Grundlagen der Programmierung für Nicht-Informatiker

Tag 2

Giuseppe Accaputo

g@accaputo.ch

Inhaltsverzeichnis – Gesamter Kurs

Inhaltsverzeichnis – Heutiger Kurstag

Universität Zürich[UZH]

Zentrale Informatik – IT Fort- und Weiterbildungen

Repetition Tag 1

Dem Computer erfolgreich Anweisungen geben

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'>
```

Aufgabe • Datentypen zuweisen [Aufgabe]

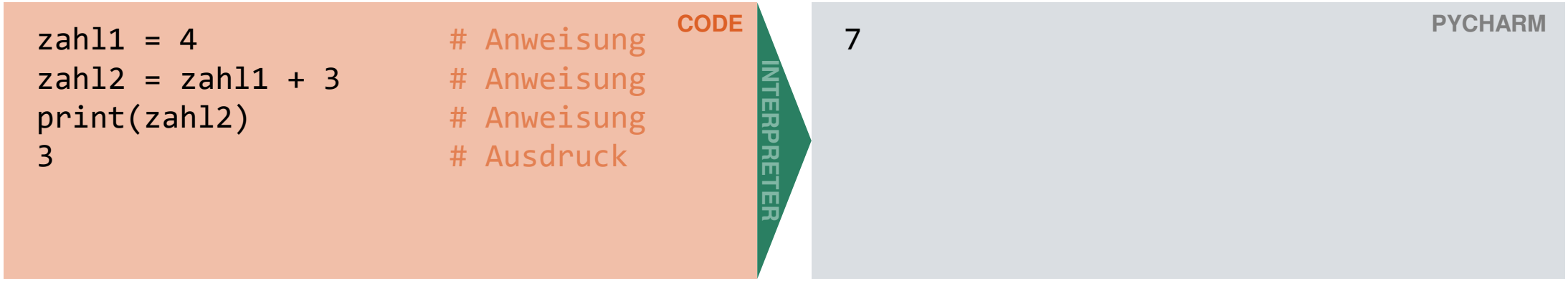
– Für jede der folgenden Anweisungen, versucht zu bestimmen welcher Datentyp ausgeben wird: Int, Float, String, oder Bool

CODE

```
print(type(1 < 2))
print(type(1.234))
print(type(1/2.0))
print(type("1.4123"))
print(type(1/4))
print(type(1 + 2.0))
print(type(0 == 0 and 1 < 2))
```

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

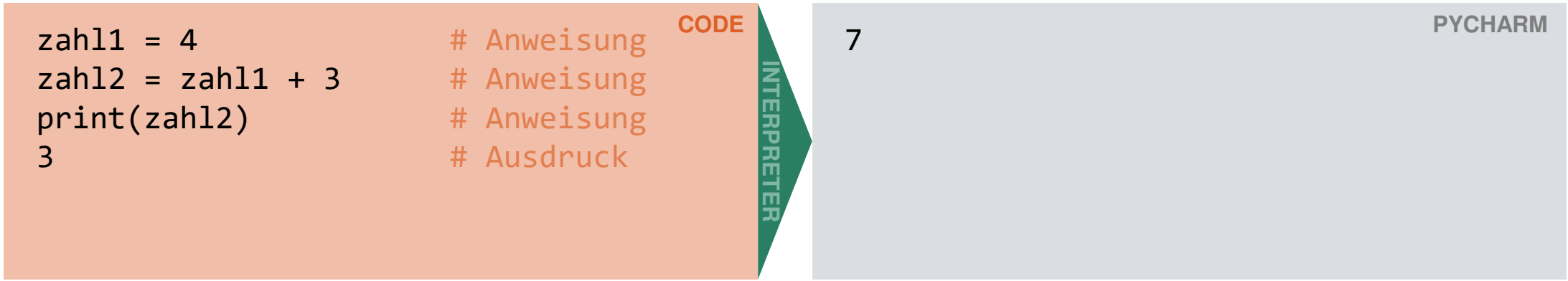


Anweisungen

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

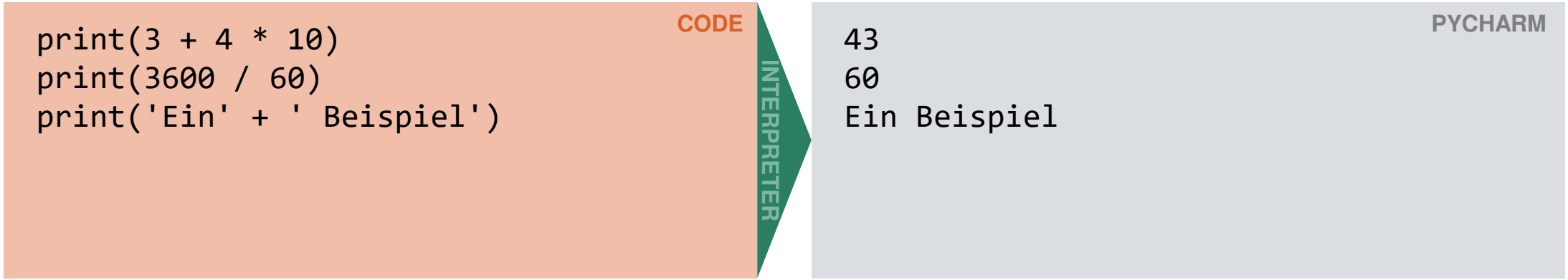
Ausdrücke

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



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



Operatoren und die Vorrangregeln

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

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

– Wir möchten gewisse Sachen nur dann ausführen, wenn eine bestimmte Bedingung erfüllt ist

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')
```

LC • Anweisungen und Ausdrücke

{Live Coding}

- Wir repetieren nochmals das Thema um Anweisungen und Ausdrücke am Whiteboard und mit Code
- Das Ziel ist es, dass wir verstehen, was die Evaluation von Anweisungen und Ausdrücken bedeutet

LC • Bedingte Anweisungen

{Live Coding}

– 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?

