Modul: Programmierung B-EPI
Einführung in die Programmierung EPR
WS 2021/2022

VE02 Kontrollstrukturen: Schleifen, Funktionen und Prozeduren mit unmutable Parametern

Prof. Dr. Franziska Matthäus / Prof. Dr. Matthias Kaschube / Dr. Karsten Tolle
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Übersicht

Schleifen und iterative Grundstrukturen

- Prinzipien
- Realisierungen von Schleifen in Python:
  - while- und for-Schleifen
  - `range()`- Funktion
  - break – continue – pass

Strukturierung von Programmen

- Prinzipien
- Funktionen und Prozeduren
- Namensräume (Sichtbarkeit)
- Realisierungen in Python:
  - def
  - Parameterübergabe für unmutable Typen

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Kontrollstrukturen - Prinzipien

Verzweigungen (letzte Woche) bilden zusammen mit den **Schleifen** und den **Unterprogrammen** die **Kontrollstrukturen** moderner Programmiersprachen.

In allen imperativen und objektorientierten Sprachen sind sie in unterschiedlichen Ausprägungen vorhanden.

Leistungsfähige Schleifenkonstrukte (zusammen mit Unterprogrammmethoden) sind essentielle Konstituenten der **strukturierten Programmierung** (dritte Programmiersprachen-Generation) und auch der **Objektorientierten Programmierung**.

Bem.: Es gibt auch Programmiersprachen, die ohne Schleifen auskommen! Bei Funktionalen Programmiersprachen (Haskell) werden Schleifen durch Rekursion (kommt später) realisiert!
… oder auch Bosque von Microsoft (2019) https://www.microsoft.com/en-us/research/project/bosque-programming-language/

Schleifen: Realisierungsformen der Iteration

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Eine Schleife wiederholt einen Teil des Codes – den so genannten **Schleifenrumpf** oder **Schleifenkörper** – so lange, bis eine **Abbruchbedingung** eintritt.

Schleifen, die ihre Abbruchbedingung niemals erreichen oder Schleifen, die keine Abbruchbedingungen haben, nennen wir **Endlosschleifen**.

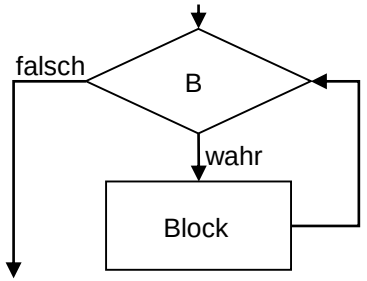
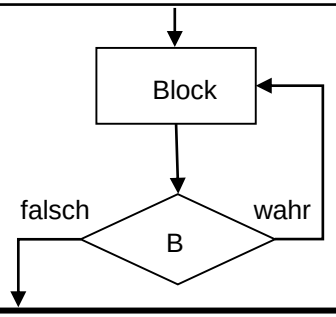
Schleifenarten (Grundformen)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

- die **kopfgesteuerte** oder **vorprüfende** Schleife, bei der erst die Abbruchbedingung geprüft wird, bevor der Schleifenrumpf durchlaufen wird meist durch das Schlüsselwort **WHILE** (= *solange-bis*) angezeigt.
- die **fußgesteuerte** oder **nachprüfende** Schleife, bei der nach dem Durchlauf des Schleifenrumpfes die Abbruchbedingung überprüft wird, z.B. durch ein Konstrukt **REPEAT**-**UNTIL** (= *wiederholen-bis*).
- die **Zählschleife**, eine Sonderform der kopfgesteuerten Schleife, meist als **FOR** (= *für*)-Schleife implementiert inkrementiert einen Zählindex i für jeden Durchlauf des Schleifenrumpfes.
- die **foreach-Schleife,** eine Sonderform für Sequenzdatentypen: Meint: ”für jedes Element in der Sequenz führe “Block” genau einmal aus.”

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Schleifen: Realisierungsformen der Iteration (1)

Schleifenart	Ablaufplan
vorprüfend (Kopfgesteuert) **while** (B) **do** Block	
nachprüfend (Fußgesteuert) **repeat** Block **until** (B)	

✓ **in Python**

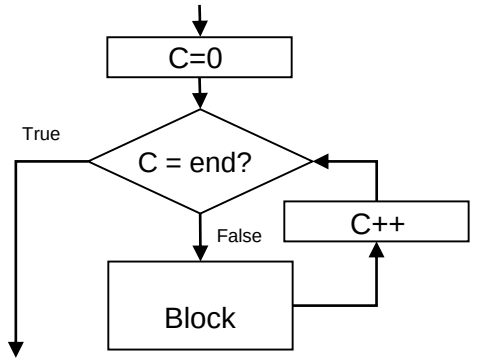
„Block“ wird ggf. nie ausgeführt.

