Matthias Mruzek-Vering, M.Sc.

Hochschule Karlsruhe
University of
Applied Sciences

DSCB 230 - Aufgabenblatt 6

Die Aufgabenblätter werden in dieser Veranstaltung in Jupyter Notebooks veröffentlicht und bearbeitet. Diese finden Sie in der Github Organisation für Data Science 2 unter dem Repository *dscb230-tutorial* (https://github.com/hka-mmv/dscb230-tutorial). Die Musterlösung wird ebenfalls in Form eines Jupyter Notebook in Github hochgeladen.

Python3 Update

In diesem Aufgabenblatt wird sich zum Teil mit **Python Version 3.10** beschäftigt. Bevor damit angefangen werden kann, müsst ihr *ggf.* euer System updaten. Hierfür gibt es im Jupyter Notebook eine Anleitung.

Aufgabenteil 1

In dieser Aufgabe behandeln wir die Neuerungen in Python 3.10.

PEP 634: Structural Pattern Matching

Um verschiedene Fälle zu unterscheiden sind euch bereits if-else Statements bekannt. In Hochsprachen wie Java, C/C++, uvm. gibt es neben dem if-else Statements noch eine weitere Variante, die Match-Case / Switch-Case Statements. Der Kompilier in Hochsprachen bzw. Interpreter von Python3 weiß bei if-else Statements die Reihenfolge der Auswertung nicht und kann somit keinerlei Optimierungen vornehmen. Bei kleinen if-else Statements kein Problem. Bei größeren if-else Statements kann es hierbei Ratsam sein, auf die bessere Match-Case / Switch-Case Alternative zu wechseln. Dort können alle Klauseln gleichzeitig ausgewertet werden und in eine effizientere Reihenfolge gebracht werden.

Stellt einen HTTP Request an die Hochschul-Webseite (nutzt hierzu das Package requests). Anschließend solltet ihr den Statuscode dieser Abfrage ermitteln und mit einer Match-Case / Switch-Case Statement auf unterschiedliche Ergebnisse überprüfen. Ist das Ergebnis 100, 200, 300, 401-403, 504, gibt eine passende Ausgabe aus. Überlegt euch, was mit den übrigen HTTP Statuscodes passiert.

Tim Ehmann
Fabian Juny
Raphaele Salvatore Licciardo, B.Sc.

Aufgabenteil 2

Qualitätssicherung mit unittest

Ein Automobilhersteller vermutet einen Fehler in der Software ihrer neuen Modelle. Um diese Vermutung zu prüfen, sollen Sie nun mit dem **unittest-Modul** Tests programmieren. Schauen Sie sich zunächst die Klassen Motor und Auto an.

Mögliche Fehler (d.h. noch nicht überprüfte Fehler) wurden identifiziert. Schreiben Sie Tests, um die Software zu überprüfen. Nutzen Sie hierfür (assertEqual, assertTrue, assertIn):

- Beim Tanken wird nicht korrekt angezeit, ob der getankte Treibstoff richtig ist
- Der Name des Modells wird durch die String Methode nicht angezeigt
- Es wird zu oft angezeigt, dass das Auto eine Wartung braucht
- Der Listenpreis wird falsch gespeichert

