



Python - Grundlagen der Programmierung

APPE4 – Herbstsemester 2020

Giuseppe Accaputo

g@accaputo.ch



Schön seid ihr heute hier



Wer seid ihr?

- Studiengang / Beschäftigung
- Programmiererfahrung:
 - 0: Keine Erfahrung
 - 1: Wenig Erfahrung
 - 2: Wesentliche Erfahrung
- Ziele für den Kurs



Über mich

- Arbeit
 - Software Entwickler, Nexxiot AG (seit April 2018) [\[Groovy, Java, Kotlin\]](#)
 - Wissenschaftlicher Mitarbeiter, ETH Zürich (2017 – 2018) [\[Bash, C, C++, MATLAB, Python\]](#)
 - Übungs- und Kursleiter, ETH Zürich (2014 – 2017) [\[C++, Java, MATLAB, Pascal\]](#)
 - Nachhilfelehrer, selbstständig (2016 – 2018) [\[C++, Java, R\]](#)
 - Software Entwickler, LTV Gelbe Seiten AG (2009 – 2011) [\[C#, JavaScript\]](#)
- Ausbildung
 - B.Sc. & M.Sc. ETH in Rechnergestützte Wissenschaften (2011 – 2017)
 - B.Sc. FH in Informatik mit Vertiefung in Software Engineering (2006 – 2009)
 - Berufslehre als Informatiker EFZ Fachrichtung Systemtechnik (2002 – 2006)



Aufbau Kurs

- Kurs besteht aus mehreren Lerneinheiten
- Pro Lerneinheit:
 - Definition der Lernziele
 - Inhalt der Lerneinheit
 - Passende Übungen
 - Live Coding
 - Kontrolle Lernziele



Python 3.0 Spickzettel

Python 3.0 Spickzettel Giuseppe Accaputo g@accaputo Kurs: Grundlagen der Programmierung für Nicht-Informatiker, HS18	
Variablen	
<code>variablen_name = <wert></code>	
<code>typ_der_variable = type(variable)</code>	
Datentypen	
Integer (int)	-25, 2, 14, 100, -20
Float	2.4123, -1.1312, 4.14123
String	'Hallo 1', 'Hallo 2'
Boolean	True, False
List	[1, 1.2423, 'zwei']
Tupel	(1, 2, 3, 'vier')
Dictionary	{'key1': wert1, 'key2': wert2}
Eingabe	
<code>eingabe = input('Bitte Name eingeben:')</code>	
<code>eingabe = int(input('Bitte Zahl eingeben:'))</code>	
<code>eingabe = float(input('Bitte Zahl eingeben:'))</code>	
Ausgabe	
<code>print('Wert der Variable:', variable)</code>	
<code>print('Typ der Variable', type(variable))</code>	
Strings – Teil 1	
<code>ein_string = 'Hallo, Welt!'</code>	Variable vom Typ String definieren
<code>ein_string[1:5]</code>	Segment von Index 1 bis und mit Index 4 selektieren
<code>ein_string[-1]</code>	Auf das letzte Zeichen zugreifen
<code>ein_string.lower()</code>	In Kleinbuchstaben umwandeln
<code>ein_string.upper()</code>	In Grossbuchstaben umwandeln
<code>ein_string.replace(alt, neu)</code>	alt durch neu in ein_string ersetzen
<code>string1 == string2</code>	Ist string1 gleich string2?
<code>zahl_als_string = str(1.234)</code>	Typumwandlung zu String

<code>hallo = 'Hallo, '</code> <code>welt = 'Welt!'</code> <code>hallo_welt = hallo + welt</code>	+ Operator: Zwei Strings verknüpfen
<code>area = 'Area '</code> <code>area_zahl = 51</code> <code>area_51 = area + str(area_zahl)</code>	+ Operator und <code>str()</code> : String und Zahl verknüpfen
Zahlen	
<code>int(variable)</code>	Typumwandlung zu Int
<code>float(variable)</code>	Typumwandlung zu Float
Mathematische Operatoren	
<code>x ** y</code>	Exponent, <code>x^y</code>
<code>x % y</code>	Modulus; berechnet den Rest der Division x geteilt durch y
<code>x / y</code>	Division
<code>x * y</code>	Multiplikation
<code>x - y</code>	Subtraktion
<code>x + y</code>	Addition
<code>x op y</code> , und <code>x, y</code> sind beide Ints	Resultat der Operation ist ein Int (op kann +, -, *, / sein)
<code>x op y</code> , und <code>x</code> oder <code>y</code> ist Float	Resultat der Operation ist ein Float (op kann +, -, *, / sein)
Funktionen	
<code>def hallo():</code> <code>print('Hallo!')</code> <code>hallo()</code> # Aufruf	Eine Funktion ohne Rückgabewert
<code>def hallo(name):</code> <code>print('Hallo, ', name)</code> <code>hallo('Klasse')</code> # Aufruf	Eine Funktion mit einem Argument
<code>def summe(x, y, z):</code> <code>return x + y + z</code> <code>print(summe(1,2,3))</code> # Aufruf	Eine Funktion mit einem Rückgabewert
<code>import math</code> <code>math.sqrt(zahl)</code> # Wurzel <code>math.log(zahl)</code> # Logarithmus	Mathematische Funktionen verwenden



Bitte sichert eure Dateien regelmässig ab



"Was war das denn? Ich verstehe gar nichts mehr"

- Ihr werdet während diesem Kurs evtl. in gewissen Situationen überfordert sein, gewisse Konzepte nicht gleich auf Anhieb verstehen, oder allgemein das Gefühl haben, dass ihr nicht für's Programmieren gemacht seid...

...und das sind alles Gefühle, die in unserer Situation völlig normal sind, denn wir lernen in diesen zwei Kurstagen einige Themen kennen, die zu einer für euch komplett neuen Disziplin gehören.

Auch mir ging es so, als ich mit dem Programmieren begonnen habe.

Dies ist alles Teil des Lernprozesses. Es ist völlig okay, sich so zu fühlen.



