



LESEPROBE · INF.SCHOOL · EINSATZ IM UNTERRICHT

Programmieren

Syntax und Grundbefehle · Funktionen und Parameter · Schleifen ·
Bedingte Anweisungen · Variablen und Datentypen

FÜR DEN INFORMATIKUNTERRICHT AM GYMNASIUM

INF  school

Inhaltsverzeichnis

1	Erste Schritte	5
	Erste Schritte mit der Schildkröte	6
	Farbe ins Spiel bringen!	11
2	Einfache Rechenoperationen	17
3	Eigene Befehle definieren	25
	Eine Funktion für viele Zwecke: Parameter	37
4	Schleifen: Befehle einfach wiederholen	45
5	Entscheidungen treffen	57
	Bedingte Anweisungen mit <code>if</code>	59
	Eine Alternative angeben mit <code>else</code>	61
	Mehrere Bedingungen prüfen mit <code>elif</code>	63
	Bedingungen zu neuen Bedingungen verknüpfen	66
6	Variablen und ihre Datentypen	69
	Ein Zuhause für Werte	70
	Datentypen	75
	Variablen als Zwischenspeicher	80
	Datentypen umwandeln	81
	Benutzerinput einlesen	83
	Eine spezielle Variable: Die Schleifenvariable	85

Über dieses Skript

Programmieren ist wohl **der** Bereich, an den die meisten denken, wenn sie das Wort Informatik hören. Viele Leute verstehen unter Programmieren entweder das Entwickeln von Spielen oder sehen vor ihrem geistigen Auge einen Hollywood-Hacker, der im Kapuzenpullover vor einem Bildschirm sitzt und wild auf der Tastatur herumhämmt.

Die Realität sieht allerdings etwas anders aus. Das Programmieren eines der wichtigsten Werkzeuge eines Informatikers. Es bietet die Möglichkeit, theoretische Konzepte in die Praxis umzusetzen. Dieses Buch wird dir den grundlegenden Umgang mit diesem Werkzeug anhand der Programmiersprache Python näher bringen und dir ermöglichen, deine eigenen Ideen zu verwirklichen.

Feedback

Egal ob du zu den Lehrpersonen oder zu den Lernenden gehörst: Wenn du Fehler oder Ungereimtheiten findest, Verbesserungsvorschläge hast, oder andere Ideen hast, wie man dieses Lehrmittel verbessern könnte, dann melde dich ungeniert bei mir! Ein kurzes Mail an feedback@koehlf.ch genügt. Davon lebt dieses Skript, dadurch wird es besser, und du kriegst nebenbei auch einen Eintrag in der *List of bug hunters*.

Danke!

Vielen Dank an Piet Hamann (<http://www.piethamann.de>) für die absolut genialen Illustrationen. Da bekommt der Begriff *Turtle Graphics* eine ganz neue Bedeutung!

Danke auch an alle KorrekturleserInnen und SchülerInnen, die zuverlässig alle (Tipp)Fehler aufspüren und melden, Feedback deponieren und neue Ideen einbringen. Nur dank ihnen kann dieses Skript laufend verbessert werden!

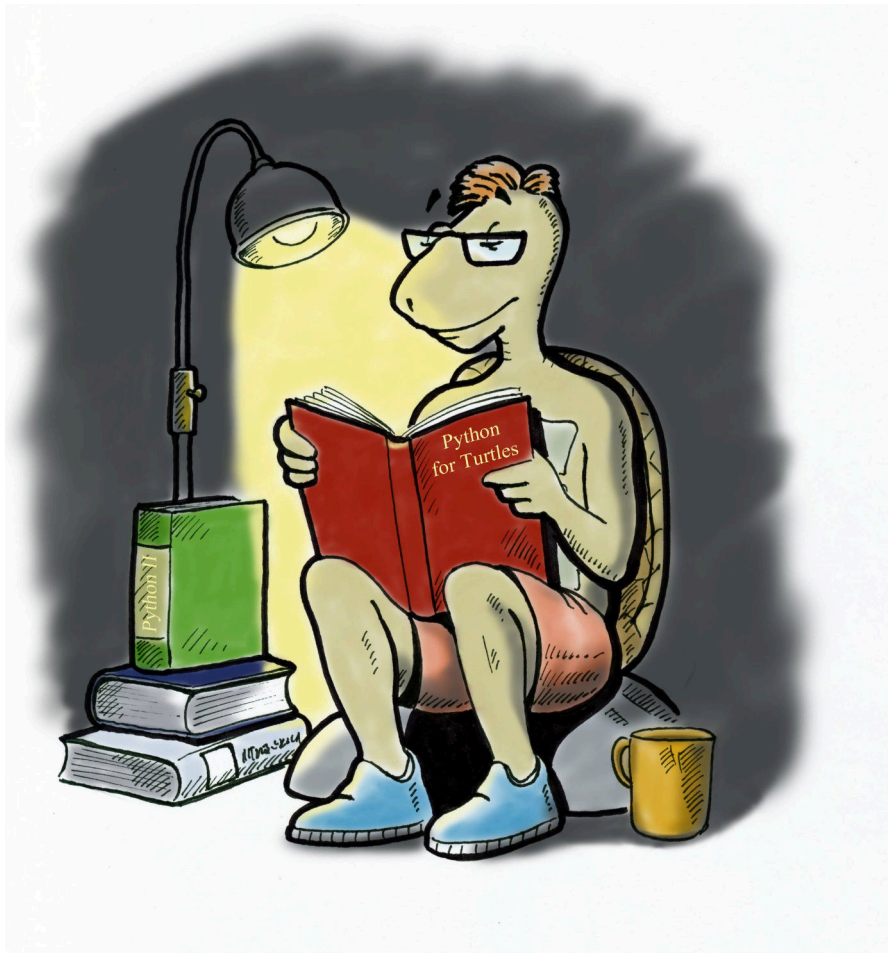
Ein grosses Dankeschön geht auch an die ganze $\text{\LaTeX}$ -Community auf StackExchange, ohne welche ich dieses Skript niemals in dieser Form hätte schreiben können.

