University of Applied Science - Online

Master: Data Science

Programmieren in Python

Tim Willkens

Matrikelnummer: IU14073577

Eiergäßle 11
89160 Dornstadt.

Advisor: Dr. Prof. Thomas Kopsch

Delivery date: 2. 5. 2024

Contents

I List of Figures

II List of Tables

1 Introduction

Python, benannt nach der britischen Komikertruppe Monty Python, hat sich in den vergangenen Jahren zu einem Schwergewicht unter den Programmiersprachen entwickelt, Steyer (2018). Sie unterstützt sowohl die objektorientierte, die prozedurale sowie die funktionale Programmierung, Häberlein (2024). Python wurde mit dem Ziel zur Förderung eines gut lesbaren sowie knappen Programmierstiels entwickelt, Steyer (2018). Im Gegensatz zu compelierten Sprachen wie Java oder C # ist zählt Python zu den interpretierten Programmiersprachen und kann somit ohne weiteres auf einem anderen Betriebssystem ausgeführt werden, Häberlein (2024).
Aufgrund der Vielzahl an Bibliotheken wie NumPy, Pandas, SciKit-Learn und MatplotLib hat Python Einzug als Standardprogramm in die Wissenschaften, die Data Science und des maschinellen Lernen erhalten, VanderPlas (2023).
In der nachfolgenden Arbeit wird auf grundlegenden Techniken zur Programmentwicklung mittels Python eingegangen. Beginnend mit der Objektorientierten Programmierung, welche sich als bewährte Methode zur Erstellung komplexer Softwaresysteme erwiesen hat, Lahres et al. (2021).
Weiter werden Teile der Datenverarbeitung mit Pandas sowie dem Arbeiten mit Datenbanken behandelt. Pandas ist eine Open Source Bibliothek welche vorallem im Data Science Umfeld zum Einsatz kommt, Nelli (2023). Die Visualisierung erfolgt mittels *bokeh*. Bokeh ist eine Python-Bibliothek zur Erstellung interaktiver Visualisierungen für moderne Webbrowser, Bokeh (2024)
Datenbanken sind ein zentraler Bestandteil im Umgang mit großen Datenmengen. Hier hat sich SQL (Strucured Query Language) zum internationaler Standard entwickelt, Tylor (2023). Mittels MySQL oder SQLite kann eine SQL Datenbank erstellt und bearbeitet werden. Auch hier bietet Python Bibliotheken an, welche das arbeiten mit SQL-Datenbanken besonders angenehm machen.
Das erzeugte Programm soll nach gängigen Entwicklungsstandards. Dies beinhaltet unter anderem auch eine klare Dokumentation. Hierzu werden sogenannte *docstrings* und *unittests* verwendet. Ein *docstring* ist ein String-Literal welcher direkt im Sourcecode eingefügt wird, Pajankar (2022). Ein *unittest* ist eine Testmethode bei der einzelne Komponenten eines Programms unabhängig voneinander getestet werden, Pajankar (2022).
Die Versionskontrolle wird über *Git* verwaltet. Speziell in großen Software-Projekten bietet Git den Entwicklern die Erstellung und Pflege der Versionskontrolle und erstellt im Vergleich zu CVS per se keine kanonische Kopie der Codebasis. Alle Kopien sind Arbeitskopien und können lokal verändert werden ohne mit einem Server verbunden zu sein Russel / Klassen (2019)

2 Objektorientierte Programmierung

Die objektorientierte Programmierung hat seine Anfänge in den 1970er Jahren und setzte sich in den 1990er Jahren mit der Einführung von Java durch, Steyer (2018). Sie unterstützt komplexe Softwareentwicklung dabei, Software einfacher erweiterbar, besser testbar und besser wartbar zu machen, Lahres et al. (2021). Als wesentliche Grundelemente der OOP gelten, Lahres et al. (2021)

- Unterstützung von **Vererbungsmechanismen**:
  Attribute, Methoden und Ereignisse werden von der Basisklasse auf die abgeleitete Klasse übertragen. Dies ermöglicht die Wiederverwendung von Code und die Erweiterung der Funktionalität.
- Unterstützung von **Datenkapselung**:
  Ist ein Konzept, Daten und Informationen vor dem direkten Zugriff von außen zu verbergen. Es ermöglicht die Kontrolle über den Zugriff auf die internen Datenstrukturen eines Objekts und erfolgt über definierte Schnittstellen.
- Unterstützung von **Polymorphie**:
  Ermöglicht, dass ein Bezeichner abhängig von seiner Verwendung Objekte unterschiedlichen Datentyps annimmt. Dies bedeutet, dass eine einzige Schnittstelle oder Methode verschiedene Implementierungen haben kann.