Fragen während und nach dem Kurs sind zu jeder Zeit erlaubt und erwünscht

- **Deshalb wichtig:** Auch wenn ihr das Gefühl habt, dass eine Frage evtl. "nicht gut genug" sein könnte oder ein Konzept einfach zu verstehen sein sollte, und ihr deshalb nicht fragen möchtet, stellt die Frage bitte trotzdem 😊
 - Dies gibt mir auch die Möglichkeit, nach besseren Erklärungen für gewisse Konzepte zu suchen
- Fragen gehören zum Lernprozess und sollen zu jeder Zeit gestellt werden dürfen
- Ihr dürft mich auch sehr gerne während oder nach dem Kurs kontaktieren, falls ihr zu irgendwelchen Themen fragen habt: g@accaputo.ch



Inhaltsverzeichnis – Gesamter Kurs

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



Kurs – Lernziele

- Nach diesem Kurs...
 - ... kennt ihr einige fundamentale Grundlagen der Programmierung
 - ... kennt ihr die wichtigsten Bausteine der Python Programmiersprache
 - ... könnt ihr Python Code ausführen
 - ... könnt ihr einfache Aufgaben in ein Python Programm abbilden und ausführen



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

Grundlagen der Programmierung



Lernziele

- Nach dieser Einheit wisst ihr...
 1. ... was ein Programm ist
 2. ... warum wir Programmiersprachen benötigen
 3. ... warum wir uns für Python entschieden haben



Ein erstes Programm

– Idee für ein Programm:

1. Zahl 1 hat Wert 10
2. Zahl 2 hat Wert 30
3. Gib Zahl 1 + Zahl 2 auf dem Bildschirm aus

IDEE

– Programm in Python umgesetzt:

```
zahl_1 = 10  
zahl_2 = 30  
print(zahl_1 + zahl_2)
```

CODE



Was ist ein Programm?

- Ein Programm ist eine Folge von Anweisungen, um bestimmte Aufgaben oder Probleme mithilfe eines Computers zu bearbeiten oder zu lösen
- Ein Rezept von Befehlen um dem Computer mitzuteilen was dieser in welcher Reihenfolge machen soll

Was ist ein Programm?

- Programm in Python umgesetzt:

```
zahl_1 = 10  
zahl_2 = 30  
print(zahl_1 + zahl_2)
```

CODE

- Das obige Programm...
 - ...besteht aus 3 Anweisungen
 - ...hat eine eindeutige Reihenfolge: Die Anweisungen werden von oben nach unten interpretiert



Dem Computer erfolgreich Anweisungen geben



Schritt 1: Anweisungen in Code erfassen

- Code enthält Anweisungen, welche in der gewünschten Programmiersprache von einem Menschen (im Weiteren *Entwickler* genannt) geschrieben sind
 - Ziel: Anweisungen sollen am Ende dieser Prozedur vom Computer ausgeführt werden
- Wird mittels Texteditor oder Entwicklungsumgebung geschrieben und in einer Textdatei abgespeichert



Schritt 2: Code vom Interpreter übersetzen lassen

- Der Interpreter – oder auch *Übersetzer* genannt – übersetzt den Code in einen maschinennahen Code, welcher vom Computer verstanden wird
 - Der maschinennahe Code ist in einer *Maschinsprache* geschrieben und enthält für den Computer verständliche Anweisungen
- Interpreter ist «Schnittstelle» zwischen Entwickler und Computer



Schritt 3: Die Anweisungen vom Computer ausführen

- Nachdem ein Code erfolgreich zu Maschinencode übersetzt wurde, werden die Anweisungen vom Computer ausgeführt
- Gängige Programmiersprachen sind äusserst *portabel*, d.h. Code, welcher mit solchen Programmiersprachen erstellt wurde kann auf verschiedenen Betriebssystemen (Windows, Mac OS X, und Linux) übersetzt und ausgeführt werden





Warum Python?

- Einfache Sprache
 - Sehr angenehm bei ersten Schritten in der Programmierung
- Hochsprache
 - Wir müssen uns nicht um «gewisse Details» kümmern; Sprache nimmt uns sehr viel ab (schränkt auch ein, ist aber in unserer Situation überhaupt nicht schlimm)
- Interpretierte Sprache
- Portierbar auf den gängigsten Betriebssystemen
 - Code kann auf Windows, Mac OS X, und Linux interpretiert und ausgeführt werden
- Umfangreiche Bibliotheken
 - Bibliotheken die numerische Methoden anbieten, Graphen generieren können, spezielle Dateien einlesen können, Statistiken berechnen, etc.



Learning by Doing (and by Making Errors)

- Programmieren ist eine *hands-on experience*
- Habt keine Angst Sachen in Python auszuprobieren
- Fehler machen ist eine wesentliche Komponente des Lernprozesses und hilft euch auch gewisse Situationen besser zu verstehen
 - Falls ihr irgendwo stecken bleibt während einer Aufgabe oder inmitten eines Slides, meldet euch bitte gleich ungeniert bei mir oder bei euren Mitstudenten
- Versucht auch Aufgaben mit Blatt und Stift zu lösen, vorallem wenn ihr gerade versucht, ein neues Konzept besser zu verstehen
 - Es kann manchmal sehr helfen eine Computer-Pause einzulegen



Hallo, integrierte Entwicklungsumgebung!

- Integrierte Entwicklungsumgebung (IE):
Sammlung von Tools um Softwareentwicklung angenehmer zu gestalten
- PyCharm: IE um Python Programme zu entwickeln
 - Code wird mit Hilfe von PyCharm geschrieben, übersetzt und gleich ausgeführt ohne
 - Ohne IE: Tool um Code zu schreiben, Terminal um Code zu übersetzen und auszuführen



«Hallo Welt!»

- Unser erstes Programm:

```
print('Hallo Welt!')
```

CODE

«Hallo Welt!» in anderen Programmiersprachen

– Java:

```
public class HalloWelt{  
    public static void main(String args[]){  
        System.out.println('Hallo Welt!');  
    }  
}
```

CODE

– C++:

```
#include <iostream>  
int main(){  
    std::cout << 'Hallo Welt!' << std::endl;  
    return 0;  
}
```

CODE



Lernziele – Check

- Nach dieser Einheit wisst ihr...
 1. ... was ein Programm ist
 2. ... warum wir Programmiersprachen benötigen
 3. ... warum wir uns für Python entschieden haben



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

Variablen, Anweisungen, Ausdrücke, und alles dazwischen



Lernziele

- Nach dieser Einheit wissen wir:
 1. ... was eine Variable ist
 2. ... wie man einer Variable einen Wert zuweist
 3. ... wie man eine oder mehrere Variablen ausgibt
 4. ... was *Strings*, *Floats*, und *Ints* sind
 5. ... wieso wir Typumwandlungen benötigen



