



Python Schnelleinstieg

Michael Inden
Freiberuflicher Consultant und Trainer

Speaker Intro



- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg
- ~8 ¼ Jahre SSE bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre TPL, SA bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre LSA / Trainer bei Zühlke Engineering AG in Zürich
- ~3 Jahre TL / CTO bei Direct Mail Informatics / ASMIQ in Zürich
- Freiberuflicher Consultant, Trainer und Konferenz-Speaker
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch

Blog: <https://jaxenter.de/author/minden>

Kurse: Bitte sprecht mich an!

<https://github.com/Michaeli71/Python-Workshop.git>





Agenda

Workshop Contents



- **PART 1: Einstieg Python**
 - Einführung & Trend
 - Syntax im Kurzüberblick
 - Funktionen definieren
 - Module ausführbar machen / main()

 - **PART 2: Strings**
 - Wichtige Funktionen / Methoden
 - Slicing
 - Formatierte Ausgaben
 - Mehrzeilige Strings
 - Pytest Quick Start
-

Workshop Contents



- **PART 3: Klassen & Objektorientierung**
 - Basics
 - Textuelle Ausgaben
 - Gleichheit `is` / `==` und `==` / `equals()`
 - Unterschiede zu Java / Information Hiding
 - Vererbung und Typprüfung
 - Enums
 - Named Tuple / Java Record
-

Workshop Contents



- **PART 4: Collections**
 - Schnelleinstieg list
 - Comprehensions
 - Slicing – Zugriff auf Teilbereiche
 - Schnelleinstieg set
 - Schnelleinstieg dict
 - **PART 5: Collections Advanced**
 - Sortierung – sort() / sorted() + Analogie Comparator
 - Iteratoren / Iterator-Java
 - Reihenfolge umkehren – reverse() und reversed()
 - Generatoren
-

Workshop Contents



- **PART 6: Exception-Handling**
 - Schnelleinstieg
 - Exceptions selbst auslösen
 - Eigene Exception-Typen definieren
 - Propagation von Exceptions
 - Context Managers “with” / ARM
 - **PART 7: Dateiverarbeitung**
 - Verzeichnisse und Dateien verwalten
 - Daten schreiben / lesen
 - JSON verarbeiten
-



- **PART 8: Datumsverarbeitung**
 - Einführung Datumsverarbeitung
 - Zeitpunkte und die Klasse `datetime`
 - Datumswerte und die Klasse `date`
 - Zeit und die Klasse `time`
- **PART 9: Einstieg in Lambdas & IterTools (Java Lambdas und Streams)**
 - Syntax von Lambdas
 - Lambdas im Einsatz mit `filter()`, `map()` und `reduce()`
 - Lambdas im Einsatz mit `groupby()` // `Collectors.groupingBy()`
 - `takewhile()` / `dropwhile()`
 - `teeing()`

Workshop Contents



- **PART 99: BONUS**
 - «Reflection»
 - «HTTP-Support»
 - «PyTest»
 - «Turtle Graphics»
 - «Databases»
 - »Python 3.10»
-



PART 1: Einstieg Python

- Einführung & Trend
 - Syntax im Kurzüberblick
 - Funktionen definieren
 - Module ausführbar machen / main()
-



- **Als Java-Entwickler fühlt man sich oft pudelwohl in seinem Universum.**
 - **ABER: Python wird immer populärer ...**
 - **In diesem Workshop wollen wir einige Java-Beispiele und die Pendants in Python kennenlernen und miteinander vergleichen.**
 - **Zudem sehen wir, wie sich JUnit und Pytest beim Formulieren lesbarer Tests schlagen.**
-



**Nach 25 Jahren Java:
Wo geht die Reise hin
und wie sieht es aktuell
mit der Konkurrenz aus?
Wieso Python?**



- Anfang der 1990er Jahre von Guido von Rossum als Skriptsprache entwickelt
 - 1994 in Version 1.0 veröffentlicht.
 - Mittlerweile hat Python zwar auch schon mehr als 25 Jahre auf dem Buckel, wird aber nicht altersschwach, sondern kontinuierlich gepflegt und weiterentwickelt.
 - Python 3 erschien bereits 2008 und hat diverse Aktualisierungen erfahren, derzeit Version 3.9.6 aktuell
 - Im Oktober erscheint dann Python 3.10
-

Zunächst ein Blick zurück (Aug 21)

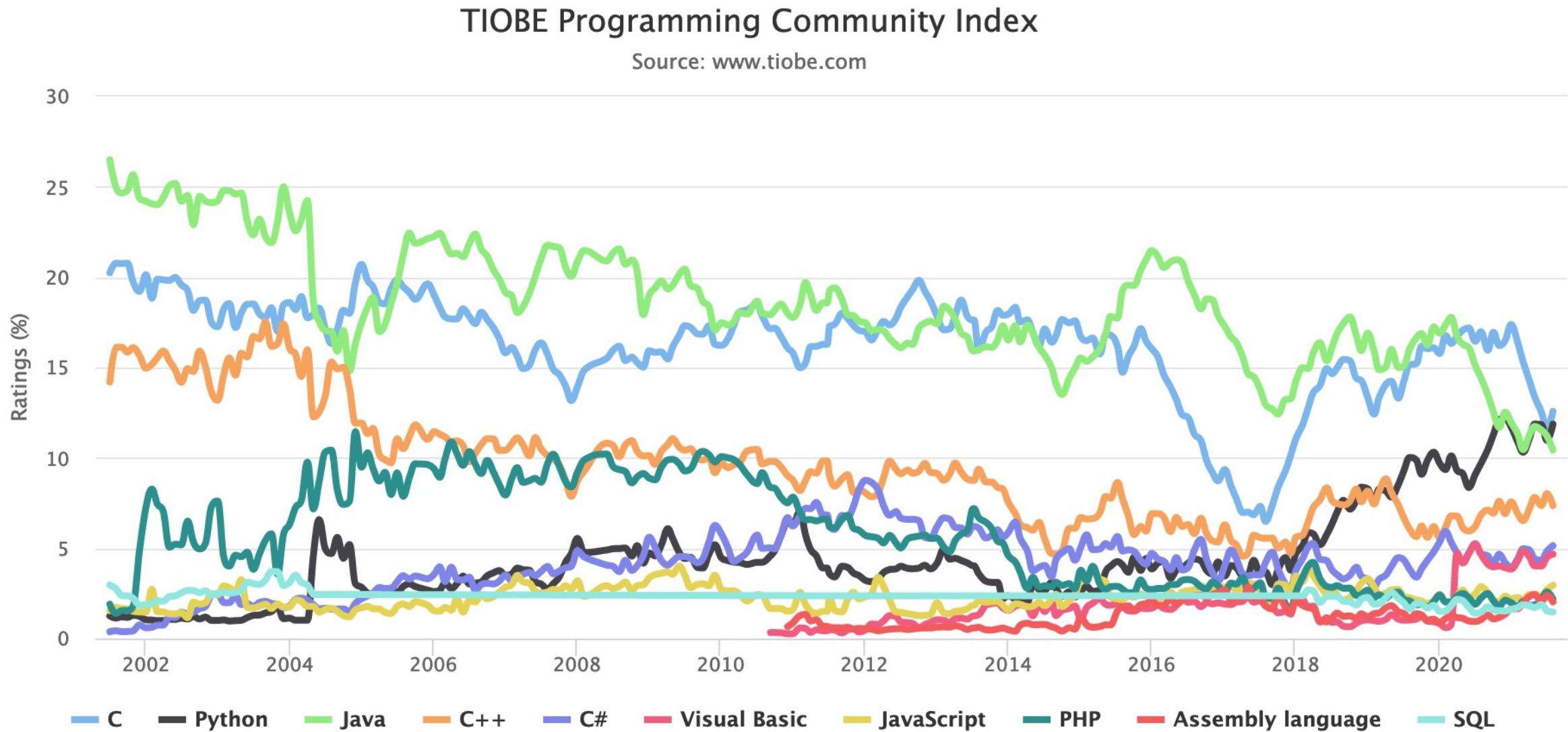


Very Long Term History

To see the bigger picture, please find below the positions of the top 10 programming languages of many years back. Please note that these are *average* positions for a period of 12 months.

Programming Language	2021	2016	2011	2006	2001	1996	1991	1986
C	1	2	2	2	1	1	1	1
Java	2	1	1	1	3	18	-	-
Python	3	5	6	8	26	24	-	-
C++	4	3	3	3	2	2	2	6
C#	5	4	5	7	13	-	-	-

Popularitätstrends der Top 10 Programmiersprachen (Aug 21)



Vergleich



- Wer nutzt Python?



YAHOO!

IBM

facebook

You Tube



Google

- Wer nutzt Java?

LinkedIn

NETFLIX

Google

CapitalOne

amazon



slack

intel



ebay

android

Spotify

Square

Kommandozeileninterpreter



- **Java - JShell**

```
$ jshell
| Welcome to JShell -- Version 16.0.1
| For an introduction type: /help intro
```

```
jshell>
```

- **Python**

```
$ python3
Python 3.9.6 (default, Jun 29 2021, 06:20:32)
[Clang 12.0.0 (clang-1200.0.32.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```



Syntax im Kurzüberblick



Syntax Quick Refresher



	Java	Python
Variablen	jshell> int age = 50 age ==> 50 jshell> long durationInMs = 2745 durationInMs ==> 2745 jshell> String message = "Hello JAX" message ==> "Hello JAX"	>>> age = 50 >>> durationInMs = 2745 >>> message = "Hello JAX"
String-Add	jshell> "The answer is " + 42 + " but " + false \$9 ==> "The answer is 42 but false"	>>> "The answer is " + str (42) + " but " + str (False) 'The answer is 42 but False'
Ausgaben	jshell> System.out.println (age + " " + durationInMs + " " + message) 50 2745 Hello JAX	>>> print (age, durationInMs, message) 50 2745 Hello JAX

Syntax Quick Refresher



	Java	Python
for-Schleife	jshell> for (int i = 0; i < 10; i++) ...> System.out.println("i: " + i);	>>> for i in range(0, 10): ... print("i: ", i)
Blöcke	if (a > b) { System.out.println("a > b") }	if a > b: print("a > b")
Logik Ops	&& !	and or not
Conditional Op	condition ? alt1 : alt2	alt1 if condition else alt2
Miniprogramm	public class SimpleExample { public static void main(String[] args) { System.out.println("7*2=" + (7*2)); } }	def main(): print("7*2", 7*2) if __name__ == "__main__": main()

Syntax Quick Refresher



	Java	Python
Listen	<pre>List<Integer> numbers = new ArrayList<>(); for (int i = 0; i < 10; i++) { numbers.add(i); } System.out.println(numbers.get(7)) 7</pre>	<pre>>>> numbers = [] >>> for i in range(10): >>> numbers.append(i) >>> numbers[7] 7</pre>
Listen II	<pre>List<Integer> numbers = List.of(0,1,2,3,4)</pre>	<pre>numbers = [0,1,2,3,4]</pre>
Listen III	<pre>-/-</pre>	<pre>even = [x for x in range(10) if x % 2 == 0] [0, 2, 4, 6, 8]</pre>

Syntax Quick Refresher



	Java	Python
Miniklasse	<pre>class TwoMembers { public int val1; public String val2; TwoMembers(int val1, String val2) { this.val1 = val1; this.val2 = val2; } }</pre>	<pre>class TwoMembers: def __init__(self, val1, val2): self.val1 = val1 self.val2 = val2</pre>
Tupel	<pre>jshell> record Tuple(int val1, int val2, ...> int val3, int val4) ...> {}</pre> <pre>jshell> new Tuple(7, 2, 7, 1).val3() \$5 ==> 7</pre>	<pre>>>> tuple = (7, 2, 7, 1) >>> tuple[2] 7</pre>



- **str – Textuelle Informationen, wie z. B. "Hallo". String-Werte sind von doppelten oder einfachen Anführungszeichen eingeschlossen**
- **int – Ganzzahlen wie 123, oder -4711**
- **float – Gleitkommazahlen, als mit Vor- und Nachkommastellen, wie 72.71 oder -1.357**
- **bool – Wahrheitswerte als wahr oder falsch, in Python: True oder False.**

```
>>> type("Python")
<class 'str'>
>>> type("1234")
<class 'str'>
>>> type(1234)
<class 'int'>
>>> type(42.195)
<class 'float'>
>>> type(True)
<class 'bool'>
```

Basisdatentypen



- **Operatoren**

```
>>> 5 + 4 - 2  
7
```

```
>>> 7 * 2  
14
```

```
>>> 14 / 3  
4.666666666666667
```

```
>>> 14 // 3  
4
```

```
>>> 14 % 3  
2
```

Funktionen definieren



- **Minimum aus 3 Werten**

```
def min_of_3(x, y, z):  
    if x < y:  
        if x < z:  
            return x  
        else:  
            return z  
    else:  
        if y < z:  
            return y  
        else:  
            return z
```

- **Basierend auf Built-in-Funktion min()**

```
def min_of_3(x, y, z):  
    return min(x, min(y, z))
```

Verschachtelte Funktionen definieren



```
def f():
    print("f1")

def g():
    print("g!")

# (noch) nicht sichtbar!
# h()
def h():
    print("h!")

h()

# nicht (mehr) sichtbar!
# h()
g()
print("f2")



---


f()
print("nach f")
```

Verschachtelte Funktionen definieren



```
def f():
    print("f1")

def g():
    print("g!")

# (noch) nicht sichtbar!
# h()
def h():
    print("h!")

h()

# nicht (mehr) sichtbar!
```

```
# h()
g()
print("f2")
```

```
f()
print("nach f")
```

f1
g!
h!
f2
nach f

Eigene Funktion mit Modulo und Division



```
>>> def extract_digits(number):
...     remaining_value = number
...     while remaining_value > 0:
...         digit = remaining_value % 10
...         remaining_value = remaining_value // 10
...         print(digit, end=' ')
...     print()
...
>>> extract_digits(1234)
4 3 2 1

>>> def extract_digits(number):
...     remaining_value = number
...     while remaining_value > 0:
...         remaining_value, digit = divmod(remaining_value, 10)
...         print(digit, end=' ')
...     print()
...
>>> extract_digits(1234)
4 3 2 1
```

Funktionen – Parameterübergabe per Position / Name



- **Gegeben sei folgende Funktion**

```
def parameter_example(first, second):  
    print("first:", first)  
    print("second:", second)
```

- **Bekanntermaßen und wie gewohnt lässt sich die Funktion wie folgt aufrufen:**

```
parameter_example(1, 22)
```

- **Als Variation ermöglicht Python eine Parameterübergabe per Name:**

```
parameter_example(second = 22, first = 1)
```

- **Beides Mal mit dem gleichen Resultat:**

```
first: 1  
second: 22
```

Funktionen definieren – Defaultwerte



- Mitunter ist es hilfreich, einen Defaultwert für einen Parameter bereitzustellen zu können

```
def parameter_with_default(x, y, opt_info = "n/a"):  
    print("(%d, %d)" % (x, y))  
    print("Info:", opt_info)
```

```
parameter_with_default(7, 2)  
parameter_with_default(72, 71, "OPT-VALUE")
```

(7, 2)
Info: n/a
(72, 71)
Info: OPT-VALUE

Funktionen definieren – Var Args



- **Mitunter ist es hilfreich, eine beliebige Anzahl an Werten an eine Funktion zu übergeben**
- **Dazu wird vor dem Parameter ein Stern, etwa *args angegeben**
- **Tatsächlich kennen wir bereits eine Funktion, die eine solche variable Anzahl an Argumenten unterstützt! Mehrmals haben wir nun schon die Funktion print() mit einer kommaseparierten Aufzählung von Werten aufgerufen.**

```
def flexi_print(*values):  
    print(*values)  
    print(values)
```

```
flexi_print("ONE")  
flexi_print("ONE", "TWO")  
flexi_print("ONE", "TWO", "THREE")
```

Funktionen definieren – Var Args



- ***values => weiterreichen als einzelne Parameter**
- **values => weiterreichen als Tupel**

```
def flexi_print(*values):  
    print(*values)  
    print(values)
```

```
flexi_print("ONE")  
flexi_print("ONE", "TWO")  
flexi_print("ONE", "TWO", "THREE")
```

```
ONE  
( 'ONE', )  
ONE TWO  
( 'ONE', 'TWO' )  
ONE TWO THREE  
( 'ONE', 'TWO', 'THREE' )
```

Funktionen definieren – Var Args



- Sinnvoller ist es wohl, auf den übergebenen Parameterwerten eine Aktion auszuführen: Hier: Summenberechnung inklusive der Ausgabe einer Meldung.
- Die einzelnen Werte lassen sich praktischerweise sehr leicht mit einer for-in-Schleife durchlaufen:

```
def var_args_sum(info, *args):  
    result = 0  
    for num in args:  
        result += num  
    return info + str(result)
```

```
print(var_args_sum("Summe: ", 1,2,3,4,5,6,7))
```

- Am Beispiel sieht man, dass neben den Var Args weitere Parameter möglich sind. Allerdings muss der Var Arg-Parameter am Ende stehen.
-

Built-In-Datentypen



- **Listen**

```
>>> names = ["Tim", "Michael", "Tom", "Mike", "Michael"]
>>> names.append("John")
>>> names += ["Jim", "James"]
>>> names
['Tim', 'Michael', 'Tom', 'Mike', 'Michael', 'John', 'Jim', 'James']
```

- **Tupel**

```
>>> pizza_top_3 = ("diavolo", "napoli", "funghi")
```

- **Sets**

```
>>> fruits = { "Apple", "Ananas" }
>>> fruits.add("Apple")
>>> fruits.add("Bananas")
>>> fruits
{'Apple', 'Bananas', 'Ananas'}
```

Built-In-Datentypen



- **Dictionaries (Maps)**

```
>>> city_inhabitants = { "Kiel" : 250_000,
...                      "Bremen" : 550_000,
...                      "Zürich" : 400_000 }
>>> city_inhabitants
{'Kiel': 250000, 'Bremen': 550000, 'Zürich': 400000}

>>> city_inhabitants.get("Zürich")
400000
>>> city_inhabitants["Zürich"]
400000

>>> city_inhabitants.keys()
dict_keys(['Kiel', 'Bremen', 'Zürich'])
>>> city_inhabitants.values()
dict_values([250000, 550000, 400000])
>>> city_inhabitants.items()
dict_items([('Kiel', 250000), ('Bremen', 550000), ('Zürich', 400000)])
```

Built-In-Datentypen



- **Wertebereiche**

```
>>> range(1, 10)  
range(1, 10)
```

```
>>> list(range(1,10))  
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Wertebereiche mit Schrittweite / auch negativ**

```
>>> list(range(2, 21, 3))  
[2, 5, 8, 11, 14, 17, 20]
```

```
>>> list(range(21, 2, -3))  
[21, 18, 15, 12, 9, 6, 3]
```

Schleifen



- **Indexbasierte for-in-Schleife**

```
>>> for i in range(2, 5):  
...     print("Durchlauf:", i)  
...  
Durchlauf: 2  
Durchlauf: 3  
Durchlauf: 4
```

- **Indexbasierte for-in-Schleife**

```
>>> cities = ['Kiel', "Bremen", 'Zürich', 'Basel']  
>>> for i in range(len(cities)):  
...     print(cities[i])  
...  
Kiel  
Bremen  
Zürich  
Basel
```

Schleifen



- **For-in-VALUES**

```
>>> for current_name in ["Barbara", "Lilija", "Sophie"]:  
...     print(current_name)  
...  
Barbara  
Lilija  
Sophie
```

- **For-In-REVERSED**

```
>>> for current_name in reversed(["Barbara", "Lilija", "Sophie"]):  
...     print(current_name)  
...  
Sophie  
Lilija  
Barbara
```



- **For-in-SORTED**

```
>>> for value in sorted([7, 2, 3, 5, 11, 17, 19, 13]):  
...     print(value)  
...  
2  
3  
5  
7  
11  
13  
17  
19
```



- **Zugriff auf Index und Wert ohne enumerate()**

```
>>> message = ["Python", "has", "several", "loop", "variants"]
>>> for i in range(len(message)):
...     print(i, message[i], end=',')
...
0 Python,1 has,2 several,3 loop,4 variants,
```

- **Mit enumerate()**

```
>>> message = ["Python", "has", "several", "loop", "variants"]
>>> for i, current_word in enumerate(message):
...     print(i, current_word, end=',')
...
0 Python,1 has,2 several,3 loop,4 variants,
```

Schleifen



- **Kombination von enumerate() und sorted()**

```
>>> for idx, value in enumerate(sorted([7, 2, 3, 5, 11, 17, 19, 13])):  
...     print(idx, value)  
...  
0 2  
1 3  
2 5  
3 7  
4 11  
5 13  
6 17  
7 19
```

Schleifen



- **Kombination von enumerate() und reversed() und sorted()**

```
>>> for idx, value in enumerate(reversed(sorted([7, 2, 3, 5, 11, 17, 19, 13]))):  
...     print(idx, value)  
...  
0 19  
1 17  
2 13  
3 11  
4 7  
5 5  
6 3  
7 2
```

Schleifen



- **while <cond>**

```
>>> i = 0
>>> while i < 5:
...     print(i)
...     i += 1
...
0
1
2
3
4
```

ABER: KEIN DO-WHILE in Python!

