Python Grundlagen

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik`
Semester	`Wintersemester 2022/23`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Grundlagen der Programmiersprache Python`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_ProzeduraleProgrammierung/blob/master/08_PythonGrundlagen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Florian Richter

www.python.org

Fragen an die heutige Veranstaltung ...

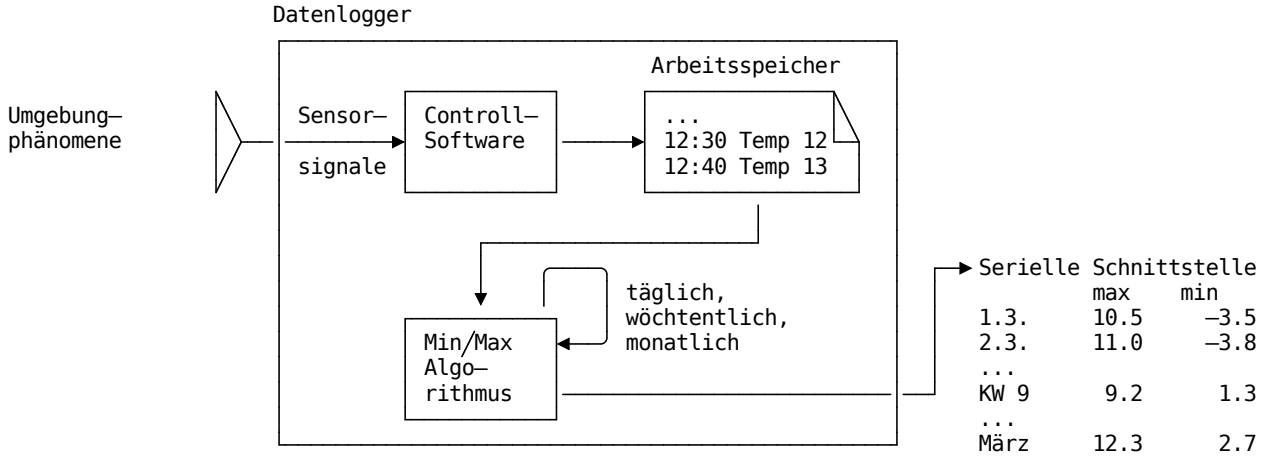
- Warum sollte man andere Programmiersprachen als C++ in Betracht ziehen?
- Welche Vorteile hat die Sprache **Python**?
- Wie unterscheidet sich der Syntax von C++?

Beispiel-Literatur:

- Schnellstart Python - Ein Einstieg ins Programmieren für MINT-Studierende (kostenloses E-Book für TUBAF-Studenten)
- Python 3 - Das umfassende Handbuch

Motivation

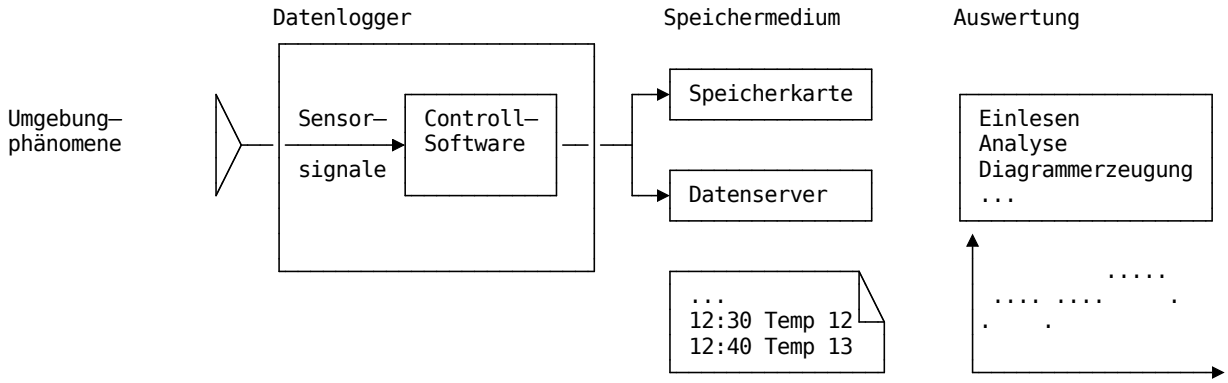
Aufgabe: Bestimmen Sie die maximalen Tages- und Nachttemperaturen pro Tag, Woche und Monat mit unserem Mikrocontroller über einem Jahr. Wie gehen Sie vor?



Frage: Welche Nachteile sehen Sie in dem Vorgehen?