Werte und Datentypen



Werte

- Fundamentale Sache wie z.B. ein Buchstabe oder eine Zahl
 - Beispiel eines Wertes: **2**
 - Weiteres Beispiel eines Wertes: **'Python'**
 - Noch ein Beispiel eines Wertes: **3.14159**

Datentypen

- Datentypen charakterisieren / beschreiben:
 - eine spezifische Menge von zusammengehörenden Werten (z.B. ganze Zahlen)
 - welche Operationen darauf ausgeführt werden können (z.B. Addition zweier Zahlen)
 - wie die Werte abgespeichert werden
- *Beispiel:* Wir haben einen Katze Namens Petra.
 - *Petra* ist eine konkrete Instanz / Realisierung einer Katze und *Katze* ist der Typ
 - Der Typ *Katze* fasst einige Eigenschaften zusammen, welche alle Katzen gemeinsam haben
 - Sehr vereinfacht kann man sagen:
 - Alle Katzen haben 4 Beine: Petra ist eine Katze → Petra hat 4 Beine
 - Alle Katzen miauen: Petra ist eine Katze → Petra kann miauen



Datentypen

- Ein Wert gehört immer zu einem bestimmten Datentyp
 - Der Wert **2** ist eine *ganze Zahl* (ein *Int*, Abkürzung für *Integer*)
 - Der Wert **'Python'** ist eine *Zeichenkette* (ein *String*)
 - Der Wert **3.14159** ist eine *Fliesskommazahl* (ein *Float*, Abkürzung für *Floating Point Number*)



Datentypen

Datentyp in Python	Beispiele
Integer (ganze Zahl)	-2, 1, 0, 1, 2, 3, 4, 5
Float (Fließkommazahl)	-4.5592, -1.0, 0.421, 1.4234, 3.14
Strings (Zeichenketten)	'hello', 'Giuseppe', 'Python', '3 Musketiere'



Aufgabe • Datentypen

[Aufgabe]

Wert	Datentyp
'Hello, there!'	?
3.14	?
9000	?
'4123.314239'	?

Datentyp ermitteln

- Mit der Hilfe von `type(wert)` können wir herausfinden, zu welchem Datentyp ein bestimmter Wert gehört

```
print(type('Hello, there!'))  
print(type(3.14))  
print(type(9000))
```

CODE

INTERPRETER

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

PYCHARM



Variablen

Variablen

- Eine *Variable* ist wie eine beschriftete Box, in welcher wir einen Wert verstauen (*speichern*) können
 - Eine Variable besteht aus einem *Namen* und einem zugewiesenen *Wert* (inklusive *Datentypen*)
- Ein Wert kann man mittels dem `=` Operator in eine Variable speichern / einer Variable zuweisen
 - Beispiel: `variablen_name = 'Ein möglicher Wert'`
- Eine Variable hat auch einen Typ, welchen wir mittels `type(variablen_namen)` herausfinden können

```
answer = 42
name = 'Giuseppe'

print(answer)
print(name)
print(type(name))
```

CODE

INTERPRETER

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

PYCHARM

Variablen

```
answer = 42
name = 'Giuseppe'

print(answer)
print(name)
print(type(name))
```

CODE

INTERPRETER

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

PYCHARM

- Versuchen wir den obigen Code als eine Liste von Befehlen anzuschauen:
 - Die Anweisung `answer = 42` sagt «weise der Variable `answer` die Ganzzahl (*Int*) `42` zu»
 - Die Anweisung `name = 'Giuseppe'` sagt «weise der Variable `name` die Zeichenkette (*String*) `'Giuseppe'` zu»
 - Die Anweisung `print(answer)` sagt «gib bitte den Wert der Variable `answer` aus»

Regeln für Variablennamen

– Regeln für Variablennamen:

1. Variablenname ist ein einzelnes Wort (keine Leerzeichen)
2. Variablenname darf nur Buchstaben, Zahlen und Underscore (_) enthalten
3. Variablenname darf nicht mit einer Zahl beginnen
4. Variablennamen sind «case-sensitive».

Beispiel: `ein_wert` und `ein_Wert` sind zwei verschiedene Variablen

Gültige Variablennamen	Ungültige Variablennamen
<code>Mein_name</code>	<code>Mein-name</code>
<code>meineStadt</code>	<code>Meine Stadt</code>
<code>_privat</code>	<code>5123privat</code>
<code>GROSS</code>	<code>GROS\$</code>



Wählt aussagekräftige Variablennamen

- Verwendet möglichst aussagekräftige Variablennamen
 - Nicht so aussagekräftig: `string1 = 'Giuseppe'`
 - Aussagekräftig: `name = 'Giuseppe'`



Anweisungen und Ausdrücke

Anweisungen («Statements»)

- Eine *Anweisung* ist eine Instruktion (oder Befehl), welche der Python-Interpreter ausführen kann
 - Beispiel: Die Wertzuweisung **sprache** = 'Python' ist eine Anweisung
- Ein Code kann eine Folge von Anweisungen enthalten;
der Python-Interpreter führt dabei jede Zeile von oben nach unten einzeln aus

```
print('Gib x aus:')  
x = 2  
print(x)
```

CODE

INTERPRETER

```
Gib x aus:  
2
```

PYCHARM

Die Auswertung von Ausdrücken («Expressions») [Wichtiges Konzept]

- Ein *Ausdruck* ist eine Kombination von Werten, Variablen, und Operatoren
- Ein Ausdruck kann immer ausgewertet werden, d.h. der Ausdruck wird zu einem Wert evaluiert
- In einem Code ist ein alleinstehender Ausdruck eine legale Anweisung

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

INTERPRETER

7

PYCHARM

- **zahl2 = zahl1 + 3** ist eine Anweisung, wobei sich nach dem Gleichzeichen (=) ein Ausdruck befindet

Operatoren

- *Operatoren* sind spezielle Symbole, die z.B. Berechnungen wie die Addition oder Multiplikation darstellen
- Werte, die durch Operatoren verknüpft werden, heissen *Operanden*
- *Wichtig*: 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

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

CODE

INTERPRETER

```
43
60
Ein Beispiel
```

PYCHARM

Eingaben einlesen

- Wir können auch Eingaben einlesen, d.h. wir können eine Zeichenfolge eingeben und z.B. in eine Variable speichern
- Mittels `input()` können wir verlangen, dass eine Zeichenfolge eingelesen wird

```
print("Bitte etwas eingeben: ")  
meine_eingabe = input()  
print(meine_eingabe)
```

CODE

INTERPRETER

```
Bitte etwas eingeben: Hallo!  
Hallo!
```

PYCHARM

Kommentare

- Grössere Programmierprojekte können aus mehreren tausenden Zeilen Code bestehen
 - Code wird immer komplizierter zu verstehen / lesen
- Kommentare helfen, den Code verständlicher darzustellen / zu erklären wo nötig
- Kommentare werden mit dem **#** Symbol markiert, und werden vom Interpreter ignoriert

```
pi = 3.14  
r = 4  
# Berechne Fläche von einem Kreis  
# mit Radius r  
print(pi * r ** 2)
```

CODE

INTERPRETER

50.24

PYCHARM



Zeichenketten / Strings

Verkettung von mehreren Strings

- Verwende den **+** Operator um Strings zu verketteten

```
string1 = 'Hallo'  
string2 = ', Welt'  
string3 = '!'

print(string1 + string2 + string3)
```

CODE

INTERPRETER

Hallo, Welt!

