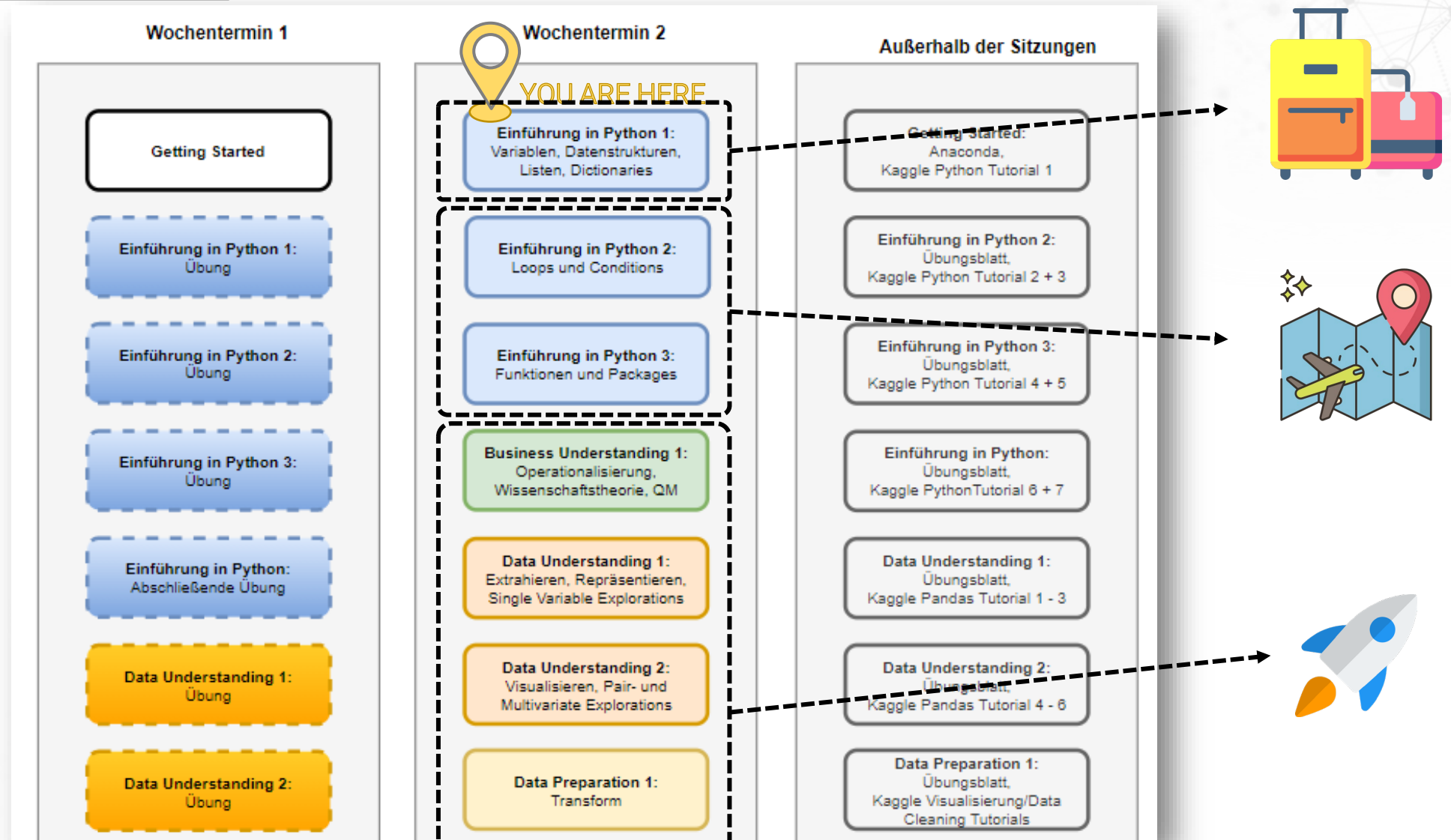


# Python Crashkurs

---

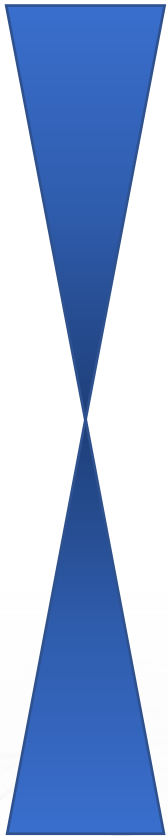
Variablen, Datentypen und -strukturen

# Wo sind wir?



# Agenda

---

- 
1. Variablen
  2. Datentypen und Operationen
    - i. Zahlen: Int, Float
    - ii. Arithmetische Operationen
    - iii. Booleans und Vergleichsoperatoren
    - iv. Strings
  3. Listen
  4. Dictionaries

# Agenda

1. Variablen
2. Datentypen und Operationen
  - i. Zahlen: Int, Float
  - ii. Arithmetische Operationen
  - iii. Booleans und Vergleichsoperatoren
  - iv. Strings
3. Listen
4. Dictionaries



Lernziele

1. Wir können mit Variablenzuweisung umgehen
2. Wir verstehen die Datenstrukturen Listen und Dictionaries
3. Wir können Manipulationen und Operationen von Datentypen und -strukturen ausführen

# Variablen



```
[1]: message = "Hello World!"  
     print(message)  
Hello World!
```

- Aus unserem „Hello World!“ kennen wir den Print-Befehl
- Wir nutzen nun eine *Variable*, um das gleiche Ergebnis zu erhalten
- Eine Variable ist ein Bereich im Memory, in dem der Wert gespeichert ist
- Der Wert in einer Variablen kann verändert werden
- Der Wert der Variable ist der *String* „Hello World!“



*So what?*

Jeder *Variable* ist ein *Wert* zugeordnet. Die Zuordnung findet in Python über „`=`“ statt.

