Projekt Zaliczeniowy z Języków Skryptowych: Kreator i Odtwarzacz Quizów Edukacyjnych

1. Strona Tytułowa

- **Nazwa Uczelni:** Politechnika Świętokrzyska
- **Nazwa Przedmiotu:** Języki Skryptowe
- **Tytuł naukowy oraz imię i nazwisko prowadzącego zajęcia:** Dr inż. Dariusz Michalski
- **Tytuł Projektu:** Kreator i Odtwarzacz Quizów Edukacyjnych
- **Imiona i nazwiska członków zespołu:** Wiktor Pacak, Wojciech Opara
- **Grupa studencka:** 2ID12B
- **Data oddania:** 20.06.2025

2. Opis Projektu

Cel projektu

Głównym celem projektu jest stworzenie prostej, ale funkcjonalnej aplikacji konsolowej w języku Python, która umożliwia użytkownikom interaktywne tworzenie własnych quizów edukacyjnych, a następnie ich odtwarzanie. Aplikacja ma również za zadanie analizować wyniki quizów i wizualizować je w czytelnej formie. Kładziono nacisk na modularyzację, testowanie i obsługę błędów.

Funkcje aplikacji

- **Tworzenie quizów**: Użytkownik może zdefiniować tytuł i opcjonalny opis quizu, a następnie interaktywnie dodawać do niego pytania, opcje odpowiedzi oraz wskazać poprawną odpowiedź.
- **Edycja quizów**: Aplikacja umożliwia wczytanie istniejącego quizu i modyfikację jego tytułu, opisu, treści pytań, opcji odpowiedzi oraz poprawnych odpowiedzi, a także dodawanie i usuwanie pytań.
- **Zapisywanie/Wczytywanie quizów**: Quizy są serializowane do plików w formacie JSON i mogą być z nich wczytywane do ponownego wykorzystania. Obsługa nadpisywania istniejących plików.
- **Odtwarzanie quizów**: Użytkownik może wybrać dostępny quiz z listy, odpowiadać na pytania i na koniec otrzymać szczegółowe podsumowanie swoich wyników.
- **Analiza wyników**: Wyniki odtworzonego quizu są analizowane z wykorzystaniem technik programowania funkcyjnego (m.in. map, filter).
- **Wizualizacja wyników**: Generowany jest wykres kołowy przedstawiający procent poprawnych i niepoprawnych odpowiedzi, który jest automatycznie zapisywany

do pliku graficznego (.png).
- **Obsługa błędów**: Aplikacja zawiera mechanizmy walidacji danych wejściowych od użytkownika oraz kompleksową obsługę błędów podczas operacji na plikach (np. brak pliku, niepoprawny format JSON).

Zakres funkcjonalny (co zostało zaimplementowane)

Projekt w pełni realizuje podstawowe wymagania dotyczące tworzenia, edycji i odtwarzania quizów w środowisku konsolowym. Obejmuje persystencję danych w plikach JSON, rozbudowaną obsługę błędów i walidację, a także wizualizację wyników. Zostały zaimplementowane kluczowe paradygmaty programowania (OOP, programowanie funkcyjne), moduły pomocnicze oraz szeroki zakres testów.

3. Struktura Projektu

Projekt jest zorganizowany w logiczne moduły i pakiety, co ułatwia zarządzanie kodem, jego rozwój i testowanie.

```
quiz_project/
├── main.py              # Główny plik uruchamiający aplikację
├── models/              # Pakiet z definicjami klas modelu
│   ├── __init__.py      # Inicjalizuje pakiet 'models'
│   ├── question.py      # Definiuje klasę Question
│   └── quiz.py          # Definiuje klasę Quiz
├── quiz_data/           # Pakiet do zarządzania danymi quizów
│   ├── __init__.py      # Inicjalizuje pakiet 'quiz_data'
│   └── manager.py       # Klasa QuizDataManager do zapisu/odczytu JSON
├── quiz_creator/        # Pakiet do tworzenia i edycji quizów
│   ├── __init__.py      # Inicjalizuje pakiet 'quiz_creator'
│   └── creator.py       # Klasa QuizCreator z logiką tworzenia/edycji UI
├── quiz_player/         # Pakiet do odtwarzania quizów
│   ├── __init__.py      # Inicjalizuje pakiet 'quiz_player'
│   └── player.py        # Klasa QuizPlayer z logiką rozgrywki i wizualizacji
├── utils/               # Pakiet z ogólnymi funkcjami pomocniczymi
│   ├── __init__.py      # Inicjalizuje pakiet 'utils'
│   └── helpers.py       # Zawiera funkcje rekurencyjne, dekoratory, testy
wydajności/pamięci
├── data/                # Katalog na przykładowe pliki JSON z quizami
│   └── quiz_examples/
│       └── example_quiz.json # Przykładowy plik quizu (tworzony przez aplikację)
├── reports/             # Katalog na wygenerowane raporty graficzne (wykresy)
```

```
├── tests/                # Pakiet z testami jednostkowymi i integracyjnymi
│   ├── __init__.py       # Inicjalizuje pakiet 'tests'
│   ├── test_models.py    # Testy dla Question i Quiz (w tym dziedziczenie w testach)
│   ├── test_data_manager.py # Testy dla QuizDataManager
│   └── test_interactive_modules.py # Testy dla QuizCreator i QuizPlayer (z mockowaniem)
└── requirements.txt      # Lista zależności projektu
```

Krótkie omówienie każdej klasy/modułu

- **main.py**: Główny punkt wejścia do aplikacji. Odpowiada za wyświetlanie menu głównego i kierowanie użytkownika do trybu tworzenia, edycji lub odtwarzania quizu.
- **models/question.py**: Definiuje klasę Question, która jest modelem pojedynczego pytania quizowego. Zawiera atrybuty takie jak treść pytania, opcje odpowiedzi i indeks poprawnej odpowiedzi.
- **models/quiz.py**: Definiuje klasę Quiz, która jest modelem całego quizu. Zawiera tytuł, opis oraz listę obiektów Question.
- **quiz_data/manager.py**: Zawiera klasę QuizDataManager, odpowiedzialną za ładowanie i zapisywanie obiektów Quiz do/z plików JSON. Obsługuje również listowanie dostępnych quizów.
- **quiz_creator/creator.py**: Zawiera klasę QuizCreator, która odpowiada za interaktywne tworzenie nowych quizów oraz edycję istniejących. Zarządza wprowadzaniem danych przez użytkownika, walidacją i przekazywaniem danych do QuizDataManager.
- **quiz_player/player.py**: Zawiera klasę QuizPlayer, odpowiedzialną za odtwarzanie wybranych quizów. Prezentuje pytania, zbiera odpowiedzi, oblicza wynik i generuje wykresy wyników. Wykorzystuje map i filter do analizy danych.
- **utils/helpers.py**: Pakiet ten zawiera zbiór ogólnych funkcji pomocniczych. W projekcie zaimplementowano w nim funkcję rekurencyjną (factorial_recursive), funkcję rekurencyjną do wyboru pytań (select_unique_questions_recursive), dekorator (timing_decorator) oraz przykład dziedziczenia (Shape, Circle), a także funkcję do testów jakości kodu (run_pylint_check).
- **tests/**: Ten pakiet zawiera testy jednostkowe i integracyjne dla wszystkich kluczowych modułów aplikacji, zapewniające ich poprawne działanie i odporność na błędy.

4. Technologie i Biblioteki

- **Python 3.x**: Główny język programowania użyty do realizacji projektu.

- **Standardowe moduły Pythona**:
  - json: Do serializacji i deserializacji danych quizów do/z formatu JSON.
  - os: Do wykonywania operacji na systemie plików (np. sprawdzanie istnienia katalogów, tworzenie ich, łączenie ścieżek).
  - sys: Do manipulacji ścieżką importu w celach testowych.
  - io.StringIO: Używany w testach do przechwytywania wyjścia konsoli.
  - unittest: Wbudowany moduł Pythona do pisania i uruchamiania testów jednostkowych (w tym asercji).
  - unittest.mock: Moduł do tworzenia obiektów mockowych i kontrolowania zachowania zewnętrznych zależności w testach (np. symulowanie input(), mockowanie print(), os.path.exists).
  - random: Do losowego wyboru pytań w funkcji rekurencyjnej.
  - time: Do pomiaru czasu wykonania (w dekoratorze timing_decorator).
  - functools: Zawiera wraps (do dekoratorów) i reduce (do agregacji danych).
  - timeit: Do precyzyjnego mierzenia czasu wykonania małych fragmentów kodu (testy wydajności).
- **Zewnętrzne biblioteki (wymagają instalacji przez pip)**:
  - matplotlib: Służy do generowania graficznych wizualizacji wyników quizów (wykresy kołowe).
  - memory_profiler: Używany do profilowania zużycia pamięci przez funkcje.
  - pylint: Narzędzie do statycznej analizy jakości kodu Python.

5. Sposób Działania Programu

Instrukcja uruchomienia

1. **Sklonuj lub pobierz projekt:**
   git clone https://github.com/Wikierk/quiz-project.git
   cd nazwa_repozytorium # Przejdź do katalogu głównego projektu (np. quiz_project)

2. **Zainstaluj wymagane zależności:**
   Upewnij się, że masz zainstalowanego pip (menedżer pakietów Pythona). Następnie, w katalogu głównym projektu, uruchom:
   pip install -r requirements.txt

3. **Uruchom aplikację:**
   python main.py

Przykładowe dane wejściowe/wyjściowe

Tworzenie nowego quizu:

--- Rozpoczęcie tworzenia nowego quizu ---
Podaj tytuł quizu (np. 'Geografia Polski'): Podstawy Pythona
Podaj krótki opis quizu (opcjonalnie): Quiz sprawdzający podstawową wiedzę o Pythonie.
Quiz 'Podstawy Pythona' został utworzony. Teraz dodaj pytania.