Objektorientierte Programmierung (OOP) ist somit ein Programmierparadigma, dass auf dem Konzept von ”Objekten” basiert die Datenstrukturen enthalten und Verhaltensweisen (Methoden) definieren. Diese Objekte sind Instanzen von Klassen, die als Blaupausen für Objekte dienen.

2.1 Klassen

Eine Klasse ist eine Vorlage oder ein Bauplan für die Erstellung von Objekten. Sie definiert die Attribute und Methoden, die ein Objekt haben wird. Ein Attribut ist eine Eigenschaft oder ein Merkmal, das ein Objekt hat, während eine Methode eine Funktion ist, die ein Objekt ausführen kann, Steyer (2018). Klassen werden durch das Schlüsselwort *class* definiert.

```
class Data():
  def __init__(self, datapath):
    try:
      self.datapath = datapath
      self.df = pd.read_csv(datapath)
    except FileNotFoundError:
      print(sys.exc_info())
    cols = self.df.columns
    for i in cols:
      self.__dict__[i] = self.df[i]
    self.dfSortByX = self.df.sort_values(['x'])
```

Listing 2.1: class Data

2.2 Methoden und Vererbung

Die Klasse *Data* hat eine Methode *__init__*, welche den Parameter *datapath* entgegen nimmt und versucht, eine CSV-Datei von diesem Pfad zu lesen und in ein DataFrame zu konvertieren. Wird die Datei nicht gefunden, wird eine Fehlermeldung ausgegeben. Die Methode *__init__* initialisiert auch andere Attribute der Klasse, wie df und *dfSortByX*.

Allgemein ist die *__init__* Methode in Python ist eine spezielle Methode, die automatisch aufgerufen wird, wenn eine Instanz (ein Objekt) einer Klasse erstellt wird. Sie wird verwendet um die Attribute der Klasse zu initialisieren. Die *self* Variable repräsentiert die Instanz der Klasse und wird verwendet um auf die Attribute und Methoden der Klasse zuzugreifen, Häberlein (2024).

Vererbung ist ein Prinzip der OOP, dass es ermöglicht eine neue Klasse zu erstellen, welche die Attribute und Methoden einer bestehenden Klasse erbt. Die neue Klasse wird als ”Unterklasse” oder ”abgeleitete Klasse” bezeichnet, während die bestehende Klasse als ”Oberklasse” oder ”Basis-Klasse” bezeichnet wird, Steyer (2018). Die Klasse *IdealFunctions* erbt von Data und fügt die Methode *GetIdealFunctions* hinzu, die eine ideale Funktion basierend auf einem Trainingsdatensatz berechnet. Anhand des kleinsten Fehler zwischen den Trainingsdaten und den Daten der Basisklasse gibt die Methode die Indizes der idealen Funktionen zurück.

```
class IdealFunctions(Data):
  def __init__(self, datapath):
    Data.__init__(self, datapath)
  def GetIdealFunctions(self, dfTrain:pd.DataFrame):
    ...
```

Listing 2.2: class IdealFunctions

Die Klasse *TestData* erbt ebenfalls von Data und enthält die Methode *Segmentation*, die eine Segmetierungsfunktion für einen gegebenen Datensatz implementiert. Diese Methode ordnet Datenpunkte basierend auf einem Schwellenwert *threshold*, welche mit $\sqrt{2}$ initialisiert wird, einer Idealen Funktion zu.

```
class TestData(Data):
  def __init__(self, datapath):
    Data.__init__(self, datapath)
  def Segemtation(self, dataFrame:pd.DataFrame, threshold= np.sqrt(2))
    ...
```

Listing 2.3: class TestData

3 Python Bibliotheken

Für Python gibt es Bibliotheken zum Laden von Daten, Visualisieren, Berechnen von Statistiken, Sprachverarbeitung, Bildverarbeitung usw. Dies gibt Data Scientists einen sehr umfangreichen Werkzeugkasten mit Funktionalität für allgemeine und besondere Einsatzgebiete, Müller / Guido (2017). Numpy ist das Modul der Wahl, wenn man wissenschaftlich rechnen möchte und wird insbesondere für statistische Auswertungen, für Machine-Learning und allgemein für sehr aufwändige Berechnungen eingesetzt. Auf Numpy setzt unter anderem auch TensorFlow, pandas, scipy, scikit-learn auf, Häberlein (2024).