- Unser Controller hat keinen "festen Speicher" - die Daten würden im Arbeitsspeicher liegen und wären damit im Fall einer Stromunterbrechung verloren
- Eine Messung pro 10 Minuten bedeutet, dass pro Tag $6 \cdot 24 = 144$ Datensätze entstehen. Im Monat sind das etwa $30 \cdot 144 = 4320$. Wenn wir davon ausgehen, dass wir die Daten als 4 Byte breite Float Werte abbilden, würde dies für eienn Monat 17.280 Byte (16.8 kB) und für ein Jahr 210.240 Byte (205 kB) Speicherbedarf bedeuten. Unser Controller verfügt über 256 kB Arbeitsspeicher.
- Die lokale Verarbeitung ist energieaufwändig. Wenn wir davon ausgehen, dass der Controller in einem Feldversuch möglichst lange mit einer Batterie betrieben werden soll, so müssen wir den Berechnungsaufwand reduzieren.

Lösungsansatz: Wir kommunizieren die Rohdaten in eine Cloud und verarbeiten diese separat.



> Wir entkoppeln damit die Datenerzeugung und Auswertung! Welche Programmiersprache benutzen wir aber für die Auswertung? Warum nicht einfach weiter C++?

Python erleichtert die schnelle Entwicklung von Scripten zur Datenauswertung.

- Schnelles Ausprobieren durch:
  - Einfache Syntax, kurze Programme
  - Interpreter statt Compiler
  - Interaktivmodus
  - Keine Deklarierung von Datentypen
  - Automatisches Speichermanagement
- Plattformunabhängigkeit (Linux, Windows, MacOS und auch Mikrocontroller)
- Einfache Erweiterbarkeit, große Standardbibliothek und viele weitere frei verfügbare Module

C++

Python

```python
print("I will not throw paper airplanes in class\n" * 10)
```

```
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
```

Python Historie

- *relativ* jung Sprache, Entwicklung 1990 in Amsterdam (vgl. C 1972, C++ 1979)
- Ziel sollte eine Sprache sein, die auch von Programmieranfängern einfach erlernt werden kann
- 2008 erschien Version 3.0 mit einer Komplettüberarbeitung und vielen Verbesserungen
- Aktuell sollte **mindestens Version 3.7** oder neuer verwendet werden (**3.11** ist die neueste Version)
    - Anaconda nutzt derzeit Version 3.9

> Achten Sie bei Ihren Recherchen auf das Alter Ihrer Quellen, bzw. die verwendete Python-Version. Man findet noch viele Anleitungen zu Python 2.7. Diese sind häufig jedoch **nicht mehr kompatibel** zu Python 3.

Beispiele für den Einsatz von Python

Typische Einsatzszenarien sind z.B.:

- Kleine Server-Programme
- Schnelles entiwckeln von Scripten zur Datenauswertung
- Konvertierung / Formatierung von Dateien bzw. großen Datenmengen
- Numerische Berechnungen und Generierung von Diagrammen
- **Maschinelles Lernen** dank Bibliotheken, wie TensorFlow, scikit-learn oder PyTorch
- Python als Scriptsprache für Erweiterungen (z.B. **ParaView**, **QGIS**, Blender, Gimp)
- Programmierung auf dem RaspberryPi
- Etwas spezieller: Programmierung von Robotern mit ROS (Robot Operating System)
- ...

QGIS Python Konsole

www.raspberrypi.com

Entwicklungsumgebungen

Wie installiere ich bei mir Python?

Unter Linux und MacOS kann Python z.B. über die Kommandozeile bzw. Paketmanager installiert werden. Für Windows empfiehlt sich die Installation der Anaconda Distribution (auch für Linux und Mac verfügbar).

> Für Python gibt es viele verschiedene Entwicklungsumgebungen. In den Übungen nutzen wir aber weiterhin Visual Studio Code.

Anaconda nutzt als Standard-Editor **Spyder** (**S**cientific **PY**thon **D**evelopment **E**nvi**R**onment). Beliebt sind auch interaktive **Jupyter**-Dokumente (Notebooks).

Spyder Editor

Jupyter Notebook

Elemente der Programmiersprache

Welche Programmierparadigmen unterstützt Python?