# Variablen: Regeln für die Anwendung

```
[5]: # Right
name = "Max"
student_name = "Max"
name_length = len(name)
```

```
[6]: # Not so right
n = "Max"
s_n = "Max"
length_of_persons_name = len(name)
```

So what?

„Gute“ Benennung von Variablen  
kommt mit der Übung!

- Variablennamen bestehen aus:
  - Buchstaben
  - Zahlen
  - Unterstrichen
- Können nicht mit Zahl beginnen
- Unterstriche ersetzen Leerzeichen
- Sog. *Reserved Words* sollten nicht verwendet werden
- Variablennamen sollten kurz, aber aussagekräftig sein
- Aber: Lesbarkeit geht vor Kürze!



Was denken Sie?

Für wen schreiben  
Sie Programme?

# Variablen: Reserved Words

False class return is finally  
None if for lambda continue  
True def from while nonlocal  
and del global not with  
as elif try or yield  
assert else import pass  
break except in raise

# Variablen

```
[1]: message = "Hello World!"  
     print(message)
```



- Ein *Traceback* zeigt uns die Stelle des Fehlers
- *NameError* bezeichnet die Art des Fehlers
- Und wir bekommen auch noch eine Beschreibung was schief gelaufen ist

Hinweis

Gewöhnen Sie sich daran, dass Fehler häufig passieren!

Google und StackOverflow sind Ihre Freunde



Demo  
How to  
Zeilennummern in  
JupyterLab



*So what?*

*NameError* kann z.B. bedeuten, dass die Variable nicht gesetzt ist oder wir sie falsch geschrieben haben.



# Beispiel

---

**Beispiel 1:** Wir weisen der Variablen `message` einen anderen Wert zu und lassen ihn uns ausgeben.

# Datentypen: Zahlen und Operationen

1	2 + 3
5	
1	2 * 3
6	
1	2 - 3
-1	
1	2 / 3
0.6666666666666666	
1	2 ** 3
2	
8	

1	10 % 3
1	

1	2 * 3.0
6.0	

1	0.1 + 0.1
0.2	

## Integers und Floats

- Integers: sind ganze Zahlen
- Floats: Zahlen mit Gleitkommastelle

## Operationen

- Addition: +
- Subtraktion: -
- Multiplikation: \*
- Division: /
- Exponent: \*\*
- Modulo: %

Punkt-vor-Strich wird berücksichtigt und Klammern sind möglich

Wenn Operationen nicht als Integer repräsentiert werden können, dann resultiert das immer in eine Float.

Sobald eine Float in Operationen verwendet wird, erhalten wir eine Float

# Datentypen: Boolean und Vergleichsoperatoren

1	1 < 2	True
1	1 <= 2	True
1	1 > 2	False
1	1 >= 2	False

1	1 == 2	False
1	1 != 2	True
1	1 + True	2

## Datentyp Logisch

- Führt man in Python Vergleiche durch, dann erhält man als Ausgabe entweder wahr (`True`) oder falsch (`False`)
- Eine Boolean Variable kann also nur zwei Ausprägungen haben
- Da Python dynamisch ist, kann man Booleans und Zahlen einfach mit Operationen verknüpfen

## Vergleichsoperatoren

- Wie in der Mathematik gibt es in Python die typischen Vergleichsoperatoren

<	kleiner als
<=	kleiner als oder gleich
>	größer als
>=	größer als oder gleich
==	gleich
!=	ungleich

# Datentypen: Boolean und Vergleichsoperatoren

```
1 x = 5
2 (x > 3) and (x < 10)
```

```
1 x = 5
2 (x == 3) or (x < 10)
```

## Verknüpfung von logischen Ausdrücken

- Wie in der Mathematik auch können in Python logische Ausdrücke miteinander verknüpft werden
- Hierzu gibt es den UND Operator

and

und den OR Operator

or

# Beispiel: Planetare Waage

Wir wollen wissen, wie viele Erden wir in einen Topf schmeißen müssen, um das Gewicht von Jupiter zu erreichen. Wir wollen hierzu Python wie eine Waage benutzen. Bei einer Waage legt man auf einer Seite so lange Gewichte in die Schale, bis sie das Gewicht in der anderen Schale gerade erreicht - bzw. gerade überschreitet. Das können wir durch geschicktes Einsetzen der Vergleichsoperatoren erzeugen.

