Python Basics

Besonderheiten: Einzelne Anweisungsblöcke werden nicht durch Semikolons und geschweifte/runde Klammern abgetrennt, sondern durch Einrückungen

Dadurch bleibt der Code sehr gut lesbar und übersichtlich.

Python Interpreter

Um Code in Python ausführen zu können, muss dieser an einen sogenannten "Interpreter" übergeben werden.

Dieser sorgt dafür, dass der Code von der Python-Umgebung verarbeitet und ausgeführt wird. Der Interpreter ist ein Programm, das der Nutzer auf seinem Rechner installieren muss, um Python-Programme ausführen zu können.

Der Code kann grundsätzlich auf zwei verschiedene Art und Weisen ausgeführt werden:

- Im interaktiven Modus

  In diesem Modus wird der Interpreter vom Nutzer geöffnet, wodurch der Anwender die Anweisungen direkt eingeben kann und die Befehle umgehend ausgeführt werden. Dass sich der Nutzer im interaktiven Python-Interpreter befindet, kann er daran erkennen, dass die Eingabezeile mit drei *größer-als*-Zeichen (`>>>`) beginnt.

  Auf Unix- und macOS-Systemen kann der interaktive Interpreter i.d.R. geöffnet werden, indem zunächst das Kommandozeilenprogramm ("Terminal") geöffnet und dort der Befehl `python3` eingegeben wird.

  Auf Windows-Systemen kann der Interpreter geöffnet werden, indem entweder das Programm "Python 3.x" (entspricht der installierten Version von Python) ausgeführt wird oder aber die Eingabeaufforderung geöffnet und dort der Befehl `python` ausgeführt wird. Kann der Befehl `python` nicht gefunden werden, muss der Installationspfad von Python zur Pfad-Umgebungsvariable hinzugefügt werden.

- Im ausführenden Modus

  Im ausführenden Modus wird dem `python`-Befehl ein Pfad zu einer Python-Datei angehangen, die den Python-Code enthält. Der Interpreter führt daraufhin alle darin enthaltenen Befehle nacheinander aus.

  Beispiel: `python C:\User\Jonas\Projects\MyPythonCode\test.py` (Windows) bzw. `python3 /Users/Jonas/Projects/MyPythonCode/test.py` (Unix/macOS)

Variablen

Variablen werden ohne Typbezeichnung deklariert (implizit) Jeder Wert (nicht Variable) besitzt einen Datentyp

Variablennamen dürfen aus Buchstaben, Zahlen und Underscores (_) bestehen. Jedoch dürfen sie nicht mit einer Zahl beginnen. Außerdem sind Variablennamen case-sensitive.

Zuweisung einer Ganzzahl (Integer):

```
x=5
type(x)
```

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

Zuweisung einer Fließkommazahl (Float):

```
y=3.14
type(y)
```

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

Zuweisung einer Zeichenkette (String)

```
z = "Hello, World"
type(z)
```

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

Zuweisung eines boolschen Wertes (bool)

```
nice_weather = True
type(nice_weather)
```

```
>> <class 'bool'>
```

Mathematische Operationen

Mit den Zahlenwerten (*int* und *float*) können in Python mathematische Operationen durchgeführt werden.

Hierzu stehen die Grundoperationen Addition (+), Subtraktion (-), Multiplikation (*) und Division (/) zur Verfügung.

```
print(2+3)
print(9/2)
```

```
>> 5
>> 4.5
```

Außerdem können Potenzen berechnet werden. Da der `^`-Operator bereits vergeben ist (XOR-Operationen), werden Potenzen mit `**` gebildet.

```
print(2**3)
```

```
>> 8
```

Darüber hinaus gehört auch der Modulo-Operator (%) zu den Standardoperationen. Mit einer Moduloberechnung kann der Rest einer Division ermittelt werden.

```
print(10%3)
```

```
>> 1
```