Hier wird „Block“ mindestens einmal ausgeführt nicht in Python

```
count = 0
while count < 4:
    print("Welcome to Python")
    count += 1
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Schleifen: Realisierungsformen der Iteration (2)

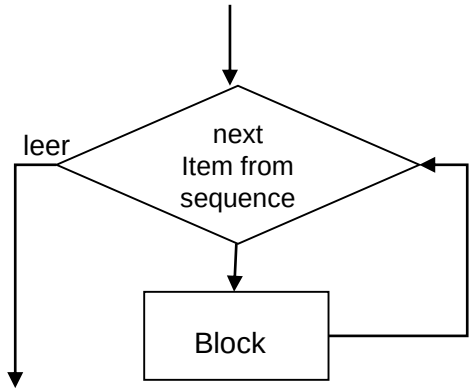
Schleifenart	Ablaufplan
Zählschleife Die Schleifenvariable C beginnt bei 0. C wird bei jedem Durchlauf inkrementiert Wenn C = end ist, bricht die Schleife ab	

✓ in Python mit **for - Schleife** und zusätzlicher Funktion **range ()**

```
for count in range(5):
    print("Welcome to Python")
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Schleifen: Realisierungsformen der Iteration (3)

Schleifenart	Ablaufplan
foreach - Schleife **for** <Iterationsvariable> **in** sequenz do Block Meint:"für jedes Element in der Sequenz führe "Block" genau einmal aus."	

✓ **in Python**

Wir kennen bisher erst einen Sequenzdatentyp: **String,** aber es kommen noch einige hinzu.

```
for c in "Hallo Welt":
    print(c)
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Historische Notiz (1)

Noch in den 60er-Jahren waren Sprunganweisungen (GOTO <Sprungziel>) in Programmen üblich, was bei größeren Programmen manchmal nahezu zur Unwartbarkeit führte, da sie schnell kaum noch überschaubar wurden.

Das „GOTO <Sprungziel>“ ist eine direkte Abbildung des Maschinenbefehls „JUMP <Adresse des Sprungziel>“ – lediglich musste das Sprungziel jetzt keine Programmadresse mehr sein, sondern konnte symbolisch als Zahl oder Name angegeben werden.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Historische Notiz (2)

Schon im Mai 1966 publizierten **Böhm und Jacopini** einen Artikel, in dem sie zeigten, dass **jedes Programm**, das GOTO-Anweisungen enthält, in ein GOTO-freies Programm umgeschrieben werden kann, das nur mit Verzweigung (IF <Bedingung> THEN ... ELSE ...) und einer Schleife (WHILE <Bedingung> DO xxx) arbeitet (gegebenenfalls unter Zuhilfenahme von etwas Codeduplikation und der Einführung von booleschen Variablen (True/False)).

Im März 1968 veröffentlichte *Edsger W. Dijkstra* seinen legendären Aufsatz „**Go To Statement Considered Harmful**“

Aber auch Schleifen bereiten Probleme! Siehe Heise Developer „Weg mit den Schleifen“ (2018) online unter: https://www.heise.de/developer/artikel/Weg-mit-den-Schleifen-4009774.html

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Kontrollstrukturen in Python - Syntax

while-Anweisungen = kopfgesteuerte Schleife

```
while_stmt  ::=  "while" expression ":" suite
                 ["else" ":" suite]
```

Dies ist übrigens eine Syntaxdefinition in erweiterter BNF (**Backus-Naur-Form** oder **Backus-Normalform)**, direkt lesbar, oder? –
Betrachten wir in GPR später noch genauer.

Das “**while-Statement**” benutzt man zur wiederholten Ausführung der “suite” (des Schleidenrumpfes), solang die “expression” zu “True” ausgewerted wird.

In „suite“ muss der „Wahrheitswert“ von „expression“ geändert werden, sonst entsteht eine Endlosschleife.

Wenn die “expression” zu “False” ausgewertet wird, wird die “suite” des optionalen else-Zweiges ausgeführt, sofern vorhanden.

While-Schleife - erste Beispiele

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

```
i = 0
while i < 4:
    print(i) #suite
    i += 1   #suite
```

```
0
1
2
3
>>>
```

```
i = 0
while i < 3:
    print(i)
    i += 1
else: print('finit')
```

```
0
1
2
finit
>>>
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Häufige Probleme …

- Bau einer Endlosschleife, z.B.:

Vergleich wie „i != 3“ sollte vermieden werden.
Sollte im Schleifenrumpf dieser konkrete Wert nicht getroffen werden (siehe rechts), kann daraus schnell eine unbeabsichtigte Endlosschleife entstehen.

```
i = 0
while i < 3:
    print(i)
    i += 1
else:
    print('finit')
```

```
i = 0
while i != 3:
    print(i)
    i += 1
else:
    print('finit')
```

```
i = 0
while i != 3:
    print(i)
    i += 1
    i += 1
else:
    print('finit')
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Beispiel Endlosschleife

