

Tipps zur Bearbeitung der Übungsblätter

Zu Beginn die Aufgabenstellung auf dem Übungsblatt aufmerksam lesen und sich somit erst einmal einen Überblick verschaffen.

Um eine Aufgabe gründlich zu lösen, ist es generell hilfreich möglichst viel Kontext mit einzubeziehen. Das heißt Vorlesungsfolien, vorhandenen Code und Kommentare sowie alles Hilfreiche, das schnell mit einer Suchmaschine recherchiert werden kann.

Auch wenn es zu Beginn länger dauert und es Zeit in Anspruch nimmt den vorhanden Code zu analysieren, wird man erstens einen größeren Lernerfolg haben sowie zweitens höchst wahrscheinlich später weniger Zeit damit verbringen einen bestimmten Teil des Programms zu vervollständigen (der Code-Teil, der Punkte gibt), wenn man eine ungefähre Vorstellung davon hat, was der Rest des Programms macht.

Ich stehe vor einer leeren oder fast leeren Methode oder Funktion, und nun?

Es kommt selten vor, dass man bei größeren Code-Blöcken den kompletten Code bereits im Kopf hat und diesen nur noch aufschreiben muss.

Wenn man noch nicht genau weiß, was alles in einem Code-Block gemacht werden soll, dann kann es hilfreich sein, sich langsam heranzutasten, statt zu versuchen alles auf einmal zu lösen.

Ist die Aufgabenstellung soweit klar? Zumindest Teile der Aufgabenstellung mit denen angefangen werden kann. Während der Bearbeitung kann noch mehr Verständnis für den restlichen Teil der Aufgabe hinzukommen, wenn der Anfang erst mal konkretisiert ist.

Die Funktions-/Methodenargumente unbedingt beachten und damit vertraut machen (Inhalt/Typ). Gibt es einen Rückgabewert, wenn ja wie sieht er aus/soll er aussehen?

- Alle relevanten Funktions-/Methodenaufrufe inspizieren
- print()-Befehl nutzen um Variableninhalt auszugeben
- Zeit nehmen um die Unittests nachzuvollziehen (Argumente, Aufrufe, Daten)

Für Programmieranfänger könnte es hilfreich sein, sich den Code wie ein Seil vorzustellen, an dem ihr euch entlang hangeln solltet. Voraussetzung um effektiv programmieren zu können ist immer zu wissen, was **vor** der aktuellen Betrachteten Code-Stelle passiert und was **danach** passiert/passieren soll.

Zentrale Fragen sind hier:

Wo wird die Methode, die ich schreiben soll überall aufgerufen?

Welche Argumente (Inhalt/Typ) werden übergeben?

Wie wird mein Rückgabewert benötigt und gegebenenfalls weiterverarbeitet?

Wenn das grobe Ziel der Aufgabenstellung klar ist, kann zunächst versucht werden mit natürlicher Sprache zu programmieren. Wichtige Elemente aus der Aufgabenstellung schon mal prototypisch in den Code als Kommentare aufnehmen.

So könnte man in Kommentaren bereits Elemente der Lösung vermerken, die man schon kennt (auch wenn es noch weit von einer programmatischen Lösung entfernt ist).

Die jeweiligen Kommentare/Anweisungen werden dann Schritt für Schritt in Code umgesetzt, damit man immer einen strukturierten Plan hat, der verfolgt werden kann und sich nicht alles im Kopf gemerkt werden muss.

Die Kommentare möglichst kurz und präzise halten, bevor man nur noch endlosen Fließtext sieht.

Zum Beispiel in der Form:

„Aufgabe: Bestimmen Sie das Sentiment der Sätze und füllen Sie damit das Dictionary“

```
# loop through dictionary, inspect data
# use given data
# maybe call class method? Which?
# calculate formula
# prepare data for return statement
```

Wenn Code umgesetzt wurde, die entsprechenden Hilfskommentare löschen, damit sie nicht verwirren und es übersichtlicher wird oder die Hilfskommentare in "echte" Kommentare umwandeln.

