Python - Grundlagen der Programmierung

Tag 2

Giuseppe Accaputo

g@accaputo.ch

Inhaltsverzeichnis – Gesamter Kurs

1. Grundlagen der Programmierung
2. Variablen, Anweisungen, Ausdrücke, und alles dazwischen
3. Funktionen Teil 1 – Ein Einstieg
4. Bedingte Anweisungen («Conditionals»)
5. Funktionen Teil 2 – Rückgabewerte, Wiederverwendbarkeit, und mehr
6. Datenstrukturen – Listen, Strings, Tupel, und Dictionaries
7. Iterationen – Werkzeuge für repetitive Aufgaben

Repetition Tag 1

Variablen

- Eine *Variable* ist wie eine beschriftete Box, in welcher wir einen Wert verstauen (*speichern*) können
  - Eine Variable hat also einen *Namen* und einen *Wert* (z.B. `3.14`, `'Hello'`, oder `123`)
  - Der *Wert* gehört wiederum zu einem spezifischen *Typen* (z.B. String, Int, Float, etc.)

CODE

```
answer = 42
mein_name = 'Giuseppe'

print(answer)
print(mein_name)
print(type(mein_name))
```

INTERPRETER

PYCHARM

```
42
Giuseppe
<class 'str'>
```

Anweisungen und Ausdrücke

- Ein *Ausdruck* ist eine Kombination von Werten, Variablen, und Operatoren
  - Ein Ausdruck kann immer ausgewertet werden
- Eine *Anweisung* ist eine Instruktion (oder Befehl), welche der Python-Interpreter ausführen kann
  - Eine Ausdruck ist auch eine Anweisung

Anweisungen

- Zuweisungen: `meine_variable = 3`
- Funktionsaufrufe: `type(3.14)`
- Ausgabe: `print('Hallo, Welt!')`
- Ausdruck: `3 * 4 + 6`

Ausdrücke

- Ein *Ausdruck* ist eine Kombination von Werten, Variablen, und Operatoren
  - Ein Ausdruck kann immer ausgewertet werden

CODE

```
zahl1 = 4                # Anweisung
zahl2 = zahl1 + 3        # Anweisung
print(zahl2)             # Anweisung
3                        # Ausdruck
```

INTERPRETER

PYCHARM

```
7
```

Operatoren

- *Operatoren* sind spezielle Symbole, die z.B. Berechnungen wie die Addition oder Multiplikation darstellen
- Werte, die durch Operatoren verknüpft werden, heissen *Operanden*
- Die Bedeutung eines Operators hängt vom Datentyp der Operanden ab
  - Z.B. Der + Operator angewendet auf zwei Zahlen addiert diese zusammen; der + Operator angewendet auf zwei Strings verkettet sie hingegen

CODE

```
print(3 + 4 * 10)
print(3600 / 60)
print('Ein' + ' Beispiel')
```

INTERPRETER

PYCHARM

```
43
60
Ein Beispiel
```

Operatoren und die Vorrangregeln

Operator	Operation
(…)	Klammern
**	Exponent
%	Modulus
/	Division
*	Multiplikation
-	Subtraktion
+	Addition

Funktionen

– Eine Funktion ist wie ein Miniprogramm im Programm selbst

CODE

```
def hello(name):
  print('Hello,' + name + '!')

hello('Giuseppe')
```

INTERPRETER

PYCHARM

```
Hello, Giuseppe!
```

Bedingte Anweisungen («Conditionals»)

Lernziele

– Nach dieser Einheit wissen wir:

1. … was der Boolesche Datentyp darstellt
2. … was bedingte Anweisungen sind (und, dass Zweige nicht nur an Bäumen zu finden sind)
3. … wie eine Bedingung aussehen kann bei bedingten Anweisungen
4. … wie wir mit logischen Operatoren mehrere Bedingungen verknüpfen können

Der Boolesche Datentyp – Wahr oder falsch

Boolescher Datentyp

{Live Coding}

– Der Boolesche Datentyp erlaubt nur zwei Werte, nämlich `True` oder `False`

CODE
```
licht_ist_an = True
tuere_ist_offen = False
```

Vergleichsoperatoren

{Live Coding}

Ausdruck	Bedeutung
`x != y`	Ist x ungleich y? Falls ja, evaluiert zu `True`, sonst zu `False`
`x > y`	Ist x grösser als y? Falls ja, evaluiert zu `True`, sonst zu `False`
`x < y`	Ist x kleiner als y? Falls ja, evaluiert zu `True`, sonst zu `False`
`x >= y`	Ist x grösser oder gleich y? Falls ja, evaluiert zu `True`, sonst zu `False`
`x <= y`	Ist x kleiner oder gleich y? Falls ja, evaluiert zu `True`, sonst zu `False`

