

# Einführung in Python

Montag

29.08.2022

15:45 - 17:30 Uhr

*Einige Folien sind Anpassungen des Foliensatzes von Wells Santo (AI4ALL),  
Laura Biester & Jule Schatz (University of Michigan)*



hessian.AI

# Erinnert ihr euch?

- Variablen
- Bedingte Anweisungen
- Schleifen

Wir schauen uns diese und viele weitere  
Konzepte in **Python** an!

**Code: Lecture-3-Python-Notebook.ipynb**

# Was ist Python?



- **Python** ist eine Programmiersprache
- Eine **Programmiersprache** legt fest, welche Aktionen wir einem Computer mitteilen können und wie genau wir dem Computer mitteilen, welche Aktion er ausführen soll

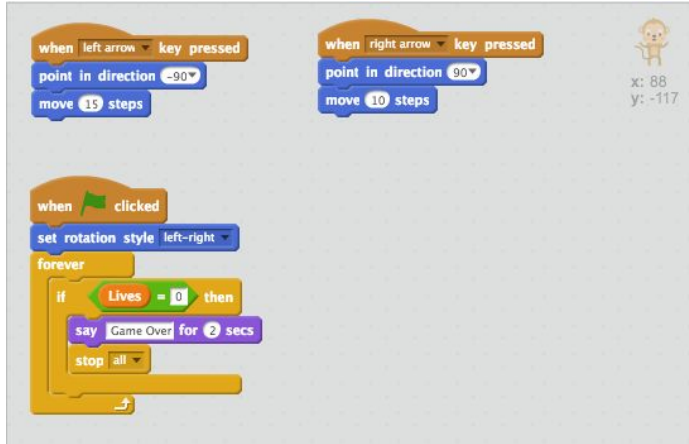


# What is Python?



- **Python** ist eine Programmiersprache
- Eine **Programmiersprache** legt fest, welche Aktionen wir einem Computer mitteilen können und wie genau wir dem Computer mitteilen, welche Aktion er ausführen soll

Ist das eine Programmiersprache?

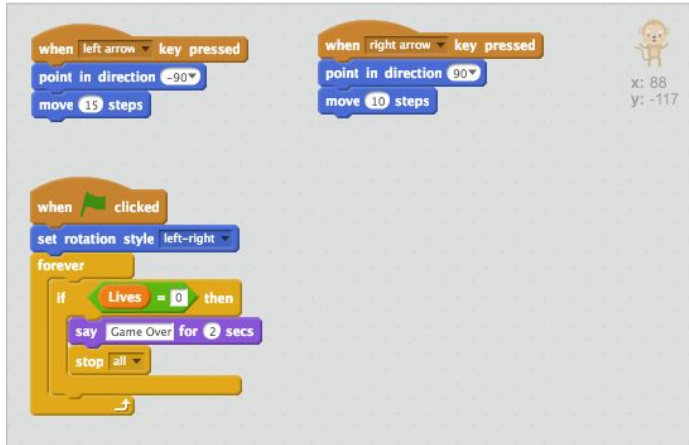


# What is Python?



- **Python** ist eine Programmiersprache
- Eine **Programmiersprache** legt fest, welche Aktionen wir einem Computer mitteilen können und wie genau wir dem Computer mitteilen, welche Aktion er ausführen soll

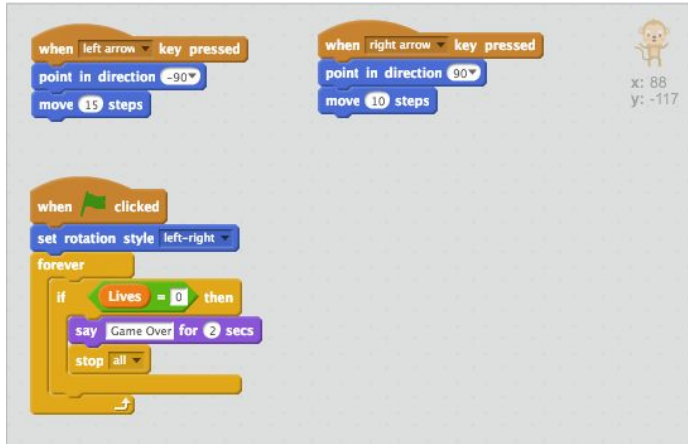
Ist das eine Programmiersprache? **JA**



# What is Python?



- **Python** ist eine Programmiersprache
- Eine **Programmiersprache** legt fest, welche Aktionen wir einem Computer mitteilen können und wie genau wir dem Computer mitteilen, welche Aktion er ausführen soll



Andere bekannte Programmiersprachen:  
JavaScript, C++, Golang, R, Java, Swift  
*Kennt ihr davon welche?*



# Coding Konzepte, die wir behandeln werden

- ❑ Variablen
- ❑ Einfache Datentypen
- ❑ Schleifen
- ❑ Bedingte Anweisungen
- ❑ Vergleichsoperatoren
- ❑ Mathematische Operationen
- ❑ Reihenfolge von Operationen

Follow along:  
`Day_1/3-Python-Notebook.ipynb`

# Coding Konzepte, die wir behandeln werden

- ✓ Variablen
  - ✓ Einfache Datentypen
  - ☐ Schleifen
  - ☐ Bedingte Anweisungen
  - ☐ Vergleichsoperatoren
  - ☐ Mathematische Operationen
  - ☐ Reihenfolge von Operationen
- Diese kennt ihr bereits in Python!



# Coding Konzepte, die wir behandeln werden

- ✓ Variablen
  - ✓ Einfache Datentypen
  - ↪ Schleifen
  - ↪ Bedingte Anweisungen
  - ❑ Vergleichsoperatoren
  - ❑ Mathematische Operationen
  - ❑ Reihenfolge von Operationen
- Diese kennt ihr bereits in Python!
- Ihr versteht dieses Konzepte und habt sie bereits in Scratch angewendet!