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

Rückgabewerte

- Funktionen können Resultate produzieren und zurückgeben
  - Dies bedeutet, dass die Funktion zu einem Wert evaluiert

CODE

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

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

INTERPRETER

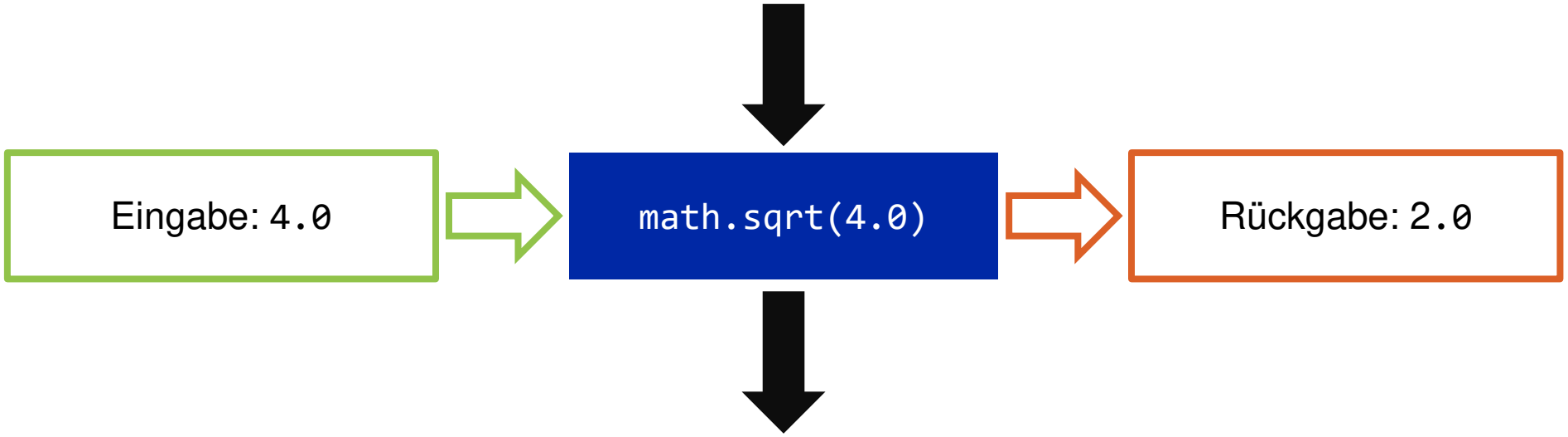
PYCHARM

```
4.0
```

Funktionen

CODE

```
wurzel_vier = math.sqrt(4.0)
```




```
wurzel_vier = 2.0
```

Die `input()` Funktion

- Eingabe: Eine Eingabe via Tastatur (Zeichenfolge, Zahl, etc.)
- Rückgabe: Die Eingabe wird zurückgegeben, d.h. wir können die Rückgabe z.B. in eine Variable speichern

CODE

```
meine_eingabe = input("Eingabe:")
print(meine_eingabe)
```

INTERPRETER

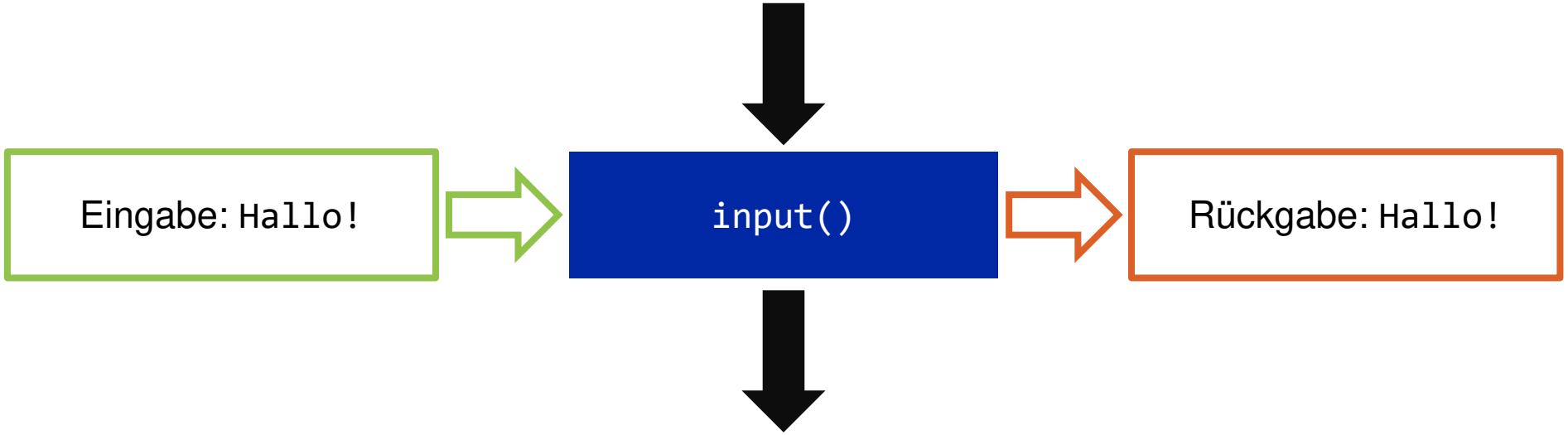
PYCHARM

```
Eingabe: Hallo!
Hallo!
```

Die `input()` Funktion

CODE

```
meine_eingabe = input()
```




```
meine_eingabe = "Hallo!"
```

Rückgabewerte: die `return` Anweisung

– Beispiel: Funktion, die Mittelwert aus vier Zahlen berechnen und *zurückgibt*

CODE

```
def mittelwert(w, x, y):
  avg = (w + x + y) / 3
  return avg

mittelwert_1 = mittelwert(1,2,3)

print(mittelwert1)
```

Verknüpfungen

- 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 Evaluation: wir gehen von innen nach aussen

CODE

```
def summe(x, y):
  return x + y

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

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

Die `return` Anweisung ohne Ausdruck – Funktionsausführung terminieren

- `return` Anweisung ermöglicht es, eine Funktion frühzeitig zu beenden
- Möglicher Grund: eintreten einer Fehlerbedingung

CODE

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

  resultat = sqrt(x)
  print('wurzel(x) = ' + str(resultat))
```

