Physik mit dem Raspberry Pi - Python-Basics

Interactive Shell

Durch die Eingabe von **python** im Linux-Terminal wird eine *interactive shell* gestartet. Diese erlaubt die Ausführung kurzer Programmsegmente und das Testen von Befehlen. Längere Programme sollten jedoch als Textdatei gespeichert und anschließend ausgeführt werden.

Aufgabe

Starte eine *interactive shell* und nutze diese als Taschenrechner!

Kommentare

Um Code lesbarer zu gestalten, ist eine Dokumentation mit Kommentaren von äußerster Wichtigkeit. Einzeilige Kommentare beginnen hierbei mit **#**, mehrzeilige Kommentare beginnen und enden mit **'''**.

Beispiele

```
# Dies ist ein Kommentar
var = "Dies ist keine Kommentar"
'''
Dies ist ein Kommentar, der
zwei Zeilen umfasst.
'''
```

Variablen

Eine Variable ist ein Container für Daten, dessen Inhalt innerhalb eines Programmablaufs gesetzt, verändert und abgerufen werden kann.

Die Wertezuweisung findet über den **=**-Operator statt. Anders als bei vielen anderen Programmiersprachen, ist die Angabe eines Datentyps bei Python nicht notwendig, da dieser anhand der zugewiesenen Information selbst erkannt wird.

Beispiele

```
zahl1 = 5        # Ganzzahl (integer)
zahl2 = 5.       # Fließkommazahl (float)
text = "Text"    # Zeichenkette (string)
```

Syntax

Eine korrekte Formatierung des Quellcodes erhöht nicht nur die Lesbarkeit des Programms, sondern ist auch Teil der Syntax. Schleifen, Funktionsdefinitionen etc. müssen eingerückt werden, da das Programm sonst nicht funktionsfähig ist.

print-Befehl

Erlaubt die Ausgabe von Text und Daten im Terminal. Die auszugebenden Variablen werden als Funktionsparameter übergeben, ihr Datentyp wird erkannt und ihr Inhalt in entsprechender Textform ausgegeben.

```
print("Hallo", 3, 7.5)
> Hallo 3 7.5
```

Standardmäßig erfolgt die Trennung der Variablen in der Ausgabe mit Leerzeichen. Über die Option **sep = ""** im Funktionsaufruf kann dies jedoch angepasst werden.

```
print("Hallo", 3, 7.5, sep="_")
> Hallo_3_7.5
```

Zahlen können direkt in Zeichenketten konvertiert werden

```
print(str(5))
> 5
```

Dies erlaubt die Addition von Zeichenketten, was einer Aneinanderreihung entspricht

```
print(str(5) + ' ist ungleich ' + str(10))
> 5 ist ungleich 10
```

Aufgabe

1. Nutze die Funktion **raw_input()**, um eine Benutzereingabe einzulesen und in einer Variable zu speichern
2. Gib den Inhalt der Variable über den **print**-Befehl aus

Boolesche Algebra

Der boolesche Datentyp ist auf zwei Werte beschränkt: **True** und **False**.

Die Wertezuweisung erfolgt analog zu anderen Datentypen

```
LOW = True      # Ausdruck ist wahr
HIGH = False    # Ausdruck ist falsch
```

Boolesche Operatoren

Operatoren erlauben die Verneinung und Verkettung von booleschen Ausdrücken, was die Formulierung komplizierter Bedingungen ermöglicht. Dabei existieren die folgenden Operatoren

- **not**: Umkehrung des Wahrheitswerts
- **and**: True, wenn linker und rechter Ausdruck True sind
- **or**: True, wenn linker oder rechter Ausdruck True sind

Vergleichsoperatoren

Die Vergleichsoperatoren zweier Variablen, hier **var1** und **var2**, liefern je einen booleschen Wert zurück, der für die spätere Steuerung des Programmablaufs äußerst wichtig ist:

```
var1 == var2    # True, wenn var1 gleich var2, sonst False
var1 > var2     # True, wenn var1 größer als var2, sonst False
var1 >= var2    # True, wenn var1 größer oder gleich var2, sonst False
# Analog für kleiner und kleiner gleich
var1 != var2    # True, wenn var1 ungleich var2, sonst False
```

Aufgaben

Wahr oder falsch? Entscheide selbst, ohne Python zu verwenden und überprüfe schließlich mittels Python deine Behauptung

```
3 > 4
3.5 != 5
not not 5 == 5
not ((not 9.3 >= 3) and (5 <= 3)) or (3 != 5)
```

if-Bedingung

Der Programmteil unterhalb der **if**-Bedingung wird ausgeführt, falls diese wahr ist. Sonst, führe Programmteil unter **else** aus, falls angegeben. Über den **elif**-Befehl wird die Verkettung mehrer **if**-Abfragen ermöglicht.