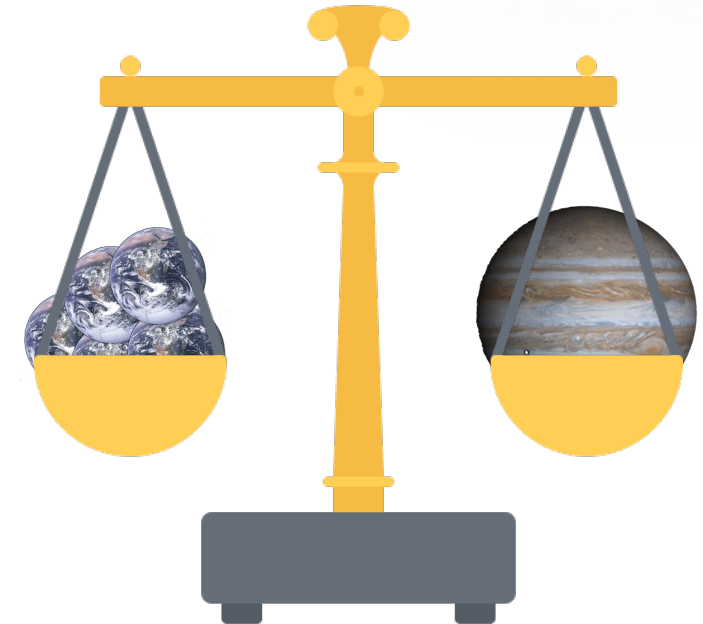
Hierzu kennen wir die Durchmesser der Planeten:

- Durchmesser Erde: 12756 km
- Durchmesser Jupiter: 142984 km

Und deren mittlere Dichten:

- Dichte Erde:  $5.51 \cdot 10^3 \text{ kg/km}^3$
- Mittlere Dichte Jupiter:  $1.33 \cdot 10^3 \text{ kg/km}^3$

Wir weisen hierzu den Variablen `earth` und `jupiter` jeweils ihre Durchmesser zu. Dann berechnen wir die Gewichte und entscheiden durch Trial and Error und Vergleichsoperatoren, wann genügend Erden zusammengekommen sind.



Planet fotos from wikipedia

# Datentypen: Strings

```
1 my_string_double_quotes = "Einführung in Python"
2 my_string_single_quotes = 'Einführung in Python'
3 print(my_string_double_quotes)
4 print(my_string_single_quotes)
```

Einführung in Python  
Einführung in Python

```
1 my_string = 'Eine "einfache" Einführung in Python'
2 print(my_string)
```

Eine "einfache" Einführung in Python

- Ein *String* ist aus *Characters* zusammengesetzt
- Alles innerhalb von Single und Double Quotes wird als String interpretiert  
→ Flexibilität

# Beispiel: automate the boring stuff: Excel

Stellen Sie sich vor in dem Unternehmen, in dem Sie arbeiten, werden von verschiedenen Mitarbeitern Excel-Listen von Bauteilen gepflegt. Jeder trägt hier nach Belieben die Bauteilnamen mit großem oder kleinem Anfangsbuchstaben ein. Sie müssen dafür sorgen, dass die Eintragungen einheitlich mit großem Anfangsbuchstaben vorgenommen werden. Hierzu haben Strings eigene **Methoden**, die dies bewirken. Wir wenden `.title()`, `.upper()`, `.lower()` auf unsere vordefinierten Strings an.

```
1 excel_cell_1 = 'engine'  
2 excel_cell_2 = 'Conveyor'  
3 excel_cell_3 = 'wheel'
```

Definition

*Methoden* sind Aktionen, die auf ein Objekt angewendet werden. Der Punkt `.` sagt Python, dass die Methode auf das vorhergehende Objekt angewendet werden soll. Die Klammern `()` rufen sie auf und in ihr können sich *Eingabeargumente* befinden.

# Datentypen: Strings und Variablen verknüpfen

```
1 first_name = "Chuck"
2 last_name = "Norris"
3 full_name = f"Mein Name ist {first_name} {last_name}"
4 print(full_name)
```

Mein Name ist Chuck Norris

## State-of-the-art String Formatting:

### f-Strings

- seit Python 3.6 mit dabei
- „formatted string literals“ oder auch „fast strings“



# Datentypen: Strings und Variablen verknüpfen

```
1 answer_to_everything = f"The answer to everything is {7 * 6}."
2 print(answer_to_everything)
```

The answer to everything is 42

```
1 answer_to_everything = f"The answer to everything is {round(42.2)}."
2 print(answer_to_everything)
```

```
1 first_name = "Chuck"
2 last_name = "Norris"
3 full_name = f"{first_name} {last_name}"
4 print(f"Mein Name ist {full_name.upper()}")
```

## Anwendung von f-Strings

- Alle validen Python Ausdrücke können f-Strings mitgegeben werden
  - Numerische Operationen
  - Funktionen
  - Methoden