Externe Funktionalität einbinden: Module



- **Zufallsfunktionalität aus dem Modul random**

```
>>> import random
```

```
>>> random.randrange(2, 27)
```

```
24
```

```
>>> random.randrange(2, 27)
```

```
12
```

```
>>> random.choice(["Pizza", "Pasta", "Steak"])
```

```
'Steak'
```

```
>>> random.choice(["Pizza", "Pasta", "Steak"])
```

```
'Pizza'
```

Eigene Module: <xyz>.py



- **Module sind in Python nur Dateien mit der Endung .py und etwas Python-Code**
- **Problem: Beim import wird der Code immer ausgeführt**
- **Abhilfe:**

```
def main():
    # Hauptprogramm

if __name__ == "__main__":
    main()
```



DEMO

[`own_module.py`](#)



Exercises Part 1

<https://github.com/Michaeli71/Python-Workshop.git>





PART 2: Strings

- Wichtige Funktionen / Methoden
 - Slicing
 - Formatierte Ausgaben
 - Mehrzeilige Strings
 - Pytest Quick Start
-

Strings



- **Definition**

```
str1 = "DOUBLE QUOTED STRING"  
str2 = 'SINGLE QUOTED STRING'
```

- **Vorteil: Mix möglich!**

```
>>> str1 = "DOUBLE 'contains' single"  
>>> str2 = 'SINGLE "contains" double'  
>>> str1  
"DOUBLE 'contains' single"  
>>> str2  
'SINGLE "contains" double'
```

Strings



- **Unicode (\u)**

```
>>> str_unicode = "\u0333\u0424"
>>> str_unicode
'_=Φ'
```

- **Unicode mit Surrogates (\U)**

```
>>> str_unicode = "\U0001F601"  
>>> str_unicode  
'😊'
```

- Unicode mit Namen (`\N{name}`)

```
>>> str_unicode = "\N{dog}\N{cat}\N{bird}\N{mouse}"
>>> str_unicode
'🐶🐱🐦🐭'
```

Strings



- **Groß- und Kleinschreibung**

```
>>> message = "IMPORTANT: Please consult the doctor"  
>>> message.upper()  
'IMPORTANT: PLEASE CONSULT THE DOCTOR'  
>>> message.lower()  
'important: please consult the doctor'
```

- **Unveränderlichkeit!!!**

```
>>> message  
'IMPORTANT: Please consult the doctor'
```

Strings



- **Konkatenation (+)**

```
>>> last_name = "Inden"  
>>> "Michael" + " " + last_name  
'Michael Inden'
```

- **ABER:**

```
>>> "Bitte " + 2 + " mal klingeln"  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: can only concatenate str (not "int") to str
```

- **Lösung:**

```
>>> "Bitte " + str(2) + " mal klingeln"  
'Bitte 2 mal klingeln'
```

Strings



- **center()**

```
>>> value = "This text has blanks at the beginning and the end"
>>> value_with_blanks = value.center(55)
>>> value_with_blanks
'    This text has blanks at the beginning and the end      '
```

- **strip()**

```
>>> print("strip(): '", value_with_blanks.strip(), "'", sep="")
strip(): 'This text has blanks at the beginning and the end'
```

- **lstrip() / rstrip():**

```
>>> print("lstrip(): '", value_with_blanks.lstrip(), "'", sep="")
lstrip(): 'This text has blanks at the beginning and the end      '
```

```
>>> print("rstrip(): '", value_with_blanks.rstrip(), "'", sep="")
rstrip(): '    This text has blanks at the beginning and the end'
```

Strings



- **Länge ermitteln – len()**

```
>>> content = "This is a short message"  
>>> len(content)  
23
```

- **Leerstring?**

```
>>> no_content = ""  
>>> len(no_content) == 0  
True
```

- **Prüfungsvariante**

```
>>> if not no_content:  
...     print("empty")  
...  
empty
```

Strings



- **Zeichen oder Ziffern?**

```
>>> pincode = "1234"
>>> pincode.isalpha()
False
>>> pincode.isdigit()
True
>>> pincode = "A4711"
>>> pincode.isalpha()
False
>>> pincode.isdigit()
False
>>> pincode = "ABC"
>>> pincode.isalpha()
True
```

Strings



- **Auf Zeichen zugreifen**

```
>>> content = "This is a short message"  
>>> content[0]  
'T'  
>>> content[3]  
's'
```

- **Besonderheit: Negative Indizes möglich (-idx ⇔ len(txt) - idx)**

```
>>> content = "This is a short message"  
>>> content[-1]  
'e'  
>>> content[-2]  
'g'  
>>> content[-3]  
'a'  
>>> content[-20]  
's'
```



```
>>> content[len(content) - 1]  
'e'  
>>> content[len(content) - 2]  
'g'  
>>> content[len(content) - 3]  
'a'  
>>> content[len(content) - 20]  
's'
```

Strings



- **Bereichsverletzung**

```
>>> content = "This is a short message"  
>>> content[25]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range
```

- **Bereichsverletzung bei negativem Index**

```
>>> content = "This is a short message"  
>>> content[-25]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
IndexError: string index out of range
```

Strings



- **String positionsbasiert durchlaufen**

```
message = "Python has several loop variants"  
for i in range(len(message)):  
    print(i, message[i], end=',')
```

- **Zeichenbasiert**

```
for current_char in message:  
    print(current_char, end=',')
```

- **Kombination: Mit Position und aktuellem Zeichen**

```
for i, current_char in enumerate(message):  
    print(i, current_char, end=',')
```

Strings



- **String positionsbasiert durchlaufen**

```
message = "Python has several loop variants"
for i in range(len(message)):
    print(i, message[i], end=',')
```

- **Zeichenbasiert**

```
for current_char in message:
    print(current_char, end=',')
```

P,y,t,h,o,n, ,h,a,s, ,s,e,v,e,r,a,l,
,l,o,o,p, ,v,a,r,i,a,n,t,s,

- **Kombination: Mit Position und aktuellem Zeichen**

```
for i, current_char in enumerate(message):
    print(i, current_char, end=',')
```

0 P,1 y,2 t,3 h,4 o,5 n,6 ,7 h,8 a,9 s,10 ,11 s,12 e,13 v,14 e,15 r,16 a,17
l,18 ,19 l,20 o,21 o,22 p,23 ,24 v,25 a,26 r,27 i,28 a,29 n,30 t,31 s,

Strings



- **Slicing => Extraktion eines neuen Strings von Position startidx bis endidx**

text[startidx:endidx]

- Beispiel "Dies ist ein String. Rest ABC"

```
print(teile[0:4])  
print(teile[5:8])  
print(teile[9:12])  
print(teile[9:19])  
print(teile[19:])
```



Dies
ist
ein
ein String
. Rest ABC

- **Index negativ**

AB

```
>>> print(teile[-3:-1])
```



ABC

Strings



- **Slicing – Besonderheiten**

text[startidx:endidx:step]

- **Beispiel**

```
teile = "Dies ist ein String. Rest ABC"
```

```
# Besonderheiten  
print(teile[:])
```

```
print(teile[::-1])  
print(teile[19::-1])  
print(teile[:8:-1])  
print(teile[-5:8:-1])
```



Dies ist ein String. Rest ABC

CBA tseR .gnirtS nie tsi seiD
.gnirtS nie tsi seiD
CBA tseR .gnirtS nie
tseR .gnirtS nie

Strings



- **Enthaltensein (in)**

```
>>> msg = "Tim arbeitet in Kiel. Michael lebt in Zürich"
```

```
>>> "Michael" in msg  
True
```

```
>>> "arbeitet" in msg  
True
```

```
>>> "Bremen" in msg  
False
```

Strings



- **Start und Ende prüfen**

```
>>> msg = "Important Info"  
>>> msg.startswith("Impo")  
True  
>>> msg.endswith("Info")  
True
```

- **Vorkommen zählen**

```
>>> msg = "tri tra tru lala"  
>>> msg.count("tr")  
3  
>>> msg.count("tra")  
1  
>>> msg.count("la")  
2  
>>> msg.count("a")  
3
```

Strings



- **Suchen**

```
>>> msg = "This is a tiny story. This tiny text ends now."  
>>> msg.find("tiny")  
10  
>>> msg.rfind("tiny")  
27  
>>> msg.index("tiny")  
10  
>>> msg.rindex("tiny")  
27  
>>> msg.find("tiny", 11) ←  
27  
>>> msg.find("Michael")  
-1  
>>> msg.index("Michael")  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: substring not found
```

Strings Besonderheiten



- **Strings wiederholen (*)**

```
>>> greeting = "MOIN"  
>>> greeting * 2  
'MOINMOIN'
```

```
>>> nonsens = "BLA"  
>>> nonsens * 3  
'BLABLABLA'
```

- **Spezialschreibweise**

```
>>> msg = "my name is michael"  
>>> msg.capitalize()  
'My name is michael'  
>>> msg.title()  
'My Name Is Michael'
```

Strings Besonderheiten



- **Strings aufspalten**

```
>>> timestamp = "11:22:33"  
>>> timestamp.split(":")  
['11', '22', '33']
```

- **Strings aufspalten / zusammenführen und ersetzen**

```
>>> msg = "One Two Three Four"  
>>> msg.split(" ")  
['One', 'Two', 'Three', 'Four']  
  
>>> "-".join(['One', 'Two', 'Three', 'Four'])  
'One-Two-Three-Four'  
  
>>> 'One-Two-Three-Four'.replace("-", "...")  
'One...Two...Three...Four'
```

Strings – Formatierte Ausgabe



- **Verschiedene Varianten**

```
product = "Apple iMac"  
price = 3699
```

```
# Varianten der formatierten Ausgabe  
print("the", product, "costs", price)  
print("the {} costs {}".format(product, price))  
print(f"the {product} costs {price}")  
print(f"the {product} costs {price}")
```

- **Führen zu folgenden Ausgaben**

```
the Apple iMac costs 3699  
the Apple iMac costs 3699  
the Apple iMac costs 3699  
the Apple iMac costs 3699
```

Strings



- **Spezielle Zeichen im String (hier ungewünscht Newline / Tabs)**

```
>>> path = "C:\newcontent\tim\news"  
>>> path  
'C:\newcontent\tim\news'  
>>> print(path)  
C:  
ewcontent im  
ews
```

- **Raw-Strings (r"""") als Abhilfe**

```
>>> path = r"C:\newcontent\tim\news"  
>>> path  
'C:\\newcontent\\\\tim\\\\news'  
>>> print(path)  
C:\newcontent\tim\news
```

Strings



- **Mehrzeilige Strings ("")**

```
>>> multi_line_string = """Line 1
...     Line 2
...     Line 3"""
>>> multi_line_string
'Line 1\n     Line 2\n     Line 3'
```

- **Mehrzeilige Strings können einfache Anführungszeichen enthalten**

```
>>> multi_line_string_with_quotes = """Line 1
... the "second" line contains 'quotes'
... last line"""
>>>
>>> multi_line_string_with_quotes
'Line 1\nthe "second" line contains \'quotes\'\nlast line'
```



- **Mehrzeilige Strings mit Platzhaltern**

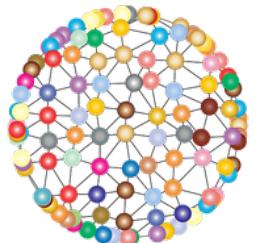
```
>>> value_dict = { "info" : "text", "costs" : "for free",
...                 "action" : "consult", "document" : "help" }
>>>
>>> multi_line_string_with_dict = """This {info} is {costs}.
... Please {action} the {document}""".format_map(value_dict)
>>>
>>> multi_line_string_with_dict
'This text is for free.\nPlease consult the help'
```

- **Sind normale Strings**

```
>>> type(multi_line_string_with_dict)
<class 'str'>
>>> len(multi_line_string_with_dict)
```



**Wie kann ich
Werte einlesen?**



Strings – Eingaben



- **Von Konsole einlesen => Achtung: die Rückgabe ist vom Typ str**

```
>>> input("What's your name?")
What's your name? Michael
'Michael'
```

```
>>> input("How old are you? ")
How old are you? 50
'50'
```

- **Als Ganzzahl einlesen**

```
>>> age = int(input("How old are you? "))
How old are you? 50
>>> age
50
```



«PyTest Quick Start»





- Für Tests sofort (nach Installation) einsetzbar (<https://docs.pytest.org/en/6.2.x/>)
- Testbehauptungen werden einfach mit dem Schlüsselwort assert formuliert

```
def test_answer():
    assert sum([1, 2, 3]) == 6
```

- Auf einfache Weise auch parametrierte Tests möglich

```
import pytest

@pytest.mark.parametrize("test_input,expected",
                      [("2 + 5", 7), ("2 ** 4", 16), ("6 * 9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Pytest Quick Start



- Gute Integration in PyCharm
- Tests beginnen mit «test_»

```
import pytest

def not_detectedd_test():
    assert sum([1, 2, 3]) == 6

def test_answer():
    assert sum([1, 2, 3]) == 6
```

```
@pytest.mark.parametrize("test_input,expected",
                         [("2 + 5", 7), ("2 ** 4", 16), ("6 * 9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Test Results		0 ms
✗	pytest_intro	0 ms
✓	test_answer	0 ms
✗	test_eval	0 ms
✓	(2 + 5-7)	0 ms
✓	(2 ** 4-16)	0 ms
✗	(6 * 9-42)	0 ms



DEMO

[`pytest_intro.py`](#)



Exercises Part 2

<https://github.com/Michaeli71/Python-Workshop.git>





PART 3: Klassen

- Basics
 - Textuelle Ausgaben
 - Gleichheit `is` / `==` und `==` / `equals()`
 - Unterschiede zu Java / Information Hiding
 - Vererbung und Typprüfung
 - Enums
 - Named Tuple / Java Record
-

Klassen definieren



- Spezielle Methode `__init__()` ist der Konstruktor
- Attribute werden einfach im Konstruktor aufgeführt
- `self` ist wie `this`
- Keine wirklichen Sichtbarkeiten
- Attribute sind standardmäßig «public»

```
class Car:  
    def __init__(self):  
        self.brand = None  
        self.color = None  
        self.horse_power = 0
```

- Objektkonstruktion einfach durch Aufruf (ohne `new`)

```
car = Car()
```

Klassen definieren



- **KEIN OVERLOADING**

```
class Car:  
    def __init__(self):  
        self.brand = None  
        self.color = None  
        self.horse_power = 0  
  
    def __init__(self, brand, color, horse_power):  
        self.brand = brand  
        self.color = color  
        self.horse_power = horse_power
```



- **Versuchen wir es mal:**

```
car = Car()
```

```
TypeError: __init__() missing 3 required positional arguments: 'brand',  
'color', and 'horse_power'
```

Klassen definieren



- **String-Ausgabe:**

```
class Car:  
    def __init__(self, brand, color, horse_power):  
        self.brand = brand  
        self.color = color  
        self.horse_power = horse_power
```

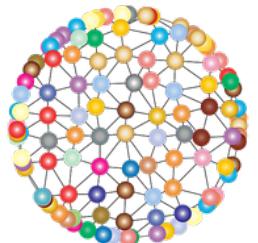
```
my_car = Car("VW", "YELLOW", 75)  
print(my_car)
```

- **Versuchen wir es mal:**

```
<__main__.Car object at 0x1036431c0>
```



Wie ist es in Java?



Klassen definieren



- **String-Ausgabe mit `__str__()` analog zu `toString()`**

```
class Car:  
    def __init__(self, brand, color, horse_power):  
        self.brand = brand  
        self.color = color  
        self.horse_power = horse_power  
  
    def __str__(self):  
        return f"Marke: {self.brand} / Farbe: {self.color} /" + \  
              f" PS: {self.horse_power}"
```

```
my_car = Car("VW", "YELLOW", 75)  
print(my_car)
```

- **Versuchen wir es mal:**

Marke: VW / Farbe: YELLOW / PS: 75

Klassen definieren



- **String-Ausgabe mit `__str__()` und `__repr__()` für die Kommandozeile**

```
class Car:  
    def __init__(self, brand, color, horse_power):  
        self.brand = brand  
        self.color = color  
        self.horse_power = horse_power  
  
    def __str__(self):  
        return f"Marke: {self.brand} / Farbe: {self.color} /" + \  
              f" PS: {self.horse_power}"  
  
    def __repr__(self):  
        return self.__str__()
```

- `__str__()` soll menschenlesbar sein
- `__repr__()` soll gut maschinenlesbar und als Objekt rekonstruierbar sein (z. B. Listen)

Attributzugriffe



- **Punkt-Notation zum Lesen und Schreiben**

```
print(my_car.brand)
```

```
my_car.color = "STEALTH_BLACK"  
my_car.horse_power += 250  
print(my_car)
```

- **Wir erhalten folgendes:**

VW

Marke: VW / Farbe: STEALTH_BLACK / PS: 325

- **ACHTUNG: Es gibt in Python keine wirkliche Sichtbarkeitseinschränkung!!**
-

Verhalten definieren



```
class Car:  
    def __init__(self, brand, color, horse_power):  
        self.brand = brand  
        self.color = color  
        self.horse_power = horse_power  
  
    def __str__():  
        return f"Marke: {self.brand} / Farbe: {self.color} /" + \  
              f" PS: {self.horse_power}"  
  
    def __repr__():  
        return self.__str__()  
  
    def paint_with(self, new_color):  
        self.color = new_color  
  
    def apply_tuning_kit(self):  
        self.horse_power += 150
```

Statische Attribute und Methoden



```
class StaticExample:  
    static_info = "Class wide info"  
  
    @staticmethod  
    def generate_info():  
        print("static methods can be called without creating objects")  
        return "Special Information"  
  
    def object_method(self):  
        print("object methods must be called on objects")  
        print("static methods/variables are accessible")  
        return StaticExample.static_info
```

Objekte vergleichen



```
toms_car = Car("Audi", "BLUE", 275)  
jims_car = Car("Audi", "BLUE", 275)  
  
print(toms_car == jims_car)
```



**Was ist das
Ergebnis?**

Objekte vergleichen



- Gegeben seien folgende Definitionen:

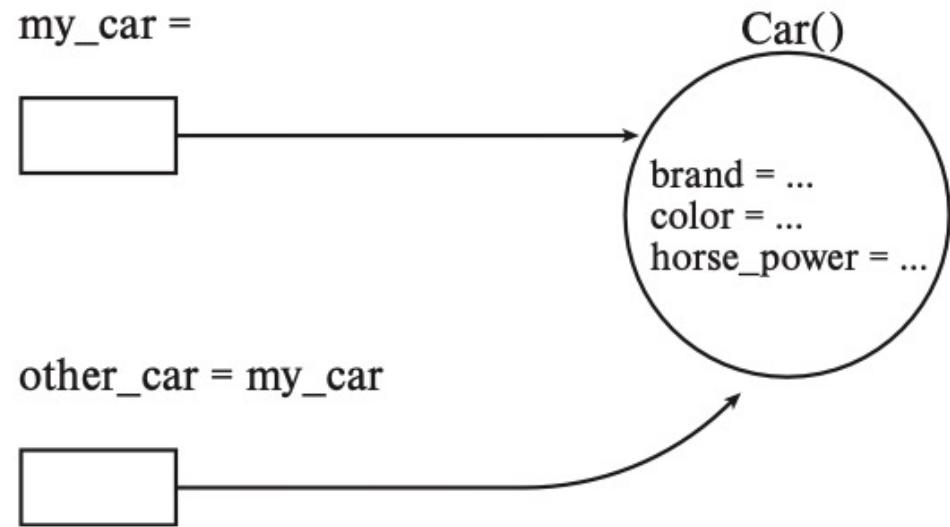
```
toms_car = Car("Audi", "BLUE", 275)  
jims_car = Car("Audi", "BLUE", 275)
```

```
print(toms_car == jims_car)
```

- Referenzgleichheit / Identität

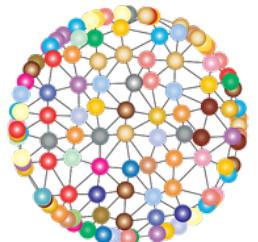
```
print(id(toms_car))  
print(id(jims_car))
```

- Inhaltliche Gleichheit / semantische Gleichheit => gleiche Werte der Attribute





Was braucht es für semantische Gleichheit in Java?



Objekte vergleichen



- Operator 'is' – Mit dem Operator 'is' werden Referenzen verglichen. Somit wird auf *Identität* geprüft, also, ob es sich um dieselben Objekte handelt.
- Operator '==' – Mit dem Operator '==' werden zwei Objekte bezüglich ihres *Zustands* (d. h. der Werteverteilung der Attribute) verglichen. Zum inhaltlichen Vergleich eigener Klassen ist dort die Methode `__eq__()` selbst zu implementieren.
 - Standardmäßig erfolgt sonst lediglich ein Vergleich der beiden Referenzen mit dem Operator 'is'.
 - In eigenen Realisierungen muss derjenige Teil des Objektzustands verglichen werden, der für die semantische Gleichheit, also die inhaltliche Gleichheit, relevant ist.



DEMO

car6_comparison.java

Implementierung von `__eq__()`



1. **Typprüfung** – Um nicht Äpfel mit Birnen zu vergleichen, sichern wir vor dem inhaltlichen Vergleich ab, dass nur Objekte des gewünschten Typs verglichen werden.
2. **Objektvergleich** – Anschließend werden diejenigen Attributwerte verglichen, die für die Aussage »Gleichheit« relevant sind.
 - Die hierzu notwendigen Attribute des Objekts werden (z. B. in der Reihenfolge ihrer Definition oder des ersten vermuteten Unterschieds) per Operator `==` auf Gleichheit geprüft.

```
def __eq__(self, other):  
    if not isinstance(other, Car):  
        return False  
  
    return self.brand == other.brand and \  
           self.color == other.color and \  
           self.horse_power == other.horse_power
```



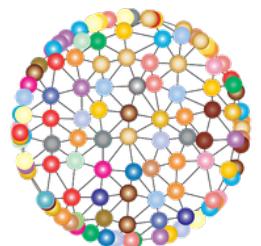
DEMO

[car7_comparison.py](#)

Java vs Python



Art		Java	Python
Referenzgleichheit / Identität		<code>==</code>	<code>is</code>
Semantische / inhaltliche Gleichheit		<code>equals()</code>	<code>==</code>



**Wie sieht es denn mit
Information Hiding aus?**



Während Sprachen wie Java oder C++ sogenannte Sichtbarkeiten besitzen, um den Zugriff auf private Klassenbestandteile steuern und schützen zu können, ist dies in Python so nicht möglich!

Information Hiding



Allerdings gibt es mit `_` und `__` zwei Varianten, um Ähnliches zu erzielen ...