PYCHARM

Verkettung von Strings mit Zahlen

- Was geschieht, wenn wir z.B. **'Area'** mit **51** verketteten möchten?

```
print('Area' + 51)
```

CODE

INTERPRETER

Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: must be str, not int

PYCHARM

- *Tipp*: Operatoren nur auf Operanden anwenden, die von ähnlichen Typen (z.B. Zahlen) stammen
 - Im obigen Code versuchen wir den Operator + auf Operanden von komplett verschiedenen Typen anzuwenden (Strings und ganze Zahlen)

Typumwandlungen

- Python bietet die Möglichkeit an, den Typ von Variablen zu ändern
- Wenn ich z.B. eine Variable `zahl` die eine ganze Zahl ist, so kann ich sie mit `str(zahl)` in einen String umwandeln

```
print('Area ' + str(51))
```

CODE

INTERPRETER

```
Area 51
```

PYCHARM

Typumwandlungen

- Wir können auch einen String in eine Zahl umwandeln, angenommen der String beinhaltet nur eine Zahl:

```
print(int('123'))  
print(float('3.14'))
```

CODE

```
# Funktioniert nicht!  
print(int('3.14'))
```

INTERPRETER

123

3.14

ValueError: invalid literal for
int() with base 10: '3.14'

PYCHARM



Mathematische Datentypen – ganze Zahlen und Fließkommazahlen

Mathematische Operatoren und die Vorrangregeln

- Eine Anweisung kann aus mehreren mathematischen Operatoren bestehen
- Reihenfolge der Auswertung hängt von den folgenden Vorrangregeln ab (höchster Vorrang oben):

Operator	Operation	Beispiel	Resultat
(...)	Klammern	$(2 * 3) ** 2$	36
**	Exponent	$2 ** 3$	8
%	Modulus	$22 \% 10$	2
/	Division	$12 / 4$	3
*	Multiplikation	$10 * 2$	20
-	Subtraktion	$18 - 8$	10
+	Addition	$1 + 1$	2



Aufgabe • Vorrangregeln

[Aufgabe]

Ausdruck	Ergebnis
$4 * 5 / 2$?
$(6 / 2) * 4 + 3$?
$2 ** (1 + 2) + 3$?
$2 ** 3 + 2 * 3$?

Addition einer Fließkommazahl mit einer ganzen Zahl

- Fließkommazahlen (**float**) enthalten in der Regel viel mehr Informationen (Nachkommastellen) als Integer (keine Nachkommastellen)
- Implizite Typumwandlung zum informationsreicheren Datentyp, nämlich **float**

```
gnz_zahl = 1000
flkom_zahl = 2.4813
summe = gnz_zahl + flkom_zahl

print(type(summe))
print(summe)
```

CODE

INTERPRETER

```
<class 'float'>
1002.4813
```

PYCHARM

Addition zweier ganzen Zahlen

```
gnz_zahl1 = 1000
gnz_zahl2 = 3000
summe = gnz_zahl1 + gnz_zahl2

print(type(summe))
print(summe)
```

CODE

INTERPRETER

```
<class 'int'>
4000
```

PYCHARM



Multiplikation einer Fließkommazahl mit einer ganzen Zahl

```
gnz_zahl = 3
flkom_zahl = 4.5
mult = gnz_zahl * flkom_zahl

print(type(mult))
print(mult)
```

CODE

INTERPRETER

```
<class 'float'>
13.5
```

PYCHARM



Multiplikation zweier ganzen Zahlen

```
gnz_zahl = 4
flkom_zahl = 5
mult = gnz_zahl * flkom_zahl

print(type(mult))
print(mult)
```

CODE

INTERPRETER

```
<class 'int'>
20
```

PYCHARM

LC 1.1 • Durchschnittslohn berechnen

{Live Coding}

Lasst uns gemeinsam ein Programm schreiben, dass folgende Anweisungen ausführt:

1. Es sind 50 Franken verfügbar. Speichert diesen Wert in die Variable `anz_franken` ab
2. Des Weiteren arbeiten gerade 4 Helfer. Speichert diesen Wert in die Variable `anz_helfer` ab
3. Nun möchten wir herausfinden, wie viele Franken jeder Helfer erhält.
Schreibt eine Anweisung (auf einer Zeile), die diesen Wert berechnet und ihn in die Variable `anz_franken_pro_helfer` speichert
4. Gebt die Variable `anz_franken_pro_helfer` auf dem Bildschirm aus

Division zweier Zahlen – Regeln

- Division zweier Zahlen mittels `/` Operator wird in Python 3.* automatisch zu einem Float umgewandelt
- Der Operator `//` kann hingegen verwendet werden, um ein Ganzzahl Ergebnis zu generieren

```
div1 = 1 / 2  
div2 = 1 // 2
```

```
print(type(div1))  
print(div1)
```

```
print(type(div2))  
print(div2)
```

CODE

INTERPRETER

```
<class 'float'>  
0.5  
<class 'int'>  
0
```

PYCHARM

Typumwandlung

- Python bietet die Möglichkeit an, den Typ von Variablen zu ändern
- `int(zahl)` → `zahl` wird in ganze Zahl umgewandelt (kann zu Informationsverlust führen)
- `float(zahl)` → `zahl` wird zu einer Fließkommazahl umgewandelt

```
flkom_zahl = 1.482392
gnz_zahl = int(flkom_zahl)

print(type(gnz_zahl))
print(gnz_zahl)
```

