

Python Grundlagen

Parameter	Kursinformationen
Veranstaltung:	<u>Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten</u>
Semester	<u>Wintersemester 2025/26</u>
Hochschule:	<u>Technische Universität Freiberg</u>
Inhalte:	<u>Grundlagen der Programmiersprache Python</u>
Link auf Repository:	https://github.com/TUBAF-lfl-LiaScript/VL_EAVD/blob/master/07_PythonGrundlagen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Florian Richter & Bernhard Jung



www.python.org

Fragen an die heutige Veranstaltung ...

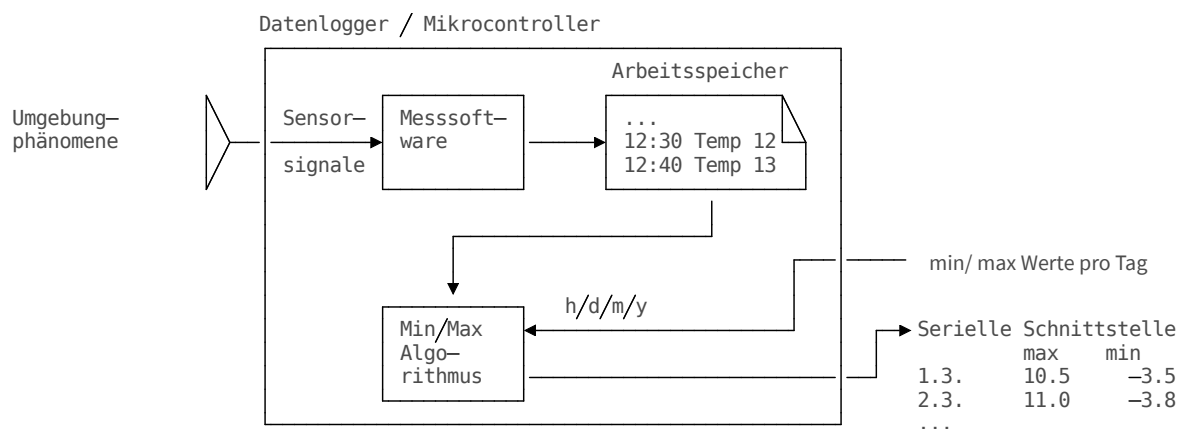
- Warum sollte man andere Programmiersprachen als C++ in Betracht ziehen?
- Welche Vorteile hat die Sprache **Python**?
- Wie unterscheidet sich die Syntax von C++?

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?



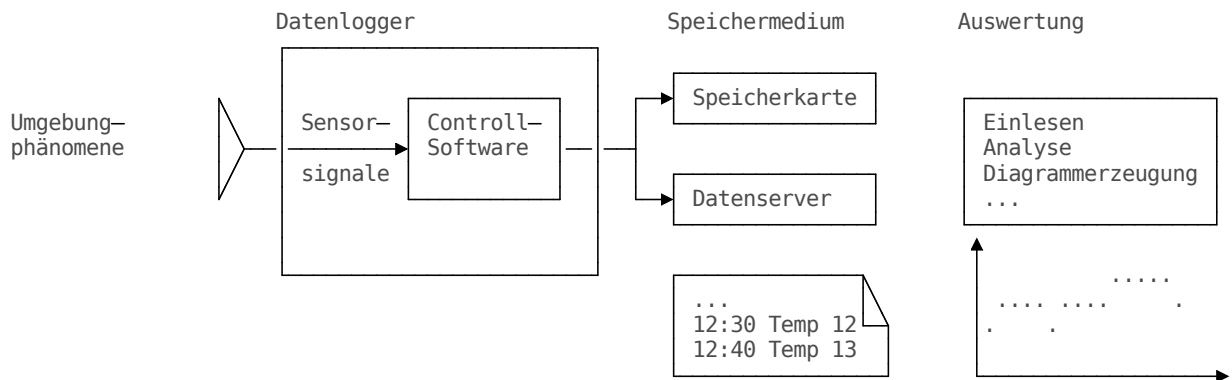
Frage: Welche Nachteile sehen Sie in dem Vorgehen?