- `_` – Wenn Attribute und Methoden in Python mit `_` beginnen, dann bedeutet der eine Unterstrich per Konvention, dass diese Methode oder dieses Attribut als privat und Implementierungsdetail der Klasse anzusehen ist.
 - Allerdings wird dies von Python nicht forciert und es werden auch keine Zugriffe unterbunden. Vielmehr ist man auf die Beachtung durch andere Programmierer angewiesen.
- `__` – Ein doppelter Unterstrich kennzeichnet eine spezielle interne Methode, wie wir dies etwa für `__str__()` oder `__eq__()` bereits gesehen haben.
 - Für Attribute eingesetzt, ist dieses Attribut nicht mehr unter seinem Namen nach außen für andere Klassen sichtbar.
 - In beiden Fällen wird der Name dann zu `_ClassName__method/attribute`. Für Methoden dient dies dazu, diese in Vererbungshierarchien speziell zugreifbar zu machen.

Information Hiding



```
class Car:  
    def __init__(self, brand, color, horse_power):  
        self.__brand = brand  
        self.__color = color  
        self.__horse_power = horse_power  
  
    def paint_with(self, new_color):  
        self.__color = new_color  
  
    def apply_tuning_kit(self):  
        self.__horse_power += 150  
  
    # Zugriff nach außen gewähren  
    def brand(self):  
        return self.__brand  
  
    def horse_power(self):  
        return self.__horse_power
```

Information Hiding – «private Methoden» => Properties



```
class Car:  
    def __init__(self, brand, color, horse_power):  
        self.__brand = brand  
        self.__color = color  
        self.__horse_power = horse_power  
  
    def paint_with(self, new_color):  
        self.__color = new_color  
  
    def apply_tuning_kit(self):  
        self.__horse_power += 150  
  
    def __horse_power(self):  
        return self.__horse_power  
  
    def __brand(self):  
        return self.__brand  
  
...
```

Information Hiding – «private Methoden» => Properties



...

```
def __get_brand(self):
    print("__get_brand")
    return self.__brand

def __get_color(self):
    print("__get_color")
    return self.__color

def paint_with(self, color):
    print("paint_with")
    self.__color = color

# property(fget, fset, fdel, doc)
brand = property(__get_brand)
color = property(__get_color, paint_with)
horse_power = property(__get_horse_power, __set_horse_power)
```



READ ONLY
READ WRITE
READ WRITE

Information Hiding – «private Methoden» => Properties



```
my_car = Car("RENAULT", "PETROL", 195)
#my_car.brand = "VW" # AttributeError: can't set attribute
my_car.color = "RED"
my_car.horse_power = 7271
```



Wertebereichsprüfungen



Die Kapselung ermöglicht es uns, innerhalb von Methoden verschiedene Prüfungen zu integrieren und ungültige Werte zurückzuweisen.

```
def set_horse_power(self, horse_power):
    if horse_power <= 0 or horse_power > 2_000:
        raise ValueError("INVALID PS: not in range 1 – 2000")
    self.__horse_power = horse_power
```

Wertebereichsprüfungen mit Properties



Man kann auch Properties mit @ verwenden:

```
@property  
def horse_power(self):  
    return self.__horse_power  
  
@horse_power.setter  
def horse_power(self, horse_power):  
    if horse_power <= 0 or horse_power > 2_000:  
        raise ValueError("INVALID PS: not in range 1 - 2000")  
  
    self.__horse_power = horse_power
```

Damit sieht es wie ein Attributzugriff aus, bietet aber mehr Möglichkeiten:

```
my_car = Car("Audi", "BLUE", 275)  
my_car.horse_power = 1234  
print(my_car.horse_power)
```



DEMO

**properties1/2.py
car8_information_hiding.py**

```
# AttributeError: 'Car' object has no attribute '__brand'  
#print(toms_car.__brand)
```

Vererbung

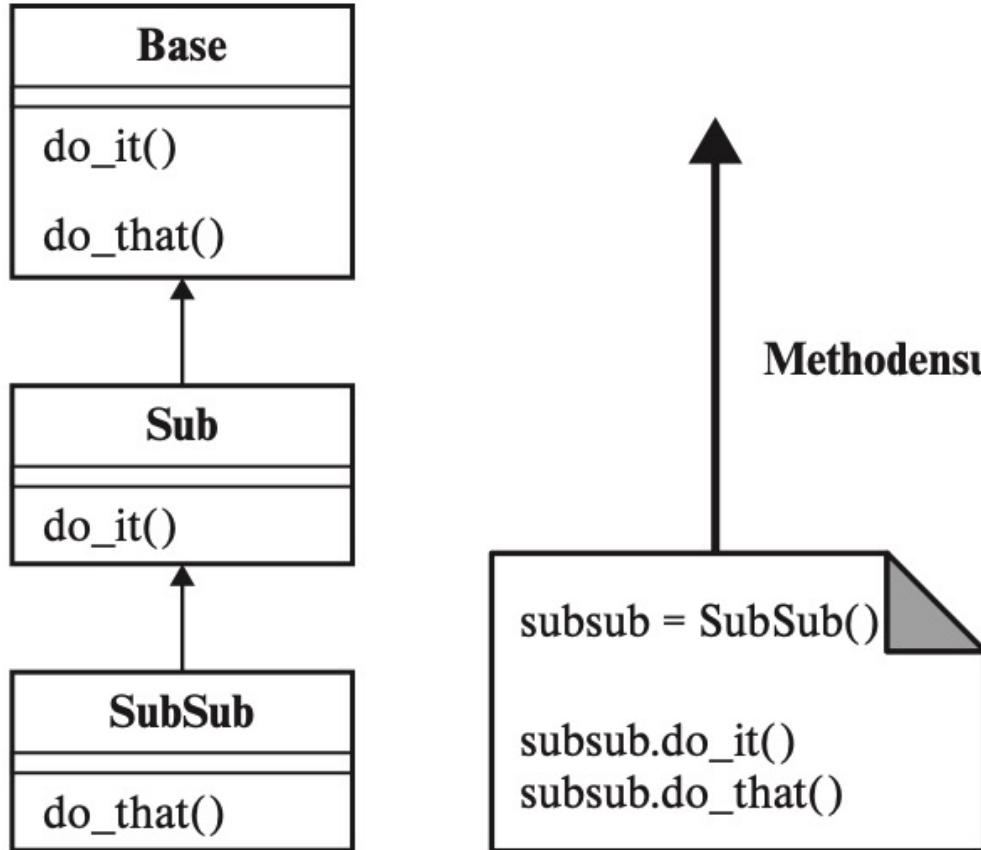


```
class BaseClass:  
    def method(self):  
        print("called method")  
  
    def other_method(self):  
        pass  
  
class SubClass(BaseClass):  
    def method(self):  
        super().method()  
        # weitere Aktionen  
        print("other actions")  
  
    # Zusätzliche Funktionalität  
    def additional_method(self):  
        pass
```

Vererbung



```
class Base:  
    def do_it(self):  
        pass  
  
    def do_that(self):  
        pass  
  
class Sub(Base):  
    def do_it(self):  
        print("do_it in Sub")  
  
class SubSub(Sub):  
    def do_that(self):  
        print("do_that in SubSub")
```



Vererbung prüfen – issubclass()



```
class Base:  
    def do_it(self):  
        pass  
  
    def do_that(self):  
        pass  
  
class Sub(Base):  
    def do_it(self):  
        print("do_it in Sub")  
  
class SubSub(Sub):  
    def do_that(self):  
        print("do_that in SubSub")  
  
print(issubclass(Sub, Base))  
print(issubclass(SubSub, Base))  
print(issubclass(SubSub, Sub))
```

Vererbung – Abstrakte Methoden nur simuliert ...



```
class Base:  
    def do_it(self):  
        pass  
  
    def do_that(self):  
        pass  
  
class Sub(Base):  
    def do_it(self):  
        print("do_it in Sub")  
  
    def do_that(self):  
        print("do_that in SubSub")  
  
obj = Base()  
obj.do_it()  
  
obj = Sub()  
obj.do_that()
```

Vererbung – Abstrakte Methoden



```
from abc import ABC, abstractmethod
```

```
class Base(ABC):
    @abstractmethod
    def do_it(self):
        pass

    @abstractmethod
    def do_that(self):
        pass
```

```
obj = Base()      TypeError: Can't instantiate abstract class Base with abstract methods do_it, do_that
```

```
obj = Sub()      TypeError: Can't instantiate abstract class Sub with abstract method do_that
```



DEMO

`inheritance4_ABC_abstract_method.py`

Aufzählungen mit Enum



```
from enum import Enum, auto
```

```
class Jahreszeiten(Enum):
    FRÜHLING = auto()
    SOMMER = auto()
    HERBST = auto()
    WINTER = auto()
```

```
class Size(Enum):
    XS = auto()
    S = auto()
    M = auto()
    L = auto()
    XL = auto()
    XXL = auto()
```

Aufzählungen mit Enum



```
from enum import Enum, auto

class Jahreszeiten(Enum):
    FRÜHLING = auto()
    SOMMER = auto()
    HERBST = auto()
    WINTER = auto()

class Size(Enum):
    XS = auto()
    S = auto()
    M = auto()
    L = auto()
    XL = auto()
    XXL = auto()
```

```
shirt_size = Size.XL
print(shirt_size)

for name in Jahreszeiten:
    print(name)
    print(name.value)
```

```
Size.XL
Jahreszeiten.FRÜHLING
1
Jahreszeiten.SOMMER
2
Jahreszeiten.HERBST
3
Jahreszeiten.WINTER
4
```

Enum



```
from enum import Enum
```

```
class Direction(Enum):
```

```
    N = (0, -1)
```

```
    NE = (1, -1)
```

```
    E = (1, 0)
```

```
    SE = (1, 1)
```

```
    S = (0, 1)
```

```
    SW = (-1, 1)
```

```
    W = (-1, 0)
```

```
    NW = (-1, -1)
```

```
print(Direction.NE.value)
```

(1, -1)

```
ne = Direction.NE
```

1 / -1

```
print(ne.value[0], "/", ne.value[1])
```

Named Tuple



- **Datenbehälterklassen**

```
class Point2D:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y
```

```
origin = Point2D(0, 0)  
print(origin)
```

<__main__.Point2D object at 0x102d09fd0>

- **Named Tuple**

```
Point2D = namedtuple('Point2D', ['x', 'y'])  
origin = Point2D(0, 0)  
print(origin)
```

Point2D(x=0, y=0)

Named Tuple – Definitionsvarianten



```
Point2D = namedtuple('Point2D', ['x', 'y'])
```

- **Angaben der Attribute als Tupel**

```
Point2D = namedtuple('Point2D', ('x', 'y'))
```

- **Als String mit kommasseparierter Angabe**

```
Point2D = namedtuple('Point2D', ('x, y'))
```

- **Als reiner String mit Space getrennt:**

```
Point2D = namedtuple('Point2D', ('x y'))
```

Named Tuple / Java Records



- **JAVA**

```
jshell> record Person(String name, int age) {}  
| created record Person
```

- **PYTHON**

```
# Named Tuple vs Class  
from collections import namedtuple  
  
Person = namedtuple('Person', ['name', 'age'])
```

- **Dabei handelt es sich um eine spezielle Unterklasse eines Tupels, die programmgesteuert basierend auf den Angaben zu den benannten Attributen erstellt wird.**
-

Named Tuple Vorteile



- **unveränderliche Datenstrukturen**
 - **Kann als Key in Dictionaries dienen**
 - **können in Mengen gespeichert werden**
 - **Bietet eine hilfreiche String-Darstellung, die den Tupel-Inhalt in einem Name=Wert-Format ausgibt**
 - **Unterstützt indizierten Zugriff**
 - **sind abwärtskompatibel mit regulären Tupeln**
 - **haben einen ähnlichen Speicherverbrauch wie reguläre Tupel**
-

Named Tuple Vorteile am Beispiel



- **Bietet eine hilfreiche String-Darstellung**
- **Unterstützt indizierten Zugriff**
- **Unveränderliche Datenstrukturen**

```
Top3 = namedtuple('Top3', ['first', 'second', 'third'])
pizza_top3 = Top3("Funghi", "Diavola", "Napoli")
```

```
print(pizza_top3)
print(pizza_top3[0], pizza_top3[1], pizza_top3[2])
pizza_top3[0] = "GYROS"
```

- =>

```
Top3(first='Funghi', second='Diavola', third='Napoli')
Funghi Diavola Napoli
Traceback (most recent call last):
  pizza_top3[0] = "GYROS"
TypeError: 'Top3' object does not support item assignment
```



Exercises Part 3

<https://github.com/Michaeli71/Python-Workshop.git>





PART 4:

Collections

- Schnelleinstieg list
 - Comprehensions
 - Slicing – Zugriff auf Teilbereiche
 - Schnelleinstieg set
 - Schnelleinstieg dict
-



Sequentielle Datentypen



Typische Schritte (Recap)



Sequenzen sind besondere Datentypen, die eine Vielzahl an Operationen bieten (für Strings schon einiges kennengelernt):

- **in** – prüft, ob sich das Element im Datencontainer befindet.
- **+ / +=** – fügt zwei Sequenz zusammen und liefert eine neue Sequenz.
- ***** – Wiederholt die Sequenz n-mal.
- **[index]** – indizierter Zugriff und liefert das i-te Element / mit **[-1]** letztes Element
- **[start:end]** – **Slicing** und liefert die Elemente von Position start bis exklusive end als neue Sequenz. Dabei gibt es zwei interessante Varianten.
 - **index(elem)** – gibt die Position des gesuchten Elements zurück
 - **len()** – Anzahl an Elementen im Datencontainer.
 - **min() / max()** – Ermitteln das Element mit dem kleinsten bzw. größten Wert
 - **sum()** – Summiert die Werte (nur für numerische Werte).



Listen



Listen



- **Erzeugen & Konkatenieren // Java-Analogie: Collection-Factory-Methoden**

```
>>> list()
[]
>>>
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> zeros = [0] * 10
>>> zeros
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>>
>>> zeros += [None] * 5
>>> zeros
[0, 0, 0, 0, 0, 0, 0, 0, 0, None, None, None, None, None]
```

- **Heterogene Zusammensetzung möglich (wie früher in Java ohne Generics)**

```
>>> [0, "ABC", 42.195, True]
[0, 'ABC', 42.195, True]
```

Listen / Sequentielle Datentypen (wie schon bei Strings)



- **Auf Elemente zugreifen**

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers[0]
0
>>> numbers[7]
7
```

- **Besonderheit: Negative Indizes möglich (-idx ⇔ len(txt) - idx)**

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers[-1]
9
>>> numbers[-2]
8
>>> numbers[-9]
1
>>> numbers[-10]
0
```



```
>>> numbers[len(content) - 1]
9
>>> numbers[len(content) - 2]
8
>>> numbers[len(content) - 9]
1
>>> numbers[len(content) - 10]
0
```



- **Bereichsverletzung (auch bei negativem Index)**

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> numbers[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
>>> numbers[-42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Listen



- **positionsbasiert durchlaufen**

```
>>> values = [0, "ABC", 42.195, True]
>>> for i in range(len(values)):
...     print(i, values[i], end=',')
0,0,1,ABC,2,42.195,3,True,
```

- **elementbasiert**

```
>>> for value in values:
...     print(value, end=',')
0,ABC,42.195,True,
```

- **Kombination: Mit Position und aktuellem Element**

```
for i, value in enumerate(values):
    print(i, value, end=',')
0,0,1,ABC,2,42.195,3,True,
```

- **Kombination mit sorted() / reversed() wie bei Schleifen möglich**

Beispiel: Konkatenation von Listen



Ziel ist eine Ausgabe von 1st of May usw.

Dazu ist eine Liste geeignet zu befüllen:

```
endings = ["st", "nd", "rd"] + 17 * ["th"] + \
          ["st", "nd", "rd"] + 7 * ["th"] + \
          ["st"]
```

```
for i in range(1, 32):
    print(str(i) + endings[i-1])
```

1st
2nd
3rd
4th
5th
6th
7th
8th
9th
10th
...
21st
22nd
23rd
24th
...
30th
31st

Listen



- **Basisfunktionalitäten**

```
numbers = [11, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
print("len:", len(numbers))  
print("min:", min(numbers))  
print("max:", max(numbers))  
print("sum:", sum(numbers))
```



```
len: 9  
min: 2  
max: 11  
sum: 55
```

- **Nettigkeiten**

```
numbers.reverse()  
print("reverse:", numbers)  
numbers.sort()  
print("sort:", numbers)
```



```
reverse: [9, 8, 7, 6, 5, 4, 3, 2, 11]  
sort: [2, 3, 4, 5, 6, 7, 8, 9, 11]
```

- **Alternativen: reversed() / sorted(), die Iteratoren liefern ... später mehr**

Listen -- Basisfunktionalitäten



- **Hinzufügen**

```
names = []
```

```
names.append("Tim")
names.append("Tom")
print(names)
```

```
['Tim', 'Tom']
```

```
names.insert(0, "Anton")
names.insert(0, "Andreas")
print(names)
```

```
['Andreas', 'Anton ', 'Tim', 'Tom']
```

```
names.insert(4, "Last")
print(names)
```

```
['Andreas', 'Anton ', 'Tim', 'Tom', 'Last']
```

Listen – Ausflug Fibonacci-Zahlen



- **Ausflug Fibonacci-Zahlen (rekursive Definition):**

```
>>> def fib(n):
...     if n <= 1:
...         return n
...     return fib(n-2) + fib(n-1)
...
>>> fib(5)
5
```

- **Fibonacci Berechnungs-Trick:**

```
>>> fib_numbers = [0, 1]
>>> for i in range(10):
...     fib_numbers.append(fib_numbers[-2] + fib_numbers[-1])
...
>>> fib_numbers
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Listen – Basisfunktionalitäten



- Ändern

```
names[1] = "Mike"  
print(names)
```

```
['Andreas', 'Mike', 'Tim', 'Tom', 'Last']
```

- Löschen

```
del names[2]  
print(names)
```

```
['Andreas', 'Mike', 'Tom', 'Last']
```

```
names.remove("Tom")  
print(names)
```

```
['Andreas', 'Mike', 'Last']
```

```
names.clear()  
print(names)
```

```
[]
```

```
names.remove("Tim")
```

```
ValueError: list.remove(x): x not in list
```

Listen -- Basisfunktionalitäten



- **Multi-Append (+= / extend())**

```
cities = ["Zürich", "Luzern"]
cities += ["Bremen", "Aachen"]

print(cities)

cities.extend(["Paris", "London"])
print(cities)
```

```
# Fallstrick bei sequentiellen Typen
cities.extend("OSLO")
cities += "KIEL"
print(cities)
```

- =>

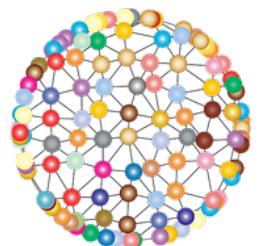
```
['Zürich', 'Luzern', 'Bremen', 'Aachen']
['Zürich', 'Luzern', 'Bremen', 'Aachen', 'Paris', 'London']
['Zürich', 'Luzern', 'Bremen', 'Aachen', 'Paris', 'London', 'O', 'S', 'L', 'O',
 'K', 'I', 'E', 'L']
```

Listen



- **Enthaltensein prüfen (in)**

```
>>> values = (0, "ABC", 42.195, True)
>>>
>>> "ABC" in values
True
>>>
>>> ["ABC", 42.195] in values
False
```



Was macht man mit Wertefolgen?

Listen



- **Enthaltensein prüfen (`in`) – Trick => String-Umwandlung!!**

```
def contains_sequence(seq, values):
    stringified_seq = "".join(str(i) for i in seq)
    stringified_values = "".join(str(i) for i in values)

    return stringified_seq in stringified_values
```

```
print(contains_sequence("ABC", values))
print(contains_sequence(["ABC", 42.195], values))
```



True
True



**Was könnte daran
nicht optimal sein?**

=>

Übungsaufgabe



Slicing



Listen



- **Slicing => Extraktion einer neuen Liste von Position startidx bis endidx**

text[startidx:endidx]

- **Beispiel**

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

```
print(numbers[2:8])  
print(numbers[0:3])  
print(numbers[-3:])
```



```
[2, 3, 4, 5, 6, 7]  
[0, 1, 2]  
[10, 11, 12]
```

Listen



- **Slicing – Besonderheiten**

text[startidx:endidx:step]

- **Beispiel**

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
print(numbers[0::2])
print(numbers[2:10:2])
print(numbers[10:2:-2])
```



```
[0, 2, 4, 6, 8, 10, 12]
[2, 4, 6, 8]
[10, 8, 6, 4]
```

Listen



- **Slicing – Trickserei**

text[startidx:endidx:step] = <WERTE>

```
numbers = [0, 1, 2, 3, 9, 10, 11, 12]
print(numbers)
numbers[4:4] = [4, 5, 6, 7, 8]
print(numbers)
numbers[4:8] = [44, 55, 66, 77, 88]
print(numbers)
numbers[0:14:2] = ["X", "X", "X", "X", "X", "X", "X"]
print(numbers)
```



```
[0, 1, 2, 3, 9, 10, 11, 12]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
[0, 1, 2, 3, 44, 55, 66, 77, 88, 8, 9, 10, 11, 12]
['X', 1, 'X', 3, 'X', 55, 'X', 77, 'X', 8, 'X', 10, 'X', 12]
```



List Comprehensions



List Comprehension



- Berechnungsvorschrift angeben

[expr for value in iterables]

- Beispiel

```
>>> [value for value in range(10)]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>>  
>>> [value * value for value in range(10)]  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List Comprehension



- Berechnungsvorschrift inklusive Bedingung angeben

[<expr> for value in <iterables> if <condition>]

- Beispiel

```
>>> [value * 2 for value in range(10) if value % 2 != 0]
[2, 6, 10, 14, 18]
>>>
>>> [value * value for value in range(10) if value % 2 == 0]
[0, 4, 16, 36, 64]
```

List Comprehension



- «Komplexere» Berechnungsvorschrift und mehrere Schleifen