# Datentypen: Strings und Operationen

```
1 programming_language = "python"
2 version = "3.8"
3 used_programming_language = programming_language + " " + version
4 print(used_programming_language)
```

python 3.8

0 1 2

```
1 one_letter = used_programming_language[0]
2 print(one_letter)
```

p

```
1 x = 5
2 print(used_programming_language[x - 2])
```

```
1 print(used_programming_language[-2])
```

```
1 length_of_my_string = len(used_programming_language)
2 print(f"Die Länge meines Strings ist: {length_of_my_string} Characters")
```

Die Länge meines Strings ist:

## Concatenation

Zwei Strings, die mit einem +-Zeichen verknüpft sind, werden zusammengefügt

## Indexing

- Strings sind Sequenzen an Characters
- Jeder Character hat eine „Hausnummer“ im String
- Um einzelne Characters zu referenzieren nutzt man eckige Klammern

[ ]

- **Achtung:** Python beginnt bei 0 zu zählen!
- Will man die letzte Stelle referenzieren, dann schreibt man -1, die zweitletzte -2, etc.

## Länge von Strings

Wenn wir die built-in Funktion len() auf einen String anwenden, bekommen wir seine Länge

# Datentypen: Strings und Operationen

```
1 programming_language = "python"
2 version = "3.8"
3 used_programming_language = programming_language + " " + version
4 print(used_programming_language)
```

python 3.8

```
1 print(used_programming_language[0:3])
```

```
1 print(used_programming_language[3:])
```

```
1 print(used_programming_language[:5])
```

```
1 print(used_programming_language[1:7:2])
```

## Slicing

- Es können nicht nur einzelne Buchstaben, sondern Sub-Sequenzen an Characters referenziert werden
- Hierzu nutzt man den Colon-Operator innerhalb eckiger Klammern

[0 : 3]

- Die erste Zahl ist **inklusive**, die zweite **exklusive**
- Lässt man die letzte Zahl weg, dann wird die letzte Stelle angenommen
- Lässt man die erste Zahl weg, dann wird die erste Stelle angenommen
- Fügen wir eine weitere Zahl mit einem Colon-Operator hinzu, dann geben wir die Schrittweite an

[<start\_index> : <end\_index> : <step\_size>]

# Datentypen: Strings und Operationen

```
1 my_string = "I will write shiny code."  
2 my_string.find("shiny")
```

13

```
1 my_string = "I will write shiny code."  
2 my_string.find("beautiful")
```

-1

```
1 my_string = "I will write shiny code."  
2 print(my_string.replace("will", "won't"))
```

I won't write shiny code.

## Suche innerhalb eines Strings

- Mit der `find()` Methode können wir nach Substrings innerhalb eines Strings suchen
- Wir erhalten die Position – den Index – wo der Substring beginnt
- Sollte der Substring nicht vorkommen, dann gibt die Methode eine -1 zurück

## Ersetzen

- Mit der `replace()` Methode kann man Substrings in Strings ersetzen
- Es werden alle auftretenden Substrings ersetzt

# Beispiel: automate the boring stuff: replace domain

---

**Beispiel 4:** Wir schreiben uns ein Programm, das in einer Mailadresse das @-Zeichen findet und alle Buchstaben nach dem @-Zeichen durch oth-regensburg.de ersetzt.

# Einschub: User Input

```
1 your_name = input()
2 print(f"Herzlich willkommen {your_name}!")
```

```
Markus
Herzlich willkommen Markus!
```

## Die input() Funktion

- Mit dieser Funktion kann User-Input angefordert werden
- Der Interpreter wartet so lange, bis der User etwas eingegeben hat und/oder die Enter-Taste gedrückt hat

# Datentypen: Konvertierung zwischen Datentypen

```
1 my_age_as_string = str(36)
2 my_age_as_string
```

'36'

```
1 this_is_when_i_retire = int(my_age_as_string) + 30
2 this_is_when_i_retire
```

66

```
1 float(1)
```

1.0

```
1 int(3.0)
```

3

```
1 int(True)
```

1

```
1 type(my_age_as_string)
```

str

```
1 type(this_is_when_i_retire)
```

int

## Umwandlung in Strings

Mit der `str()` Funktion können Zahlen in Strings umgewandelt werden

## Umwandlung in Zahlen

Mit den Funktionen `int()` und `float()` können Strings und Booleans in Integer und Float umgewandelt werden

## Datentyp abfragen

Den Datentyp einer Variable können Sie durch die Funktion `type()` überprüfen.

# Kommentare

```
1 # This is my comment for the next line
2 my_variable = 5

1 def my_func(x):
2     '''
3     When I have to write a lot in my comment,
4     I use three quote marks.
5     This is mostly done to describe the working of functions.
6     '''
7     return x**2
```

0

## So what?

Gründe für Kommentare:

- Programme werden viel häufiger gelesen als geschrieben – das bedeutet: zwischen Schreiben und Lesen kann eine lange Zeit vergehen
- Beim Programmieren arbeitet man zusammen
- Man vergisst schnell, was man beim Programmieren gedacht hat

