Programmiergrundkurs in Python

- Grundlagen
  - Einleitung
  - Variablen und Listen
  - Bedingungen
  - Schleifen
  - Rechnungen
  - Funktionen
- Programmieren mit Python
  - Rechnen mit Python

Grundlagen

Einleitung

Diesen Einführungskurs gibt es auch als Jupyter Notebook, für die interaktive Bearbeitung der Codebeispiele:

- Einführungskurs: Einführung.ipynb
- Praktische Übungen: Einführung_II.ipynb
- Auf dieser Seite können die Dateien geöffnet und bearbeitet werden: jupyter.org (Hinweis: Alle Funktionen im Zusammenhang mit dem `input`-Befehl werden nicht funktionieren.)

Besser ist die Installation des Jupyter Notebooks in VisualStudioCode oder Thonny. Hier stehen alle Funktionen zur Verfügung.

In der Schule funktioniert das Jupyter Notebook leider nicht. Deshalb muss man die Codebeispiele von dieser Seite direkt in Thonny kopiert werden. Thonny ist übrigens Open-Source-Software und kann daher ohne Kosten von der Seite thonny.org heruntergeladen werden. Ich empfehle, das Programm zu Hause zu installieren, um für Informatik sinnvoll üben zu können.

Linkliste externe Einführungen

Python Spacebug

Ein einfaches Programm

Ein einfaches Computerprogramm besteht üblicherweise aus Eingaben und Ausgaben. Zwischen der Eingabe und der Ausgabe gibt es meist eine Verarbeitung der Daten.

Hier ist erst einmal ein Beispiel für Ein- und Ausgabe ohne weitere Verarbeitung:

```
name = input("Bitte gib' deinen Namen ein! ")
print(f"Dein Name lautet {name}.")
```

Dieses Programm fragt mithilfe des `input`-Befehls nach deinem Namen. Dieser Name wird in der Variablen `name` gespeichert und in der nächsten Zeile mit `print` wieder ausgegeben. Die

geschwungenen Klammern teilen dem Python-Interpreter mit, dass `name` eine Variable ist. Versuche es einmal ohne die Klammern.

Mehr zu Variablen.

Bauen wir jetzt einmal eine Verarbeitung der Eingabe mit ein:

```
name = input("Bitte gib' deinen Namen ein! ")
print(f"Dein Name lautet {name.upper()}.")
```

In diesem Fall sorgt der Befehl `upper()` dafür, dass die Buchstaben als Großbuchstaben ausgegeben werden. Das ist zwar nicht sehr beeindruckend, aber dennoch ein Verarbeitungsschritt. Interessanter wird es mit if-Abfragen.

Importe

Ein Pythonprogramm beginnt üblicherweise mit dem Import von benötigten Funktionen. Z. B. wird in dieser Zeile die Bibliothek 'math' eingebunden, mit deren Hilfe man mit Zeit rechnen kann:

```
import math
```

Variablen und Listen

Variablen

Die Variablen, die wir bisher benutzt haben, haben entweder Zahlen oder Buchstaben enthalten. Je nachdem, welchen Inhalt sie haben, kann man unterschiedliche Dinge mit ihnen tun. Mit Zahlen kann man rechnen:

```
a=1 # Probiere hier auch gerne andere Zahlen aus.
b=6
print(a+b)
```

Was passiert aber, wenn wir Variablen addieren, die Buchstaben enthalten?

```
a="1"
b="6"
print(a+b)
```

Genau, die Inhalte der Variablen werden einfach miteinander verkettet. Das ist zum Beispiel dann praktisch, wenn man eine Programmausgabe zusammenbauen möchte, die unterschiedlich ausfällt, je nachdem, wie das Programm abläuft. In der Programmierung spricht man bei diesem Datentyp von Strings.

Folgende Regeln gelten für die Benennung von Variablen:

- Eine Variable beginnt mit einem Buchstaben oder mit einem Unterstrich. Um Probleme zu vermeiden, sollen nur englische Buchstaben verwendet werden.
- Der Variablenname darf keine Sonderzeichen außer dem Unterstrich einhalten.
- Variablennamen unterscheiden Groß-/Kleinschreibung. `fruit` und `Fruit` sind unterschiedliche Variablen.