CODE

INTERPRETER

```
<class 'int'>
1
```

PYCHARM



Lernziele – Check

- Nach dieser Einheit wissen wir:
 1. ... was eine Variable ist
 2. ... wie man einer Variable einen Wert zuweist
 3. ... wie man eine oder mehrere Variablen ausgibt
 4. ... was *Strings*, *Floats*, und *Ints* sind
 5. ... wieso wir Typumwandlungen benötigen



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

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

- Der Boolesche Datentyp erlaubt nur zwei Werte, nämlich **True** oder **False**

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

CODE

Vergleichsoperatoren

Ausdruck	Bedeutung
$x \neq y$	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
$x \geq y$	Ist x grösser oder gleich y? Falls ja, evaluiert zu True , sonst zu False
$x \leq y$	Ist x kleiner oder gleich y? Falls ja, evaluiert zu True , sonst zu False

Logische Operatoren : Mehrere Bedingungen überprüfen

and Logisches UND

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

not Logische Negation

Ausdruck	Wert
not False	True
not True	False

or Logisches ODER

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

Logische Operatoren: Mehrere Bedingungen überprüfen

Ausdruck	Interpretation
$(x > 0) \text{ and } (x < 10)$	Ausdruck evaluiert zu True nur dann wenn x grösser als 0 und x kleiner als 10 ist
$(y < 0) \text{ or } (x < 10)$	Ausdruck evaluiert zu True wenn y kleiner als 0 oder x kleiner als 10 ist (oder beides erfüllt ist)
$\text{not}(x < y)$	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

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

```
alter = 19
```

CODE

Bedingung

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

Snippet 1: Nur if-Anweisung

```
alter = 19
```

CODE

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

Verzweigung

Snippet 2: if-else Kombination

Alternative Ausführung – Mehrfache Verzweigung

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

```
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')
```

CODE

- **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:

```
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')
```

CODE

- 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



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

Funktionen Teil 1 – Ein Einstieg



Lernziele

- Nach dieser Einheit wissen wir:
 1. ... was eine Funktion ist
 2. ... wie wir eine Funktion definieren können
 3. ... wie wir Funktionen um Parameter erweitern können



Funktionen

- Thema ist erfahrungsgemäss anfänglich kompliziert / evtl. schwer verständlich (vorallem auch, weil wir schon seit einigen Stunden hier sind)
- Fragen sind zu jedem Zeitpunkt erlaubt und wirklich erwünscht



Vorschau

```
def hello(first, last):  
    print('Hello,' + first + ' '  
          + last + '!')  
  
hello('Giuseppe', 'Accaputo')
```

CODE

INTERPRETER

Hello, Giuseppe Accaputo!

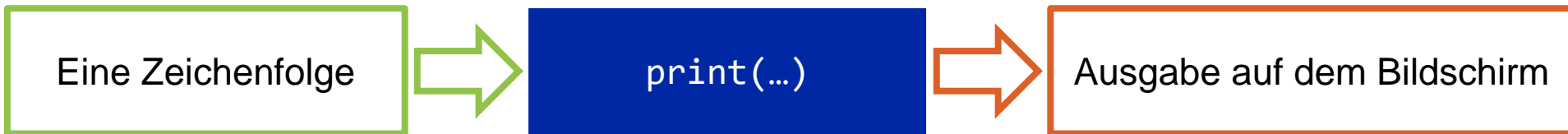
PYCHARM

Funktionen



- Beispiele von Funktionen (Thema für nächste Woche):
 - Funktionen, die eine Ausgabe auf dem Bildschirm generieren, z.B. `print()`
 - Funktionen, die zu einem Wert evaluieren, z.B. mathematische Funktion oder auch `input()`

Die `print()` Funktion



- Eingabe: Eine Zeichenfolge bestehend aus Zeichen, Zahlen, etc.
- Ausgabe: Die Zeichenfolge wird auf dem Bildschirm ausgegeben



Die `print()` Funktion

```
print('Eins')  
print('Zwei')  
print('Drei')
```

CODE

INTERPRETER

```
Eins  
Zwei  
Drei
```

PYCHARM

Eine Funktion definieren

- Eine Funktion ist wie ein Miniprogramm im Programm selbst

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

```
hello()  
hello()
```

CODE

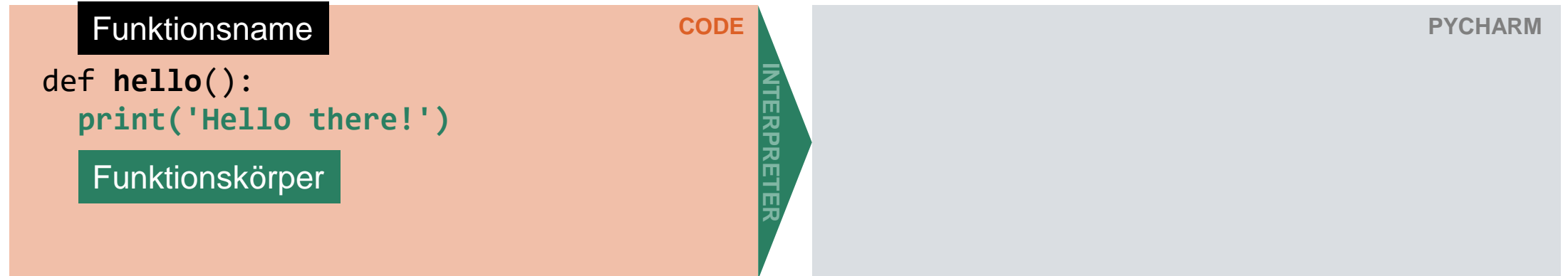
INTERPRETER

```
Hello!  
Hello!
```

PYCHARM

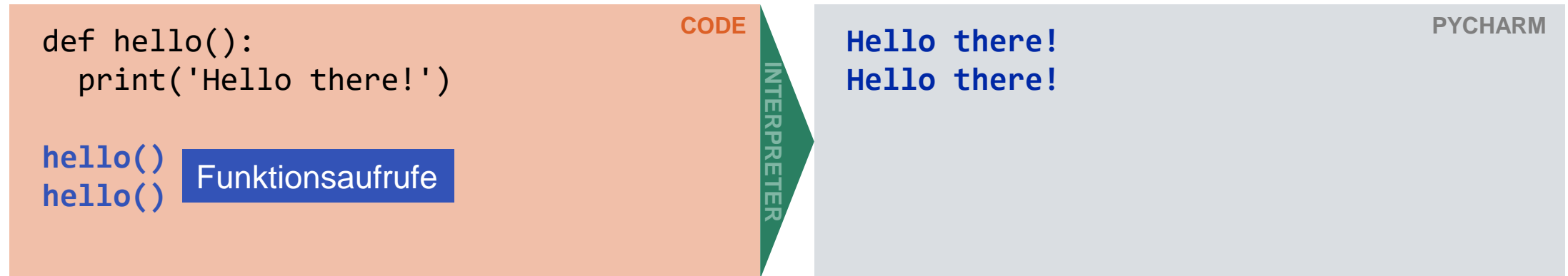
Eine Funktion definieren

- Mittels **def** Keyword kann man eine Funktion definieren
 - Verlangt wird dabei der **Funktionsname** und der **Funktionskörper**
 - Der Code im Funktionskörper wird erst ausgeführt, wenn die Funktion aufgerufen wird
 - *Wichtig:* Die Definition einer Funktion wird lediglich *registriert*