```
>>> [[x, y * 2] for x in range(3) for y in range(5)]
```

- Ohne Comprehension

```
>>> result = []
>>> for x in range(3):
...     for y in range(5):
...         result.append([x, y * 2])
```

- =>

```
[[0, 0], [0, 2], [0, 4], [0, 6], [0, 8],
[1, 0], [1, 2], [1, 4], [1, 6], [1, 8],
[2, 0], [2, 2], [2, 4], [2, 6], [2, 8]]
```

List Comprehension – Komplexere Berechnungsvorschrift und mehrere Schleifen



- **Ausgangsdaten**

```
>>> boys = ["tim", "tom", "michael", "john", "james"]
>>> girls = ["tanja", "michaela", "julia", "lilija", "anne"]
```

- **Alle mit gleichem Startbuchstaben:**

```
>>> [b+"+"+g for b in boys for g in girls if b[0] == g[0]]
['tim+tanja', 'tom+tanja', 'michael+michaela', 'john+julia', 'james+julia']
```

- **Alle Kombinationen, wo ein Buchstabe aus dem Frauennamen im
Männernamen vorkommt:**

```
>>> [b+"+"+g for b in boys for g in girls if g[0] in b]
['tim+tanja', 'tim+michaela', 'tom+tanja', 'tom+michaela', 'michael+michaela',
'michael+lilija', 'michael+anne', 'john+julia', 'james+michaela',
'james+julia', 'james+anne']
```



Tupel



Tupel



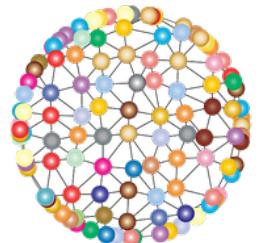
- **Nahezu wie Listen, aber unveränderlich**

```
>>> (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>>
>>> (0, "ABC", 42.195, True)
(0, 'ABC', 42.195, True)
>>> (1)
1
>>> (1,)
(1,)
```

- **Aber Achtung: KEINE Tuple Comprehension!**

```
>>> (value for value in range(10))
<generator object <genexpr> at 0x1050b7220>
```

- **Was ist denn ein Generator?? => später!**
-



**Aber noch wichtiger: Wie
komme ich an die Daten
aus dem Generator?**

Tupel – Ausflug Generatoren



- **Mit `list()` lassen sich die Werte aus einem Generator aufbereiten:**

```
>>> list((value for value in range(10)))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- **Generator stellt Werte «auf Verlangen» bereit, dazu Built-in-Funktion `next()`**

```
>>> squares_generator_expr = (x ** 2 for x in range(2, 70))  
>>> next(squares_generator_expr)  
4  
>>> next(squares_generator_expr)  
9  
>>> next(squares_generator_expr)  
16  
>>> next(squares_generator_expr)  
25  
>>> next(squares_generator_expr)  
36
```

Tuple Packing / Unpacking



- Mehrere Werte (als Tupel) mehreren Variablen zuweisen (Unpacking)

```
>>> x, y, z = (1, 2, 3)
>>> print(x, y, z)
1 2 3
```

- Mehrere Werte einer Variablen als Tupel zuweisen (Packing)

```
>>> values = 1, 2, 3
>>> values
(1, 2, 3)
```

- SWAP-Trick

```
>>> x, y = y, x
>>> print(x, y, z)
2 1 3
```

Tuple Unpacking bei Dicts



- Mehrere Werte (als Tupel) mehreren Variablen zuweisen (Unpacking)

```
cities_inhabitants = { "Kiel" : 250_000,  
                      "Bremen" : 550_000,  
                      "Zürich" : 400_000 }
```

```
city, inhabitant_count =  
cities_inhabitants.popitem()  
print(city, inhabitant_count)
```

```
for key, value in cities_inhabitants.items():  
    print(key, value)
```

- =>

```
Zürich 400000  
Kiel 250000  
Bremen 550000
```

Tuple Unpacking



- **Problem beim Unpacking, wenn zu viele Werte**

```
>>> first, second = (1, 2, 3, 4, "END")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

- **Trick, falls zu viele Werte**

```
>>> first, second, *rest = (1, 2, 3, 4, "END")
>>> print(first, second, rest)
1 2 [3, 4, 'END']
```

Tuple Unpacking



- **Trick, falls zu viele Werte**

```
>>> first, second, *rest = (1, 2, 3, 4, "END")
>>> print(first, second, rest)
1 2 [3, 4, 'END']
```

- **Alternative: Mitte oder gar vorne auffüllen**

```
>>> first, *rest, third = (1, 2, 3, 4, "END")
>>> print(first, rest, third)
1 [2, 3, 4] END
>>>
>>> *rest, first, second = (1, 2, 3, 4, "END")
>>> print(rest, first, second)
[1, 2, 3] 4 END
```

Praxisbeispiel: rindex() selbst realisiert



Suche vom Ende mit rindex() und rfind() leider nur in Strings, nicht aber in sequenziellen Containern:

```
>>> print("Hello".rindex("l")) # => 3
3
>>> print("Hello".rfind("l")) # => 3
3
>>> print("Hello".rfind("x")) # => -1
-1
>>> print("Hello".rindex("x")) # => ValueError: substring not found
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

Praxisbeispiel: rindex() selbst realisiert



Suche vom Ende mit rindex() selbst gebaut:

```
def rindex(values, item):
    reversed_values = values[::-1]
    return len(values) - reversed_values.index(item) - 1
```

```
last_index = lambda values, item: len(values) - values[::-1].index(item) - 1
```

```
values = ['Start', 'End', 'Mid', 'End']
print(values.index("End"))          1
print(rindex(values, "End"))        3
print(last_index(values, "End"))    3
print(rindex(values, "Start"))      0
```

Praxisbeispiel: Stack selbst realisiert



```
class Stack:  
    def __init__(self):  
        self._values = []  
  
    def push(self, elem):  
        self._values.append(elem)  
  
    def pop(self):  
        if self.is_empty():  
            raise StackIsEmptyException()  
        return self._values.pop()  
  
    def peek(self):  
        if self.is_empty():  
            raise StackIsEmptyException()  
        return self._values[-1]  
  
    def is_empty(self):  
        return len(self._values) == 0
```

- `push(element)` – Ein Element oben hinzufügen.
- `pop()` – Das oberste Element nehmen.
- `peek()` – Einen Blick auf das oberste Element werfen.
- `is_empty()` – Prüft, ob der Stack leer ist.



DEMO

`lists_*.py`



Mengen (set)



Sets – Basisfunktionalitäten



- Ein set ist eine Sammlung (Menge) von Elementen.
- mathematische Konzept => keine Duplikate
- Somit bilden Sets eine ungeordnete, duplikatfreie Datenstruktur, aber bieten keinen indizierten Zugriff.
- Stattdessen einige Mengenoperationen insbesondere die Berechnung von Vereinigungs-, Schnitt-, Differenz- und symmetrischen Differenzmengen:

Vereinigung $A \mid B \rightarrow A \cup B$

Schnitt $A \& B \rightarrow A \cap B$

Differenz $A - B \rightarrow A \setminus B$ $[A - B \rightarrow A \setminus B]$

Symmetrische
Differenz $A \wedge B \rightarrow A \Delta B$

Sets



- **Erzeugen // Java-Analogie: Collection-Factory-Methoden**

```
>>> set()
set()
>>>
>>> numbers = {0, 1, 2, 3, 4, 5, 6, 7}
>>> numbers
{0, 1, 2, 3, 4, 5, 6, 7}
>>> type(numbers)
<class 'set'>
```

- **Heterogene Zusammensetzung möglich (wie früher in Java ohne Generics)**

```
>>> {0, "ABC", 42.195, True}
{0, True, 42.195, 'ABC'}
```

Sets



- **Basisfunktionalitäten**

```
numbers = {11, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
print("len:", len(numbers))  
print("min:", min(numbers))  
print("max:", max(numbers))  
print("sum:", sum(numbers))
```



```
len: 9  
min: 2  
max: 11  
sum: 55
```

- **elementbasiert durchlaufen**

```
>>> values = {0, "ABC", 42.195, True}
```

```
>>> for value in values:  
...     print(value, end=',')
```



```
0,ABC,42.195,True,
```

Sets -- Basisfunktionalitäten



- **Hinzufügen**

```
names = set()  
names.add("Tim")  
names.add("Tom")  
print(names)           {'Tim', 'Tom'}
```

```
names.add("Tim")  
names.add("Last")  
print(names)           {'Tim', 'Last', 'Tom'}
```

```
names.update(["Mike", "Peter", "John"])  
print(names)           {'Mike', 'Tom', 'John', 'Tim', 'Peter', 'Last'}
```

- **Fallstrick bei sequentiellen Typen**

```
names.update("Michael")  
print(names)
```

```
{'John', 'i', 'Mike', 'l', 'M', 'Peter', 'e', 'Tom', 'a', 'c', 'Last', 'Tim', 'h'}
```

Sets – Basisfunktionalitäten



- **Löschen**

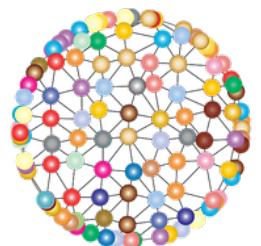
```
for letter in "Michael":  
    names.remove(letter)  
print(names)                                {'Last', 'Mike', 'John', 'Tim', 'Peter', 'Tom'}  
  
names.remove("Tom")  
print(names)                                {'Peter', 'John', 'Tim', 'Mike'}  
  
names.clear()  
print(names)                                set()  
  
names.remove("Tim")                          KeyError: 'Tim'
```

Set



- **Enthaltensein prüfen (in)**

```
>>> values = {0, "ABC", 42.195, True}  
>>>  
>>> "ABC" in values  
True  
>>>  
>>> {"ABC", 42.195} in values  
False
```



**Was macht man mit
Wertefolgen?**

Sets – Mengenoperationen



- Mengenoperationen: Berechnung von Vereinigungs-, Schnitt-, Differenz- und symmetrischen Differenzmengen:

Vereinigung $A \mid B \rightarrow A \cup B$

Schnitt $A \& B \rightarrow A \cap B$

Differenz $A - B \rightarrow A \setminus B$ $[A - B \rightarrow A \setminus B]$

Symmetrische
Differenz $A \wedge B \rightarrow A \Delta B$

Sets – Mengenoperationen

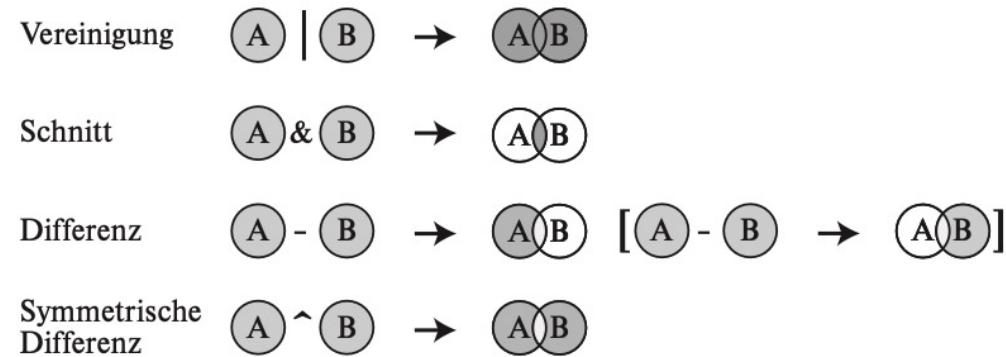


- **Mengenoperationen**

```
number_set1 = {1, 2, 3, 4, 5, 6, 7, 8}  
number_set2 = {2, 3, 5, 7, 9, 11, 13}  
print("union: %s\nintersection: %s" \  
      "\ndiff 1-2: %s\ndiff 2-1: %s" \  
      "\nsym diff: %s" %  
      ((number_set1 | number_set2),  
       (number_set1 & number_set2),  
       (number_set1 - number_set2),  
       (number_set2 - number_set1),  
       (number_set1 ^ number_set2)))
```

- =>

```
union: {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13}  
intersection: {2, 3, 5, 7}  
diff 1-2: {8, 1, 4, 6}  
diff 2-1: {9, 11, 13}  
sym diff: {1, 4, 6, 8, 9, 11, 13}
```



Sets – Mengenoperationen



- **Kommen wir zur Subset-Prüfung zurück**
- **Wenn A & B = B => dann ist es ein Subset**

```
>>> {"ABC", 42.195} & {0, "ABC", 42.195, True} == {"ABC", 42.195}  
True
```

- **Dafür existieren bereits passende Methoden**

```
print({"ABC", 42.195}.issubset({0, "ABC", 42.195, True}))  
print({0, "ABC", 42.195, True}.issuperset({"ABC", 42.195}))
```



Dictionaries (dict)



Dicts – Basisfunktionalitäten



- **Abbildungen von Schlüsseln auf Werte**
 - **Java-Analogie: Map**
 - **auch als Dictionary oder als Lookup-Tabelle bezeichnet – andere Begriffe sind assoziatives Array bzw. Hash.**
 - **zugrunde liegende Idee, jedem gespeicherten Wert einen eindeutigen Schlüssel zuzuordnen.**
 - **Ein intuitiv verständliches Beispiel sind Telefonbücher, bei denen Namen auf Telefonnummern abgebildet werden. Eine Suche über einen Namen (Schlüssel) liefert meistens recht schnell eine Telefonnummer (Wert).**
 - **Falls keine Rückabbildung von Telefonnummer auf Namen existiert, wird das Ermitteln eines Namens zu einer Telefonnummer ziemlich aufwendig.**
-

Dicts



- **Erzeugen (leer und mit vordefinierten Werten) // Java-Analogie: Collection-Factory-Methoden**

```
>>> dict()
{}
>>>
>>> {}
{}
>>> type({})
<class 'dict'>
>>>
>>> city_inhabitants_map = {}
>>>
>>> bern_berlin_map = { "Bern" : 170_000, "Berlin" : 3_500_000 }
>>> bern_berlin_map
{'Bern': 170000, 'Berlin': 3500000}
```

Dicts -- Basisfunktionalitäten



- ## • Hinzufügen

```
city_inhabitants_map = {}
```

```
city_inhabitants_map["Zürich"] = 400_000
```

```
city_inhabitants_map["Hamburg"] = 2_000_000
```

```
city_inhabitants_map["Kiel"] = 250_000
```

```
print(city_inhabitants_map)
```

```
{'Zürich': 400000, 'Hamburg': 2000000, 'Kiel': 250000}
```

```
city_inhabitants_map.update({ "Bern" : 170_000, "Berlin" : 3_500_000 })
```

```
print(city_inhabitants_map)
```

```
{'Zürich': 400000, 'Hamburg': 2000000, 'Kiel': 250000,  
'Bern': 170000, 'Berlin': 3500000}
```

Dicts -- Durch die Elemente iterieren



- Bei Dictionaries existiert kein indizierter Zugriff
- Allerdings lassen sich die Abbildungen mit folgenden Methoden auslesen:
 1. `items()` – Erzeugt eine Liste, die alle Schlüssel-Wert-Paare des Dictionaries als Tupel.
 2. `keys()` – Liefert eine Liste mit allen im Dictionary hinterlegten Schlüsseln.
 3. `values()` – Liefert eine Liste mit allen im Dictionary hinterlegten Werten.

```
for key, value in city_inhabitants_map.items():
    print("The city of", key, "has", value, "inhabitants")
```

Dicts -- Durch die Elemente iterieren



- **Beispiel:**

```
for key, value in city_inhabitants_map.items():
    print("The city of", key, "has", value, "inhabitants")

print("-".join(city_inhabitants_map.keys()))
print("Gesamteinwohner:", sum(city_inhabitants_map.values()))
```

- =>

The city of Zürich has 400000 inhabitants
The city of Hamburg has 2000000 inhabitants
The city of Kiel has 250000 inhabitants
The city of Bern has 170000 inhabitants
The city of Berlin has 3500000 inhabitants
Zürich-Hamburg-Kiel-Bern-Berlin
Gesamteinwohner: 6320000

Dicts



- Prüfen, ob ein Eintrag existiert (in)

```
print("Zürich" in city_inhabitants_map)  
print("Bremen" in city_inhabitants_map)
```

True
False

- Prüfen, ob ein Wert existiert (in)

```
print(400_000 in city_inhabitants_map.values())  
print(1_234_567 in city_inhabitants_map.values())
```

True
False

Listen – Basisfunktionalitäten



- **Modifikation ausführen**

```
city_inhabitants_map["Bern"] = 180_000
```

- **Löschen**

```
del city_inhabitants_map["Bern"]
```

```
print(city_inhabitants_map)
```

```
{'Zürich': 400000, 'Hamburg': 2000000,  
'Kiel': 250000, 'Berlin': 3500000}
```

```
city_inhabitants_map.pop("Kiel")
```

```
print(city_inhabitants_map)
```

```
{'Zürich': 400000, 'Hamburg': 2000000,  
'Berlin': 3500000}
```

```
city_inhabitants_map.clear()
```

```
print(len(city_inhabitants_map))
```

```
0
```



Mehrdimensionale Listen



Mehrdimensionale Listen



- Abschließend beschäftigen wir uns kurz mit mehrdimensionalen Listen.
- Weil es in der Praxis häufiger vorkommt sowie weil man es sich auch visuell gut vorstellen kann, betrachten wir nur zweidimensionale Listen.

	0	1	2	3	4
0	0,0	0,1	0,2	0,3	0,4
1	1,0	1,1	1,2	1,3	1,4
2	2,0	2,1	2,2	2,3	2,4

- Eine zweidimensionale rechteckige Liste kann man etwa zur Modellierung eines Spielfelds, eines Sudoku-Rätsels oder einer Landschaft nutzen.

```
#####
# # P      #
###   $ X  ###
### # $  #####
#####
```

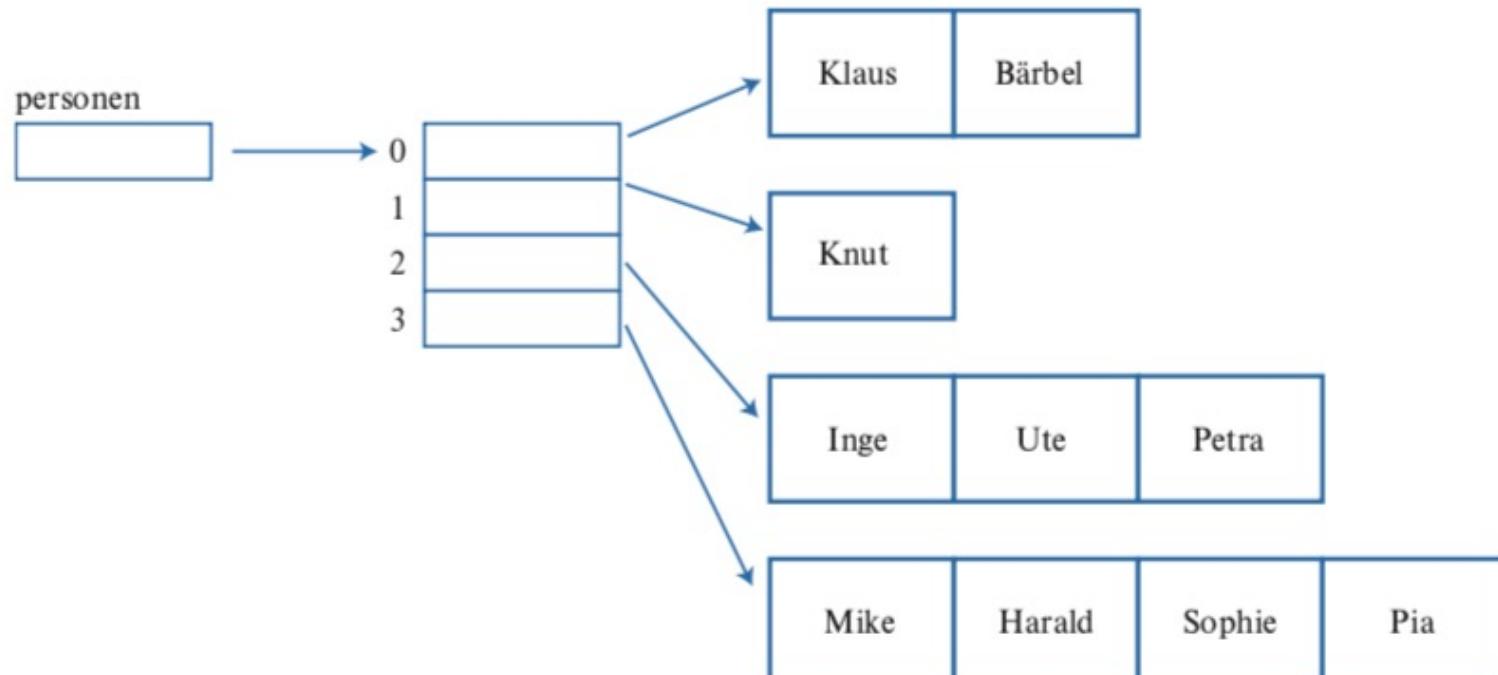
Mehrdimensionale Listen



- **Mehrdimensionale Listen müssen nicht rechteckig sein. Dreieck:**

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

- **Beliebige Verschachtelung:**





Exercises Part 4

<https://github.com/Michaeli71/Python-Workshop.git>





PART 5: Collections Advanced

- Sortierung – Analogie Comparator
 - Iteratoren / Iterator-Java
 - Reihenfolge umkehren – reverse() und reversed()
 - Generatoren
-



Sortierung



Java Comparator



*JAVA Fallstrick ...

```
jshell> var names = List.of("Mike", "Andy", "Peter", "Jim", "Tim")
names ==> [Mike, Andy, Peter, Jim, Tim]
```

```
names.sort(Comparator.naturalOrder())
| Exception java.lang.UnsupportedOperationException
```

```
jshell> var modifiable = new ArrayList<>(names)
modifiable ==> [Mike, Andy, Peter, Jim, Tim]
```

```
jshell> modifiable.sort(Comparator.naturalOrder())
```

```
jshell> modifiable
modifiable ==> [Andy, Jim, Mike, Peter, Tim]
```