- Imperativ?
- Prozedural?
- Objektorientiert?

```
1 for i in range(10):
2   print("Hallo Welt", i)
```

```
Hallo Welt 0
Hallo Welt 1
Hallo Welt 2
Hallo Welt 3
Hallo Welt 4
Hallo Welt 5
Hallo Welt 6
Hallo Welt 7
Hallo Welt 8
Hallo Welt 9
```

Welche Programmierparadigmen unterstützt Python? → **Alle 3** 😛

- Imperativ ✔️
- Prozedural ✔️
- Objektorientiert ✔️

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

```
for i in range(10):
  print("Hallo Welt", i)
```

```
def hallo():
    for i in range(10):
        print("Hallo Welt", i)
```

```
class Hallo:
    def __init__(self):
        for i in range(10):
            print("Hallo Welt", i)
```

Zuweisungen und Datentypen

Python nutzt eine dynamische Typisierung, d.h. eine Variable wird erst zur Laufzeit mit Typ des zugewiesenen Werts assoziiert. Sie kann auch mit einem beliebigen neuen Datentypen überschrieben werden.

```
1  a = 1
2  a += 3
3  type(a)
4
5  a = "hello"
6  type(a)
7
8  # Python kann ohne Hilfsvariable, 2 Variablen tauschen:
9  x, y = 3.0, -2.0
10 x, y = y, x
11 print(x, y)
12
13 # a = b
```

```
-2.0 3.0
```

Verkürzte Schreibweisen für Rechenoperationen, wie += -= *= und /= existieren. Ein ++ oder -- gibt es in Python aber nicht.

```
1  x = 10
2  y = 5
3  x /= y
4  x += 8.5
```

```
5 x *= 4
6 print(x)
```

```
42.0
```

Standard Datentypen

Die Standard-Datentypen von Python bringen neben elementaren Datentypen wie **int**, **float** und **bool** auch komplexere Datentypen mit, die uns bei der schnellen Implementierung von mathematischen Formeln oder großen Datenstrukturen helfen.

Numerisch	int, float, complex
Text	str
Sequenz	list, tuple, range
Weitere	z.B. bool, dict, set, bytearray
NoneType	None

Namenswahl für Variablen

Namen von Variablen, Funktionen, Klassen,... bestehen aus Buchstaben, Ziffern, Underscore (_)

- Ziffer darf nicht am Anfang stehen
- Ab Python 3 sind auch Buchstaben außerhalb des ASCII-Bereichs erlaubt
  - Buchstaben aus Alphabeten fast aller Sprachen, z.B. ä, ö, ü, θ, π, ラ
  - nicht erlaubt sind Sonderzeichen wie €, $, @,§ sowie Emojis
  - guter Stil ist aber, trotzdem nur Zeichen aus ASCII-Bereich zu verwenden!
- lesbare aber nicht zu lange Namen
  - z.B. sum, value
- Hilfsvariablen, die man nur über kurze Strecken braucht, eher kurz:
  - z.B. i, j, x
- mit Kleinbuchstaben beginnen, Worttrennung durch "_" oder Großbuchstaben, z.B.
  - input_text („Snake Case“), empfohlen in Style Guide for Python Code
  - inputText („Camel Case“)
- Variablen, die man im ganzen Programm braucht, eher länger:
  - z.B. input_text

Reservierte Bezeichner:

False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Siehe auch Python Tutorial zum Thema Coding Style:

- https://docs.python.org/3/tutorial/controlflow.html#intermezzo-coding-style
- https://peps.python.org/pep-0008/

Ausgabe

Die **print**-Funktion ist unser Äquivalent zu std::cout aus C++. Beliebig viele Konstanten und Variablen können als Parameter übergeben werden. Die uns bereits bekannten Sonderzeichen **\t** (Tabulator) und **\n** (Neue Zeile) können wie gewohnt eingebaut werden.

```
1  x = 3.141
2  y = x**2
3  print("5 * 5 =", 25)
4  print("x \t=\t", x, "\nx² \t=\t", y)
```

```
5 * 5 = 25
x       =       3.141
x²      =       9.865881
```

Mit dem optionalen Parameter *end=* können Sie den standardmäßigen Zeilumbruch durch ein anderes Zeichen ersetzen. Die erweiterte Formatierung von Ausgaben ist Teil der nächsten Vorlesung.

```
print(x, y, z, end=",")
```

Eingabe

Ergebnis der **input**-Funktion ist immer ein **String**. Für die Verarbeitung als Zahl ist daher immer eine **Typenkonvertierung** erforderlich. Eine Eingabeaufforderung kann optional als Parameter übergeben werden.

