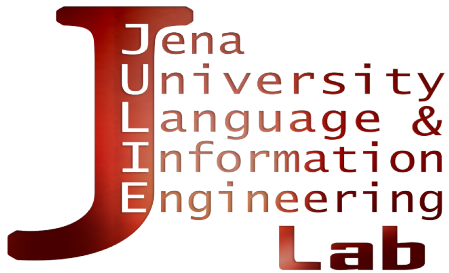
Software-Praktikum: Softwaretechnologien für Natürlichsprachliche Systeme

Sven Büchel

Jena Language & Information Engineering (JULIE) Lab
Friedrich-Schiller-Universität Jena, Germany

https://julielab.de/

Sommersemester 2019



Algorithmik und Programmierung

- Einleitung
- Variablen
- Datentypen und Operationen
- Funktionen
- Kontrollstrukturen
- Rekursion

Abschnitt 1

Algorithmik und Programmierung

Unterabschnitt 1

Einleitung

Ziele

- Kein Programmierkurs
  (keine Anwendungsentwicklung, nur minimale technischen Hintergründe, ...)
- Keine Einführung in die Informatik
- Lesen und Schreiben von Algorithmen
- Lauffähiger Code
- Grundlage für Verständnis computerlinguistischer Methoden

Algorithmen

- Abfolge *gültiger* Anweisungen
- I.d.R. zur Lösung eines Problems. Z.B. ...
  - Wortarten erkennen
  - Syntaxbaum berechnen
  - Emotion eines Satzes bestimmen
- I.d.R zur Ausführung durch Computer bestimmt
  - Gegenbeispiel: schriftliches Addieren

Programme

- Abfolge gültiger Anweisungen einer konkreten Programmiersprache (z.B. C, Java, Python) (Quellcode)
  Implementierung eines abstrakten Algorithmus
- Durch Computer ausführbar
- Derselbe Algorithmus kann in unterschiedlichen Programiersprachen implementiert werden
- In diesem Kurs nutzen wir **Python** 3

Hello, World!

```
1 print('Hello, World!')  # Ausgabe: Hello, World!
```

- Der `print`-Befehl erzeugt eine Ausgabe auf dem Bildschirm
- Das Doppelkreuz '#' markiert **Kommentare**. Dieses und alles rechts davon in einer Zeile wird vom **Interpreter** ignoriert.

Sequenzielle Bearbeitung

Ein Programm wird von oben nach unten, Zeile für Zeile ausgeführt.

```
1 print('Hello, World!')
2 print(42)
3 print(2+2)
```

Ausgabe:

```
Hello, World!
42
4
```

Unterabschnitt 2

Variablen

Variablen im Mathematikunterricht

- Variablen sind sind Symbole, die verschiedene Werte einnehmen können
- $f : \mathbb{R} \to \mathbb{R}$, $f(x) = x^2 + 5$
- Zuweisung verbal z.B. “Sei x gleich 7”.

Variablen in der Informatik

- Variablen als “benamte Zeiger”
- Referenzieren Werte im Speicher
- Zugriff über den Variablennamen

```
1  # Variablenzuweisungen
2  x = 5
3  y = 7
4  z = 8
5  print(y)   # Ausgabe 7
6  # Rechnen mit Variablen
7  x = 3 * z
8  x = x / 2
9  z = x + y
10 print(z)   # Ausgabe 19
```

Werte und Variablen

Variablennamen

- Dürfen Großbuchstaben, Kleinbuchstaben, Zahlen und den Unterstrich enthalten (keine Sonderzeichen, kein Leerzeichen)
- Müssen mit Buchstabe oder Unterstrich anfangen (i.d.R klein)
- Dürfen ansonsten beliebig ausgewählt werden
- Sollten knapp und präzise bezeichnen, was referenziert wird
  - gut: `data`, `results`, `input_text`
  - weniger gut: `dskjfhal`, `var7`, `Baumhaus`
- Mehrere Wörter werden per Konvention mit Unterstrich (`variable_name`) oder Binnenmajuskel (`variableName`) getrennt