- 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 einen Monat 17.280 Byte (16.8 kB) und für ein Jahr 210.240 Byte (205 kB) Speicherbedarf bedeuten. Unser beispielhafter MxChip-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.

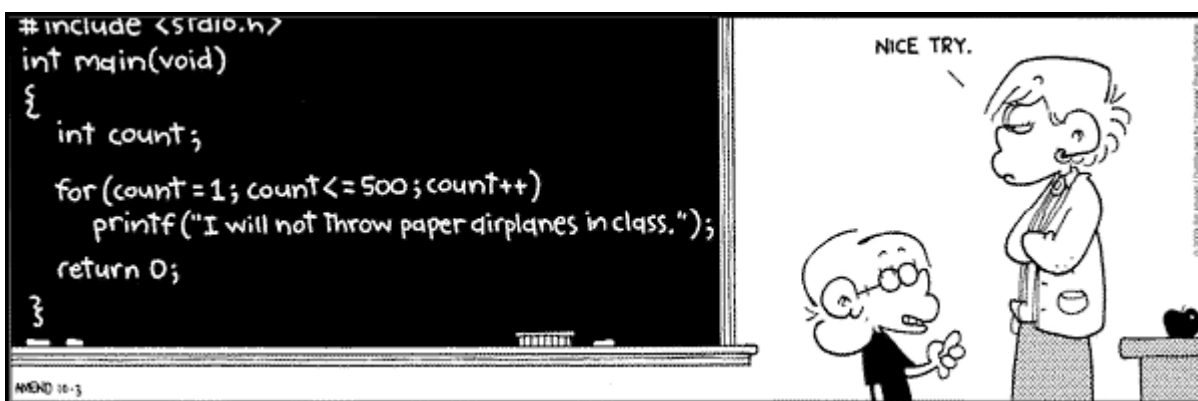
Lösungsansatz: Wir kommunizieren die Rohdaten in eine Cloud und verarbeiten diese separat.



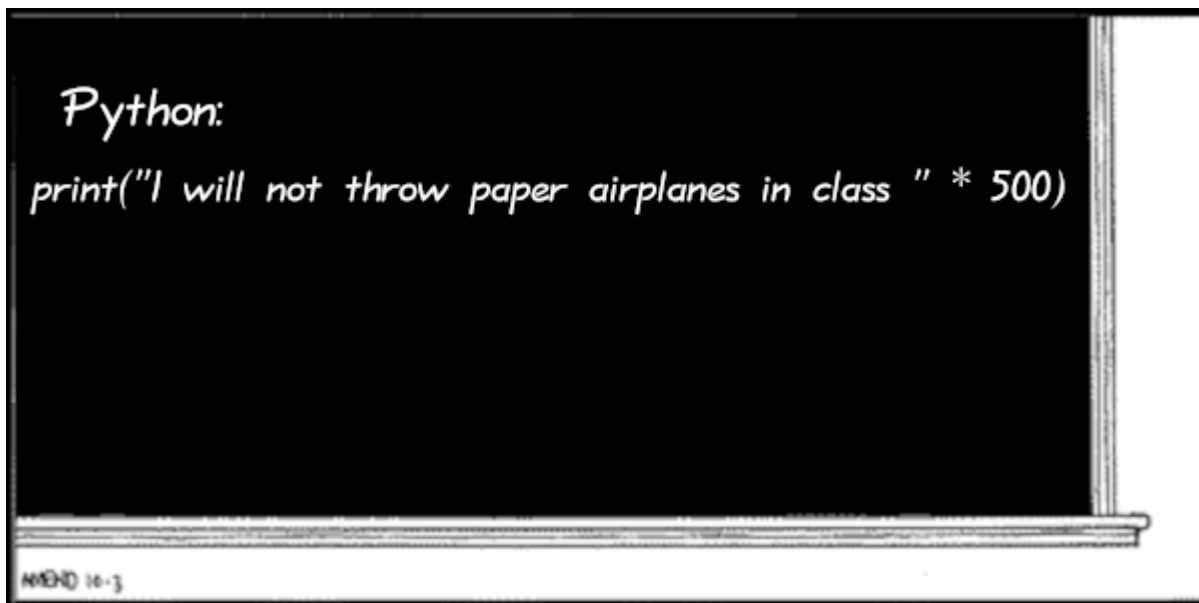
Wir entkoppeln damit die Datenerzeugung und Auswertung! Welche Programmiersprache benutzen wir aber für die Auswertung? Warum nicht einfach weiter C++?