Falsch:

```
1  """
2  Liefert einen Fehler, da x ein String ist
3  """
4  x = input("x eingeben:")
5  y = x**2
6  print("y = ", y)
```

```
x eingeben:
```

Richtig:

```
1  x = float( input("x eingeben:") )
2  y = x**2
3  print("y = ", y)
```

```
x eingeben:
```

Zusätzliche Module einbinden

Mit **import** lassen sich Module einbinden. Ähnlich zu den Standard C++ Headern sind die Bestandteile der Python Module in einen Namensraum gekapselt.

```cpp
#include <cmath>

x = std::sin(M_PI)
std::cout << x;
```

```python
import math

x = math.sin(math.pi)
print(x)

#---- Formatiertes Beispiel:
print(f"{x:.5f}")
```

Bei Bedarf lassen sich alle oder einzelne Komponenten in den eigenen Namensraum importieren:

```python
from math import *       # Einfach, aber sollte wenn möglich vermieden werd
from math import sin,cos
```

Eine Umbennung ist alternativ auch möglich:

```python
import math as m
print(m.pi)
```

Listen

In Python ist alles ein Objekt! Z.B. auch elementare Datentypen, wie **Integer**. Komplexere Datentypen, wie Listen verfügen über Methoden zu deren Manipulation. Die Länge einer Liste fragen wir hingegen mit der **Funktion len** ab.

Der Aufruf von Methoden funktioniert wie in C++. Der Datentyp **list** verfügt über mehrere spezielle Methoden:

- reverse(), sort(), extend(), append(), remove(), clear(), ...

```
liste = [4,2,3,1]
print( len(liste) )

liste.reverse()
print(liste)

liste.sort()
print(liste)

liste.append(7)
liste.extend([5,6])
print(liste)

liste.remove(3)
print(liste)

print( len(liste) )
```

```
4
[1, 3, 2, 4]
[1, 2, 3, 4]
[1, 2, 3, 4, 7, 5, 6]
[1, 2, 4, 7, 5, 6]
6
```

Listen-Indizierung

Die Möglichkeiten Listenelemente zu indizieren sind sehr vielfältig in Python. Wir können beispielsweise auch vom Ende der Liste starten oder bestimmte Bereiche auswählen. **Wichtig:** Listen beginnen immer mit dem Index **0**.

```
data = [10,20,50,200,300,350]

print( data[-1] )
print( data[1:] )
print( data[1:4] )
print( data[1:4:2] )
print( data[-3:] )
```

```
350
[20, 50, 200, 300, 350]
[20, 50, 200]
[20, 200]
[200, 300, 350]
```

Konstrollstrukturen

Wiederholung:

1. **Verzweigungen (Selektion):** In Abhängigkeit von einer Bedingung wird der Programmfluss an unterschiedlichen Stellen fortgesetzt.

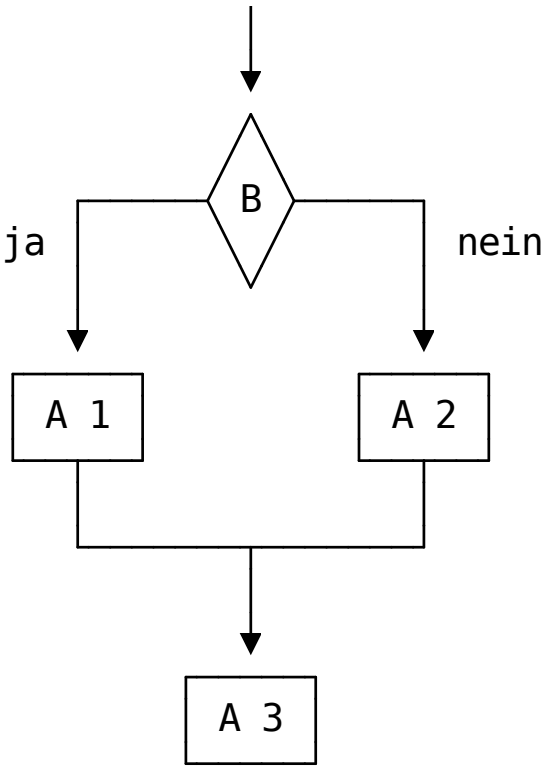
Beispiel: Wenn bei einer Flächenberechnung ein Ergebnis kleiner Null generiert wird, erfolgt eine Fehlerausgabe. Sonst wird im Programm fortgefahren.

2. **Schleifen (Iteration):** Ein Anweisungsblock wird so oft wiederholt, bis eine Abbruchbedingung erfüllt wird.

Beispiel: Ein Datensatz wird durchlaufen um die Gesamtsumme einer Spalte zu bestimmen. Wenn der letzte Eintrag erreicht ist, wird der Durchlauf abgebrochen und das Ergebnis ausgegeben.

Selektion mit if

Auch in Python sind innerhalb einer Selektion mehrere Alternativen möglich. Eine **if**-Anweisung kann von beliebig vielen **elif** Zweigen gefolgt werden. Am Ende darf **höchstens** ein **else**-Zweig stehen.