## Kommentare allgemein

- Kommentare sind fundamentale Bausteine in (vermutlich) allen Programmiersprachen
- Kommentare sollten immer auf Englisch geschrieben werden
- Python ignoriert jeglichen Kommentar  
→ Kommentare werden auch zum Debuggen verwendet – oder einfach wenn man Code-Zeilen nicht mehr braucht, aber nicht ganz löschen will
- Kommentare beschreiben funktional abgeschlossene Bausteine

## Kommentarzeichen

- Um Kommentare zu schreiben setzt man das Hash-Zeichen an den Anfang der Zeile

```
# Kommentar
```

- Blockkommentare werden innerhalb dreier Quote-Zeichen gesetzt

```
'''
Kommentar
'''
```



# Coding Yoga: Variablen & Datentypen

```
1 a_number = '42'.replace(str(5-1), '3')
2 another_number = int(a_number[0]) + int(a_number[-2]) - 2
3 print(another_number)
```

# Beispiel: Guessing Game

**Beispiel 5:** Wir wollen nun ein Programm schreiben, das eine Zufallszahl zwischen 1-10 zieht und uns diese raten lässt. Wir lassen den User die Zahl eingeben und lassen fünf Ratevorgänge zu. Hierzu **importieren** wir uns die Funktion `randint`. Wir werden später sehen, dass wir diese Aufgabe mit Loops und If-Abfragen sehr einfach erledigen können.

Wir werden in dieser Aufgabe sehen, dass wir hier einige Schritte manuell ausführen, die man automatisieren könnte. Außerdem werden wir sehen, dass das Programm entweder zu früh oder zu spät stoppt.

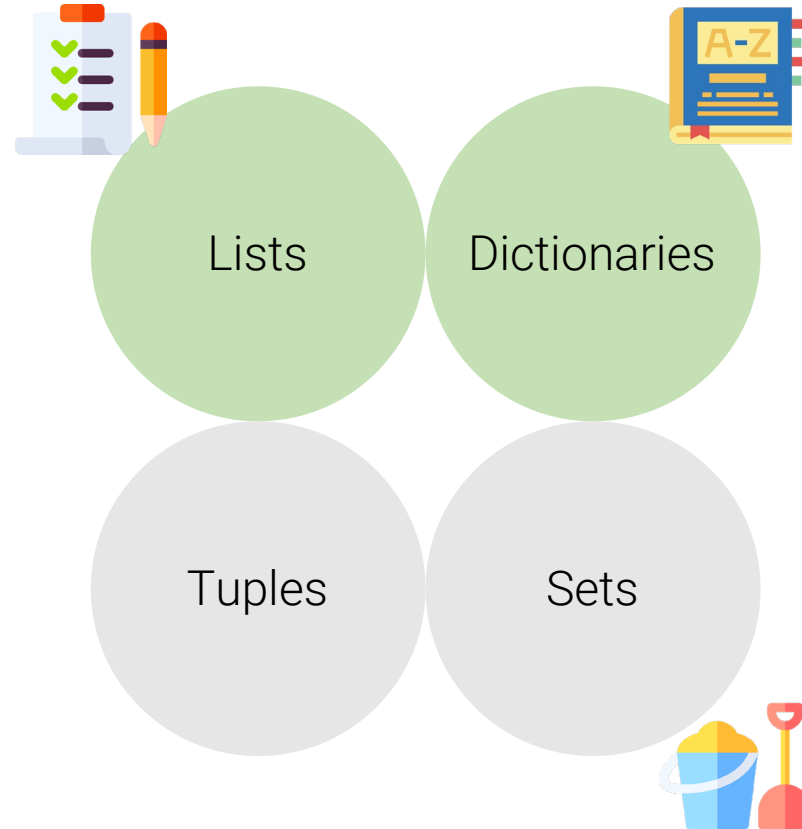
```
1 from numpy.random import randint
2 random_number = randint(1, 10)
```

0

*So what?*

Wir haben bei diesem Beispiel zusätzlich gelernt, dass wir auch Strings auf Gleichheit vergleichen können.  
→ Mit dem „==“ Vergleichsoperator kann man überprüfen ob Strings identisch sind

# Python Datenstrukturen



# Listen



# Was ist eine Liste?

```
1 oth_faculties = ['Angewandte Natur- und Kulturwissenschaften',
2                 'Architektur',
3                 'Bauingenieurwesen',
4                 'Betriebswirtschaft',
5                 'Elektro- und Informationstechnik',
6                 'Informatik und Mathematik',
7                 'Maschinenbau',
8                 'Angewandte Sozial- und Gesundheitswissenschaften']
9 print(oth_faculties)
```

```
['Angewandte Natur- und Kulturwissenschaften', 'Architektur', 'Bauingenieurwesen', 'Betriebswirtschaft', 'Elektro- und Informationstechnik', 'Informatik und Mathematik', 'Maschinenbau', 'Angewandte Sozial- und Gesundheitswissenschaften']
```

```
1 arbitrary_list = ["My hat", 1, [1, 2, 3], "bike"]
2 print(arbitrary_list)
```

```
['My hat', 1, [1, 2, 3], 'bike']
```

```
1 empty_list = []
2 print(empty_list)
```

```
[]
```



Was denken Sie?  
Wie greifen Sie auf eine Liste in einer Liste zu?  
→ Schauen wir uns im JupyterLab an

0

*So what?*

Listen erlauben uns viele Werte in einer Variablen zu speichern

Eine *Liste* ist eine geordnete Menge von Werten. Jeder Wert wird mit einem *Index* angesprochen. Die Werte, die die Liste bilden, werden *Elemente (Items)* genannt.