Die Kommentare können als unfertiger Code angesehen werden und es gilt diese Kommentare aktuell zu halten und im Laufe der Bearbeitung zu präzisieren.

```
for key in tweets: # Dict with tweet data # changed lines
    print(key)
# maybe call class method with given data? # changed line
# calculate formula
# prepare data for return statement
```

Sinnvolle und aussagekräftige Variablennamen verwenden. (**Spätestens** wenn ein Code Block abgeschlossen ist und funktioniert.)

```
for tweet in tweets: # Dict with tweet data # changed line
    # Get sentiment from tweet # changed lines
    sentiment_score = get_sentiment_score(tweet)
    print(sentiment_score) # TODO remove # check output
```

```
# calculate formula, score is positive, negative or zero      # changed line
# save sentiment in dict                                     # new comment
# prepare data for return statement
```

Die Kommentare stets aktuell halten, Variablennamen anpassen, und Inhalte printen. Den Code möglichst übersichtlich halten und entfernen, was nicht mehr gebraucht wird.

Auch kann es hilfreich sein Berechnungen/Transformationen/etc mit einem Toy-Beispiel – falls möglich - auf Papier selbst durchzuführen.

```
for tweet in tweets: # Dict with tweet data

    # Get sentiment from tweet
    sentiment_score = get_sentiment_score(tweet)

    sentiment = "positive" # Sentiment score > 0           # changed lines
    if sentiment_score == 0:
        sentiment = "neutral"
    elif sentiment_score < 0:
        sentiment = "negative"

    print(sentiment_score, sentiment) # TODO remove       # check output

# save sentiment from tweet in dict as value               # changed line
# prepare data for return statement
```

Zwischendurch immer wieder die Ausgabe kontrollieren:

```
def get_sentiment_score(tweet):
    # Simple demo score calculation
    vocab_scores = {"awesome": 10, "shit": -8, "John": 0}
    return vocab_scores[tweet.split()[-1]]

tweets = {"This is awesome": "", "This is shit": "", "My name is John": ""}

for tweet in tweets: # Dict with tweet data

    # Get sentiment from tweet
    sentiment_score = get_sentiment_score(tweet)

    sentiment = "positive" # Sentiment score > 0
    if sentiment_score == 0:
        sentiment = "neutral"
    elif sentiment_score < 0:
        sentiment = "negative"

    print(sentiment_score, sentiment) # TODO remove

# save sentiment from tweet in dict as value
# prepare data for return statement, create function     # changed lines
# print function results
```

Konsolenausgabe:

```
10 positive
0 neutral
-8 negative
Process finished with exit code 0
```

Kontinuierlich Kommentare abarbeiten und neue aufkommende Probleme/Lösungsschritte dokumentieren.

```
def get_sentiment_score(tweet):
    # Simple demo score calculation
    vocab_scores = {"awesome": 10, "shit": -8, "John": 0}
    return vocab_scores[tweet.split()[-1]]

tweets = {"This is awesome": "", "This is shit": "", "My name is John": ""}

def get_sentiment_bulk(tweets): # changed lines
    # TODO: Add function description
    for tweet in tweets: # Dict with tweet data

        # Get sentiment from tweet
        sentiment_score = get_sentiment_score(tweet)

        sentiment = "positive" # Sentiment score > 0
        if sentiment_score == 0:
            sentiment = "neutral"
        elif sentiment_score < 0:
            sentiment = "negative"

        # Save sentiments # changed lines
        tweets[tweet] = sentiment

    return tweets

print(get_sentiment_bulk(tweets))
```

Konsolenausgabe:

```
{'This is awesome': 'positive', 'My name is John': 'neutral', 'This is shit': 'negative'}
Process finished with exit code 0
```

Mit das Wichtigste beim Programmieren ist zu wissen, für was eine Variable steht und vor allem um welche Datenstruktur es sich handelt und welche Daten genau in dieser Variable gespeichert sind.

Ohne Kenntnis um den Inhalt und die Bedeutung einer Variable wird es extrem schwer produktiven Code zu verfassen. (Es ist ein wenig so, als würde man versuchen ein Buch weiterzuschreiben, ohne die bereits vorhandenen Seiten lesen zu dürfen und sich nicht darüber bewusst zu sein, was auf den vorherigen Seiten stand. Ja, es klappt durchaus, aber es wird höchstwahrscheinlich eine ganz andere Geschichte, die nicht mehr viel mit der angefangen Story zu tun hat, auch wenn sie trotzdem [auf den ersten Blick] gut und erfolgreich sein mag.)