--- Dodawanie nowego pytania ---
Wpisz treść pytania (naciśnij Enter, aby zakończyć dodawanie pytań): Jaki jest wynik 2 + 2?
Wpisuj opcje odpowiedzi. Naciśnij Enter na pustej linii, aby zakończyć dodawanie opcji.
Opcja 1: 3
Opcja 2: 4
Opcja 3: 4
Opcja 4: 5
Opcja 5:
Dostępne opcje:
1. 3
2. 4
3. 5

Wpisz numer poprawnej odpowiedzi: 2
Pytanie dodane pomyślnie!

--- Dodawanie nowego pytania ---
Wpisz treść pytania (naciśnij Enter, aby zakończyć dodawanie pytań): Który kontener jest niezmienny?
Wpisuj opcje odpowiedzi. Naciśnij Enter na pustej linii, aby zakończyć dodawanie opcji.
Opcja 1: Lista
Opcja 2: Słownik
Opcja 3: Krotka
Opcja 4: Zbiór
Opcja 5:
Dostępne opcje:
1. Lista
2. Słownik
3. Krotka
4. Zbiór

Wpisz numer poprawnej odpowiedzi: 3
Pytanie dodane pomyślnie!

--- Dodawanie nowego pytania ---
Wpisz treść pytania (naciśnij Enter, aby zakończyć dodawanie pytań):
Zakończono dodawanie pytań.
Podaj nazwę pliku, pod którą zapisać quiz (bez rozszerzenia .json): python_basics_quiz
Quiz został pomyślnie zapisany!

Naciśnij Enter, aby kontynuować...

Odtwarzanie quizu:

--- Rozpoczęcie odtwarzania quizu ---
Dostępne quizy:
1. python_basics_quiz
Wybierz numer quizu do odtworzenia: 1
Quiz 'Podstawy Pythona' loaded successfully from
data\quiz_examples\python_basics_quiz.json

--- Rozpoczęcie quizu: Podstawy Pythona ---
Opis: Quiz sprawdzający podstawową wiedzę o Pythonie.

--- Pytanie 1/2 ---
Pytanie: Jaki jest wynik 2 + 2?
1. 3
2. 4
3. 5
Wpisz numer odpowiedzi: 2
Poprawna odpowiedź!

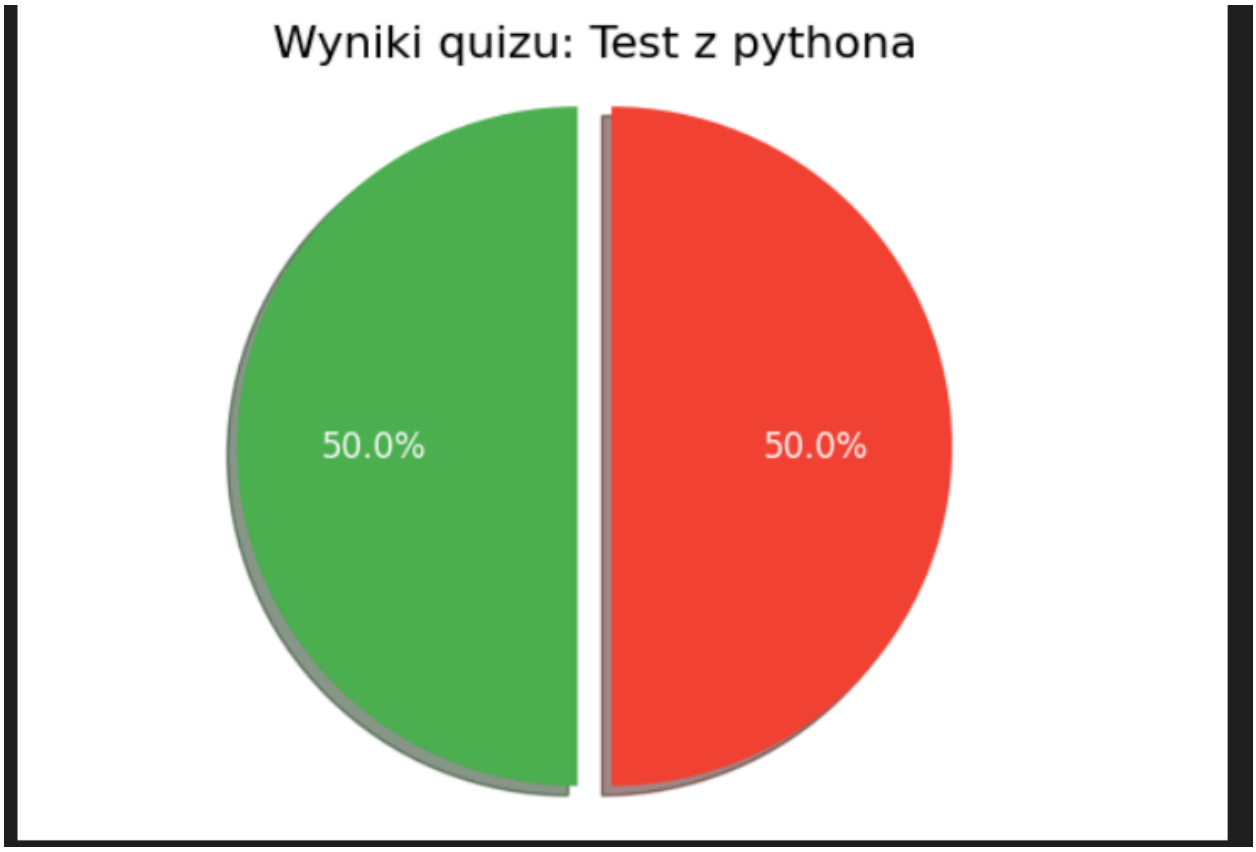
--- Pytanie 2/2 ---
Pytanie: Który kontener jest niezmienny?
1. Lista
2. Słownik
3. Krotka
4. Zbiór
Wpisz numer odpowiedzi: 1
Niepoprawna odpowiedź. Poprawna to: Krotka

--- Koniec quizu! ---
Twój wynik: 1/2 poprawnych odpowiedzi.

Pytań, na które odpowiedziałeś/aś błędnie:
- Który kontener jest niezmienny?

Szczegółowe wyniki zostały zapisane w raporcie graficznym.
Wykres wyników został zapisany w: reports\wyniki_podstawy_pythona.png

Naciśnij Enter, aby kontynuować...



6. Przykłady Kodu (z wyjaśnieniem)

Poniżej przedstawiono wybrane fragmenty kodu z projektu, ilustrujące kluczowe funkcjonalności i wymagania.

Wprowadzanie danych (UI)

Ten fragment ilustruje, jak aplikacja zbiera dane od użytkownika.

```
# quiz_project/quiz_creator/creator.py (fragmenty z metody
QuizCreator.create_new_quiz)

# Wprowadzanie tytułu quizu
title = input("Podaj tytuł quizu (np. 'Geografia Polski'): ").strip()

# Wprowadzanie opisu quizu
description = input("Podaj krótki opis quizu (opcjonalnie): ").strip()

# Wprowadzanie treści pytania
question_text = input("Wpisz treść pytania (naciśnij Enter, aby zakończyć dodawanie
pytań): ").strip()

# Wprowadzanie opcji odpowiedzi (w pętli)
option = input(f"Opcja {option_count}: ").strip()

# Wprowadzanie numeru poprawnej odpowiedzi
user_input = input("Wpisz numer poprawnej odpowiedzi: ").strip()
```

Wyjaśnienie: Powyższe fragmenty kodu z QuizCreator.create_new_quiz demonstrują zbieranie danych od użytkownika (tytuł, opis, pytania, opcje, poprawna odpowiedź) za pomocą funkcji input().

Wyświetlanie danych (UI)

Ten fragment pokazuje, jak aplikacja prezentuje informacje użytkownikowi.

```
# quiz_project/quiz_player/player.py (fragmenty z metod QuizPlayer.play_quiz)

# Wyświetlanie dostępnych quizów
print("Dostępne quizy:")
for i, quiz_name in enumerate(available_quizzes):
```

```
    print(f"  {i + 1}. {quiz_name}")

# Wyświetlanie tytułu i opisu quizu
print(f"\n--- Rozpoczęcie quizu: {quiz.title} ---")
if quiz.description:
    print(f"Opis: {quiz.description}")

# Wyświetlanie pytania i opcji (metoda Question.display() wewnętrznie używa print)
print(f"\n--- Pytanie {i + 1}/{total_questions} ---")
print(question.display()) # Gdzie question.display() to:
# Pytanie: {self.question_text}\n  1. {opcja1}\n  2. {opcja2}\n...

# Wyświetlanie wyników końcowych
print("\n--- Koniec quizu! ---")
print(f"Twój wynik: {correct_answers_count}/{total_questions} poprawnych
odpowiedzi.")
```

Wyjaśnienie: Powyższe fragmenty kodu ilustrują **wyświetlanie danych** w aplikacji. Używana jest funkcja print() do prezentowania użytkownikowi dostępnych quizów, tytułu i opisu wybranego quizu, a także poszczególnych pytań wraz z opcjami odpowiedzi (za pośrednictwem metody Question.display()). Na zakończenie quizu wyświetlane jest podsumowanie wyników.