3.1 Numpy

NumericalPython, kurz NumPy, bietet Datenstrukturen und Algorithmen speziell für wissenschaftliche Anwendungen, McKinney (2019). Grund hierfür ist, dass die Berechnung bei großen Daten-Arrays besonders effizient ist und somit schneller als normaler Python Code sein kann, McKinney (2019).
Viele weitere Bibliotheken wie Pandas bauen auf NumPy auf, VanderPlas (2023). Weitere Informationen der einzelnen Module können aus der NumPy Dokumentation entnommen werden, NumPy (2024)

3.2 Pandas

Pandas ist eine neuere Bibliothek und findet in der DataScience hohe Anwendung. Sie bietet eine komfortable Schnittstelle zum speichern von Daten sowie eine Vielzahl an nützlicher Operationen. Die drei wichtigsten Datenstrukturen sind *Series*, *Index* und *DataFrames*, VanderPlas (2023).
Ein DataFrame ist eine zweidimensionale, potenziell heterogene tabellarische Datenstruktur mit beschrifteten Achsen (Zeilen und Spalten). Diese entsprechen einem Zeilen- sowie Spaltenindex. Es kann als eine Art Container für Series-Objekte betrachtet werden, ähnlich wie ein Wörterbuch, das Spaltennamen als Schlüssel und Series-Objekte als Werte enthält. Ein DataFrame kann aus verschiedenen Datenquellen wie Listen, Dictionaries oder anderen DataFrames erstellt werden. Es bietet eine Vielzahl von Funktionen zur Datenmanipulation und -analyse und ist besonders nützlich für Aufgaben im Bereich der Datenanalyse. Pandas DataFrames können beispielsweise aus CSV-Dateien, Excel-Dateien oder SQL-Datenbanken eingelesen werden, VanderPlas (2023).
Beispielsweise kann ein Datensatz der Form

x	y_1	y_2	y_3	y_4
-20.0	100.21	-19.75	0.34	19.77
-19.9	99.89	-19.70	0.61	19.78
-19.8	99.39	-19.56	0.17	19.44
-19.7	98.24	-19.85	0.73	19.86

Table 1: Beispiel Panda DataFrame

folgendermaßen

```
data =
{
        'x': [-20.0, -19.9, -19.8, -19.7],
```

```
        'y1': [100.216064, 99.894684, 99.397385, 98.24446],
        'y2': [-19.757296, -19.70282, -19.564255, -19.858267],
        'y3': [0.3461139, 0.61786354, 0.1743704, 0.7310719],
        'y4': [19.776287, 19.789793, 19.441765, 19.869267]
}
df = pd.DataFrame(data)
```

Listing 3.1: Pandas Dataframe

in ein Dataframe geschrieben werden. Dieses DataFrame hat den Spaltenindex *'x', 'y1', y2', 'y3', 'y4'*, abrufbar über *df.columns*, sowie den Zeilenindex *'0', '1', '2', '3'*, abrufbar über *df.index*.

In der Klasse *Data* aus Kapitel 2.1 wird mittels der Pandas Funktion

```
self.df = pd.read_csv(datapath)
```

eine .csv Datei eingelesen und als DataFrame gespeichert.

3.3 Bokeh

Bokeh ist eine interaktive Datenvisualisierungsbibliothek für Python, die sich dadurch auszeichnet, umfassende interaktive Grafiken zu erstellen. Sie ist kompatibel mit Webtechnologien wie HTML, CSS sowie JavaScript und kann auf verschiedenen Plattformen wie Jupyter Notebook, JupyterLab, Flask und Django eingesetzt werden. Sie bietet eine breite Palette von Diagrammtypen, darunter Linien-, Streu-, Balken-, Flächen-, Histogramme- und zahlreiche weitere Diagramme. Bokeh zeichnet sich durch seine Schnelligkeit bei der Arbeit mit großen Datensätzen aus und ermöglicht es Nutzern, Grafiken anzupassen und interaktiv zu betrachten.

Abbbildung 1 dient beispielhaft für die Erstellung eines Plots mittels Bokeh. Diese wurde mit dem *scatter*-Befehl erstellt und verarbeitet die *x* und *y* werte eines Pandas *DataFrame*.

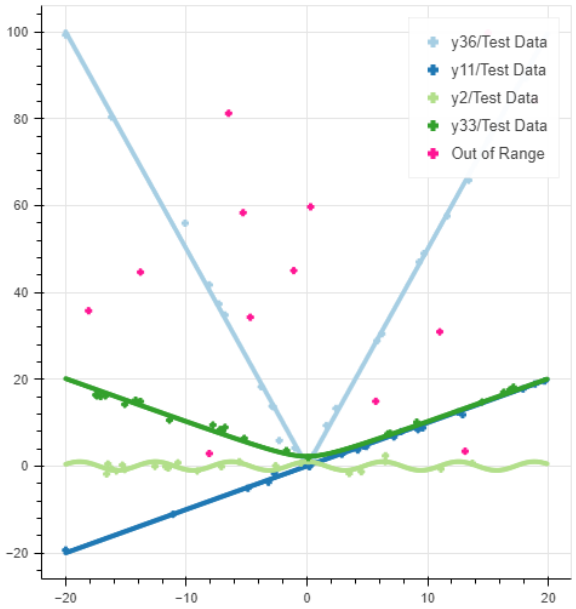

Figure 1: Beispiel eines Plots mit Bokeh

```
ax.scatter(idFcnsDf['x'].values, idFcnsDf['y'].values,size=3,
        color=colors[i])
```