Also, wenn man nicht sicher ist was eine Variable repräsentieren soll oder beinhaltet, dann unbedingt BEVOR etwas anderes gemacht wird, sicher stellen, dass man den Zweck und den Inhalt einer Variable kennt.

Entweder durch vorangegangenen Code sich den Zweck und Inhalt erschließen, Kommentare lesen in denen Variablen beschrieben werden oder sich die entsprechende Variable über das Terminal ausgeben lassen.

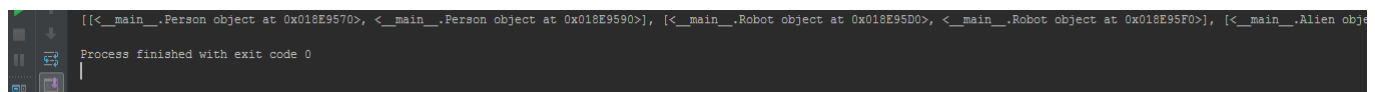
Eine Beispielsaufgabe „Lebensformen“ zur Verdeutlichung:

```
def name_life_forms(data):  
    """  
    Nenne die Namen aller Lebensformen aus den gesammelten Daten.  
    """
```

Da wir nur diese Information haben, bleibt uns nichts anders übrig, als die Variable *data* zunächst einmal auszugeben: (oder den Funktionsaufruf inspizieren)

```
def name_life_forms(data):  
    """  
    Nenne die Namen aller Lebensformen aus den gesammelten Daten.  
    """  
  
    print(data)
```

Die Konsolenausgabe zeigt folgendes:



```
[[<__main__.Person object at 0x018E9570>, <__main__.Person object at 0x018E9590>], ... ]
```

Die äußersten eckigen Klammern zeigen uns, dass es sich um eine Liste handelt, von der – sofern es nicht zu viel Daten sind - jedes Element (sichtbar) in der Konsole aufgelistet wird. Das erste Element ist ebenfalls eine Liste, die wiederum zwei Objekte enthält (ersichtlich an den inneren eckigen Klammern). In der inneren Liste sehen wir schon, dass die einzelnen Elemente in der Liste Objekte der Klasse „Person“ sind.

Grob gibt es 3 verschiedene „visuelle“ Arten von Variablen, die man sehen wird, wenn man in Python Variablen printet.

- Primitive Datentypen wie Strings, Zahlen und Booleans
- Iterables (Lists, Set, Dictionary, ... [Strings gehören hier auch dazu, da sie iterierbar sind])
- Komplexere Objekte (die beliebig definiert sein können)

Es gibt natürlich noch Tuples und weitere Datentypen. (Die 5 Standard Datentypen in Python sind Strings, Zahlen, Listen, Dictionaries und Tuple)

Im Folgenden ist erklärender Beispielcode zu sehen mit der dazugehörigen Konsolenausgabe:

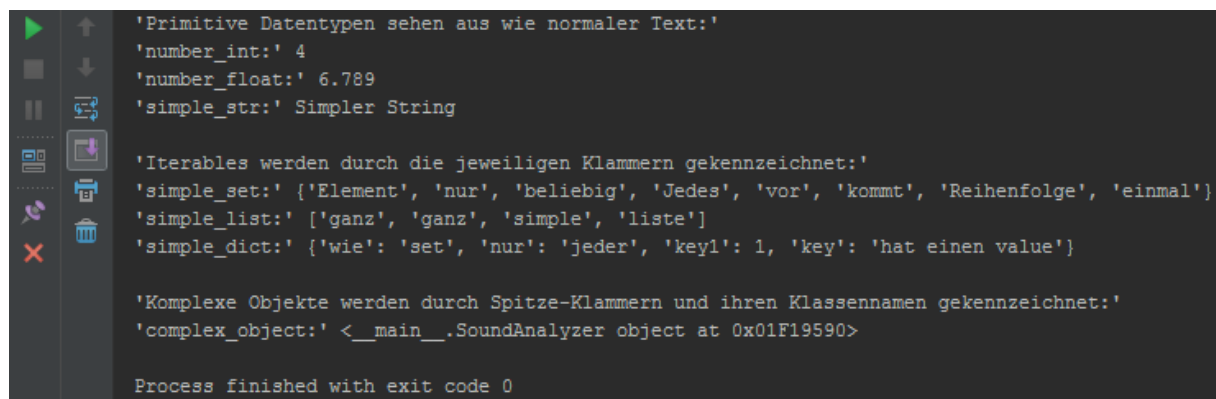
```
class SoundAnalyzer:
    pass

# Primitive Datentypen
print("'Primitive Datentypen sehen aus wie normaler Text:')")
number_int = 4
print("'number_int:'", number_int)
number_float = 6.789
print("'number float:'", number float)

simple_str = "Simpler String"
print("'simple_str:'", simple_str)

# Iterables
print("\n'Iterables werden durch die jeweiligen Klammern gekennzeichnet:')")
simple_set = {"Jedes", "Element", "kommt", "nur", "einmal", "vor", "Reihenfolge", "beliebig"}
print("'simple_set:'", simple_set)
simple_list = ["ganz", "ganz", "simple", "liste"]
print("'simple_list:'", simple_list)
simple_dict = {"key1": 1, "wie": "set", "nur": "jeder", "key": "hat einen value"}
print("'simple dict:'", simple dict)

# Komplexe Objekte
print("\n'Komplexe Objekte werden durch Spitze-Klammern & den Klassennamen gekennzeichnet:')")
complex_object = SoundAnalyzer()
print("'complex object:'", complex object)
```



```
'Primitive Datentypen sehen aus wie normaler Text:'
'number_int:' 4
'number_float:' 6.789
'simple_str:' Simpler String

'Iterables werden durch die jeweiligen Klammern gekennzeichnet:'
'simple_set:' {'Element', 'nur', 'beliebig', 'Jedes', 'vor', 'kommt', 'Reihenfolge', 'einmal'}
'simple_list:' ['ganz', 'ganz', 'simple', 'liste']
'simple_dict:' {'wie': 'set', 'nur': 'jeder', 'key1': 1, 'key': 'hat einen value'}

'Komplexe Objekte werden durch Spitze-Klammern und ihren Klassennamen gekennzeichnet:'
'complex_object:' <_main__.SoundAnalyzer object at 0x01F19590>

Process finished with exit code 0
```

Der print-Befehl bei Objekten gibt den Rückgabewert der `__str__`-Methode, falls vorhanden, aus:

```
class SoundAnalyzer:
    def __str__(self):
        return "Hilfreiche Beschreibung oder Ausgabe von Klassenvariablen"

complex_object = SoundAnalyzer()
print("'complex_object:', complex_object)
```

```
'complex_object:' Hilfreiche Beschreibung oder Ausgabe von Klassenvariablen
Process finished with exit code 0
```

Zurück zur „Lebensformen“-Aufgabe. Laufen wir nun durch die Listen-Variable `data` hindurch und geben uns die einzelnen Listen aus, damit wir eine übersichtlichere Konsolenausgabe erhalten.

```
def name_life_forms(data):
    """
    Nenne die Namen aller Lebensformen aus den gesammelten Daten.
    """

    for x in data:
        print(x)
```

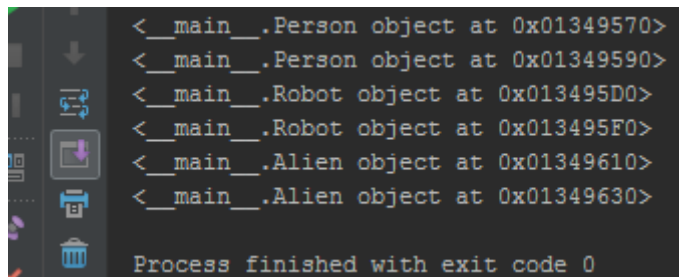
```
[<__main__.Person object at 0x01409570>, <__main__.Person object at 0x01409590>]
[<__main__.Robot object at 0x014095D0>, <__main__.Robot object at 0x014095F0>]
[<__main__.Alien object at 0x01409610>, <__main__.Alien object at 0x01409630>]
Process finished with exit code 0
```

In der aktuellen Konsolenausgabe sehen wir drei Listen (gekennzeichnet durch die eckigen Klammern) mit jeweils zwei Objekten (der Klassen `Person`, `Robot` und `Alien`).