Die `return` Anweisung inklusive Ausdruck – Einen Wert zurückgeben

- `return` Anweisung inklusive Ausdruck: «Verlasse die Funktion sofort und verwende dabei den folgenden Ausdruck als Rückgabewert der Funktion»
  - *Reminder Nr. 1*: Ausdruck kann Kombination aus Werten, Variablen, und Operatoren sein
  - *Reminder Nr. 2*: Ein Ausdruck hat auch einen Typ, d.h. wir können Funktionen definieren, die Floats, Strings, Ints, Bools, etc. zurückgeben können
  - Wenn eine Funktion einen Rückgabewert hat, dann wird sie beim Aufruf zu einem Wert evaluiert

CODE

```
def mittelwert(w, x, y):
  avg = (w + x + y + z) / 3.0
  return avg

mittelwert_1 = mittelwert(1,2,3)
```

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
```

Funktionen ohne Rückgabewerte

- Falls bis zum Ende der Funktion kein `return` mit Anweisung ausgeführt wird, so gibt die Funktion `None` «zurück»
  - D.h. der Funktionsaufruf evaluiert zu `None`

CODE

```
def hello():
  print('Hello!')

hello_var = hello()
print(hello_var)
print(type(hello_var))
```

INTERPRETER

PYCHARM

```
Hello!
None
<class 'NoneType'>
```

Allgemeine Tipps für die Definition von Funktionen

- Welche Argumente soll die Funktion entgegen nehmen?
  - z.B. wenn eine Funktion den Mittelwert aus drei Zahlen berechnen soll, so muss die Funktion drei Argumente entgegennehmen
- Was soll die Funktion genau machen?
  - Falls die Funktion ein Resultat zurückgeben sollte, unbedingt `return resultat` einbauen
  - Eine Funktion kann auch nichts zurückgeben und z.B. 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 4.2 • Der Absolutwert

{Live Coding}

- Der Absolutwert einer reellen Zahl x ist gegeben als
  - $|x| = x$ falls $x \geq 0$ oder
  - $|x| = -x$ falls $x < 0$
- Eine Implementation des Absolutwerts ist in folgender Form gegeben:

CODE

```
def absolut_wert(x):
  if x < 0:
    return -x
  if x > 0:
    return x
```

- Was geschieht, wenn wir `absolut_wert(0)` aufrufen? Zu was evaluiert dieser Funktionsaufruf?

LC 4.1 • Kreisfläche berechnen {Live Coding}

– Wir implementieren eine Funktion `kreisflaeche`, die basierend auf einen Kreisradius r die dazugehörige Kreisfläche $A(r)$ berechnet und zurückgibt:

$$A(r) = \pi r^2$$

– Fragen:
  – Muss die Funktionen ein Argument entgegen nehmen? Wenn ja, welchen?
  – Welchen Wert gibt die Funktion zurück? Ist dieser abhängig von einem Argument (siehe erste Frage)?

Aufgabe 4.1 • Zahlen vergleichen [Aufgabe]

- Schreibe eine Funktion `vergleiche`, die folgende Werte **zurückgibt**:
  - `1`, falls `x > y`
  - `0`, falls `x == y`
  - `-1`, falls `x < y`
- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`vergleiche(3,1)`	`1`
`vergleiche(-1,4)`	`-1`
`vergleiche(42,42)`	`0`

- **Wichtig**: Die Funktion muss für beliebige Zahlen funktionieren

Boolesche Funktionen

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

CODE

```
# Gibt zurück, ob x durch y (ohne Rest) teilbar ist
def ist_teilbar(x,y):
  if (x % y) == 0:
    return True
  else:
    return False
```

Boolesche Funktionen

– Da das Resultat eines Vergleichs mittels `==` ein Boolescher Wert ist (z.B. evaluiert `3 == 3` zu `True`), so können wir die `ist_teilbar` Funktion ein bisschen prägnanter schreiben:

CODE

```
# Gibt zurück, ob x durch y teilbar ist
def ist_teilbar(x,y):
  return (x % y) == 0
```

LC 4.3 • 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 **keine** gültige Python Syntax; wie können wir diese Bedingung in Python ausdrücken? (Stichwort: logische Operatoren)

- 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
- 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

Aufgabe 4.2 • Umrechnung von Fahrenheit zu Celsius [Aufgabe]

- Definiere eine Funktion `celsius`, welche eine Temperaturwert f (gegeben in Fahrenheit) entgegennimmt, diesen in einen Celsius Temperaturwert c umwandelt und ihn anschliessend **zurückgibt**.
- Dabei könnt ihr die folgende Formel für die Umwandlung verwenden:

$$c(f) = (f - 32)\,\frac{5}{9}$$

Aufruf der Funktion im Programm	Rückgabewert
`celsius(33.8)`	`1`
`celsius(50)`	`10`
`celsius(95)`	`35`

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

Generalisierung von Funktionen

- Wir können auch mehrere Funktionen mit dem gleichen Namen zur Verfügung stellen in einem Programm
  - Die Funktionen müssen sich in der Anzahl der Parameter unterscheiden

CODE

```
def summe(x,y):
  return x + y

def summe(x,y,z):
  return summe(x,y) + z
```

Gültigkeitsbereich von Variablen

- Variable innerhalb einer Funktion ist nur lokal gültig (kann ausserhalb der Funktion nicht verwendet werden)
  - Parameter sind Funktions-lokale Variablen

CODE

```
# Globale Variable
nachname = 'Accaputo'

def hello(vorname):
  # vorname ist eine lokale Variable
  print('Hello,' + vorname +
                ' ' + nachname + '!')