In Python werden Listenelemente mit eckigen Klammern umgeben und durch Kommas getrennt

```
my_list = [item1, item2, item3]
```

Definition

- Die Elemente müssen in keiner Beziehung zueinander stehen
- Eine Liste kann „alles“ enthalten – sogar andere Listen
- Listen können auch leer sein

# Auf Listenelemente zugreifen

```
1 oth_faculties[0]
```

```
'Angewandte Natur- und Kulturwissenschaften'
```

```
1 oth_faculties[3]
```

```
'Betriebswirtschaft'
```

```
1 oth_faculties[0].upper()
```

```
'ANGEWANDTE NATUR- UND KULTURWISSENSCHAFTEN'
```

```
1 oth_faculties[-1]
```

```
'Angewandte Sozial- und Gesundheitswissenschaften'
```

```
1 len(oth_faculties)
```

```
8
```

## Indexing

- Der Zugriff auf Listenelemente ist analog zum Zugriff auf Characters in einem String
- Da Listen geordnet sind, können wir durch Indizes auf die einzelnen Elemente zugreifen
- Auch bei Listen beginnt Python von 0 an zu zählen
- Auf die einzelnen Elemente kann man dann die Type-spezifischen Methoden anwenden
- Auch bei Listen kann man das letzte Element mit einer -1 erreichen und das vorletzte mit -2 etc.
- Auch die len() Funktion funktioniert bei Listen  
→ Auf viele Python-Objekte anwendbar

# Beispiel

**Beispiel 6:** Uns liegt eine Liste aller Fakultäten der OTH Regensburg vor. Wir wählen eine bestimmte Fakultät durch Indexing aus und lassen uns den Satz

"Ich studiere an der Fakultät <meine Fakultät hier als Listenelement eingefügt>."

ausgeben.

```
1 oth_faculties = ['Angewandte Natur- und Kulturwissenschaften',  
2                 'Architektur',  
3                 'Bauingenieurwesen',  
4                 'Betriebswirtschaft',  
5                 'Elektro- und Informationstechnik',  
6                 'Informatik und Mathematik',  
7                 'Maschinenbau',  
8                 'Angewandte Sozial- und Gesundheitswissenschaften']
```

# Auf Listenelemente zugreifen

```
1 oth_faculties = ['Angewandte Natur- und Kulturwissenschaften',
2                 'Architektur',
3                 'Bauingenieurwesen',
4                 'Betriebswirtschaft',
5                 'Elektro- und Informationstechnik',
6                 'Informatik und Mathematik',
7                 'Maschinenbau',
8                 'Angewandte Sozial- und Gesundheitswissenschaften']
9 print(oth_faculties)
```

```
1 oth_faculties[1:4]
```

```
1 oth_faculties[:3]
```

```
1 oth_faculties[3:]
```

## Slicing

- Auch wie bei Strings können wir Teillisten von Listen referenzieren, indem wir den Colon-Operator verwenden

[ : ]

- Die erste Zahl ist **inklusive**, die zweite **exklusive**
- Lässt man die letzte Zahl weg, dann wird das letzte Element angenommen
- Lässt man die erste Zahl weg, dann wird das erste Element angenommen
- Fügen wir eine weitere Zahl mit einem Colon-Operator hinzu, dann geben wir die Schrittweite an

[<start\_index> : <end\_index> : <step\_size>]

```
1 oth_faculties[3::2]
```



# Listen verändern

```
1 groceries = ['banana', 'apple', 'sugar', 'butter', 'ice cream', 'salad']
2 print(groceries)
```

```
['banana', 'apple', 'sugar', 'butter', 'ice cream', 'salad']
```

```
1 groceries[2] = 'potatoes'
2 print(groceries)
```

```
['banana', 'apple', 'potatoes', 'butter', 'ice cream', 'salad']
```

```
1 groceries[-1] = groceries[2]
2 print(groceries)
```

```
1 groceries.append('tooth paste')
2 print(groceries)
```

```
1 groceries = []
2 groceries.append('chocolate')
3 groceries.append('more chocolate')
4 print(groceries)
```

## Einzelne Elemente verändern

- Durch Indexing können wir einzelnen Listenelementen andere Werte zuweisen (Im Gegensatz zu Strings – die sind unveränderlich!)
- Wir könnten sogar ein Listenelement in ein anderes setzen

## Elemente hinzufügen

- Durch die `append()` Methode kann man Elemente an das Ende der Liste anfügen
- Wichtig, um mit Listen *dynamisch* zu arbeiten  
→ Hierzu werden Listen oft leer *initialisiert*

# Listen verändern

```
1 groceries = ['banana', 'apple', 'sugar', 'butter', 'ice cream', 'salad']
2 del groceries[0]
3 print(groceries)
```

```
1 put_already_in_basket = groceries.pop()
2 print(groceries)
3 print(put_already_in_basket)
```

```
1 groceries.pop(2)
2 print(groceries)
```

```
1 groceries.remove('apple')
2 print(groceries)
```

## Elemente entfernen

- Wenn man die **Position** des zu entfernenden Elements **weiß**, dann nutzt man den `del` Ausdruck

```
del my_list[0]
```

Das Element ist permanent gelöscht.