# Coding Konzepte, die wir behandeln werden

- ✓ Variablen
  - ✓ Einfache Datentypen
  - ↪ Schleifen
  - ↪ Bedingte Anweisungen
  - 💡 Vergleichsoperatoren
  - 💡 Mathematische Operationen
  - 💡 Reihenfolge von Operationen
- Diese kennt ihr bereits in Python!
- Ihr versteht dieses Konzepte und habt sie bereits in Scratch angewendet!
- Ihr kennt diese Konzepte noch nicht, seid aber wahrscheinlich trotzdem schon Experten ;)

# Variablen (refresh)

- Genau wie in der Mathematik können wir Variablen verwenden, um Werte darzustellen.
- Diese Werte können überschrieben werden.

```
x = 10
y = 2

x = x * 2

print(x)

x = "Hello"

print(x)
```

✓ 0.2s Python

# Datentypen

## Zahlen

*Integers:*

x = 10

y = 2

*Floats:*

x = 15.5

y = 3.14

## Zeichen

*Characters:*

x = 'b'

y = 'a'

*Zeichenketten = Strings:*

my\_string = "Hello, World!"

new\_string = x + y

# Variablen (Aufgaben)

Schreibe in VSCode den Code für die folgenden Aufgaben. Gebe den Wert von x für jede Aufgabe mit der Funktion **print()** aus.

Aufgabe 1:

```
# Erstelle eine Variable x und setze sie auf 200
```

```
print(x) # print x!
```

Aufgabe 2:

```
# Addiere 2 zu x, um x auf 202 zu updaten
```

Aufgabe 3:

```
# Überschreiben Sie x, um den String "Hello, world!"  
darzustellen.
```

# Variablen (Lösungen)

## Aufgabe 1:

```
x = 200  
print(x)
```

## Aufgabe 2:

```
x = x + 2  
print(x)
```

## Aufgabe 3:

```
x = 'Hello, world!'  
# Or  
x = "Hello, world!"  
print(x)
```

# Benennung von Variablen: 3 Dinge die zu beachten sind

(1) Variablennamen sollten mit einem Buchstaben oder `_` beginnen

```
# Will this work?  
4ever = 100000000  
print(4ever)
```

```
# NO!  
4ever = 100000000  
print(4ever)
```

⊗ 0.3s

Python

```
File "/tmp/ipykernel_48747/1299897  
248.py", line 1  
    4ever = 100000000  
    ^  
SyntaxError: invalid syntax
```

```
# Will this work?  
for_ever = 100000000  
print(for_ever)
```

```
# YES!  
for_ever = 100000000  
print(for_ever)
```

✓ 0.2s

100000000

```
# Will this work?  
_4ever = 100000000  
print(_4ever)
```

```
# YES!  
_4ever = 100000000  
print(_4ever)
```

✓ 0.2s

100000000

# Benennung von Variablen: 3 Dinge die zu beachten sind

(2) Bei Variablen wird zwischen Groß- und Kleinschreibung unterschieden (sie sind **case-sensitive**)

```
# Was ist der Output?  
cool_Variable = 2.1  
cool_variable = "Hola :)"  
print(cool_Variable)
```



# Benennung von Variablen: 3 Dinge die zu beachten sind

## (3) Keine Leerzeichen in Variablennamen

```
# Will this work?
```

```
cool variable = "Hola :)"
```

```
# NO!!
```

```
cool variable = "Hola :)"
```



```
0.2s
```

```
Python
```

```
File "/tmp/ipykernel_48747/5887962  
87.py", line 2
```

```
    cool variable = "Hola :)"  
      ^
```

```
SyntaxError: invalid syntax
```

# Check-in!

Was sind die Datentypen der Variablen var1, var2 und var3?

```
var1 = "Game Over : ("
var2 = 200
var3 = 20.0
```



# Check-in!

Was sind die Datentypen der Variablen var1, var2 und var3?

```
var1 = "Game Over :(" # string  
var2 = 200 # integer  
var3 = 20.0 # float
```

```
# Try this:  
var1 = "Game Over :("  
type(var1)
```



# Mathe

Mathe ist so einfach mit Python

+

## Addition

```
# x = 1 plus 1  
x = 1 + 1  
print(x)
```

\*

## Multiplikation

```
# x = 8 mal 8  
x = 8 * 8  
print(x)
```

-

## Subtraktion

```
# x = -15.2 minus -31  
x = -15.2 - 31  
print(x)
```

/

## Division

```
# x = 3 geteilt durch 4  
x = 3 / 4  
print(x)
```

# Mathe

Mathe ist so einfach mit Python

## Potenzierung

**\*\***

```
# x = 8 zum Quadrat  
x = 8 ** 2  
print(x)
```

**%**

## Modulo

```
# x = Rest von 10 / 3  
x = 10 % 3  
print(x)
```

# Reihenfolge von Operationen

Python berücksichtigt die Reihenfolge mathematischer Operationen.

```
# addiere 1 zu 8 Quadrat  
x = 8 * 8 + 1  
print(x)
```

Ist das gleiche wie

```
# addiere 1 zu 8 Quadrat  
  
x = 8 ** 2 + 1  
print(x)
```

```
# Was kommt hier raus?  
x = 8 ** (2 + 1)  
print(x)
```

```
# keine implizite Multiplikation mit Klammern  
x = 8(8)  
print(x)
```

# Coding Konzepte

- ✓ Variablen
- ✓ Einfache Datentypen
- ☐ Schleifen
- ☐ Bedingte Anweisungen
- ☐ Vergleichsoperatoren
- ✓ Mathematische Operationen
- ✓ Reihenfolge von Operationen

**Fast geschafft!**

# Vergleichsoperatoren

Wir verwenden Vergleichsoperatoren, um zu prüfen, ob die Bedingungen **wahr** oder **falsch** sind

Welchen Operator verwenden wir, um zu prüfen, ob zwei Werte gleich sind?

**Zwei Gleichheitszeichen ==**



## Neuer Datentyp! **Boolean**

Eine boolesche Variable kann nur zwei Optionen darstellen: Wahr oder Falsch

```
# Boolean?  
what_is_this = True  
type(what_is_this)
```

```
# What happens?  
what_is_this = True  
i_am_true = 1  
what_is_this == i_am_true
```



# Vergleichsoperatoren

Wir verwenden Vergleichsoperatoren, um zu prüfen, ob die Bedingungen **wahr** oder **falsch** sind

Welchen Operator verwenden wir, um zu prüfen, ob zwei Werte gleich sind?

**Zwei Gleichheitszeichen ==**

Falsch ist gleich 0 und Wahr kann alles außer 0 sein.