Zu guter Letzt möchte ich mich bei meinen Eltern bedanken, die mich gross¹ gezogen haben, mich jederzeit in all meinen Zielen unterstützt haben und immer an mich geglaubt haben. Tusen takk! ♥

©2026 Florian Köhl

¹sehr gross

Erste Schritte



Erste Schritte mit der Schildkröte

Grundgerüst

Wenn wir mit der Schildkröte zeichnen möchten, müssen wir Python zuerst mitteilen, dass wir gerne mit dem Turtle Modul arbeiten würden.

Dies erreichen wir mit der ersten Zeile: `from turtle import *`

Wenn du möchtest, dass die zeichnende Turtle auch wie eine Schildkröte aussieht, kannst du auf der folgenden Zeile noch den Befehl `shape("turtle")` einfügen. Danach wartet die Schildkröte auf deine Anweisungen!

Das Grundgerüst für ein Schildkrötenprogramm sieht also wie folgt aus:

```
1 from turtle import *
2
3 # Optional: change the shape of the turtle:
4 shape("turtle")
5
6 # Your code here!
```



CODE KOMMENTIEREN



In diesem Gerüst siehst du zwei Zeilen, die mit einem `#` beginnen. Diese Zeilen nennt man **Kommentare**. Sie werden von Python beim Ausführen nicht interpretiert, können also verwendet werden, menschlichen Leserinnen und Lesern zusätzliche nützliche Informationen zu geben.

So werden beispielsweise oft Erklärungen zu bestimmten Codezeilen als Kommentare geschrieben, damit man später noch weiss, was man sich damals überlegt hat. Auch beim Lesen von 'fremdem' Code ist es durchaus hilfreich, wenn man ein paar erläuternde, erklärende Zeilen auffindet! ¹

¹Dies hilft auch deiner Lehrerin oder deinem Lehrer, deine Gedankengänge besser zu verstehen ;-)

Grundbefehle

Du hast bereits einige Befehle kennen gelernt, mit denen du die Schildkröte steuern kannst. Tabelle 1.1 gibt dir eine Übersicht:

Befehl	Beschreibung
<code>forward(d)</code> oder <code>fd(d)</code>	Bewegt die Schildkröte um d Schritte vorwärts in Blickrichtung.
<code>backward(d)</code> oder <code>bk(d)</code>	Bewegt die Schildkröte um d Schritte rückwärts in Blickrichtung.
<code>left(a)</code> oder <code>lt(a)</code>	Dreht die Schildkröte um a Grad nach links.
<code>right(a)</code> oder <code>rt(a)</code>	Dreht die Schildkröte um a Grad nach rechts.
<code>penup()</code> oder <code>pu()</code>	Hebt den Stift von der Zeichenfläche ab. Die Schildkröte bewegt sich nun, ohne zu zeichnen.
<code>pendown()</code> oder <code>pd()</code>	Setzt den Stift auf die Zeichenfläche ab. Die Schildkröte bewegt sich nun wieder malend.