```
while True:
    i = 0
```

Was macht man?

Rechner herunterfahren und neu booten? NEIN!

„Control-C“ drücken, das ist der `KeyboardInterrupt`

Menuepunkt „Restart"

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

„Abbrechen“ von Schleifen(-durchläufen)

Um eine **Schleife** ohne Erreichen der Bedingung (expression == False) zu **verlassen (beenden)**, verwendet man die

- **break-Anweisung**.

Um „nur“ in die **nächste Schleifeniteration zu springen** (also, den Rest des Schleifenrumpfes (der suite) zu überspringen) verwendet man die

- **continue-Anweisung**.

Beide Statements dürfen überall im Schleifenrumpf stehen, aber nicht außerhalb einer Schleife.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

„Abbrechen“ von Schleifen(-durchläufen)

```
i = 0
while i < 4:
    i += 1
    if i == 2: break  #
    print(i)
else: print('finit')
```

```
1
>>>
```

```
i = 0
while i < 4:
    i += 1
    if i == 2: continue  #
    print(i)
else: print('finit')
```

```
1
3
4
finit
>>>
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Ein „Nichtstuer“

- **pass**-Anweisung
- Macht gar nichts – geht zur nächsten Anweisung weiter.
- Kann benutzt werden, wenn die Syntax eine Anweisung erfordert, doch die Programmlogik benötigt keine Aktion.
- **Während der Code-Entwicklung als Platzhalter** (`# TODO remember to implement this`).
- Kann überall stehen, wo ein Statement erwartet wird.

```
while i < 4:   # Ist so alleine nicht gültig
    #  TODO remember to implement this
```

```
while i < 4:   # ... das geht
    pass
    #  TODO remember to implement this
```

Nutzung der pass-Anweisung

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

```
i = 42
pass
print(i)
```

```
42
>>>
```

```
i = 42
if i == 42:
    pass
else: print(i)
```

```
>>>
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

In Python for- Anweisung

for_stmt ::= "for" target_list "in" expression_list ":" suite
["else" ":" suite]

Das for_stmt kann relativ kompliziert werden: Wir betrachten zunächst einen einfachen Fall, die **Variante 1 (**die "for each" Schleife):

- In target_list steht als **target ein Name** (identifier) **als sogenannte Schleifenvariable**
- In expression_list muss dann stehen: **ein** iterierbarer Typ (ein ***iterable*** – alle Sequenztypen: List, Tupel, **String**) aber auch Sets, Frozensets, Dictionaries. ... alle Objekte, die ihre Elemente eins nach dem anderen hergeben können und speziell dafür die Methode (Funktion) `__next__()` haben.

```
for c in "Hallo Welt":
    print(c)
else:
    print("fertig")
```

Mit range () wird die „foreach-Scheife“ zur Zählschleife

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Die foreach-Schleife wird im Zusammenhang mit der **range-Funktion** (als virtuelle „expression“) zu einer **Zählschleife**

```
range([start,]stop [,step])
```

```
for x in range([start,]stop [,step]):
```

range liefert (virtuell) eine Sequenz von aufeinander folgenden Integern zwischen *start* und *stop-1*.

Mit nur einem Parameter (*stop*) ergeben sich die Zahlen **0**...stop-1.

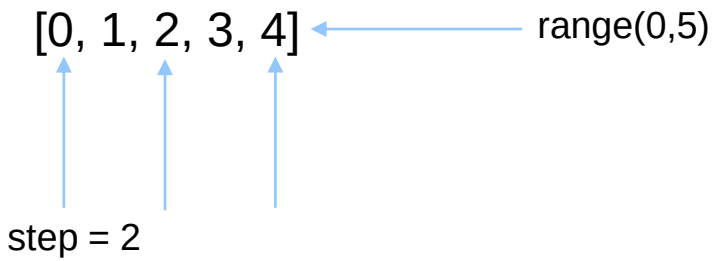
step ist optional und ist die Schrittweite – kann negativ sein.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Beispiele zur Zählschleife mit `range([start,]stop [,step])`

```
for x in range(0,5,2):
    print(x)
```

```
0
2
4
>>>
```



Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Die Funktion `range()` erzeugt, eine "virtuelle Sequenz"

- Der Iterator muss mit __next__() das nächste Element der virtuellen Sequenz liefern und dann, wenn keins mehr vorhanden ist eine
- StopIteration exception (Ausnahme) wird ausgelöst, wenn das Ende erreicht ist

- Das kann auch ein Programm, man muss also die Sequenz nicht vollständig erzeugen und speichern, sondern man kann das nächste Element "on-demand" liefern.

Vorteile von range():

es wird garantiert, dass es auch terminiert → keine Endlosschleife
mit dem Aufruf ist auch die Zählweise klar → einfacher zu lesen

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Beispiele zur Zählschleife mit `range([start,]stop [,step])`