Python erleichtert die schnelle Entwicklung von Scripten zur Datenauswertung.

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



C++



Python

ZumAusprobieren.py

```
1 print("Mit Python ist das Programmieren viel einfacher!")
2 print("Zu generelles Vorurteil! " * 3)
```

```
Mit Python ist das Programmieren viel einfacher!
Zu generelles Vorurteil! Zu generelles Vorurteil! Zu generelles
Vorurteil!
```

Python Historie

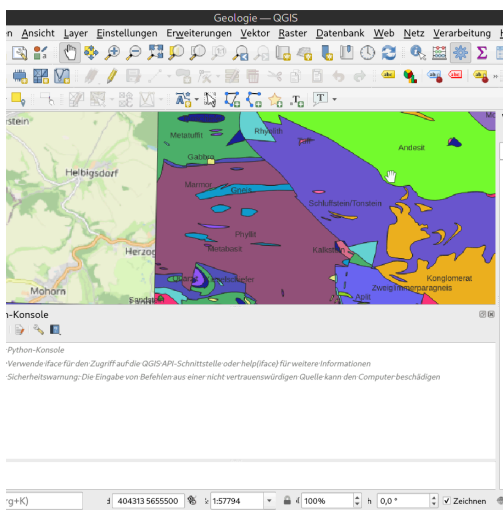
- *relativ* junge Sprache, Entwicklung 1990 in Amsterdam (vgl. C 1972, C++ 1979)
- Ziel sollte eine Sprache sein, die auch von Programmieranfängern einfach erlernt werden kann
- 2008 erschien Version 3.0 mit einer Komplettüberarbeitung und vielen Verbesserungen
- Aktuell sollte **mindestens Version 3.7** oder neuer verwendet werden (**3.14** ist die neueste Version)
 - Anaconda nutzt derzeit Version 3.13 [Link zur Dokumentation](#) [Link zum Download](#)

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 Entwickeln von Skripten 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 Skriptsprache 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

Demo: Unterschied zwischen Python und C++

Welche Programmierparadigmen kann ich damit abdecken?

Welche Programmierparadigmen unterstützt Python?

- Imperativ?
- Prozedural?
- Objektorientiert?

```
1 for i in range(3):
2     print("Hallo Welt", i)
```



```
Hallo Welt 0
Hallo Welt 1
Hallo Welt 2
Hallo Welt 0
Hallo Welt 1
Hallo Welt 2
```

Welche Programmierparadigmen unterstützt Python? → Alle 3 🤪

- Imperativ ✓
- Prozedural ✓
- Objektorientiert ✓

```
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)
```

Entwicklungsumgebungen

Wie installiere ich bei mir Python?

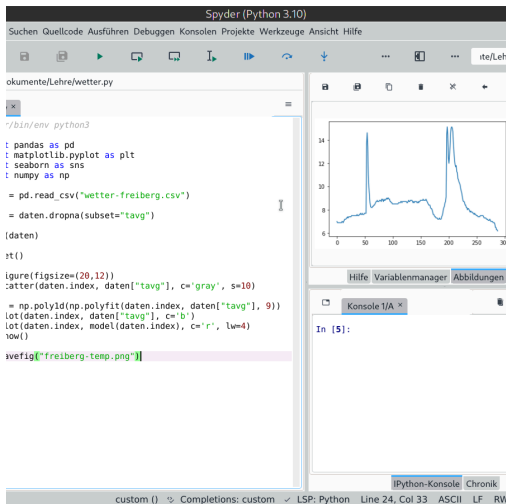
Unter Linux und MacOS kann Python z.B. über die Kommandozeile bzw. Paketmanager installiert werden. Für Windows empfiehlt sich die Installation der Anaconda Distribution (auch für Linux und Mac verfügbar).