Tabelle 1.1: Grundbefehle zur Steuerung der Schildkröte



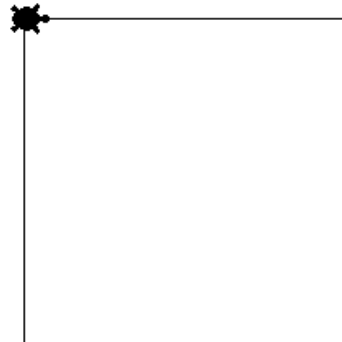
BEISPIEL: EIN EINFACHES QUADRAT

Um der Schildkröte zu sagen, dass sie ein Quadrat mit Seitenlänge 200 zeichnen soll, können wir wie folgt vorgehen:

```

1 from turtle import *
2 shape("turtle")
3
4 forward(200)
5 right(90)
6 forward(200)
7 right(90)
8 forward(200)
9 right(90)
10 forward(200)
11 right(90)

```



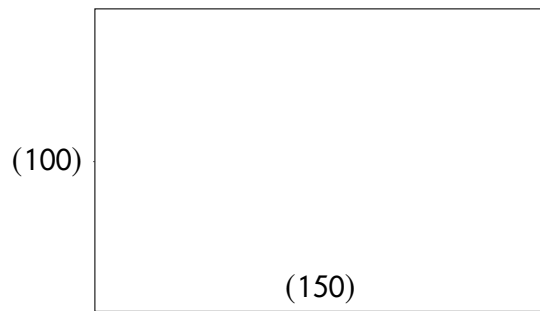
Die Schildkröte bewegt sich also als Erstes um 200 Schritte (oder Pixel) vorwärts (nach rechts), dreht sich danach um 90 Grad nach rechts, geht wiederum 200 Schritte vorwärts, dreht sich wieder 90 Grad nach rechts, und so weiter...

Hinweis: Denk daran, dass die Schildkröte immer in der Mitte des Fensters startet und initial nach rechts schaut!

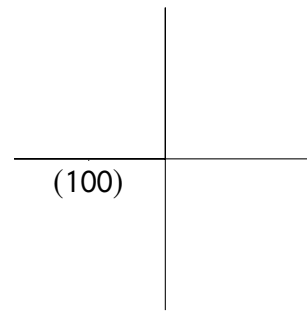
**AUFGABE 1: GEOMETRIE MIT SCHILDKRÖTEN**

In dieser Aufgabe wollen wir unsere Schildkröte ein paar einfache geometrische Figuren malen lassen.

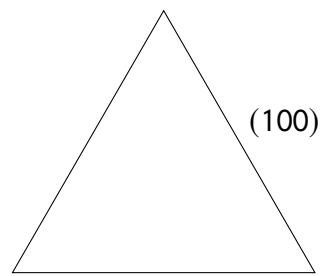
Erstelle für jede der Teilaufgaben ein eigenes kleines Programm, welches die entsprechende Grafik zeichnet.



(a)



(b)

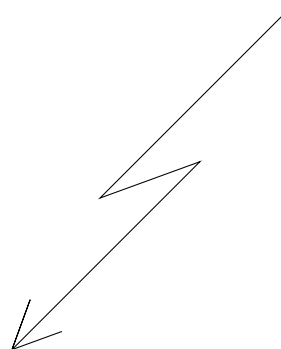


(c)

**AUFGABE 2: WIE DER BLITZ**

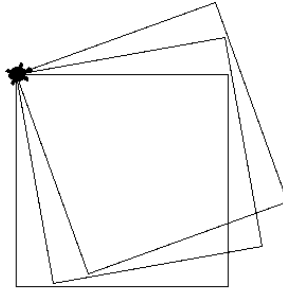
Schaffst du es, mit Hilfe der Schildkröte einen Blitz zu zeichnen?

Die Seitenlängen und Winkel kannst du selber wählen.

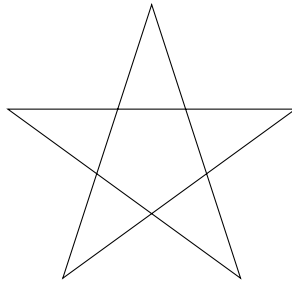


**★ AUFGABE 3: MEHRERE QUADRATE**

Zeichne drei Quadrate, wobei das zweite und dritte um 10 bzw. 20 Grad verdreht sind.

**★★ AUFGABE 4: FORTGESCHRITTENE GEOMETRIE**

Schaffst du es, mit Hilfe der Schildkröte ein Pentagramm zu zeichnen?



Fehler, Syntax und Semantik

Sicherlich sind dir beim Lösen dieser Aufgaben ein paar Fehler unterlaufen - und das ist auch gut so! Fehler helfen dir, dein geschriebenes Programm kritisch zu analysieren und zu verbessern. Wir unterscheiden grundsätzlich zwei Arten von Fehlern: Syntaktische und semantische Fehler.