Boolsche Operationen und Vergleiche

Anders als in den meisten anderen Programmiersprachen werden für binäre Ausdrücke nicht die Symbole "&&", "||" bzw. "!" verwendet, sondern die entsprechenden englischsprachigen Wörter, um die gute Codelesbarkeit und Übersichtlichkeit zu gewährleisten.

```
x or y
x and y
not x
```

Die Auswertung Operationen mit `and` und `or` folgen den Wahrheitstabellen:

Wert 1	Wert 2	Ergebnis (AND)
1	1	1
1	0	0
0	1	0
0	0	0

Wert 1	Wert 2	Ergebnis (OR)
1	1	1
1	0	1
0	1	1
0	0	0

(Wert *1* entspricht dem Wert *True*, Wert *0* dem Wert *False*)

Für Vergleiche zwischen den Werten werden die gleichen Symbole genutzt, die auch in anderen Programmiersprachen zum Standard gehören.

z.B. `x < y` , `x <= y` , `x == y` , `x !=y`

Eine Besonderheit ergibt sich bei der Prüfung auf Gleichheit zweier Werte. Soll lediglich überprüft werden, ob zwei Werte übereinstimmen (Gleichheit), wird der `==` bzw. der `!=` Operator genutzt. Soll dagegen überprüft werden, ob es sich bei beiden Werten um das selbe Objekt handelt (Identität) wird das `is` bzw. `is not` Keyword genutzt.

```
x = list()
y = list()

print(x == y)
```

```
>> True
```

```
print(x is y)
```

```
>> False
```

Datenstrukturen

Neben einfachen Variablen gibt es auch höhere Datenstrukturen. Dazu gehören:

- Liste (Array): Veränderbare Sequenzen, in denen typischerweise homogene Werte gespeichert werden

  Erzeugung einer neuen (leeren) Liste: `colors = list()` oder `colors = []`

  Erzeugung einer vorgefüllten Liste: `colors = ['yellow', 'red', 'green', 'blue']`

  Zugriff durch Index (0-basiert):

  ```
  favourite_color = colors[2]
  print(favourite_color)
  ```

  ```
  >> 'green'
  ```

  `in` - Keyword:

  ```
  print('yellow' in colors)
  ```

  ```
  >> True
  ```

  ```
  print('black' in colors)
  ```

  ```
  >> False
  ```

- Tuple: Unveränderbare Sequenzen, in denen typischerweise heterogene Werte gespeichert werden

  Erzeugung eines neuen (leeren) Tuples: `fruits = tuple()` oder `fruits = ()`

  Erzeugung eines vorgefüllten Tuples:

  ```
  fruits = ('banana', 'strawberry', 'apple', 'coconut')
  ```

  Zugriff durch Index (0-basiert):

  ```
  favourite_fruit = fruits[0]
  print(favourite_fruit)
  ```

  ```
  >> 'banana'
  ```

- Dict: Stellt eine Sammlung von *key-value-Paaren* dar. Mappt beliebige Werte auf hashbare Objekte

  Erzeugung eines neuen (leeren) Dict: `politicans = dict()` oder `politicans = {}`

  Erzeugung eines vorgefüllten Dicts:

  ```
  politicans = {'Angela Merkel'= 'CDU', 'Horst Seehofer' = 'CSU', 'Christian
  Lindner' = 'FDP', 'Olaf Scholz'= 'SPD'}
  ```

  Zugriff durch Key:

  ```
  ruling_party = politicans['Angela Merkel']
  ```

```
print(ruling_party)
```

```
>> 'CDU'
```

- Set: Unsortierte Sammlung von unterschiedlichen Objekten

  Erzeugung eines neuen (leeren) Sets: `politicans = set()`

  Erzeugung eines vorgefüllten Sets:

  ```
  politicans = {'Angela Merkel', 'Horst Seehofer', 'Christian Lindner', 'Olaf Scholz', 'Angela Merkel'}
  print(politicans)
  ```

  ```
  >> {'Angela Merkel', 'Horst Seehofer', 'Christian Lindner', 'Olaf Scholz'}
  ```

  Da *set* eine unsortierte Sammlung von Objekten ist, kann auf die einzelnen Elemente nicht durch einen Index zugegriffen werden

Built-in Funktionen

Python bietet eine Palette an Standard-Funktionen, die Teil des Interpreters sind und ohne zusätzlichen Import genutzt werden können.