Wertezuweisung

```
1  # Variablen wird mit "=" Werte zugewiesen
2  x = 5
3
4  # Dies darf jedoch nicht als Gleichheit
       verstanden werden, wie folgedes Beispiel
       zeigt
5  x = x + 1   # Der Wert von x wird um 1 erhoeht
6  print(x)    # Ausgabe 6
```

Unterabschnitt 3

Datentypen und Operationen

Motivation

- In der Schule nehmen Variablen meist nur Zahlenwerte (ganze, reelle, komplexe, usw.)
- In der Computerlinguistik verarbeiten wir jedoch Sprache und brauchen daher (auch) andere “Arten von Werten” (**Datentypen**)
- Datentypen unterscheiden sich hinsichtlich ...
  - des erlaubten Wertebereichs (“Haus” ist keine Zahl) und
  - der erlaubten Operationen (4 + 3 vs. 4 + “Haus”)

Für die Übung relevante Datentypen

- Wahrheitswerte (`booleans`)
- Zahlen (`integer, floats`)
- Zeichenketten (`strings`)
- Listen (`lists`)
- `dictionaries`

Anders als in vielen anderen Programmiersprachen werden Datenypen in Python nicht explizit festgelegt und sind veränderlich (dynamisches Typensystem).

```
1  var1 = 5   # Java: int var1 = 5;
2  var2 = 7
3  print(var1 + var2)   # Ausgabe 12
4  var1 = "Haus"   # geaenderter Variablentyp
5  print(var1 + var2)   # Fehler
```

Wahrheitswerte (Booleans)

- Nehmen entweder den Wert `True` oder `False` an
- Aussagenlogische Operatoren:
  - und (`and`)
  - oder (`or`)
  - nicht (`nicht`)

Wahrheitswerte (Booleans) II

```
1 var1 = True
2 var2 = False
3 print(var1)
4 print(var1 and var2)
5 print(var1 or var2)
6 print(not var1)
7 print(not (var1 or var2))
```

Exkurs: Aussagenlogik

Semantik (Bedeutung) der aussagenlogischen Operatoren in Form einer Wahrheitswert-Tabellen

a	b	`not` a	a `or` b	a `and` b
0	0	1	0	0
0	1	1	1	0
1	0	0	1	0
1	1	0	1	1

Exkurs: Aussagenlogik II

- Wahrheitswert-Tabelle zur Bestimmung des Wahrheitswerts komplexer Aussagen in Abhängigkeit des Wahrheitswerts von Elementaraussagen (logischer Atome) an

a	b	`not` a	(`not` a) `or` b
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1

Exkurs: Übung zur Aussagenlogik

Ermitteln Sie den Wahrheitswertverlauf der komplexen Aussagen “(a `and` b) `or` b” mithilfe einer Wahrheitswerttabelle.

Exkurs: Übung zur Aussagenlogik (Lösung)

a	b	a `and` b	(a `and` b) `or` b
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

Zahlen

- Die wichtigsten Zahlentypen in Python sind `Integer` (Ganzzahlen) und `Floats` (Fließkommazahlen)
- Seit Python 3 ist der Unterschied zwischen beiden für den Einsteiger eher irrelavant
- Grundrechenarten (`+ - * /`) funktionieren jetzt für beide Datentypen im Ergebnis gleich

```
1  x = 9   # Integer
2  y = 4   # Integer
3  print(x / y) # Ausgabe 2.25 (Float)
4  # In Python 2 waere hier '2' ausgegeben worden.
5  x = 9.0   # Float. Achtung, Punkt statt Komma!
6  y = 4.0   # Float
7  print(x / y) # Ausgabe 2.25 (Float)
```

Vergleichsoperatoren

- Werden zu Booleans ausgewertet
- Gleicheit ==
- Kleiner <
- Größer >
- Kleinergleich <=
- Größergleich >=

```
1   print(4 == 4)
2   print(4 > 4)
3   print(4 <= 4)
```