[Link zum Anaconda Download](#)

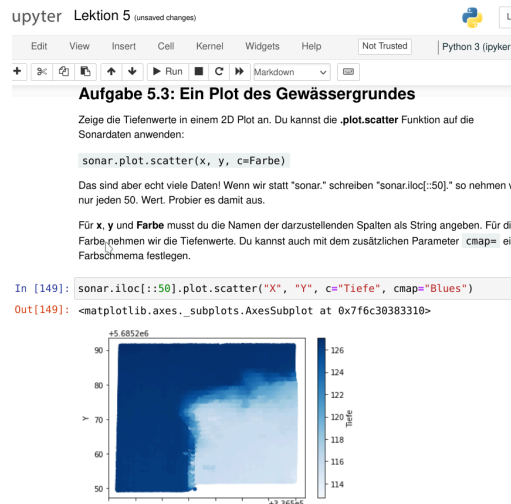
Wie arbeite ich bei mir Python?

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

Anaconda nutzt als Standard-Editor **Spyder** (Scientific **PY**thon **D**evelopment **E**nviRonment). Beliebte sind auch interaktive **Jupyter**-Dokumente (Notebooks).



Spyder Editor



Jupyter Notebook

Demo: Notebooks

Sprachelemente

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.

Zuweisungen_1.py

```
1 a = 1
2 a += 3
3 print(type(a))
4
5 a = "hello"
6 print(type(a))
```

```
<class 'int'>
<class 'str'>
<class 'int'>
<class 'str'>
```

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

Zuweisungen_2.py



```
1 x = 10
2 y = 5
3 x /= y
4 x += 8.5
5 x *= 4
6 print(x)
7
8 y=y**2    # Potenzieren mit **
9 print(y)
```

```
42.0
25
42.0
25
```

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, ㄚ
 - 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/>

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

Alles ist ein Objekt!

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()`, ...

```
1 liste = [10,20,50,200,300,350]
2 print( len(liste) )
3
4 liste.reverse()
5 print(liste)
6
7 liste.sort()
8 print(liste)
9
10 liste.append(7)
11 liste.extend([5,6])
12 print(liste)
13
14 liste.remove(300)
15 print(liste)
16
17 print( len(liste) )
```



```

6
[350, 300, 200, 50, 20, 10]
[10, 20, 50, 200, 300, 350]
[10, 20, 50, 200, 300, 350, 7, 5, 6]
[10, 20, 50, 200, 350, 7, 5, 6]
8
6
[350, 300, 200, 50, 20, 10]
[10, 20, 50, 200, 300, 350]
[10, 20, 50, 200, 300, 350, 7, 5, 6]
[10, 20, 50, 200, 350, 7, 5, 6]
8

```

Auch C++ offeriert ähnliche "Container" wie die Python Liste. Diese offerieren auch ein OOP API. Hierauf sind wir aber aus Zeitgründen nicht eingegangen.

Negative Indices und Slices

Python unterstützt etwa im Gegensatz zu C++ auch **negative Indices** bei Listen, Tupeln und Strings. Z.B. ist -1 der letzte Eintrag, -2 der vorletzte Eintrag, usw. Außerdem sind **Slices** möglich, d.h. es können Teilbereiche extrahiert werden. Dazu werden Start- und Endindex sowie eine Schrittweite mit einem Doppelpunkt getrennt angegeben.