Eine Funktion aufrufen

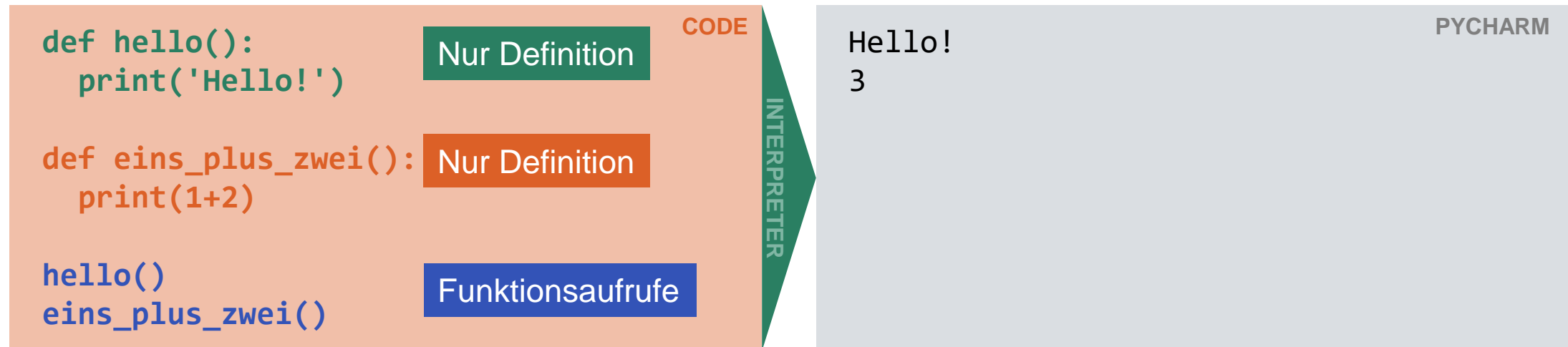
- Eine selbst-definierte **Funktion kann man aufrufen**, indem man den Funktionsnamen gefolgt von einem Klammerpaar im Code tippt:



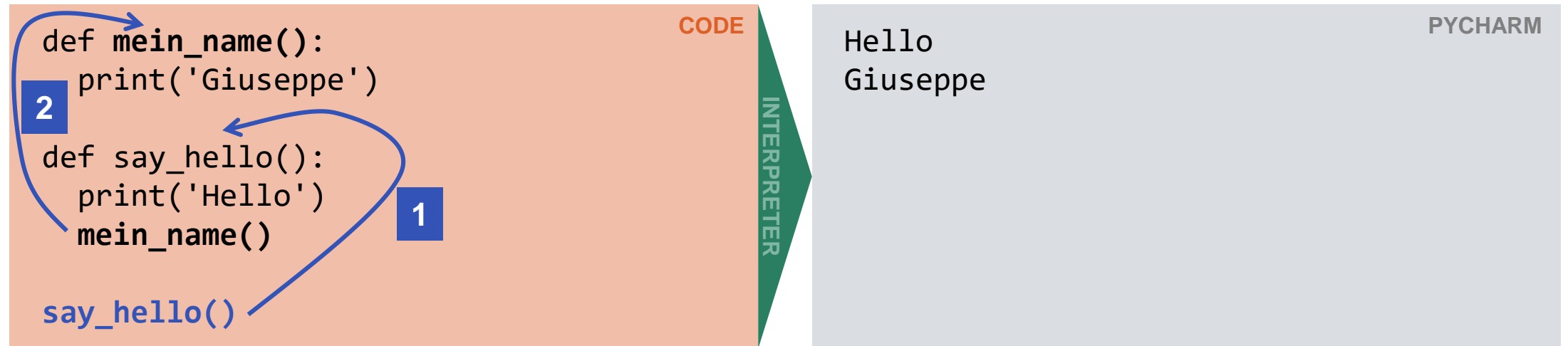
- Erst bei einem Funktionsaufruf wird die Funktion (bzw. der dazugehörige Funktionskörper) ausgeführt

Mehrere Funktionen

- Nachdem die Funktion ausgeführt / evaluiert wurde, geht der Code an der Stelle weiter, an welcher der Funktionsaufruf getätigt wurde
- Im folgenden Code wird zuerst `hello()` ausgeführt, und danach `eins_plus_zwei()`:



Funktionen in Funktionen aufrufen



Strukturierung durch Einrückung

- Anweisungen, die zusammen gehören, müssen die gleiche Einrückungstiefe haben. Dabei kann man z.B. zwei Leerzeichen für solch eine Einrückungstiefe verwenden:

CODE

```
def grosse_funktion():  
    print('Eine')  
    print('grosse')  
    print('Funktion')  
  
def kleine_funktion():  
    print('Ganz klein')  
  
print('Gehört zu keiner Funktion!')  
  
def weitere_funktion():  
    print('Noch eine Funktion!')
```


Strukturierung durch Einrückung

- Anweisungen, die zusammen gehören, müssen die gleiche Einrückungstiefe (**Gelb**) haben. Dabei kann man z.B. zwei Leerzeichen für solch eine Einrückungstiefe verwenden:

CODE

```
def grosse_funktion():  
    print('Eine')  
    print('grosse')  
    print('Funktion')
```

```
def kleine_funktion():  
    print('Ganz klein')
```

```
print('Gehört zu keiner Funktion!')
```

```
def weitere_funktion():  
    print('Noch eine Funktion!')
```



Aufgabe • Ausgaben verstehen

[Aufgabe]

- Was wird bei Ausführung des folgenden Codes ausgegeben?