- Konvertierungsfunktionen wie bin(), bool(), complex(), float(), int(), str(), …
- Input / Output:
  - print(): Ausgabe von Objekten in die Standardausgabe
  - input(): Einlesen von Eingaben
  - open(): Öffnen einer Datei
- Mathematische Grundfunktionen wie max(), min(), round(), sum(), abs(), pow(), …
- Hilfsfunktionen für Iterables:
  - len(): Größe/Länge eines Iterables (Liste, Tuple, Dict, …) `len(['a', 'b', 'c'])` `>> 3`
  - range(): Repräsentiert eine unveränderbare Sequenz von Zahlen. Wird meistens für Loops genutzt `range(5)` : Bildet eine Range / einen Bereich von 0 - 4 ab `range(2,7)` : Bildet eine Range von 2 - 6 ab `range(1,9,2)` : Bildet eine Range von 1 - 8 in Zweierschritten ab.
  - sorted(), reversed(), …
- viele weitere (=> siehe Docs)

Steueranweisungen

- Verzweigungen / Conditions: `if - else` Konstrukte Es wird anhand von Bedingungen (boolschen Operationen) festgelegt, was ausgeführt werden soll

```
if 'blue' in colors:
        print("Blue is already in the list")
else:
        colors.append('blue')
```

Sollen mehrere Bedingungen untersucht werden, kann zusätzlich `elif` (else if) genutzt werden

```
if condition1 is True:
        # executed if condition1 is True
elif condition2 is True:
        # executed if condition1 is False and condition2 is True
else:
        # executed if condition1 is False and condition2 is True
```

In einer Verzweigung sind sowohl `elif` als auch `else` optional.

- Loops / Schleifen:
  - `for`-Loop: Iteriert über jedes der gegebenen Objekte einer Sequenz Dafür können alle Iterables genutzt werden (*Liste*, *Tuple*, *Range*, …)

    Syntax:

```
colors = ['blue', 'green', 'red', 'yellow']
for color in colors:
        print(color)
```

```
>> 'blue'
>> 'green'
>> 'red'
>> 'yellow'
```

```
for index in range(1,9,2):
        print(index)
```

```
>> 1
>> 3
>> 5
>> 7
```

  - `while`-Loop: Anweisungen in der Schleife werden wiederholt, solange der Bedingung wahr ist

    Syntax:

```
sum = 0
i = 0

while i<10:
        sum += i
        i+=1

print(sum)
```

```
>> 45
```

Funktionen

Um wiederkehrende Funktionalität auszulagern und an anderer Stelle im Code erneut zu verwenden (DRY-Prinzip, "Don’t repeat yourself"), können die Anweisungen in Funktionen ausgelagert werden.

Die Funktionen werden mit dem `def`-Keyword deklariert, anschließend folgt der Name der Funktion sowie die Liste der Parameter, die der Funktion übergeben werden, in runden Klammern.

Die Funktionen können einen oder mehrere Werte mit dem `return` Keyword zurückgeben. Wird kein expliziter Rückgabewert definiert, wird standardmäßig der Wert `None` zurückgegeben.

```
def sum (x, y):
        result = x + y
        return result
```

Die Funktion kann ausgeführt werden, indem sie durch den Funktionsnamen sowie den benötigten Parametern aufgerufen wird.

```
calculated_sum = sum(3,6)
print(calculated_sum)
```

```
>> 9
```

Named arguments: Bezeichner können bei Funktionsaufruf mit angegeben werden:
`calculated_sum = sum(x=3, y=6)`

Optionale Argumente: Parameter können Standardwerte annehmen, falls sie nicht bei Funktionsaufruf explizit mit angegeben werden.

```
def sum (x, y=2):
        return x+y

calculated_sum = sum(3)
print(calculated_sum)
```

```
>> 5
```

Module

Analog zu den Funktionen, in denen wiederkehrende Funktionalitäten zusammengefasst und abstrahiert werden, können Funktionen, Klassen etc. in dedizierte Dateien zusammengefasst werden, die *Module* genannt werden. Funktionalitäten in diesen Modulen können an anderen Stellen genutzt werden, indem sie *importiert* werden.