Logische Operatoren : Mehrere Bedingungen überprüfen

{Live Coding}

and Logisches UND

Ausdruck	Wert
True **and** True	True
True **and** False	False
False **and** True	False
False **and** False	False

or Logisches ODER

Ausdruck	Wert
True **or** True	True
True **or** False	True
False **or** True	True
False **or** False	False

not Logische Negation

Ausdruck	Wert
not False	True
not True	False

Logische Operatoren: Mehrere Bedingungen überprüfen

Ausdruck	Interpretation
`(x > 0) and (x < 10)`	Ausdruck evaluiert zu `True` nur dann wenn `x` grösser als `0` **und** `x` kleiner als `10` ist
`(y < 0) or (x < 10)`	Ausdruck evaluiert zu `True` wenn `y` kleiner als `0` **oder** `x` kleiner als `10` ist (oder beides erfüllt ist)
`not(x < y)`	Ausdruck evaluiert zu `True`, wenn `x` **nicht** kleiner als `y` ist.

Vorrangregeln aktualisiert

Operator	Operation
(…)	Klammern
**	Exponent
%	Modulus
/	Division
*	Multiplikation
-	Subtraktion
+	Addition
<, <=, >, >=, !=, ==	Vergleichsoperatoren
not	Negation
and	UND Verknüpfung
or	ODER Verknüpfung

Bedingte Anweisungen

Bedingte Anweisungen

{Live Coding}

- Wir möchten gewisse Sachen nur dann ausführen, wenn eine bestimmte Bedingung erfüllt ist
  - Beispiel: Nur volljährige Personen sollen in Klub Eintritt erhalten

CODE

```
alter = 19

if alter >= 18:
  print('Eintritt gewährleistet')
```

Bedingung

Snippet 1: Nur `if`-Anweisung

CODE

```
alter = 19

if alter >= 18:
  print('Eintritt gewährleistet')
else:
  print('Sorry, kein Einlass')
```

Verzweigung

Snippet 2: `if-else` Kombination

Alternative Ausführung – Mehrfache Verzweigung

{Live Coding}

– Wenn es mehr als zwei Möglichkeiten gibt benötigen wir mehr als zwei Zweige

CODE

```
if x < y:
  print(x, ' ist kleiner als ', y)
elif x > y:
  print(x, ' ist grösser als ', y)
else:
  print(x, 'und ', y, 'sind gleich gross')
```

– `elif` ist Abkürzung für «else if»

– Es wird wieder nur ein Zweig ausgeführt und Bedingungen werden der Reihe nach überprüft

Aufgabe • Bedingte Anweisungen

[Aufgabe]

– Gegeben sei folgender Code:

CODE

```
x = 10
y = 20

if x > y or x % 2 == 0:
  print('A')
elif x < y and y % 2 == 0:
  print('B')
Else:
  print('C')
```

– Wird **A**, **B**, oder **C** ausgegeben?

Lernziele – Check

– Nach dieser Einheit wissen wir:

1. … was der Boolesche Datentyp darstellt
2. … was bedingte Anweisungen sind (und, dass Zweige nicht nur an Bäumen zu finden sind)
3. … wie eine Bedingung aussehen kann bei bedingten Anweisungen
4. … wie wir mit logischen Operatoren mehrere Bedingungen verknüpfen können

Funktionen Teil 2 – Rückgabewerte, Wiederverwendbarkeit, und mehr

Lernziele

- Nach dieser Einheit wisst ihr...
  - …wie wir aus einer Funktion aus einen Wert zurückgeben können
  - …wie wir mit einem Rückgabewert weiterarbeiten können
  - …mathematische Funktionen schreiben, die gewisse Berechnungen durchführen und Resultate liefern

Mathematische Funktionen

{Live Coding}

– Das `math` Modul enthält eine Sammlung der gängigsten Funktionen aus der Mathematik

CODE

```
import math

log_e = math.log(10.0)
wurzel = math.sqrt(25)
```

– Übersicht der verfügbaren Funktionen: https://docs.python.org/3/library/math.html

Die Auswertung von Funktionen [Wichtiges Konzept]

{Live Coding}

– Viele Funktionen* können ausgewertet werden, d.h. der konkrete Funktionsaufruf wird zu einem Wert evaluiert:

CODE

```
import math
zwei = 2.0
wurzel_vier = math.sqrt(4.0)

summe = zwei + wurzelvier
print(summe)
```

*: Alle Funktionsaufrufe können ausgewertet werden, nur dass gewisse zu `None` (Datentype) ausgewertet werden

Die Auswertung von Funktionen [Wichtiges Konzept]

{Live Coding}

– Ein Funktionsaufruf kann zu einem Wert evaluiert werden, wenn die Funktion selbst *einen Wert zurückgibt*. Dies erreichen wir mit der Hilfe des **`return`** Keywords gefolgt von einem Ausdruck:

CODE

```
def summe(a, b):
  return a + b


summe1 = summe(10, 20)
summe2 = summe(3, 4)
```

– Ausdruck nach **`return`** kann Kombination aus Werten, Variablen, und Operatoren sein

– Ein Ausdruck hat auch einen Typ, d.h. wir können Funktionen definieren, die Floats, Strings, Ints, Bools, etc. zurückgeben können

Die `return` Anweisung inklusive Ausdruck – Einen Wert zurückgeben

- Es ist auch möglich mehrere `return` Anweisungen in eine Funktion zu haben – z.B. bei Funktionen, welche bedingte Anweisungen haben
  - Dabei ist es wichtig zu versichern, dass dabei jeder Pfad / Zweig ein Ergebnis zurückgibt

CODE

```
def abs(x):
  if x < 0:
    return -x
  else:
    return x
```

Step-by-Step Ausführung: {Live Coding}

Was geschieht genau, wenn wir `abs(-1)` aufrufen? Was, wenn wir `abs(1)` aufrufen?

Die `return` Anweisung ohne Ausdruck – Funktion verlassen

- `return` Anweisung ermöglicht es, eine Funktion frühzeitig zu verlassen / beenden
- Möglicher Grund: Fehlerbedingung tritt ein

{Live Coding}

CODE

```
def wurzel(x):
  if x < 0:
    print('Nur positive Zahlen und die 0 sind erlaubt')
    return

  return math.sqrt(x)
```

Universität Zürich[UZH]

Zentrale Informatik – IT Fort- und Weiterbildungen

Verknüpfungen

{Live Coding}

- Ausdrücke wie Werte und Variablen darf man auch als Teil anderer Ausdrücke (z.B. Funktionen) verwenden
- Auch hier gibt es eine Reihenfolge zu beachten bei der Auswertung:
  wir evaluieren von innen nach aussen

CODE

```
def summe(a, b):
  return a + b

def produkt(x, y):
  return x * y

resultat = summe(3, produkt(4, 5))
```

Boolesche Funktionen

{Live Coding}

– Wir können also auch Funktionen definieren, welche Boolesche Werte zurückgeben:

CODE

```
def ist_teilbar(x,y):
  if (x % y) == 0:
    return True
  else:
    return False

a = 20
b = 3
if ist_teilbar(a, b):
  print(str(a) + " ist teilbar durch " + str(b))
else:
  print(str(a) + " ist NICHT teilbar durch " + str(b))
```

Boolesche Funktionen können wir auch in Bedingungen verwenden, da sie zu **True** oder **False** evaluieren

Allgemeine Tipps für die Definition von Funktionen

- Welche Parameter soll die Funktion entgegen nehmen?
  - z.B. wenn eine Funktion Mittelwert aus drei Zahlen berechnen soll, so muss die Funktion drei Parameter entgegennehmenen
- Was soll die Funktion genau machen?
  - Falls die Funktion ein Resultat zurückgeben sollte, unbedingt `return resultat` einbauen
  - Eine Funktion kann auch nichts zurückgeben, z.B. um lediglich etwas auf dem Bildschirm ausgeben

Programmieren beginnt meistens auf dem Papier

- Scheut nicht Aufgaben und Programmskizzen zuerst auf Papier zu lösen oder aufzuschreiben
- Gedanken und Ideen zu Programmen zuerst auf Papier zu bringen hilft sehr

LC 4.1 • Sandwich-Funktion Teil 2 {Live Coding}

- Wir schreiben die Funktion `ist_sandwich` aus Aufgabe 3.3 so um, dass sie `True` zurückgibt, falls `x <= y <= z`, und `False` in allen anderen Fällen
  - **Frage**: `x <= y <= z` ist gültige Python 3.0 Syntax, jedoch können wir diese Bedingung auch mit einem logischen Operator ausdrücken. Wie würde der Ausdruck aussehen?

- Des Weiteren möchten wir, dass das Programm basierend auf dem Rückgabewert der `ist_sandwich` Funktion eine Meldung auf dem Bildschirm ausgibt, ob das Tripel `x`, `y`, und `z` ein Sandwich darstellt oder nicht

Inkrementelle Programmentwicklung

- Immer wieder bisschen Code schreiben / hinzufügen, und dann gleich testen (z.B. `print` einbauen)
- Verwende temporärere Variablen um Werte aus Zwischenschritten auszugeben und zu überprüfen
- Wenn Programm funktioniert, kann Code evtl. weiter vereinfacht werden
  - nur soweit vereinfachen, dass der Code doch noch lesbar bleibt

Wiederverwendung von Code mittels Funktionen

- Code Segmente, welche an verschiedenen Stellen des Programms in gleicher Form vorkommen / benötigt werden können wir in eine Funktion packen
- Funktionen können innerhalb von anderen Funktionen aufgerufen werden
- Wiederverwendung von Code durch Funktionen macht Code übersichtlicher und erlaubt es leichter Änderungen vorzunehmen
  - Bei Anpassungen müssen wir nur den Code der Funktion anpassen statt jede Stelle im Programm einzeln
  - Wir vermeiden unnötige Duplikation von Code