Hinweis: unittest.main schaut auf sys.argv und der erste Parameter ist, was IPython oder Jupyter gestartet hat (daher der Fehler, dass die Kernel-Verbindungsdatei kein gültiges Attribut ist). Die Übergabe einer expliziten Liste (argv=['first-arg-is-ignored'], exit=False) an unittest.main verhindert, dass IPython und Jupyter auf sys.argv schauen. Die Übergabe von exit=False verhindert, dass unittest.main den Kernell-Prozess herunterfährt

```
class Motor():
    """ Klasse, die einen Motor repräsentiert """

    def __init__(self, zylinder, ps, treibstoff):
        self.zylinder = zylinder
        self.ps = ps
        self.treibstoff = treibstoff

    def __str__(self):
           return f"Motor mit {self.zylinder} Zylindern und
           {self.ps} PS, der {self.treibstoff} benötigt."

    def richtig_getankt(self, getankter_treibstoff):
        """ Prüft, ob ein Treibstoff dem benötigten Treibstoff
           entspricht """

        if getankter_treibstoff == self.treibstoff:
            return "Treibstoff richtig getankt"

        else:
            return "Treibstoff richtig getankt"
```

```
from datetime import datetime
class Auto():
    """ Klasse, die ein Auto repräsentiert """

    def __init__(self, modell_name, motor, tankfüllung, baujahr,
      listenpreis):
        self.modell_name = modell_name
        self.motor = motor
        self.tankfüllung = tankfüllung
        self.baujahr = baujahr
        self.listenpreis = 0

    def __str__(self):
        return f"{self.modell_name} aus dem Jahr {self.baujahr}
           hat einen {self.motor}"

    def tanken(self, treibstoff):
        """ Simuliert das Tanken und führt eine Prüfung auf
           Richtigkeit des Treibstoffs aus """

        self.tankfüllung = treibstoff
        return self.motor.richtig_getankt(self.tankfüllung)

    def benoetigt_wartung(self):
        """ Das Auto braucht alle fünf Jahre eine Wartung, dies
           soll mit dieser Methode geprüft werden """

        if (datetime.now().year - self.baujahr) % 3 == 0:
            return f"Das Auto benötigt eine Wartung."
        return f"Das Auto benötigt zuzeit keine Wartung"
```

Aufgabenteil 3

Try-Except inklusive eigener Fehlermeldungen

In einer Trading App können verschiedene Anlageprodukte gehandelt werden. Dabei sollen die Nutzer dadurch geschützt werden, dass bestimmte Fehlermeldungen in dem Kauf-/Verkaufsprozess ausgegeben werden. Liegt der Preis, für den ein Nutzer ein Produkt verkaufen will, unter dem Preis, den ein anderer Nutzer bereits für dieses Produkt bietet, soll ein "PriceTooSmallError" ausgegeben werden und der gebotene Preis vorgeschlagen werden. Dabei soll das Programm nicht abstürzen, sondern der Nutzer soll dann die Möglichkeit haben, einen neuen Preis einzugeben. Das Gleiche Prinzip gilt bei dem Kauf. Will ein Nutzer ein Produkt für ein Preis kaufen, der über dem minimal verlangten Preis liegt, so soll ein "PriceTooHighError" zusammen mit dem vorgeschlagenem Preis ausgegeben werden und der Nutzer soll auch hier einen neuen Preis eingeben können.

```
class PriceTooSmallError(Exception):
    """Falls ein Nutzer ein Anlageprodukt zu einem Preis
verkaufen   will, der unter dem Preis liegt, den ein anderer
Nutzer für diesen bietet, so wird dieser Fehler ausgegeben."""
    pass

class PriceTooHighError(Exception):
    """Falls ein Nutzer ein Anlageprodukt zu einem Preis kaufen
will, der über dem Preis liegt, den ein anderer Nutzer für diesen
verlangt, so wird dieser Fehler ausgegeben."""
    pass

class Produkt():

    def __init__(self, name, ask, bid):
        self.name = name
        self.ask = ask      # ask ist der niedrigste Preis, für
den ein Nutzer ein Produkt verkaufen will
        self.bid = bid      # bid ist der höchste Preis, für den
ein Nutzer ein Produkt kaufen will

    def __str__(self):
        return f"Produktname: {self.name}, ask: {self.ask}, bid:
{self.bid}"

    def sell_product(self, sell_price):
        if sell_price < self.bid:
            raise PriceTooSmallError(sell_price, self.bid)
        elif sell_price < self.ask:
            self.ask = sell_price

    def buy_product(self, buy_price):
        if buy_price > self.ask:
            raise PriceTooHighError(buy_price, self.ask)
        elif buy_price > self.bid:
            self.bid = buy_price


def sell_product(product: Produkt, price: float):
    """ Es soll versucht werden, ein Produkt für einen Preis zu
verkaufen"""
    pass

def buy_product(product: Produkt, price: float):
    """ Es soll versucht werden, ein Produkt für einen Preis zu
kaufen"""
    pass


msft = Produkt("Microsoft", 260, 255)

print(msft)
sell_product(msft, 245)
sell_product(msft, 258)
buy_product(msft, 265)
print(msft)
```