Zmiana danych (UI)

Ten fragment kodu ilustruje możliwość edycji istniejących quizów, co stanowi zmianę danych w interfejsie użytkownika.

```
# quiz_project/quiz_creator/creator.py (fragment z metody
QuizCreator.edit_existing_quiz)

        # ... (kod wyboru quizu do edycji) ...

        # Główna pętla edycji
        while True:
            # ... (wyświetlanie menu edycji) ...
            edit_choice = input("Wybierz opcję edycji (1-6): ").strip()

            if edit_choice == '1': # Edycja tytułu i opisu
```

```
                new_title = input(f"Nowy tytuł quizu (obecny: '{quiz_to_edit.title}'): ").strip()
                if new_title:
                    quiz_to_edit.title = new_title
                # ... (logika dla opisu) ...

            elif edit_choice == '3': # Edycja istniejącego pytania
                # ... (wyświetlanie pytań i wybór pytania do edycji) ...
                question_to_edit = quiz_to_edit.questions[q_index]

                # Edycja treści pytania
                new_q_text = input(f"Nowa treść pytania (obecna:
'{question_to_edit.question_text}'): ").strip()
                if new_q_text:
                    question_to_edit.question_text = new_q_text

                # Edycja opcji odpowiedzi
                print("\nEdytuj opcje odpowiedzi. Naciśnij Enter, aby pozostawić bez
zmian.")
                new_options = []
                for i, opt in enumerate(question_to_edit.options):
                    edited_opt = input(f"Opcja {i+1} (obecna: '{opt}'): ").strip()
                    new_options.append(edited_opt if edited_opt else opt)

                # Edycja poprawnej odpowiedzi
                new_correct_input = input(f"Nowy numer poprawnej odpowiedzi (obecny:
{current_correct + 1}): ").strip()
                # ... (konwersja i walidacja) ...
            # ... (inne opcje, zapis) ...
```

Wyjaśnienie: Powyższy fragment kodu z metody QuizCreator.edit_existing_quiz ilustruje, jak aplikacja umożliwia **zmianę danych** w interfejsie użytkownika. Użytkownik może wybrać istniejący quiz, a następnie skorzystać z menu edycji, aby zaktualizować jego **tytuł i opis**, **treść istniejącego pytania**, a także **opcje odpowiedzi** i **numer poprawnej odpowiedzi** dla wybranego pytania. Zmiany te są wprowadzane interaktywnie za pomocą funkcji input() i bezpośrednio modyfikują dane obiektów Quiz i Question w pamięci.

Wyszukiwanie danych (UI)

Ten fragment kodu pokazuje, jak aplikacja pozwala użytkownikowi "wyszukiwać" dostępne quizy.

```
# quiz_project/quiz_player/player.py (fragment metody QuizPlayer.play_quiz)

class QuizPlayer:
    # ... (inne metody) ...

    @staticmethod
    def play_quiz():
        """
        Allows the user to select and play an existing quiz.
        """
        # Wyszukiwanie (listowanie) dostępnych quizów
        available_quizzes = QuizDataManager.list_available_quizzes()

        if not available_quizzes:
            print("Brak dostępnych quizów. Najpierw utwórz quiz w trybie kreatora.")
            return

        print("Dostępne quizy:")
        for i, quiz_name in enumerate(available_quizzes):
            print(f"  {i + 1}. {quiz_name}")

        # ... (kod wyboru quizu przez użytkownika) ...
```

Wyjaśnienie: Powyższy fragment kodu z modułu quiz_player ilustruje funkcjonalność **wyszukiwania danych**. Metoda QuizDataManager.list_available_quizzes() przegląda określony katalog i zwraca listę dostępnych plików quizów. Ta lista jest następnie prezentowana użytkownikowi, co pozwala mu "wyszukać" i wybrać konkretny quiz do dalszej interakcji.

Przedstawienie wyników (UI)

Ten fragment ilustruje, jak aplikacja prezentuje wyniki quizu użytkownikowi.

```
# quiz_project/quiz_player/player.py (fragmenty z metody QuizPlayer.play_quiz i generate_and_save_results_chart)
```

```
class QuizPlayer:
    # ... (inne metody i kod) ...

    @staticmethod
    def play_quiz():
        # ... (logika odtwarzania pytań i zbierania odpowiedzi) ...

        print("\n--- Koniec quizu! ---")
        # Prezentacja ogólnego wyniku tekstowo
        print(f"Twój wynik: {correct_answers_count}/{total_questions} poprawnych odpowiedzi.")

        # ... (analiza wyników z map/filter) ...

        # Wizualizacja i zapis wykresu wyników
        QuizPlayer.generate_and_save_results_chart(num_correct, num_incorrect, quiz.title)

        print("\nSzczegółowe wyniki zostały zapisane w raporcie graficznym.")

    @staticmethod
    def generate_and_save_results_chart(correct_count: int, incorrect_count: int, quiz_title: str):
        """
        Generates a pie chart showing the distribution of correct vs. incorrect answers.
        """
        # ... (kod generujący wykres matplotlib) ...
        print(f"Wykres wyników został zapisany w: {chart_filepath}")
```

Wyjaśnienie: Powyższy fragment kodu z modułu quiz_player ilustruje **przedstawienie wyników**. Po zakończeniu quizu aplikacja wyświetla ogólny wynik użytkownika w formie tekstowej. Dodatkowo, generowany jest wizualny raport w postaci wykresu kołowego (z wykorzystaniem matplotlib), który jest zapisywany do pliku graficznego, informując użytkownika o lokalizacji zapisanego raportu.

Zmienne (Podstawy)

Ten fragment ilustruje użycie zmiennych do przechowywania danych.

```
# models/question.py (fragment)
class Question:
    def __init__(self, question_text: str, options: list, correct_answer_index: int):
        self.question_text = question_text.strip() # Zmienna instancji
        self.options = [opt.strip() for opt in options] # Zmienna instancji (lista)
        self.correct_answer_index = correct_answer_index # Zmienna instancji

# quiz_creator/creator.py (fragment z metody create_new_quiz)
class QuizCreator:
    @staticmethod
    def create_new_quiz():
        title = input("Podaj tytuł quizu...").strip() # Zmienna lokalna
        description = input("Podaj krótki opis...").strip() # Zmienna lokalna
        new_quiz = Quiz(title, description) # Zmienna lokalna, przechowująca obiekt
```

Wyjaśnienie: Zmienne takie jak question_text, options, title, description są wykorzystywane do przechowywania stringów, list, intów oraz obiektów, co jest fundamentalne dla operacji w całym projekcie.

Typy danych (Podstawy)

Ten fragment ilustruje użycie różnych typów danych.

```
# models/question.py (fragmenty)
class Question:
    def __init__(self, question_text: str, options: list, correct_answer_index: int):
        self.question_text = question_text.strip() # Typ danych: str
        self.options = [opt.strip() for opt in options] # Typ danych: list
        self.correct_answer_index = correct_answer_index # Typ danych: int

# quiz_data/manager.py (fragment)
class QuizDataManager:
    @staticmethod
    def save_quiz(quiz: Quiz, filename: str):
        if not isinstance(quiz, Quiz): # Typ danych: bool
            raise TypeError("Only Quiz objects can be saved.")
```

Wyjaśnienie: Projekt wykorzystuje typy danych takie jak str (treść pytań), int (indeksy), list (opcje, pytania), dict (dane odpowiedzi) i bool (wartości logiczne warunków), co jest podstawą reprezentacji i przetwarzania informacji.

Komentarze (Podstawy)

Ten fragment ilustruje użycie komentarzy w kodzie.

```
# models/question.py (fragmenty)
class Question:
    """
    Represents a single question in a quiz. # Docstring dla klasy
    """
    def __init__(self, question_text: str, options: list, correct_answer_index: int):
        # Sprawdzanie, czy tekst pytania nie jest pusty # Komentarz jednowierszowy
        if not isinstance(question_text, str) or not question_text.strip():
            raise ValueError("Question text cannot be empty.")

# quiz_data/manager.py (fragment)
class QuizDataManager:
    @staticmethod
    def save_quiz(quiz: Quiz, filename: str):
        """
        Saves a Quiz object to a JSON file. # Docstring dla metody
        """
        if not filename.endswith(".json"):
            filename += ".json" # Zapewnienie rozszerzenia .json
```

Wyjaśnienie: Komentarze (jednowierszowe # i wieloliniowe docstringi """...""") są używane do wyjaśniania przeznaczenia klas, metod, funkcji oraz złożonych fragmentów kodu, zwiększając jego czytelność i ułatwiając przyszłe modyfikacje.

Operatory (Podstawy)

Ten fragment ilustruje użycie różnych operatorów.

```
# models/question.py (fragmenty)
class Question:
    def __init__(self, question_text: str, options: list, correct_answer_index: int):
```

```
        self.question_text = question_text.strip() # Operator przypisania (=)
        if not isinstance(question_text, str) or not question_text.strip(): # Operator
logiczny (or), porównania (!=)
            raise ValueError("Question text cannot be empty.")
        if not (0 <= correct_answer_index < len(options)): # Operatory porównania (<=, <),
logiczny (and)
            raise ValueError("Invalid index.")

    def is_correct(self, user_answer_index: int) -> bool:
        return user_answer_index == self.correct_answer_index # Operator porównania
(==)

# quiz_player/player.py (fragmenty)
class QuizPlayer:
    @staticmethod
    def play_quiz():
        correct_answers_count = 0
        correct_answers_count += 1 # Operator arytmetyczny (+=)
        answer_index = int(user_input) - 1 # Operator arytmetyczny (-)
        print(f"Twój wynik: {correct_answers_count}/{total_questions}") # Operator
formatowania f-string
```

Wyjaśnienie: W projekcie wykorzystano operatory przypisania (=, +=), arytmetyczne (-), porównania (==, !=, <, <=), logiczne (and, or, not) oraz formatowania f-string, co jest podstawą do wykonywania obliczeń, porównań i budowania dynamicznych komunikatów.