Lernziele – Check

- Nach dieser Einheit wisst ihr...
  - …wie wir aus einer Funktion aus einen Wert zurückgeben können
  - …wie wir mit einem Rückgabewert weiterarbeiten können
  - …mathematische Funktionen schreiben, die gewisse Berechnungen durchführen und Resultate liefern

Datenstrukturen

Lernziele

- Nach dieser Einheit wisst ihr...
  - …was der Unterschied zwischen einer *veränderbaren* und *unveränderbaren* Datenstruktur ist
  - …was ein Index ist und wie man damit auf einzelne Elemente der Datenstrukturen zugreifen kann
  - …wie man durch die Elemente der einzelnen Datenstrukturen durchiterieren kann

Datenstrukturen: Disclaimer

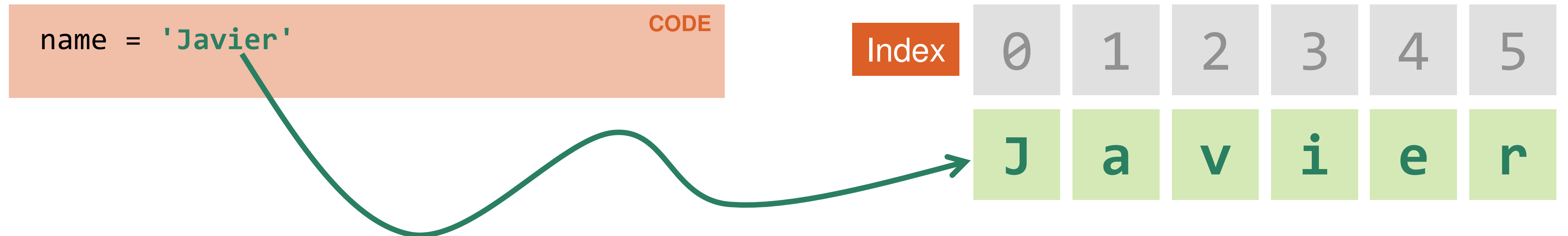
- Thema enthält viel neue Theorie und kann am Anfang überwältigend sein
- Wir werden viel skizzieren / zeichnen, da das bildliche Vorstellungsvermögen sehr helfen kann bei diesen neuen Themen
  - Ich empfehle auch hier wieder zuerst viel auf Papier versuchen zu lösen/nachprogrammieren
- Fragen sind zu jedem Zeitpunkt erlaubt und wie immer erwünscht

Datenstrukturen Teil 1 – Mit Strings Zeichenketten darstellen

Strings – Eine Sequenz von Zeichen

{Live Coding}

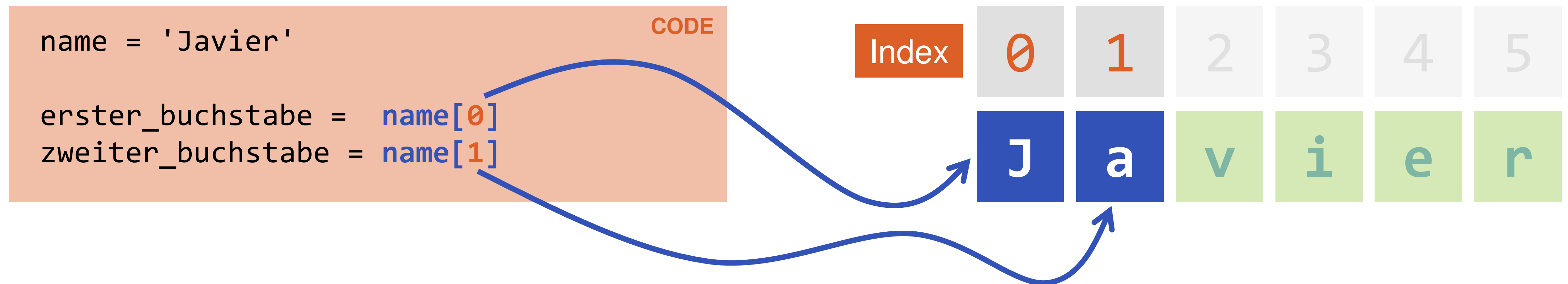
– Ein String ist eine *Sequenz von Zeichen*:



Strings – Eine Sequenz von Zeichen

{Live Coding}

– Wir können auf jedes Zeichen eines Strings zugreifen:



– Die Zahl im Ausdruck `name[0]` nennt man einen Index
  – Der *Index* zeigt an, auf welches Element in der Sequenz wir zugreifen möchten
  – Der *Index* ist eine ganze Zahl (`name[1.5]` ist kein gültiger Ausdruck)
  – Das erste Element hat den *Index* `0`