Listing 3.2: Bsp.: Bokeh Scatter-Plot

Müssen mehrere Farben verwendet werden, bietet Bokeh eine breite Auswahl an Farbpaletten. Abbildung 1 wurde mit der *brewer paired* Farbpalette erstellt.

```
colors = palettes.brewer['Paired'][4]
```

Listing 3.3: Bokeh Farb-Palette

4 Datenbanken

Ein zentraler Bestandteil bei der Programmierung ist das arbeiten mit Daten und Dateien. Um mit großen und komplexen Datenmengen effizient und widerspruchsfrei zu arbeiten wird ein Verwaltungssystem benötigt. Dies führt zu Datenbankkonzepten wie SQL, Steyer (2018). Dabei ist SQL (Structured Query Language) ein internationaler Standard zum arbeiten mit Daten, aber keine Universalsprache wie etwa *Java* oder *C*, Tylor (2023). Es wurde entwickelt um mit Relationalen Datenbanken zu arbeiten, welche Daten in Tabellenform organisieren sowie Beziehungen zwischen Tabellen herstellen können.

Python bietet Bibliotheken zum arbeiten mit SQL Datenbankkonzepten wie *MySql*, *SQLite* und weitere an.

Nachfolgend sind die wichtigsten schritte zum arbeiten einer SQLite - Datenbanken sowie der *SQLAlchemy*-Bibliothek aufgeführt.

- Verbinden mit der Datenbank *dbName* über *conncet()*

```
my_db = sqlite3.connect(dbName)
```

- Erstellen eines *Engine*- Objekts mit dem Pfad zur Datenbank

```
engine = db.create_engine(databasePath)
```

- Speichern eines DataFrames in eine Datenbank

```
dataFrame.to_sql(dbTableName, con=engine, if_exists='replace', index = False)
```

- Lesen von Daten aus der Datenbank und in einem DataFrame zur Verfügung stellen

```
tableDF = pd.read_sql_table(self.dbTableName, engine, columns = None)
```

- Verbindung zur Datenbank beenden (um Ressourcen zu schonen)

```
engine.dispose()
my_db.close()
```

5 Versionskontrolle mit Git

Speziell in großen und komplexen Software-Projekten ist eine gute Versionskontrolle des zu entwickelnden Codes unabdingbar. Hierfür hat sich Git in den letzten Jahren als Standard entwickelt. Git ist besonders leistungsfähiges und flexibles Versionskontrollwerkzeug mit geringem Aufwand, dass die Entwicklung im Team immens vereinfacht, Loeliger (2012).

Hierbei ermöglicht Git das arbeiten in sogenannten *Repositories*. Dies sind einfache Datenbanken, welche alle nötigen Informationen enthält, die für die Aufbewahrung und Verwaltung der Revisionen und des Verlaufs eines Projekts erforderlich sind, Loeliger (2012). Dabei werden zwischen lokalen und remote Repositories unterschieden. Auf das remote Repository kann das gesamte Entwicklungsteam zurückgreifen und enthält den aktuellsten Stand der zu entwickelnden Software. Die Entwicklung hingegen erfolgt meistens auf lokalen Repositories, auf welche nur einzelne Entwickler, bzw kleine Entwicklungsteams zugriff haben.

Über den **clone**- Befehl kann ein Klon des remote Repository im lokalen Repository erzeugt werden. Weitere wichtige Befehle zum arbeiten mit Repositories sind:

- **pull**: Änderungen aus einem (remote) Repository in das (lokale) Repository übernehmen.
- **push**: Änderungen des (lokalen) Repository in das (remote) Repository einfügen.

Zentraler Bestandteil der Versions-Kontrolle sind sogenannte *commits*. Werden Änderungen am Code bzw. am Software-Projekt vorgenommen, dann kann einen neue Version über den **commit**- Befehl erstellt werden. Hierbei werden auch nützliche Informationen wie der Author, Kommentare und eine Commit-Id, sowie alle Änderungen gespeichert. Somit ist zu jedem Zeitpunkt möglich diesen Stand wiederherzustellen.

Weiter ist es möglich, dass mehrere Entwickler parallel an der Entwicklung arbeiten. Hierzu bietet Git das arbeiten in sogenannten Branches. Eine Branche ist eine Abzweigung eines bestimmten Commits und somit einem bestimmten Stand. Diese werden über den **branch**- Befehl erzeugt. Weitere wichtige Befehle zum Arbeiten mit Branches sind:

- **chekout**: Wechsel in eine andere Branch.
- **merge**: Zwei Branches zu einer Branch zusammenführen.