SYNTAX UND SEMANTIK



Unter der **Syntax** eines Programms versteht man eine Menge von Regeln, die bestimmen, ob ein Programm wohlgeformt, also syntaktisch korrekt ist. Syntax kann man etwa mit der Grammatik in natürlichen Sprachen vergleichen: Schreibfehler, falsche Kommasetzung, ein vergessener Doppelpunkt, inkonsistente Einrückungen: All das sind syntaktische Fehler. Wenn Python über einen syntaktischen Fehler stolpert, bricht das Programm ab.

Die **Semantik** hingegen beschreibt die Bedeutung, oder den Sinn des Programms. Was soll das Programm bewirken? Falls es genau das tut, ist es semantisch korrekt. Wenn es eigentlich ein Viereck zeichnen sollte, aber ein Dreieck herauskommt, so handelt es sich um einen semantischen Fehler. Python kann diese nicht erkennen!

Wenn dein Programm plötzlich mittendrin abbricht oder gar nicht erst startet, dann handelt es sich in den meisten Fällen² um einen syntaktischen Fehler: Python versteht in diesem Fall nicht, was du genau sagen willst. Wenn du beispielsweise `foward(100)` statt `forward(100)` schreibst, wird dein Programm stoppen, sobald es diese Zeile erreicht, und dir eine entsprechende Fehlermeldung ausgeben:

```
NameError: name 'foward' is not defined
```

Wenn dein Programm allerdings fehlerfrei durchläuft, aber einfach nicht das zeichnet, was du eigentlich wolltest, so handelt es sich um einen semantischen Fehler. Nehmen wir an, jemand will ein Dreieck zeichnen und schreibt dafür folgendes Programm:

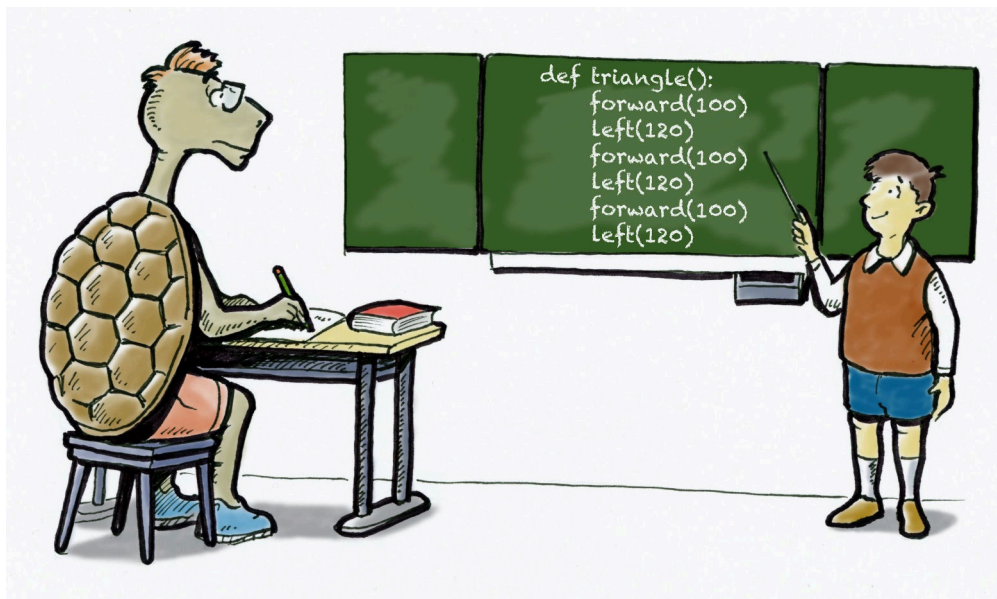
```
1 from turtle import *
2
3 forward(100)
4 left(60)
5 forward(100)
6 left(60)
7 forward(100)
8 left(60)
```



Das Programm läuft ohne Probleme durch und zeichnet etwas - es ist einfach kein Dreieck. In diesem Fall kann Python dir nicht direkt helfen... Python weiss ja nicht, was du genau zeichnen willst.

²Zumindest bei einfacheren Programmen, wie wir sie zunächst schreiben

Eigene Befehle definieren



Wir haben bis anhin einige Grundbefehle kennen gelernt, mit denen wir unsere Schildkröte auf dem Bildschirm bewegen können. Es wird Zeit, dass wir das Vokabular der Schildkröte erweitern und ihr zusätzliche, eigene Befehle und Tricks beizubringen! In diesem Abschnitt schauen wir uns also an, wie wir unsere Schildkröte 'dressieren' können.