Natürliche Ordnung



- **JAVA**

```
jshell> var modifiable = new ArrayList<>(names)  
modifiable ==> [Mike, Andy, Peter, Jim, Tim]
```

```
jshell> modifiable.sort(Comparator.naturalOrder())
```

```
jshell> modifiable  
modifiable ==> [Andy, Jim, Mike, Peter, Tim]
```

- **PYTHON**

```
>>> names = ["Mike", "Andy", "Peter", "Jim", "Tim"]  
>>> names.sort()  
>>> names  
['Andy', 'Jim', 'Mike', 'Peter', 'Tim']
```

Umgekehrte Ordnung



- **JAVA: Reverse order**

```
jshell> modifiable.sort(Comparator.reverseOrder())
```

```
jshell> modifiable  
modifiable ==> [Tim, Peter, Mike, Jim, Andy]
```

- **PYTHON: Reverse order**

```
>>> names.sort(reverse=True)  
>>> names  
['Tim', 'Peter', 'Mike', 'Jim', 'Andy']
```

Spezielle Sortierung nach 2. Buchstaben



- **JAVA: by2ndChar**

```
jshell> modifiable.sort(Comparator.comparing(name -> name.charAt(2)))
```

```
jshell> modifiable  
modifiable ==> [Andy, Mike, Tim, Jim, Peter]
```

- **PYTHON: by2ndChar**

```
>>> names.sort(key=lambda name: name[2])  
>>> names  
['Andy', 'Mike', 'Jim', 'Tim', 'Peter']
```

Spezielle Sortierung nach letztem Buchstaben



- **JAVA: byLastChar**

```
jshell> modifiable.sort(Comparator.comparing(name -> name.charAt(name.length() - 1)))
```

```
jshell> modifiable  
modifiable ==> [Mike, Tim, Jim, Peter, Andy]
```

- **PYTHON: byLastChar**

```
>>> names.sort(key=lambda name: name[-1])  
>>> print(names)  
['Mike', 'Tim', 'Jim', 'Peter', 'Andy']
```

Spezielle Sortierung nach letztem Buchstaben



- **JAVA: byLastChar**

```
jshell> modifiable.sort(Comparator.comparing(String::length))
```

```
jshell> modifiable  
modifiable ==> [Jim, Tim, Mike, Andy, Peter]
```

- **PYTHON: Nach Länge und auch absteigend**

```
>>> names = ["Mike", "Andy", "Peter", "Jim", "Tim"]  
>>>  
>>> names.sort(key=len)  
>>> names  
['Jim', 'Tim', 'Mike', 'Andy', 'Peter']  
>>>  
>>> names.sort(key=len, reverse=True)  
>>> names  
['Peter', 'Mike', 'Andy', 'Jim', 'Tim']
```

Case-Insensitive Sortierung



- **JAVA:**

```
jshell> var names = new ArrayList<>(List.of("tim", "TOM", "MIKE", "Michael",  
"andreas", "STEFAN"))  
names ==> [tim, TOM, MIKE, Michael, andreas, STEFAN]
```

```
jshell> names.sort(Comparator.comparing(String::toLowerCase))
```

```
jshell> names  
names ==> [andreas, Michael, MIKE, STEFAN, tim, TOM]
```

- **PYTHON: Nach Länge und auch absteigend**

```
names = ["tim", "TOM", "MIKE", "Michael", "andreas", "STEAFN"]
```

```
names.sort(key=str.lower)
```

```
print(names)
```

```
names.sort(key=str.casefold)
```

```
print(names)
```

```
['andreas', 'Michael', 'MIKE', 'STEFAN', 'tim', 'TOM']
```

Mehrfach Sortierung



- Nach Länge, dann alphabetisch durch Angabe eines Tupels

```
values = ["a", "cc", "bbb", "dddd", "ee", "f", "d", "eee"]
print(sorted(values, key=lambda x: (len(x), x)))
```

- =>

```
['a', 'd', 'f', 'cc', 'ee', 'bbb', 'eee', 'dddd']
```

Mehrfach Sortierung



- **Absteigend nach Länge und aufsteigend / absteigend nach Wert**

```
values = ["a", "cc", "bbb", "dddd", "ee", "f", "d", "eee"]
print(sorted(values, key=lambda x: (-len(x), x)))
```

```
values = ["a", "cc", "bbb", "dddd", "ee", "f", "d", "eee"]
print(sorted(values, key=lambda x: (len(x), x), reverse=True))
```

- =>

```
['dddd', 'bbb', 'eee', 'cc', 'ee', 'a', 'd', 'f']
['dddd', 'eee', 'bbb', 'ee', 'cc', 'f', 'd', 'a']
```

Mehrfach Sortierung



- Nach Anzahl Programmiersprachen absteigend, nach Namen aufsteigend

```
programmers = [("Tim", ["Java"]), ("Tom", ["Java", "Python"]),
               ("Polyglotti", ["Java", "Python", "C#", "C++"])]
```

```
print(sorted(programmers, key=lambda x: (-len(x[1]), x[0])))
```

- =>

```
[('Polyglotti', ['Java', 'Python', 'C#', 'C++']),
 ('Michael', ['Java', 'Python']), ('Tom', ['Java', 'Python']),
 ('Jim', ['Python']), ('Tim', ['Java'])]
```

Datenmodell



```
from collections import namedtuple

Person = namedtuple('Person', ['name', 'age'])

persons = [Person("Mike", 30), Person("Tim", 50),
           Person("Michael", 50), Person("Tom", 30),
           Person("Arne", 7), Person("Jim", 30)]
```

Sortierung – nach einem Kriterium



```
# Alter absteigend
persons.sort(key=lambda p: p.age, reverse=True)
print(persons)
```

```
persons.sort(key=lambda p: -p.age)
print(persons)
```

```
# Name absteigend
persons.sort(key=lambda p: p.name, reverse=True)
print(persons)
```

```
#persons.sort(key=lambda p: -p.name)
```

=>

```
[Person(name='Tim', age=50), Person(name='Michael', age=50), Person(name='Mike', age=30),
Person(name='Tom', age=30), Person(name='Jim', age=30), Person(name='Arne', age=7)]
[Person(name='Tim', age=50), Person(name='Michael', age=50), Person(name='Mike', age=30),
Person(name='Tom', age=30), Person(name='Jim', age=30), Person(name='Arne', age=7)]
[Person(name='Tom', age=30), Person(name='Tim', age=50), Person(name='Mike', age=30),
Person(name='Michael', age=50), Person(name='Jim', age=30), Person(name='Arne', age=7)]
```

Sortierung – nach zwei Kriterien



```
# Name, Alter absteigend
persons.sort(key=lambda p: (p.name, -p.age))
print(persons)
```

```
# Alter, Name
persons.sort(key=lambda p: (p.age, p.name))
print(persons)
```

```
# Alter, Name absteigend
persons.sort(key=lambda p: (-p.age, p.name), reverse=True)
print(persons)
```

=>

```
[Person(name='Arne', age=7), Person(name='Jim', age=30), Person(name='Michael', age=50),
Person(name='Mike', age=30), Person(name='Tim', age=50), Person(name='Tom', age=30)]
```

```
[Person(name='Arne', age=7), Person(name='Jim', age=30), Person(name='Mike', age=30),
Person(name='Tom', age=30), Person(name='Michael', age=50), Person(name='Tim', age=50)]
```

```
[Person(name='Arne', age=7), Person(name='Tom', age=30), Person(name='Mike', age=30),
Person(name='Jim', age=30), Person(name='Tim', age=50), Person(name='Michael', age=50)]
```



Iteratoren



Iteratoren – Grundlagen



- Iteratoren werden in Python vielfach verwendet, etwa für **for-Schleifen, Comprehensions usw.**
- Ein Iterator ist eine Art Zeiger, der auf das aktuelle Element verweist und das nächste Element eines Datencontainers liefern kann.
- Dazu muss ein Iterator-Objekt in Python zwei Methoden implementieren:
 - `__iter__()`
 - `__next__()`
- Diese bilden zusammen das **Iterator-Protokoll**.
- Man bezeichnet ein Objekt als Iterable, wenn man von diesem einen Iterator erhalten kann. Dazu dient die Funktion `iter()`, die intern `__iter__()` aufruft. Die Eigenschaft Iterable gilt etwa für die meisten eingebauten Container in Python wie `list` und `set`, aber auch für `str`.

Iteratoren



- **Iteratoren im Einsatz – Analog zu Java-Iterator**

```
>>> it = iter([2, 4, 6, 8, 10])
>>> next(it)
2
>>> next(it)
4
>>> next(it)
6
>>> next(it)
8
>>> next(it)
10
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iteratoren

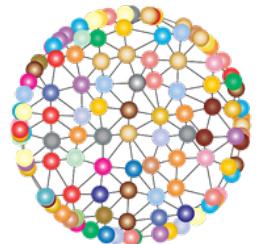


- Iteratoren im Einsatz – Analog zu Java-Iterator
- Verarbeitung analog zu while-Schleife in Java

```
# Holen eines Iterators
iter_obj = iter(iterable)
# Endlosschleife
while True:
    try:
        # Zugriff auf nächstes Element
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # Abbruch im Falle einer StopIteration
        break
```

```
// Holen eines Iterators
Iterator<T> it = ds.iterator()
while (it.hasNext())
{
    // Zugriff auf nächstes Element
    T element = it.next();
    // do something with element
}
```

- PYTHON: DON'T ASK FOR PERMISSION – ASK FOR FORGIVENESS



**Das wirkt aber nicht
wirklich elegant?**

Iteratoren



- **Abhilfe durch die for-in-Schleife!**

```
for element in iterable:  
    # do something with element
```

- **Intern funktioniert die for-in-Schleife wie gerade gesehen also wie folgt:**

```
# Holen eines Iterators  
iter_obj = iter(iterable)  
# Endlosschleife  
while True:  
    try:  
        # Zugriff auf nächstes Element  
        element = next(iter_obj)  
        # do something with element  
    except StopIteration:  
        # Abbruch im Falle einer StopIteration  
        break
```

Eigene Iteratoren



- Implementieren wir einen eigenen Iterator für die Fakultät mit `__iter__` und `__next__`:

```
class FactorialIterator:  
    def __init__(self, n=0):  
        self.n = n  
        self.result = 1  
        self.iteration = 1  
  
    def __iter__(self):  
        return self  
  
    def __next__(self):  
        if self.iteration > self.n:  
            raise StopIteration  
  
        self.result *= self.iteration  
        self.iteration += 1  
        return self.result
```

Eigener Iterator im Einsatz



- **Handgestrickt**

```
# create an Iterator
```

```
numbers = FactorialIterator(5)
```

```
# create an iterable from the object
```

```
i = iter(numbers)
```

```
# Using next to get to the next iterator element
```

```
print(next(i))
```

1

```
print(next(i))
```

2

```
print(next(i))
```

6

```
print(next(i))
```

24

```
print(next(i))
```

120

- **Mit for-Schleife**

```
for i in FactorialIterator(5):  
    print(i)
```



DEMO

[factorial_iterator.py](#)



Generatoren





- **RECAP Iterator-Implementierung:**
 - Es ist etwas Arbeit, einen Iterator in Python zu erstellen.
 - Wie gesehen, müssen wir dazu in einer Klasse die beiden Methoden `__iter__()`- und `__next__()` implementieren.
 - Zudem wird eine interne Datenhaltung benötigt, ebenso wie das Auslösen einer `StopIteration`, wenn es keine Werte mehr zurückzugeben gibt.
- **Wenn man es noch etwas einfacher haben möchte, dann sind Generatoren eine einfache Möglichkeit, Iteratoren zu erstellen.**
- **Praktischerweise werden dann die zuvor beschriebenen und bei der Implementierung des eigenen Iterators kennengelernten Aktionen in Python automatisch von Generatoren erledigt.**

Generatoren



- In Python ist es sehr einfach, einen Generator zu erstellen: Man muss nämlich lediglich eine normale Funktion definieren, die jedoch bei der Rückgabe der Werte statt return die yield-Anweisung nutzt:

```
def my_first_generator():
    n = 1
    print('before first')
    yield n
    n *= 2
    print('before second')
    yield n
    n *= 3
    print('before third and last')
    yield n
```

```
gen = my_first_generator()
print(next(gen))
print(next(gen))
print(next(gen))
```

```
before first
1
before second
2
before third and last
6
```

Generatoren – Unterschied `yield` und `return`



- Eine Funktion wird automatisch zu einer Generatorfunktion, falls sie mindestens eine `yield`-Anweisung enthält.
- Worin besteht der Unterschied zwischen `yield` und `return`, wenn beide doch zur Rückgabe eines Werts aus einer Funktion dienen?
- Bekanntermaßen beendet ein Aufruf von `return` die Abarbeitung der Funktion vollständig. Dagegen gibt `yield` zwar den Wert zurück und auch die Kontrolle an den Aufrufer, aber mit folgender Besonderheit:
 - Innerhalb der Funktion werden alle ihre Zustände (also die Wertbelegungen der Variablen) gespeichert.
 - Bei späteren Aufrufen wird dann basierend darauf an dieser Stelle weitergearbeitet.

Eigener einfacher Generator



- **Generator, der vier Werte (Grüsse) liefert:**

```
def greet_generator():
    yield "Hallo"
    yield "Hello"
    yield "Moin"
    yield "Grüezi"
```

```
greetings = greet_generator()
print(next(greetings))          Hallo
print(next(greetings))          Hello
print(next(greetings))          Moin
print(next(greetings))          Grüezi
```

→

```
for greet in greet_generator():
    print(greet)
```

```
Haloo
Hello
Moin
Grüezi
```

Generatoren – Unterschied Generator und Iterator



- Eine Generatorfunktion enthält eine oder mehrere `yield`-Anweisungen.
 - Eine Generatorfunktion gibt einen Iterator zurück.
 - Die Methoden `__iter__()` und `__next__()` werden ohne unser Zutun automatisch implementiert. Wie von Iterator bekannt, können wir mit `next()` durch die Elemente iterieren.
 - Sobald die Funktion einen Wert mit `yield` zurückliefert, wird die Ausführung der Generatorfunktion **pausiert** und die Kontrolle an den Aufrufer übergeben.
 - Lokale Variablen und ihre Zustände werden zwischen Aufrufen gemerkt.
 - Am Ende wird automatisch eine StopIteration ausgelöst.
 - => Generatoren sind ein mächtiges Sprachmittel
-

Eigener Fakultäts-Generator



```
def fac_generator(n = 0):
    iteration = 1
    result = 1
    while iteration <= n:
        yield result
        iteration += 1
        result *= iteration
```

```
numbers = fac_generator(5)
print(next(numbers))
print(next(numbers))
print(next(numbers))
print(next(numbers))
print(next(numbers))
```



```
1
2
6
24
120
1
2
6
24
120
```

```
for i in fac_generator(5):
    print(i)
```

Kombination von Generatoren



```
def fac_generator(n = 0):  
    iteration = 1  
    result = 1  
    while iteration <= n:  
        yield result  
        iteration += 1  
        result *= iteration
```

```
def square(nums):  
    for num in nums:  
        yield num ** 2
```



```
1  
4  
36  
576  
14400
```

```
for i in square(fac_generator(5)):  
    print(i)
```

Unendliche Wertefolge-Generator



- Generatoren modellieren einen unendlichen Datenstrom. Eigentlich können unendliche Datenströme nicht im Speicher verwaltet werden.
- Mit Generatoren möglich, da jeweils nur ein Element erzeugt wird:

```
def endless_generator():
    result = 1
    while True:
        yield result
        result += 1
```

```
numbers = endless_generator()
print(next(numbers))
print(next(numbers))
```

```
for i in endless_generator():
    print(i)
    if i > 1_000:
        break
```



```
1
2
1
2
3
4
...
999
1000
1001
```

Generator-Comprehension



- Bei Listen sind wir schon darüber gestolpert ...

```
>>> nums_squared_lc = [num**2 for num in range(5)]
>>> nums_squared_lc
[0, 1, 4, 9, 16]
>>> nums_squared_gc = (num**2 for num in range(5))
>>> nums_squared_gc
<generator object <genexpr> at 0x10591f450>
```

- Was passiert aber, wenn wir den Wertebereich massiv vergrößern?

Generator-Comprehension



- Bei Listen sind wir schon darüber gestolpert ...

```
>>> nums_squared_gc = (num**2 for num in range(5))
>>> nums_squared_gc
<generator object <genexpr> at 0x10591f450>
```

- Was passiert aber, wenn wir den Wertebereich massiv vergrößern?

```
>>> import sys
>>> nums_squared_lc = [n * 2 for n in range(1_000_000)]
>>> sys.getsizeof(nums_squared_lc)
8448728
>>>
>>> nums_squared_gc = (i ** 2 for i in range(1_000_000))
>>> print(sys.getsizeof(nums_squared_gc))
104
```



DEMO

[factorial_generator.py](#)



Exercises Part 5

<https://github.com/Michaeli71/Python-Workshop.git>





PART 6: Exception Handling



- Fehler können beim Programmieren eigentlich immer und überall auftreten:

```
>>> 7 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> names = ["Tim", "Tom", "Mike"]
>>> names[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

- Wenn als Folge des Fehlers eine Exception (oftmals mit Namensendung Error oder Exception) auftritt, dann stoppt die Programmausführung.
 - Wichtig ist eine geeignete Reaktion darauf
-



- **Exceptions sehr ähnlich zu Java mit try-catch in Python mit try-except**
- **Für verschiedene Arten von Problemen verschiedene Typen von Exceptions. Vier recht gebräuchliche vordefinierte Typen sind diese:**
 - **ValueError** – Mit einem ValueError können falsche Belegungen von Parametern ausgedrückt werden.
 - **IndexError** – Zu einem IndexError kommt es bei fehlerhaften Indexangaben beim Zugriff etwa auf eine Liste.
 - **KeyError** – Mit einem KeyError werden fehlerhafte Zugriffe (nicht existenter Schlüssel) auf Dictionaries ausgedrückt.
 - **ZeroDivisionError** – Auf eine unerlaubte Division durch Null kann mittels einem ZeroDivisionError hingewiesen werden.



- Exceptions sehr ähnlich zu Java mit try-catch in Python mit try-except

```
try:  
    # Hier können Exceptions auftreten  
except Exception-Typ1:  
    # Hier können Fehlersituation behandelt werden  
except Exception-Typ2:  
    # Hier können Fehlersituation behandelt werden
```

- Beispiel

```
try:  
    value = int("7271")  
    print("after parsing 1. result", value)  
    value = int("ERROR")  
    print("after parsing 2. result", value)  
except ValueError:  
    print("can't parse to integer")
```

Auf mehrere Exceptions reagieren



- **Auf mehrere Exceptions reagieren (Java: Multi Catch)**

```
names = ["Tim", "Tom", "Mike"]
for i in range(5):
    try:
        value = int(names[i])
    except (ValueError, ZeroDivisionError):
        print("can't parse to integer")
    except IndexError:
        print("wrong index")
```

- **Ergebnis**

```
can't parse to integer
can't parse to integer
can't parse to integer
wrong index
wrong index
```

Auf mehrere Exceptions reagieren



- **Auf mehrere Exceptions reagieren und Zugriff auf Exception**

```
names = ["Tim", "Tom", "Mike"]
for i in range(5):
    try:
        value = int(names[i])
    except (ValueError, ZeroDivisionError) as ex:
        print("can't parse to integer:", ex)
    except IndexError as ie:
        print("wrong index:", ie)
```

- **Ergebnis**

```
can't parse to integer: invalid literal for int() with base 10: 'Tim'
can't parse to integer: invalid literal for int() with base 10: 'Tom'
can't parse to integer: invalid literal for int() with base 10: 'Mike'
wrong index: list index out of range
wrong index: list index out of range
```

Unspezifisches Exception Handling



- **Unspezifisch auf mehrere Exceptions reagieren**

```
names = ["Tim", "Tom", "Mike"]
try:
    print("INVALID INDEX: " + names[42])
except:
    print("an unspecified error occurred. seams to be wrong index")
```

- **Ergebnis**

an unspecified error occurred. seams to be wrong index

- **ABER: Mit der gezeigten Art lassen sich Fehlersituationen nicht unterscheiden und somit kann man nicht adäquat auf unterschiedliche Probleme reagieren.**
-



- **Folgende grundsätzliche Struktur**

try:

Hier können Exceptions auftreten

except:

Hier werden Exceptions abgearbeitet

finally:

Wird immer durchlaufen, ist allerdings optional

- **Beispiel**

```
names = ["Tim", "Tom", "Mike"]
```

try:

```
    print("INVALID INDEX: " + names[42])
```

except:

```
    print("wrong index")
```

finally:

```
    print("ALWAYS EXECUTED")
```



- **Freigabe von Systemressourcen**

```
file = open("test.txt", encoding = 'utf-8')
try:
    # Datei lesen und verarbeiten
finally:
    file.close()
```

- **Kurzform analog zu Java-ARM mit ContextManager (with)**

```
with open("test.txt", encoding = 'utf-8') as file:
    # Datei lesen und verarbeiten
```

Python-Besonderheit try-else



- In manchen Situationen möchten Sie vielleicht bestimmte Anweisungen nur dann ausführen, wenn der try-Block ohne Fehler abgearbeitet wurde.
- finally wird aber in jedem Fall ausgeführt
- Hier kommt der abschließende else-Block ins Spiel

```
try:  
    num = int(input("Enter a number between 0 - 99: "))  
    assert 0 <= num < 100  
except AssertionError:  
    print("Not a valid number between 0 - 99!")  
else:  
    reciprocal = 1/num  
    print(reciprocal)
```

Exceptions selbst auslösen – raise



- Mittlerweile haben wir schon mehrmals gesehen, dass bei der Abarbeitung von Programmen in gewissen Fehlersituationen automatisch Exceptions ausgelöst werden.
- Aber auch wir als Programmierer können selbst Exceptions auslösen.
- Dazu dient das Schlüsselwort `raise` in Kombination mit einem Ausnahmetyp.

```
def ensure_value_in_range(value, lower_bound, upper_bound):  
    if value < lower_bound or value > upper_bound:  
        raise ValueError("out of bounds")  
    pass
```

Eigene Exception-Typen definieren



- Neben der Verwendung vordefinierter Exceptions problemlos möglich, eigene Typen von Exceptions zu definieren.

```
class CustomerNotFoundException(Exception):  
    pass
```

- Sinnvoller als eine derart pure Exception ist es natürlich, dort weitere Informationen bereitzuhalten

```
class CustomerNotFoundException(Exception):  
    def __init__(self, customer_id, customer_name):  
        self.id = customer_id  
        self.name = customer_name  
  
    def __str__():  
        return "customer {} with id {} not found".format(self.name, self.id)
```

Eigene Exception-Typen definieren: Exception für Bereichsprüfung



- Wir können natürlich auch die Bereichsprüfung ein wenig netter gestalten, indem wir eine eigene Klasse `ValueOutOfBoundsError` basierend auf `ValueError` definieren:

```
class ValueOutOfBoundsError(ValueError):  
    def __init__(self, value, lower, upper):  
        self.value = value  
        self.lower = lower  
        self.upper = upper  
  
    def __str__():  
        return "{} not in {} - {}".format(self.value, self.lower, self.upper)
```

Propagation von Exceptions



- In `func3()` wird ein `ValueError` ausgelöst. In keiner der Funktionen findet eine Fehlerbehandlung statt:

```
def func1():
    func2()
```

```
def func2():
    func3()
```

```
def func3():
    raise ValueError
```

```
def main():
    func1()
```

- Aufruf: `main() → func1() → func2() → func3()` und Rückpropagation
-

Propagation von Exceptions



- Integrieren wir eine Fehlerbehandlung und lösen zwei unterschiedliche Exceptions aus. Dadurch erfolgt die Propagation bis zum passenden except-Block bzw. bis ganz zur main()-Funktion:

```
def func1(value):  
    func2(value)
```

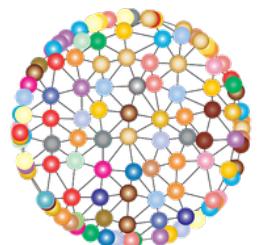
```
def func2(value):  
    try:  
        func3(value)  
    except ValueError as voobe:  
        print(voobe)
```

```
def main():  
    func1(123)  
    func1(None)
```

```
def func3(value):  
    if not value:  
        raise ValueError  
    raise ValueError("value", 1, 100)
```



Was ist eigentlich mit Assertions?