Instrukcje warunkowe (if, elif, else) (Podstawy)

Ten fragment ilustruje użycie instrukcji warunkowych.

```
# quiz_project/main.py (fragment z funkcji main)
def main():
    while True:
        choice = input("Wybierz opcję (1-4): ").strip()
        if choice == '1': # Warunek: jeśli wybór to '1'
            QuizCreator.create_new_quiz()
        elif choice == '2': # Warunek: jeśli wybór to '2'
            QuizPlayer.play_quiz()
```

```
        elif choice == '3': # Warunek: jeśli wybór to '3'
            QuizCreator.edit_existing_quiz()
        else: # Warunek: jeśli żaden z powyższych nie jest spełniony
            print("Nieprawidłowy wybór.")

# models/question.py (fragment z konstruktora Question)
class Question:
    def __init__(self, question_text: str, options: list, correct_answer_index: int):
        if not isinstance(question_text, str): # Warunek: walidacja typu
            raise ValueError("Invalid type.")
```

Wyjaśnienie: Instrukcje if, elif, else są szeroko wykorzystywane do kontrolowania przepływu programu, np. w menu głównym do obsługi wyboru użytkownika, w konstruktorach klas do walidacji danych wejściowych, a także w logice quizu do sprawdzania poprawności odpowiedzi i obsługi błędów.

Instrukcje iteracyjne (for, while) (Podstawy)

Ten fragment ilustruje użycie instrukcji iteracyjnych.

```
# quiz_project/quiz_player/player.py (fragmenty)
class QuizPlayer:
    @staticmethod
    def play_quiz():
        # Pętla 'for' do iteracji przez dostępne quizy
        for i, quiz_name in enumerate(available_quizzes):
            print(f"  {i + 1}. {quiz_name}")

        # Pętla 'for' do iteracji przez pytania w quizie
        for i, question in enumerate(quiz.questions):
            # ... logika zadawania pytań ...

# quiz_creator/creator.py (fragmenty)
class QuizCreator:
    @staticmethod
    def create_new_quiz():
        # Pętla 'while' do wymuszenia poprawnego tytułu
        while True:
            title = input("Podaj tytuł quizu...").strip()
```

```
        if title: break
        else: print("Tytuł nie może być pusty.")

    # Pętla 'while' do zbierania opcji odpowiedzi
    while True:
        option = input(f"Opcja {option_count}: ").strip()
        if not option: break
        # ... logika dodawania opcji ...
```

Wyjaśnienie: W projekcie wykorzystano pętle for do iterowania przez kolekcje (np. listy quizów, pytań w quizie, opcji odpowiedzi), oraz pętle while do kontrolowania przepływu programu na podstawie warunków (np. pętle menu głównego, pętle wymuszające poprawne wprowadzenie danych).

Operacje wejścia (input) (Podstawy)

Ten fragment ilustruje, jak aplikacja zbiera dane wejściowe od użytkownika.

```
# quiz_project/main.py (fragment z funkcji main)
choice = input("Wybierz opcję (1-4): ").strip()

# quiz_project/quiz_creator/creator.py (fragmenty)
title = input("Podaj tytuł quizu...").strip()
question_text = input("Wpisz treść pytania...").strip()
user_input = input("Wpisz numer poprawnej odpowiedzi: ").strip()

# quiz_project/quiz_player/player.py (fragmenty)
choice = input("Wybierz numer quizu do odtworzenia: ").strip()
user_input = input("Wpisz numer odpowiedzi: ").strip()
```

Wyjaśnienie: Funkcja input() jest kluczowa dla interakcji z użytkownikiem w trybie konsolowym, umożliwiając zbieranie różnego rodzaju danych: wybory z menu, tytuły i opisy quizów, treści pytań, opcje odpowiedzi, indeksy poprawnych odpowiedzi, a także potwierdzenia działań.

Operacje wyjścia (print) (Podstawy)

Ten fragment pokazuje, jak aplikacja wyświetla informacje użytkownikowi.

```
# quiz_project/main.py (fragmenty)
```

```
print("Witaj w Aplikacji Quizowej!")
print("\n--- Menu Główne ---")
print("1. Utwórz nowy quiz")
print("Dziękujemy za skorzystanie z aplikacji. Do widzenia!")

# quiz_project/quiz_creator/creator.py (fragmenty)
print("\n--- Rozpoczęcie tworzenia nowego quizu ---")
print(f"Quiz '{title}' został utworzony.")
print("Pytanie dodane pomyślnie!")

# quiz_project/quiz_player/player.py (fragmenty)
print("Dostępne quizy:")
print(f"--- Rozpoczęcie quizu: {quiz.title} ---")
print("Poprawna odpowiedź!")
print(f"Twój wynik: {correct_answers_count}/{total_questions} poprawnych
odpowiedzi.")
```

Wyjaśnienie: Funkcja print() jest intensywnie wykorzystywana do komunikacji z użytkownikiem, wyświetlając wiadomości powitalne, menu, instrukcje, potwierdzenia operacji, wyniki quizu oraz komunikaty o błędach.

Funkcje z parametrami i wartościami zwracanymi (Podstawy)

Ten fragment ilustruje funkcje, które przyjmują parametry i zwracają wartości.

```
# models/question.py (fragmenty)
class Question:
    def is_correct(self, user_answer_index: int) -> bool:
        return user_answer_index == self.correct_answer_index

# quiz_data/manager.py (fragmenty)
class QuizDataManager:
    @staticmethod
    def list_available_quizzes(directory: str = "data/quiz_examples") -> list[str]:
        if not os.path.exists(directory): return []
        # ... (logika listowania plików) ...
        return quiz_files
```

Wyjaśnienie: Wiele funkcji i metod w projekcie (is_correct, list_available_quizzes)

przyjmuje jeden lub więcej parametrów (np. user_answer_index, directory) i zwraca wynik swojego działania (np. bool dla poprawności odpowiedzi, list[str] dla listy quizów), co zapewnia modułowość i ponowne użycie kodu.

Funkcje rekurencyjne (Podstawy)

Ten fragment ilustruje zaimplementowaną funkcję rekurencyjną.

```
# quiz_project/utils/helpers.py
import random

def select_unique_questions_recursive(
    all_questions: list,
    num_to_select: int,
    _selected_indices: set = None,
    _result_questions: list = None
) -> list:
    if _selected_indices is None: _selected_indices = set()
    if _result_questions is None: _result_questions = []
    if len(_result_questions) == num_to_select: # Warunek bazowy 1
        return _result_questions
    if len(_selected_indices) == len(all_questions): # Warunek bazowy 2
        return _result_questions

    available_to_pick_indices = [i for i in range(len(all_questions)) if i not in
_selected_indices]
    chosen_index = random.choice(available_to_pick_indices)
    _selected_indices.add(chosen_index)
    _result_questions.append(all_questions[chosen_index])

    return select_unique_questions_recursive( # Wywołanie rekurencyjne
        all_questions, num_to_select, _selected_indices=_selected_indices,
_result_questions=_result_questions
    )
```

Wyjaśnienie: Funkcja select_unique_questions_recursive losowo wybiera unikalne pytania. Posiada warunki bazowe do zakończenia rekursji (osiągnięcie pożądanej liczby pytań lub wyczerpanie dostępnych) oraz krok rekurencyjny, w którym funkcja wywołuje samą siebie z zaktualizowanymi argumentami.

Funkcje przyjmujące inne funkcje jako argumenty (Podstawy)

Ten fragment ilustruje użycie funkcji wyższego rzędu.

```
# quiz_player/player.py (fragment metody QuizPlayer.play_quiz)
        # Filter dla poprawnych odpowiedzi
        correct_results = list(filter(lambda ans: ans["is_correct"], user_answers))

        # Filter dla niepoprawnych odpowiedzi
        incorrect_results = list(filter(lambda ans: not ans["is_correct"], user_answers))

        # Użycie map do pobrania treści pytań
        incorrect_question_texts = list(map(lambda ans: ans["question_text"],
incorrect_results))
```

Wyjaśnienie: Funkcje filter() i map() są używane jako funkcje wyższego rzędu, przyjmując funkcje lambda jako argumenty. filter() selekcjonuje elementy na podstawie warunku z lambda, a map() transformuje elementy za pomocą funkcji lambda, co jest przykładem programowania funkcyjnego.

Dekoratory (Podstawy)

Ten fragment ilustruje zaimplementowany dekorator.

```
# quiz_project/utils/helpers.py (fragment)
import time
from functools import wraps

def timing_decorator(func):
    """A decorator that measures the execution time of a function."""
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        execution_time = end_time - start_time
        print(f"[{func.__name__}] Czas wykonania: {execution_time:.4f} sekund")
        return result
    return wrapper
```

Wyjaśnienie: Dekorator timing_decorator przyjmuje funkcję jako argument i zwraca nową funkcję (wrapper), która mierzy czas wykonania oryginalnej funkcji, a następnie go wyświetla. Użycie @wraps(func) pozwala zachować metadane dekorowanej funkcji.

