Case Study I

Julian Huber & Matthias Panny

Grundgerüst der App

Übersicht

- Ihnen ist eine Beispiel-Implementierung gegeben, welche die Grundfunktionalität der App enthält
- Diese können Sie nach Bedarf erweitern
- Business-Logik und Datenbankabfragen
  - `users.py` und `devices.py` enthalten die Klassen, welche die Daten verarbeiten
- User-Interface
  - `ui_device.py` enthält die Visualisierung für das Gerätemanagement
- Datenbankabfragen
  - `queries.py` enthält die Datenbankabfragen
  - `database.json` enthält die Datenbank

Decorators

Definition

- eine Funktion, die eine andere Funktion (oder Methode) als Eingabe nimmt, sie erweitert oder verändert
- gibt die erweiterte oder veränderte Funktion zurück
- Hierzu kann man zusätzliche Funktionalität hinzufügen, ohne die ursprüngliche Funktion zu verändern.

Beispiel `decorators.py`

```python
def print_params(func):
    def wrapper(*args, **kwargs):
        print(f"Arguments: {args}, Keyword arguments: {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@print_params
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

# Example usage
result = greet("Alice", greeting="Hi")
print(result)
```

Decorators

Häufige Decorators in Python

- `@abstractmethod`: erzwingt, dass eine Methode in einer abgeleiteten Klasse überschrieben wird
- `@classmethod`: erlaubt es, eine Methode zu definieren, die auf die Klasse und nicht auf die Instanz zugreift und übernimmt `cls`, z.B. wenn auch ein geteiltes Attribut verwendet wird
- `@staticmethod`: erlaubt es, eine Methode zu definieren, die weder auf die Instanz noch auf die Klasse zugreift
- `@property`: erlaubt es, eine Methode zu definieren, die wie ein Attribut aufgerufen wird, also die Klammern weggelassen werden können z.B. `np.array.shape`

Klassen-Variablen und -Methoden

- Bisher waren Daten und Methoden immer an Instanzen gebunden
- Klassen-Variablen, wie `counter` werden direkt unterhalb der Klassendefinition definiert und können von jedem Objekt der Klasse gelesen und geändert werden
- Klassen-Methoden, wie `von_typ` sind Methoden, die auf die Klasse und nicht auf die Instanz zugreifen

Beispiel `class_static_methods.py`

```python
class Device:
    counter = 0  # Klassenattribut für die Anzahl der Instanzen

    def __init__(self, name, farbe):
        self.name = name
        self.farbe = farbe
        Device.counter += 1  # Erhöhe den Zähler bei jeder Instanzerstellung

    @classmethod
    def von_typ(cls, typ):  # Alternativer Konstruktor
        if typ == "Laptop":
            return cls("Standard-Laptop", "Silber")
        else:
            return cls("Unbekanntes Gerät", "Weiß")

gerät1 = Device.von_typ("Laptop")
gerät2 = Device.von_typ("Tablet")
gerät3 = Device("Smartphone", "Schwarz")

print("Erstellte Geräte:", Device.counter)  # Ausgabe: 3
```

Serialisierung mit tinyDB

- Hierbei werden Daten in eine JSON-Datei an einem gewissen Pfad geschrieben und wieder ausgelesen
- tinyDB ist eine einfache Datenbank, die das Lesen, Schreiben und Suchen für uns vereinfacht
- Vorraussertzung, dass wir unsere Objekte speichern können ist, dass diese als JSON serialisierbar sind
- In der JSON-Datei werden gleiche Objekte als Liste in einer Tabelle (Key) gespeichert

Serialisierung mit tinyDB

Beispiel tiny_db.py

```
import os

from tinydb import TinyDB, Query

class Device():
    # Class variable that is shared between all instances of the class
    db_connector = TinyDB(os.path.join(os.path.dirname(os.path.abspath(__file__)), 'database.json')).table('devices')

    # Constructor
    def __init__(self, device_name : str, managed_by_user_id : str):
        self.device_name = device_name
        # The user id of the user that manages the device
        # We don't store the user object itself, but only the id (as a key)
        self.managed_by_user_id = managed_by_user_id
        self.is_active = True

    def store_data(self):
        print("Storing data...")
        # Check if the device already exists in the database
        DeviceQuery = Query()
        result = self.db_connector.search(DeviceQuery.device_name == self.device_name)
        if result:
            # Update the existing record with the current instance's data
            result = self.db_connector.update(self.__dict__, doc_ids=[result[0].doc_id])
            print("Data updated.")
        else:
            # If the device doesn't exist, insert a new record
            self.db_connector.insert(self.__dict__)
            print("Data inserted.")

if __name__ == "__main__":
    device = Device("Test_Device", "Test_User")
    device.store_data()
```

Erzeugte JSON-Datei database.json

```
{"devices": {"1": {"device_name": "Test_Device", "managed_by_user_id": "Test_User", "is_active": true}}}
```

Serialisierung mit tinyDB

```
db_connector = TinyDB('database.json',
                      storage=serializer).table('devices')
```