Elegante Prüfungen mit assert



- **assert gibt es in vielen Programmiersprachen**
- **hilft dabei, Probleme frühzeitig in Ihrem Programm erkennen zu können.**
- **Dazu Absicherungen mit folgender Syntax angeben:**
`assert condition`
- **Äquivalent zu**
`if not condition:
 raise AssertionError()`

Elegante Prüfungen mit assert



- **Einsatzgebiete**

- Zusicherungen zur Fehlersuche
- Zusicherungen zur Parameterprüfung

```
def years_to_retirement(age, working_years):  
    assert 18 <= age <= 65, 'age must be in range 18 - 65'  
    assert 10 < working_years < 50, 'years must be in range 11 - 49'
```

```
return 65 - age
```

```
years_to_retirement(70, 10)
```

- **Schlägt wie folgt fehl:**

```
AssertionError: age must be in range 18 - 65
```



PART 7: Dateiverarbeitung





- Ein wichtiger Bestandteil vieler Anwendungen ist die Verarbeitung von Informationen aus Dateien
 - Python besitzt das Modul os, das uns viele nützliche Funktionalitäten zur Verfügung stellt, um mit Verzeichnissen (und auch Dateien) zu arbeiten.
 - Einige weitere Funktionalitäten werden durch das Modul shutil bereitgestellt.
 - bei der Kommunikation und Ein- und Ausgabe immer auch zu Fehlern oder Problemen möglich => Exception Handling schon thematisiert
 - Beispiele bieten nur rudimentäre Fehlerbehandlung, um diese kurz zu halten
-



- Beispiele bieten nur rudimentäre Fehlerbehandlung, um diese kurz zu halten
- Beispiele nutzen folgende Verzeichnisstruktur als Ausgangsbasis

```
files-examples-dir
|-- example-data.csv
|-- example-file.txt
|-- rename-dir
|-- subdir1
\-- subdir2
```



- Beispiele sollen folgende Verzeichnisstruktur als Ausgangsbasis nutzen:

```
files-examples-dir
|--- example-data.csv
|--- example-file.txt
|--- rename-dir
|--- subdir1
\--- subdir2
```

- Wir können ein neues Verzeichnis mit der Funktion `mkdir()` erstellen

```
>>> import os
>>>
>>> os.mkdir('files-examples-dir')
>>> os.mkdir('files-examples-dir/rename-dir')
>>> os.mkdir('files-examples-dir/subdir1')
>>> os.mkdir('files-examples-dir/subdir2')
```



- Zum Erzeugen von Dateien gibt es keinen separaten Funktionsaufruf, sondern, es wird über das Öffnen und einen speziellen Modus realisiert.

```
>>> new_csv_file = open("files-examples-dir/example-data.csv", "x")
```

- Wir wechseln das aktuelle Verzeichnis mit der Funktion `chdir()`:

```
>>> os.chdir("files-examples-dir")
```

- Nun erstellen wir die letzte Datei (ohne Verzeichnisangabe):

```
>>> new_txt_file = open("example-file.txt", "x")
```

- Das aktuelle Verzeichnis lässt sich mit `getcwd()` wie folgt abfragen:

```
>>> print(os.getcwd())
```



- Das aktuelle Verzeichnis lässt sich mit `listdir()` wie folgt auslesen:

```
>>> os.listdir()  
['example-file.txt', 'rename-dir', 'subdir2', 'example-data.csv', 'subdir1']
```

- Datei oder Verzeichnis? (`isfile()` / `isdir()`)

```
>>> dircontent = os.listdir()  
>>> for path in dircontent:  
...     if os.path.isdir(path):  
...         print(path, "is a directory")  
...     if os.path.isfile(path):  
...         print(path, "is a file")  
...
```



- **Größe mit `getsize()` bestimmen:**

```
>>> os.path.getsize(".")  
384
```

- **Existenzprüfung? (`exists()`)**

```
>>> def check_existence(path):  
...     if os.path.exists(path):  
...         print("File", path, "exists")  
...     else:  
...         print("No such file", path)
```

```
>>> check_existence("example-file.txt")  
File example-file.txt exists  
>>> check_existence("UNKNOWN.UFO")  
No such file UNKNOWN.UFO
```

Dateiverarbeitung – Zeitpunkte



Die Zeitpunkte der Erstellung, des letzten Zugriffs bzw. der letzten Änderungen können mit passenden Funktionen abgefragt werden:

```
import datetime
import os

def seconds_to_time(seconds):
    return datetime.datetime.fromtimestamp(seconds)

print("created", seconds_to_time(os.path.getctime("personenliste.txt")))
print("modified", seconds_to_time(os.path.getmtime("personenliste.txt")))
print("accessed", seconds_to_time(os.path.getatime("personenliste.txt")))
```

Achtung: Nur unter Windows liefert `getctime()` den Erstellungszeitpunkt. Unter Unix-artigen Betriebssystemen wird die Zeit der letzten Änderung der Metadaten geliefert.

<https://docs.python.org/3.9/library/os.path.html#os.path.getctime>



DEMO

[**dir_tree_v1/2/3.py**](#)

Dateiverarbeitung – Verarbeitungsmodi



1. Datei mit `open()` öffnen
2. Aktionen (Lesen und / oder schreiben)
3. Datei mit `close()` schließen

```
file = open("test.txt", encoding = 'utf-8')  
# perform file operations  
file.close()
```

- **Verarbeitungsmodi**

- READ - r
- WRITE – w
- APPEND – a
- CREATE - x

```
file_to_read = open("test.txt")      # Standard 'r'  
file_to_write = open("test.txt", 'w') # Schreibmodus  
binary_file_rw = open("img.bmp", 'r+b') # Lesen und Schreiben im Binärmodus
```

Dateiverarbeitung – mit Ressourcehandling & «ARM» (with)



1. Datei mit `open()` öffnen
2. Aktionen (Lesen und / oder Schreiben)
3. Datei mit `close()` schließen

```
file = open("test.txt", encoding='utf-8')
try:
    # perform file operations
    pass
finally:
    file.close()
```

- Praktischerweise wird `close()` passend aufgerufen, wir müssen es nicht mehr explizit schreiben – der Aufruf erfolgt intern

```
with open("test.txt", encoding = 'utf-8') as file:
    # perform file operations
    pass
```



1. Daten schreiben (write())

```
with open("../personenliste.txt", 'w', encoding ='utf-8') as file:  
    file.write("Person1: Michael\n")  
    file.write("Person2: Peter\n")  
    file.write("This file contains three lines\n")
```

2. Daten lesen (read())

```
with open("../personenliste.txt", 'r', encoding = 'utf-8') as file:  
    content = file.read()  
print(content)
```

- Tatsächlich ist das noch leicht unhandlich und erfordert eine Nacharbeit, um an die einzelnen Informationen bzw. Zeilen zu kommen.

Dateiverarbeitung – Texte schreiben / einlesen



1. Daten lesen II (in)

```
with open("../personenliste.txt", 'r', encoding = 'utf-8') as file:  
    # Zeilenweises Lesen  
    for line in file:  
        print(line, end='')
```

2. Daten lesen III (readlines())

```
with open("../personenliste.txt", 'r', encoding = 'utf-8') as file:  
    lines = file.readlines()  
    print("Lines", lines)
```



1. Daten lesen IV (readline())

```
with open("../personenliste.txt", 'r', encoding = 'utf-8') as file:  
    count = 1  
    line_content = file.readline()  
    while line_content:  
        print("Line", count, ":", line_content, end = "")  
        line_content = file.readline()  
        count += 1
```

2. Daten lesen V (readline() mit Walross-Operator := (seit Python 3.8 Assignment-Expr))

```
with open("../personenliste.txt", encoding = 'utf-8') as file:  
    count = 1  
    while line_content := file.readline():  
        print("Line", count, ":", line_content, end = "")  
        count += 1
```



DEMO

personenliste.py

Dateiverarbeitung – Kopieren / Umbenennen / Löschen (shutil)



```
import os
import shutil

shutil.copy('personenliste.py', "copy_of_personenliste.py")
print(os.listdir())

os.rename('copy_of_personenliste.py', 'to-be-deleted.py')
print(os.listdir())

os.remove('to-be-deleted.py')
print(os.listdir())

os.mkdir("new_and_empty")
os.rmdir('new_and_empty')
```

Dateiverarbeitung – Löschen von Verzeichnissen (shutil)



```
import os
import shutil

os.mkdir("dir-to-be-deleted")
open("dir-to-be-deleted/to-be-deleted-1.txt", "x")
os.rmdir('dir-to-be-deleted')
OSError: [Errno 66] Directory not empty: 'dir-to-be-deleted'

shutil.rmtree("dir-to-be-deleted")
```



JSON-Verarbeitung





1. **JSON steht für JavaScript Object Notation**
 2. **ist ein leichtgewichtiges Format für den Datenaustausch**
 3. **In den letzten 5 bis 10 Jahren war JSON eine der, wenn nicht sogar die beliebteste Art, Daten zu verarbeiten.**
- **Daten in einem dict zu speichern. Dort können weitere verschachtelte Dictionaries und Listen oder andere Typen wie Ganzzahlen und Strings enthalten sein.**
 - **Modul json ermöglicht es ...**
 - dict in einen JSON-String umwandeln – man spricht von Serialisieren.
 - Rückwandlung aus JSON-String in ein dict. Dann spricht man von Deserialisieren

JSON-Verarbeitung



- Daten schreiben wir durch Aufruf von `dump()`:

```
import json

data = { "TIM" : { "name": "Tim", "age" : 50, "city" : "Kiel" },
         "MIKE" : { "name": "Mike", "age" : 50, "city" : "Zürich" },
         "INFO": ["This", "is", "an", "information"],
         "CHECKSUM": 4711 }

with open('data.json', 'w') as jsonfile:
    json.dump(data, jsonfile)
```

- Daten lesen wir durch Aufruf von `load()`:

```
with open('data.json') as json_file:
    data = json.load(json_file)
    print(data)
```

JSON-Verarbeitung



```
data = { "TIM" : { "name": "Tim", "age" : 50, "city" : "Kiel" },
         "MIKE" : { "name": "Mike", "age" : 50, "city" : "Zürich" },
         "INFO": [ "This", "is", "an", "information" ],
         "CHECKSUM": 4711 } {  
    "TIM": {  
        "name": "Tim",  
        "age": 50,  
        "city": "Kiel"  
    },  
    "MIKE": {  
        "name": "Mike",  
        "age": 50,  
        "city": "Zu\u0308rich"  
    },  
    "INFO": [  
        "This",  
        "is",  
        "an",  
        "information"  
    ],  
    "CHECKSUM": 4711  
}
```



Exercises Part 7

<https://github.com/Michaeli71/Python-Workshop.git>





PART 8: Datumsverarbeitung





- Python stellt im Modul `datetime` diverse nützliche Funktionalitäten zur Datumsverarbeitung zur Verfügung, deswegen standardmäßig:

```
import datetime
```

- Im Vergleich zum Java Date & Time API aber nicht so umfangreich und ausgereift
- Ein wichtiger Bestandteil vieler Anwendungen ist die Verarbeitung von Informationen aus Datums- und Zeitangaben. Python bietet

	Python	Java
Zeitpunkte	<code>datetime</code>	<code>LocalDateTime</code>
Datumsangaben	<code>date</code>	<code>LocalDate</code>
Zeitangaben	<code>time</code>	<code>LocalTime</code>



- Durch Aufruf der Methode `now()` erhalten wir ein **datetime-Objekt**, das das aktuelle lokale Datum und die Uhrzeit enthält. Zur einfacheren Handhabung erstellen wir eine gleichnamige Hilfsfunktion:

```
def now():
    return datetime.datetime.now()
```

- Wir können ein **datetime-Objekt basierend auf Angaben zu Jahr, Monat, Tag und optional auch Zeit erzeugen.**

```
# datetime(year, month, day)
date_no_time_set = datetime.datetime(2020, 11, 23)
```

```
# datetime(year, month, day, hour, minute, second, microsecond)
date_and_time = datetime.datetime(2017, 11, 28, 23, 55, 59, 342380)
```

Datumsverarbeitung – Zeitpunkte und die Klasse datetime



- **Zuvor haben wir bereits gesehen, dass ein Zeitpunkt aus diversen Bestandteilen besteht. Auf diese können wir per .-Notation und deren Namen zugreifen, etwa auf year, hour und minute:**

```
sunday_afternoon = datetime.datetime(2021, 6, 13, 16, 52, 59)
print("year =", sunday_afternoon.year)
print("month =", sunday_afternoon.month)
print("day =", sunday_afternoon.day)
print("weekday() =", sunday_afternoon.weekday())
print("isoweekday() =", sunday_afternoon.isoweekday())
```

- =>

```
year = 2021
month = 6
day = 13
weekday() = 6
isoweekday() = 7
```



- **datetime-Objekte kann man auch aus einem Unix-Zeitstempel* erzeugen.**
- **Die Methode fromtimestamp() wandelt einen Zeitstempel in ein Datum um.**
- **Eine Rückwandlung geschieht mit timestamp():**

```
datetime_from_ts = datetime.datetime.fromtimestamp(43211234)  
print(datetime_from_ts.timestamp())
```

- =>

43211234.0

- **Praktische Funktion:**

```
def seconds_to_datetime(seconds):  
    return datetime.datetime.fromtimestamp(seconds)
```

*Das ist die Anzahl der Sekunden zwischen einem bestimmten Datum und dem 1. Januar 1970 in UTC.

Datumsverarbeitung – Zeitpunkte und die Klasse `datetime`



- **`datetime`-Objekte bestehen bekanntlich aus**
 - Datum und
 - Uhrzeit
- **Bislang aber immer in Kombination, nun einzeln betrachten:**

```
sunday_afternoon = datetime.datetime(2021, 6, 13, 16, 52, 59)
print("date() =", sunday_afternoon.date())
print("time() =", sunday_afternoon.time())
```
- =>

```
date() = 2021-06-13
time() = 16:52:59
```
- **GLEICH MEHR DAZU ...**

Datumsverarbeitung – Zeitdifferenzen und die Klasse timedelta



- **Zeitdifferenzen ermitteln**

```
start = datetime.datetime(2021, 2, 7, 10, 10, 10)
end = datetime.datetime(2021, 2, 8, 11, 12, 13)
print("end - start =", end - start)
print(type(end - start))
```

- =>

```
end - start = 1 day, 1:02:03
<class 'datetime.timedelta'>
```

Datumsverarbeitung – Zeitdifferenzen und die Klasse timedelta



- **Zeitdifferenzen ermitteln**

```
td1 = datetime.timedelta(weeks=2, days=3, hours=5, seconds=6)
td2 = datetime.timedelta(days=7, hours=8, minutes=9, seconds=10)
print(td1 - td2)
print(td1 + td2)
```

```
michas_birthday = datetime.date(1971, 2, 7)
print(michas_birthday + td2 * 2)
```

- =>

```
9 days, 20:50:56
24 days, 13:09:16
1971-02-21
```

Datumsverarbeitung – Datumswerte und die Klasse date



- Die Klasse date besitzt eine today()-Methode. Um den Zugriff innerhalb etwas einfacher zu gestalten, erstellen wir eine gleichnamige Hilfsfunktion:

```
def today():
    return datetime.date.today()
```

- Wir können ein date-Objekt basierend auf Angaben zu Jahr, Monat, Tag erzeugen oder basierend auf einem ISO-String:

```
# date(year, month, day)
sophies_birthday = datetime.date(2020, 11, 23)

date_from_iso = datetime.date.fromisoformat('2020-11-23')
```

Datumsverarbeitung – Datumswerte und die Klasse date



- Ein Zeitpunkt besteht aus diversen Bestandteilen. Auf diese können wir per .-Notation und deren Namen zugreifen, etwa auf year, month und day:

```
date_from_iso = datetime.date.fromisoformat('2020-11-23')
print("Current year:", date_from_iso.year)
print("Current month:", date_from_iso.month)
print("Current day:", date_from_iso.day)
print("weekday():", date_from_iso.weekday())
print("isoweekday():", date_from_iso.isoweekday())
```

- =>

```
Current year: 2020
Current month: 11
Current day: 23
weekday(): 0
isoweekday(): 1
```



- Um den Wochentag zu einem Datum zu ermitteln, schreiben wir uns folgende Hilfsfunktion, die auf strftime() basiert

```
def get_week_day(date):  
    return date.strftime('%A')
```

- Tag im Monat

```
def day_of_month(date):  
    return date.day
```

- Tag im Jahr

```
def day_of_year(date):  
    return date.timetuple().tm_yday
```

```
def day_of_year_2(date):  
    return date.strftime('%j')
```



- **Länge eines Monats über `timedelta.days` bestimmen:**

```
print((datetime.date(2012, 3, 1) - datetime.date(2012, 2, 1)).days)
print((datetime.date(2014, 3, 1) - datetime.date(2014, 2, 1)).days)
```

- **Allgemeingültige Hilfsfunktion:**

```
def length_of_month(month, year):
    start = datetime.date(year, month, 1)
    month += 1
    if month > 12:
        year += 1
        month = 1

    end = datetime.date(year, month, 1)
    return (end - start).days
```



- Länge eines Jahres über `timedelta.days` bestimmen. Allgemeingültige Hilfsfunktion:

```
def length_of_year(year):  
    start = datetime.date(year, 1, 1)  
    end = datetime.date(year + 1, 1, 1)  
    return (end - start).days
```

- Schaltjahr bestimmen: Es ist wünschenswert, programmatisch ermitteln zu können, ob ein Jahr ein Schaltjahr ist. Dabei hilft uns das Modul calendar und die Methode `isleap()`.

```
import calendar  
  
print("2012 is leap?", calendar.isleap(2012))
```

Datumsverarbeitung – Datumswerte und die Klasse date



- **Datum aus Zeitstempel erzeugen**

```
date_from_ts = datetime.date.fromtimestamp(34774300)  
print(date_from_ts)
```

- => 1971-02-07

- **Manchmal ist es hilfreich, die Tage seit Christi Geburt fortlaufend als Zahlenwert zu zählen. `toordinal()` wandelt ein Datum in einen Zahlenwert, der der Anzahl an Tagen seit dem 1. Januar im Jahr 1 entspricht:**

```
christmas_eve_20 = datetime.date.fromisoformat('2020-12-24')  
print(christmas_eve_20.toordinal())
```

- => 737783
-



- Die Klasse `datetime` besitzt die Methoden `now()` und `time()`, Um den Zugriff auf die aktuelle Uhrzeit etwas einfacher zu gestalten, erstellen wir eine Hilfsfunktion:

```
def current_time():
    return datetime.datetime.now().time()
```

- Wir können ein `time` basierend auf Angaben zu Stunde, Minute und Sekunde erzeugen oder optional noch Millisekunde:

```
# time(hour, minute, second)
print(datetime.time(11, 28, 45))
```

```
# time(hour, minute, second, microsecond)
print(datetime.time(11, 28, 45, 6789))
```



DEMO

`datetime_intro.py`

Datumsverarbeitung – Formatierung und Parsing



- Die Art und Weise, wie Datum und Uhrzeit dargestellt werden, variiert von Land zu Land.
- Während man in Deutschland ein dd.mm.YYYY nutzt (mit den Platzhaltern d für Tage, m für Monate, Y für Jahre), so ist in den USA mm/dd/YYYY üblich, während in Großbritannien eher dd/mm/YYYY gebräuchlich ist.
- Python bietet zur einfacheren Handhabung die Methoden `strftime()` und `strptime()`.