## Neuer Datentyp! **Boolean**

Eine boolesche Variable kann nur zwei Optionen darstellen: Wahr oder Falsch

```
# Boolean?  
what_is_this = True  
type(what_is_this)
```

```
# What happens?  
what_is_this = True  
i_am_true = 1  
what_is_this == i_am_true
```

# Vergleichsoperatoren

Wir verwenden Vergleichsoperatoren, um zu prüfen, ob die Bedingungen **wahr** oder **falsch** sind

== gleich

!= ungleich

< kleiner

<= kleiner gleich

> größer

>= größer gleich

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = 30  
y = 15  
print(x == y)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = 30  
y = 15  
print(x == y)
```

True or **False**

```
# Was wird ausgegeben?  
x = 30.0  
y = 15 * 2  
print(x == y)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = 30  
y = 15  
print(x == y)
```

True or False

```
# Was wird ausgegeben?  
x = 30.0  
y = 15 * 2  
print(x == y)
```

True or False

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x > 0)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = 30  
y = 15  
print(x == y)
```

True or False

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x > 0)
```

True or False

```
# Was wird ausgegeben?  
x = 30.0  
y = 15 * 2  
print(x == y)
```

True or False

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x < 0)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = 30  
y = 15  
print(x == y)
```

True or False

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x > 0)
```

True or False

```
# Was wird ausgegeben?  
x = 30.0  
y = 15 * 2  
print(x == y)
```

True or False

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x < 0)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x != 0)
```

True or False



# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x != 0)
```

True or False

```
# Was wird ausgegeben?  
i_am_true = True  
i_am_false = False  
print(i_am_true >= i_am_false)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x != 0)
```

True or False

```
# Was wird ausgegeben?  
i_am_true = True  
i_am_false = False  
print(i_am_true >= i_am_false)
```

True or False

```
# Was wird ausgegeben?  
big_number = 1000  
small_number = 2  
print(small_number <= big_number)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x != 0)
```

True or False

```
# Was wird ausgegeben?  
i_am_true = True  
i_am_false = False  
print(i_am_true >= i_am_false)
```

True or False

```
# Was wird ausgegeben?  
big_number = 1000  
small_number = 2  
print(small_number <= big_number)
```

True or False

```
# Was wird ausgegeben?  
big_number = 1000  
small_number = 2  
print(small_number * 500 + 1 <= big_number)
```

True or False

# Vergleichsoperatoren

```
# Was wird ausgegeben?  
x = (2 - 3) * 5  
print(x != 0)
```

True or False

```
# Was wird ausgegeben?  
i_am_true = True  
i_am_false = False  
print(i_am_true >= i_am_false)
```

True or False

```
# Was wird ausgegeben?  
big_number = 1000  
small_number = 2  
print(small_number <= big_number)
```

True or False

```
# Was wird ausgegeben?  
big_number = 1000  
small_number = 2  
print(small_number * 500 + 1 <= big_number)
```

True or False

# Bedingte Anweisungen (review)

Code, der nur ausgeführt wird,  
wenn eine bestimmte Bedingung  
erfüllt ist.

## Im Sprachgebrauch:

“Wenn Äpfel im Korb sind, sag mir  
bitte, wie viele Äpfel im Korb sind.”



3 Äpfel!

# Bedingte Anweisungen (Aufgabe)

## Aufgabe:

```
apples_in_basket = 3
```

```
"""
```

```
Wie können wir feststellen, ob der Korb leer ist?
```

```
Verwende einen Vergleichsoperator, um zu prüfen  
ob der Korb Äpfel enthält
```

```
"""
```

# Bedingte Anweisungen (Lösung)

**Aufgabe:**

```
apples_in_basket = 3

print(apples_in_basket != 0)

# or

print(apples_in_basket > 0)
```

# Bedingte Anweisungen (if)

**if** **Bedingung** **:**

**Einzug** **Code der ausgeführt wird, wenn Bedingung true**

Einzug: Kann ein Tabulator oder 4 Leerzeichen sein.  
Es muss lediglich konsistent sein!



# Bedingte Anweisungen (if)

`if` `apples_in_basket > 0` `:`

`Einzug` `print(apples_in_basket)`

**Aufgabe:**

```
# Versuche es!  
apples_in_basket = 3  
  
# Schreibe den Python code:
```

# Bedingte Anweisungen (if)

`if` `apples_in_basket > 0` `:`

`Einzug` `print(apples_in_basket)`

**Lösung:**

```
# Versuche es!
apples_in_basket = 3

# Schreibe den Python code:
if apples_in_basket > 0:
    print(apples_in_basket)
```

# Bedingte Anweisungen (if ... else)

Was ist, wenn wir etwas anderes tun wollen?

**if** **condition** **:**

**Einzug** **Code der ausgeführt wird, wenn Bedingung true**

**else** **:**

**Einzug** **Code der ausgeführt wird, wenn Bedingung false**

# Bedingte Anweisungen (if ... else)

```
if apples_in_basket > 0:  
    indent print(apples_in_basket)  
else:  
    indent print("No more apples :)")
```

```
# Versuche es!  
apples_in_basket = 3  
  
# Schreibe den Python code für  
"else"  
if apples_in_basket > 0:  
    print(apples_in_basket)
```

# Bedingte Anweisungen (if ... else)

if	apples_in_basket > 0	:
Einzug	print(apples_in_basket)	
else	:	
Einzug	print("No more apples :(")	