- Wenn man ein Element löschen, es aber **noch verwenden** will, dann nutzt man die Methode `pop()`
- Die Methode `pop(<index>)` kann auch mit **Indexangabe** verwendet werden
- Wenn man den Wert, aber **nicht** die **Position**, kennt, dann nutzt man die Methode `remove(<value>)`

# Listen verändern

```
1 list_1 = [1, 2, 3, 4]
2 list_2 = [5, 6, 7, 8]
3 concatenated_list = list_1 + list_2
4 print(concatenated_list)
```

## Concatenating Lists

- Wie bei Strings hat auch das +-Zeichen bei Listen einen Sinn: es werden zwei Listen aneinandergefügt (concatenate)

```
1 my_list = [1, 2, 3, 4, 5]
2 multiples_of_my_list = my_list * 5
3 print(multiples_of_my_list)
```

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

## List Repetition

- Mit dem \* Operator können Listen wiederholt aneinandergefügt werden

# Built-in Funktionen und Listen

```
1 my_list = [2, 5, 1, 3, 10]
2 print(max(my_list))
```

```
1 print(min(my_list))
```

```
1 print(sum(my_list))
```

Python hat einige eingebaute Funktionen, die man auf Listen anwenden kann

- Maximum der Werte meiner Liste

`max()`

- Minimum der Werte meiner Liste

`min()`

- Summe der Werte meiner Liste

`sum()`

# Best Friends: Strings und Listen

```
1 message = 'Strings and lists are best friends'
2 parts_of_message = message.split()
3 print(parts_of_message)

['Strings', 'and', 'lists', 'are', 'best', 'friends']
```

```
1 new_message = '-'.join(parts_of_message)
2 print(new_message)

Strings-and-lists-are-best-friends
```

## Die Split Methode

- Mit der `split()` Methode kann man in Python einen String in eine Liste an Substrings zerlegen
- Man kann angeben anhand welches Zeichens man trennen will

## Die Join Methode

- Mit der `join()` Methode kann man eine Liste an Strings miteinander verbinden
- Man kann angeben, mit welchem Zeichen man die Listeneinträge miteinander verbinden will

# Coding Yoga: Listen

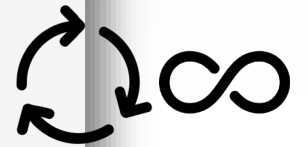
```
1 my_list = [1,3,5,2,3,4,7]
2 i_dont_like_this_one = my_list.pop()
3 and_this_one = my_list.pop()
4 my_list[-3] = i_dont_like_this_one
5 print(sum(my_list))
```

```
1 my_list = [1, 2, 3, 4]
2 value = 2
3 index = 2
4 my_list.remove(value)
5 print(my_list[index])
```

```
1 # Import
2 from numpy.random import randint
3
4 # Initialize
5 list_of_outcomes = []
```

```
1 # Throw dice
2 one_dice = randint(1, 7)
3
4 # Save result
5 list_of_outcomes.append(one_dice)
```

```
1 print(sum(list_of_outcomes)/len(list_of_outcomes))
```



```
1 another_list = [1, 2, 3]
2 another_list = another_list[-2:] + another_list[:3:2]
3 another_list = another_list * another_list[0]
4 print(sum(another_list))
```

# Dictionaries

# Was ist ein Dictionary?

```
1 me = {}  
2 me['height'] = 180  
3 me['hair_color'] = 'blonde'  
4 me['age'] = 36  
5 print(me)  
  
{'height': 180, 'hair_color': 'blonde', 'age': 36}
```

Beachte:  
Wieder leere  
Initialisierung

```
1 print(me['height'])  
  
180
```

```
1 me['height'] = me['height'] + 10  
2 print(me)
```

Ein Dictionary wird in Python durch geschweifte Klammern und Key-Value-Pairs, getrennt durch Kommas, erzeugt. Key-Value-Pairs sind durch einen Doppelpunkt miteinander verbundene Schlüsselwörter und Werte.

```
my_dict = { 'key_1': 0, 'key_2': 1 }
```

Zugriff auf Einträge über eckige Klammern und Key:

```
my_dict['key1']
```

Definition

0

## So what?

- Listen indizieren ihre Elemente anhand der Position
- Dictionaries sind nicht geordnet im Sinne der Position, sondern durch ein Tag – dem Key
- Mit Dictionaries kann man reale Objekte passend abbilden
  - Eine Person und so viel Informationen zu dieser wie nötig – z.B. Alter, Beruf, Größe, etc.
  - Eine Maschine und ihre Komponenten – z.B. Motor, Kugellager, Förderbänder, etc.