Użycie listy (Kontenery)

Ten fragment ilustruje użycie list.

```
# models/question.py (fragment)
class Question:
    def __init__(self, question_text: str, options: list, correct_answer_index: int):
        self.options = [opt.strip() for opt in options] # Lista opcji odpowiedzi

# models/quiz.py (fragment)
class Quiz:
    def __init__(self, title: str, questions: list = None):
        self.questions = [] # Lista obiektów Question
        if questions:
            for q in questions: self.questions.append(q)

    def add_question(self, question: Question):
        self.questions.append(question) # Dodawanie do listy
```

Wyjaśnienie: Listy są wykorzystywane do przechowywania uporządkowanych kolekcji elementów, takich jak opcje odpowiedzi w pytaniu (Question.options), pytania w quizie (Quiz.questions) oraz szczegóły odpowiedzi użytkownika (user_answers w QuizPlayer).

Użycie słownika (Kontenery)

Ten fragment ilustruje użycie słowników.

```
# models/question.py (fragment z metody to_dict)
class Question:
    def to_dict(self) -> dict:
        return { # Słownik do serializacji JSON
            "question_text": self.question_text,
            "options": self.options,
            "correct_answer_index": self.correct_answer_index
```

```
        }

# quiz_player/player.py (fragment)
class QuizPlayer:
    @staticmethod
    def play_quiz():
        user_answers = []
        user_answers.append({ # Słownik do przechowywania szczegółów odpowiedzi
            "question_text": "Q1 text",
            "is_correct": True
        })
```

Wyjaśnienie: Słowniki są kluczowe do reprezentacji ustrukturyzowanych danych, takich jak obiekty Question w formacie JSON (metoda to_dict()) oraz szczegóły odpowiedzi użytkownika, gdzie pary klucz-wartość efektywnie przechowują powiązane informacje.

Użycie zbioru (Kontenery)

Ten fragment ilustruje użycie zbioru.

```
# quiz_project/utils/helpers.py (fragment z funkcji select_unique_questions_recursive)
def select_unique_questions_recursive(
    all_questions: list,
    num_to_select: int,
    _selected_indices: set = None, # Zbiór indeksów już wybranych pytań
    _result_questions: list = None
) -> list:
    if _selected_indices is None:
        _selected_indices = set() # Inicjalizacja pustego zbioru

    # ...
    if i not in _selected_indices: # Sprawdzanie unikalności
        pass
    # ...
    _selected_indices.add(chosen_index) # Dodanie do zbioru
```

Wyjaśnienie: Zbiór _selected_indices jest używany w funkcji rekurencyjnej do losowego wyboru unikalnych pytań. Dzięki właściwości zbiorów (przechowują tylko

unikalne elementy), zapewnia się, że żadne pytanie nie zostanie wybrane dwukrotnie.

Użycie krotki (Kontenery)

Ten fragment ilustruje użycie krotki.

```
# quiz_project/quiz_player/player.py (fragment z metody
generate_and_save_results_chart)
class QuizPlayer:
    @staticmethod
    def generate_and_save_results_chart(correct_count: int, incorrect_count: int,
quiz_title: str):
        labels: tuple[str, str] = ('Poprawne', 'Niepoprawne') # Krotka etykiet
        colors: tuple[str, str] = ('#4CAF50', '#F44336') # Krotka kolorów
        # ... (kod rysowania wykresu) ...
```

Wyjaśnienie: Krotki labels i colors są używane do przechowywania stałych, niezmiennych wartości (etykiet i kodów kolorów) dla wykresu wyników. Ich niezmienność gwarantuje, że dane te nie zostaną przypadkowo zmodyfikowane w trakcie działania programu.

Zastosowano zmienne lokalne (Przestrzenie nazw)

Ten fragment ilustruje użycie zmiennych lokalnych.

```
# quiz_project/quiz_creator/creator.py (fragmenty)
class QuizCreator:
    @staticmethod
    def create_new_quiz():
        title = input("Podaj tytuł quizu...").strip() # Zmienna lokalna
        description = input("Podaj krótki opis...").strip() # Zmienna lokalna
        new_quiz = Quiz(title, description) # Zmienna lokalna
        # ...

# quiz_project/quiz_player/player.py (fragmenty)
class QuizPlayer:
    @staticmethod
    def play_quiz():
        correct_answers_count = 0 # Zmienna lokalna
        total_questions = len(quiz.questions) # Zmienna lokalna
```

```
    user_answers = [] # Zmienna lokalna
```

Wyjaśnienie: Zmienne takie jak title, description, correct_answers_count są definiowane wewnątrz funkcji i metod. Są one dostępne tylko w obrębie danego bloku kodu, co sprzyja enkapsulacji i minimalizuje ryzyko kolizji nazw.

Zastosowano zmienne globalne (Przestrzenie nazw)

Ten fragment ilustruje użycie zmiennych globalnych (na poziomie modułu).

```
# quiz_project/quiz_player/player.py (fragment)
import matplotlib.pyplot as plt
import os

REPORTS_DIRECTORY = "reports" # Zmienna globalna na poziomie modułu

class QuizPlayer:
  @staticmethod
  def generate_and_save_results_chart(correct_count: int, incorrect_count: int,
quiz_title: str):
    if not os.path.exists(REPORTS_DIRECTORY): # Użycie zmiennej globalnej
      os.makedirs(REPORTS_DIRECTORY)
    chart_filepath = os.path.join(REPORTS_DIRECTORY, chart_filename) # Użycie
zmiennej globalnej
```

Wyjaśnienie: Zmienna REPORTS_DIRECTORY jest zdefiniowana na najwyższym poziomie modułu quiz_player.py. Jest to zmienna globalna w ramach tego modułu, co pozwala na dostęp do niej z dowolnej funkcji czy metody w tym pliku, zapewniając spójność w ścieżkach zapisu raportów.

Zastosowano zakresy funkcji (Przestrzenie nazw)

Ten fragment ilustruje zastosowanie zakresów funkcji.

```
# quiz_project/quiz_creator/creator.py (fragmenty)
class QuizCreator:
  @staticmethod
  def create_new_quiz():
    title = input("Podaj tytuł quizu...").strip() # 'title' lokalna dla create_new_quiz
    # ...
```

```
        QuizCreator._add_questions_to_quiz(new_quiz)
        # 'title' jest dostępna tutaj, ale zmienne z _add_questions_to_quiz nie są

    @staticmethod
    def _add_questions_to_quiz(quiz: Quiz):
        question_text = input("Wpisz treść pytania...").strip() # 'question_text' lokalna dla
_add_questions_to_quiz
        options = [] # 'options' lokalna dla _add_questions_to_quiz
        # ...
        # 'question_text' i 'options' są dostępne tylko tutaj
```

Wyjaśnienie: Każda funkcja (create_new_quiz, _add_questions_to_quiz) tworzy swój własny zakres, w którym zmienne lokalne są dostępne i odizolowane od zmiennych w innych funkcjach. Promuje to modułowość i zapobiega nieoczekiwanym interakcjom między różnymi częściami kodu.

Zastosowano zakresy klas (Przestrzenie nazw)

Ten fragment ilustruje zastosowanie zakresów klas.

```
# models/question.py (fragment)
class Question:
    question_text: str # Atrybut klasy/instancji, dostępny w jej zakresie

    def __init__(self, question_text: str, options: list, correct_answer_index: int):
        self.question_text = question_text.strip() # Odwołanie do atrybutu w zakresie
klasy

    def is_correct(self, user_answer_index: int) -> bool:
        return user_answer_index == self.correct_answer_index # Metoda w zakresie
klasy

# models/quiz.py (fragment)
class Quiz:
    title: str # Atrybut klasy/instancji
    questions: list # Atrybut klasy/instancji

    def add_question(self, question: Question):
        self.questions.append(question) # Modyfikacja atrybutu w zakresie klasy
```

Wyjaśnienie: Atrybuty (question_text, options) i metody (display, is_correct) zdefiniowane wewnątrz klas Question i Quiz są dostępne w ich własnym zakresie. Pozwala to na dostęp do danych i zachowań obiektu za pomocą self, zapewniając spójność obiektową.

Projekt podzielony na moduły (import, __init__) (Moduły i pakiety)

Ten fragment ilustruje strukturę modułów i pakiety oraz sposób importowania.

```
# Przykładowa struktura katalogów i plików:
# quiz_project/
# ├── main.py
# ├── models/
# │   ├── __init__.py  # Inicjalizuje pakiet
# │   └── question.py
# └── quiz_data/
#     ├── __init__.py  # Inicjalizuje pakiet
#     └── manager.py

# Przykład importu:
# main.py
from quiz_creator.creator import QuizCreator
from quiz_player.player import QuizPlayer

# quiz_data/manager.py
from models.question import Question
from models.quiz import Quiz
```

Wyjaśnienie: Projekt jest podzielony na logiczne pliki .py (moduły) i katalogi zawierające __init__.py (pakiety), co umożliwia modularność. Instrukcje import (np. from pakiet.moduł import Klasa) pozwalają na łączenie tych modułów i pakietów, tworząc spójną strukturę aplikacji.

Własne pakiety/funkcje pomocnicze w osobnych plikach .py (Moduły i pakiety)

Ten fragment ilustruje umieszczenie własnych funkcji pomocniczych.

```
# quiz_project/utils/helpers.py (fragment)
import random
```

```
import time

def factorial_recursive(n: int) -> int:
    """Calculates the factorial recursively."""
    # ... (kod funkcji) ...

def timing_decorator(func):
    """A decorator that measures execution time."""
    # ... (kod dekoratora) ...
```

Wyjaśnienie: Pakiet utils zawiera moduł helpers.py, w którym zdefiniowano ogólne funkcje pomocnicze, takie jak factorial_recursive (funkcja rekurencyjna) i timing_decorator (dekorator). Oddziela to kod pomocniczy od głównych modułów aplikacji, poprawiając organizację i możliwość ponownego użycia.