```
# Versuche es!  
apples_in_basket = 3  
  
# Schreibe den Python code für  
"else"  
if apples_in_basket > 0:  
    print(apples_in_basket)  
else:  
    print("No more apples :(")
```

## Bedingte Anweisungen (elif)

Was ist, wenn wir den Code für mehr als diese beiden Bedingungen ausführen wollen?

Bedingung 1: Wenn sich mehr als ein Apfel im Korb befindet, print "Es befinden sich x **Äpfel** im Korb".

Bedingung 2: Wenn sich ein Apfel im Korb befindet, print "ein Apfel ist im Korb".

Bedingung 3: Wenn der Korb leer ist, print "Es sind keine Äpfel im Korb".

## Bedingte Anweisungen (elif)

Was ist, wenn wir den Code für mehr als diese beiden Bedingungen ausführen wollen?

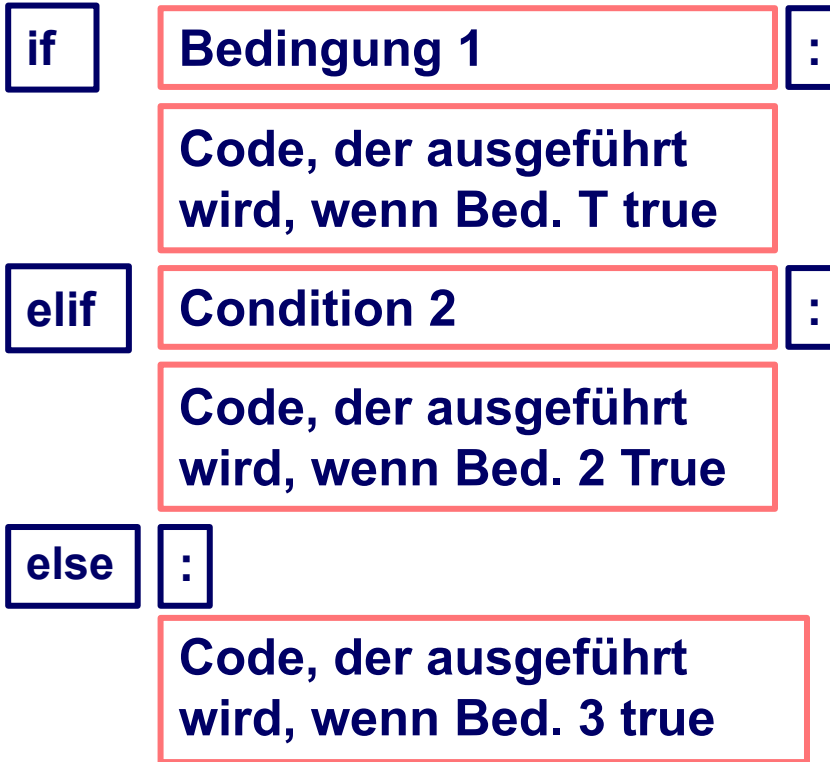
Bedingung 1: Wenn sich mehr als ein Apfel im Korb befindet, print "Es befinden sich x **Äpfel** im Korb". if

Bedingung 2: Wenn sich ein Apfel im Korb befindet, print "ein Apfel ist im Korb". elif (steht für "else if")

Bedingung 3: Wenn der Korb leer ist, print "Es sind keine Äpfel im Korb".

else

# Bedingte Anweisungen (elif)



Übung: Schreibe Python-Code auf Basis der Vorlage links, der die 3 Bedingungen des Apfelkorbs behandelt.

1. Wenn der Korb mehr als einen Apfel enthält, print "es sind x Äpfel im Korb".
2. Wenn der Korb nur einen Apfel enthält, print "es befindet sich ein Apfel im Korb".
3. Wenn der Korb leer ist, print "es sind keine Äpfel im Korb".

Teste jede Bedingung, indem Du den Wert von `apples in basket` änderst.



# Bedingte Anweisungen (elif)

Mögliche

Lösung:

```
apples_in_basket = 0

if apples_in_basket > 1:
    print("there are", apples_in_basket, "apples in the basket")
elif apples_in_basket == 1:
    print("there is one apple in the basket")
else:
    print("No more apples :(")
```

# Zusammengesetzte Bedingungen

- Wir können auch die Operatoren **und** und **oder** in unseren Bedingungen verwenden
- Was wäre die folgende Ausgabe?

```
num = 50
```

```
if num > 0 and num < 100:
```

```
    print("This is a number between 0 and 100!")
```

```
favorite_subject = "CS"
```

```
if favorite_subject == "CS" or favorite_subject == "math":
```

```
    print("You like STEM!")
```

# Zusammengesetzte Bedingungen

- Wir können auch die Operatoren **und** und **oder** in unseren Bedingungen verwenden

```
num = 50
if num > 0 and num < 100:
    print("This is a number between 0 and 100!" )
```

# Schleifen (review)

- Schleifen werden dazu verwendet wiederholt Aufgaben auszuführen.
- Schleifen verwenden bedingte Anweisungen

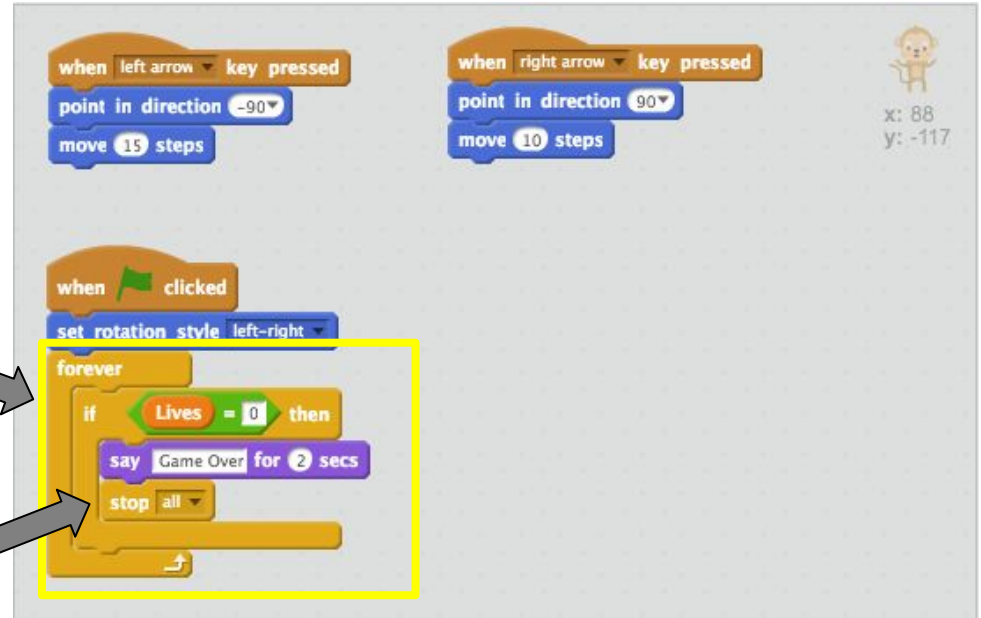


Solange der Korb mit den Äpfeln nicht leer ist, nimm einen Apfel heraus und zähle ihn.

# Schleifen (review)

Die Bedingung für die Fortsetzung der Schleife basiert auf dem Wert von *lives*

Lives = 0 ist die Bedingung für das Ende der Schleife



# Schleifen

Wir könnten versuchen, Scratch so direkt wie möglich in Python zu übersetzen:

```
Lives = 3
forever = True # make a Boolean variable
"forever" and set to True

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```



# Schleifen: Unendliche Schleifen

In diesem Code gibt es ein Problem!

```
Lives = 3
forever = True # make a Boolean variable
"forever" and set to True

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```

Unendliche Schleifen: Diese Schleife läuft *unendlich* lange, da der Wert von Lives **in diesem Bereich** nie aktualisiert wird, Lives wird nie gleich 0 sein, und **break** wird nie aufgerufen, um die Schleife zu beenden.

# Schleifen: Unendliche Schleifen

In diesem Code gibt es ein Problem!