In Abbildung 2 ist beispielhaft ein Git-Projekt dargestellt. Hier sind vier Branches (testing, dev, master, Stabel) sowie mehrere Commits (A - Z) zu sehen.

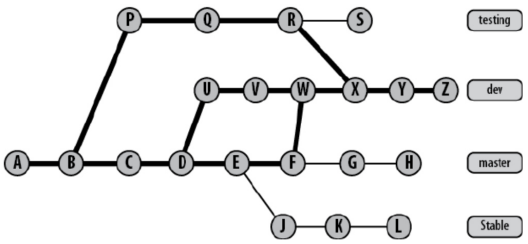

Figure 2: Branches und Commits, Loeliger (2012)

6 Zusammenfassung

Zusammenfassend lässt sich sagen, dass Python eine moderne Programmiersprache ist, welche vollständig auf die objektorientierte Programmierung (OOP) setzt, um die Modellierung komplexer Systeme zu erleichtern und dabei leicht verständlich ist. Die OOP in Python orientiert sich an der realen Welt, indem sie Software-Objekte schafft, die für bestimmte Aufgaben zuständig sind. Diese Objekte sind Sammlungen von Daten und Verhalten, die in Klassen organisiert sind, welche als Vorlagen für die Objekte dienen.

Die OOP-Konzepte in Python umfassen die Vererbung, bei der Klassen Eigenschaften von anderen Klassen erben können. Diese Konzepte erleichtern die Wiederverwendung von Code und tragen zur Erstellung sicherer und zuverlässiger Software bei.

Python verfügt über ein umfangreiches Ökosystem an Bibliotheken, die die Datenanalyse und -visualisierung unterstützen. Pandas ist eine Bibliothek, die sich für die Arbeit mit tabellarischen Daten eignet und sich nahtlos in Bokeh integrieren lässt, um interaktive Datenvisualisierungen im Webbrowser zu erstellen. Weiter erleichtert Pandas das Arbeiten mit Datenbanken. Numpy ist eine weitere zentrale Bibliothek, die vor allem für numerische Berechnungen verwendet wird.

Auch für das arbeiten mit Datenbanken wie SQLite bietet Python diverse Bibliotheken und Lösungen. SQLite ist eine leichtgewichtige Datenbank, die direkt in Python-Anwendungen integriert werden kann. Das Arbeiten mit SQLite in Python ermöglicht es, Daten effizient zu speichern und abzufragen, was besonders nützlich ist, wenn keine vollwertige Datenbankserver-Infrastruktur benötigt wird.

Abschließend ist Git ein unverzichtbares Werkzeug für die Versionskontrolle in der Softwareentwicklung. Es ermöglicht Entwicklern, Änderungen am Code nachzuverfolgen, im Team zusammenzuarbeiten und verschiedene Versionen eines Projekts zu verwalten. Python-Entwickler nutzen Git häufig, um ihre Projekte zu organisieren und mit der Community zu teilen.

Zusammenfassend ist Python eine vielseitige Sprache, die durch ihre OOP-Fähigkeiten, ein starkes Ökosystem an Bibliotheken und die Integration mit Werkzeugen wie SQLite und Git eine solide Grundlage für moderne Softwareentwicklung bietet.

Bibliography

Bokeh (**2024**): *Bokeh documentation*. (URL: `https://docs.bokeh.org/en/latest/` [last access: 2. 5. 2024]).

Häberlein, T. (**2024**): *Programmierung mit Python: Eine Einführung in die Prozedurale, Objektorientierte und Funktionale Programmierung*. 1. Auflage, Springer Nature, Berlin.

Lahres, B./Rayman, G./ Strich, S. (**2021**): *Objektorientierte Programmierung: Das umfassende Handbuch*. 5. Auflage, Reihnwerk Verlag, Bonn.

Loeliger, J. (**2012**): *Version control with Git*. 2. Auflage, O'Reilly Media, Sebastopol, Calif.

McKinney, W. (**2019**): *Datenanlyse mit Python: Auswertung von Daten mit Pandas, NumPy und Jupiter*. 3. Auflage, O'Reilly Media, Sebastopol, Calif.

Müller, A./ Guido, S. (**2017**): *Einführung in Machine Learning mit Python : Praxiswissen Data Science*. 1. Auflage, O'Reilly Media, Sebastopol, Calif.

Nelli, F. (**2023**): *Python Data Analytics : With Pandas, NumPy, and Matplotlib*. 2. Auflage, Apress L. P., ProQuest Ebook Central.

NumPy (**2024**): *NumPy documentation*. (URL: `https://numpy.org/devdocs/` [last access: 2. 5. 2024]).

Pajankar, A. (**2022**): *Python Unit Test Automation - Automate, Organize, and Execute Unit Tests in Python*. 2. Auflage, Apress, ProQuest Ebook Central.