- Die Methode `strftime()` ist für die Klassen `date`, `datetime` und `time` definiert. Die Methode erzeugt basierend auf einem String mit Platzhaltern einen formatierten String aus einem gegebenen `date`-, `datetime`- oder `time`-Objekt.
 - Die Methode `strptime()` erzeugt aus einer gegebenen Zeichenkette (die Datum und Uhrzeit darstellt) ein `datetime`-Objekt. Damit die Informationen geeignet extrahiert werden können, wird über einen String mit Platzhaltern der erwartete Aufbau der Eingabedaten spezifiziert.
-



- **Gebräuchliche und wesentliche Platzhalter sind folgende:**
 - %Y -- Jahr aus dem Bereich 1 bis 9999
 - %m -- Monat im Bereich [01, 02, ..., 11, 12]
 - %d -- Tag aus dem Bereich [01, 02, ..., 30, 31]
 - %H -- Stunde aus dem Bereich [00, 01, ..., 22, 23]
 - %M -- Minute aus dem Bereich [00, 01, ..., 58, 59]
 - %S -- Sekundenangabe aus dem Bereich [00, 01, ..., 58, 59]
 - %A -- Name des Tages
 - %B – Monatsname
- **ACHTUNG: Englische Namen!! Lokalisierung muss speziell programmiert werden!**

Datumsverarbeitung – Formatierung und Parsing



- **Beispiel**

```
>>> now_ = now()
>>>
>>> print(now_.strftime("%H:%M:%S"))
13:21:06
>>> print(now_.strftime("%d.%m.%Y, %H:%M:%S"))
13.06.2021, 13:21:06
>>>
>>> print("US:", now_.strftime("%m/%d/%Y, %H:%M:%S"))
US: 06/13/2021, 13:21:06
>>>
>>> print("GB:", now_.strftime("%d/%m/%Y, %H:%M:%S"))
GB: 13/06/2021, 13:21:06
```

- **Java bietet deutlich mehr Komfort und erfordert weniger Handarbeit für verschiedene Locales**
-

Datumsverarbeitung – Formatierung und Parsing



• Beispiel

```
>>> now_ = now() ←
>>>
>>> print(now_.strftime("%H:%M:%S"))
13:21:06
>>> print(now_.strftime("%d.%m.%Y, %H:%M:%S"))
13.06.2021, 13:21:06
>>>
>>> print("US:", now_.strftime("%m/%d/%Y, %H:%M:%S"))
US: 06/13/2021, 13:21:06
>>>
>>> print("GB:", now_.strftime("%d/%m/%Y, %H:%M:%S"))
GB: 13/06/2021, 13:21:06
```

```
now = now()
now2 = now()
object is not callable
```

- Java bietet deutlich mehr Komfort und erfordert weniger Handarbeit für verschiedene Locales

Datumsverarbeitung – Formatierung und Parsing



- **Beispiel**

```
date_string = "14 July, 1979"
```

```
date_object = datetime.datetime.strptime(date_string, "%d %B, %Y")
print(date_object)
```

- **Mit Locale**

```
import locale
```

```
date_string = "14. Juli 1979"
locale.setlocale(locale.LC_ALL, 'de_DE')
localized_date_object = datetime.datetime.strptime(date_string, "%d. %B %Y")
locale.setlocale(locale.LC_ALL, locale.getdefaultlocale())
print("Parsed with locale:", localized_date_object)
```

Datumsverarbeitung – Formatierung und Parsing



- Monatsnamen locale-spezifisch ermitteln:

```
import locale
```

```
def get_localized_month(month_nr, desired_locale):  
    locale.setlocale(locale.LC_ALL, desired_locale)  
    month_name = datetime.datetime.strptime(month_nr, "%m").strftime("%B")  
    locale.setlocale(locale.LC_ALL, locale.getdefaultlocale())  
    return month_name
```

```
for i in range(1, 13):  
    print(get_localized_month(str(i), 'de_DE'),  
          get_localized_month(str(i), 'fr_FR'),  
          get_localized_month(str(i), 'it_IT'))
```

Datumsverarbeitung – Formatierung und Parsing



- Monatsnamen locale-spezifisch ermitteln:

```
for i in range(1, 13):
    print(get_localized_month(str(i), 'de_DE'),
          get_localized_month(str(i), 'fr_FR'),
          get_localized_month(str(i), 'it_IT'))
```

Januar janvier Gennaio
Februar février Febbraio
März mars Marzo
April avril Aprile
Mai mai Maggio
Juni juin Giugno
Juli juillet Luglio
August août Agosto
September septembre Settembre
Oktober octobre Ottobre
November novembre Novembre
Dezember décembre Dicembre



- **Wochentagsnamen locale-spezifisch ermitteln:**

```
import locale
```

```
def get_localized_weekday(weekday_nr, desired_locale):  
    locale.setlocale(locale.LC_ALL, desired_locale)  
    weekday_name = datetime.datetime.strptime(str(weekday_nr), "%d").strftime("%A")  
    locale.setlocale(locale.LC_ALL, locale.getdefaultlocale())  
    return weekday_name
```

```
for i in range(1, 8):  
    print(get_localized_weekday(str(i), 'de_DE'),  
          get_localized_weekday(str(i), 'fr_FR'),  
          get_localized_weekday(str(i), 'it_IT'),  
          get_localized_weekday(str(i), 'es_ES'))
```



- Wochentagsnamen locale-spezifisch ermitteln:

```
for i in range(1, 8):  
    print(get_localized_weekday(str(i), 'de_DE'),  
          get_localized_weekday(str(i), 'fr_FR'),  
          get_localized_weekday(str(i), 'it_IT'),  
          get_localized_weekday(str(i), 'es_ES'))
```

Montag Lundi Lunedì lunes

Dienstag Mardi Martedì martes

Mittwoch Mercredi Mercoledì miércoles

Donnerstag Jeudi Giovedì jueves

Freitag Vendredi Venerdì viernes

Samstag Samedi Sabato sábado

Sonntag Dimanche Domenica domingo



DEMO

`calendar_printer.py`



Exercises Part 8

<https://github.com/Michaeli71/Python-Workshop.git>





Part 9

Einstieg in Lambdas & IterTools

(Recap Java Lambdas und Streams)



Lambdas: Einstieg



- **JAVA**

(Parameterliste) -> Anweisungen / Ausdruck

`n -> n * 2`

`person -> person.age() > 30`

- **PYTHON**

lambda Parameterliste: Ausdruck

lambda `n: n * 2`

lambda `person: person.age > 30`

Lambdas



- **Definition einfacher Lambdas in Python**

```
add_one = lambda x: x + 1
double_it = lambda x: x * 2
mult = lambda a, b : a * b
power_of = lambda x, y: x ** y
```

- **Aufruf wie normale Funktion / Methode:**

```
print(double_it(7))
print(power_of(2,8))
```

- =>

14
256

Lambdas im Einsatz (filter())



- **Lambda in Kombination mit Built-in filter()**

```
sample_numbers = [1, 5, 4, 6, 8, 11, 3, 12]  
is_even = lambda x: (x % 2 == 0)
```

```
only_even_numbers = list(filter(is_even, sample_numbers))  
print(only_even_numbers)
```

- =>

```
[4, 6, 8, 12]
```

- **ACHTUNG: Unterschied zu Java: Man kann das Ganze natürlich auch ein wenig komplexer machen, jedoch lassen sich keine if oder andere Anweisungen im Lambda nutzen – boolesche Verknüpfungen mit and und or sind dagegen möglich!**

Lambdas im Einsatz (map())



- **Lambda in Kombination mit Built-in map()**

```
sample_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
doubled_numbers = list(map(lambda x: x * 2, sample_numbers))  
print(doubled_numbers)
```

```
squared_numbers = list(map(lambda x: x * x, sample_numbers))  
print(squared_numbers)
```

- =>

```
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]  
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

map() im Einsatz



- Built-in map() und len()

```
continents = ["Afrika", "Asia", "South America", "North America",
              "Europe", "Australia", "Antarctica"]
result = map(len, continents)
print(list(result))
```

- =>

```
[6, 4, 13, 13, 6, 9, 10]
```

Lambdas im Einsatz Itertools (takewhile() / dropwhile())



- Weitergehende Funktionalitäten aus dem Modul **itertools**

```
import itertools
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
print(list(itertools.dropwhile(lambda x: x < 7, numbers)))
```

```
print(list(itertools.takewhile(lambda x: x < 7, numbers)))
```

- =>

```
[7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
[1, 2, 3, 4, 5, 6]
```



Vergleich mit Java-Stream-API



Datenmodell



- **JAVA**

```
jshell> record Person(String name, int age) {}  
| created record Person
```

- **PYTHON**

```
# Named Tuple vs Class  
from collections import namedtuple  
  
Person = namedtuple('Person', ['name', 'age'])
```

Ausgangsdatenbestand



- **JAVA**

```
jshell> var persons = List.of(new Person("Mike", 50), new Person("Tim", 50),
...>                           new Person("Tom", 7), new Person("Jim", 30))
persons ==> [Person[name=Mike, age=50], Person[name=Tim, age= ...
Person[name=Jim, age=30]]
```

- **PYTHON**

```
persons = [Person("Mike", 50), Person("Tim", 50),
Person("Tom", 7), Person("Jim", 30)]
```

Filtering



- **JAVA**

```
jshell> persons.stream().filter(person -> person.age() > 30).toList()
$7 ==> [Person[name=Mike, age=50], Person[name=Tim, age=50]]
```

- **PYTHON**

```
older_than_30 = list(filter(lambda person: person.age > 30, persons))
print(older_than_30)
[Person(name='Mike', age=50), Person(name='Tim', age=50)]
```

Mapping



- **JAVA**

```
jshell> persons.stream().map(person -> person.age() / 2).toList()
$8 ==> [25, 25, 3, 15]
```

- **PYTHON**

```
half_aged = list(map(lambda person: person.age // 2, persons))
print(half_aged)
[25, 25, 3, 15]
```



Während das Stream-API in Java seine Stärke ausspielt, sind `filter()`, `map()` usw. in Python nicht so stark. Einiges lässt sich allerdings mit *List Comprehensions* kürzer, eleganter und verständlicher lösen!

filter() / map() vs List Comprehension



- **Mit filter() / map()**

```
older_than_30 = list(filter(lambda person: person.age > 30, persons))
print(older_than_30)
```

```
half_aged = list(map(lambda person: person.age // 2, persons))
print(half_aged)
```

- **Mit List Comprehension**

```
older_than_30 = [person for person in persons if person.age > 30]
print(older_than_30)
```

```
half_aged = [person.age // 2 for person in persons]
print(half_aged)
```

- =>

```
[Person(name='Mike', age=50), Person(name='Tim', age=50)]
[25, 25, 3, 15]
```



DEMO

`filter_map_example.py`

Group by: Gruppieren von Werten



```
things = [("animal", "Bear"), ("sport", "Golf"), ("sport", "Karate"),
          ("sport", "Bowling"), ("vehicle", "Bicycle"), ("animal", "Dog"),
          ("vehicle", "Car"), ("animal", "Tiger")]
```

```
import itertools
```

```
for key, group in itertools.groupby(things, lambda entry: entry[0]):
    for entry in group:
        print("%s is a %s. (Key: %s)" % (entry[1], entry[0], key))
```

Group by: Aber oftmals unpraktisch



```
things = [("animal", "Bear"), ("sport", "Golf"), ("sport", "Karate"),
          ("sport", "Bowling"), ("vehicle", "Bicycle"), ("animal", "Dog"),
          ("vehicle", "Car"), ("animal", "Tiger")]

for key, group in itertools.groupby(things, lambda entry: entry[0]):
    for entry in group:
        print("%s is a %s. (Key: %s)" % (entry[1], entry[0], key))
```

```
Bear is a animal. (Key: animal)
Golf is a sport. (Key: sport)
Karate is a sport. (Key: sport)
Bowling is a sport. (Key: sport)
Bicycle is a vehicle. (Key: vehicle)
Dog is a animal. (Key: animal)
Car is a vehicle. (Key: vehicle)
Tiger is a animal. (Key: animal)
```

Spezialfall – Partitionierung

Manchmal hat man den Spezialfall, dass die Gruppierung durch ein Kriterium auf den Daten bestimmt wird, etwa dass ein bestimmter Teilstring enthalten ist, oder ein Wort mit einem gewissen Begriff startet, etwa mit »Obst«:

```
>>> words = ["Salami", "Obstkorb", "Käsekuchen", "Obstkuchen", "Obstsalat"]
```

Wir gruppieren dies wie folgt:

```
import itertools

result = {}
extract_category = lambda entry: entry.startswith("Obst")
for key, group in itertools.groupby(words, extract_category):
    members = result.get(key, []) + \
              list([entry for entry in group])
    result.update({key: members})

print(result)
```

Das liefert folgendes Ergebnis:

```
{False: ['Salami', 'Käsekuchen'], True: ['Obstkorb', 'Obstkuchen', 'Obstsalat']}
```

Analogien zu Java's groupingBy() / partitioningBy()



```
def grouping_by(values, key_extractor):
    result = {}
    sorted_by_key = sorted(values, key=key_extractor)
    for key, group in itertools.groupby(sorted_by_key, key_extractor):
        result[key] = list(group)
    return result
```

```
def partitioning_by(values, criterion):
    result = {}
    for key, group in itertools.groupby(values, criterion):
        members = result.get(key, []) + \
                  list([entry for entry in group])
        result.update({key: members})
    return result
```



DEMO

[`grouping_by1/2/3.py`](#) / [`partitioning_by.py`](#)



Vergleich mit Teeing-Kollektor

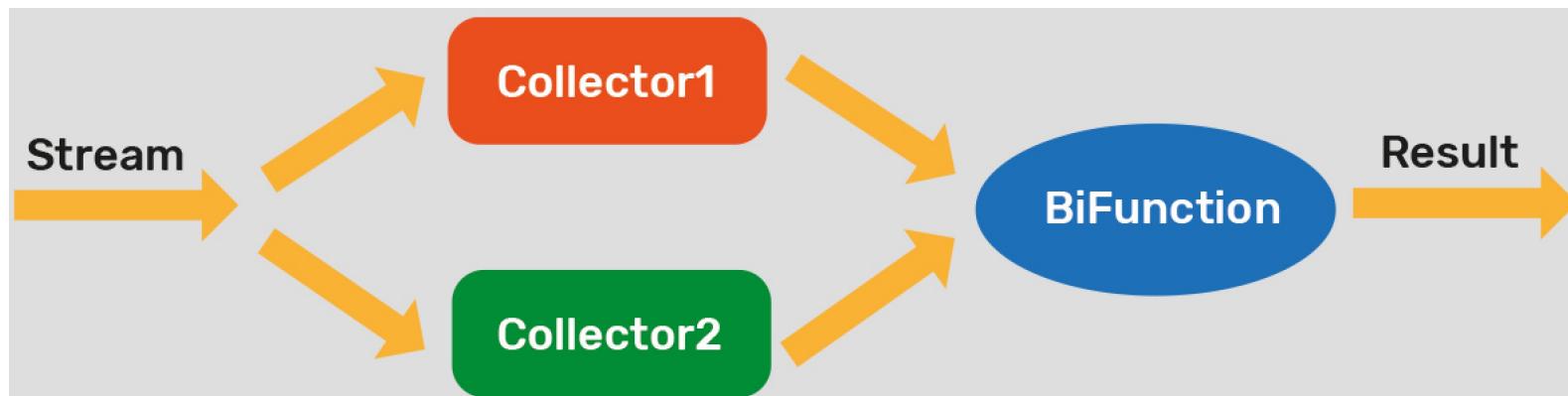




Wie sieht es denn mit dem Aufsplitten und dem Zusammenfassen von Streams aus?

"...returns a Collector that is a composite of two downstream collectors. Every element passed to the resulting collector is processed by both downstream collectors, then their results are merged using the specified merge function into the final result."

```
static <T, R1, R2, R> Collector<T, ?, R> teeing(Collector<? super T, ?, R1> downstream1,  
                                         Collector<? super T, ?, R2> downstream2,  
                                         BiFunction<? super R1, ? super R2, R> merger)
```



Teeing Collector Intro



```
var result = Stream.of(1,2,3,4,5,6,7).  
    collect(Collectors.teeing(  
        // first collector  
        Collectors.summingInt(n -> n),  
        // second collector  
        Collectors.counting(),  
        // merger - automatic type inference doesn't work here  
        (res1, res2) -> List.of(res1, res2)));  
  
System.out.println(result);
```

- =>

```
[28, 7]
```



Analogien zu Java's Teeing Collector Intro: tee()



```
import itertools

numbers = [1, 2, 3, 4, 5, 6, 7]

# aufspalten in n Iteratoren
it1, it2 = itertools.tee(numbers, 2)

sum_ = sum(it1)
count_ = sum(1 for _ in it2)

print((sum_, count_))

• =>

(28, 7)
```

Teeing Collector Intro



```
var result2 = Stream.of(1,2,3,4,5,6,7,8,9,10,11,12,13).  
    collect(Collectors.teeing(  
        // first collector  
        Collectors.filtering(n -> n % 2 != 0, Collectors.toList()),  
        // second collector  
        Collectors.averagingDouble(n -> n),  
        (res1, res2) -> List.of(res1, res2)));  
System.out.println(result2);
```

- =>

```
[[1, 3, 5, 7, 9, 11, 13], 7.0]
```

Analogien zu Java's Teeing Collector: tee()



```
import itertools
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

```
# aufspalten in n Iteratoren
```

```
it1, it2, it3 = itertools.tee(numbers, 3)
```

```
odd_numbers = list(filter(lambda n: n % 2 != 0, it1))
```

```
sum_ = sum(it2)
```

```
count_ = sum(1 for _ in it3)
```

```
print((odd_numbers, sum_ / count_))
```

- =>

```
([1, 3, 5, 7, 9, 11, 13], 7.0)
```

Analogien zu Java's Teeing Collector: tee()



```
import itertools
```

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]  
# aufspalten in n Iteratoren  
it1, it2, it3 = itertools.tee(numbers, 3)  
  
odd_numbers = [n for n in it1 if n % 2 != 0] # PYTHONIC !  
sum_ = sum(it2)  
count_ = sum(1 for _ in it3) ←
```

```
print((odd_numbers, sum_ / count_))
```

- =>

```
([1, 3, 5, 7, 9, 11, 13], 7.0)
```

Praxisbeispiel: Teeing-Kollektor



Aufgabe: Aus einem Stream von Strings, sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

- Hier hilft der `filtering()`-Kollektor aus Java 9 sowie `teeing()`:

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");
final BiFunction<List<String>, List<String>, List<List<String>>> combineLists =
    (list1, list2) -> List.of(list1, list2);

var result = names.collect(teeing(filtering(startsWithMi, toList()),
                                 filtering(endsWithM, toList()),
                                 // (list1, list2) -> List.of(list1, list2)
                                 combineLists));
System.out.println(result);
```

Praxisbeispiel: Teeing-Kollektor



Aufgabe: Aus einem Stream von Strings, sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");

var result = names.collect(teeing(filtering(startsWithMi, toList()),
    [Michael, Mike]
    filtering(endsWithM, toList())),
    (list1, list2) -> List.of(list1, list2));

System.out.println(result);
```

Praxisbeispiel: Teeing-Kollektor



Aufgabe: Aus einem Stream von Strings, sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");

var result = names.collect(teeing(filtering(startsWithMi, toList()),
    [Michael, Mike]
    filtering(endsWithM, toList())),
    [Tim, Tom]
    (list1, list2) -> List.of(list1, list2));
System.out.println(result);
```

Praxisbeispiel: Teeing-Kollektor



Aufgabe: Aus einem Stream von Strings, sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

```
var names = Stream.of("Michael", "Tim", "Tom", "Mike", "Bernd");

final Predicate<String> startsWithMi = text -> text.startsWith("Mi");
final Predicate<String> endsWithM = text -> text.endsWith("m");

var result = names.collect(teeing(filtering(startsWithMi, toList()),
    [Michael, Mike]
    filtering(endsWithM, toList()),
    [Tim, Tom]
    (list1, list2) -> List.of(list1, list2));

System.out.println(result);
    [[Michael, Mike], [Tim, Tom]]
```



Aufgabe: Aus einer Liste von Strings, sollen unterschiedliche Elemente herausgefiltert und dann die Ergebnisse kombiniert werden.

```
import itertools

names = ["Michael", "Tim", "Tom", "Mike", "Bernd"]

it1, it2 = itertools.tee(names, 2)

starts_with_mi = list(filter(lambda text: text.startswith("Mi"), it1)) ['Michael', 'Mike']

ends_with_m = [text for text in it2 if text.endswith("m")]

print([starts_with_mi, ends_with_m])
```



DEMO

[teeing_example1/2/3.py](#)



Exercises Part 9

<https://github.com/Michaeli71/Python-Workshop.git>





PART 99: BONUS





«Reflection»





- **Als Ausgangsbasis dient folgende Klasse mit ein paar Attributen**

```
class MyClass:  
    def __init__(self):  
        self.attr = 7271  
        self.info = "This is PUBLIC"  
        self.__private_info = "My birthday"  
  
    def get_private_info(self):  
        return self.__private_info  
  
obj = MyClass()
```

- **Ein Attribut ist «privat» und bietet nur lesenden Zugriff per Methode `get_private_info()`**

Möglichkeiten analog zu Java's Reflection



- Prüfe auf Existenz (hasattr())

```
print("hasattr()")  
print(hasattr(obj, "__init__"))  
print(hasattr(obj, "info"))  
print(hasattr(obj, "__private_info"))  
print(hasattr(obj, "__MyClass__private_info")) ←  
print(hasattr(obj, "unknown"))
```

- =>

```
hasattr()  
True  
True  
False  
True ←  
False
```

Möglichkeiten analog zu Java's Reflection



- Zugriff auf Werte mit Fallback (getattr())

```
print("getattr()")
print(getattr(obj, "__init__"))
print(getattr(obj, "attr"))
print(getattr(obj, "info"))
print(getattr(obj, "unknown", "FALLBACK"))
```