Einige weiterführende Zahlenoperatoren

```
1  # Potenzen
2  print(4**4) # Ausgabe 256
3
4  # Zuweisung mit Addition
5  x = 4
6  x += 1   # wie x = x + 1
7  print(x)
8  # genauso -=, *= und /=
9
10 # Division mit Rest
11 print(9 % 8)
```

Zeichenketten (Strings)

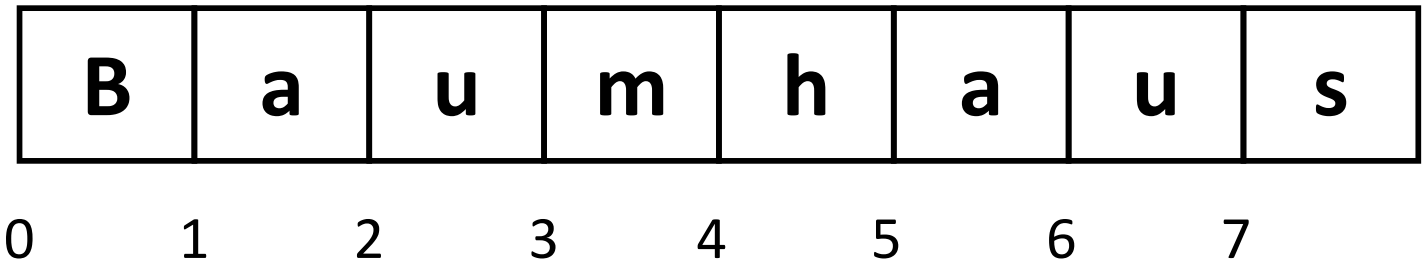
```
1  # Strings werden mit Anfuehrungszeichen
       ausgezeichnet (doppelt oder einfach)
2  x = "Baum"
3
4  # Strings, die Zahlen enthalten sind von den
       Zahlwerten an sich zu unterscheiden
5  x = "8"
6  y = 9
7  print(x + y) # Fehler
```

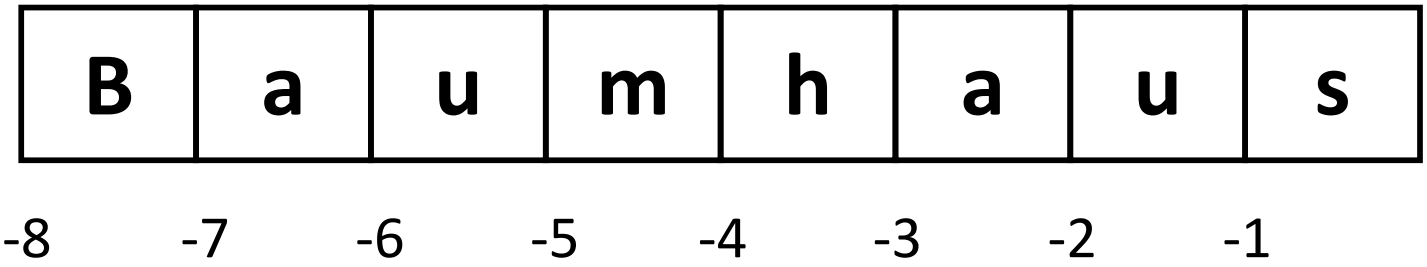
Verarbeiten von Strings

```
1  # Konkatenation (Verbinden) von Strings
2  print("Baum" + "Haus")
3
4  # Laengen-Funktion
5  a = "Baumhaus"
6  b = len(a)
7  print(b)
```

Strings: Indexzugriff

- Strings sind Folgen von einzelnen Zeichen
- Auf diese kann durch den jeweiligen **Index** (“Stellenbezeichner”) einzeln zugegriffen werden
- Vorwärtsindexierung (beginnend mit 0) vs. Rückwärtsindexierung (beginnend mit -1)





