definiert sind.

Damit wird auch die Herkunft runden Klammerpaars verständlich, das beim Aufruf jeder Funktion auch ohne Parameter stehen muss.

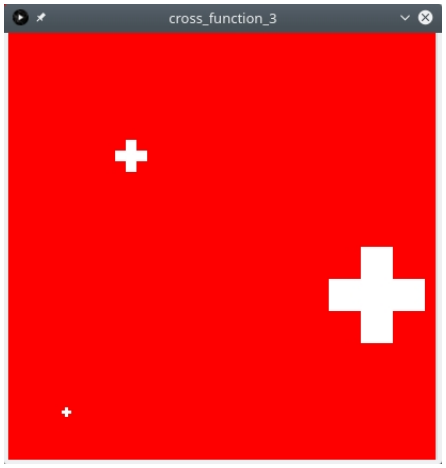
Arbeitsauftrag Schreibe ein Processing.py-Programm, dass eine diagonale Reihe von Schweizerkreuzen zeichnet.

`cross_function_2`



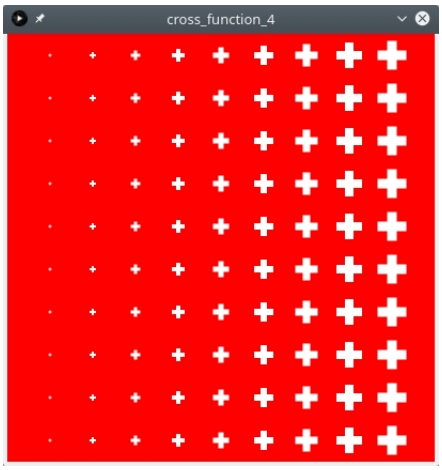
Arbeitsauftrag Ändere die Funktion `swiss_cross` so ab, dass sie einen zusätzlichen, dritten Parameter `s` enthält, der die Grösse des Kreuzes bestimmt (d. h. die Breite seiner Arme).

`cross_function_3`



Arbeitsauftrag Verwende die Version der Funktion `swiss_cross` aus der letzten Aufgabe, um ein Gitter von Kreuzen zu zeichnen, deren Grösse von links nach rechts jeweils zunimmt.

`cross_function_4`



Arbeitsauftrag Verwende dieselbe Version der Funktion `swiss_cross` für eine Animation eines Kreuzes, das in der Mitte des Pixelfeldes steht und dessen Grösse kontinuierlich zu- und abnimmt.

`cross_function_5`