# List vs. Dictionary

List

Dictionary

Aufbau

„Key“	Value
[0]	1
[1]	2

```
1 some_dict = {  
2     'key1': 1,  
3     'key2': 2  
4 }  
5 some_list = [1, 2]
```

Key	Value
<b>key1</b>	1
<b>key2</b>	2

Überschreiben

```
some_list[1] = 0
```

```
1 some_dict['key1'] = 0  
2 some_list[0] = 0
```

```
some_dict['key1'] = 0
```

Setzen neuer Werte

```
.append()  
.insert()
```

```
1 some_dict['key3'] = 3  
2 some_list.append(3)
```

```
some_dict['key3'] = 3
```

Error Index/Key

```
1 some_list[3]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-21-4cbc3bc8181d> in <module>  
----> 1 some_list[3]  
  
IndexError: list index out of range
```

```
1 some_dict['key4']
```

```
-----  
KeyError                                Traceback (most recent call last)  
<ipython-input-20-282850914122> in <module>  
----> 1 some_dict['key4']  
  
KeyError: 'key4'
```

# Listen an Keys, Values und beides zusammen

```
1 some_dict.keys()
dict_keys(['key1', 'key2', 'key3', 'key4'])
```

```
1 list(some_dict.keys())
['key1', 'key2', 'key3', 'key4']
```

```
1 some_dict.values()
dict_values([4, 2, 3, 7])
```

```
1 some_dict.items()
dict_items([('key1', 4), ('key2', 2), ('key3', 3), ('key4', 7)])
```

## **.keys()** Methode

Ausgabe es *Iterables* aller Keys in einem Dictionary  
→ Funktion `list()` nutzen, um in eine Liste zu transformieren

## **.values()** Methode

Ausgabe einer Liste aller Werte in einem Dictionary

## **.items()** Methode

Ausgabe einer Liste an Tuple der Keys und Werte in einem Dictionary



### *So what?*

Diese drei Methoden erzeugen Listen, die sehr häufig bei Iterationen verwendet wird  
→ Kapitel Looping!

# Coding Yoga: Dictionaries



```
1 my_dict = {
2     '2': 1,
3     '3': 3,
4     '4': 5
5 }
6 keys_list = list(my_dict.keys())
7 my_dict['5'] = int(keys_list[-2]) ** my_dict['3']
8 print(sum(my_dict.values()))
```

```
1 # A dictionary
2 my_dict = {
3     'I_can_count': [1, 2, 3, 'a lot!'],
4     'python': 'is_like_a_good_cup_of_coffee'
5 }
6
7 # Yoga
8 list_of_keys = list(my_dict.keys())
9
10 print(list_of_keys[0][0] + " "
11       + my_dict['python'][3:7] + " "
12       + list_of_keys[1] + " "
13       + str(my_dict['I_can_count'][2]) + " "
14       + my_dict[list_of_keys[0]][-1])
```

```
1 # Your data
2 sensor_data_dict = {
3     '1602914400000': {
4         'machine_state': 'starting',
5         'batch_data': [34, 36, 40, 41]
6     },
7     '1602921600000': {
8         'machine_state': 'running',
9         'batch_data': [55, 60, 54, 52]
10    }
11 }
12
13 # Yoga
14 list_of_keys = list(sensor_data_dict.keys())
15 print(int(list_of_keys[1]) - int(list_of_keys[0]))
16 print(max(sensor_data_dict[list_of_keys[1]]['batch_data']) - max(sensor_data_dict[list_of_keys[0]]['batch_data']))
```

# Beispiel: Temperaturüberwachung

Ihnen liegen Daten von Temperatursensoren an einer Maschine vor. Zu drei verschiedenen Zeitpunkten und Maschinenzuständen wurden Messreihen aufgezeichnet. Von dieser Messreihe wissen Sie aber, dass jeder zweite Wert nicht verlässlich ist. Ihre Aufgabe ist es festzustellen wie sich die Temperaturen der unterschiedlichen Maschinenzustände im Mittel voneinander unterscheiden.

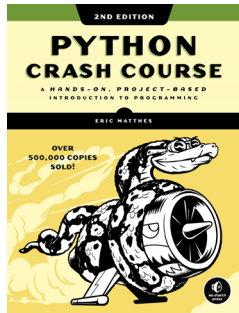


```
1 sensor_data_dict = {
2     '1602914400000': {
3         'machine_state': 'starting',
4         'batch_data': [34, 36, 40, 41, 33, 35, 39, 42, 40, 41, 35, 39]
5     },
6     '1602921600000': {
7         'machine_state': 'running',
8         'batch_data': [55, 60, 54, 52, 50, 55, 53, 61, 60, 59, 56, 57]
9     },
10    '1602928800000': {
11        'machine_state': 'stopping',
12        'batch_data': [65, 67, 70, 72, 70, 65, 73, 71, 68, 69, 66, 67]
13    }
14 }
```

# Literatur und Quellen



[www.py4e.com](http://www.py4e.com)



**Python Crash Course, 2nd Edition**

A Hands-On, Project-Based Introduction to Programming  
by Eric Matthes