Strings: Indexzugriff

- Beim Indexzugriff wird durch die Syntax `string_name[n]` auf das *n*-te Zeichen des Strings zugegriffen
- Für nachfolgende Stringoperationen (insbesondere Slicing) hilft es sich vorzustellen, dass die Indizes “zwischen” den Stellen stehen und beim normalen Indexzugriff immer das Element rechts davon abgerufen wird (gilt für Vorwärts- und Rückwärtsindexierung)

```
1   x = "Computerlinguistik"
2   print(x[0])   # "C"
3   print(x[8])   # "l"
4   print(x[-1])   # "k"
```

Übung zur Stringindexierung

Schreiben Sie ein Programm, dass den String `Computerlinguistik ist toll!"` in der Variable `Meinung` speichert. Erzeugen Sie die Variablen `a, b, c, d` die jeweils das erste Zeichen der Worte in `Meinung` sowie das Interpunktionszeichen speichert. Anschließend soll das Programm den Wert von `Meinung` mit dem String `"Wirklich!"` konkatenieren und dies dann ausgeben. Stellen Sie sicher, dass das ausgegebene Endergebnis orthografisch korrekt ist.

Übung zur Stringindexierung (Lösung)

```
1 Meinung = "Computerlinguistik ist toll!"
2 a = Meinung[0]
3 b = Meinung[19]
4 c = Meinung[23]
5 d = Meinung[-1]
6 #print(a,b,c,d)
7 print( Meinung + " Wirklich!")
```

Strings: Slicing

- Um längere Teilketten (Substrings) abzurufen, kann man sogenanntes **Slicing** verwenden.

```
1  some_string = "Computerlinguistik"
2
3  # Allgemeine Syntax: string[beginIndex:endIndex]
4  print(some_string[1:4]) # "omp"
5
6  # Der erste bzw. letzt Index kann weggelassen
       werden, wenn der Teilstring vom Anfang bzw.
       bis zum Ende geht.
7  print(some_string[:8]) # Computer
8  print(some_string[8:]) # linguistik
```

Listen

- Eine Liste ist eine Datenstruktur, die mehrere unterschiedliche Werte verschiedenen Datentyps speichern kann
- Syntaktisch durch eckige Klammer `[]` markiert. Einzelne Elemente werden mit Kommata abgegrenzt.
- Listen können Listen enthalten!
- Erlaubte Operationen weitgehend identisch zu String (Konkatenation, `len`, Indexzugriff, Slicing)
- Listen können nur mit Listen (nicht einzelnen Elementen) konkateniert werden. Um einzelne Elemente anzuhängen, kann die `append`-Methode verwendet werden (s. nächste Folie).

Erzeugen und Manipulieren von Listen

```
1  l = [] # leere Liste initialisieren
2  l = [1, "1", True, "Computerlinguistik"] # Liste
       mit Elementen initialisieren
3  print(l[0]) # 1
4  print(l[1]) # "1"
5
6  l = l + ["ist", "toll"] # Konkatenation von
       Listen
7  l.append("!") # Syntax zum anhaengen einzelner
       Elemente
8  print(l[-1]) # "!"
9
10 # Slicing von Listen
11 print(l[-4:]) # ["Computerlinguistik", "ist", "
       toll", "!"]
```

Listen — Achtung: Referenzkopie vs. Wertekopie

```
1  list1 = [1,2,3]
2  list2 = list1
3  list2.append(4)
4  print(list1[-1])   # 4 !!!
```

Übung zu Slicing und Listen

Schreiben Sie ein Programm, dass den String `Computerlingustik ist toll"` in einer Variable speichert. Erzeugen Sie eine leere Liste. Verwenden Sie Slicing, um die einzelnen Wörter im obigen Satz als einzelne Listenelemente zu speichern. Lassen Sie die Länge der Liste ausgeben.