hello('Giuseppe')
print(vorname) # nicht möglich!
```

INTERPRETER

PYCHARM

```
Hello, Giuseppe Accaputo!
```

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

Aufgabe 4.3 • Kugelvolumen berechnen [Aufgabe]

- Schreibe eine Funktion `kugel_volumen`, die anhand des Radius r das Volumen $V(r)$ einer Kugel berechnet und **zurückgibt**
- Die Formel für das Volumen V lautet

$$V(r) = \frac{4}{3} \cdot \pi \cdot r^3$$

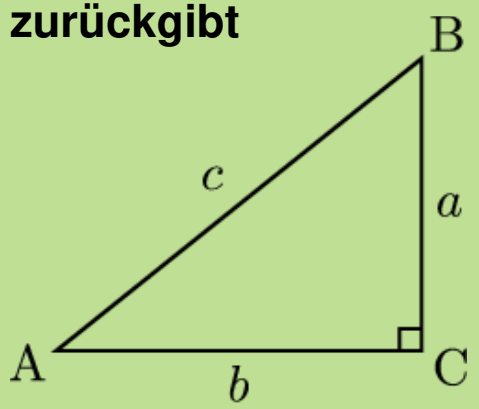
Aufruf der Funktion im Programm	Rückgabewert
`vol(1)` (Einheitskugel)	`4.18879020479` = $\frac{4}{3} \cdot \pi$
`vol(0)`	`0`
`vol(0) + vol(1)`	`4.18879020479`

Aufgabe 4.4 • Hypotenuse berechnen [Aufgabe]

– Schreibe eine Funktion `hypotenuse`, welche abhängig von den Seitenlängen a und b der Katheten eines rechtwinkligen Dreiecks die Länge c der Hypotenuse berechnet und **zurückgibt**

– Die Formel für die Länge c der Hypotenuse ist gegeben durch

$$c = \sqrt{a^2 + b^2}$$



– Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`hypotenuse(0,0)`	`0`
`hypotenuse(3,4)`	`5`
`hypotenuse(-1,2)`	`None` (negative Längen sind nicht erlaubt)

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*)

Mehrere Zuweisungen

– Einer Variable können wir mehrmals neue Werte zuweisen

CODE

```
name = 'Giuseppe'
print(name)
name = 'Marco'
print(name)
```

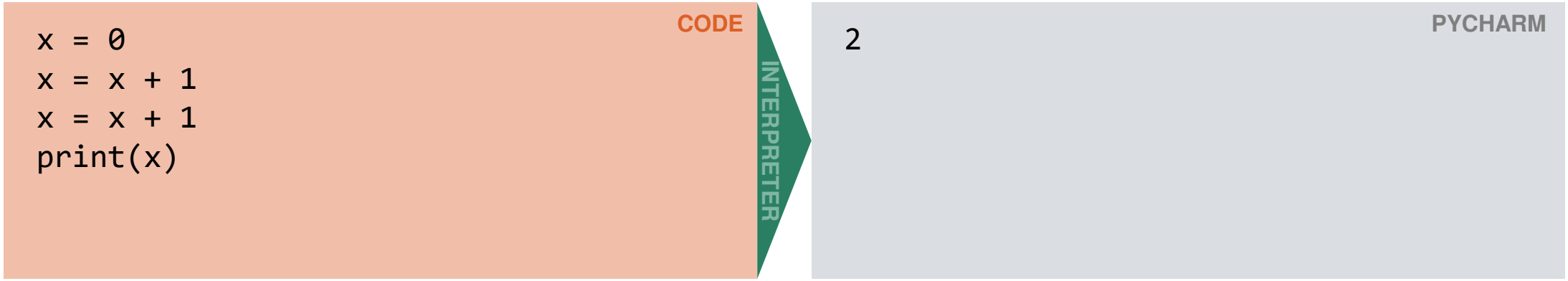
INTERPRETER

PYCHARM

```
Giuseppe
Marco
```

Variablen aktualisieren

- 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



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

LC 5.1 • Evaluierungen

{Live Coding}

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**, wie der von **y** nach der Ausführung des obigen Codes?

Die `while` Schleife

– Repetitive Aufgaben (wie z.B. das Inkrementieren einer Variable) können wir mittels while-Schleife ausführen

CODE

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

  print('LOS!')
```

– Das obige Programm kann man wie folgt *lesen*: "Solange `n` grösser als `0` ist, gib den Wert von `n` aus und dekrementiere `n` um `1`. Wenn `n` den Wert `0` hat, gib `LOS!` aus"

Die `while` Schleife – Ablauf

1. Evaluiere Bedingung, die entweder `True` or `False` ergibt
2. Falls die Bedingung nicht erfüllt ist – also `False` ergibt –, verlasse die `while`-Schleife und führe die nächste Anweisung aus
3. Falls die Bedingung erfüllt ist – also `True` ergibt –, führe den Block in der Schleife aus und gehe zu Schritt 1

```
def countdown(n):
  while n > 0: # Schleifen-Bedingung
    print(n)   # Schleifen-Block
    n = n - 1  # Schleifen-Block

  print('LOS!')# Erste Anweisung nach while-Schleife
```

LC 5.2 • Code vereinfachen

{Live Coding}

– Den folgenden Code möchten wir mittels while Schleife vereinfachen:

CODE

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
```

«Bis zur Unendlichkeit und noch viel weiter»

– Im Schleifen-Block sollte sichergestellt werden, dass die Schleifen-Bedingung innerhalb einer endlichen Anzahl Iterationen ungültig wird (also zu `False` evaluiert), sodass die Schleife abgebrochen wird

CODE

```
def countdown(n):
  while n > 0:
    print(n)

  print('LOS!')
```

Aus einer Schleife ausbrechen – Die `break` Anweisung

– 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!')
```

Aufgabe 5.1 • Login Abfrage (Gruppenarbeit) [Aufgabe]

- Schreibe ein Programm, dass dem Benutzer erlaubt sich mittels Benutzername und Passwort für ein Service einzuloggen
- Dabei hat der Benutzer maximal 3 Versuche um das Login richtig einzugeben, danach wird sein Account gesperrt
  - Der Benutzer soll über eine erfolgte Sperrung via Meldung am Bildschirm informiert werden
- Die Login Daten lauten:
  - Benutzername: `user`
  - Passwort: `mypass`
- **Tipp**: Wir brauchen sicher mal eine Variable, um die Anzahl Versuche zu zählen. Wie könnte dann die Schleifen-Bedingung aussehen?