Wie dir vielleicht aufgefallen ist, müssen wir uns bis anhin oft wiederholen. Möchten wir beispielsweise zwei Quadrate nebeneinander zeichnen, so hätten wir das bis anhin etwa so gelöst:

```
1 from turtle import *
2
3 shape("turtle")
4
5 # Draw the first square:
6 forward(100)
7 right(90)
8 forward(100)
9 right(90)
10 forward(100)
11 right(90)
12 forward(100)
13 right(90)
14
15 # Move to the next square:
16 penup()
17 forward(150)
18 pendown()
19
20 # Draw the second square:
21 forward(100)
22 right(90)
23 forward(100)
24 right(90)
25 forward(100)
26 right(90)
27 forward(100)
28 right(90)
```



Wie du siehst, wiederholen wir dabei die Anweisungen, um ein Quadrat zu zeichnen. Es ist gut möglich, dass du dies bisher einfach mit dem 'copy and paste'-Verfahren gelöst hast: Den Code einmal schreiben, und dann dorthin kopieren, wo er noch benötigt wird.

Neben der Tatsache, dass dein Quelltext dadurch unnötig lang und unübersichtlich wird, führt dies zu einem weiteren Problem: Durch das Kopieren und Einfügen von Codezeilen werden auch die darin enthaltenen Fehler dupliziert! Wenn man einen solchen dann findet und beheben möchte, muss man erst wissen, an wieviele Orte er hinkopiert wurde, um ihn überall zu korrigieren.

Wäre es nicht toll, wenn wir bestimmte Abschnitte unseres Codes irgendwie zusammenfassen und ihnen Namen geben könnten? Im obigen Codebeispiel würden wir beispielsweise sagen, dass die Zeilen 6 bis 13 ein Quadrat zeichnen. Noch besser wäre es natürlich, wenn wir Python zu einem späteren Zeitpunkt mitteilen könnten, dass wir diese paar Zeilen nochmals ausführen möchten, ohne sie Zeile für Zeile zu duplizieren.

Durch das Definieren von **Funktionen** können wir eine Folge von Befehlen zusammenfassen, ihr einen Namen geben und zu einem späteren Zeitpunkt aufrufen.

Mit dem Schlüsselwort `def` leiten wir eine Funktionsdefinition ein. Danach schreiben wir den Namen der Funktion, gefolgt von einem Paar Klammern `()` und einem Doppelpunkt. Die Anweisungen, welche zu der Funktion gehören, stehen auf den folgenden Zeilen, und zwar je um 4 Leerschläge eingerückt.

Um eine definierte Funktion anschliessend aufzurufen, schreiben wir den Namen der Funktion gefolgt von einem Klammerpaar `()`:

```
1 def my_function():
2     forward(100)
3     left(42)
4     # more lines...
5
6 my_function()
```



Die Definition der Funktion `my_function()` beginnt auf Zeile 1 und endet bei der nächsten nicht eingerückten Zeile. Der Aufruf erfolgt schliesslich auf Zeile 6.



BEISPIEL: EINEN NEUEN BEFEHL DEFINIEREN UND AUFRUFEN

Wir wollen nun unsere zwei Quadrate von oben mit Hilfe einer Funktion zeichnen! Dazu definieren wir eine Funktion `square()`, welche – du ahnst es – ein Quadrat zeichnet, und rufen diese anschliessend zweimal auf:

```
1 from turtle import *
2
3 def square():
4     forward(100)
5     right(90)
6     forward(100)
7     right(90)
8     forward(100)
9     right(90)
10    forward(100)
11    right(90)
12
13 shape("turtle")
14
15 # Draw the first square:
16 square()
17
18 # Move to the next square:
19 penup()
20 forward(150)
21 pendown()
22
23 # Draw the second square:
24 square()
```



So müssen wir uns viel weniger wiederholen und sparen erst noch Schreibarbeit. Top!

Wichtig ist, dass die Definition einer Funktion immer **vor** ihrem ersten Aufruf geschieht!

Python und die Schildkröte versuchen wirklich, all deine Befehle zu befolgen. Aber wenn die Turtle etwas ausführen soll, das sie noch gar nicht kennt, bleibt ihr nichts anderes übrig als aufzugeben und eine Fehlermeldung anzuzeigen.



BEISPIEL: DEFINITION UND AUFRUF EINER FUNKTION

In folgendem Beispiel möchten wir eine Funktion `triangle()` definieren, welche ein Dreieck zeichnet. Das Beispiel links resultiert in einer Fehlermeldung, da die Funktion zum Zeitpunkt des Aufrufs noch nicht definiert wurde. Im rechten Beispiel stimmt die Reihenfolge von Definition und Aufruf.