Obsługa wyjątków (try, except, finally) (Obsługa błędów)

Ten fragment ilustruje obsługę wyjątków.

```
# quiz_data/manager.py (fragment metody QuizDataManager.load_quiz)
    @staticmethod
    def load_quiz(filename: str):
        # ...
        try: # Blok 'try'
            with open(file_path, 'r', encoding='utf-8') as f:
                quiz_data = json.load(f)
            # ...
        except json.JSONDecodeError as e: # Obsługa konkretnego wyjątku
            print(f"Error decoding JSON: {e}")
            raise
        except KeyError as e: # Obsługa innego konkretnego wyjątku
            print(f"Missing required data: {e}")
            raise
        except Exception as e: # Ogólna obsługa innych wyjątków
            print(f"An unexpected error occurred: {e}")
            raise
        finally: # Blok 'finally' - zawsze wykonywany
            pass
```

Wyjaśnienie: Aplikacja stosuje bloki try-except-finally do zarządzania błędami. Na przykład, podczas ładowania quizu z pliku JSON, obsługiwane są specyficzne wyjątki takie jak json.JSONDecodeError (nieprawidłowy format JSON) i KeyError (brakujące dane), a także ogólne Exception. Blok finally zapewnia wykonanie kodu czyszczącego zasoby.

Użycie assert do testów i walidacji (Obsługa błędów)

Ten fragment ilustruje użycie assert w testach.

```
# tests/test_models.py (fragmenty)
import unittest

class TestQuestion(unittest.TestCase):
    def test_question_initialization_valid(self):
        question = Question("What is 2+2?", ["3", "4"], 1)
        self.assertEqual(question.question_text, "What is 2+2?") # Asercja równości

    def test_question_initialization_empty_text_raises_error(self):
        with self.assertRaisesRegex(ValueError, "Question text cannot be empty."):
            Question("", ["a", "b"], 0) # Asercja rzucenia wyjątku
```

Wyjaśnienie: W testach jednostkowych wykorzystuje się metody asercji (self.assertEqual, self.assertRaisesRegex) z modułu unittest.TestCase. Służą one do weryfikacji, czy kod działa zgodnie z oczekiwaniami, w tym czy poprawnie rzuca wyjątki w przypadku nieprawidłowych danych (np. pusty tekst pytania), co jest formą walidacji i testowania przypadków granicznych.

Operacje na stringach (m.in. formatowanie, dzielenie, wyszukiwanie) (Łańcuchy znaków)

Ten fragment ilustruje operacje na stringach.

```
# models/question.py (fragment)
self.question_text = question_text.strip() # Czyszczenie białych znaków

# quiz_creator/creator.py (fragment)
print(f"Quiz '{title}' został utworzony.") # Formatowanie f-stringiem
if not filename.endswith(".json"): # Wyszukiwanie sufiksu
    filename += ".json" # Konkatenacja
```

```
# quiz_player/player.py (fragment)
chart_filename = f"wyniki_{quiz_title.replace(' ', '_').lower()}.png" # Zastępowanie, małe litery
```

Wyjaśnienie: W projekcie stringi są intensywnie przetwarzane. Używane są metody takie jak .strip() (usuwanie białych znaków), .lower() (zmiana na małe litery), .replace() (zamiana podciągów), .endswith() (sprawdzanie sufiksu), a także f-stringi do zaawansowanego formatowania i tworzenia dynamicznych komunikatów.

Odczyt z plików .txt, .csv, .json, .xml (min. 1) (Obsługa plików)

Ten fragment ilustruje odczyt z pliku JSON.

```
# quiz_data/manager.py (fragment metody QuizDataManager.load_quiz)
    @staticmethod
    def load_quiz(filename: str):
        # ...
        with open(file_path, 'r', encoding='utf-8') as f:
            quiz_data = json.load(f) # Odczyt danych z pliku JSON
        # ...
```

Wyjaśnienie: Aplikacja odczytuje dane quizów z plików w formacie JSON. Metoda json.load() służy do deserializacji zawartości pliku na obiekt Pythona (słownik), co pozwala na ponowne wczytanie i wykorzystanie wcześniej zapisanych quizów.

Zapis do plików .txt, .csv, .json, .xml (min. 1) (Obsługa plików)

Ten fragment ilustruje zapis do pliku JSON.

```
# quiz_data/manager.py (fragment metody QuizDataManager.save_quiz)
    @staticmethod
    def save_quiz(quiz: Quiz, filename: str):
        # ...
        with open(file_path, 'w', encoding='utf-8') as f:
            json.dump(quiz_data, f, indent=4, ensure_ascii=False) # Zapis danych do pliku JSON
        # ...
```

Wyjaśnienie: Aplikacja zapisuje obiekty quizów do plików w formacie JSON. Metoda json.dump() serializuje obiekt Pythona (słownik, będący reprezentacją quizu) do pliku tekstowego. Użycie indent=4 zapewnia czytelny format zapisanego JSON-a, a ensure_ascii=False pozwala na poprawne kodowanie znaków specjalnych (np. polskich liter).

Klasy (OOP)

Ten fragment ilustruje definicję klas.

```
# models/question.py (fragment)
class Question:
    """Represents a single question in a quiz."""
    # ...

# models/quiz.py (fragment)
class Quiz:
    """Represents a collection of questions that form a quiz."""
    # ...
```

Wyjaśnienie: Projekt opiera się na programowaniu obiektowym (OOP), definiując klasy Question i Quiz. Question modeluje pojedyncze pytanie z jego atrybutami (treść, opcje, poprawna odpowiedź), a Quiz agreguje obiekty Question wraz z własnymi atrybutami (tytuł, opis). Klasy te służą jako szablony do tworzenia obiektów odzwierciedlających byty problemu.

Metody (OOP)

Ten fragment ilustruje zastosowanie metod.

```
# models/question.py (fragmenty)
class Question:
    def display(self) -> str: # Metoda instancji
        # ...
    def is_correct(self, user_answer_index: int) -> bool: # Metoda instancji z
parametrami
        # ...
    def to_dict(self) -> dict: # Metoda instancji
        # ...
```

```
# models/quiz.py (fragmenty)
class Quiz:
    def add_question(self, question: 'Question'): # Metoda instancji
        # ...
    def remove_question(self, index: int): # Metoda instancji
        # ...
```

Wyjaśnienie: W klasach Question i Quiz zaimplementowano liczne metody, które definiują zachowanie obiektów. Przykłady to display() (formatowanie tekstu), is_correct() (sprawdzanie poprawności), to_dict() (serializacja danych), add_question() i remove_question() (modyfikacja stanu quizu). Metody te operują na atrybutach (self.atrybut) i są kluczowe dla obiektowego podejścia.

Konstruktory (OOP)

Ten fragment ilustruje zastosowanie konstruktorów.

```
# models/question.py (fragment)
class Question:
    def __init__(self, question_text: str, options: list, correct_answer_index: int):
        # Konstruktor inicjalizujący atrybuty obiektu
        self.question_text = question_text.strip()
        self.options = [opt.strip() for opt in options]
        self.correct_answer_index = correct_answer_index

# models/quiz.py (fragment)
class Quiz:
    def __init__(self, title: str, description: str = "", questions: list = None):
        # Konstruktor inicjalizujący atrybuty obiektu Quiz
        self.title = title.strip()
        self.description = description.strip() if description else ""
        self.questions = []
```

Wyjaśnienie: Konstruktory (__init__) w klasach Question i Quiz są używane do inicjalizacji nowych obiektów. Przyjmują one niezbędne parametry (np. treść pytania, opcje, tytuł quizu) i przypisują je do atrybutów instancji (self.atrybut), ustawiając początkowy stan obiektu.

Dziedziczenie (OOP)

Ten fragment ilustruje zastosowanie dziedziczenia.

```
# quiz_project/utils/helpers.py (fragment)
class Shape: # Klasa bazowa
    def __init__(self, color: str):
        self.color = color
    def describe(self) -> str:
        return f"To jest {self.color} kształt."

class Circle(Shape): # Klasa pochodna dziedzicząca po Shape
    def __init__(self, color: str, radius: float):
        super().__init__(color) # Wywołanie konstruktora klasy bazowej
        self.radius = radius
    def describe(self) -> str: # Nadpisanie metody z klasy bazowej
        return f"To jest {self.color} koło o promieniu {self.radius}."
```

Wyjaśnienie: Dziedziczenie jest zademonstrowane przez klasy Shape (bazowa) i Circle (pochodna). Circle dziedziczy atrybut color od Shape i wywołuje konstruktor klasy bazowej za pomocą super().__init__(). Nadpisuje również metodę describe(), rozszerzając jej funkcjonalność, co jest kluczowym aspektem polimorfizmu w OOP.

map (Programowanie funkcyjne)

Ten fragment ilustruje użycie funkcji map.

```
# quiz_player/player.py (fragment metody QuizPlayer.play_quiz)
        # Użycie map do pobrania treści pytań, na które odpowiedziano błędnie
        incorrect_question_texts = list(map(lambda ans: ans["question_text"],
incorrect_results))
```

Wyjaśnienie: Funkcja map() jest wykorzystywana do transformacji danych. W tym przypadku, w połączeniu z funkcją lambda, stosuje ona funkcję lambda ans: ans["question_text"] do każdego elementu listy incorrect_results, tworząc nową listę zawierającą wyłącznie teksty pytań, na które użytkownik odpowiedział błędnie.

filter (Programowanie funkcyjne)