Einen String traversieren ("durchqueren")

{Live Coding}

– Wir können mit einer `for`-Schleife einen String traversieren (jedes Zeichen besuchen):

CODE

```
name = 'Javier'

for zeichen in name:
  print(zeichen)
```

String Segmentierung

{Live Coding}

– Wir können auf bestimmte *Segmente* in einem String zugreifen:

CODE

```
name = 'Javier'

# Zeichen 1, 2, und 3
segment1 = name[1:4]
```

CODE

```
name = 'Javier'

# Alle Zeichen ab Index 3
segment2 = name[2:]
```

Strings sind unveränderbar

{Live Coding}

– Wir können einen existierenden String nicht verändern:

CODE

```
sprache = 'Python'
# Ändere den ersten Buchstabe
sprache[0] = 'J'
```

INTERPRETER

PYCHARM

```
Traceback (most recent call last):
  File "…", line 3, in <module>
    sprache[0] = 'J'
TypeError: 'str' object does not
           support item assignment
```

– Wir können jedoch einen neuen String erstellen, der eine Variation des originalen Strings ist:

CODE

```
sprache = 'Python'
# Ändere den ersten Buchstabe
neue_sprache = 'J' + sprache[1:]
print(neue_sprache)
```

INTERPRETER

PYCHARM

```
Jython
```

Negative Indizes und die Länge eines Strings

{Live Coding}

– Wir können auch negative Indizes verwenden um Zeichen zu wählen:

Negativer Index:	-6	-5	-4	-3	-2	-1
	J	a	v	i	e	r

CODE
```
name = 'Javier'

letztes_z = name[-1]
```

– Mit der Hilfe der `len`-Funktion können wir die Länge eines Strings berechnen:

CODE
```
name = 'Javier'

laenge = len(name)
```

Strings vergleichen und durchsuchen

{Live Coding}

– Mittels `==` Operator können wir zwei Strings miteinander vergleichen:

CODE

```
name = 'Python'

if name == 'Python':
  print('Beide Strings sind gleich!')
```

– Mittels `in` Operator können wir stattdessen herausfinden, ob sich eine Zeichenfolge im String befindet:

CODE

```
text = 'Ein Fehler ist aufgetreten'

if 'Fehler' in text:
  print('"Fehler" kommt vor!')
```

Universität Zürich[UZH]

Zentrale Informatik – IT Fort- und Weiterbildungen

Datenstrukturen Teil 2 – Listen

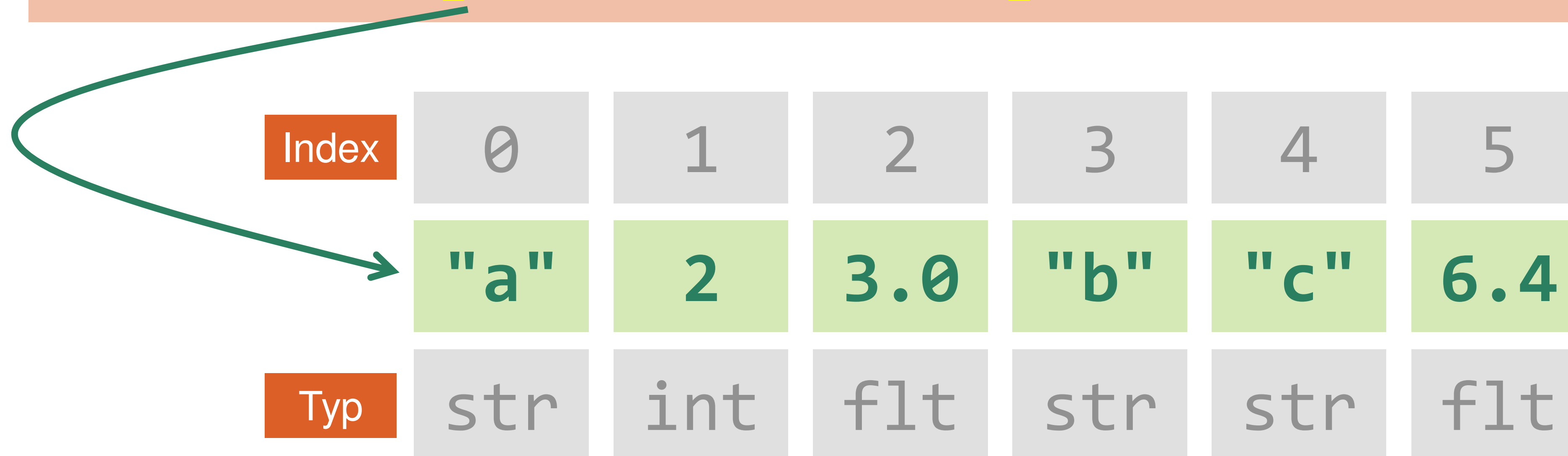
Listen – Eine Sequenz von beliebigen Werten