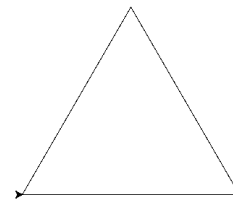
```
1 from turtle import *
2
3 triangle()
4
5 def triangle():
6     forward(100)
7     left(120)
8     forward(100)
9     left(120)
10    forward(100)
11    left(120)
```



```
1 from turtle import *
2
3 def triangle():
4     forward(100)
5     left(120)
6     forward(100)
7     left(120)
8     forward(100)
9     left(120)
10
11 triangle()
```



`NameError: name 'triangle'
is not defined`



Die Turtle meint: Wenn du mir nicht zuerst sagst, wie ein Trick funktioniert, kann ich diesen auch nicht ausführen!

BENENNEN VON FUNKTIONEN

Bei der Wahl des Namens einer Funktion hast du grundsätzlich freie Wahl. Es ist jedoch empfehlenswert, eine Funktion nach dem zu benennen, was sie macht. Solche aussagekräftigen Namen machen deine Programme viel lesbarer – auch für Andere!

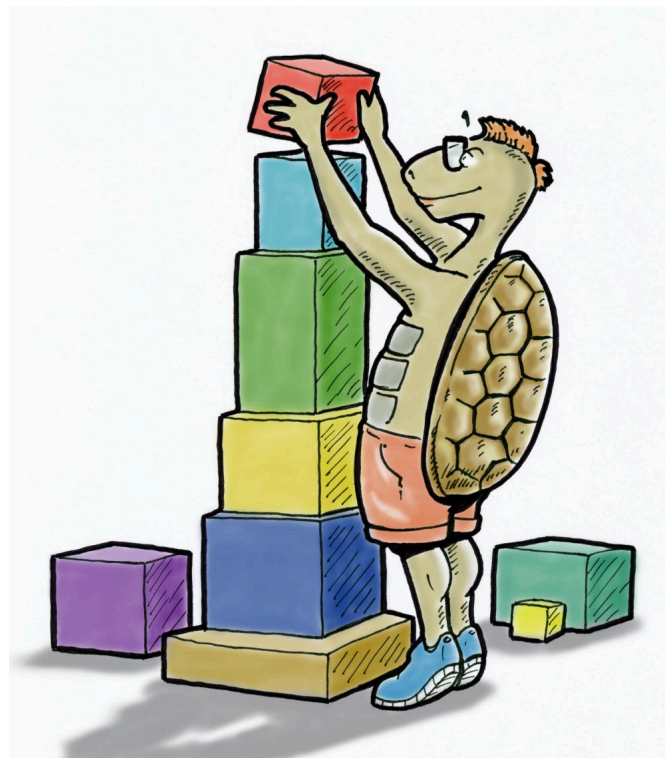
Des Weiteren solltest du folgende Regeln befolgen:

- Funktionsnamen bestehen aus Kleinbuchstaben.
- Falls der Name aus mehreren Wörtern besteht, werden diese durch Unterstriche getrennt. Beispiel: `draw_figure()`
- Kein Abstand zwischen dem Funktionsnamen und den Klammern.
- Konsistente Sprache: Schreibe alle Funktionsnamen in derselben Sprache. Am Besten auf Englisch.



Die Turtle meint: Vergiss nicht, deine Funktion nach der Definition auch aufzurufen. Sonst hab ich nämlich nichts zu tun!

Wie bereits erwähnt, erspart uns der Gebrauch von Funktionen etwas Schreibarbeit. Doch dies ist nicht der Hauptgrund, warum du Funktionen lernen und schätzen solltest. Vielmehr wollen wir Funktionen als **Bausteine** ansehen. Bausteine, die wir immer wieder verwenden können. Bausteine, die uns helfen, Probleme zu lösen. Bausteine, die wir miteinander kombinieren können, um daraus wiederum grössere Bausteine zu erzeugen, um damit noch grössere und komplexere Probleme lösen zu können.



Damit wir die Macht von Funktionen voll ausschöpfen können, ist es deshalb wichtig, Probleme in Teilprobleme aufteilen oder zerlegen zu können. Dieser Skill ist einer der grundlegendsten in der gesamten Welt der Informatik. Ach was, einer der wichtigsten Skills allgemein! Egal in welchem Bereich du dich später vertiefen wirst, du wirst immer auf komplexe Probleme oder Aufgaben stossen. Probleme, die dich auf den ersten Blick überfordern, ja unlösbar scheinen. Aufgaben, bei denen du erst nicht mal im Ansatz weisst, wo du überhaupt beginnen sollst.

Genau in solchen Situationen ist es wichtig, einen kühlen Kopf zu bewahren, einen Schritt zurück zu machen und sich zu fragen: Wie kann ich diese grosse, komplexe Anforderung in kleinere, weniger komplizierte Teilprobleme zerlegen? Kann ich diese wiederum in noch kleinere, überschaubare Teilprobleme unterteilen, bei denen ich weiss, wie ich sie lösen kann? Und wie kann ich diese Lösungen danach wieder zu einem Ganzen zusammenfügen?