Aufbau:

- **if** – Schlüsselwort
- Bedingung → kann zu True oder False evaluiert werden
  - auch komplexe Bedingungen, z.B. verknüpft mit **and** oder **or**
  - *if a > 5 and a < 10 :*
- Doppelpunkt ':'
- Anweisungen, die ausgeführt werden, falls Bedingung gilt (**eingerückt**)
- optionaler **else**-Zweig

```
if Bedingung:
  Anweisung1
else:
  Anweisung2

Anweisung3
```

```
if x == y:
  print("x and y are equal")
elif x < y:
  print("x is less than y")
else:
  print("x is greater than y")
```

Achten Sie in Python immer auf eine korrekte und möglichst konsistente **Einrückung**!

Kleines Beispiel

```
1  a = int( input("Erste Zahl:") )
2  b = int( input("Zweite Zahl:") )
3
4  if a%2 == 0 and b%2 == 0:
5    print("Beide Zahlen sind gerade!")
6  elif a%2 == 0:
7    print("Nur die erste Zahl", a, "ist gerade!")
8  elif b%2 == 0:
9    print("Nur die zweite Zahl", b, "ist gerade!")
10 else:
11   print("Keine Zahl ist gerade!")
```

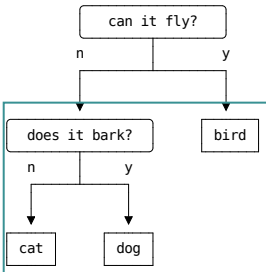
```
Erste Zahl:
```

Erinnerung: Der **Modulo**-Operator % liefert den Rest einer ganzzahligen Division.

Verschachtelte If-Anweisungen

If-Anweisungen lassen sich beliebig verschachteln. Dabei ist wieder ganz besonders auf eine korrekte **Einrückung** zu achten!

Beispiel: Tiere raten

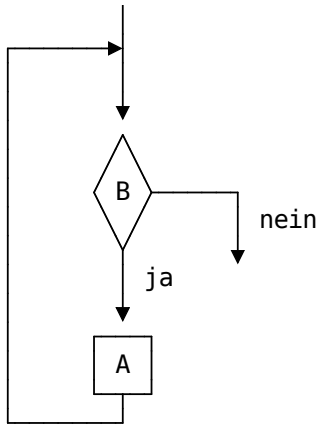



```
1 if input("can it fly? ") == 'y':
2   print("bird")
3 else:
4   if input("does it bark? ") == 'y':
5     print("dog")
6   else:
7     print("cat")
```

```
can it fly?
```

While-Schleife

In Python gibt es zwei Arten von Schleifen: **for** und **while** (abweisende Schleifen). Beide Schleifenarten kennen Sie auch aus C++. Erinnerung: eine while-Schleife wird **solange** ausgeführt, wie die Bedingung erfüllt ist (**True**).




```
while <bedingung>:
  <anweisungen>
```

Beispiel: Fibonacci-Zahlen < 100

```
1  a, b = 0, 1
2  while b < 100:
3    print(b, end='|')
4    a, b = b, a+b
```

```
1|1|2|3|5|8|13|21|34|55|89|
```

For-Schleife

For-Schleifen erlauben auch in Python eine Wiederholung mit festgelegten Start- und Endwerten. Die Syntax unterscheidet sich aber von C++. Das Schlüsselwort **in** ist ein fester Bestandteil und wird gefolgt von einer **Sequenz**.

```
for <variable> in <sequenz>:
  <anweisungen>
```

Die Schreibweise ermöglicht ein sehr einfaches Abarbeiten von Listen und anderen indizierbaren Datenstrukturen:

Beispiele mit verschiedenen Sequenzen:

```
for i in ['a','b','c']:
  print(i, end=",")
for i in "abc":
  print(i, end=",")

for i in [5,7,9]:
  print(i, end=",")
for i in [3.4,7.9,6.78]:
  print(i, end=",")

```

```
a,b,c,a,b,c,5,7,9,3.4,7.9,6.78,
```

Die **range**-Funktion kann mit 1, 2 oder 3 Parametern aufgerufen werden. Sie wird hauptsächlich für for-Schleifen benötigt. **Wichtig:** der Endwert ist in der Menge **nicht** enthalten!