{Live Coding}

– In einem String sind die einzelnen Elemente lediglich Zeichen
– In einer Liste können die einzelnen Werte jedoch von einem beliebigen Typ sein

CODE

```
liste_von_strings = ["eins", "zwei", "drei"]
liste_von_ints = [1,2,3,4,5,6]
gemischte_liste = ["a", 2, 3.0, "b", "c", 6.4]
```



Listen – Auf einzelne Elemente zugreifen

{Live Coding}

– Genau wie bei den Strings können wir auch bei Listen auf einzelne Elemente mittels ganzzahligem Index zugreifen:

CODE

```
liste = ["eins", 2, 3.0, "vier"]
erstes_element = liste[0]
drittes_element = liste[2]

print(type(erstes_element))
print(type(drittes_element))
```

Listen sind veränderbar

{Live Coding}

– Im Gegensatz zu Strings sind Listen jedoch *veränderbar*:

CODE
```
liste = ["eins", 2, 3, "vier"]
print("alt:", liste)

# Ändere den Wert des ersten Elements
liste[0] = 1

print("neu:", liste)
```

Eine Liste durchqueren – Wir besuchen alle Elemente einzeln

– Wir können mit einer `for`-Schleife eine Liste traversieren (jedes Element besuchen):

CODE

```
liste = ["eins", 2, 3, "vier"]

for element in liste:
  print(element)
```

{Live Coding}

Negative Indizes und die Anzahl Elemente in einer Liste

{Live Coding}

– Wir können auch bei Listen negative Indizes verwenden

CODE
```
liste = ["a", "b", "c"]

letztes_e = liste[-1]
```

– Mit der Hilfe der `len`-Funktion können wir die Anzahl Element in einer Liste ausfindig machen:

CODE
```
liste = ["a", "b", "c"]

laenge = len(liste)
```

Listen vergleichen und durchsuchen

{Live Coding}

– Mittels `==` Operator können wir zwei Listen miteinander vergleichen.
  Zwei Listen sind genau dann gleich, wenn sie dieselben Elemente in derselben Reihefolge enthalten:

CODE

```
liste1 = ["a", "b", "c"]
liste2 = ["a", "b"]

if liste1 == liste2:
  print('Beide Listen sind gleich!')
```

– Mittels `in` Operator können wir herausfinden, ob sich ein Element in der Liste befindet:

CODE

```
liste = ["a", "b", "c"]

if "a" in liste:
  print('"a" kommt vor!')
```

Operationen mit Listen: Verkettung

{Live Coding}

– Mit dem + Operator können zwei Listen verkettet werden:

CODE

```
zahlen1 = [1,2,3,4]
zahlen2 = [5,6,7,8]

zahlen_gesamt = zahlen1 + zahlen2
```

Operationen auf Listen: Einfügen von Elementen

{Live Coding}

– Gewisse Operationen können wir auf Listen *direkt* anwenden, d.h. die Liste wird nach Anwendung der Operation verändert (man nennt diese auch *in-place* Operationen)

– Mit **append** können wir ans Ende einer Liste ein Element anfügen:

CODE

```
liste = []

liste.append("a")
liste.append("b")
```

Operationen auf Listen: Elemente entfernen

{Live Coding}

– Um Elemente aus einer Liste zu entfernen gibt es mehrere Möglichkeiten:

CODE
```
liste = [1,2,"drei","vier",5,6,"sieben",8,9.0]

# Entfernt Element beim Index 2
liste.pop(2)
# Entfernt das Element "vier"
liste.remove("vier")
# Entfernt Element beim Index 0
del liste[0]
# Entfernt die ersten drei Elemente
del liste[0:3]
```

– **Wichtig**: Nach jeder obigen Operation verändert sich die Liste

Operationen auf Listen: Eine Liste sortieren

{Live Coding}

– Mit `sort` können wir die Elemente einer Liste sortieren:

CODE

```
liste = ["b", "c", "a"]

liste.sort()
```

Datenstrukturen Teil 3 – Tupel

Tupel sind unveränderbar

{Live Coding}

- Wie eine Liste, ist auch ein Tupel eine Sequenz von Werten mit verschiedenen Typen
- Auf die einzelnen Werte können wir auch mit einem ganzzahligen Index zugreifen
- **Wichtiger Unterschied**: Im Vergleich zu Listen sind Tupel *unveränderbar*

CODE

```
tupelA = (1,2,3,4,"fuenf")
tupelB = "eins", "zwei", 3
tupelC = ("test")

# Wir können Tupel nicht verändern
tupelA[0] = 1
```

Tupel als Rückgabewert

{Live Coding}

– Wir können einem Tupel von Variablen ein Tupel von Werten zuweisen:

CODE
```
a, b, c = 1, 2, 3
```

– Des Weiteren können wir Tupel auch als Rückgabewert kombiniert mit obiger Syntax verwenden:

CODE
```
def summe_prod(a, b):
  return a+b, a*b

s, p = summe_prod(10,20)
```

Datenstrukturen Teil 4 – Dictionaries

Dictionaries

- Ein *Dictionary* ist eine Datenstruktur, die Schlüssel-Werte Paare (die *Elemente*) enthält
- Mit dem Schlüssel können wir die enthaltenen Elemente adressieren und dabei auf den Wert zugreifen
- Der Schlüssel kann dabei von einem beliebigem Datentyp sein
  - Strings, Listen, und Tupel verwenden ganzzahlige Indizes
  - Ein Dictionary kann z.B. einen String als Datentyp für den Schlüssel verwenden

Ein einführendes Beispiel

- Ein *Dictionary* ist eine Datenstruktur, die Schlüssel-Werte Paare (die *Elemente*) enthält
- Mit dem Schlüssel können wir dabei auf den Wert zugreifen

Schlüssel	Wert
"Buch"	"book"
"lernen"	"to study"
"Samstag"	"Saturday"
"Schleife"	"loop"
"solange"	"while"

Ein Dictionary definieren und verwenden

{Live Coding}

– Syntax um ein Dictionary zu definieren:

```
woerterbuch = {schluessel1: wert1, schluessel2: wert2, …}
```

Schlüssel	Wert
"Buch"	"book"
"lernen"	"to study"
"Samstag"	"Saturday"
"Schleife"	"loop"
"solange"	"while"

CODE

```
woerterbuch = {
  "Buch":        "book",
  "lernen":      "to study",
  "Samstag":     "Saturday",
  "Schleife":    "loop",
  "solange":     "while"
}
```

Auf Dictionary Einträge zugreifen

{Live Coding}

– Mit `dictionary[key]` können wir auf den Wert des Eintrags mit dem Schlüssel `key` im `dictionary` zugreifen:

CODE
```
woerterbuch = {
  "Buch":       "book",
  "lernen":     "to study",
  "Samstag":    "Saturday",
  "Schleife":   "loop",
  "solange":    "while"
}

buch_auf_englisch = woerterbuch["Buch"]
```

– **Neu**: **Index** ist jetzt vom Typ String

Elemente einfügen

{Live Coding}

– Wir können Elemente in ein Dictionary einfügen oder bestehende Elemente auch ersetzen:

CODE
```
woerterbuch = {
  "Buch":       "book",
  "lernen":     "to study",
  "Samstag":    "Saturday",
  "Schleife":   "loop",
  "solange":    "while"
}

woerterbuch["Kurs"] = "course"
```

Schlüssel	Wert
"Buch"	"book"
"lernen"	"to study"
"Samstag"	"Saturday"
"Schleife"	"loop"
"solange"	"while"
"Buch"	"book"
"Kurs"	**"course"**

Ein Dictionary durchqueren

{Live Coding}

– Mit einer **for**-Schleife und der Dictionary-Methode **items()** können wir jeden einzelnen Eintrag (Schlüssel-Wert Paar) im Dictionary besuchen:

CODE

```
woerterbuch = {...}

for (deutsch, englisch) in woerterbuch.items():
  print(" > Deutsch: ", deutsch, ". Englisch: ", englisch, sep="")
```

Dictionary-Elemente zählen, suchen, und löschen

{Live Coding}

- Mit der Hilfe der `len`-Funktion können wir die Anzahl Schlüssel-Werte Paare in einem Dictionary Liste ausfindig machen
- Mit dem `in` Operator können wir herausfinden, ob ein Eintrag mit ienem gegeben Schlüssel existiert:

CODE
```
wb = {"a": 2, "b", 3}
# Gibt es Eintrag mit Schlüssel "a"?
if "a" in wb:
  print("Gefunden!")
```

- Mit der Hilfe von `del` können wir ein Eintrag aus dem Dictionary löschen:

CODE
```
wb = {"a": 2, "b", 3}

# Lösche Eintrag mit Schlüssel "a"
del wb["a"]
```

Lernziele – Check

- Nach dieser Einheit wisst ihr...
  - …was der Unterschied zwischen einer *veränderbaren* und *unveränderbaren* Datenstruktur ist
  - …was ein Index ist und wie man damit auf einzelne Elemente der Datenstrukturen zugreifen kann
  - …wie man durch die Elemente der einzelnen Datenstrukturen durchiterieren kann

Iterationen – Werkzeuge für repetitive Aufgaben

Lernziele

- Nach dieser Einheit wisst ihr...
  - …was für einen Einfluss mehrere Anweisungen auf dieselbe Variable haben
  - …wie man repetitive Aufgaben mittels `while`-Schleifen implementieren kann
  - …warum es wichtig ist, die Schleifen-Bedingung nach einigen Schleifendurchgängen nicht mehr zu erfüllen (Stichwort *unendliche Schleifen*)