Module können wiederum Submodule enthalten.

Es stehen bereits einige hilfreiche Module in der Python-Standardbibliothek bereit, darunter beispielsweise:

- *math*: Mathematische Funktionen wie `math.floor()`, `math.log()`, `math.tan()`, …
- *time*: Funktionen zur Verarbeitung von Zeiten, Datumsangaben u.ä. wie `time.time()`, `time.sleep()`, …
- *os*: Modul für plattformunabhängige Funktionalitäten wie `os.getlogin()`, `os.mkdir()`
- *sqlite3*: Funktionen zum Zugriff auf *SQLite*-Datenbanken wie `sqlite3.connect()`, `sqlite3.close()`, …
- *threading* und *multiprocessing* : Funktionalitäten zum (Multi-)Threading und (Multi-)Processing wie `threading.main_thread()`, `multiprocessing.Process.run()`
- *json*: Modul für JSON-Funktionalitäten wie `json.load()`, `json.dump()`
- *http*: Modul, das mehrere Untermodule für Funktionalitäten bezogen auf HTTP enthält wie `http.client`, `http.server`, …
- viele weitere (siehe Docs)

Klassen und Objekte (Objektorientierte Programmierung)

Klassen dienen dazu, zusammengehörige Daten und Funktionalitäten zu bündeln. Eine Klasse stellt dabei sozusagen eine *Blaupause* dar, mit der beschrieben wird, über welche Eigenschaften und Funktionen jedes Objekt dieser Klasse verfügt.

Ein *Objekt* dagegen stellt eine Instanz dieser Klasse dar.

Oder anders ausgedrückt: Eine Klasse beschreibt, welche Eigenschaften, Variablen und Funktionalitäten jedes der aus der *Blaupause* dieser Klasse erzeugten Objekte besitzt.

Die zugrundeliegende Idee soll an folgendem Beispiel veranschaulicht werden: Folgende Liste beschreibt die Eigenschaften von verschiedenen Personen:

- Person1 Vorname: Sebastian
- Person1 Nachname: Thiemt
- Person1 Alter: 31
- Person1 Wohnort: Esslingen
- Person1 Geschlecht: männlich
- Person2 Vorname: Jonas
- Person2 Nachname: Miederer
- Person2 Alter: 25
- Person2 Wohnort: Stuttgart
- Person2 Geschlecht: männlich
- Person3 Vorname: Dieter
- Person3 Nachname: Zetsche
- Person3 Alter: 66
- Person3 Wohnort: Stuttgart
- Person3 Geschlecht: männlich

Die Liste enthält zwar alle wichtigen Informationen zu den beiden Personen, jedoch ist sie sehr unübersichtlich. Folgendes Format ist besser geeignet:

- Person1:
  - Vorname: Sebastian
  - Nachname: Thiemt
  - Alter: 31
  - Wohnort: Esslingen
  - Geschlecht: männlich
- Person2:
  - Vorname: Jonas
  - Nachname: Miederer
  - Alter: 25
  - Wohnort: Stuttgart
  - Geschlecht: männlich
- Person3:
  - Vorname: Dieter
  - Nachname: Zetsche
  - Alter: 66
  - Wohnort: Stuttgart

- Geschlecht: männlich

Diese Liste enthält die gleichen Informationen, ist jedoch einfacher zu handhaben, da die entsprechenden Daten zu den Personen jeweils nach Person gruppiert sind. Die 1. Liste stellt sogesehen einfach eine Menge von Variablen dar, die die Personen beschreiben, jedoch in keinster Weise geordnet sind. Die 2. Liste gibt den Informationen eine Struktur. Eine Klasse beschreibt also, welche Informationen jede Instanz besitzt, während eine Instanz die jeweiligen Daten beschreibt.