Für unser späteres Ziel können wir die Ausgabe schon mal übersichtlicher gestalten:

```
def name_life_forms(data):  
    """  
    Nenne die Namen aller Lebensformen aus den gesammelten Daten.  
    """  
  
    # Liste mit 3 Listen von Lebensformen (Personen, Roboter und Aliens)  
    for life_form_list in data:  
        for life_form in life_form_list:  
            print(life_form)
```

Die dazugehörige Konsolenausgabe sieht entsprechend aus:



```
<_main_.Person object at 0x01349570>  
<_main_.Person object at 0x01349590>  
<_main_.Robot object at 0x013495D0>  
<_main_.Robot object at 0x013495F0>  
<_main_.Alien object at 0x01349610>  
<_main_.Alien object at 0x01349630>  
Process finished with exit code 0
```

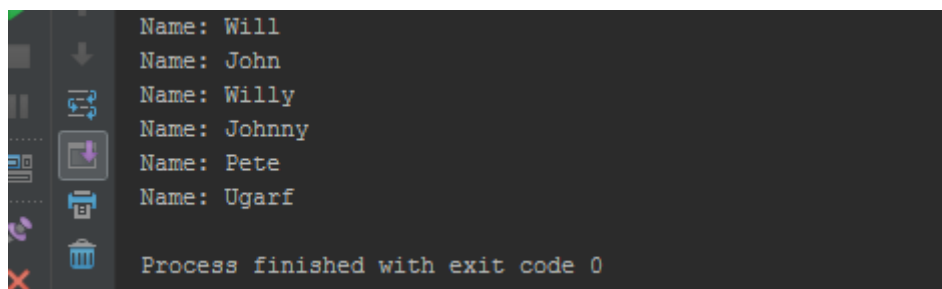
Unsere Vorgabe ist es nun die Namen aller Lebensformen auszugeben. Wir haben keine Möglichkeit zu wissen, wie wir das machen können ohne in die Klassendefinition reinschauen – außer Raten natürlich.

[Abgesehen davon, dass in Entwicklungsumgebungen Vorschläge (Attribute/Methoden) über den Punktoperator angezeigt werden.]

Höchstwahrscheinlich wird jede Klasse eine *name* Variable haben.

```
def name_life_forms(data):  
    """  
    Nenne die Namen aller Lebensformen aus den gesammelten Daten.  
    """  
  
    # Liste mit 3 Listen von Lebensformen (Personen, Roboter und Aliens)  
    for life_form_list in data:  
        for life_form in life_form_list:  
            print("Name:", life_form.name) # Line changed
```

Wie man sieht hat es (glücklicherweise) geklappt:



```
Name: Will  
Name: John  
Name: Willy  
Name: Johnny  
Name: Pete  
Name: Ugarf  
Process finished with exit code 0
```


Wenn wir jetzt jedoch spezifische Attribute der einzelnen Klassen/Lebensformen-Repräsentationen auslesen wollen, dann sollten wir unbedingt in die jeweilige Klassendefinition schauen:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Robot:
    def __init__(self, name, build_lang):
        self.name = name
        self.build_lang = build_lang

class Alien:
    def __init__(self, name, home_planet):
        self.name = name
        self.home_planet = home_planet
```

Nun kennen wir die Klassendefinition und können dieses Wissen in unsere Schleife mit einbauen:

```
def name_life_forms(data):
    """
    Nenne die Namen aller Lebensformen aus den gesammelten Daten.
    """

    # Liste mit 3 Listen von Lebensformen (Personen, Roboter und Aliens)
    for life_form_list in data:
        for life_form in life_form_list:
            print("\nName:", life_form.name)

            if isinstance(life_form, Person): # Klassenüberprüfung
                print("Age:", life_form.age)
            if isinstance(life_form, Robot):
                print("Build Lang:", life_form.build_lang)
            if isinstance(life_form, Alien):
                print("Home Planet:", life_form.home_planet)
```

Die dazugehörige Konsolenausgabe sieht wie folgt aus:

```
Name: Will
Age: 23

Name: John
Age: 26

Name: Willy
Build Lang: Python

Name: Johnny
Build Lang: Q#

Name: Pete
Home Planet: QZ3Y

Name: Ugarf
Home Planet: Mars

Process finished with exit code 0
```

Und diese Aufgabe ist damit gelöst :)

Generell sollte unbedingt immer darauf geachtet werden, was von einer Funktion oder Methode verlangt wird.

Selten ist es die Aufgabe, innerhalb einer Funktion Ergebnisse direkt zu printen.

Es ist häufiger der Fall, dass Ergebnisse zurückgeliefert werden sollen:

```
def get_life_form_names(data):  
    """  
    Gibt eine Liste mit den Namen aller übergebener Lebensformen zurück.  
    """  
  
    # Liste mit 3 Listen von Lebensformen (Personen, Roboter und Aliens)  
    names = [life_form.name for life_form_list in data for life_form in life_form_list]  
  
    return names  
  
print("Lebensformnamen:", get_life_form_names(collected_data))
```

Konsolenausgabe:

```
Lebensformnamen: ['Will', 'John', 'Willy', 'Johnny', 'Pete', 'Ugarf']  
  
Process finished with exit code 0
```

Man erkennt (bzw. sollte) bereits am Namen der Funktion oder Methode erkennen, ob es einen Rückgabewert gibt. Wörter wie *get*, *extract*, *calculate*, ... , welche suggerieren, dass nach der Funktion ein Ergebnis erwartet wird, weisen üblicherweise auf die Notwendigkeit eines *return*-Statements hin.

Nomen und Eigennamen im Funktions-/Methodennamen weisen ebenfalls auf einen Rückgabewert hin. Bei allem das eine Transformation, Berechnungen oder ähnliches suggeriert ist normalerweise ein Rückgabewert zu erwarten.

Übrigens, bei komplexeren Codestellen (mit vorhandenem Konsolenoutput) oder auch während Unittest-Läufen empfiehlt es sich nie nur die Variable zu printen, sondern stattdessen noch zusätzlich einen individuellen, hilfreichen und hervor stechenden Text auszugeben. Sonst könnte es passieren, dass eine Variable leer ist oder nur einen Wert beinhaltet und in anderen Ausgaben (z.B. Unittest-Feedback) untergeht oder der Output mit etwas anderem verwechselt wird.

Zum Beispiel in der Form:

```
for word_pair, inverse, mapping in word_pairs_data:  
    print("#### word_pair", word_pair)  
    print("#### inverse", inverse)  
    print("#### mapping type", type(mapping))
```

Nochmal alles auf einen Blick:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

class Robot:
    def __init__(self, name, build_lang):
        self.name = name
        self.build_lang = build_lang

class Alien:
    def __init__(self, name, home_planet):
        self.name = name
        self.home_planet = home_planet

persons = [Person("Will", 23), Person("John", 26)]
robots = [Robot("Willy", "Python"), Robot("Johnny", "Q#")]
aliens = [Alien("Pete", "QZ3Y"), Alien("Ugarf", "Mars")]

collected_data = [persons, robots, aliens]

def get_life_form_names(data):
    """
    Gibt eine Liste mit den Namen aller übergebener Lebensformen zurück.
    """
    # Liste mit 3 Listen von Lebensformen (Personen, Roboter und Aliens)
    names = [life_form.name for life_form_list in data
              for life_form in life_form_list]

    return names

print("Lebensformnamen:", get_life_form_names(collected_data))
```

Konsolenausgabe:

```
Lebensformnamen: ['Will', 'John', 'Willy', 'Johnny', 'Pete', 'Ugarf']
Process finished with exit code 0
```

Es werden andauernd Fehler in der Konsole angezeigt und die Tests schlagen fehl, was nun?

So doof, das klingt, zunächst einmal sind Fehler hilfreich. Man lernt aus Fehlern und sollte daher künftig Zeitersparnisse bei der Lösung eines ähnlichen Problems haben oder dieses sogar direkt vermeiden können. Sicherlich sind Fehler jedoch ohne Aussicht auf Lösung sehr ärgerlich.

Auf Syntax-Fehler achten! Hier kann eine Entwicklungsumgebung wie PyCharm sehr hilfreich sein, da diese den Programmierer auf entsprechende Fehler aufmerksam macht. Das Programmieren in einem normalen Editor fördert und trainiert sicherlich auch den sicheren Umgang mit einer Programmiersprache, jedoch ist eine Entwicklungsumgebung während produktiver Arbeit sehr empfehlenswert.