Sinnvolle Konventionen für Variablennamen

- Variablen beginnen mit einem Kleinbuchstaben.
- Konstanten (Variablen, die sich im Laufe des Programms nicht ändern) bestehen aus Großbuchstaben.
- Bei einer Verkettung von verschiedenen Wörtern trennt man die Wörter mit einem Unterstrich. Beispiel: `my_very_complicated_variable`. Diese Schreibweise heißt snake_case.
- In anderen Programmiersprachen wird auch CamelCase verwendet: `myVeryComplicatedVariable`

Listen (optionales Kapitel)

```
fruit = ["Apples", "Bananas", "Raspberries"]
myGrades = [3, 4, 2, 5]

for x in fruit:
    print (x)

sum =0
for x in myGrades:
    sum +=x
    print (x)

medianGrade = sum/len(myGrades)

print("Durchschnitt: ", medianGrade)
```

Bedingungen

Bedingungen

Hier ein Beispiel für eine einfache `if`-Abfrage:

```
name = input("Bitte gib' deinen Namen ein! ")
if name == "Alice":
    print("Willkommen, Alice.")
else:
    print("Du bist nicht Alice.")
```

Eine `if`-Abfrage überprüft eine Bedingung. Als Ergebnis kommt immer entweder `True` oder `False` heraus. Hat man nicht den Namen "Alice" eingegeben, dann ist das Ergebnis der Überprüfung `False` und es wird der Code in dem `else`-Block ausgeführt. Es geht aber noch komplexer:

```
name = input("Bitte gib' deinen Namen ein! ")
if name == "Alice":
    print("Willkommen, Alice.")
elif name == "Bob":
    print("OK Bob, du darfst auch hier rein. ")
else:
    print("Du bist nicht Alice.")
```

In diesem Fall ist auch Bob zugelassen. Das geht aber auch kürzer:

```
name = input("Bitte gib' deinen Namen ein! ")
if name == "Alice" or name == "Bob":
    print(f"Willkommen, {name}.")
else:
    print("Du bist nicht Alice oder Bob.")
```

Hier ist ein Beispiel für eine Passwort-Abfrage mithilfe einer Funktion.

```
def password_correct(passwd):
  if passwd == "1234":
    return True
  else:
    return False

if password_correct(input ("Passwort eingeben: ")):
  # Hier läuft das Programm ab.
  pass
else:
  print("Kein Zugang")
```

Vergleichsoperatoren

```
==
!=
<
>
<=
>=
```

Hier ist ein Beispiel für eine Vergleichsoperation. Schreibe für die anderen Vergleichsoperatoren auch Vergleiche.

```
print(3==4)
a=3
b=4
if a == b:
    print("Die Werte sind gleich.")
else:
    print("Die Werte sind nicht gleich.")
```

Schleifen

Schleifen

Schleifen wiederholen Code, der in ihnen enthalten ist. Jeder Durchlauf einer Schleife heißt "Iteration". Dabei gibt es zwei grundsätzliche Möglichkeiten: Schleifen prüfen mit jedem Durchlauf, ob eine Bedingung erfüllt/nicht erfüllt ist und brechen gegebenenfalls ab, oder sie zählen bis zu einem bekannten Wert und brechen dann ab. Die erste Möglichkeit wird durch eine While-Schleife realisiert.

While-Schleifen

In dem folgenden Beispiel ist eine Endlosschleife dabei. Das bedeutet, dass das Programm nie normal beendet wird. Du musst daher das Programm selbst beenden, in dem du auf den Stop-Knopf drückst, der neben dem Code oder in der oberen Menüleiste erscheint.

```
import time
i=0
while i < 10:
  i+=1 # Hier wird der Wert der Variablen i um 1 erhöht. Man kann das auch so schreiben: i = i
+ 1
  print(i)
print("Die Schleife ist durchgelaufen.")

# Dieser Code wird unendlich lange ausgeführt, da die Bedingung immer wahr ist.
while True:
  print("Schleifendurchlauf")
  time.sleep(0.1)
```

Schreibe in die Bedingung der zweiten `while`-Schleife andere Bedingungen, die immer wahr sind. Z. B. 1==1 oder 2+3==5. Das Gegenteil von `True` ist übrigens `False`. Jede Bedingung, die überprüft wird, endet immer in entweder `True` oder `False`.