slice.py

```

1  a_list = list(range(0, 10))
2  a_string = "mydata.csv"
3
4  print(a_list)      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5  print(a_list[-1])  # 9
6  print(a_list[2:10]) # [2, 3, 4, 5, 6, 7, 8, 9]
7  print(a_list[0:10:2]) # [0, 2, 4, 6, 8]
8  print(a_list[::2])  # [0, 2, 4, 6, 8]
9
10 print(a_string[-4:]) # .csv ...letzte 4 Zeichen
11 print(a_string[:-4]) # mydata
12 print(a_string[::])  # mydata.csv
13 print(a_string[::-1]) # vsc.atadym

```

```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
9
[2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[0, 2, 4, 6, 8]
.csv
mydata
mydata.csv
vsc.atadym
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
9
[2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
[0, 2, 4, 6, 8]
.csv
mydata
mydata.csv
vsc.atadym

```

Eingabe und Ausgabe

Ausgabe

Die **print**-Funktion ist unser Äquivalent zu `std::cout` aus C++. Beliebige 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.

printAusgabe.py

```

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

```

```

5 * 5 = 25
x      =      3.141
x^2    =      9.865881
5 * 5 = 25
x      =      3.141
x^2    =      9.865881

```

Mit dem optionalen Parameter `end=` können Sie den standardmäßigen Zeilenumbruch 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:

inputFehler.py



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

```
x eingeben:x eingeben:
```

Richtig:

inputRichtig.py



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

```
x eingeben:x eingeben:
```

Ausgabe in Dateien, Lesen aus Dateien

`print` kann auch zum Schreiben in Dateien verwendet werden. Zu beachten ist, dass jedes `print`-Statement (normalerweise) eine Zeile schreibt, d.h. den eigentlichen Ausgabertext gefolgt vom Zeilenendezeichen `\n`.

```
with open('mydata.txt', 'w') as f: # 'w': open for writing
    print('Hallo', file=f)         # writes 'Hallo\n'
    print('Welt', file=f)          # writes 'Welt\n'
```



Die Funktion `open` öffnet eine Datei, in diesem Fall ('w') zum Schreiben. Existiert die Datei noch nicht, wird die Datei neu erzeugt. Das `with`-Statement ist eine Besonderheit von Python und sorgt dafür, dass die Datei nach Ende des eingerückten Blocks, hier also den beiden `print`-Befehlen, auch wieder geschlossen wird.

Geöffnete Dateien sollten nicht nur der guten Ordnung halber wieder geschlossen werden. Jede offene Datei benötigt etwas Arbeitsspeicher. Wenn sehr viele Dateien offen sind, kann dies den Computer verlangsamen. Sind sehr, sehr, sehr viele Dateien offen, kann es zum "Einfrieren" kommen, d.h. der Computer reagiert überhaupt nicht mehr. Nutzen Sie daher **with open(...)** **as** zum Öffnen (und automatischen Schließen) von Dateien.

Eine Textdatei kann wie folgt zeilenweise (jede Zeile einschließlich des Zeilenendezeichens '\n') eingelesen werden:

```
with open('mydata.txt', 'r') as f: # 'r': open for reading
    for line in f:
        print(line)          # print line to the console
        #print(line[:-1])    # print line without its last character, i.e. '\n'
```

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.

```
#include <cmath>

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

importBeispiel.py

```
1 import math
2
3 x = math.sin(math.pi) # math.pi ist eine Konstante
4                       # math.sin ist eine Funktion des Moduls math
5                       # Checken Sie das Ergebnis!
6 print(x)
7
8
9 #---- Formatiertes Beispiel:
10 print(f"{x:.5f}")
```

```
1.2246467991473532e-16
0.00000
1.2246467991473532e-16
0.00000
```

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

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

Eine Umbenennung ist alternativ auch möglich:

```
1 import math as m
2 print(m.pi)
```

```
3.141592653589793
3.141592653589793
```

Im obigen Beispiel ist uns eine numerische Ungenauigkeit von Python (und anderen Programmiersprachen) aufgefallen: der Sinus von π ist nicht exakt 0! Bei der Überprüfung, ob ein Float einen bestimmten Wert annimmt, ist die Funktion `isclose` nützlich.