- **Signatur:** range([start,] stop[, step])
  - start, step sind optionale Parameter
  - 1 Parameter → nur Endwert angeben (start=0, step=1)
  - 2 Parameter → Start- und Endwert angeben (step=1)
- Ergebnis der Funktion ist ein **range-Objekt**, keine Liste! Es lässt sich aber in eine Liste konvertieren:

```
x = range(0,5)
print(x, type(x))
print( list(x) )
```

```
range(0, 5) <class 'range'>
[0, 1, 2, 3, 4]
```

Beispiele mit der range-Funktion:

```
for x in range(5):
  print(x+1, end=" ")
print("\n")

for x in range(1,6):
  print(x, end=" ")
print("\n")
```

```
for x in range(20,-1,-2):
  print(x, end=" ")
print("\n")
```

```
1 2 3 4 5

1 2 3 4 5

20 18 16 14 12 10 8 6 4 2 0
```

Schleifen oder List Comprehension

List Comprehension oder Listen-Abstraktion ermöglicht eine sehr kurze Schreibweise bei der Arbeit mit Listen. Es erinenrt etwas an mathematische Mengendefinitionen, wie $\{ x^2 \mid x \in \mathbb{N} \wedge x < 20 \}$.

```
square = [x**2 for x in range(20)]
print( square )

# alternative Darstellung mit einer Schleife
square = []
for x in range(20):
  square.append(x**2)
print( square )
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361]
```

List Comprehensions werden oft als "pythonischer" beschrieben als loops (oder `map()`). Sie sind auch aussagekräftiger als Schleifen, was bedeutet, dass sie einfacher zu lesen und zu verstehen sind. Schleifen entwickeln einen Iterationsprozess, während Sie sich mit einem *List Comprehensions* stattdessen auf das konzentrieren, was Sie in die Liste umsetzen wollen.

Beispiele:

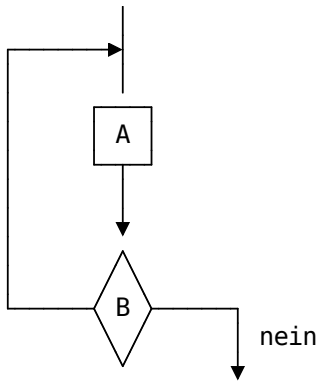
```
square = [x**2 for x in range(20)]
print( square )

cities = ["Dresden", "Freiberg", "Leipzig", "Frankenberg"]
print( [c.upper() for c in cities] )

```

```
 7  # Nur Städte, die mit F beginnen
 8  cities = ["Dresden", "Freiberg", "Leipzig", "Frankenberg"]
 9  print( [c.upper() for c in cities if c[0] == "F"] )
10
11  # Wir können auch das Ergebnis manipulieren
12  cities = ["Dresden", "Freiberg", "Leipzig", "Frankenberg"]
13  print( [c.upper() if c != "Freiberg" else c.upper()+" (Sachs)" for c
      in cities if c[0] == "F"] )
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256,
289, 324, 361]
['DRESDEN', 'FREIBERG', 'LEIPZIG', 'FRANKENBERG']
['FREIBERG', 'FRANKENBERG']
['FREIBERG (Sachs)', 'FRANKENBERG']
```

Und do-while???

> Eine Nichtabweisschleife oder do-while-Schleife existiert in Python **nicht**! Aber wir könne mit einer while-Schleife jede andere Schleifenart nachbilden.



Für den folgenden Workaround nutzen wir eine Endlosschleife. Auch wenn Endlosschleifen manchmal unabsichtlich verursacht werden und zu Fehlverhalten führen, können sie manchmal erwünscht sein (z.B. *main loops* von Menüs oder graphischen Benutzeroberflächen). Wie in C++ können wir mit **break** eine Schleife dennoch abbrechen.

Wir bauen uns eine do-while Schleife nach:

```
while True:
  <anweisungen>

  if not <bedingung>:
    break
```

Ein Programmmenü als Typisches Beispiel:

```python
data = []
while True:
  print("-"*15, "\n 1: Daten eingeben\n 2: Summe ausgeben\n 3: Daten
    löschen\n 4: Beenden")
  opt = int(input("Wählen Sie eine Programmfunktion:"))

  if opt == 1:
    data.extend( float(x) for x in input("Zahlen mit Komma getrennt
      eingeben:").split(",") )
  elif opt == 2:
    print("Summe =", sum(data))
  elif opt == 3:
    data.clear()
  elif opt == 4:
    break
  else:
    print("Wählen Sie eine gültige Option!")
```

```
---------------
 1: Daten eingeben
 2: Summe ausgeben
 3: Daten löschen
 4: Beenden
Wählen Sie eine Programmfunktion:
```

Formatierte Ausgabe

Bislang haben wir uns mit dem "einfachen" `print` Mechanismus für einzelne Ausgabe zufrieden gegeben. Um aber gut lesbaren Code zu schreiben genügt dies nicht.