Beispiel

```
if x < 3:
    print('x ist kleiner als 3')
elif x == 3:
    print('x ist gleich 3')
else:
    print('x ist groesser als 3')
```

Aufgaben

Einfache if-Abfrage

- Nutze die Funktion **raw_input()**, um eine Benutzereingabe einzulesen und in einer Variable zu speichern
- Nutze den modulo-Operator (**%**), um zu überprüfen, ob eine Zahl durch 2 teilbar ist

```
# Beispiele für modulo-Operator
print( 5 % 3 )  # nicht durch 3 teilbar
> 2
print( 6 % 3 )  # durch 3 teilbar
> 0
```

Verschachtelte if-Abfragen

- Nutze die Funktion **raw_input()**, um eine Benutzereingabe einzulesen und in einer Variable zu speichern
- Überprüfe, ob die eingegebene Zahl durch 2 und 3 teilbar ist. Falls ja, gib dies in Textform wieder. Falls nicht, überprüfe ob die Zahl durch 2 und 5 teilbar ist. Treffen alle Bedingungen nicht zu, gib das Ergebnis in entsprechender Textform wieder.

Vereinfachung von if-Abfragen (P)

Vereinfache das folgende Programm, sodass es mit lediglich einer **if**-Abfrage auskommt

```
zahl = int(raw_input('Bitte Zahl eingeben: '))

y = False
if zahl > 5:
  y = True
  if zahl <= 100:
    y = True
  else:
    y = False

print(y)
```

while-Schleife

Eine **while**-Schleife wird ausgeführt, solange ihre Bedingung erfüllt ist. Ein Abbruch der Schleife kann mittels **break**-Befehl erfolgen.

Beispiele

- Endlosschleife

```
while True:
    [...]
```

- Zählerinkrementierung

```
i = 0
while i < 100:
    i += 1  # Entspricht: i = i + 1
print( i )
> 100
```

Aufgaben

Abbruch einer Endlosschleife mittels break

- Schreibe eine Endlosschleifen, in der ein Zähler erhöht wird
- Brich die Schleife ab, sobald der Zähler größer als 50 ist

Ausgabe der Fibonacci-Zahlen (P)

- Fibonacci-Zahlen: Erstes und zweites Element sind 0 und 1
- Jedes folgende Element wird gebildet, indem die letzten beiden Elemente addiert werden
- Es ergibt sich die Folge: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
- Gib alle Fibonacci-Zahlen < 1000 auf dem Bildschirm aus

Datentyp: Listen

Eine Liste ist ein Container, der der Speicherung von mehreren Elementen dient. Diese müssen dabei nicht vom selben Datentyp sein, was insbesondere die Speicherung von Listen in Listen erlaubt (was letztendlich gleichwertig zu einer Matrix ist).

- Eine leere Liste **l** wird mittels

```
l = []
```

initialisiert. Eine Initialisierung mit Elementen ist jedoch auch möglich, z.B.

```
l = [1, 5, "Haus", 3.7]
```

- Der Zugriff auf die Elemente der Liste erfolgt über den Index (**Wichtig:** Der Index beginnt bei 0!)

  Z.B. Zugriff auf das erste Element über

```
print(l[0])
> 1
```

- Bei Listen in Listen, wie

```
ll = [[1, 2], [3, 4]]
```

  erfolgt der Zugriff über mehrere eckige Klammern, z.B.

```
print(ll[1][0])
> 3
```

- Über den **append**-Befehl können Elemente an bereits bestehende Listen angefügt werden

```
l = [0, 1, 2]
l.append( 3 )
print(l)
> [0, 1, 2, 3]
```

- Die Addition zweier Listen kombiniert diese

```
l = [0, 1, 2]
l2 = [3, 4, 5]
print( l + l2 )
> [0, 1, 2, 3, 4, 5]
```

- Die Anzahl der Elemente in einer Liste kann mittels **len()**-Befehl ausgegeben werden

```
l = [0, 1, 2]
print( len(l) )
> 3
```

Aufgabe

- Erzeuge eine leere Liste

- Füge in einer **while**-Schleife die Zahlen von 0 bis 50 in die Liste ein

for-Schleife

Eine Schleife, die über jedes Element in einer Liste iteriert.

In diesem Zusammenhang wird häufig die Funktion **range()** verwendet. Diese besitzt drei Parameter, die ein Minimum, ein Maximum und eine Schrittgröße angeben. Anschließend wird eine Liste erzeugt, die von Minimum bis Maximum - 1 reicht.

```
print( range(2, 8, 2) )
> [2, 4, 6]
```

Die Angabe des Minimums und der Schrittgröße sind hierbei optional und sind standardmäßig auf Werte von 0 und 1 gesetzt.