Übung zu Slicing und Listen (Lösung)

```
1  s = "Computerlinguistik ist toll"
2  liste = [s[:18], s[19:22], s[23:]]
3  # print(liste)
4  print(len(liste))
```

Zerlegungen und Verbinden von Strings bzw. Listen

```
1 # split-Methode: Listen aus Strings erzeugen
2 s = "Dies ist ein Satz"
3 l = s.split(" ")
4 print(l) # ["Dies", "ist", "ein", Satz"]
5
6 # join-Methode um Strings aus Listen
     zusammenzufuegen
7 l = ["Dies", "ist", "ein", "Satz"]
8 s = " ".join(l)
9 print(s)  # "Dies ist ein Satz"
```

Dictionaries

- Menge (ungeordnet) von Schlüssel-Wert-Paaren (Key-Value)
- Angabe des Schlüssel erlaubten direkten Zugriff auf Wert
- Häufig werden `int`- und `String`-Datentypen als Schlüssel verwendet

```
1  # Neues dictionary erstellen
2  d = {1: "Mo", 2:" Di", 3: "Mi", 4: "Do",
3      5: "Fr", 6: "Sa", 7: "So"}
4
5  # Wert durch einen Schluessel abrufen
6  print(d[1])       # Montag
7  print(d.get(1))   # Alternative, gibt None wenn
       Schluessel nicht enthalten
8
9  # Neues Schluessel-Wert-Paar hinzufuegen
10 d[8] = "Fanatsietag"
```

Dictionaries — Linguistische Beispiele

```
1  # Beispiel mentales Lexikon
2  lexicon = {"werfen": {"1sg": "werfe",
3                        "2sg": "wirfst"},
4             "gehen": {"1sg": "gehe",
5                       "2sg": "gehst"}
6             }
7  print("ich " + lexicon["werfen"]["1sg"])  # "ich
       werfe"
8
9  # Beispiel haeufigste Wortart
10 most_freq_pos = {"der": "det",
11                  "ein": "det",
12                  "Baum": "NN",
13                  "fahren": "V"}
```

Unterabschnitt 4

Funktionen

Grundlegende Bemerkungen

- Die bisherigen Programme sollten “einzigartige” Probleme lösen und waren zum einmaligen Ausführen gedacht
- Ein wesentliches Konzept der Informatik ist dagegen, Probleme in leichter zu lösende Teilprobleme zu zerlegen
- Hierzu sollte aus Effizienzgründen auf schon bekannten Teillösungen aufgebaut werden können
- Einfaches “Rüberkopieren” von Code-Schnipseln führt unübersichtlichen Programmen (Wartbarkeit)
- **Funktionen** fassen einen Codeblock, der häufig wiederverwertet wird und ein spezifisches Problem löst, zusammen
- Beispiele: Sortieren, Ausgabe in einem bestimmten Format, einen Satz in Wörter zerteilen (Tokenisieren) ...
- Wesentlicher Vorteil: Andere können auf einmal geschriebene Funktionen zurückgreifen, ohne deren innere Abläufe zu kennen

Funktionen Aufrufen

- Funktionen tragen Namen (ähnlich wie Variablen)
- Funktionen werden durch ihren Namen gefolgt von runden Klammern aufgerufen: `print()`
- Um abstrakte Probleme lösen zu können, müssen Funktionen mit unterschiedlichen Eingaben versorgt werden können
- Diese Eingaben (**Argumente**) werden mit dem Funktionsaufruf übergeben

Funktion Aufrufen

```
1 print("Beispiel") # Print-Funktion wird mit
      einem Argument ausgerufen
2
3 # Fuer eine Funktion ist jeweils festegelegt,
      wie viele Argumente uebergeben werden duerfen
4 # Dies kann jedoch tw. flexibel sein
5
6 print("Ein", "Beispiel")
7 len("Beispiel")
8 len("Ein", "Beispiel") # FEHLER
```

Funktionen definieren

```
1 #Funktionsdefinition
2 def pretty_print(x):
3     print("~~~~" + x + "~~~~")
4
5 # Funktionsaufruf
6 pretty_print("Hallo") # ~~~~Hallo~~~~
```

- **Funktionskopf**:
  - Kontrollwort `def`
  - Funktionsname (frei wählbar, aber siehe Variablennamen)
  - **Parameter** (Platzhalter für Argumente) in Klammern gefolgt von Doppelpunkt
- **Funktionskörper**
  - Der eigentliche Code der ausgeführt wird
  - Eingerückt (per Konvention 4 Leerzeichen)
  - Abschließend eine Leerzeile