`While`-Schleifen eignen sich für die wiederholte Überprüfung von Bedingungen, bis die Abbruchbedingung erreicht ist und der Code hinter der Schleife ausgeführt wird. Beispiel:

```
i_am_bored = False
counter = 0
while not i_am_bored:
    counter += 1
    print(counter)
    if counter > 30:
        i_am_bored = True
print("I am bored!")
```

Hier ist eine verbesserte Version der Passwortabfrage aus dem Kapitel Bedingungen. Hier wird die Passwortabfrage in einer Schleife durchgeführt, solange, bis das Passwort richtig ist.

```
def password_correct(passwd):
  if passwd == "1234":
    return True
  else:
    return False

while not password_correct(input ("Passwort eingeben: ")):
  # überprüfe Passwort
  print("Du kommst hier nicht rein.")

print("Nun geht es los. ")
```

Ein realistischeres Beispiel wäre der Algoritmus zur Berechnung des größten gemeinsamen Teilers:

```
x,y = 1741446609,671084182
while x!=y:
    if x>y:
        x=x-y
    elif y>x:
        y=y-x
print(x)
```

For-Schleifen

Bei For-Schleifen ist beim ersten Durchgang schon klar, wie viele Iterationen es geben wird. Im ersten Beispiel wird von 0 bis 9 gezählt. Das erledigt die Funktion `range()`. In den anderen

Beispieln wird bei jedem Durchgang ein Element einer Liste ausgegeben und beim letzen Beispiel werden die Buchstaben im String `s` einzeln ausgegeben.

```
for i in range (10):
  print(i)
  pass
print("Die Schleife ist durchgelaufen.")

fruit = ["Apples", "Bananas", "Raspberries"]
for x in fruit:
  print(x)

## Iteration über einen String
s = "Theodor-Heuss-Schule"
for i in s:
    print(i)
```

Rechnungen

Rechnen mit dem Computer

Modulo

Modulo ist eine praktische Art, um zu überprüfen, ob bei einer Division ein Rest bleibt.

Beispiel:

```
print(f"Der Rest der Division von 4/2 lautet: {4%2}") #Der Rest dieser Division ist 0.
```

Potenzen

Eine Potenz schreibt man in Python mit zwei Sternchen: `**`

```
print(f"3 hoch 4 ist: {3**4}")
```

Funktionen

Definieren von Funktionen

Funktionen in Python sind definierte Codebereiche, die aber erst ausgeführt werden, wenn sie aufgerufen wurden. In diesem Fall hier gibt die Funktion mit dem Befehl `return` ein Ergebnis zurück. Die Zahlen `a` und `b`, die addiert werden sollen, werden als Parameter übergeben. Diese werden in den runden Klammern der Funktion bekanntgegeben (initialisiert). Die Funktionen sind mit Blöcken in Scratch vergleichbar.

```
# Hier stehen die Imports, wenn sie benötigt werden.
import math # Wird allerdings für die grundlegegenden Funktionen nicht benötigt.

# Definition der Addition.
def add(a,b):
  return a+b

# a = int(input("Erste Zahl eingeben: ")) die Funktion int() wird benötigt, um aus der Eingabe
eine Zahl zu machen. Wenn du keine Zahl eingibst, dann wird hier ein Fehler ausgegeben.
# b = int(input("Zweite Zahl eingeben: "))
ergebnis = add(1,2) # Das Ergebnis wird in einer Variablen gespeichert.
print("a+b ergeben: "+ str(ergebnis)) # Das Ergebnis wir auf der Konsole ausgegeben.
```

Wie du sehen kannst, ist der Befehl `return` eingerückt. Dies muss auch sein, sonst funktioniert das Programm nicht. Probier es ruhig aus. In Python werden mit den Einrückungen Codeblöcke, die zusammengehören, markiert. Andere Programmiersprachen benutzen hierfür z. B. Klammern. Entferne die Kommentarzeichen der beiden Zeilen beginnend mit a und b. Schreibe das Programm so um, dass nun die Zahlen verwendet werden, die du in der Eingabe eingibst.

Aufgabe

a) Jetzt schreibe selber eine Funktion, die a-b rechnet.

b) Schreibe auch hier das Programm so um, dass du die Zahlen eingeben kannst.

```
ergebnis = sub(1,2) # Das Ergebnis wird in einer Variablen gespeichert.
print("1-2 ergeben: "+ str(ergebnis)) # Das Ergebnis wir auf der Konsole ausgegeben.
```

Reihenfolge von Code

Code, der in einer Python-Datei steht wird in der Reihenfolge ausgeführt, in der er steht. Code, der in einem `def`-Block steht, wird erst dann ausgeführt, wenn er im Programm aufgerufen wird. Daher müssen die Definitionen auch vor dem Code stehen, der ihn aufruft. Korrigiere den nachfolgenden Code:

(Die print-Funktion in diesem Beispiel hat eine andere Formatierung. Du kannst frei wählen, welche du benutzt.)

```
zahl =  input("Bitte gib eine Zahl zwischen 1 und 10 ein.")
print(number(zahl))

def number(z):
    return f"Das ist deine Zahl: {z}"
```