Der Grund, warum die zurückgegebene Liste bis zu einem Wert von Maximum - 1 reicht, ist, dass die Funktion häufig in einer **for**-Schleife verwendet wird, d.h.

```
for x in range(10):
    [...]
```

wird genau 10 Iterationen durchlaufen.

Beispiel

```
l = [3, 5, 8]   # Initialisierung einer Liste

for element in l:
    print(element)
> 3
> 5
> 8
```

Aufgaben:

Einfache Schleife

Erzeuge eine Schleife mit 5 Durchläufen und gib jeweils den aktuellen Wert des Schleifeniterators aus

Kumulative Summe (P)

Berechne die kumulative Summe der Liste

```
l = [3, 8, 2, 5, 8]
```

- Zuerst, initialisiere eine Summenvariable mit 0
- Iteriere über die Elemente von **l** und addiere in jedem Schritt den Wert des jeweiligen Elements zur Summenvariable und gib diese anschließend aus

Funktionsdefinition

Eine Funktion wird über das Keyword **def** definiert. Die genaue Syntax ist hierbei:

```
def <Funktionsname>(parameter1, paramater2, ...):
  [...]
  return <Wert>
```

Die Rückgabe des Ergebnisses erfolgt über den **return**-Befehl, welcher optional ist. Auch eine Rückgabe mehrerer Werte ist möglich - diese werden dann über ein Komma abgetrennt.

Der Aufruf erfolgt schließlich über:

```
# Ein Rückgabewert
ergebnis = <Funktionsname>(parameter1, parameter2, ...)
# Zwei Rückgabewerte
ergebnis1, ergebnis2 = <Funktionsname>(parameter1, parameter2, ...)
```

Beispiel

```
def Summe(zahl1, zahl2):
    return zahl1 + zahl2

print( Summe(3, 4) )
> 7
```

Aufgabe

- Definiere eine Funktion, der zwei Zahlen übergeben werden. Die Funktion addiert und multipliziert diese miteinander und gibt schließlich die beiden Ergebnisse zurück
- Stelle die Ergebnisse anhand zweier Beispielzahlen am Bildschirm dar

Ausgabe in Datei

Ergebnisse können nicht nur über **print** auf dem Bildschirm ausgegeben werden, sondern auch direkt in einer Textdatei gespeichert werden.

Dazu muss diese erst über den entsprechenden Befehl geöffnet werden

```
f = open(<Dateiname>, 'w')
```

Der Parameter 'w' gibt in diesem Fall den Modus an. Wichtig sind hierbei 'w' für 'write' und 'r' für 'read'.

Anschließend kann über den **write()**-Befehl eine Zeichenkette in die geöffnete Datei geschrieben werden

```
f.write("<Zu speichernder Text>" + "\n")
```

Wichtig: Das newline-Zeichen ("\n") beschreibt hierbei den Zeilenumbruch, der nicht automatisch erfolgt!

Ist eine Datei fertig beschrieben, so wird diese über den **close()**-Befehl geschlossen

```
f.close()
```

Aufgabe (P)

- Öffne eine Datei
- Beschreibe diese in einer Schleife deiner Wahl mit den Zahlen von 0 bis 50 in der ersten Spalte und den Zahlen von 50 bis 0 in der zweiten Spalte. **Beachte**: Addition von Zeichenketten über **+**-Operator möglich (so z.B. Einfügung von Tabulatoren möglich: str(zahl1) + '\t' + str(zahl2))

import-Befehl

Der **import**-Befehl erlaubt die Einbindung von Modulen in ein Programm. Diese stellen wichtige Unterfunktionen (Zugriff erfolgt über **.**-Operator) bereit.

Beispiel

```
import math
math.sin(5)
```

Dies erlaubt auch den import einzelner Unterfunktionen

```
from math import sin
sin(5)
```

Dies kann jedoch die Lesbarkeit des Codes erschweren, da nicht mehr klar erkennbar ist, zu welchem Modul welche Funktion gehört.

Bei import eines Moduls kann dessen Name neu vergeben werden

```
import math as m
m.sin(5)
```

time-Modul

Umfasst eine Vielzahl an Funktionen, die sich mit dem Auslesen, der Ausgabe und der Manipulation von Zeitangaben beschäftigen. Wichtige Funktionen sind hierbei

- **time()**: Gibt die aktuelle Zeit in Unixzeit zurück, d.h. die Zeit in Sekunden seit dem 1.1.1970 00:00

  ```
  import time
  print( time.time() )
  > 1523817961.406057
  ```

- **sleep()**: Eine Zeit in Sekunden wird übergeben. Der Programmcode wird an der Stelle des Aufrufs für die angegebene Zeit angehalten

  ```
  import time

  zeit = time.time()
  time.sleep(5)
  print( time.time() - zeit )
  > 5.0015411377
  ```