```
Lives = 3
forever = True # make a Boolean variable
"forever" and set to True

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```

Unendliche Schleifen: Diese Schleife läuft *unendlich* lange, da der Wert von Lives **in diesem Bereich** nie aktualisiert wird, Lives wird nie gleich 0 sein, und **break** wird nie aufgerufen, um die Schleife zu beenden.

Wenn wir Schleifen schreiben, müssen wir sicherstellen, dass die Bedingung zum Beenden der Schleife **erreichbar** ist. Um die Schleife zu beenden, muss **forever** in false geändert werden, oder Lives muss in 0 geändert werden.



# Loops: Infinite Loops

There is a problem with this code

```
Lives = 3
forever = True # make a Boolean variable
"forever" and set to True

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stops the loop
```

Unendliche Schleifen: Diese Schleife läuft *unendlich* lange, da der Wert von Lives **in diesem Bereich** nie aktualisiert wird, Lives wird nie gleich 0 sein, und **break** wird nie aufgerufen, um die Schleife zu beenden.

Wenn wir Schleifen schreiben, müssen wir sicherstellen, dass die Bedingung zum Beenden der Schleife **erreichbar** ist. Um die Schleife zu beenden, muss **forever** in false geändert werden, oder Lives muss in 0 geändert werden.

Hinweis: Der Code funktioniert in Scratch, weil es einen anderen Code gibt, der den Wert von Lives aktualisiert, der zur gleichen Zeit läuft. Ihr werdet später in dieser Woche mehr über Spielschleifen lernen, aber das Wichtigste ist jetzt, dass die Abbruchbedingung erreicht werden muss!

# Schleifen: Behebung der Endlosschleife

```
Lives = 3
forever = True

while forever:
    if Lives == 0:
        print("Game Over")
        break # stop
    # update the value of Lives
    Lives = Lives - 1
```

Hier wird die Abbruchbedingung erreicht,  
da der Wert von Lives 0 erreichen wird

# Schleifen: Iterationen

```
Lives = 3
forever = True
num_iterations = 0

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stop
    # update the value of Lives
    Lives = Lives - 1
    num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

Eine **Iteration** ist ein Durchlauf durch die Schleife.

Was wird dieser Code ausgeben?

# Schleifen: Iterationen

```
Lives = 3
forever = True
num_iterations = 0

while forever:
    if Lives == 0: # stopping condition
        print("Game Over")
        break # stop
    # update the value of Lives
    Lives = Lives - 1
    num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

Eine **Iteration** ist ein Durchlauf durch die Schleife.

Was wird dieser Code ausgeben?

```
Game Over
The loop ran 3 times
```

Die Schleife lief für 3 Iterationen

# Schleifen und Verbesserung in Python

**DE: Ich denke ich spinne!**

**EN: I think I spider!**

Genau wie in der natürlichen Sprache machen direkte Übersetzungen von einer Programmiersprache in eine andere keinen Sinn oder sind nicht so schön formuliert :)



# Schleifen und Verbesserung in Python

```
Lives = 3

# Wir brauchen "forever" nicht
# forever = True
num_iterations = 0

while True: # Wir können einfach sagen "while True"
    if Lives == 0:
        print("Game Over")
        break
    Lives = Lives - 1
    num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

# Schleifen und Verbesserung in Python

```
Lives = 3
num_iterations = 0

# Anstelle von "True" können wir eine Bedingung
# verwenden
while Lives != 0:
    # Bedingung innerhalb der Schleife löschen
    # if Lives == 0:
    #     print("Game Over")
    #     break
    Lives = Lives - 1
    num_iterations = num_iterations + 1
print("The loop ran", num_iterations, "times")
```

# Schleifen (Aufgabe)

Schreibe eine while-Schleife in Python, die in jeder Iteration "einen Apfel aus dem Korb entfernt" (verringere `apples in basket` um 1 in jeder Iteration).



Lasse dein Programm die Bedingungen ausgeben, die wir zuvor geschrieben haben (die Anzahl der Äpfel ausgeben und ausgeben, wenn keine Äpfel mehr da sind).

Denke daran, dass die **Abbruchbedingung** erreicht werden muss, um eine **Endlosschleife** zu vermeiden!



# Schleifen (Aufgabe)

Was wird dieser Code ausgegeben?

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket")
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

# Schleifen (Aufgabe)

Was wird dieser Code ausgeben?

```
apples_in_basket = 3
while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket")
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1

there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

# Schleifen (Aufgabe)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

Was wird dieser Code  
ausgeben?

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

# Schleifen (Aufgabe)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

Was wird dieser Code  
ausgeben?

Wie kommt es, dass "No  
more apples :(" niemals  
ausgegeben wird?

# Schleifen (Aufgabe)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

Was wird dieser Code  
ausgegeben?

Wie kommt es, dass "No  
more apples :(" niemals  
ausgegeben wird?

Welche **zwei Möglichkeiten**  
gibt es, diesen Code so zu  
ändern, dass es ausgegeben  
wird?

# Schleifen (Aufgabe)

```
apples_in_basket = 3

while apples_in_basket != 0:
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
```

Was wird dieser Code  
ausgegeben?

Wie kommt es, dass `"No  
more apples :("` niemals  
ausgegeben wird?

Welche **zwei Möglichkeiten**  
gibt es, diesen Code so zu  
ändern, dass es  
ausgegeben wird?