```
1 import math
2 x = math.sin(math.pi)
3 print("x =", x)
4 print("is x exactly 0.0?", x == 0)
5 print("is x approximately 0.0?", math.isclose(x, 0, abs_tol=1e-10))
```

```
x = 1.2246467991473532e-16
is x exactly 0.0? False
is x approximately 0.0? True
x = 1.2246467991473532e-16
is x exactly 0.0? False
is x approximately 0.0? True
```

Kontrollstrukturen

Wiederholung

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

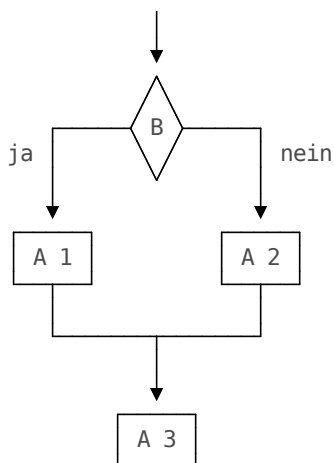
Beispiel: Wenn bei einer Flächenberechnung ein Ergebnis kleiner Null generiert wird, erfolgt eine Fehlerausgabe. Sonst wird im Programm fortgefahren.

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.

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 Bedingung:
    Anweisung1
else:
    Anweisung2

Anweisung3
```





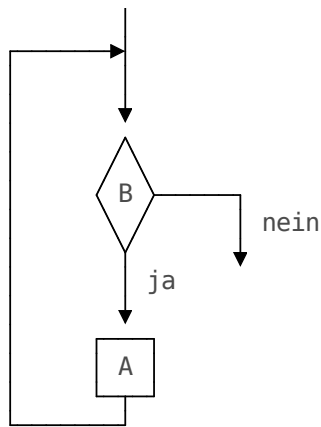
```
1 x=5
2 y=3
3
4 # Variante 1:
5 if x == y:
6     print("x and y are equal")
7 elif x < y:
8     print("x is less than y")
9 else:
10    print("x is greater than y")
11
12 # Variante 2:
13 if x == y:
14     print("x and y are equal")
15 else:
16     if x < y:
17         print("x is less than y")
18     else:
19         print("x is greater than y")
```

```
x is greater than y
x is greater than y
x is greater than y
x is greater than y
```

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

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

fibonacci.py



```
1 a, b = 0, 1
2 while a < 100:
3     print(a, end='|')
4     aux = a + b
5     a = b
6     b = aux
7     # analoger Einzeiler a, b = b, a+b
```

```
0|1|1|2|3|5|8|13|21|34|55|89|0|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:

```

1 ▾ for i in ['a', 'b', 'c']:
2     print(i, end=",")
3
4 ▾ for i in "abc":
5     print(i, end=",")
6
7 ▾ for i in [5, 7, 9]:
8     print(i, end=",")
9
10 ▾ for i in [3.4, 7.9, 6.78]:
11     print(i, end=",")
12

```

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

rangeBeispiel.py

```

1 x = range(0,5)
2 print(x, type(x))
3 print( list(x) )

```

```

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

```

Beispiele mit der range-Funktion:

```

1 ▾ for x in range(5):
2     print(x+1, end=" ")
3     print("\n")
4
5 ▾ for x in range(1, 6):
6     print(x, end=" ")

```

```

6     print(x, end=" ")
7     print("\n")
8
9     for x in range(20, -1, -2):
10         print(x, end=" ")
11     print("\n")

```

```

1 2 3 4 5

1 2 3 4 5

20 18 16 14 12 10 8 6 4 2 0

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 erinnert etwas an mathematische Mengendefinitionen, wie $\{x^2 \mid x \in \mathbb{N} \wedge x < 20\}$.

listcomprehension.py



```

1 squares = [x**2 for x in range(20)]
2 print(squares)
3
4 # alternative Darstellung mit einer Schleife
5 squares = []
6 for x in range(20):
7     squares.append(x**2)
8 print(squares)