- =>

```
getattr()
<bound method MyClass.__init__ of <__main__.MyClass object at 0x101009fd0>>
7271
This is PUBLIC
FALLBACK
```

Möglichkeiten analog zu Java's Reflection



- Prüfung auf Ausführbarkeit (`callable()`)

```
print("callable()")
print(callable(getattr(obj, "__init__")))
print(callable(getattr(obj, "get_private_info")))
print(callable(getattr(obj, "attr")))
```

- =>

```
callable()
True
True
False
```

Möglichkeiten analog zu Java's Reflection



- **Modifikation von Werten (setattr())**

```
print("setattr()")  
setattr(obj, "attr", 1234)  
setattr(obj, "info", "PIN-CODE")  
setattr(obj, "__MyClass__private_info", "!!UNEXPECTED!!")  
print(obj.attr)  
print(obj.info)  
print(obj.__MyClass__private_info)
```

- =>

```
setattr()  
1234  
PIN-CODE  
!!UNEXPECTED!!
```



«HTTP-Support»





- Zugriff auf die Python Home Page

```
import requests
```

```
def print_response_info(address):
    response = requests.get(address)
    print(response)
    print(response.status_code)
    print(response.headers)
    print(response.text)
    print(response.content)
```

```
print_response_info("https://www.python.org/")
```



- **Daten als Datei sichern**

```
import requests
```

```
def print_response_info(address):  
    response = requests.get(address)  
    print(response.text)  
    print(response.content)
```

```
with open("../ch99_web_http/python.html", "w") as html_file:  
    html_file.write(response.text)
```

```
print_response_info("https://www.python.org/")
```



- Daten als Datei sichern

```
import requests
```

```
response = requests.get("https://imgs.xkcd.com/comics/modern_tools.png")
print(response)
print(response.status_code)

with open("../ch99_web_http/modern_tools.png", "wb") as png_file:
    png_file.write(response.content)
```



- Wetterdaten auslesen

```
import requests
```

```
# https://openweathermap.org/api
api_key = "74e5520ecf1361c1de73decb5cc4a9c9"
city = input("Bitte geben Sie den Städtenamen für die Wetterabfrage ein: ")

url = f"https://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}"

data = requests.get(url).json()
print(data)

temp = data["main"]["temp"]
print("Temperatur in °F:", temp)

humidity = data["main"]["humidity"]
print("Feuchtigkeit:", humidity)
```



DEMO

`second_requests.py`

`third_requests_post.py`

`image_download.py`

`weather_api.py`



«Testing mit PyTest»



Pytest Quick Start (RECAP)



- Für Tests sofort (nach Installation) einsetzbar (<https://docs.pytest.org/en/6.2.x/>)
- Testbehauptungen werden einfach mit dem Schlüsselwort assert formuliert

```
def test_answer():
    assert sum([1, 2, 3]) == 6
```

- Auf einfache Weise auch parametrierte Tests möglich

```
import pytest

@pytest.mark.parametrize("test_input,expected",
                      [("2 + 5", 7), ("2 ** 4", 16), ("6 * 9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Pytest Quick Start (RECAP)



- **Gute Integration in PyCharm**

```
import pytest

def test_answer():
    assert sum([1, 2, 3]) == 6
```

Test Results		0 ms
✗	test_ex02_repeat_chars	0 ms
✓	test_repeat_chars_ABCD	0 ms
✓	test_repeat_chars_ABCDEF	0 ms
✓	test_answer	0 ms
✗	test_eval	0 ms
✓	(2 + 5-7)	0 ms
✓	(2 ** 4-16)	0 ms
✗	(6 * 9-42)	0 ms

```
@pytest.mark.parametrize("test_input,expected",
                         [ ("2 + 5", 7), ("2 ** 4", 16), ("6 * 9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```



Primzahlen und das Sieb des Eratosthenes



Aufgabenstellung



Schreiben Sie eine Methode / Funktion `calcPrimesUpTo()` zur Berechnung aller Primzahlen bis zu einem vorgegebenen Wert. Zur Erinnerung: Eine Primzahl ist eine natürliche Zahl, die größer als 1 und ausschließlich durch sich selbst und durch 1 teilbar ist.

Eingabe	Resultat
15	[2, 3, 5, 7, 11, 13]
25	[2, 3, 5, 7, 11, 13, 17, 19, 23]
50	[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]



Sieb des Eratosthenes => Primzahlen bis zu einem Maximalwert bestimmen:

1. Initial werden alle Zahlen von zwei bis zu dem Maximalwert aufgeschrieben, etwa:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

2. Schrittweise diejenigen Zahlen gestrichen, die keine Primzahlen sein können.

1. Die kleinste unmarkierte Zahl, hier die 2, entspricht der ersten Primzahl. Nun streicht man alle Vielfachen davon. also im Beispiel 4. 6. 8. 10. 12. 14:

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15

2. Weiter geht es mit der Zahl 3. Das ist die zweite Primzahl. Nun werden wieder die Vielfachen gestrichen, nachfolgend 6, 9, 12, 15:

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15



```
public static List<Integer> calcPrimesUpTo(final int maxValue)
{
    final boolean[] isPotentiallyPrime = new boolean[maxValue + 1];
    Arrays.fill(isPotentiallyPrime, true);

    for (int i = 2; i <= maxValue / 2; i++)
        if (isPotentiallyPrime[i])
            eraseMultiplesOfCurrent(isPotentiallyPrime, i);

    return buildPrimesList(isPotentiallyPrime);
}
```



```
def calc_primes_up_to(max_value):
    is_potentially_prime = [True for _ in range(1, max_value + 2)]

    for number in range(2, int(max_value / 2) + 1):
        if is_potentially_prime[number]:
            erase_multiples_of_current(is_potentially_prime, number)

    return build_primes_list(is_potentially_prime)
```





```
void eraseMultiplesOfCurrent(boolean[]
                           isPotentiallyPrime, int i)
{
    for (int n = i + i;
         n < isPotentiallyPrime.length; n = n + i)
        isPotentiallyPrime[n] = false;
}

List<Integer> buildPrimesList(boolean[]
                           isPotentiallyPrime)
{
    List<Integer> primes = new ArrayList<>();
    for (int i = 2;
         i < isPotentiallyPrime.length; i++)
    {
        if (isPotentiallyPrime[i])
            primes.add(i);
    }
    return primes;
}
```

```
def erase_multiples_of_current(values, number):
    for n in range(number + number, \
                   len(values), number):
        values[n] = False

def build_primes_list(is_potentially_prime):
    return [number for number in range(2, \
                                       len(is_potentially_prime))
           if is_potentially_prime[number]]
```



```
@ParameterizedTest(name = "calcPrimes({0}) = {1}")
@MethodSource("argumentProvider")
void testCalcPrimesUpTo(int n, List<Integer> expected)
{
    List<Integer> result = calcPrimesUpTo(n);
    assertEquals(expected, result);
}

static Stream<Arguments> argumentProvider()
{
    return Stream.of(Arguments.of(2, List.of(2)),
                    Arguments.of(3, List.of(2, 3)),
                    Arguments.of(10, List.of(2, 3, 5, 7)),
                    Arguments.of(15, List.of(2, 3, 5, 7,
                                         11, 13)),
                    Arguments.of(25, List.of(2, 3, 5, 7,
                                         11, 13, 17,
                                         19, 23)));
}
```

```
def input_and_expected():
    return [(2, [2]),
            (3, [2, 3]),
            (10, [2, 3, 5, 7]),
            (15, [2, 3, 5, 7, 11, 13]),
            (25, [2, 3, 5, 7, 11, 13, 17, 19, 23])]

@pytest.mark.parametrize("n, expected",
                       input_and_expected())
def test_calc_primes_up_to(n, expected):
    assert calc_primes_up_to(n) == expected
```



Wohlgeformte Klammern



Aufgabenstellung



Schreiben Sie eine Methode / Funktion `checkBraces()`, die prüft, ob die als String gegebene Folge von runden Klammern jeweils passende (sauber geschachtelte) Klammerpaare enthält.

Eingabe	Resultat	Kommentar
"()"	True	
"()()	True	
"((()))(((())"	False	zwar gleich viele öffnende und schließende Klammern, aber nicht sauber geschachtelt
"(((("	False	keine passende Klammerung



- 1. Idee: Alle möglichen Kombinationen auszuprobieren => SCHLECHT
- 2. Idee: Zähle die Anzahl öffnender Klammern und vergleiche diese mit der Anzahl schließender Klammern
- Detail: Reihenfolge: Es darf keine schließende Klammer vor öffnender Klammer kommen!
- Algorithmus: Durchlaufe den String von vorne nach hinten. Ist das aktuelle Zeichen eine öffnende Klammer, so wird der Zähler für öffnende Klammer um eins erhöht. Ist es eine schließende Klammer, so reduziere den Zähler um den Wert eins. Ist dieser kleiner 0, wurde eine schließende Klammer ohne zugehörige öffnende Klammer gefunden. Zum Schluss muss die Anzahl gleich 0 sein, damit es einer sauberen Klammerung entspricht.



```
boolean checkBraces(final String input)
{
    int openingCount = 0;

    for (char ch : input.toCharArray())
    {
        if (ch == '(')
        {
            openingCount++;
        }
        else if (ch == ')')
        {
            openingCount--;
            if (openingCount < 0)
                return false;
        }
    }

    return openingCount == 0;
}
```

```
def check_braces(input):
    opening_count = 0

    for ch in input:
        if ch == "(":
            opening_count += 1
        elif ch == ")":
            opening_count -= 1
            if opening_count < 0:
                return False

    return opening_count == 0
```



```
@ParameterizedTest(name="checkBraces('{{0}}'') -- Hinweis: {2}" )
@CsvSource({ "(())", true, "ok",
             "()()", true, "ok",
             "(())((()))", false, "nicht sauber geschachtelt",
             "(()", false, "keine passende Klammerung",
             ")()", false, "startet mit schliessender Klammer" })
void checkBraces(String input, boolean expected, String hint)
{
    boolean result = Ex09_SimpleBracesChecker.checkBraces(input);

    assertEquals(expected, result);
}

@pytest.mark.parametrize("input, expected, hint",
                      [("(())", True, "ok"),
                       ("()()", True, "ok"),
                       ("(())((()))", False, "nicht sauber geschachtelt"),
                       ("(()", False, "keine passende Klammerung"),
                        ")()", False, "startet mit schliessender Klammer")])
def test_check_braces(input, expected, hint):
    assert check_braces(input) == expected
```





«Graphics»





DEMO

ch99_graphics



«Databases»





DEMO

`sql_lite_example.py`



«Python 3.10»



Fehlermeldungen bei Zuweisungen



- Nehmen wir zur Demonstration der Verbesserungen der Fehlermeldungen an, wir hätten in einem `if` versehentlich eine Zuweisung (`=`) statt eines Vergleichs (`==`) angegeben.

```
>>> if x = 6:  
    File "<stdin>", line 1  
        if x = 6:  
            ^  
SyntaxError: invalid syntax
```

- Python 3.10:

```
>>> if x = 6:  
    File "<stdin>", line 1  
        if x = 6:  
            ^^^^^^  
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

Fehlermeldungen bei unvollständigen Strings



- **Wir definieren ein Set mit einigen Namen, wobei jedoch der letzte nicht korrekt mit einem Anführungszeichen endet:**

```
>>> data = { "Tim", "Tom", "Mike}
      File "<stdin>", line 1
          data = { "Tim", "Tom", "Mike}
                      ^
SyntaxError: EOL while scanning string literal
```

- **Python 3.10:**

```
>>> data = { "Tim", "Tom", "Mike"
      File "<stdin>", line 1
          data = { "Tim", "Tom", "Mike"
                      ^
SyntaxError: unterminated string literal (detected at line 1)
```

Erweiterung bei zip()



- Python bietet eine Built-in-Funktion namens `zip()`, um zwei (bzw. genauer mehrere) Iterables zu einer Einheit zu verbinden, etwa zwei Listen, eine mit Programmiersprachen und eine andere mit Versionsnummern:

```
>>> languages = ['Java', 'Python']
>>> versions = [16.0, 3.9]
>>>
>>> print(list(zip(languages, versions)))
[('Java', 16.0), ('Python', 3.9)]
```

- Sofern ein Datenbestand mehr Elemente als der andere enthält, wird die Zusammenführung abgebrochen, sobald alle Elemente des kürzeren Datenbestands verarbeitet wurden:

```
>>> number_list = [1, 2, 3, 4, 5, 6]
>>> str_list = ['one', 'two', 'three']
>>>
>>> print(list(zip(number_list, str_list)))
[(1, 'one'), (2, 'two'), (3, 'three')]
```

Erweiterung bei `zip()`



- Seit Python 3.10 kann man an `zip()` noch den Parameter `strict` mit dem Wert `True` übergeben, um festzulegen, dass eine Exception ausgelöst wird, wenn eine der Iterables vor den anderen erschöpft ist.
- Python 3.10:

```
>>> number_list = [1, 2, 3, 4, 5, 6]
>>> str_list = ['one', 'two', 'three']
>>>
>>> print(list(zip(number_list, str_list, strict=True)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is shorter than argument 1
```

Verbesserungen bei Kontextmanager



- **Mit Python 3.10 wird ihre Syntax noch ein klein wenig angenehmer, wenn man mehrere Kontextmanager innerhalb einer with-Anweisung definieren möchte.**
- **Python 3.10:**

```
with (
    open("input1.txt") as input_file1,
    open("input2.txt") as input_file2,
):
```

Verbesserungen der Performance



- An diversen Stellen im Python wurden Dinge bezüglich der Performance verbessert.
 - Das betrifft vor allem diverse Konstruktoren wie `str()`, `bytes()` und `bytearray()`.
 - Diese sind um rund 30 % schneller geworden.
 - Details dazu finden Sie unter <https://bugs.python.org/issue41334>.
-

Fallunterscheidungen «switch» => match



- Python 3.9:

```
http_code = 201

if http_code == 200:
    print("OK")
elif http_code == 201:
    print("CREATED")
elif http_code == 404:
    print("NOT FOUND")
elif http_code == 418:
    print("I AM A TEAPOT")
else:
    print("UNMATCHED CODE")
```

Fallunterscheidungen «switch» => match



- Python 3.10:

```
http_code = 501
match http_code:
    case 200:
        print("OK")
    case 201:
        print("CREATED")
    case 404:
        print("NOT FOUND")
    case 418:
        print("I AM A TEAPOT")
    case _:
        print("UNMATCHED CODE")
```

Fallunterscheidungen «switch» => match



- Einfache Fallunterscheidungen, aber mit Fallback

```
def ide_full_name(short_name):  
    match short_name:  
        case "PyC":  
            return "PyCharm"  
        case "VSC":  
            return "Visual Studio Code"  
        case "ECL":  
            return "Eclipse"  
        case _:  
            return "Not Supported"
```

Fallunterscheidungen «switch» => match



- Kombination von Werten

```
def get_info(day):
    match day:
        case 'Monday':
            return "I don't like..."
        case 'Thursday' | 'Friday':
            return 'Nearly there!'
        case 'Saturday' | 'Sunday':
            return 'Weekend!!!'
        case _:
            return 'In Between...'
```

Fallunterscheidungen «switch» => match



- Komplexeres Matching

```
values = (2,3,4)
match values:
    case [1,2,3,4]:
        print("4 in a row")
    case [1,2,3] | [2,3,4]:
        print("3 in a row")
    case [1,2,4] | [1,3,4]:
        print("3 but not connected")
    case _:
        print("SINGLE OR DOUBLE")
```



DEMO

`python_3_10_1/_2/_3_zipping.py`



Weitere Infos

<https://rubikscode.net/2021/07/12/python-3-10-top-5-features-in-the-new-python-version/>



Fazit

Fazit



Java 

Python 

faster

easier

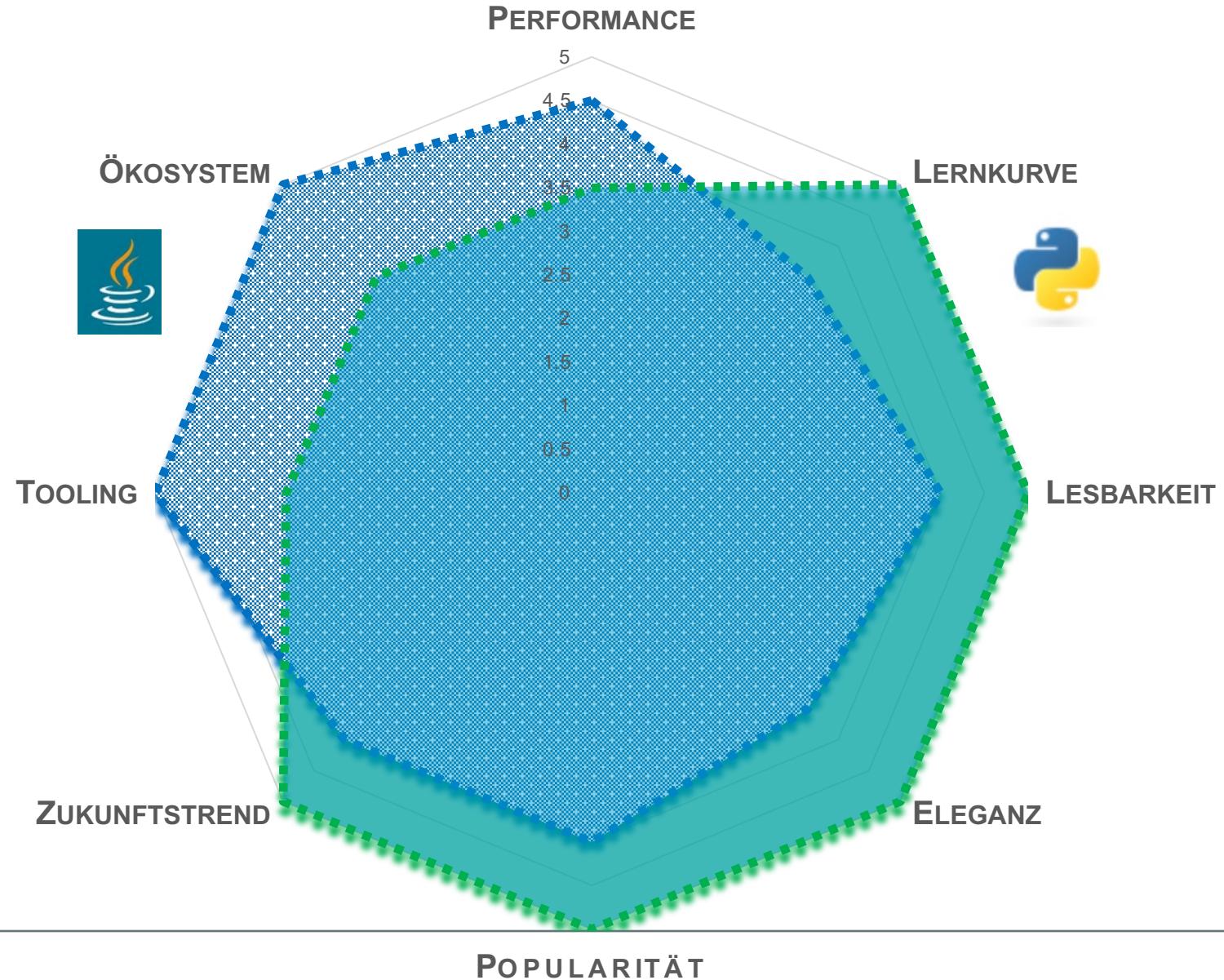
a bit verbose

short and concise

reliable

prone to careless mistakes

Vergleich Java - Python





Questions?

Help



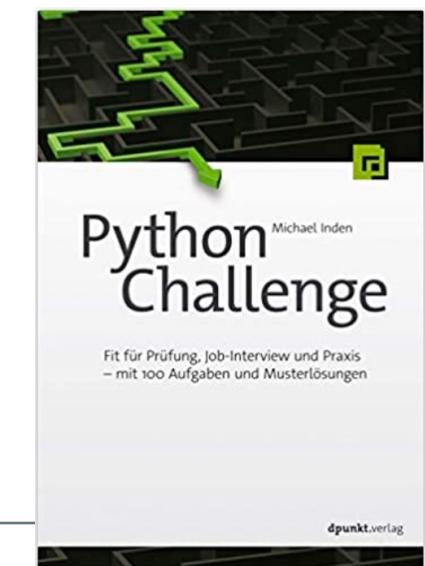
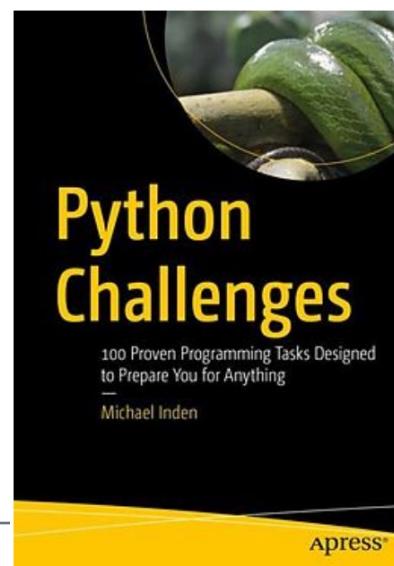
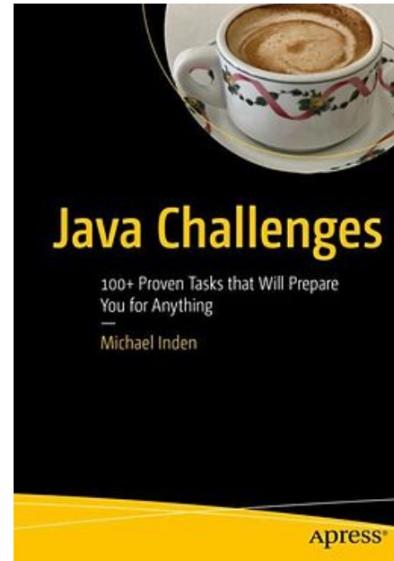
Einfach Python

Michael Inden

Gleich richtig
programmieren
lernen



dpunkt.verlag





Thank You