Diese Lösungsstrategie, man kann es auch Denkmuster nennen, hat viele Namen. Wir wollen es hier **Modularer Entwurf** nennen.

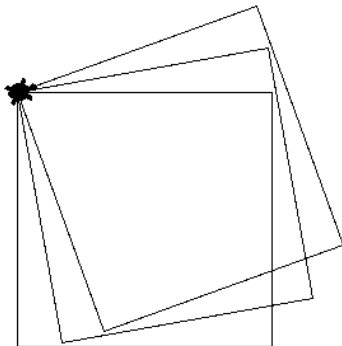
MODULARER ENTWURF

Unter dem Begriff **Modularer Entwurf** verstehen wir einerseits die Kunst, ein grosses, komplexes Problem in kleinere, überschaubare und lösbare Teilprobleme zu zerlegen. Andererseits beschreibt er auch die Fähigkeit, Lösungen für diese Teilprobleme zu einer Lösung für das ursprüngliche Problem zu kombinieren.



BEISPIEL: SCHIEFE QUADRATE

Wir wollen uns Aufgabe 3 noch einmal unter dem Aspekt des modularen Entwurfs anschauen. Zur Erinnerung: Wir sollen das Bild links zeichnen.



```
1 from turtle import *
2
3 def square():
4     forward(100)
5     right(90)
6     forward(100)
7     right(90)
8     forward(100)
9     right(90)
10    forward(100)
11    right(90)
12
13 square()
14 left(10)
15 square()
16 left(10)
17 square()
```

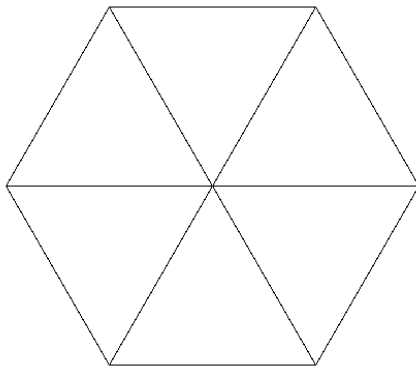
Schnell wird uns klar, dass wir hier unsere Funktion `square()` verwenden können. Wir müssen nur noch dafür sorgen, dass sich die Schildkröte nach jedem Aufruf unserer Quadratfunktion um 10° dreht!

Dieses Beispiel zeigt schön auf, wie wir dieses Problem in Teilprobleme zerlegen konnten: Quadrat und Drehung. Theoretisch hätten wir die Drehung um 10° auch direkt in unsere Quadratfunktion hinein nehmen können. Das hätte dann allerdings den Nachteil, dass wir diese Funktion nur noch dann benützen können, wenn auch eine solche Drehung erforderlich ist. Falls mehrere Quadrate einfach in gerader Linie nebeneinander gezeichnet werden sollen, würde das die Sache nur unnötig erschweren!



BEISPIEL: SECHSECK... ODER DREIECKE?

Manchmal sieht man die Teilprobleme nicht auf den ersten Blick. Handelt es sich beim Bild unten links um ein Sechseck, bei dem gewisse Ecken miteinander verbunden sind? Das wäre sicherlich ein valider Lösungsansatz. Aber wir können das Bild auch als mehrere Dreiecke auffassen! Nach jedem Dreieck dreht sich die Schildkröte um 60° , damit sie richtig ausgerichtet ist, um ein nächstes Dreieck zu zeichnen. Nach insgesamt 6 solchen Dreiecken ergibt sich das gewünschte Bild.



```
1 from turtle import *
2
3 def triangle():
4     forward(100)
5     left(120)
6     forward(100)
7     left(120)
8     forward(100)
9     left(120)
10
11 triangle()
12 left(60)
13 triangle()
14 left(60)
15 triangle()
16 left(60)
17 triangle()
18 left(60)
19 triangle()
20 left(60)
21 triangle()
22
23 hideturtle()
```



Schleifen: Befehle einfach wiederholen



Die Turtle meint:

Warum etwas 100 Mal schreiben, wenn wir es einfach 100 Mal wiederholen können?

Funktionen haben uns schon einiges an Schreibarbeit gespart. Und trotzdem haben wir viele Beispiele gesehen, bei denen wir Befehle oft wiederholen mussten, um das gewünschte Resultat zu erreichen. Geht das nicht einfacher? Kann man mit Python nicht gewisse Abfolgen von Befehlen einfach ein paar Mal wiederholen, ohne dass wir alles kopieren müssen?