Nützliche Hinweise zur Datenbankverbindung

- Achten Sie, darauf, dass Sie mit der richtigen JSON-Datei arbeiten und definieren Sie im Notfall einen absoluten Pfad
- Achten Sie darauf, dass Sie mit der richtigen Table arbeiten
- Nur primitve Datentypen werden mit der `__dict__` Methode automatisch in eine serialisierbare Form gebracht
  → für andere Datentypen müssen Sie einen eigenen Serializer schreiben, wie es in `serializer.py` zum Beispiel mit dem `datetime`-Objekt gemacht wurde
- dieser sollte dann in der TinyDB-Instanz als `storage`-Parameter übergeben werden

Grundgerüst der App

- Klasse `Device` repräsentiert ein Gerät und enthält alle relevanten Daten
- Zudem enthält sie die Methoden, um die Daten zu speichern und zu laden
- Weitere Methoden können hinzugefügt werden, um die Daten zu manipulieren

Beispiel `devices.py`

```python
import os
from tinydb import TinyDB, Query
from serializer import serializer

class Device():
    # Class variable that is shared between all instances of the class
    db_connector = TinyDB(os.path.join(os.path.dirname(os.path.abspath(__file__)), 'database.json'), storage=serializer).table('devices')

    # Constructor
    def __init__(self, device_name : str, managed_by_user_id : str):
        self.device_name = device_name
        # The user id of the user that manages the device
        # We don't store the user object itself, but only the id (as a key)
        self.managed_by_user_id = managed_by_user_id
        self.is_active = True

    # String representation of the class
    def __str__(self):
        return f'Device {self.device_name} ({self.managed_by_user_id})'

    # String representation of the class
    def __repr__(self):
        return self.__str__()

    ...
```

Grundgerüst der App

- Das Speichern vergleicht zunächst, ob das Gerät bereits in der Datenbank existiert und updatet es, falls es existiert
- Das Laden wurde das Klasse-Methoden implementiert, um ein Gerät anhand des Namens zu laden und ruft dann mit `cls(data['device_name'], data['managed_by_user_id'])` den Konstruktor auf

Speichern und Laden von Device-Daten mittels `device.py`

```
def store_data(self):
    print("Storing data...")
    # Check if the device already exists in the database
    DeviceQuery = Query()
    result = self.db_connector.search(DeviceQuery.device_name == self.device_name)
    if result:
        # Update the existing record with the current instance's data
        result = self.db_connector.update(self.__dict__, doc_ids=[result[0].doc_id])
        print("Data updated.")
    else:
        # If the device doesn't exist, insert a new record
        self.db_connector.insert(self.__dict__)
        print("Data inserted.")

# Class method that can be called without an instance of the class to
# construct an instance of the class
@classmethod
def load_data_by_device_name(cls, device_name):
    # Load data from the database and create an instance of the Device class
    DeviceQuery = Query()
    result = cls.db_connector.search(DeviceQuery.device_name == device_name)

    if result:
        data = result[0]
        return cls(data['device_name'], data['managed_by_user_id'])
    else:
        return None
```

Grundgerüst der App

- Alternativ zu Klassenmethoden können auch Funktionen in einem Modul definiert werden
- Hier können auch komplexere Abfragen definiert werden z.B. macht es nicht immer Sinn nur die Namen der Geräte zu laden, sondern auch die zugehörigen User-Objekte oder auch eine Liste von Geräten, um die Wartungsdaten auszulesen
- Listen von Strings als Rückgabewert sind hierbei sinnvoll, da sie direkt in der UI dargestellt werden können

Beispiel Datenbankabfragen mit `queries.py`

```python
import os
from tinydb import TinyDB, Query

def find_devices() -> list:
    """Find all devices in the database."""
    # Define the database connector
    db_connector = TinyDB(os.path.join(os.path.dirname(os.path.abspath(__file__)), 'database.json')).table('devices')
    # Create a query object
    DeviceQuery = Query()
    # Search the database for all devices that are active
    result = db_connector.all()

    # The result is a list of dictionaries, we only want the device names
    if result:
        result = [x["device_name"] for x in result]

    return result
```

Grundgerüst der App

- Einfaches Beispiel, wie die Geräte in der UI dargestellt und der zugehörige User geändert werden kann
- Neu ist hierbei, dass wird im `st.session_state`-Dictionary das aktuelle Gerät als Objekt speichern und damit alle Attribute und Methoden des Geräts zur Verfügung haben

Beispiel `ui_device.py`

```
# Eine Auswahlbox mit Datenbankabfrage, das Ergebnis wird in current_device gespeichert
devices_in_db = find_devices()

if devices_in_db:
    current_device_name = st.selectbox(
        'Gerät auswählen',
        options=devices_in_db, key="sbDevice")

    if current_device_name in devices_in_db:
        loaded_device = Device.find_by_attribute("device_name", current_device_name)
        if loaded_device:
            st.write(f"Loaded Device: {loaded_device}") # nutzt __str__ Methode
        else:
            st.error("Device not found in the database.")

        with st.form("Device"):
            st.write(loaded_device.device_name) # Direkter Zugriff auf die Attribute

            text_input_val = st.text_input("Geräte-Verantwortlicher", value=loaded_device.managed_by_user_id)
            loaded_device.set_managed_by_user_id(text_input_val) # Nutzt die Setter-Methode

            # Every form must have a submit button.
            submitted = st.form_submit_button("Submit")
            if submitted:
                loaded_device.store_data()
                st.write("Data stored.")
                st.rerun()
...
```