Variablen aktualisieren

{Live Coding}

- Eine oft-vorkommende Form von vermehrten Zuweisungen ist die *Aktualisierung einer Variable*, wobei der neue Wert der Variable vom alten Wert abhängt
  - Bevor man den Wert einer Variable aktualisiert, muss man sie initialisieren

CODE

```
x = 0
x = x + 1
x = x + 1
print(x)
```

INTERPRETER

PYCHARM

```
2
```

  - Variable um 1 erhöhen: *Inkrement*
  - Variable um 1 verringern: *Dekrement*

Aufgabe • Evaluierungen

[Aufgabe]

CODE

```
x = 1
y = 2
x = 2 * x + y
y = 3 * x + 2
y = y + 2
x = x * 2
```

– Wie lautet der Wert von **x** und wie der von **y** nach der Ausführung des obigen Codes?

Ein Countdown: Naive Version

{Live Coding}

– Naive Implementation eines Countdowns:

CODE

```
n = 3
print(x)
n = 2
print(x)
n = 1
print(x)
n = 0
print("Los geht's")
```

– **Frage**: Was geschieht mit dem obigen Code wenn wir den Countdown bei `x = 60` starten?

Ein Countdown: Verbesserte Version

{Live Coding}

– Wir möchten einen Weg finden, um eine variable Anzahl Schritte herunter zu zählen:

CODE

```
n = 3
print(x)
n = 2
print(x)
n = 1
print(x)
n = 0
print("Los geht's")
```

PSEUDOCODE

`n` ist `3`
Solange `n` grösser als `0` ist:
- Gib `n` aus
- Verkleinere `n` um `1`

Gib `Los geht's` aus

Die `while` Schleife

{Live Coding}

– Repetitive Aufgaben können wir mittels `while`-Schleife ausführen:

SYNTAX

`while` [**Bedingung erfüllt ist**]:
- Führe hier Anweisungen aus
- Bedingung muss hier aktualisiert werden

– Wir können den Countdown einfacher darstellen mit der Hilfe einer `while`-Schleife:

PSEUDOCODE

`n` ist `3`
Solange `n` grösser als `0` ist:
- Gib `n` aus
- Verkleinere `n` um `1`

Gib `Los geht's` aus

CODE

```
n = 3
while n > 0:
  print(n)
  n = n - 1

print("Los geht's!")
```

Aus einer Schleife ausbrechen – Die `break` Anweisung

{Live Coding}

- Mit der **`break`** Anweisung können wir aus einer **`while`**-Schliefe ausbrechen – unabhängig vom Zustand der Schleifen-Bedingung

CODE

```
while True:
  eingabe = input('Gib bitte das Zauberwort ein: ')

  if eingabe == 'Bitte':
    break

  print('Falsche Eingabe…')

print('Ende!')
```

Weitere Schleifenart: Die `for` Schleife

– Diese Schleifenart werden wir in Zusammenhang mit den Datenstrukturen kennenlernen

Lernziele – Check

- Nach dieser Einheit wisst ihr...
  - …was für einen Einfluss mehrere Anweisungen auf dieselbe Variable haben
  - …wie man repetitive Aufgaben mittels `while`-Schleifen implementieren kann
  - …warum es wichtig ist, die Schleifen-Bedingung nach einigen Schleifendurchgängen nicht mehr zu erfüllen (Stichwort *unendliche Schleifen*)

Bitte sichert eure Dateien

Feedback

- Ihr werdet im Anschluss an diesen Kurs von der Kursorganisation gebeten, Feedback zu diesem Kurs zu geben
- Ich bin über jedes einzelne Feedback wirklich froh und vorallem überaus dankbar, da ich den Kurs stetig verbessern möchte wo nötig
- Konstruktive Kritik ist mir wichtig, jedoch freue ich mich auch sehr über positive Kommentare
  - Es ist auch sehr wertvoll zu erfahren, was den Studenten z.B. speziell geholfen hat während dem Kurs

Fragen

– Ihr dürft mich jederzeit sehr gerne nach dem Kurs kontaktieren, falls ihr zu irgendwelchen Themen (Kurs-bezogen oder nicht) fragen habt: g@accaputo.ch

Referenzen

- Kursinhalt:
  - Allen B. Downey, "Think Python – How to Think Like a Computer Scientist" (Version 2.0.17), http://www.thinkpython.com
- Inspirationen einiger Aufgaben:
  - Michael Kündig, "Python - Grundlagen der Programmierung" (FS 2017), https://bitbucket.org/mkuendig/uzh-python-course
  - Jason Cannon, "Python Programming for Beginners" (2014)
  - https://www.practicepython.org/
  - https://www.w3resource.com/