In unserem Lieblingsbeispiel, dem guten alten Quadrat, schreiben wir ja eigentlich vier mal das Gleiche:

```
1 from turtle import *
2
3 forward(100)
4 right(90)
5 forward(100)
6 right(90)
7 forward(100)
8 right(90)
9 forward(100)
10 right(90)
```



Bei vier Ecken geht das ja noch. Aber stell dir jetzt mal vor, du müsstest ein Achteck zeichnen. Oder ein 20-Eck... Da kommt man ja mit zählen nicht mehr nach.

Wiederum fragen wir uns: geht das nicht einfacher? Natürlich, Python¹ hat für solche Situationen eine Lösung parat. Und diese nennt sich **Schleife**, oder auf Englisch **loop**.

Wenden wir uns zunächst wieder unserem Quadrat zu. Was wir der Turtle ja eigentlich sagen möchten ist sowas wie:

Geh 100 Schritte nach vorne und dreh dich um 90° nach rechts. Wiederhole dies viermal!

Mit Python können wir das wie folgt ausdrücken:

```
1 from turtle import *
2
3 for _ in range(4):
4     forward(100)
5     right(90)
```



Wow, nur noch 3 Zeilen, und schon haben wir ein Quadrat!? Und es kommt noch besser: Mit der gleichen Anzahl Zeilen können wir ein beliebiges n -Eck zeichnen!

Mehr dazu aber später, zuerst wollen wir uns das Konstrukt und die Syntax einer Schleife genauer anschauen.

¹wie die allermeisten anderen Programmiersprachen

SCHLEIFEN



Eine **Schleife** ist ein Sprachkonstrukt einer Programmiersprache, mit dessen Hilfe du einen oder mehrere Befehle wiederholt ausführen kannst.

Python kennt mehrere Arten von Schleifen. Wir beschränken uns in diesem Kapitel auf eine vereinfachte Version der so genannten **for-Schleife**, um eine Folge von Anweisungen mehrere Male hintereinander auszuführen. Die Syntax einer solchen Schleife in Python sieht wie folgt aus:

```
1 for _ in range(n):
2     function_a()
3     function_b()
4     # possibly more statements here...
5
6 # Loop finished here.
7 function_c()
```



Die eingerückten Zeilen nennen wir **Körper** der Schleife. Das ist der Teil des Codes, der wiederholt ausgeführt wird.

Der Wert von `n` bestimmt, wie oft der Schleifenkörper ausgeführt wird.

WIE BITTE?



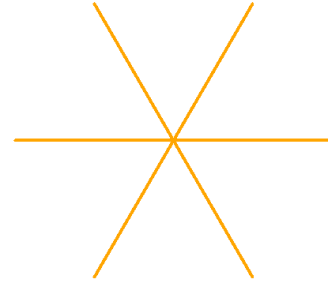
Du fragst dich jetzt vielleicht, was denn dieser Unterstrich, das Wort `in` oder dieses komische `range()` genau bedeuten. Im Moment fehlen uns leider noch ein paar Konzepte, um dies zu erklären. Aber sobald wir diese erarbeitet haben, werden wir darauf zurückkommen und dieses Mysterium auflösen. Versprochen!



BEISPIEL: STERN

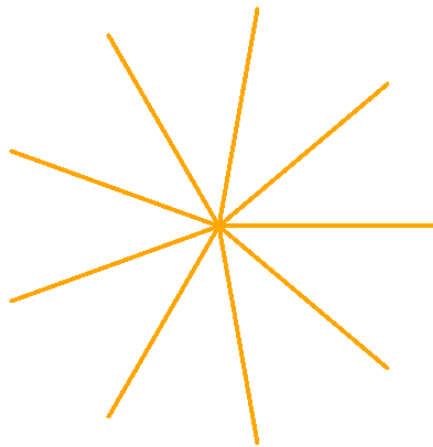
Mit Hilfe einer `for`-Schleife können wir ganz einfach eine Art "Stern" mit sechs Strahlen zeichnen. Dafür gehen wir 100 Schritte vorwärts, danach wieder rückwärts, und drehen uns um 60° nach rechts. Das Ganze wiederholen wir dann sechs mal:

```
1 from turtle import *
2
3 pencolor("orange")
4 pensize(2)
5
6 for _ in range(6):
7     forward(100)
8     backward(100)
9     right(360/6)
10
11 ht()
```



AUFGABE 29: EIN STERN MIT 9 STRAHLEN

Ändere den Code in obigem Beispiel so ab, dass er einen Stern mit 9 Strahlen zeichnet!



AUFGABE 30: EIN STERN MIT N STRAHLEN

Verpacke den Stern-Code in eine Funktion, der du als Parameter die Anzahl Strahlen sowie die Länge der Strahlen mitgeben kannst!