Mit einer Klasse ausgedrückt kann eine Personen-Klasse also folgendermaßen aussehen:

```
class Person:
        def __init__(self, first_name, last_name, age, city):
                self.fist_name = first_name
        self.last_name = last_name
                self.age = age
            self.city = city

        def greet(self):
                print("Hallo {}!".format(self.first_name))
```

Eine Klasse wird mit dem Schlüsselwort `class` definiert, gefolgt vom Namen der Klasse. Anschließend folgt im Beispiel die Spezialfunktion `__init__()`. Da dies eine Standardfunktion ist, die automatisch Bestandteil jeder Klasse ist, wird dies durch die `__` verdeutlicht. Diese Funktion wird automatisch ausgeführt, sobald ein Objekt dieser Klasse erzeugt wird ("Konstruktor"). Diese Methode erwartet 3 Argumente (`self`, `name` und `age`). `name` und `age` sind selbsterklärend, doch wofür steht das `self`-Argument? Jedes Objekt, das nach Vorbild der Person-Klasse erzeugt wird, besitzt zwar die gleichen Funktionen und Variablen wie jedes andere Person-Objekt, jedoch ist die Belegung der Werte zwischen den Objekten unterschiedlich (Person A hat einen anderen Namen und ein anderes Alter als Person B). Daher kann mit `self` das jeweilige Objekt referenziert werden, das auf die Daten zugreift. `self` wird also bei jeder Funktion innerhalb einer Klasse automatisch als erster Parameter übergeben. Nun können in der `__init__`-Funktion dem jeweiligen Objekt (`self`) der Name sowie das Alter zugewiesen werden.

Außerdem beinhaltet die Klasse eine weitere Funktion, die `greet()`-Funktion. Da auch diese Funktion teil der Person-Klasse ist, wird auch dieser Funktion das `self`-Objekt als erster Parameter übergeben. Beim Aufruf dieser Funktion wird die jeweilige Person anhand ihres Namens gegrüßt. Hier wird auch der Zweck von `self` ersichtlich: Es soll der Name der Person desjenigen Objekts ausgegeben werden, mit der die Funktion aufgerufen wurde.

Ist eine Funktion Bestandteil einer Klasse, wie oben die `__init__()`- sowie die `greet()`-Funktionen, werden diese nicht als "Funktionen", sondern als *"Methoden"* bezeichnet.

Nun wurde das Grundgerüst einer Person erstellt, das festlegt, aus welchen Eigenschaften (`name` und `age`) sowie aus welchen Funktionen (`greet()`) jede Person besteht, jedoch wurde noch keine explizite Person erstellt. Hierfür wird ein Objekt dieser Person erzeugt:

```
person = Person("Jonas", "Miederer", 26, "Stuttgart")
```

Im Beispiel wurde also eine Person mit dem Namen "Jonas" und dem Alter 25 erzeugt und in der Variablen `person` gespeichert. Das `self`-Objekt muss nicht mit übergeben werden, da es sich automatisch auf das jeweilige Objekt bezieht.

Nun kann die Person mit der `greet()`-Methode gegrüßt werden:

```
person.greet()
```

```
>> "Hallo, Jonas!"
```

Die Eigenschaften des Objekts können einfach durch einen simplen Zugriff ausgelesen werden:

```
print(person.first_name)
print(person.age)
```

```
>> "Jonas"
>> 26
```

Vererbung

Zum Konzept der objektorientierten Programmierung gehört auch die Möglichkeit der Vererbung von Klassen. Wenn eine Klasse von einer anderen *erbt*, so übernimmt sie automatisch alle Instanzvariablen und Methoden der *"Parent"-Klasse*. Dies trägt zur Wiederverwendung von Klassen bei, da auf diese Weise Komponenten in den Kindklassen genutzt werden, ohne diese erneut definieren/implementieren zu müssen. Außerdem ist Vererbung hilfreich, wenn in einer Klasse bereits Methoden implementiert sind, jedoch eine oder mehrere dieser Methoden in der Funktionalität abgewandelt werden soll. In diesem Fall kann die jeweilige Methode in der Kindklasse *überschrieben* werden, während alle anderen Methoden von der Elternklasse übernommen werden.