BILDSCHIRM AUSGABE

```
Benutzername: user
Passwort: mypas
Falsches Login (Fehlversuch 1 von 3)

Benutzername: user
Passwort: mypass
Login erfolgreich!
```

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*)

Universität Zürich[UZH]

Zentrale Informatik – IT Fort- und Weiterbildungen

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

- Eine String ist eine Sequenz von Zeichen
- Wir können auf jedes Zeichen eines Strings zugreifen:

CODE

```
name = 'Marco'

erster_buchstabe = name[0]
zweiter_buchstabe = name[1]

print(erster_buchstabe)
print(zweiter_buchstabe)
```

INTERPRETER

PYCHARM

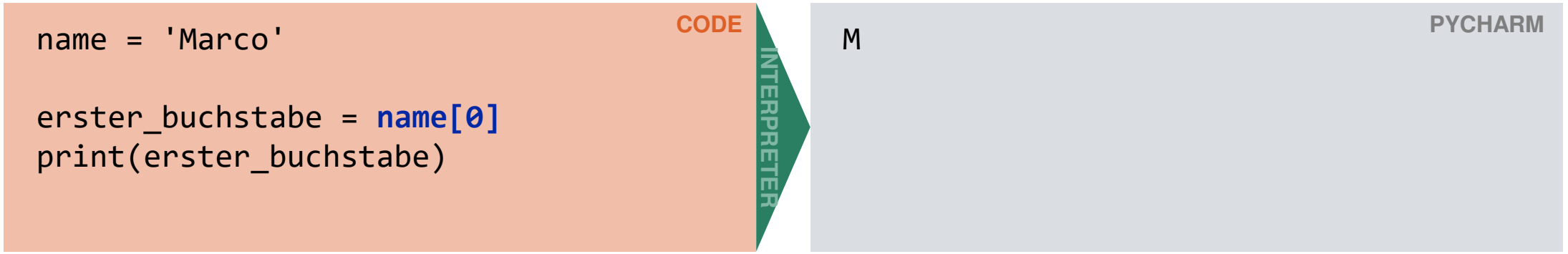
```
M
a
```

Strings – Auf einzelne Zeichen zugreifen

Index:	0	1	2	3	4
Zeichen:	M	a	r	c	o

Strings – Auf einzelne Zeichen zugreifen

- Die Zahl im Ausdruck `name[0]` nennt man einen *Index*
- Der Index zeigt an, auf welches Element in der Sequenz wir zugreifen möchten
- **Wichtig**: Der Index des ersten Elementes einer Sequenz hat in Python den Wert 0 (und nicht 1)



- Der Typ des Index muss immer Integer sein
  - `name[1.5]` ist daher kein gültiger Ausdruck

Strings – Auf einzelne Zeichen zugreifen

Index:	0	1	2	3	4	5
Zeichen:	P	y	t	h	o	n

Länge eines Strings

– Mit der Funktion `len(variable)` kann man die Anzahl Zeichen in einem String ermitteln:

CODE

```
name = 'Giuseppe'

print(len(name))
```

INTERPRETER

PYCHARM

```
8
```

Einen String durchqueren – Wir besuchen alle Zeichen einzeln

– Wir möchten jedes Zeichen eines Strings auf einer neuen Zeile ausgeben:

CODE

```
sprache = 'Python'

index = 0
anz_zeichen = len(sprache)

while index < anz_zeichen:
  zeichen = sprache[index]
  print(zeichen)
  index = index + 1
```

Einen String traversieren ("durchqueren")

– Wir können viel eleganter mit einer `for` Schleife jedes Zeichen in einem String traversieren:

CODE

```
sprache = 'Python'

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

– Beim Traversieren der Buchstaben wird in jedem Schritt das aktuelle Zeichen in der Variable `zeichen` gespeichert

Letztes Zeichen in einem String

– Versucht bitte folgenden Code auszuführen:

CODE

```
sprache = 'Python'
index_lz = len(sprache)
letztes_zeichen = sprache[index_lz]
```

Letztes Zeichen in einem String

– Weil wir bei Index 0 beginnen zu zählen, befindet sich das letzte Zeichen des Strings auf der Position `len(sprache) - 1`

CODE

```
sprache = 'Python'
index_lz = len(sprache) - 1
letztes_zeichen = sprache[index_lz]

print(letztes_zeichen)
```

INTERPRETER

PYCHARM

```
n
```

Warm-Up • Strings besser kennenlernen [Aufgabe]

- Definiert eine Funktion, welche als Argument einen String entgegennimmt
- Die Funktion gibt jedes Zeichen des Strings mittels vorangehender Aufzählung (*) auf einer neuen Zeile aus
- Beispielaufruf der Funktion :

CODE

```
aufzaehlen('Test')
```

INTERPRETER

PYCHARM

```
* T
* e
* s
* t
```

Letztes Zeichen in einem String – Negative Indizes

– Wir können auch negative Indizes verwenden. Dabei liefert sprache[-1] das letzte Zeichen, sprache[-2] das zweitletzte Element, etc.

CODE

```
sprache = 'Python'
letztes_zeichen = sprache[-1]

print(letztes_zeichen)
```

INTERPRETER

PYCHARM

```
n
```

String Segmentierung

- Wir können auf bestimmte Segmente in einem String zugreifen
- Der Zugriff geschieht mittels **`string[0:5]`**, wobei in diesem Fall die Zeichen **`string[0], string[1], …, string[4]`** selektiert werden

CODE

```
name = 'Barack Obama'

vorname = name[0:6]
nachname1 = name[7:12]
nachname2 = name[7:] # Segment von Index 7 aus bis ans Ende des Strings

print(vorname)
print(nachname1)
print(nachname2)
```

Strings sind unveränderbar

– Wir können einen existierenden String nicht verändern (z.B. in dem wir ein bestimmtes Zeichen ersetzen):

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
```

Strings sind unveränderbar

– 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
```

LC 6.1 • Zeichensuche

{Live Coding}

– Wir schreiben eine Funktion `suche(zeichen, wort)`, welches `True` zurückgibt, falls `zeichen` in `wort` vorkommt, und `False` falls nicht

– Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`suche("t", "Python")`	`True`
`suche("J", "Python")`	`False`

Aufgabe 6.1 • Vorkommnisse zählen [Aufgabe]

- Schreibe eine Funktion `anz_vorkommnisse(zeichen, wort)`, welche zurückgibt, wie oft `zeichen` in `wort` vorkommt (auf Grosskleinschreibung achten!)
- *Tipp*: Wir müssen einen Zähler verwenden, der sich merkt, wie oft das Zeichen bereits vorgekommen ist

- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`anz_vorkommnisse("a", "Halleluja")`	`2`
`anz_vorkommnisse("e", "Mount Everest")`	`2`
`anz_vorkommnisse("k", "Kathedrale")`	`0`

Kommt den eine Zeichenfolge überhaupt vor? – Der `in` Operator

- Mit dem `in` Operator können wir herausfinden, ob eine bestimmte Zeichenfolge (oder auch nur ein einzelnes Zeichen) in einem String vorkommt
  - Solch eine Zeichenfolge nennt man auch *Substring*

CODE