CODE

```
def funktion_a():  
    print("A")
```

```
def funktion_b():  
    print("B")  
    funktion_a()
```

```
def funktion_c():  
    funktion_b()  
    print("C")  
    funktion_a()
```

```
funktion_c()
```

Einer Funktion ein Parameter übergeben

- Einer Funktion kann man *Werte* mitgeben. Diese nennt man *Parameter* oder *Argument* einer Funktion
 - Beispiel: `print('Hello')`
 - Wir übergeben der Funktion `print` den Wert (oder das Argument) `'Hello'`; dieser wird anschliessend ausgegeben
 - Wir möchten also der `print` Funktion mitteilen, welchen String sie ausgeben soll
- Bezogen auf die Funktion `summe(a,b,c)` sind die folgenden Aussagen korrekt:
 1. Die Funktion `summe` nimmt 3 Parameter entgegen
 2. Wir können die Funktion `summe` mit 3 Parameter aufrufen (oder: Wir können der Funktion `summe` 3 Parameter übergeben)

Einer Funktion ein Parameter übergeben

- Wir wollen nun ermöglichen, dass der Funktion `hello` der Wert `'Giuseppe'` übergeben werden kann, und danach `'Hello, Giuseppe!'` ausgegeben wird
- Wie erreichen wir das? Unser Ziel:

```
hello('Giuseppe')
```

CODE

INTERPRETER

```
Hello, Giuseppe!
```

PYCHARM

Funktion um ein Parameter erweitern

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

CODE

INTERPRETER

Hello, Giuseppe!

PYCHARM

- Die Funktion **hello** wird um den Parameter **name** erweitert
- Der Parameter ist dabei eine Variable, in welcher der Wert des Parameters (z.B. **'Giuseppe'**) gespeichert wird
- Wenn wir also **hello('Giuseppe')** aufrufen, so wird beim Eintritt in die Funktion **hello** der Variable **name** der Wert **'Giuseppe'** zugewiesen («**name = 'Giuseppe'**»)

Gültigkeitsbereich («Scope») von Parametern

CODE

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

- Die Variable `name` ist nur in der Funktion `hello` gültig und kann von ausserhalb nicht verwendet werden
 - *Wichtig:* Unbedingt auf Einrückungstiefe (`Gelb`) achten

Gültigkeitsbereich («Scope») von Parametern

```
def hello(name):  
    print('Hello,' + name + '!')  
  
hello('Giuseppe')  
print(name) # nicht möglich!
```

CODE

INTERPRETER

```
Hello, Giuseppe!  
NameError: name 'name' is not defined
```

PYCHARM

- Die Variable **name** ist nur in der Funktion **hello** gültig und kann von ausserhalb nicht verwendet werden
 - *Wichtig:* Unbedingt auf Einrückungstiefe achten: **name** ist nur im markierten Bereich verwendbar

Funktion um mehrere Parameter erweitern

```
def hello(first, last):  
    print('Hello,' + first + ' '  
          + last + '!')  
  
hello('Giuseppe', 'Accaputo')
```

CODE

INTERPRETER

Hello, Giuseppe Accaputo!

PYCHARM

- Die Funktion **hello** kann nun mit zwei Parameter aufgerufen werden
- Wir können auch Funktionen mit mehr als zwei Parameter definieren (getrennt mittels Komma)



Lernziele – Check

- Nach dieser Einheit wissen wir:
 1. ... was eine Funktion ist
 2. ... wie wir eine Funktion definieren können
 3. ... wie wir Funktionen um Parameter erweitern können



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

Funktionen Teil 2 – Rückgabewerte, Wiederverwendbarkeit, und mehr



Lernziele

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



Mathematische Funktionen

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

```
import math
```

```
log_e = math.log(10.0)
```

```
wurzel = math.sqrt(25)
```

CODE

- Übersicht der verfügbaren Funktionen: <https://docs.python.org/3/library/math.html>

Die Auswertung von Funktionen [Wichtiges Konzept]

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

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

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

CODE

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

Die Auswertung von Funktionen [Wichtiges Konzept]

- 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:

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

CODE

```
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, Booleans, 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

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

CODE

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

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

CODE

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 Auswertung:
wir evaluieren von innen nach aussen

```
def summe(a, b):  
    return a + b  
  
def produkt(x, y):  
    return x * y  
  
resultat = summe(3, produkt(4, 5))
```

CODE

Boolesche Funktionen

- 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 entgegennehmen
- 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 einen Wert zurückgeben können
 - ...wie wir mit einem Rückgabewert weiterarbeiten können
 - ...mathematische Funktionen schreiben, die gewisse Berechnungen durchführen und Resultate liefern



**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



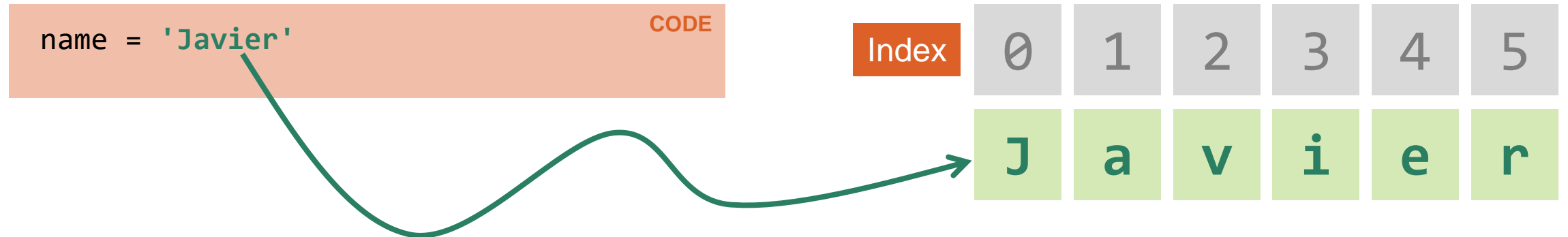
**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

Datenstrukturen Teil 1 – Mit Strings Zeichenketten darstellen

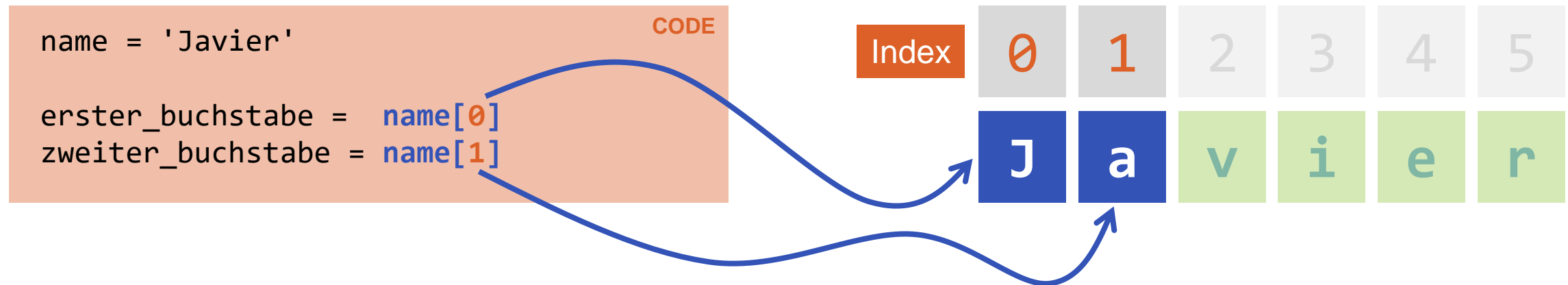
Strings – Eine Sequenz von Zeichen

- Ein String ist eine *Sequenz von Zeichen*:



Strings – Eine Sequenz von Zeichen

- 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")

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