Beispiel: Jeder Student ist auch eine Person, jedoch nicht jede Person ein Student. Daher besitzt jeder Student die gleichen Eigenschaften (Variablen) sowie Fähigkeiten (Methoden) wie eine Person. Die Klasse `Student` kann also von der Klasse `Person` erben:

```
class Student(Person):
  def __init__(self, first_name, last_name, age, city, courses):
    self.courses = courses
    super().__init__(first_name, last_name, age, city)

  def study(self):
    import random
    print("{} is studying {}.".format(self.first_name, random.choice(self.courses)))
```

Indem der Name der Elternklasse in Klammern hinter dem eigentlichen Namen der Klasse angegeben wird, kann von dieser Klasse geerbet werden. Hier wird die `__init__`-Methode der Elternklasse *überschrieben*, da diese in der Kindklasse neu definiert und implementiert ist. Zusätzlich zu den Attributen, die eine Person beschreiben, wird hier ein weiteres Attribut `courses` übergeben. Dieses wird als Instanzattribut der *Student*-Klasse gespeichert. Alle anderen Attribute werden an den Konstruktor der Eltern- bzw. Superklasse übergeben, der sich um die weitere Verarbeitung kümmert. Somit kann der Aufwand durch Vererbung also minimiert werden. Zusätzlich besitzt die *Student*-Klasse eine Methode *study()*, die einen zufälligen Kurs ausgibt.

Zusammenfassend besitzt die *Studentklasse* also 5 Attribute (*first_name*, *last_name*, *age*, *city*, *courses*) sowie 2 Methoden (*study()* und die geerbte *greet()*-Methode).

```
student = Student("Jonas", "Miederer", 25, "Stuttgart", ["Computer Science", "Project
Management", "Programming" ])

student.study()
student.greet()
```

```
>> "Jonas is studying Programming."
>> "Hallo, Jonas!"
```

Paketverwaltung

Mithilfe von Python können eigene *"Pakete"* erzeugt werden, also zusammengehörige Module. Diese können beispielsweise auch von anderen Entwicklern installiert und anschließend genutzt werden. Um einen einfachen Austausch von Paketen und Workflow zur Installation zu gewährleisten wird das Paketverwaltungsprogramm pip genutzt, das als rekursives Akronym für "pip installs packages" steht.

Mithilfe von pip können auch somit auch Pakete installiert werden, die von fremden Entwicklern bereitgestellt wurden. Als zentraler Paketpool steht dafür der *Python Package Index (PyPI)* bereit, auf dem jeder seine Python-Packages zum Download anbieten kann.

Die Installation kann einfach mit dem Befehl

```
pip install <package_name>
```

ausgeführt werden. Ist die Installation erfolgreich kann das Paket anschließend durch einen Import genutzt werden.

Aufgaben

Input & Output

1. Schreibe ein Programm, bei dem der Nutzer zwei Zahlen eingeben kann, anschließend wird dem Nutzer die Summe dieser Zahlen ausgegeben.
2. Schreibe ein Programm, bei dem der Nutzer seinen Namen eingeben kann, anschließend soll der Nutzer mit seinem Namen gegrüßt werden (z.B. Eingabe "Jonas", Ausgabe "Guten Tag, Jonas")
3. Schreibe ein Programm, bei dem der Nutzer eine Zahl im Bogenmaß eingeben kann, anschließend wird dem Nutzer der entsprechende Winkel im Gradmaß ausgegeben.
4. Schreibe ein Programm, durch das quadratische Gleichungen der Form $ax^2 + bx + c = 0$ gelöst werden können. Der Nutzer kann die entsprechenden Koeffizienten eingeben.