```
name = 'Python'

if 'yth' in name:
  print('"yth" kommt vor!')
```

INTERPRETER

PYCHARM

```
"yth" kommt vor!
```

Strings vergleichen

– Um zu schauen ob zwei Strings gleich sind kann man den == Operator verwenden:

CODE

```
name = 'Python'

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

INTERPRETER

PYCHARM

```
Beide Strings sind gleich!
```

Strings vergleichen

– Wir können die Vergleichs-Operatoren < und > dazu verwenden, um die *lexikografische Reihenfolge* zwischen zwei Strings zu überprüfen:

CODE

```
name1 = 'Python'
name2 = 'Jython'

if name1 < name2:
  print(name1, 'kommt vor', name2)
else:
  print(name1, 'kommt nach', name2)
```

INTERPRETER

PYCHARM

```
Python kommt nach Jython
```

Datenstrukturen Teil 2 – Listen

Listen – Eine Sequenz von beliebigen Werten

- 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]
```

CODE

Listen – Auf einzelne Elemente zugreifen

Index:	0	1	2	3
Elemente:	"eins"	2	3.0	"vier"

Listen – Auf einzelne Elemente zugreifen

Index:	0	1	2	3
Elemente:	"eins"	2	3.0	"vier"
Typ:	`str`	`int`	`float`	`str`

Listen – Eine Sequenz von beliebigen Werten

– Eine Liste ist auch wieder ein Datentyp:

CODE

```
liste_von_strings = ["ab", "cd"]
liste_von_ints = [1,2,3,4,5,6]
gemischte_liste = [1,2,"drei"]

print(type(liste_von_strings))
print(type(liste_von_ints))
print(type(gemischte_liste))
```

INTERPRETER

PYCHARM

```
<class 'list'>
<class 'list'>
<class 'list'>
```

– Wir kennen nun bereits 5 Datentypen: *Ints*, *Floats*, *Strings*, *Bools*, und *Lists*

Listen – Eine Sequenz von beliebigen Werten

– In einer Liste dürfen die Elemente von verschiedenen Typen sein:

CODE
```
liste_gemischt = ["eins", 2, 3, "vier"] # Liste enthält Ints und Strings
```

– Wir können auch eine leere Liste definieren:

CODE
```
leere_liste = []
```

Listen – Auf einzelne Elemente zugreifen

- Wir können wieder mittels eines Index auf ein einzelnes Element der Liste zugreifen:
- **Wichtig**: Der Index des ersten Elementes einer Sequenz hat in Python den Wert 0 (und nicht 1)

CODE

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

print(erstes_element)
print(drittes_element)
```

INTERPRETER

PYCHARM

```
eins
3.0
```

- Der Typ des Index muss immer Integer sein
  - `name[1.5]` ist daher kein gültiger Ausdruck

Listen – Eine Sequenz von beliebigen Werten

– In einer Liste dürfen die Elemente von verschiedenen Typen sein:

CODE

```
liste_gemischt = ["eins", 2, 3, "vier"] # Liste enthält Ints und Strings

print(type(liste_gemischt[0]))
print(type(liste_gemischt[1]))
print(type(liste_gemischt[2]))
print(type(liste_gemischt[3]))
```

Listen sind veränderbar

– Im Gegensatz zu Strings sind Listen veränderbar:

CODE

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

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

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

INTERPRETER

PYCHARM

```
alt: ["eins", 2, 3, "vier"]
neu: [1, 2, 3, "vier"]
```

Eine Liste durchqueren – Wir besuchen alle Elemente einzeln

– Wir können viel eleganter mit einer `for` Schleife jedes Element in einer Liste traversieren:

CODE

```
liste_gemischt = ["eins", 2, 3, "vier"] # Liste enthält Ints und Strings

for element in liste_gemischt:
  print(type(element))
```

Länge einer Liste

– Mit der Funktion `len(liste)` kann man die Anzahl Elemente in einer Liste ermitteln:

CODE

```
liste = [1,2,3,4,"fuenf"]

print(len(liste))
```

INTERPRETER

PYCHARM

```
5
```

Warm-Up • Listen besser kennenlernen [Aufgabe]

- Definiert eine Funktion, welche als Argument eine Liste entgegennimmt
- Die Funktion überprüft zuerst die Länge der Liste, diese muss nämlich mindestens 3 Elemente enthalten
- Wenn die Anzahl Elemente korrekt ist, ersetzt die Funktion das 1. und 3. Element mit dem Wert **0**
- Die Funktion gibt danach jedes Element der Liste auf einer neuen Zeile aus

- Beispielaufruf der Funktion :

CODE

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

INTERPRETER

PYCHARM

```
0
2
0
4
```

Eine Liste durchqueren – Wir besuchen alle Elemente einzeln

- Die `for` Schleife aus dem vorherigen Slide erlaubt es, die einzelne Elemente zu lesen
- Wollen wir stattdessen bspw. jedes Element verändern, so können wir die `range` Funktion verwenden und mittels Index direkt auf das Element zugreifen::

CODE

```
zahlen = [1,2,3,4] # Liste enthält Ints und Strings
anz_zahlen = len(zahlen)
indizes = range(anz_zahlen)

for index in indizes:
  zahlen[index] = zahlen[index] * 2 # Verdopple jedes Element

print(zahlen)
```

Die `range` Funktion

- **`range(n)`** gibt eine Liste von Indizes von **`0`** bis **`n-1`**, also **`[0,1,2,3, …,n-1]`**
- **`range(a,b)`** gibt eine Liste von Indizes von **`a`** bis **`b`** (**`a<b`**), also **`[a,a+1,…,b-1]`**

CODE

```
range1 = range(5)
range2 = range(3,8)

print(range1)
print(type(range1))
print(range2)
print(type(range2))
```