```
name = 'Javier'  
  
for zeichen in name:  
    print(zeichen)
```

CODE

String Segmentierung

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

```
name = 'Javier'
```

CODE

```
# Zeichen 1, 2, und 3
```

```
segment1 = name[1:4]
```



```
name = 'Javier'
```

CODE

```
# Alle Zeichen ab Index 3
```

```
segment2 = name[3:]
```



Strings sind unveränderbar

- Wir können einen existierenden String nicht verändern:

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

CODE

INTERPRETER

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

PYCHARM

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

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

CODE

INTERPRETER

Jython

PYCHARM

Negative Indizes und die Länge eines Strings

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

Negativer Index:

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

```
name = 'Javier'
```

CODE

```
letztes_z = name[-1]
```

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

```
name = 'Javier'
```

CODE

```
laenge = len(name)
```

Strings vergleichen und durchsuchen

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

```
name = 'Python'

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

CODE

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

```
text = 'Ein Fehler ist aufgetreten'

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

CODE



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

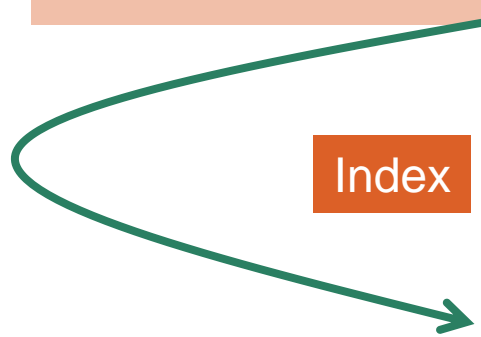
Datenstrukturen Teil 2 – Listen

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

```
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]
```

CODE



Index	0	1	2	3	4	5
	"a"	2	3.0	"b"	"c"	6.4
Typ	str	int	flt	str	str	flt

Listen – Auf einzelne Elemente zugreifen

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

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

CODE

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

Listen sind veränderbar

- Im Gegensatz zu Strings sind Listen jedoch *veränderbar*.

```
liste = ["eins", 2, 3, "vier"]  
print("alt:", liste)  
  
# Ändere den Wert des ersten Elements  
liste[0] = 1  
  
print("neu:", liste)
```

CODE

Eine Liste durchqueren – Wir besuchen alle Elemente einzeln

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

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

CODE

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


Negative Indizes und die Anzahl Elemente in einer Liste

- Wir können auch bei Listen negative Indizes verwenden

```
liste = ["a", "b", "c"]  
  
letztes_e = liste[-1]
```

CODE

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

```
liste = ["a", "b", "c"]  
  
laenge = len(liste)
```

CODE

Listen vergleichen und durchsuchen

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

```
liste1 = ["a", "b", "c"]  
liste2 = ["a", "b"]  
  
if liste1 == liste2:  
    print('Beide Listen sind gleich!')
```

CODE

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

```
liste = ["a", "b", "c"]  
  
if "a" in liste:  
    print('"a" kommt vor!')
```

CODE



Operationen mit Listen: Verkettung

- Mit dem **+** Operator können zwei Listen verkettet werden:

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

CODE

Operationen auf Listen: Einfügen von Elementen

- 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:

```
liste = []  
  
liste.append("a")  
liste.append("b")
```

CODE

Operationen auf Listen: Elemente entfernen

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

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

CODE

```
# 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

- Mit **sort** können wir die Elemente einer Liste sortieren:

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

CODE

```
liste.sort()
```



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

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*

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

CODE

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


Tupel als Rückgabewert

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

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

CODE

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

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

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

CODE



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

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

- Syntax um ein Dictionary zu definieren:

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

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



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

CODE

Auf Dictionary Einträge zugreifen

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

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

CODE

- **Neu:** `Index` ist jetzt vom Typ `String`

Elemente einfügen

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

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

CODE

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

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

Ein Dictionary durchqueren

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

```
woerterbuch = {...}
```

CODE

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


Dictionary-Elemente zählen, suchen, und löschen

- 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:

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

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



**Universität
Zürich** ^{UZH}

Zentrale Informatik – IT Fort- und Weiterbildungen

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

- 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

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

CODE

INTERPRETER

2

PYCHARM

- 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

- Naive Implementation eines Countdowns:

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

CODE

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

Ein Countdown: Verbesserte Version

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

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

CODE

n ist **3**
Solange **n** grösser als **0** ist:

- Gib **n** aus
- Verkleinere **n** um **1**

Gib **Los geht's** aus

PSEUDOCODE

Die **while** Schleife

- Repetitive Aufgaben können wir mittels **while**-Schleife ausführen:

while [Bedingung erfüllt ist]:

SYNTAX

- Führe hier Anweisungen aus
- Bedingung muss hier aktualisiert werden

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

n ist 3

PSEUDOCODE

Solange **n** grösser als 0 ist:

- Gib **n** aus
- Verkleinere **n** um 1

Gib **Los geht's** aus

n = 3

CODE

while **n** > 0:

print(**n**)

n = **n** - 1

print("Los geht's!")

Aus einer Schleife ausbrechen – Die **break** Anweisung

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

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

    if eingabe == 'Bitte':
        break

    print('Falsche Eingabe...')

print('Ende!')
```

CODE



Weitere Schleifenart: Die **for** Schleife

- Diese Schleifenart haben wir im Zusammenhang mit den Datenstrukturen kennengelernt



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/>