```

```
[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]
[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 *List Comprehensions* stattdessen auf das konzentrieren, was Sie in die Liste umsetzen wollen.

Beispiele:

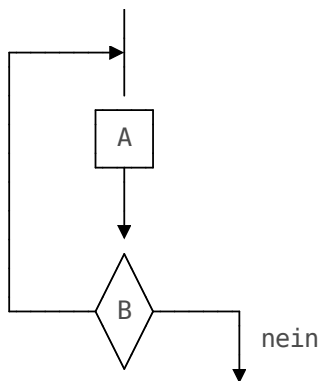
listcomprehension2.py

```
1 squares = [x**2 for x in range(20)]
2 print(squares)
3
4 cities = ["Dresden", "Freiberg", "Leipzig", "Frankenberg"]
5 print( [c.upper() for c in cities] )
6
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
14         cities if c[0] == "F"] )
15
16 # Im Unterschied zum letztgenannten Ausdruck die korrespondierende Sc
17 new_cities = [] # leere Liste
18 for c in cities:
19     if c[0] == "F":
20         if c != "Freiberg":
21             new_cities.append(c.upper())
22         else:
23             new_cities.append(c.upper()+" (Sachs)")
24 print(new_cities)
```

```
[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']
['FREIBERG (Sachs)', 'FRANKENBERG']
[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']
['FREIBERG (Sachs)', 'FRANKENBERG']
```

Und do-while???

Eine Nichtabweisschleife oder do-while-Schleife existiert in Python **nicht**! Aber wir können 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:

```
1 data = []
2 while True:
3     print("-"*15, "\n 1: Daten eingeben\n 2: Summe ausgeben\n 3: Daten
      löschen\n 4: Beenden")
4     opt = int(input("Wählen Sie eine Programmfunktion:"))
5
6     if opt == 1:
7         data.extend( float(x) for x in input("Zahlen mit Komma getrennt
      eingeben:").split(",") )
8     elif opt == 2:
9         print("Summe =", sum(data))
10    elif opt == 3:
11        data.clear()
12    elif opt == 4:
13        break
14    else:
15        print("Wählen Sie eine gültige Option!")
```

```
-----
1: Daten eingeben
2: Summe ausgeben
3: Daten löschen
4: Beenden
Wählen Sie eine Programmfunktion:-----
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 verwendeten C++ bezogenen Ausgabeformate ...

c++output.cpp



```
1 #include <iostream>
2 #include <iomanip>
3
4 int main(){
5     std::cout<<std::setbase(16)<< std::fixed<<55<<std::endl;
6     std::cout<<std::setbase(10)<< std::fixed<<55<<std::endl;
7     int a = 15;
8     std::cout<<std::setbase(10)<< std::fixed<<55 + a <<std::endl;
9     return 0;
10 }
```

```
37
55
70
37
55
70
```

Die für C Varianten mit `sprintf` oder `printf` funktioniert auch unter C++. 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:

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

f-strings bieten auch vielfältige Möglichkeiten zur formatierten Ausgabe von Zahlen.

fstrings.py

```
1 print(f'{123:>7}')          # '    123'
2 print(f'{123:=+7}')         # '+    123'
3 print(f'{123456789:_}')     # '123_456_789', Tausendertrennzeichen
4 print(f'{44252:X}')         # 'ACDC', Hexadezimalformat in Großbuchstaben
5 print(f'{0.9267:.1%}')      # '92.7%', Prozentformat, eine Nachkomma-Stelle
6 print(f'{3.14159:.3f}')     # '3.142', Festkommaformat, drei Nachkomma-Stellen
```

```
    123
+    123
123_456_789
ACDC
92.7%
3.142

    123
+    123
123_456_789
ACDC
92.7%
3.142
```

#

Quiz

Welche Programmierparadigmen unterstützt Python?

- ☐ Imperativ
- ☐ Prozedural
- ☐ Objektorientiert

Sprachelemente

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

Eingabe / 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?

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([_____])
```



Alles ist ein Objekt

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`-Anweisungen 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])
```