```
for x in range(6,-5,-2):
    print(x)
```

```
6
4
2
0
-2
-4
>>>
```

```
x = 6
while x > -5:
    print(x)
    x = x-1
    pass
    x = x-1
```

Als while-Schleife wäre es fehleranfälliger!

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Ausblick (denn auch das führt hier zu weit):

for_stmt ::= "for" target_list "in" expression_list ":" suite
["else" ":" suite]

```
namensliste = [('Max','Meyer'), ('Marlene', 'Müller')]

for vorname, nachname in namensliste:
    print(vorname, nachname)
```

erzeugt:

```
Max Meyer
Marlene Müller
```

- Die Anzahl der Elemente in expression list muss genau der in target-list entsprechen.
- Die Anzahl der Elemente in den expressions muss gleich groß sein.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Grundsätzlich: Endlosschleifen sind eine Gefahr!

Können **nur** von „außen“ unterbrochen werden durch ein Zurücksetzen (engl. *reset*), durch eine Unterbrechung (engl. *interrupt*), durch Ausnahmen (engl. *exeptions*), Abschalten des Gerätes oder ähnliches (Achtung, sind dann streng genommen kein Algorithmus).

Manchmal ist eine Endlosschleife aber gewollt: Z.B. bei Programmen zur Überwachung und Reaktion auf einen externen (gemeldet z.B: durch einen Interrupt) oder internen Fehlerzustand (gemeldet durch eine Exception)!

Der else-Zweig, break, continue, pass bei der for-Schleife

verhalten sich genauso wie bei while.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Wofür braucht man Schleifen?
Iterative Grundstrukturen (1)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Die **Iteration** (von lateinisch *iterare*, "wiederholen"; engl. *iteration*) ist ein **grundlegender Lösungsansatz** sowohl in der Mathematik als auch der Informatik mit zwei verschiedenen Ausprägungen:

1. **Ausgrägung:**
   Iteration (in Mathematik und Informatik) ist eine Methode, sich der Lösung eines „Rechenproblems“ schrittweise, **aber zielgerichtet** anzunähern. Sie besteht in der **wiederholten Anwendung desselben Rechenverfahrens.**

   Meistens iteriert man mit Rückkopplung: Die Ergebnisse eines Iterationsschrittes (oder alle bisher erzielten Ergebnisse) werden als Ausgangswerte des jeweils nächsten Schrittes genommen - bis das Ergebnis zufrieden stellt (z.B. sich in der dritten Nachkommastelle nicht mehr ändert, d.h. man hat aus Tausenstel genau gerechnet.)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Beispiele

- Summen $$a = \sum_{i=0}^{n} a_i = a_0 + a_1 + a_2 + \ldots + a_n$$

  Beispiel: Die natürliche Exponentialfunktion e^x wird durch ihre Taylorreihe mit Entwicklungsstelle 0 dargestellt:

  $$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, \; -\infty < x < \infty$$ konvergiert

- Produkte $$a = \prod_{i=0}^{n} a_i = a_0 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_n$$

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Iterative Grundstrukturen (2)

2. Ausprägung:

In der Informatik wird **auch** von Iteration gesprochen, wenn ein **schrittweiser Zugriff auf Einzelelemente eines zusammengesetzten Datentyps (Sammlungen, in Python Sequenztypen),** z.B. auf jedes Zeichen eines Strings, erfolgt.

Z.B String ist ein "iterierbare" Datenstruktur. Es gibt die Methode `__next__()`. Nicht iterierbar sind Integer oder Float.

In einer Iteration mit dem String `s = "otto"` kann auf die einzelnen Buchstaben zugegriffen wird:

z.B. in dem String “otto“ sollen alle Kleinbuchstaben durch Großbuchstaben ersetzt werden: “OTTO“.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Zusammenfassung

Schleifen sind Kontrollstrukturen (und ermöglichen die Iteration).

In Python gibt es while- und for-each Schleifen. Zusammen mit der Funktion range() auch Zählschleifen.

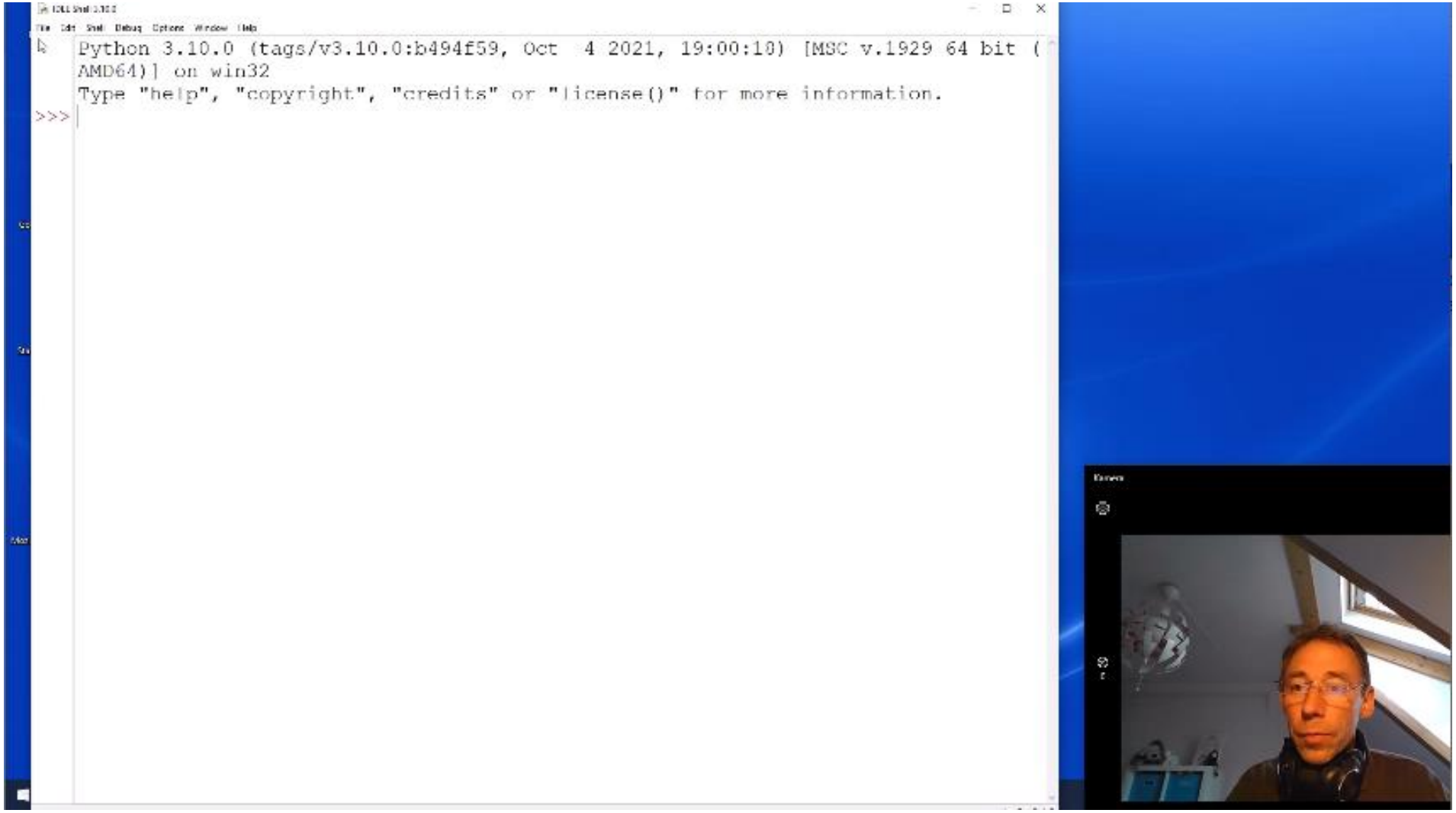
*Kontrollstrukturen verändern den **Programmfluss.***

Wir haben diese Konzepte eingeführt

... die Praxis, das Programmieren damit müssen Sie jetzt üben!

.

Video – Python live



Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Übersicht

Schleifen und iterative Grundstrukturen
- Prinzipien
- Realisierungen von Schleifen in Python:
  - while- und for-Schleifen
  - `range()`- Funktion
  - break – continue – pass

Strukturierung von Programmen
- Prinzipien
- Funktionen und Prozeduren
- Namensräume (Sichtbarkeit)
- Realisierungen in Python:
  - def
  - Parameterübergabe für unmutable Typen

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Prozeduren – Funktionen – Methoden
Grundsätzliche Ziele - Motivation

Seit der Frühzeit des Computings werden Unterprogramme (engl. *subroutines*) eingesetzt, um verschiedene Ziele zu erreichen:

- **eine bessere, übersichtlichere Strukturierung von Programmen**
- zur Abstraktion (**was** gemacht wird muss klar sein, aber nicht genau wie es gemacht wird!)
- zur "Modularisierung", also **wiederbenutzbare** Programmteile einmal schreiben und pflegen
- bei mehrfacher Verwendung zum Einsparen von (Programm-)Speicherplatz, also mit dem Ziel, den Code möglichst kompakt zu halten.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Unterprogramm (Prinzip, allgemein)

Durch ein Unterprogramm wird eine Folge von Anweisungen, unter einem **Namen** zusammengefasst.

Die **Definition** eines Unterprogramms erfolgt in Python durch:

```
def <name>(<parameter>):
        <suite>
        [return <value>]
```

Der `return`-Zweig ist optional.

Es können **Parameter** an diese Folge übergeben, und ggf. auch *ein Wert* oder mehrere Werte *zurückgeliefert* werden, mit `return <value>`.

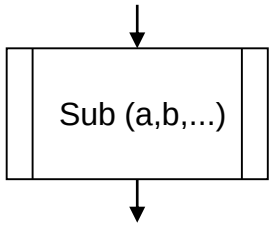
Die Parameter werden in der Regel durch Reihenfolge, Typ und Anzahl und/oder durch Namen festgelegt.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Graphische Repräsentationen

Erst Sub **definieren (x,y,…)** ← **formale** Parameter

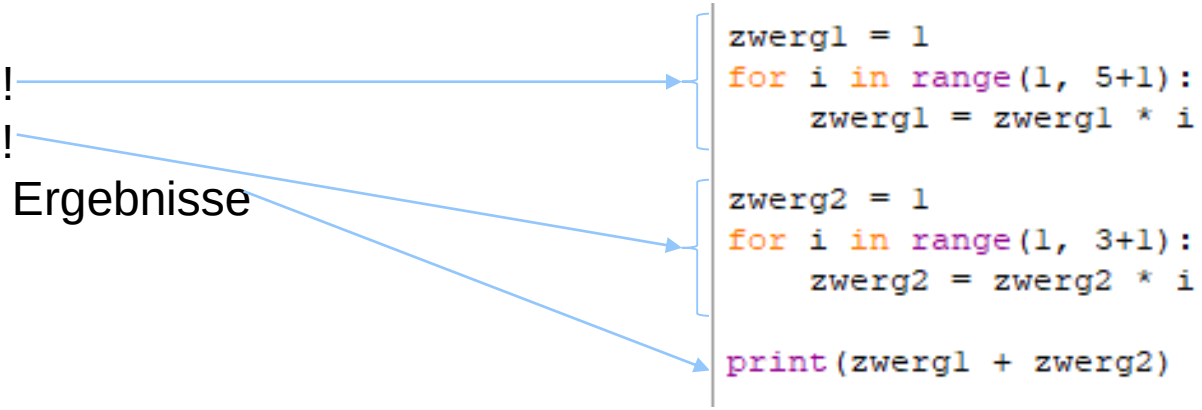
Dann Sub **aufrufen (a,b,...)** ← **aktuelle** Parameter

Pseudocode	**Ablaufplan**
Call Sub (a,b,...) oder Sub (a,b,...)	

Ein **Unterprogramm** Sub ist ein "normales" Programm, an das man Werte (= aktuelle Parameter) übergeben kann und das selbst Werte zurückgeben kann, als Funktionswert.

Aufgabe: Berechne 5! + 3!

1. Berechne 5!
2. Berechne 3!
3. Addiere die Ergebnisse



```
zwergl = 1
for i in range(1, 5+1):
    zwergl = zwergl * i

zwerg2 = 1
for i in range(1, 3+1):
    zwerg2 = zwerg2 * i

print(zwergl + zwerg2)
```


Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Aufgabe: Berechne 5! + 3!

1. Berechne 5!
2. Berechne 3!
3. Addiere die Ergebnisse

Unterprogramm zur Berechnung

```
def fak(n):
    fak = 1
    for i in range(1, n+1):
        fak = fak * i
    return fak

print(fak(5) + fak(3))
```

Vorteile:

- Code nicht redundant
- Kann beliebig wiederverwendet werden

→ Vermeidung von Fehlern:

- Gibt es einen Fehler und wird er korrigiert, muss er nur an einer Stelle korrigiert werden
- Durch eine breitere Verwendung fallen Fehler eher auf

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Wir unterscheiden: Funktionen versus Prozeduren

- **Funktionen** erzeugen (errechnen) einen Wert, der an das rufende Programm als Wert der Funktion zurückgegeben wird, wie in der Mathematik üblich, z.B. **sin(30°)=** 0.5.
  - Der Wert 0.5 ersetzt also den Ausdruck **sin(30°).**
  - Damit kann eine Funktion u.a. in Ausdrücken als Entität auftreten: `a = 1-sin(x)`. Funktionen werden typischerweise in Bibliotheken (Modulen) thematisch (Mathematik-Funktionen) gebündelt.
  - **Haben (optionale) Eingabeparameter und einen Ausgabewert**
- **Prozeduren** führen eine Aktion aus. Hierdurch können entweder interne Variablen verändert werden, eine Zustandsänderung erwirkt werden z.B. ein `print()`.
  - **Haben (optionale) Eingabeparameter und keinen Ausgabewert**

Dr. Franziska Matthäus
Dr. Matthias Kaschube
arsten Tolle

Beispiele in Python – Die Definition

```
def <name>(<parameter>):
        <suite>
        [return <value>]
```

Funktionen, Prozeduren (und Methoden) werden mit der **def-Anweisung** definiert.