Rückgabewert von Funktionen

- Damit ein Programm mit dem “Ergebnis” eine Funktion weiterechnen kann, muss diese einen Wert zurückgeben
- Rückgaben werden im Funktionskörper mit dem Kontrollwort `return` markiert
- Der Funktionsaufruf endet sobald der erste `return`-Befehl erreicht ist
- Achtung: Unterschied von Rückgabe und Ausgabe!

```
1  def addiere_1(x):
2      x = x +1
3      return x   # Rueckgabe
4
5  y = addiere_1(5)
6  print(y)   # Ausgabe
```

Übung: Funktion zum Quadrieren einer Zahl

Schreiben Sie eine Funktion, die ein Zahl entgegennimmt, diese quadriert und das Ergebnis **zurückgibt**.

Übung: Funktion zum Quadrieren einer Zahl (Lösung)

```
1  def quadriere(x):
2      x = x ** 2
3      return x
4
5  y = quadriere(2)
6  print(y)
```

Übung: Funktion zum Addieren von Zahlen

Schreiben Sie eine Funktion, die zwei Zahlen entgegennimmt, diese addiert und das Ergebnis **zurückgibt**.

Übung: Funktion zum Addieren von Zahlen (Lösung)

```
 1  def summe(x,y):
 2      z = x + y
 3      return z
 4
 5  # x = 2
 6  # y = 5
 7  # z = summe(x,y)
 8  # print(z)
 9
10  print(summe(2,5))
```

Übung: Naiver Tokenizer

Schreiben Sie eine Funktion, die einen String (z.B. einen Satz) entgegen nimmt und diesen in eine Liste von Wörter zerlegt. Diese Liste soll zurückgegeben werden. Die Funktion soll mindestens für die Eingaben ”Anne liebt ihren Kaffee heiß” und ”Anne liebt heißen Kaffee” funktionieren.

Übung: Naiver Tokenizer (Lösung)

```
1  def zerlegen(s):
2      w = s.split(" ")
3      return w
4
5  s1 = "Anne liebt ihren Kaffee heiss"
6  s2 = "Anne liebt heissen Kaffee"
7  print(zerlegen(s1))
8  print(zerlegen(s2))
```

Achtung: Computer denken nicht mit!

```
1  def foo():
2      print("foooooooo")
3      foo()
```

Achtung: Dynamisches Typensystem!

- Anders als in anderen Programmiersprachen wird der Datentyp der Argumente und der Rückgabe erst zur **Laufzeit** bestimmt
- Dies kann mitunter überraschende Effekte haben, da der Rückgabewert von der Verarbeitung innerhalb des Funktionskörpers abhängt

```
1  def add(x, y):
2      return x + y
3
4  print(add(3,4))           # 7
5  print(add("3", "4"))      # 34
6  print(add("3", 4))        # FEHLER
```

Variablenskopus

- Variablennamen haben jeweils einen bestimmten Gültigkeitsbereich (Variablenskopus)

```
 1  def pretty_print(x):
 2      b = "foo"
 3      print("~~~~" + x + "~~~~")
 4  pretty_print("bar")
 5  print(b) # Fehler: b nur innerhalb der Funktion,
        nicht global definiert
 6
 7  # Andersherum funktioniert es aber
 8  b = "foo"
 9  def pretty_print(x):
10      print(b)
11      print("~~~~" + x + "~~~~")
12  pretty_print("bar")
```