Russel, M./ Klassen, M. (**2019**): *Mining the Social Web : Data Mining Facebook, Twitter, LinkedIn, Instagram, GitHub, and More*. 2. Auflage, O'Reilly Media, Sebastopol, Calif.

Steyer, R. (**2018**): *Programmierung in Python: Ein kompakter Einstieg für die Praxis*. 1. Auflage, Springer Nature, Wiesbaden.

Tylor, A. G. (**2023**): *SQL All-In-One for Dummies*. 3. Auflage, John Wiley and Sons, Incorporated.

VanderPlas, J. (**2023**): *Python Data Science Handbook*. 2. Auflage, O'Reilly Media, Sebastopol, Calif.

```
"""
    import necessary libs
"""

import numpy as np
import pandas as pd
from bokeh.plotting import figure, show
from bokeh import palettes
import mysql.connector
import sqlalchemy as db
import sys
import os
import unittest
import sqlite3
```

```
'''
--------------------------------
---       Classes File         ---
--------------------------------
'''

from Bibs import pd, np, sys, mysql, db, sqlite3

# Class Data
class Data():
    """
   This class is used to import data from a CSV file and store it in a
   pandas DataFrame.
   """
    def __init__(self, datapath):
        """
        Initialize the Data class with the path to the data file.
        Parameters:
        datapath (str): The path to the data file.
        """
        self.datapath = datapath
        try:
            # write csv Data into DataFrame
            self.df = pd.read_csv(datapath)
            #self.df.set_index('x', inplace=True)
            cols = self.df.columns
            for i in cols:
                self.__dict__[i] = self.df[i]

        except FileNotFoundError:
            print(sys.exc_info())



class IdealFunctions(Data):
    """
    This class is a child of the Data class. It is used to get the best
    fitting functions from a DataFrame.
    """
    def __init__(self, datapath):
        """
       Initializes the IdealFunctions class with the given datapath.

       Parameters:
           - datapath (str): The path to the data.

       """
        Data.__init__(self, datapath)

    def GetIdealFunctions(self, dfTrain:pd.DataFrame):
        """
        Get the best fit function from an input DataFrame (Training Data).

        Parameters:
            - dfTrain (pd.DataFrame): The input Training DataFrame.

        Returns:
```

```
            - list: A DataFrame of the best fit functions, and a
            DataFrame of the FunctionNumber with the MSE.

        Raises:
            - Errors: If there is an error during the process of
            getting the best fit functions.
        """
        try:
            bestFitFcns = []
            bestFitError = []
            # Interate over all train - data
            for y in dfTrain.columns[1:]:
                # Initialize Error DF
                error = pd.Series(np.zeros(self.df.shape[1]-1), index=self.df.columns[1:])
                for idx, row in dfTrain[['x',y]].iterrows():
                    # get best Fit X- Value
                    bestFitIdx = (abs((self.df['x'] - row['x'])))
                    # get Y-Values to best Fit X-Value
                    bestFitData = self.df.iloc[bestFitIdx.argmin()]
                    # calc Square Error (without X)
                    y_diff = ((bestFitData.drop('x').values - row[y])**2)
                    error[:] = error.values + y_diff
                # get best fit function
                bestFitFcns.append(error.idxmin())
                try:
                    # calc mean square error
                    bestFitError.append(error[error.idxmin()] / dfTrain[y].shape[0])
                except ZeroDivisionError():
                    print(ZeroDivisionError())

            ErrorDF = pd.DataFrame(bestFitError, index=bestFitFcns, columns=['mse'])
            FiltDf = pd.concat([self.df['x'],self.df[ErrorDF.index]],axis=1)
            return [FiltDf, ErrorDF]
        except Errors:
            print(Errors().my_message1)

# Class TestData child of Data
class TestData(Data):
    """
    This class is a child of the Data class. It is used to segment the test data.
    """
    def __init__(self, datapath):
        Data.__init__(self, datapath)

     # Function:
    def Segmentation(self, idFcnDf:pd.DataFrame(), threshold = np.sqrt(2)):
        """
        Segment the test data based on the ideal functions and a threshold.

        Parameters:
            - idFcnDf (pd.DataFrame): The DataFrame of ideal functions.
            - threshold (float): (optional) The threshold for segmentation.
            Default is sqrt(2).

        Returns:
            - DataFrame: The segmented test data, which assigns the
            test data to an ideal function if the distance (Test - Fcn) < threshold.
```

```
        Raises:
            - Errors: If there is an error during the segmentation process.
        """
        try:
            result      = self.df.sort_values(['x'])