```
def spam(x, y):
    print(x, y)
    return x // y
```

Die Liste der formalen Parameter darf auch leer sein!

Achtung: Der **Funktionsrumpf** muss eingerückt (*indent*) werden (macht IDLE nach dem : automatisch); das Ende der Funktionsdefinition wird durch Rücknehmen der Einrückung (*dedent)* angegeben (macht IDLE bei Eingabe einer Leerzeile auch automatisch - aufpassen!!!)

`return` ist das Schlüsselwort, das veranlasst, dass der Wert ‚x // y' als Funktionswert dem „spam(x,y)" zugewiesen **und die Funktion beendet wird.**
`return` ist optional (ohne return haben wir eine Prozedur).
`return` kann auch mehrfach vorkommen.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

return -Anweisung

Gibt **einen Wert** aus der Funktion zurück und beendet die Ausführung des Unterprogramms.

Wird kein Wert angegeben oder wird die `return` -Anweisung weggelassen, so wird das **Objekt None** zurückgegeben – genau das nennen wir dann allgemein Prozedur (und nicht Funktion).

Python unterscheidet formal aber nicht zwischen Prozeduren und Funktionen … es sind alles "*functions*".

Es können in einer Funktion mehrere `return` stehen.

Um mehrere Werte zurückzugeben, setzt man diese in ein Tupel ...
Sehen wir später.

```
>>> x = print()

>>> print(x)
None
```

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Aufruf einer Funktion

Die **Funktionsdefinition** `def` muss im Programmtext **vor dem Aufruf** erfolgen. (Erst dann ist der Name und Art der Funktion bekannt.)

Ihrem Namen werden die Argumente in „runden Klammern“ unmittelbar nachgestellt, wie in

```
def add(x, y):
    print(x, y)
    return x + y
```

```
>>> a = 1 + add(3, 4)
3 4
>>> a
8
```

Die **Reihenfolge und Anzahl von** Argumenten (**Parametern**) müssen bei **Positionsparameter** (solche haben wir hier) mit jenen der Funktionsdefinition übereinstimmen.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Die aktuellen Parameter dürfen Ausdrücke (expressions) sein.

```
def add(x, y):
    print(x, y)
    return x + y
```

```
a = 8 + 2
>>> add(a, 8 // 4)
10 2  # durch print()verursacht
12  #dies ist der Funktionswert von add(a, 8 // 4)
```

Offensichtlich werden die Werte der Ausdrücke berechnet und dann an dem formalen Parametern zugewiesen ???
Die Identifikation erfolgt entsprechend der Position! 1➔ 1, 2 ➔ 2, …

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Aufruf mit Parameter-Name Keyword Arguments und Default-Werte

- Die Parameter eine Funktion können über den Namen der Parameter angesprochen werden und Default-Werte haben! …

```
def auto_preis(breite, laenge, sitze=5, marke=None):
    print('b: ', breite, ' l: ', laenge, ' s: ', sitze, 'm: ', marke)
    preis = sitze*breite*laenge
    if (marke == "BMW"):
        preis = preis * 2
    return preis
```

```
>>>
>>> auto_preis(
              (breite, laenge, sitze=5, marke=None)
```

```
>>> print(auto_preis(2, 3))
breite:  2  laenge:  3  sitze:  5 marke:  None
30
>>> print(auto_preis(laenge=3, breite=2))
breite:  2  laenge:  3  sitze:  5 marke:  None
30
>>>
```

Sieh auch: https://www.w3schools.com/python/gloss_python_function_keyword_arguments.asp

Aufruf mit Parameter-Name Keyword Arguments

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

- … und bei range()

```
>>> range(
        range(stop) -> range object
        range(start, stop[, step]) -> range object
```

```
>>> range(stop=10, step=3, start=2)
```

Ist eine Ausnahme, da die build_in Funktionen in C implementiert sind
und daher anders reagieren☹

Aufruf mit Parameter-Name Keyword Arguments

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

- Die Parameter eine Funktion können über den Namen der Parameter angesprochen werden!
  - … z.B. bei print()

```
>>> print(
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

```
>>> x= 'Hallo Welt'
>>> y = ':-)'
>>> print(x, y, sep='TRENNZEICHEN')
Hallo WeltTRENNZEICHEN:-)
>>>
```

Ln: 755 Col:

Mechanismen zur Parameterübergabe (klassisch)

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Wir unterscheiden:

- **Wertparameter (*call by value*)**
- Referenzparameter (*call by reference*)
- Namensparameter (*call by name*)
  (Hat nur noch historische Bedeutung! – wurde in Algol 60 und Cobol genutzt.)
- Es gibt neuere Konzepte, wie **"call-by-Object",** was z.B. auch in Python genutzt wird – lernen wir später noch genau.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Eigenschaften von Call by Value "allgemein"

	Wertparameter "call by value"
Formale Parameter	Einfache Variablen und strukturierte Variablen
Aktuelle Parameter	Beliebige Ausdrücke wie *1.0*, *2*X*, *sin(x)*, y[i]
Übergabe	Als *Wert**kopie** an den formalen Parameter* (hoher Aufwand bei großen Datenstrukturen)
Zuweisung an Parameter innerhalb des Unterprogramms	möglich oder verboten
Rückgabe eines Wertes an den aktuellen Parameter bei Unter-programmende	nein **Echte Kapselung!** Schutz vor Fehlerausbreitung!