Wir erinnern uns an die von uns verwendenten C++ und Arduino-bezogenen Ausgabeformate ...

c++output.cpp

```cpp
#include <iostream>
#include <iomanip>

int main(){
  std::cout<<std::setbase(16)<< std::fixed<<55<<std::endl;
  std::cout<<std::setbase(10)<< std::fixed<<55<<std::endl;
  int a = 15;
  std::cout<<std::setbase(10)<< std::fixed<<55 + a <<std::endl;
  return 0;
}
```

```
37
55
70
```

ArduinoSerial.cpp

```cpp
void setup() {
    Serial.begin(9600);

    // Variante 1: print / println Methoden von Serial
    Serial.println("Das ist ein Test");

    float gleitkommaZahl = 3.123456;
    Serial.print(gleitkommaZahl, 4); //liefert 3.1234
    Serial.print(" ");
    Serial.println(gleitkommaZahl, 6); //liefert 3.123456

    Serial.print(15, BIN); //liefert den Wert 1111
    Serial.print(" ");
    Serial.print(15, HEX); //liefert den Wert f
    Serial.print(" ");
    Serial.println(15, OCT); //liefert den Wert 17

    // Variante 2: sprintf Formatierte Ausgabe
    int zahl = 65;
    char buffer[100];
    sprintf(buffer, "Dezimal %05i Dezimal %5i Hexadezimal %x Characte
      zahl, zahl , zahl, zahl);
    Serial.println(buffer);
}

void loop() {
}
```

```
Sketch uses 4668 bytes (14%) of program storage space. Maximum is 32256
bytes.
Global variables use 260 bytes (12%) of dynamic memory, leaving 1788
bytes for local variables. Maximum is 2048 bytes.


Das ist ein Test
3.1235 3.123456
1111 F 17
Dezimal 00065 Dezimal    65 Hexadezimal 41 Character A
```

> Die für den Arduino gezeigte Variante mit `sprintf` oder `printf` funktioniert natürlich auch unter C++ generell. Aus didaktischen Gründen wurde aber die `cout` Variante vorrangig genutzt.

Die Ausgabe in Python ist außerordentlich konfigurierbar. Man unterscheidet zwischen

1. der `printf` motivierten Ausgabe mit dem String modulo operator (`%`) und
2. den Ausgaben mit `format()`
3. den `f-strings`.

`print` folgt dem Muster `%[flags][width][.precision]type` für die Spezifikation der Ausgabe. Die zugehörige Verwendung unter Python stellt sich wie folgt dar:

prinf.py

```
1  print("Total students : %3d, Fak. 3 : %2d" % (80, 65))
2  print("Total students : %-5d, Fak. 3 : %05d" % (80, 65))
```

```
Total students :  80, Fak. 3 : 65
Total students : 80   , Fak. 3 : 00065
```

`f-strings` integrieren eigenen Code direkt in die Ausgabe - dies reicht von der einfachen Benennung einer Variablen, über Operationen mit Variablen bis hin zu Bedingungen und Funktionsaufrufen.

```
1  num1 = 83
2  num2 = 9
3  print(f"The product of {num1} and {num2} is {num1 * num2}.")
4
5  print(f"{num1} and {num2} are equal - {True if num1 == num2 else False
```

```
The product of 83 and 9 is 747.
83 and 9 are equal - False.
```

Quiz

Elemente der Programmiersprache

> Welche Programmierparadigmen unterstützt Python?

- [ ] Imperativ
- [ ] Prozedural
- [ ] Objektorientiert

Zuweisung und Datentypen

Der Variable `d` soll erstmalig der Wert `3.14` zugewiesen werden. Welche Umsetzung ist korrekt?

- ◯ `float d = 3.14;`
- ◯ `float d = 3.14`
- ◯ `d = 3.14;`
- ◯ `d = 3.14`
- ◯ `int d = 3.14;`

Sie haben der Variable `d` soeben erfolgreich `3.14` zugewiesen. Ist es möglich der Variable `d` nun einen Wert von einem anderen Datentyp zuzuweisen?

- ◯ Ja
- ◯ Nein

Wie können die Werte von den Variablen `x` und `y` in einer Zeile getauscht werden? Geben Sie die Lösung ohne Leerzeichen an.

Welche der folgenden Operatoren existieren in Python?

- [ ] `++`
- [ ] `+`
- [ ] `-`
- [ ] `/`
- [ ] `+=`
- [ ] `*=`
- [ ] `--`

Standard Datentypen

Welche dieser beispielhaft angegebenen Datentypen existieren in Python?

- [ ] tupel
- [ ] dictionary
- [ ] set
- [ ] list
- [ ] None
- [ ] float
- [ ] int
- [ ] string
- [ ] complex

Ausgabe

Mit welcher Funktion können Konstanten und Variablen in Python in der Konsole ausgegeben werden?

Mit welchem Parameter kann der standardmäßige Zeilenumbruch nach der Ausgabe mit `print` durch andere Zeichen ersetzt werden?