# Schleifen (Aufgabe)

```
apples_in_basket = 3

while apples_in_basket >= 0: # change the stopping condition
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(")
    apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
No more apples :(
```

Was wird dieser Code  
ausgeben?

Wie kommt es, dass "No  
more apples :(" niemals  
ausgegeben wird?

Welche **zwei Möglichkeiten**  
gibt es, diesen Code so zu  
ändern, dass es  
ausgegeben wird?

1. Ändere die  
Abbruchbedingung

# Schleifen (Aufgabe)

```
apples_in_basket = 3

while apples_in_basket > -1: # change the stopping condition
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    else:
        print("No more apples :(")
        apples_in_basket = apples_in_basket - 1
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
No more apples :(
```

Was wird dieser Code  
ausgeben?

Wie kommt es, dass "No more  
apples :(" niemals  
ausgegeben wird?

Welche **zwei Möglichkeiten**  
gibt es, diesen Code so zu  
ändern, dass es ausgegeben  
wird?

1. Ändere die  
Abbruchbedingung  
(mind. 2 Möglichkeiten)



# Schleifen (Aufgabe)

```
apples_in_basket = 3

while apples_in_basket != 0: # original stopping condition
    if apples_in_basket > 1:
        print(f"there are {apples_in_basket} apples in the
basket")
    elif apples_in_basket == 1:
        print("there is one apple in the basket" )
    # remove else from loop
    apples_in_basket = apples_in_basket - 1
print("No more apples :)") # print statement outside of loop
```

```
there are 3 apples in the basket
there are 2 apples in the basket
there is one apple in the basket
No more apples :(
```

Was wird dieser Code  
ausgeben?

Wie kommt es, dass "No more  
apples :(" niemals ausgegeben  
wird?

Welche **zwei Möglichkeiten** gibt  
es, diesen Code so zu ändern,  
dass es ausgegeben wird?

1. Ändere die  
Abbruchbedingung  
(mind. 2 Möglichkeiten)
2. Verschiebe die  
print-Anweisung  
**außerhalb des Bereichs**  
der Schleife

# Coding Konzepte

- ✓ Variablen
- ✓ Einfache Datentypen
- ✓ Schleifen
- ✓ Bedingte Anweisungen
- ✓ Vergleichsoperatoren
- ✓ Mathematische Operationen
- ✓ Reihenfolge von Operationen

**Diese Konzepte hast  
Du heute alle gelernt in  
Python umzusetzen!**

# Coding Konzepte

Als nächstes!

- ✓ Variablen
- ✓ Einfache Datentypen - Neuer Datentyp: **Listen**
- ✓ Schleifen - Neuer Schleifentyp: **For-Schleifen**
- ✓ Bedingte Anweisungen
- ✓ Vergleichsoperatoren
- ✓ Mathematische Operationen
- ✓ Reihenfolge von Operationen

# For-Schleifen

For-Schleifen sind eine weitere Art von Schleife, die verwendet werden können, wenn ein bestimmter Wertebereich durchlaufen werden soll.

```
for i in range(10):  
    print(i)
```

Teste den Code auf der rechten Seite in deinem VS Code-Notebook.

- Was sind der erste und der letzte Wert?
- Wie viele Werte werden ausgegeben?

# For-Schleifen

For-Schleifen sind eine weitere Art von Schleife, die verwendet werden können, wenn ein bestimmter Wertebereich durchlaufen werden soll.

Teste den Code auf der rechten Seite in deinem VS Code-Notebook.

- Was sind der erste und der letzte Wert? **0 und 9**
- Wie viele Werte werden ausgegeben? **10**

```
for i in range(10):  
    print(i)
```

**range beinhaltet nicht den Maximalwert!**

# Teste dein Verständnis

Vervollständige die for-Schleife so, dass sie nur die geraden Zahlen im Bereich ausgibt.

```
for i in range(20):
```

Hinweis: Der **Modulo**-Operator liefert den Rest nach der Division

# For-Schleifen

Man kann den Anfangspunkt  
des Bereichs angeben

```
for i in range(1, 11):  
    print(i)
```

Man kann den zu erhöhenden  
Wert angeben

```
for i in range(1, 11, 2):  
    print(i)
```

Man kann von großen zu  
kleinen Werten iterieren

```
for i in range(11, 1, -2):  
    print(i)
```

# Listen

Listen können verwendet werden, um viele zusammengehörende Elemente an einem Ort zu speichern.

Eine Python-Liste kann wie eine Variable erstellt werden. Die Elemente müssen durch Kommas und in Klammern getrennt werden

*Einkaufsliste:*

- *Zwiebeln*
- *Tomaten*
- *Reis*
- *Suppe*

```
Einkaufsliste = ["Zwiebeln", "Tomaten", "Reis", "Suppe"]
```



# Iteration über Listen

Wir können die Liste mit einer Schleife durchlaufen

```
Einkaufsliste = ["Zwiebeln", "Tomaten", "Reis", "Suppe"]  
for item in Einkaufsliste:  
    print(item)
```

# Zugriff auf Listenelemente

Jeder Eintrag in der Liste hat eine Indexnummer

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]  
print(grocery_list[0])  
print(grocery_list[1])  
print(grocery_list[2])  
print(grocery_list[3])
```

# Zugriff auf Listenelemente

Du kannst auch von hinten auf die Listenelemente zugreifen

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]  
print(grocery_list[-1])  
print(grocery_list[-2])
```

# Hinzufügen zur Liste

Füge etwas der Liste hinzu

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]  
  
grocery_list.append("Ice cream")  
print(grocery_list)  
  
# The item is "appended" to the end of the list  
print(grocery_list[-1])
```

# Ändere Elemente in der Liste

## Ändere Werte der Liste

```
grocery_list = ["Onions", "Tomatoes", "Rice"]  
grocery_list[0] = "Ice cream"  
print(grocery_list)
```

# Listen

Listen können verschiedene Datentypen beinhalten

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
grocery_list.append(5)
grocery_list.append(5.5)
print(grocery_list)
```

# Iteration mit for-Schleifen

Wir können die Liste mit einer Schleife durchlaufen

```
grocery_list = ["Onions", "Tomatoes", "Rice"]  
for grocery_item in grocery_list:  
    print(grocery_item)
```