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

In Python: Genau hinschauen, wo und wann etwas passiert!

```
def add(a, y):  # UP
    print(a, y)
    a = 5  #Zuweisung an
           #formalen Parameter
    return a + y

a = 4  # Hauptprogramm
print(a, add(a, 5))
```

```
4 5
4 10
>>>
```

- Man darf in Unterprogrammen auch zu formalen Parametern Zuweisungen machen.
- Diese werden dann aber für das Unterprogramm zu lokalen Variablen.
- Im rufenden Programm haben wir hierdurch keine Auswirkungen.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Wie macht nun Python die Parameterübergabe genau (1)?

Python nutzt einen „Zwitter“! -- "Call-by-Object" (manchmal auch "Call by Object Reference" oder "Call by Sharing" genannt).

1) Wenn man an **unveränderliche** *(unmutable)* aktuelle Parameter (wie Integers, Strings, **kurz alle Typen, die wir bisher kennen**) an eine Python-Funktion übergibt, verhält sich die Übergabe wie eine **Wertübergabe (wie bei call by value).**

   Die Referenz auf das **unveränderliche** (unmutable) Objekt wird an den formalen Parameter der Funktion übergeben. Innerhalb der Funktion kann der Inhalt des Objektes nicht verändert werden (dann würde ja ein neues Objekt unter gleichem Namen entstehen).

2) Wenn man **veränderliche** *(mutable)* aktuelle Parameter hat, dann …

   Solche „mutable“ Variablen lernen wir erst später kennen. In dem Zusammenhang behandeln wir dann auch deren Parameterübergabe.

Verhalten bei Argumenten (aktuellen Parametern), die *unmutable* sind:

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

```
>>> def spiel(a, y):
        a = 'hallo'
        print(a, y)
        return y
```

```
>>> a = 10
>>> spiel(a, 8%3)
hallo 2
2
>>> print(a)
10
```

- Offensichtlich wurde **a** im rufenden Programm **nicht verändert.**
- … verhält sich wie ein **call by value**.
- Wir haben eine **echte Kapselung!**

- Die Variablen der a in Prozedur und a im rufenden Programm sind echt verschieden,
  haben einen anderen Kontext.

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Zusammenfassung: Funktionen in Python

Wir erreichen eine echte Kapselung für unmutable Datentypen:
String, Float, Integer, Boolean

- Namen dürfen innerhalb und außerhalb von Unterprogrammen gleich sein: Variablen sind trotzdem nicht die selben!
- Werte von Namensgleichen Variablen innerhalb und außerhalb eines Unterprogramms (Funktion, Prozedur) sind verschieden.
- Realisiert ein Call by Value

Die Rückgabe eines Wertes erfolgt nur durch Funktionen die mit
return <value>
einen Wert zurückgeben.

Was macht man nun, wenn man mehrere Werte zurückgeben will? … siehe später mit anderen Datentypen!

Zusammenfassung Schleifen und Unterprogramme

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

Als weitere Kontrollstrukturen haben wir kennengelernt: while- und for-Schleifen

- Wir haben kennengelernt Positionsparameter für unmutale Typen!
- Es gibt noch andere Übergabemechanismen in Python:
  - Python verhält sich anders bei mutable Datentypen
  - Python kennt noch weitere Varianten für die Parameterübergabe: Defaultwerte, Keyword-Parameter, unbekannte Anzahl der Parameter) … *lernen wir später*

Auf jeden Fall die zwei Quizzes machen!

Prof. Dr. Franziska Matthäus
Prof. Dr. Matthias Kaschube
Dr. Karsten Tolle

- Es wird jetzt immer wichtiger, Ihre Programme zu strukturieren.
- Denken Sie daran: Programme werden viel häufiger gelesen als geschrieben. Dann sollte aber der Stil möglichst einheitlich sein.
- Hierzu gibt es Hinweise im Programmier-Handbuch (Style Guide) Angaben zu unseren Konventionen: Das sind nur Vereinbarungen unter ProgrammiererInnen, **aber bindend für die Übungen!**
- Im wesentlichen Auszüge aus PEP 8.
  Unser READING in der nächsten Woche!
  D.h. unbedingt einmal anschauen.

... und, danke für Ihre Aufmerksamkeit!