Unterabschnitt 5

Kontrollstrukturen

Grundlegende Bemerkungen

- Bisher: das Programm wird von oben nach unten, Zeile für Zeile abgearbeitet
- Was wir noch nicht können: Fallunterscheidungen, Wiederholung von Vorgängen eine bestimmte Anzahl von Malen (wesentliche Bausteine der Algorithmik)
- **Kontrollstrukturen** erlauben es den Programmfluss abhängig von der Eingabe zu ändern
- Wichtig für die Übung: `if/else`, `for`-Schleifen, `while`-Schleifen

if/else — Beispiel zum Einstieg

```
1  def groesser_3(zahl):
2      if zahl > 3:
3          return True
4      elif zahl == 3:
5          print("Eingabe war 3!")
6          return False
7      else:
8          return False
```

if/else

- Verzweigung des Programmflusses abhängig von einer Bedingung (`boolean`)
- Aufbau:
  - Beginnt mit Kontrollwort `if <Bedingung>:`
  - Es folgt ein eingerückter Block von Anweisungen. Wird nur ausgeführt, wenn Bedingung wahr.
  - Optional beliebig viele `elif <Bedingung>:`-Blöcke, die jeweils nur dann ausgeführt werden, wenn die Bedingung wahr ist und keiner der vorherigen Blöcke ausgeführt wurde.
  - Abschließend optional ein `else:`-Block, der nur ausgeführt wird, wenn keiner der Blöcke darüber ausgeführt wurde.
- **Also**, es kann immer nur höchstens einer der Blöcke ausgeführt werden

if/else — Weiteres Beispiel

```
1  def foo(x):
2      if x > 1000:
3          print("Sehr gross")
4      elif x > 100:
5          print("Immer noch gross")
6      elif x > 10:
7          print("Nicht mehr ganz so gross")
8      # Achtung: kein else!
```

Übung zu if/else

Schreiben Sie eine Funktion, die einen String entgegennimmt und überprüft, ob dieser länger als 10 Zeichen ist.

Übung zu if/else (Lösung)

```
1 def laenger_10(string):
2     if len(string) > 10:
3         return True
4     else:
5         return False
6
7 print(laenger_10("Stundenplan"))
```

for-Schleife

- Ermöglicht die Wiederholung eines Programmblocks (**Schleifenkörper**) eine bestimmte Anzahl von Malen
- Der **Schleifenkopf** gibt die **Schleifenvariable** und einen listenartigen Wert an
- Die Schleifenvariable nimmt beim Durchlaufen des Schleifenkörpers jeweils den nächsten (bzw. den ersten) Wert der Liste an

```
1  ...
2  for SCHLEIFENVARIABLE in LISTE:  # Schleifenkopf
3      ANWEISEUNG_1 # Anfang Schleifenkoerper
4      ANWEISUNG_2
5      ...
6      ANWEISUNG_N  # Ende Schleifenkoerper
7  ...
```

for-Schleife — Beispiele

```
1  for x in [1,2,3]:
2      print x
3
4
5  for x in ["eine", "die"]:
6      for y in ["Ampel", "Strasse"]:
7          print x + " " + y
```

Range-Funktion

- Kann genutzt werden um eine Folge von Zahlen darzustellen
- Wird nur ein Argument angegeben, umfasst `range` alle Zahlen von 0 bis exklusive dem Argument
- Werden zwei Argumente verwendet, gibt das erste den Startwert an (inklusive)

```
1  for x in range (3):
2      print(x)   # 0, 1, 2
3
4  for x in range(1, 5):
5      print(x)   # 1, 2, 3, 4
```

Übung: Summen-Funktion

Schreiben Sie ein Funktion, die zu einer angegebenen natürlichen Zahl, die Summe aller Zahlen von 1 bis inklusive der angebenen Zahl zurückgibt.

Übung: Summen-Funktion (Lösung)

```
1  def summe(x):
2      teilergebnis = 0
3      for i in range(x+1):
4          teilergebnis = teilergebnis + i
5      return teilergebnis
```

While-Schleife

- Wiederholt einen Programmblock, solange eine Bedingung erfüllt ist (bzw. bleibt)
- Nachdem der Schleifenkörper durchlaufe wurde, wird erneut geprüft, ob die Bedingung wahr ist
- Mächtiger, aber “risikobehafteter” als for-Schleife
- Programmierer muss sicherstellen, dass die Schleifenbedingung falsch werden kann