Wir können durch jedes Objekt was "iterable" ist iterieren.

```
for character in "Ice cream":  
    print(character)
```

Strings sind iterable!

```
for x in 497:  
    print(x)
```

```
TypeError: 'int' object is not iterable
```

# len()

Die Funktion **len()** kann die Länge des iterierbaren Objekts angeben (Anzahl der Elemente in einer Liste, Anzahl der Zeichen in einer Zeichenkette)

```
grocery_list = ["Onions", "Tomatoes", "Rice"]  
print(len(grocery_list))  
print(len("Ice cream"))
```



# min() und max()

Mit den Funktionen min() und max() können wir den kleinsten und den größten Wert in einer Liste ermitteln. Dann können wir list.index(a\_number) verwenden, um den Index des Wertes in der Liste zu erhalten.

```
some_numbers = [5, 2, 1, 5, 7, 10, 32, 3]
smallest_number = min(some_numbers) # 1
index_of_smallest_number = some_numbers.index(smallest_number) #
2
print(some_numbers[index_of_smallest_number]) # 1
```

# Verschachtelte Schleifen

Es gibt Schleifen in Schleifen

```
grocery_list = ["Onions", "Tomatoes", "Rice"]  
for i in grocery_list:  
    for j in grocery_list:  
        if i != j:  
            print(i, j)
```

# Dictionaries

- Bei Listen greift man auf die Elemente über deren Indexnummer zu.
- Für viele Dinge ist dies nicht sehr nützlich, vor allem wenn man die Indexnummer des Elements nicht kennt.
- Ein Dictionary ermöglicht den Zugriff auf Elemente anhand von **Schlüsseln (keys)**

```
my_dict = {"key": "value"}  
  
print(my_dict["key"])
```

# Dictionary Syntax

1. Inhalt mit geschweiften Klammern umgeben { }
2. Der **key** steht links von einem Doppelpunkt : und der Wert (**value**) steht nach dem Doppelpunkt

```
my_dict = {"key": "value"}
```

# Dictionary Syntax

1. Inhalt mit geschweiften Klammern umgeben { }
2. Der **key** steht links von einem Doppelpunkt : und der Wert (**value**) steht nach dem Doppelpunkt
3. Genau wie in Listen werden die Einträge im Wörterbuch durch Kommas getrennt

```
my_dict = {"key": "value", 'dictionary': 'Wörterbuch'}
```

# Dictionaries

Man kann sich Dictionaries ähnlich wie Textwörterbücher vorstellen.

```
my_dict = {"word": "a definition of that word"}  
print(my_dict["word"])  
  
en2de = {'apple': 'Apfel', 'onion': 'Zwiebel', 'dictionary':  
        'Wörterbuch'}  
print(en2de["apple"])  
print(en2de["onion"])  
print(en2de["dictionary"])
```

# Dictionaries

Keys und Values können unterschiedliche Datentypen sein

```
my_dict = {"xy-coordinates": [(2,3), (3,4), (4,5)], 'closest': (3,4)}  
print(type(my_dict["xy-coordinates"])) # <class 'list'>  
print(type(my_dict["closest"])) # <class 'tuple'>
```

```
my_dict = {3: 'three', 'three': 3}
```

# Dictionaries

Wörterbücher sind *iterable*. Durch einfaches Iterieren über das Wörterbuch werden die Zeichenketten durchlaufen.

```
groceries = {"fruit": ["apple", "orange"],  
             "vegetable": ["onion", "potato"]}  
  
for category in groceries:  
    for item in groceries[category]:  
        print(item, "is a", category)
```



# Dictionaries

- Iteriere über keys und values mit **dict.items()**
- Iteriere über keys: **dict.keys()**
- Iteriere über values: **dict.values()**

```
for category, item_list in groceries.items():  
    print(item_list, "are", category)  
  
print(groceries.keys())  
print(groceries.values())
```

# Aufgabe

Vervollständige die for-Schleife, sodass das Element in der Liste im Dictionary auf seinen Index abgebildet wird (keys = items, values = indices)

Die Funktion enumerate kann dir sagen, in welcher Iteration (Index) du dich gerade befindest. Dieser Wert wird in der Index-Variablen gespeichert, und das Listenelement wird in der Item-Variablen gespeichert.

```
grocery_list = ["Onions", "Tomatoes", "Rice", "Soup"]
item_to_index = {}

for index, item in enumerate(grocery_list):
    """ complete the for-loop """
```

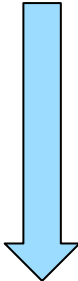
# Funktionen (Methoden)

Mithilfe von Funktionen kann Code wiederverwendet werden

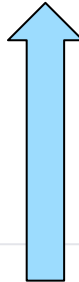
```
""" Hier definieren wir eine Funktion """  
  
def add_one(value):  
    new_value = value + 1  
    return new_value  
  
""" Hier verwenden wir die Funktion """  
print(add_one(10))  
print(add_one(2))
```

# Funktionen: Parameter

**Parameter** sind Werte, die an eine Funktion übergeben werden




```
def add_one(value):  
    new_value = value + 1  
    return new_value  
  
print(add_one(10))
```



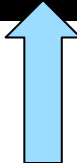
# Funktionen: Return Values

**Return values** sind Werte, die die Funktion zurückgibt

Man kann den zurückgegebenen Wert einer neuen Variablen zuweisen oder sie als Parameter an andere Funktionen übergeben, wie wir es hier mit print tun



```
def add_one(value):  
    new_value = value + 1  
    return new_value  
  
print(add_one(10))  
value_returned = add_one(10)
```



# Funktionen: Return Values

**Return values** sind Werte, die die Funktion zurückgibt

Rückgabewerte werden nicht immer benötigt

```
def my_print_function(message):  
    print("Message:", message)  
  
my_print_function("Hello!")
```

# Aufgabe

Schreibe eine Funktion **make\_list\_index**.