Fehlermeldung richtig lesen und auf die Zeilenangaben in der Fehlermeldung achten.

Sind Klammerungen korrekt gesetzt? Die Programmier-Syntax erlaubt eine Fülle von für die aktuelle Aufgabe semantisch nutzlosen Klammerungskombinationen.

Sind Variablen leer? Verwendung von `len()`, `print()`

Deckt sich das mentale Modell mit den tatsächlichen Datenstrukturen?

Datentypen der Variablen überprüfen mit `type()`, `print()`

Testen, testen, testen! Wenn man noch am Lernen einer Programmiersprache ist, dann sollte man unbedingt ein Programm nicht erst am Ende komplett durchlaufen lassen, wenn man meint, dass alles fertig ist und funktionieren sollte. Ein/e Aufgabe/Problem in möglichst viele kleine Schritte zerlegen ist hier sehr hilfreich und vor allem jeden Schritt sofort überprüfen, ob das gewünschte Ergebnis eintritt. Das heißt Zwischenergebnisse printen und Schlüsselvariablen überprüfen (printen!). Nur wenn sichergestellt ist, dass die Variablen den Inhalt haben, von dem man ausgeht und den sie haben sollten, macht es Sinn, den nächsten Schritt in Angriff zu nehmen.

Das Problem ist sonst, dass man am Ende einen langen Code hat und eine Fehlermeldung nach der anderen bekommt und gar nicht mehr eingeschätzt werden kann, welche Teile von dem Code überhaupt stimmen. Eine Code-Zeile baut auf der anderen auf. Ausschließen zu können, welche Code-Zeile für einen konzeptionellen Fehler verantwortlich ist, kann den Unterschied von Sekunden/Minuten zu Stunden der Fehlersuche machen.

Der Code ist zu komplex zum printen oder testen, was nun?

Nehmen wir zum Beispiel list-Comprehensions:

```
result = [retrieve(x[:3].run_stripping()) for x in clean(preprocess(v))]
```

Der Fehler liegt offensichtlich in einer komplexen Zeile mit vielen Aufrufen.

Den Trace in der Fehlermeldung immer genau verfolgen.

Den Einzeiler auf mehrere Zeilen aufteilen und Aufrufe einzeln ausprobieren. Einen großen Datensatz/Liste/etc. reduzieren auf einige wenige Test-Daten um die Aufrufe einzeln zu testen. Wenn nötig/möglich, in einer separaten Python-Datei oder Variablen unmittelbar vor einem Aufruf clearen oder für eine simplere Testumgebung sorgen (die Datenstruktur jedoch unbedingt beibehalten, da es sonst nur noch zusätzlich zu vermeintlichen Fehlern führen kann):

```
v = {(["Info1", "Infos2"], 1): 5600} # TODO remove Test-Override
result = [retrieve(x[:3].run_strip()) for x in clean(preprocess(v))]
```

Shortcuts der Entwicklungsumgebung nutzen (PyCharm):

Strg/Ctrl+B Jump to Declaration (oder Rechtsklick auf die Code-Stelle -> Go To -> Declaration)

Strg/Ctrl+/ Un/Comment selection

...

Grundlagen:

Auch wenn es schwer fällt und zunächst sehr zeitaufwendig scheint: Je mehr Zeit für das Meistern der Grundlagen investiert wird, desto weniger Zeit wird später für komplexere Aufgaben benötigt.

Was sind Klassen, was bedeutet *self*?

Was ist ein Dictionary, Keys und Values, und wie erhält man die Values zu bekannten Keys?

Wie funktioniert ein Funktions-/Methodenaufruf? Was ist ein Argument?

Wo in/außerhalb einer Schleife setzt man Variablen zurück?

...

Für interaktives Lernen und Programmieren in PyCharm:

<https://www.jetbrains.com/pycharm-edu/learners/>

https://www.python-kurs.eu/python3_kurs.php

Das Wichtigste: Spaß und mit kleinen und großen Erfolgen immer am Ball bleiben!



Viel Spaß beim Programmieren!