While-Schleife — Beispiele

```
 1  def countdown(x)
 2      while x >= 0:
 3          print(x)
 4          x = x-1
 5
 6  # Ueber Liste iterieren (aehnlich for-Schleife)
 7  some_list = [1, 2, 3]
 8  pointer = 0
 9  while pointer < len(some_list):
10      print(some_list[pointer])
11      pointer +=1
```

While-Schleife — Negativbeispiele

```
 1  # Beispiel 1
 2  x = 10
 3  while x >= 0
 4      print(x)
 5
 6  # Beispiel 2
 7  some_var = 17
 8  while some_var > 10:
 9     print("Immernoch groesser 10!")
10     some_var = some_var+1
11
12  # Beispiel 3
13  def countdown(x):
14      while x <= 0:
15          print(x)
16          x = x - 1
```

Übung: Zahlen mit while-Schleife aufsummieren

Schreiben Sie eine Funktion, die eine Liste von Zahlen entgegennimmt und diese mithilfe einer `while`-Schleife aufsummiert.

Übung: Zahlen mit while-Schleife aufsummieren (Lösung)

```
1  def summe(l):
2      teilergebnis = 0
3      pointer = 0
4      while pointer < len(l):
5          teilergebnis = teilergebnis + l[pointer]
6          pointer = pointer + 1
7      return teilergebnis
```

Übung: Verbesserter Tokenizer

Verbessern Sie den Tokenizer aus der letzten Übung, sodass einzelne Interpunktionszeichen am Ende von Tokens in eigene Tokens überführt werden. Aus “Ich mag, dass es regnet.” Soll `['Ich', 'mag', ',', 'dass', 'es', 'regnet', '."]` werden.

Unterabschnitt 6

Rekursion

Betrachten Sie erneut folgendes Negativbeispiel:

```
1  def foo():
2      print("fooooooo")
3      foo()
```

- Schon bekannt: Ein Funktionsaufruf ist mit der letzten Zeile des Funktionskörpers abgearbeitet
- Jeder Funktionsaufruf erzeugt eine neue **Instanz** der Funktion
- Es existieren hier potenziell unendliche viele Instanzen der Funktion (“Box in der Box”)

Rekursion

- “Art Funktionen zu schreiben”
- Beinhaltet Selbstaufrufe der Funktion
- Häufiger Aufbau:
  - Ist das Problem schon gelöst? Wenn ja, Ende der Rekursion.
  - Wenn nicht, Problem vereinfacher (zerlegen in Teilprobleme)
  - Selbstaufruf der Funktion mit zur Lösung des Teilproblems
  - Abschließend Vereinigung der Teillösungen zur Gesamtlösung.

Rekursion in der Mathematik

$a^0 := 1$
$a^x := a \cdot a^{x-1}$ für $x \in \mathbb{N}_+, a \in \mathbb{N}$

$4^1 = 4 \cdot 4^0 = 4 \cdot 1 = 4$
$2^3 = 2 \cdot 2^2 = 2 \cdot 2 \cdot 2^1 = 2 \cdot 2 \cdot 2 \cdot 2^0 = 2 \cdot 2 \cdot 2 \cdot 1 = 8$

```
1  def potenz(basis, exponent):
2      if exponent = 0:
3          return 1
4      else:
5          return basis * potenz(basis, exponent -
               1)
```

Übung: Multiplikation Rekursiv

Schreiben eine **rekursive** Funktion, die zwei Zahlen (positiv und ganzzahlig) entgegennimmt und das Produkt dieser Zahlen zurückgibt.

Übung: Multiplikation Rekursiv (Lösung)

```
1  # Code goes here
```

Übung: Palindrom-Erkenner

Ein Palindrom ist eine Zeichenkette, die vorwärts und rückwärts gelesen gleich ist (“otto”, “rentner”, “rotor”). Schreiben Sie eine Funktion, die einen String entgegenimmt und eine Boolean zurückgibt, ob es sich bei der Eingabe um ein Palindrom handelt.

Übung: Palindrom-Erkenner (Lösung)

```
1  # Code goes here
```