            minDist     = []
            bestFitFcn  = []
            for idx, row in result.iterrows():
                distance = []
                for y in idFcnDf.columns[1:]:
                    # get squared X- Error
                    bestFitIdx = (idFcnDf['x'] - row['x'])**2
                    # get squared Y- Error
                    bestFitData = (idFcnDf[y] - row['y'])**2
                    # get squared Error / Distance
                    distance.append(np.sqrt(bestFitIdx + bestFitData).min())
                if min(distance) < threshold:
                    # get min squared Error
                    minDist.append(min(distance))
                    # get best Fit Function
                    bestFitFcn.append(idFcnDf.columns[distance.index(min(distance)) + 1])
                else:
                    minDist.append(min(distance))
                    bestFitFcn.append('Data out of range')

            result['Delta'] = minDist
            result['Nummer Ideale Funktion'] = bestFitFcn
            return result
        except Errors:
            print(Errors().my_message2)


# Class Database Handling
class DB_Handling:
    '''
    DB Handling:
        Connect to a Database
        Store Data (Tabel) to a Database
        Get Data (Tabel) from a Database
    '''
    def __init__(self, dbName):
        '''
        Initializes the DB_Handling class with the given database name.

        Parameters:
            - dbName (str): The name of the database.
        '''
        self.dbName = dbName
        #self.databasePath = 'mysql+mysqlconnector://TestUser:MyT3st_SQL@localhost/'
        self.databasePath = 'sqlite:///' + self.dbName

    def Db_Connect(self):
        '''
        Connects to the database.

        Raises:
```

```
        - DBErrorHandling: If there is an error connecting to the database.
    '''
    try:
        # Connection to SQLite
        my_db = sqlite3.connect(self.dbName)
        my_cursor = my_db.cursor()
        my_db.close()

        # Connection to MySQL
        '''
        my_db = mysql.connector.connect(host="localhost",
            user="TestUser", password="***")
        my_cursor = my_db.cursor()
        my_cursor.execute("SHOW DATABASES")
        xInit = False
        for dat in my_cursor:
            if dat[0] == self.dbName:
                xInit = True
                break
        try:
            if not xInit:
                print(DBErrorHandling(self.dbName,self.dbTableName).my_message1)
                my_cursor = my_db.cursor()
                createDataBase = "CREATE DATABASE "  + self.dbName
                my_cursor.execute(createDataBase)
                print(DBErrorHandling(self.dbName,self.dbTableName).my_message3)
        except DBErrorHandling:
            print(DBErrorHandling(self.dbName,self.dbTableName).my_message4)
        my_db.close()
        '''
    except DBErrorHandling:
        print(DBErrorHandling(self.dbName,self.dbTableName).my_message5)


def Db_StoreTable(self, dbTableName, dataFrame:pd.DataFrame):
    '''
    Stores a table in the database.

    Parameters:
        - dbTableName (str): The name of the table.
        - dataFrame (pd.DataFrame): The data to be stored in the table.

    Raises:
        - DBErrorHandling: If there is an error connecting to the
        database or storing the table.

    '''
    self.dbTableName = dbTableName
    self.dataFrame = dataFrame
    try:
        self.Db_Connect()
        engine = db.create_engine(self.databasePath)
        dataFrame.to_sql(dbTableName, con=engine, if_exists='replace',
                         index = False)
        engine.dispose()
    except DBErrorHandling:
        engine.dispose()
```

```
            print(DBErrorHandling(self.dbName, self.dbTableName).my_message8)


    def DB_GetTable(self, dbTableName):
        '''
        Retrieves a table from the database.

        Parameters:
            - dbTableName (str): The name of the table to retrieve.

        Returns:
            - tableDF (pd.DataFrame): The retrieved table.

        Raises:
            - DBErrorHandling: If there is an error connecting to the database
            or retrieving the table.

        '''
        self.dbTableName
        try:
            self.Db_Connect()
            engine = db.create_engine(self.databasePath)
            tableDF = pd.read_sql_table(self.dbTableName, engine, columns = None)
            engine.dispose()
        except DBErrorHandling:
            engine.dispose()
            print(DBErrorHandling().my_messageConError)

        return tableDF

class Errors(Exception):
    '''
    Error of Functions
    '''
    def __init__(self):
        self.my_message1 = 'Best Fit Functions Error'
        self.my_message2 = 'Segmentation Error'

class DBErrorHandling(Exception):
    '''
    Error of DbHandling
    '''
    def __init__(self, dbName, dbTable):