Ten fragment ilustruje użycie funkcji filter.

```
# quiz_player/player.py (fragment metody QuizPlayer.play_quiz)
        # Filter dla poprawnych odpowiedzi
        correct_results = list(filter(lambda ans: ans["is_correct"], user_answers))

        # Filter dla niepoprawnych odpowiedzi
        incorrect_results = list(filter(lambda ans: not ans["is_correct"], user_answers))
```

Wyjaśnienie: Funkcja filter() jest używana do selekcjonowania elementów z listy na podstawie warunku. Wraz z funkcjami lambda, filtruje ona listę user_answers, tworząc nowe listy zawierające odpowiednio tylko poprawne lub tylko niepoprawne odpowiedzi, zgodnie z wartością pola is_correct.

lambda (Programowanie funkcyjne)

Ten fragment ilustruje użycie funkcji lambda.

```
# quiz_player/player.py (fragment metody QuizPlayer.play_quiz)
        # Funkcja lambda dla filter
        correct_results = list(filter(lambda ans: ans["is_correct"], user_answers))
        # Funkcja lambda dla map
        incorrect_question_texts = list(map(lambda ans: ans["question_text"],
incorrect_results))
```

Wyjaśnienie: Funkcje lambda są zwięzłymi, anonimowymi funkcjami używanymi jako argumenty dla funkcji wyższego rzędu, takich jak filter() i map(). Pozwalają one na szybkie definiowanie prostych operacji (np. sprawdzanie wartości pola is_correct lub pobieranie question_text) bezpośrednio w miejscu ich użycia, bez konieczności definiowania pełnoprawnych funkcji.

reduce (Programowanie funkcyjne)

Ten fragment ilustruje użycie funkcji reduce.

```
# quiz_project/utils/helpers.py (fragment)
from functools import reduce

def sum_list_elements(numbers: list[int]) -> int:
    return reduce(lambda acc, x: acc + x, numbers, 0) # Sumowanie listy
```

Wyjaśnienie: Funkcja reduce() z modułu functools jest używana do skumulowania elementów listy w jedną wartość poprzez kolejną aplikację funkcji (w tym przypadku sumowania za pomocą lambda acc, x: acc + x). Parametr 0 jest początkową wartością akumulatora.

Wygenerowano wykres (np. matplotlib, seaborn) (Wizualizacja danych)

Ten fragment ilustruje generowanie wykresu.

```
# quiz_player/player.py (fragment metody
QuizPlayer.generate_and_save_results_chart)
import matplotlib.pyplot as plt

    @staticmethod
    def generate_and_save_results_chart(correct_count: int, incorrect_count: int,
quiz_title: str):
        fig1, ax1 = plt.subplots() # Tworzenie figury i osi
        ax1.pie(sizes, explode=explode, labels=labels, colors=colors, # Generowanie
wykresu kołowego
                autopct='%1.1f%%', shadow=True, startangle=90)
        plt.title(f'Wyniki quizu: {quiz_title}') # Ustawienie tytułu
```

Wyjaśnienie: Aplikacja wykorzystuje bibliotekę matplotlib do wizualizacji wyników quizu. Funkcja plt.subplots() tworzy figurę i osie wykresu, a metoda ax1.pie() generuje wykres kołowy, przedstawiający proporcje poprawnych i niepoprawnych odpowiedzi. Tytuł wykresu jest dynamicznie generowany na podstawie nazwy quizu.

Zapisano wykres do pliku graficznego (.png lub .jpg) (Wizualizacja danych)

Ten fragment ilustruje zapis wykresu do pliku.

```
# quiz_player/player.py (fragment metody
QuizPlayer.generate_and_save_results_chart)
import os
import matplotlib.pyplot as plt

    @staticmethod
    def generate_and_save_results_chart(correct_count: int, incorrect_count: int,
quiz_title: str):
        # ... (kod generowania wykresu) ...
```

```
    reports_dir = "reports"
    if not os.path.exists(reports_dir):
        os.makedirs(reports_dir)
    chart_filename = f"wyniki_{quiz_title.replace(' ', '_').lower()}.png"
    chart_filepath = os.path.join(reports_dir, chart_filename)

    plt.savefig(chart_filepath, bbox_inches='tight', dpi=100) # Zapisywanie wykresu
    print(f"Wykres wyników został zapisany w: {chart_filepath}")
```

Wyjaśnienie: Po wygenerowaniu wykresu za pomocą matplotlib, jest on automatycznie zapisywany do pliku graficznego PNG. Kod sprawdza i tworzy katalog reports/ jeśli nie istnieje, a następnie używa plt.savefig() do zapisania pliku na dysku, informując użytkownika o jego lokalizacji.

Testy jednostkowe (assert, unittest, pytest) (Testowanie)

Ten fragment ilustruje użycie unittest i metod asercji.

```
# tests/test_models.py (fragment)
import unittest

class TestQuestion(unittest.TestCase):
    def test_question_initialization_valid(self):
        self.assertEqual(question.question_text, "What is 2+2?") # self.assertEqual

    def test_question_initialization_empty_text_raises_error(self):
        with self.assertRaisesRegex(ValueError, "Question text cannot be empty."): #
self.assertRaisesRegex
            Question("", ["a", "b"], 0)
```

Wyjaśnienie: W projekcie wykorzystano moduł unittest do pisania testów jednostkowych. Klasy testowe dziedziczą z unittest.TestCase, co daje dostęp do metod asercji (self.assertEqual, self.assertTrue, self.assertRaisesRegex) do weryfikacji poprawności działania kodu i obsługi błędów w izolowanych komponentach.

Testy funkcjonalne (Testowanie)

Ten fragment ilustruje testy funkcjonalne.

```
# tests/test_interactive_modules.py (fragment)
```

```
from unittest.mock import patch
from io import StringIO
# ...
class TestQuizCreator(unittest.TestCase):
    @patch('builtins.input', side_effect=['Test Quiz Title', ..., 'test_quiz_output'])
    def test_create_new_quiz_success(self, mock_input):
        QuizCreator.create_new_quiz()
        output = self.held_output.getvalue()
        self.assertIn("Quiz został pomyślnie zapisany!", output)
        self.mock_save_quiz.assert_called_once()
```

Wyjaśnienie: Testy funkcjonalne (w test_interactive_modules.py) symulują pełne przepływy użytkownika (np. tworzenie quizu od początku do końca, odtwarzanie quizu). Wykorzystują mockowanie input() i przechwytywanie print() do weryfikacji, czy cała funkcjonalność działa poprawnie z perspektywy użytkownika końcowego i czy produkuje oczekiwane efekty uboczne (np. zapis quizu).

Testy Integracyjne (Testowanie)

Ten fragment ilustruje testy integracyjne.

```
# tests/test_interactive_modules.py (fragment)
from unittest.mock import patch
# ...
class TestQuizCreator(unittest.TestCase):
    @patch('quiz_data.manager.QuizDataManager.list_available_quizzes',
return_value=["quiz_name"])
    @patch('quiz_data.manager.QuizDataManager.load_quiz')
    @patch('builtins.input', side_effect=['1', '1', 'New Title', 'New Description', '5',
'quiz_name', 'tak'])
    def test_edit_quiz_title_and_description(self, mock_input, mock_load, mock_list):
        sample_quiz = self._create_sample_quiz_file_for_editing()
        mock_load.return_value = sample_quiz
        QuizCreator.edit_existing_quiz()
        self.mock_save_quiz.assert_called_once() # Sprawdza interakcję z mockiem
QuizDataManager
```

Wyjaśnienie: Testy integracyjne weryfikują interakcje między różnymi modułami (np. QuizCreator z QuizDataManager i obiektami Quiz/Question). Mockowane są tylko

zewnętrzne zależności (np. input, list_available_quizzes, load_quiz), natomiast kluczowa interakcja między komponentami jest testowana. Przykładem jest test edycji quizu, który weryfikuje współpracę UI edytora z logiką ładowania/zapisu danych.

Testy graniczne / błędne dane (Testowanie)

Ten fragment ilustruje testy graniczne i błędnych danych.

```
# tests/test_models.py (fragment)
class TestQuestion(unittest.TestCase):
    def test_question_initialization_empty_text_raises_error(self):
        with self.assertRaisesRegex(ValueError, "Question text cannot be empty."):
            Question("", ["a", "b"], 0) # Test graniczny: pusta treść

    def test_question_initialization_invalid_index_raises_error(self):
        with self.assertRaisesRegex(ValueError, "out of bounds or not an integer."):
            Question("Q?", ["a", "b"], 99) # Test błędnych danych: indeks poza zakresem

# tests/test_data_manager.py (fragment)
class TestQuizDataManager(unittest.TestCase):
    def test_load_quiz_invalid_json(self):
        # Tworzy plik z nieprawidłowym JSON-em
        # ...
        with self.assertRaises(json.JSONDecodeError):
            QuizDataManager.load_quiz("bad_format.json", self.test_dir) # Test błędnych danych: uszkodzony JSON
```

Wyjaśnienie: Wiele testów celowo podaje dane na skrajach zakresów lub dane nieprawidłowe (np. puste stringi, indeksy poza zakresem, źle sformułowane pliki JSON). Użycie self.assertRaisesRegex() i self.assertRaises() pozwala upewnić się, że aplikacja poprawnie obsługuje takie sytuacje, rzucając odpowiednie wyjątki i wyświetlając stosowne komunikaty.