INTERPRETER

PYCHARM

```
[0, 1, 2, 3, 4]
<class 'list'>
[3, 4, 5, 6, 7]
<class 'list'>
```

Ist ein bestimmtes Element in der Liste vorhanden? – Der `in` Operator

– Mit dem `in` Operator können wir herausfinden, ob ein bestimmtes Element in einer Liste vorhanden ist

CODE

```
liste = [1,2,3,4,"fuenf"]

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

INTERPRETER

PYCHARM

```
"fuenf" kommt vor!
```

Elemente aus Liste entfernen

– Es gibt mehrere Möglichkeiten um ein Element aus einer Liste zu löschen:

CODE

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

liste.pop(2) # Entfernt Element beim Index 2, also den String "drei"
print(liste) # [1,2,"vier",5,6,"sieben",8,9.0]

liste.remove("vier") # Entfernt "vier"
print(liste) # [1,2,5,6,"sieben",8,9.0]

del liste[0] # Entfernt Element beim Index 0, also die Zahl 1
print(liste) # [2,5,6,"sieben",8,9.0]

del liste[0:3] # Entfernt die ersten der Elemente, also die Zahlen 2, 5, und 6
print(liste) # ["sieben",8,9.0]
```

Listen als Argumente einer Funktion

– **Achtung**: Wenn eine Liste als Argument einer Funktion übergeben wird, so kann die Liste von der Funktion verändert werden, sodass die Änderung auch "ausserhalb" sichtbar wird

CODE

```
def ersetze_element(liste):
  liste[0] = "neu"

liste = [1,2,3]
ersetze_element(liste)

print(liste) # ["neu", 2, 3]
```

Listen verketten / zusammenknüpfen / verbinden

– Mit dem + Operator können zwei Listen verkettet werden:

CODE

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

zahlen_gesamt = zahlen1 + zahlen2

print(zahlen_gesamt)
```

Eine Liste sortieren

CODE

```
liste = [5,1,3,2,4]

liste.sort()

print(liste)
```

INTERPRETER

PYCHARM

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

Ein Element ans Ende einer Liste anfügen

CODE

```
liste = [1,2,3]

liste.append("vier")

print(liste)
```

INTERPRETER

PYCHARM

```
[1, 2, 3, "vier"]
```

LC 6.2 • Beliebig grosse Liste

{Live Coding}

- Wir schreiben ein Programm, mit welchem der Benutzer mittels Eingabe eine beliebige Anzahl Elemente in eine Liste schreiben kann.
- Sobald `exit` eingegeben wird, wird das Programm beendet und die Listen Elemente werden vorher noch ausgegeben (inkl. die Anzahl Elemente in der Liste)

Aufgabe 6.2 • Mitte [Aufgabe]

– Schreibt eine Funktion `mitte`, welche eine neue Liste zurückgibt, die alle Elemente enthält, ausser dem ersten und letzten Element

– Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`mitte([1,2,3,4])`	`[2,3]`
`mitte([1,3,2])`	`[3]`
`mitte([3,2,5,20,5,100])`	`[2,5,20,5]`

Aufgabe 6.3 • Durchschnitt berechnen [Aufgabe]

– Schreibt eine Funktion `durchschnitt`, welche den Durchschnitt aus allen Elementen in einer Liste berechnet und zurückgibt (Bedingung: Liste darf nur Zahlen enthalten)

– Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`durchschnitt([1,2,3,4])`	`2.5`
`durchschnitt([4,18,30,-20])`	`8`
`durchschnitt([3,3,3,3])`	`3`

Aufgabe 6.4 • Grösstes und kleinstes Element finden [Aufgabe]

- Schreibt eine Funktion `min_max`, welche eine Liste bestehend aus zwei Elementen zurückgibt, wobei das erste Element das kleinste Element in der Liste, und das zweite Element das grösste Element in der Liste ist
  - **Tipp**: `min(liste)` gibt das kleinste Element in `liste` zurück, `max(liste)` jeweils das grösste Element

- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`min_max([102,-2,30,400])`	`[-2,400]`
`min_max([-123,430,5000,-300])`	`[-300,5000]`

Datenstrukturen Teil 3 – Tupel

Tupel sind unveränderbar

- 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

```
tupel1 = (1,2,3,4,"fuenf")
tupel2 = "eins","zwei",3
tupel3 = ("test") # Achtung!

type(tupel1)
type(tupel2)
type(tupel3)
```

INTERPRETER

PYCHARM

```
<class 'tuple'>
<class 'tuple'>
<class 'str'>
```

- Wir kennen nun bereits 6 Datentypen: *Ints*, *Floats*, *Strings*, *Bools*, *Lists*, und *Tuples*

Tupel sind unveränderbar

CODE

```
t = ('a', 'b', 'c')

print(t[0])
print(t[1:3])

t[0] = 'A' # Nicht möglich!
```

INTERPRETER

PYCHARM

```
a
('b', 'c')

TypeError: 'tuple' object does not
           support item assignment
```

Tupel Zuweisung

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

CODE

```
a, b, c = 1, 2, 3
print(a)
print(b)
print(c)
```

Tupel als Rückgabewerte

- Strikt gesprochen kann eine Funktion nur einen Wert zurückgeben
- Ist der Rückgabewert jedoch ein Tupel, so ist der Effekt derselbe, als würde man mehrere Werte zurückgeben

CODE

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

s, p = summe_prod(10,20)

print('s = ', s)
print('p = ', p)
```

INTERPRETER

PYCHARM

```
s = 30
p = 200
```

LC 6.3 • Division mit Rest

{Live Coding}

- Wir schreiben eine Funktion `div_mit_rest(a,b)`, welches als ersten Wert den Quotient zwischen `a` und `b`, und als zweiten Wert den Rest zurückgibt
- Beispiel: Bei der Division `10 / 3` beträgt das Resultat `3 Rest 1`, also beträgt der Quotient `3` und der Rest `1`

- Funktionsaufrufe und erwartete Rückgabewerte:

Aufruf der Funktion im Programm	Rückgabewert
`div_mit_rest(10, 3)`	`(3, 1)`
`div_mit_rest(12, 4)`	`(3, 0)`
`div_mit_rest(28, 5)`	`(5, 3)`