        my_message1 = 'Datenbank ' + dbName + ' existier nicht!'
        self.my_message1 = my_message1

        my_message2= 'Fehler beim Speichern von ' + dbTable +': Tabelle existiert nicht!'
        self.my_message2 = my_message2

        my_message3 = 'Datenbank ' + dbName + ' erfolgreich erstellt'
        self.my_message3 = my_message3

        my_message4 = 'Fehler beim erstellen der Datenbank ' + dbName
        self.my_message4 = my_message4

        my_message5 = 'Fehler beim Verbinden mit Datenbank ' + dbName
        self.my_message5 = my_message5
```

```
my_message6 = 'Speichern der Tabelle ' + dbTable + ' erfolgreich!'
self.my_message6 = my_message6

my_message7 = 'Tablle ' + dbTable + ' erfolgreich erstellt'
self.my_message7= my_message7

my_message8 = 'Fehler beim Speichern von ' + dbTable
self.my_message8= my_message8
```

```
'''
-------------------------
---       main        ---
-------------------------

'''
# Import libs and Classes
from Bibs import pd, np, sys, mysql, os, figure, show, palettes
from Classes import Data, TestData, IdealFunctions, DB_Handling
# Init-Paths and Files
testFile    = r'\test.csv'              # Using Raw-String => r'\'
trainFile   = r'\train.csv'
fcnsFile    = r'\ideal.csv'

codePath    = os.getcwd()
projectPath = os.path.dirname(codePath)
dataPath    = projectPath + r'\DataSet'


# Create Data-Objects as Instances of the classes
Train   = Data(dataPath + trainFile)
Test    = TestData(dataPath + testFile)
Fcns    = IdealFunctions(dataPath + fcnsFile)

# Get Ideal Functions for Training-Data
[idFcnsDf, mseDf] = Fcns.GetIdealFunctions(Train.df)
# Check the Ideal-Functions for Test-Data
SegmentationCheck = Test.Segmentation(idFcnsDf)

# Store Data to Databese
dbName  = 'prg_python_database'
tab1    = 'tabelle_1_Trainings_Daten'      # Traning-Data
tab2    = 'tabelle_2_Ideale_Funktionen'    # Ideal-Functions
tab3    = 'tabelle_3_Test_Daten'           # Test-Data

DB = DB_Handling(dbName)
DB.Db_StoreTable(tab1, Train.df)
DB.Db_StoreTable(tab2, Fcns.df)
DB.Db_StoreTable(tab3, SegmentationCheck)

tst = DB.DB_GetTable(tab3)     # Test: Getting Data from Database


# Plotting Data
# Create Colors by brewer palette
colors = palettes.brewer['Paired'][idFcnsDf.columns.shape[0]]

ax = figure(width=500, height=500)
i = 0
for y in idFcnsDf.columns[1:]:
    ax.scatter(idFcnsDf['x'].values, idFcnsDf[y].values,size=3,color=colors[i])
    FiltDf = SegmentationCheck[SegmentationCheck[SegmentationCheck.columns[3]]==y]
    xFilt =  FiltDf['x'].values
    yFilt = FiltDf[FiltDf.columns[1]].values
    ax.scatter(xFilt, yFilt, size=5,color=colors[i],
               legend_label = y + '/Test Data', marker='plus')
    i=i+1
```

```
FiltDf = SegmentationCheck[SegmentationCheck
                           [SegmentationCheck.columns[3]]=='Data out of range']
xFilt =  FiltDf['x'].values
yFilt = FiltDf[FiltDf.columns[1]].values
ax.scatter(xFilt, yFilt, size=5, color='deeppink',
           legend_label = 'Out of Range', marker='plus')
show(ax)
```

```
'''
---------------------------------
---       unittests           ---
---------------------------------
'''

from Bibs import unittest, np, pd,os
from Classes import Data, IdealFunctions, TestData, DB_Handling


class TestDataClass(unittest.TestCase):

    def test_Data_class(self):
        trainFile   = r'\train.csv'
        codePath    = os.getcwd()
        projectPath = os.path.dirname(codePath)
        dataPath    = projectPath + r'\DataSet'
        self.assertEqual(self.data.datapath, dataPath+trainFile)
        self.assertTrue(os.path.exists(self.data.datapath))
        self.assertIsInstance(self.data.df, pd.DataFrame)

    def test_IdealFunctions_class(self):
        trainFile   = r'\train.csv'
        idealFile   = r'\ideal.csv'
        codePath    = os.getcwd()
        projectPath = os.path.dirname(codePath)
        dataPath    = projectPath + r'\DataSet'

        ideal_func = IdealFunctions(dataPath+idealFile)
        trainData = Data(dataPath+trainFile)
        result = ideal_func.GetIdealFunctions(trainData.df)
        self.assertIsInstance(result, list)
        self.assertIsInstance(result[0], pd.DataFrame)
        self.assertIsInstance(result[1], pd.DataFrame)

    def test_TestData_class(self):
        testFile   = r'\test.csv'
        codePath    = os.getcwd()
        projectPath = os.path.dirname(codePath)
        dataPath    = projectPath + r'\DataSet'

        test_data = TestData(dataPath+testFile)
        idFcnDf = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})
        result = test_data.Segmentation(idFcnDf)
        self.assertIsInstance(result, pd.DataFrame)


if __name__ == '__main__':
    unittest.main()
```

INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

Eidesstattliche Erklärung

I hereby certify...

.....	.....
Place, date	Signature