Steueranweisungen

5. Schreibe ein Programm, bei dem der Nutzer zwei Ganzzahlen eingeben kann. Berechne den Rest der Division beider Zahlen, ohne den Modulo-Operator zu nutzen.
6. Schreibe ein Programm, bei dem der Nutzer eine Ganzzahl eingeben kann, anschließend soll folgende Gleichung mithilfe einer Schleife berechnet und das Ergebnis wieder ausgegeben werden.

$$\prod_{i=1}^{k} i^{\frac{1}{i+0.5}} + \log_e i - \cos(2i)$$

7. Schreibe eine Funktion, die eine Zahl als Argument entgegennimmt und die Quersumme dieser Zahl zurückliefert.
8. Schreibe eine Funktion, die 2 Argumente entgegennimmt und überprüft, ob der erste Parameter durch den zweiten Parameter teilbar ist. Der Rückgabewert ist dementsprechend ein boolscher Wert (True / False). Teste diese Funktion, indem der Nutzer zwei Zahlen eingeben kann und durch den Aufruf der Funktion überprüft wird, ob diese Zahlen teilbar sind. Falls die erste Zahl durch die zweite Zahl teilbar ist, erfolgt die Ausgabe "Die Zahlen sind teilbar", andernfalls "Die Zahlen sind nicht teilbar".
9. Schreibe ein Programm, bei dem die Zahlen von 1 bis 100 ausgegeben werden sollen. Allerdings soll bei jeder Zahl, die durch 5 teilbar ist, das Wort "fizz" ausgegeben werden. Bei jeder Zahl, die durch 7 teilbar ist, soll das Wort "buzz" ausgegeben werden. Bei Zahlen, die sowohl durch 5 als auch durch 7 teilbar sind, erfolgt die Ausgabe "fizzbuzz".
10. Schreibe ein Programm, bei dem über die Zahlen von 1 bis 100 iteriert wird. Erzeuge ein Dictionary mit den den Ziffern von 1 bis 9 als Keys. Falls die jeweilige Zahl des Loops durch 1 teilbar ist, speichere diese Zahl in einer Liste unter dem Key "1". Falls die Zahl durch 2 teilbar ist, speichere diese Zahl in einer Liste unter dem Key "2" usw.

Zeichenkettenverarbeitung

11. Schreibe ein Programm, bei dem der Nutzer einen Satz eingeben kann, anschließend soll dieser Satz in umgekehrter Reihenfolge ausgegeben werden. ("Hallo ich bin Jonas" => "Jonas bin ich Hallo")
12. Schreibe ein Programm, bei dem der Nutzer ein Wort / einen Satz eingeben kann, anschließend soll ein Histogramm aller darin enthaltenen Buchstaben erstellt werden, d.h. für jeden Buchstaben soll die Anzahl, wie häufig er vorkommt, ausgegeben werden.

Objektorientierung

13. Erstelle eine Klasse "Auto", die ein Auto beschreibt. Die Klasse besitzt mehrere Instanzvariablen deiner Wahl, die

bei der Erzeugung eines Auto-Objekts übergeben werden sollen.

14. Außerdem besitzt die Klasse eine Instanzvariable "speed", die die Geschwindigkeit des Fahrzeugs beschreibt. Diese Instanzvariable wird nicht bei der Erzeugung eines Auto-Objekts übergeben, da der Wert dieser Variable zu Beginn standardmäßig 0 ist.
15. Die Auto-Klasse besitzt mehrere Methoden, darunter die `accelerate()` sowie die `stop()`-Methode. Wird die `accelerate()`-Methode aufgerufen, so soll die Geschwindigkeit (=> `speed`-Variable) des Fahrzeugs auf 100 (km/h) gesetzt werden, durch die `stop()`-Methode dagegen auf 0.
16. Implementiere weitere Methoden, die in ähnlicher Weise ein Fahrzeug beschreiben (z.B. `get_speed()`-Methode, die die aktuelle Geschwindigkeit ausgibt ("Das Fahrzeug fährt gerade X km/h")).
17. Erstelle ein Objekt dieser Auto-Klasse und rufe mit dieser die zuvor erstellten Methoden auf.