**Parameter:** Der Funktion wird ein Parameter, eine Liste, übergeben.

**Funktionsbody:** Die Funktion nimmt den Parameter (eine Liste) und erstellt ein Wörterbuch, das die Listenelemente auf ihre Indizes abbildet (siehe letzte Aufgabe)

**Return value:** Die Funktion gibt das Dictionary zurück

```
grocery_list = ["Onions", "Tomatoes", "Rice"]
item_to_index = {}

for index, item in enumerate(grocery_list):
    item_to_index[item] = index
```

# Lösung

```
def make_list_index(list_in):  
    item_to_index = {}  
    for index, item in enumerate(list_in):  
        item_to_index[item] = index  
    return item_to_index  
  
grocery_index = make_list_index(["apples", "bananas", "oranges"])  
print(grocery_index)  
  
materials_index = make_list_index(["clay", "wood", "stone"])  
print(materials_index)
```



# Klassen

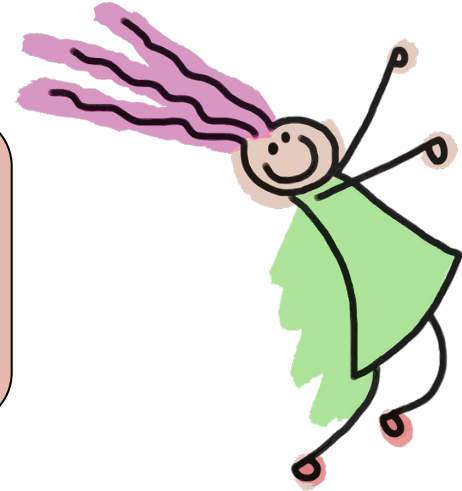
## Spielerin 1:

Name: Anna  
Geburtsjahr: 2004  
Haarfarbe: orange



## Spielerin 2:

Name: Julia  
Geburtsjahr: 2005  
Haarfarbe: lila



Wie können wir die beiden Spielerinnen in Python modellieren?

# Klassen

- Python: *objektorientierte Programmiersprache*
    - Nahezu alles besteht aus Objekten mit gewissen Eigenschaften und Funktionn/ Funktionen
  - Eine Klasse ist wie ein Bauplan für Objekte
- ★ Schreibe eine Klasse Player und erzeuge zwei verschiedene Player!

Schlüsselwort

Name der Klasse

Eine Klasse hat mind. 1 Zeile

Zwei **Instanzen** der Klasse Player

```
class Player:
    pass

player_1 = Player()
player_2 = Player()
```



Aber wo sind jetzt die Eigenschaften von Anna und Julia?

# Klassen

- In der `__init__` Funktion einer Klasse können Eigenschaften der Klasse implementiert werden (*Konstruktor*).
- Jede Klasse besitzt eine `__init__` Funktion, die immer ausgeführt wird, wenn eine Instanz der Klasse erzeugt wird.
- In der Funktion werden den Eigenschaften der Objekte Werte zugewiesen.
- Zugriff auf die Variablen möglich

Variablen der Klasse

Zuweisungen der Werte

```
class Player:
    def __init__(self, name, birth_year, hair_color):
        self.name = name
        self.birth_year = birth_year
        self.hair_color = hair_color

player_1 = Player('Anna', 2004, 'orange')
player_2 = Player('Julia', 2005, 'lila')
print(player_1.name)
print(player_2.hair_color)
```

Der `self`-Parameter ist ein Verweis auf die aktuelle Instanz der Klasse und wird für den Zugriff auf Variablen verwendet, die zur Klasse gehören.

# Klassen

- ★ Wir möchten von den Spielerinnen wissen, wie alt sie sind und welche Haarfarbe sie haben.
- Objekte können auch Funktionen enthalten. Funktionen in Objekten sind Funktionen, die zum Objekt gehören.

```
class Player:
    def __init__(self, name, birth_year, hair_color):
        self.name = name
        self.birth_year = birth_year
        self.hair_color = hair_color

    def get_hair_color(self):
        print('Meine Haarfarbe ist ' + self.hair_color)

    def get_age(self):
        age = 2022 - self.birth_year
        print('Ich bin ' + str(age) + ' Jahre alt.')

player_1 = Player('Anna', 2004, 'orange')
player_2 = Player('Julia', 2005, 'lila')
player_1.get_age()
player_1.get_hair_color()
```

Die Funktionen greifen auf Werte einer Instanz zu (Haarfarbe, Geburtsjahr). Wir übergeben hier wieder den self-parameter!

# Vererbung

- Erstelle eine Klasse Schachspieler!
- Parameter:
  - Namen
  - Geburtsjahr
  - Haarfarbe
  - Spielfarbe
- Funktionen:
  - Welche Haarfarbe
  - Welches Alter
  - Welche Spielfarbe



# Vererbung

```
class Chess_player():  
    def __init__(self, name, birth_year, hair_color, game_color):  
        self.name = name  
        self.birth_year = birth_year  
        self.hair_color = hair_color  
        self.game_color = game_color  
  
    def get_hair_color(self):  
        print('Meine Haarfarbe ist ' + self.hair_color)  
  
    def get_age(self):  
        age = 2022 - self.birth_year  
        print('Ich bin ' + str(age) + ' Jahre alt.')  
  
    def get_game_color(self):  
        print("My game color is: " + self.game_color)
```

- Ganz schön viel Schreibarbeit!
- Die meisten Variablen und Funktionen sind die gleichen wie in der Klasse Player
- Lässt sich das irgendwie vereinfachen?

# Vererbung

Erbe von der Klasse Player!

- Die Klasse Chess\_Player verfügt dann ebenfalls über alle Parameter und Funktionen, wie die Klasse Player
  - Player: Eltern-Klasse (*super class*)
  - Chess\_Player: Kind-Klasse (*sub class*)
- Das Schlüsselwort `super()` steht für „superclass“, sprich Oberklasse. Wir stellen damit eine Verbindung zwischen der Eltern-Klasse (Klasse, von der Kind erbt) und Kind-Klasse her.

# Vererbung

Übersichtlicher und kürzer!

Erbe von der Klasse Player

```
class Chess_player(Player):  
  
    def __init__(self, name, birth_year, hair_color, game_color):  
        super().__init__(name, birth_year, hair_color)  
        self.game_color = game_color  
  
    def get_game_color(self):  
        print("Meine Spielfarbe ist: " + self.game_color)  
  
chess_player_1 = Chess_player("Kim", 2003, "rot", "weiss")  
chess_player_1.get_age()  
chess_player_1.get_game_color()
```

Aufruf Super-Konstruktor  
(Konstruktor der  
Eltern-Klasse)

Verwendung von Methoden  
der Eltern-Klasse möglich