Eingabe

Mit welcher Funktion können String-Variablen in Python in der Konsole eingegeben werden?

Liefert dieses Programm einen Fehler?

```
x = input("x eingeben:")
y = x**2
print("y = ", y)
```

- ◯ Ja
- ◯ Nein

Zusätzliche Module einbinden

Mit welchem Schlüsselwort werden Module in Python eingebunden?

- ◯ `using`
- ◯ `import`
- ◯ `#include`

Wodurch muss `[_____]` ersetzt werden, um `Pi` aus dem eingebundenen Modul `math` auszugeben?

```
import math as m

print([_____].pi)
```

Wodurch muss `[_____]` ersetzt werden um `Pi` aus dem eingebundenen Modul `math` auszugeben?

```
from math import *

print([_____])
```

Listen

Wodurch muss `[_____]` ersetzt werden um die Länge der Liste `l` auszugeben?

```
l = ['r', 6, 8.0]
print([_____](l))
```

Wie lautet die Ausgabe dieses Programms? Geben Sie die Antwort ohne eckige Klammern oder Leerzeichen an.

```
liste = [1, 7]

liste.append(2)
liste.sort()
liste.reverse()
liste.extend([5,6])
liste.remove(1)

print(liste)
```

Listen-Indizierung

Wie lautet die Ausgabe dieses Programms?

```
data = ['a', 'b', 'c', 'd', 'e', 'f']
print(data[2])
```

Wie lautet die Ausgabe dieses Programms? Geben Sie die Antwort ohne eckige Klammern oder Leerzeichen an.

```
data = ['a', 'b', 'c', 'd', 'e', 'f']
l = []
l.extend(data[-2])
l.extend(data[3:])
l.extend(data[-4:])
l.extend(data[::2])
print(l)
```

Kontrollstrukturen

Selektion mit `if`

Wie viele `elif` Zweige darf es maximal in einer `if`-Anweisung geben?

- ◯ 0
- ◯ 1
- ◯ beliebig viele

Wie viele `else` Zweige darf es maximal in einer `if`-Anweisung geben?

- ◯ 0
- ◯ 1
- ◯ beliebig viele

Wodurch müssen die `[_____]` ersetzt werden?

```
a = 3.4
if a > 9[_____]
  print('A')
else[_____]
  print('B')

print('C')
```

Wie lautet die Ausgabe dieses Programms?

```
a = 3.4
if a > 9:
  print('A')
elif a > 2:
  print('B')
else:
  print('C')
```

Worauf muss bei verschachtelten `if`-Anweisungengen geachtet werden?

- ◯ Die innere Anweisung muss in eckigen Klammern stehen
- ◯ Alle beteiligten Anweisungen müssen einen `else`-Zweig enthalten
- ◯ Auf die richtige Einrückung aller Zweige der inneren Anweisung

Wie lautet die Ausgabe dieses Programms?

```
a = 3.4
if a > 9:
  print('A')
elif a > 2:
  if a < 3:
    print('B')
  else:
    print('C')
else:
  print('D')
```

`while`-Schleife

Wodurch muss `[_____]` ersetzt werden?

```
while True[_____]
  print('Und nochmal...')
```

`for`-Schleife

Wie lautet die Ausgabe dieses Programms?

```
for x in [7, 8, 9]:
  print(x, end=",")
```

Wie lautet die Ausgabe dieses Programms?

```
for x in "Hallo":
  print(x, end="o")
```

Was gibt die `range`-Funktion zurück?

- ◯ Den Abstand zwischen zwei Punkten
- ◯ Eine Liste von Integer-Werten
- ◯ Ein in die Liste von Integer-Werten konvertierbares `range`-Objekt

Welche Parameter der `range`-Funktion sind optional?

- ☐ start
- ☐ stop
- ☐ step

Wie lautet die Ausgabe dieses Programms?

```
for x in range(4):
  print(x, end=",")
```

Wie lautet die Ausgabe dieses Programms?

```
for x in range(1,5):
  print(x, end=",")
```

Wie lautet die Ausgabe dieses Programms?

```
for x in range(5,-1,-1):
  print(x, end=",")
```

Schleifen oder List Comprehension

Wie lautet die Ausgabe dieses Programms? Bitte geben Sie die Lösung ohne eckige Klammern oder Leerzeichen an.

```
a = [x*2 for x in range(5)]
print(a)
```

Wie lautet die Ausgabe dieses Programms? Bitte geben Sie die Lösung ohne eckige Klammern oder Leerzeichen an.

```
a = [0,1,2,3,4]
print([x*2 if x <= 2 else x for x in a])
```