Rückgabewerte von Funktionen

Funktionen müssen nicht unbedingt Werte mit `return` zurückgeben. Sie können auch einfach nur einen Teil der Programmfunktion übernehmen. Hier gibt die Funktion nur den Willkommensgruß nach dem Start des Programms aus.

```
def hello():
    print("Willkommen bei meinem tollen Programm")

print("Programm startet ...")
hello()
```

Parameter

Wie man hier sieht, wurde der Funktion auch kein Parameter übergeben. Jetzt wollen wir die Funktion etwas intelligenter gestalten. Dazu benötigen wir eine Liste mit zugelassenen Usern für dieses Programm (siehe Variablen)

```
def hello(u):
    users = ["Mr J", "Gamer32", "NoClue01"]
    if u in users:
        print("Willkommen bei meinem tollen Programm")
    else:
        print("Du darfst hier nicht sein. ")

print("Programm startet ...")
user = input("Bitte gib deinen Benutzernamen ein.")
hello(user)
```

Natürlich könnten wir auch mit einem Rückgabewert arbeiten, damit das Programm auch erfährt, ob die Anmeldung funktioniert hat oder nicht.

```
def hello(u):
    users = ["Mr J", "Gamer32", "NoClue01"]
    if u in users:
        return True
    else:
        return False

print("Programm startet ...")
user = input("Bitte gib deinen Benutzernamen ein.")
if hello(user):
    print("Willkommen bei meinem tollen Programm")
else:
    print("Du darfst hier nicht sein. ")
    exit(1) # Dieser Befehl würde das Programm an dieser Stelle mit dem Fehlercode 1 beenden.
```

Sinnvolle Namen

Es ist wichtig, Funktionen und Variablen sinnvolle Namen zu geben. Die Funktion `hello()` hat keinen sinnvollen Namen. Man muss sich überlegen, welche Funktion im Programm die `Funktion()` haben soll. In diesem Fall soll überprüft werden, ob der User im System bekannt ist. Ein Name wie `is_user()` oder `valid_user()` wären hier sinnvoller. Ändere daher den Namen `hello()` in etwas sinnvolles und sofge dafür, dass das Programm noch funktioniert.

Programmieren mit Python

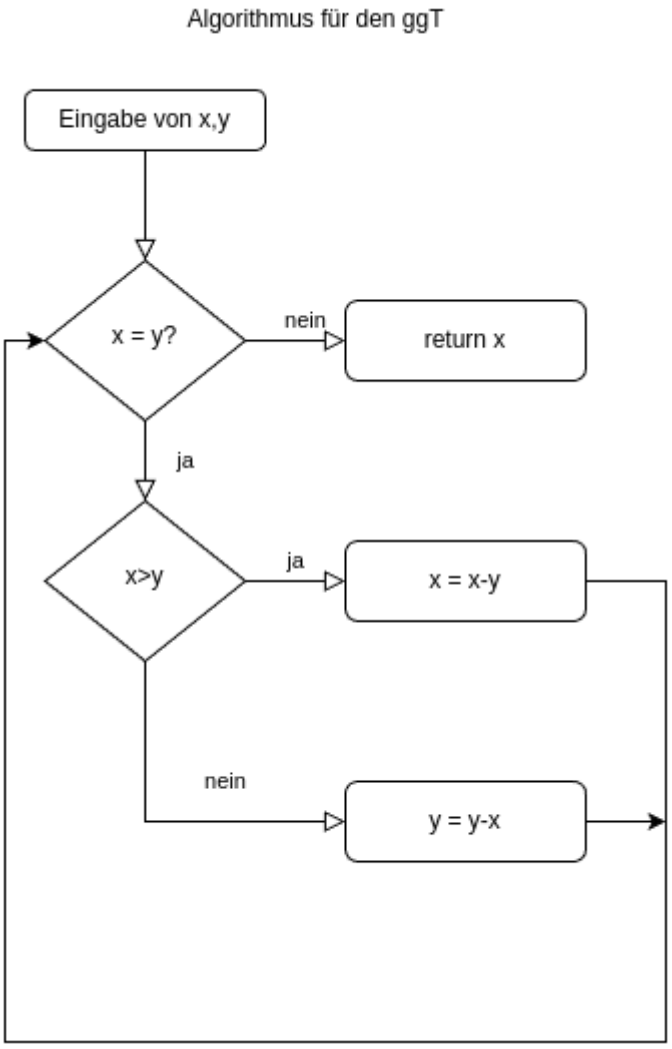
Rechnen mit Python

Python Compiler

- Online-Compiler
- Programmierumgebung Thonny

Kürzen eines Bruchs

Programmiere mit Python ein Programm, das einen Bruch (rationale Zahl) kürzt. Dazu benötigt man den ggT: Hier ist der Algorithmus für den ggT:



Dann muss der Zähler und Nenner nur noch durch den ggT geteilt werden. Das benötigt ihr dazu: Die Informationen gibt es im Unterricht. Hier ist das Grundgerüst für dein Programm:

```
def ggt(x,y):
    """
    Die Funktion berechnet den größten gemeinsamen Teiler aus den beiden
    Parametern x und y.
    """
    while x!=y:
        #Dein Code
    return x


def kuerzen(x,y):
    """
    Diese Funktion kürzt den Bruch x/y:
    """
    g=ggt(x,y)
```

```
    #Dein Code
    ergebnis= str(int(x))+ '/'+ str(int(y))
    return ergebnis

print ("Der Bruch lautet gekürzt: ", kuerzen(65,135))
```

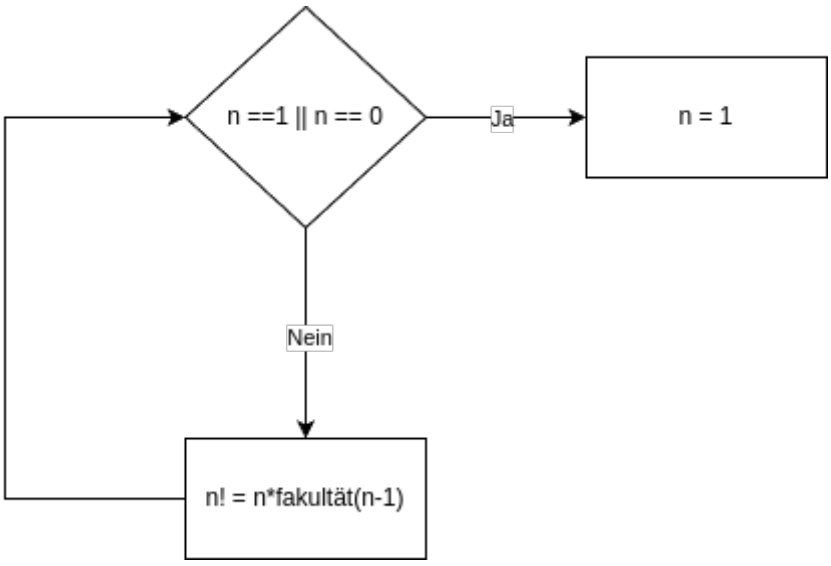
Fakultät

Als nächstes berechnen wir die Fakultät. ACHTUNG: Zum Ausprobieren wählt keine großen Zahlen, da der Rechner schnell überfordert sein wird. Die Fakultät ist folgendermaßen definiert:

$$n! = n * (n-1)!$$

$$n! = 1 \begin{cases} n = 0 \\ n = 1 \end{cases}$$

Um das zu berechnen, können wir einfach folgenden Algorithmus verwenden:



Fibonacci-Folge

Für diejenigen, die schnell fertig sind, gibt es noch die Fibonaccifolge, die folgendermaßen definiert ist: 0,1,1,2,3,5,8,13... . Na, wie geht es wohl weiter?

Ackermann-Funktion

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m \geq 1 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m \geq 1 \text{ and } n \geq 1 \end{cases}$$

Die Ackermann-Funktion gilt nur für Ganzzahlen >=0!
Folgendermaßen kann diese Funktion umgesetzt werden:

```
#### Die Ackermann-Funktion, umgesetzt in Python
import sys
sys.setrecursionlimit(10000) # Das Limit für Rekursion in Python ist 1000. Das ist schnell
erreicht.

def ackermann(m,n):
  # Für den ersten Fall, dass m=0 ist, wird das Ergebnis n+1 zurückgegeben.
  # Schreibe hier deinen Code für diesen Fall.

  # Für den zweiten Fall, dass m>=1 und n=0 ist, wird als Ergebnis der erneute Aufruf der
Funktion übergeben mit den beiden Parametern: ackermann(m-1,1)
  # Schreibe hier den Code für diesen Fall.

  # Für den dritten Fall gilt, dass m und n >= 1 sind. In diesem Fall ist das Ergebnis
ackermann(m-1, ackermann(m,n-1))
  Schreibe hier deinen Code für den dritten Fall.
```

Teste zunächst mit Zahlen <=4!!