Datenstrukturen Teil 4 – Dictionaries

Dictionaries – Die Wörterbücher der Informatik

- 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
- Wir kennen nun bereits 7 Datentypen: *Ints*, *Floats*, *Strings*, *Bools*, *Lists*, *Tuples*, und *Dictionaries*

Beispiel: ein Dictionary

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

Ein Dictionary definieren und verwenden

– Syntax um ein Dictionary zu definieren:
variable_name = {schluessel1: wert1, schluessel2: wert2, …}

CODE

```
englisch_woerterbuch = {
# Schlüssel     Wert
# ---------     ----
  'Buch':       'book',
  'lernen':     'to study',
  'Samstag':    'Saturday',
  'Schleife':   'loop',
  'solange':    'while'
}

buch_auf_englisch = englisch_woerterbuch['Buch']
```

Auf Dictionary Einträge zugreifen

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

CODE

```
englisch_woerterbuch = {
# Schlüssel     Wert
# ---------     ----
  'Buch':       'book',
  'lernen':     'to study',
  'Samstag':    'Saturday',
  'Schleife':   'loop',
  'solange':    'while'
}

buch_auf_englisch = englisch_woerterbuch['Buch']
```

Ein leeres Dictionary erstellen

– Wir können mittels `{}` ein leeres Dictionary erstellen:

CODE

```
franz_woerterbuch = {}

franz_woerterbuch['programmieren'] = 'programmer'
franz_woerterbuch['Wörterbuch'] = 'dictionnaire'
franz_woerterbuch['lernen'] = 'apprendre'
```

Schlüssel	Wert
'programmieren'	'programmer'
'Wörterbuch'	'dictionnaire'
'lernen'	'apprendre'

Ein Dictionary traversieren ("durchqueren")

– Mit einer `for`-Schleife und der Dictionary-Funktion `items()` können wir durch ein Dictionary durchiterieren:

CODE

```
franz_woerterbuch = {}

franz_woerterbuch['programmieren'] = 'programmer'
franz_woerterbuch['Wörterbuch'] = 'dictionnaire'
franz_woerterbuch['lernen'] = 'apprendre'

print("Einträge im Wörterbuch:")

for (deutsch, franz) in franz_woerterbuch.items():
  print(deutsch, ":", franz)
```

Warm-Up • Dictionaries besser kennenlernen [Aufgabe]

- Wir möchten eine Speisekarte erstellen
- Definiert ein Dictionary, dass als Schlüssel ein Gericht hat (`String`), und als Wert den Preis (`Float`)
- Fügt einige Speisen zum Dictionary hinzu
- Die Funktion gibt danach die Preisliste aus (Gericht und Preis müssen ersichtlich sein)

- Beispielaufruf der Funktion :

CODE

```
menu = {
    …
}

preisliste_ausgeben(menu)
```

INTERPRETER

PYCHARM

```
Unsere Preisliste:
 * Burger, 10.5 CHF
 * Pommes, 4.0 CHF
 * Chicken Nuggets, 8.25 CHF
```

Wert eines Dictionary Eintrages ändern

- In einem Dictionary werden immer Schlüssel-Werte Paare gespeichert
- Wir können mit `dictionary[key] = value` den Werte des Eintrags mit dem Schlüssel `key` im `dictionary` auf `value` setzen
  - Falls ein Eintrag mit dem Schlüssel `key` nicht existiert, so wird dieser gleich zum Dictionary hinzugefügt

CODE

```
kontakte = {
  "Giuseppe": "g@accaputo.ch",
  "Peter M.": "peter@muster.ch",
}

kontakte[©Peter M."] = "peter@muster2.ch"
kontakte["Jens L."] = "jens@luchs.ch"

print(kontakte)
```

Dictionary Eintrag löschen

– Elemente kann man wie üblich mittels **`del`** aus einem Dictionary löschen:

CODE

```
kontakte = {
  "Giuseppe": "g@accaputo.ch",
  "Peter M.": "peter@muster.ch",
}

del kontakte["Peter M."]

print(kontakte)
```

Anzahl Dictionary Einträge

– Die Anzahl Elemente in einem Dictionary findet man mit Hilfe von `len`:

CODE

```
kontakte = {
  "Giuseppe": "g@accaputo.ch",
  "Peter M.": "peter@muster.ch",
}

anz_kontakte = len(kontakte)

print(anz_kontakte)
```

INTERPRETER

PYCHARM

```
2
```

Einen Schlüssel im Dictionary finden

– Mit `in` können wir herausfinden, ob ein Eintrag mit einem bestimmten Schlüssel im Dictionary befindet:

CODE

```
kontakte = {
  "Giuseppe": "g@accaputo.ch",
  "Peter M.": "peter@muster.ch",
}

if "Giuseppe" in kontakte:
  print("Giuseppe's Email autet:", kontakte["Giuseppe"])
else:
  print("Leider haben wir keinen Eintrag zu 'Giuseppe'")
```

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

Aufgabe 6.5 • Die Wissensdatenbank [Aufgabe]

- In den nächsten Aufgaben programmiert ihr eine Wissensdatenbank, welche Stichworte und dazugehörige Beschreibungen enthalten kann
- Folgende Funktionalität soll das Programm anbieten:
  - Hinzufügen eines Stichwortes und dessen Beschreibung zur Wissensdatenbank
  - Löschen eines Stichwortes (inkl. Beschreibung) aus der Wissensdatenbank
  - Nach der Beschreibung eines bestimmten Stichwortes in der Wissensdatenbank suchen
  - Auflistung aller Stichworte und der dazugehörigen Beschreibung die aktuell in der Wissensdatenbank vorhanden sind anzeigen

- Die komplette Aufgabe inklusive Teilaufgaben findet ihr in der Aufgabensammlung zum Tag 2

Bitte Daten sichern nicht vergessen

Vielen herzlichen Dank für die tolle Zusammenarbeit

– Es hat sehr viel Spass gemacht, diesen Kurs mit euch durchzuführen

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 nämlich 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, "Grundlagen der Programmierung für Nicht-Informatiker" (FS 2017), https://bitbucket.org/mkuendig/uzh-python-course
  - Jason Cannon, "Python Programming for Beginners" (2014)