Testy wydajności (np. czas wykonania, timeit) (Testowanie)

Ten fragment ilustruje testy wydajności.

```
# quiz_project/utils/helpers.py (fragment)
import timeit
```

```
def perform_factorial_calculation():
    factorial_recursive(10)

def measure_function_performance(func_name: str, setup_code: str, num_runs: int =
100000) -> float:
    time_taken = timeit.timeit(stmt=f"{func_name}()", setup=setup_code,
number=num_runs)
    average_time = time_taken / num_runs
    print(f"[{func_name}] Średni czas: {average_time:.6f} sekund")
    return average_time
```

Wyjaśnienie: W module utils/helpers.py zaimplementowano funkcję measure_function_performance, która wykorzystuje moduł timeit do precyzyjnego mierzenia czasu wykonania innych funkcji. Pozwala to na analizę wydajności kodu (np. factorial_recursive) poprzez wielokrotne uruchamianie go i obliczanie średniego czasu wykonania.

Testy pamięci memory_profiler (Testowanie)

Ten fragment ilustruje testy pamięci.

```
# quiz_project/utils/helpers.py (fragment)
try:
    from memory_profiler import profile
except ImportError:
    def profile(func): return func # Mock, jeśli brak biblioteki

@profile
def generate_large_data(size_mb: int):
    """Generates a large list to demonstrate memory usage."""
    num_elements = int(size_mb * 1024 * 1024 / 28)
    data = [i for i in range(num_elements)]
    print(f"[memory_profiler] Wygenerowano listę z {len(data)} elementami.")
```

Wyjaśnienie: Biblioteka memory_profiler jest wykorzystywana do profilowania zużycia pamięci. Dekorator @profile umieszczony nad funkcją generate_large_data umożliwia śledzenie alokacji pamięci linia po linii podczas jej wykonania. Umożliwia to identyfikację obszarów kodu zużywających dużo pamięci.

Test jakości kodu (flake8, pylint) (Testowanie)

Ten fragment ilustruje uruchomienie narzędzia do testowania jakości kodu.

```
# quiz_project/utils/helpers.py (fragment)
import os
try:
    import pylint.lint
except ImportError:
    class MockPyLint:
        def run_pylint(self, args):
            print("Pylint nie jest zainstalowany.")
            return 1 # Symulujemy błąd
    pylint_runner = MockPyLint()
else:
    class RealPyLint:
        def run_pylint(self, args):
            print(f"Uruchamiam pylint z argumentami: {args}")
            return os.system(f"pylint {' '.join(args)}") # Faktyczne uruchomienie
    pylint_runner = RealPyLint()

def run_pylint_check(file_path: str) -> int:
    return pylint_runner.run_pylint([file_path, "--reports=no"])
```

Wyjaśnienie: W module utils/helpers.py zaimplementowano funkcję run_pylint_check, która programowo wywołuje narzędzie pylint. Pozwala to na automatyczną analizę statyczną kodu pod kątem zgodności ze standardami (np. PEP 8) i wykrywanie potencjalnych problemów, co przyczynia się do utrzymania wysokiej jakości kodu.

7. Testowanie

Projekt jest kompleksowo testowany, aby zapewnić jego stabilność, poprawność działania i zgodność z wymaganiami. Testy są zorganizowane w pakiecie tests/.

Opis sposobu testowania

- **Testy jednostkowe (unittest, assert)**: Skupiają się na izolowanym testowaniu najmniejszych, testowalnych jednostek kodu (np. pojedynczych klas jak Question, Quiz oraz ich metod). Wykorzystuje się wbudowany moduł unittest i jego metody asercji (self.assertEqual, self.assertTrue, self.assertRaisesRegex) do weryfikacji oczekiwanych zachowań i poprawności walidacji danych.

- **Testy funkcjonalne**: Sprawdzają, czy całe funkcjonalności aplikacji działają zgodnie z wymaganiami z perspektywy użytkownika. Symulują pełne interakcje (np. tworzenie quizu od początku do końca, odtwarzanie quizu z odpowiedzią na wszystkie pytania) za pomocą mockowania input() i przechwytywania print() (unittest.mock). Weryfikują końcowy stan aplikacji i wyświetlane komunikaty.
- **Testy Integracyjne**: Weryfikują interakcje między różnymi modułami i komponentami systemu (np. QuizCreator współpracujący z QuizDataManager, który z kolei wykorzystuje klasy Question i Quiz). Mockowane są tylko zewnętrzne zależności (np. operacje na systemie plików), aby skupić się na współpracy wewnętrznych komponentów.
- **Testy graniczne / błędne dane**: Celowo podaje się dane na skrajach dopuszczalnych zakresów (np. puste stringi, indeksy poza zakresem, minimalne/maksymalne wartości) oraz dane nieprawidłowe (np. źle sformatowane pliki JSON). Testy te używają self.assertRaises (lub assertRaisesRegex) do weryfikacji, czy program poprawnie rzuca oczekiwane wyjątki i obsługuje takie sytuacje.
- **Testy wydajności (timeit)**: W module utils/helpers.py zaimplementowano funkcję measure_function_performance, która wykorzystuje moduł timeit do precyzyjnego pomiaru czasu wykonania określonych fragmentów kodu (np. funkcji rekurencyjnej). Pozwala to na identyfikację potencjalnych wąskich gardeł wydajnościowych.
- **Testy pamięci (memory_profiler)**: W module utils/helpers.py zintegrowano bibliotekę memory_profiler. Dekorator @profile pozwala na śledzenie zużycia pamięci linia po linii dla wybranych funkcji, co pomaga w identyfikacji obszarów o wysokim zużyciu pamięci i potencjalnych wycieków.
- **Test jakości kodu (pylint)**: W module utils/helpers.py zaimplementowano funkcję run_pylint_check, która umożliwia programowe uruchomienie narzędzia pylint. Pozwala to na statyczną analizę kodu pod kątem zgodności ze standardami kodowania (np. PEP 8), błędów stylistycznych i potencjalnych problemów programistycznych.

Jak uruchomić wszystkie testy

Aby uruchomić wszystkie testy dla projektu, przejdź do katalogu głównego projektu (quiz_project/) w terminalu i użyj następującej komendy:

python -m unittest discover tests

8. Wnioski

Co się udało

- **Kompleksowa funkcjonalność**: Stworzono w pełni funkcjonalną aplikację konsolową do tworzenia, edycji i odtwarzania quizów, spełniającą wszystkie podstawowe wymagania.
- **Trwała persystencja danych**: Zaimplementowano niezawodny mechanizm zapisu i odczytu quizów do/z plików JSON, zapewniający trwałość danych.
- **Robustność i obsługa błędów**: Aplikacja skutecznie radzi sobie z błędnymi danymi wejściowymi użytkownika oraz problemami z plikami dzięki rozbudowanym mechanizmom obsługi wyjątków.
- **Modularna architektura**: Projekt został zaprojektowany z myślą o modułowości, co ułatwia zarządzanie kodem, jego czytelność i potencjalną rozbudowę.
- **Zastosowanie paradygmatów programowania**: Skutecznie wykorzystano programowanie obiektowe (klasy, metody, konstruktory, dziedziczenie) oraz programowanie funkcyjne (map, filter, lambda, reduce).
- **Wizualizacja danych**: Pomyślnie zaimplementowano generowanie wykresów wyników, co zwiększa użyteczność aplikacji.
- **Kompleksowe testowanie**: Stworzono szeroki zestaw testów jednostkowych, funkcjonalnych i integracyjnych, które w znaczący sposób zwiększają zaufanie do jakości i poprawności działania kodu. Dodatkowo zaimplementowano testy wydajności i pamięci oraz narzędzie do sprawdzania jakości kodu.
- **Kontrola wersji**: Projekt był zarządzany w systemie kontroli wersji Git, co zapewniało śledzenie zmian i porządek w rozwoju.

Co można było zrobić lepiej

- **Bardziej zaawansowane UI**: Obecne UI jest konsolowe. Można by rozważyć implementację graficznego interfejsu użytkownika (GUI) przy użyciu bibliotek takich jak Tkinter, PyQt lub Kivy, co znacznie poprawiłoby doświadczenia użytkownika.
- **Więcej typów pytań**: Obecnie obsługiwane są tylko pytania jednokrotnego wyboru. Można by dodać pytania wielokrotnego wyboru, pytania otwarte, czy pytania typu prawda/fałsz.
- **Zarządzanie użytkownikami/wynikami**: Aplikacja nie przechowuje historii wyników dla poszczególnych użytkowników. Można by zaimplementować system logowania/rejestracji i bazę danych do przechowywania profili użytkowników i ich wyników.
- **Integracja z reduce w głównej logice**: Chociaż reduce zostało zaimplementowane, jego zastosowanie w głównej logice analizy wyników mogłoby być bardziej bezpośrednie.
- **Zaawansowane raporty**: Wizualizacje są podstawowe. Można by tworzyć

bardziej złożone raporty, np. z trendami wyników, porównaniem wyników między użytkownikami.
- **Automatyzacja testów jakości kodu**: Chociaż funkcja do testowania jakości kodu jest obecna, jej automatyczne uruchamianie (np. jako hook Git) mogłoby zapewnić jeszcze większą spójność.

Jakie kompetencje zostały rozwinięte

- **Projektowanie i architektura oprogramowania**: Rozwinięto umiejętność projektowania modułowych i skalowalnych aplikacji w Pythonie.
- **Programowanie obiektowe (OOP)**: Pogłębiono zrozumienie i praktyczne zastosowanie klas, obiektów, metod, konstruktorów oraz dziedziczenia do modelowania problemów.
- **Programowanie funkcyjne**: Nabyto doświadczenie w wykorzystywaniu funkcji wyższego rzędu (map, filter, lambda, reduce) do efektywnego przetwarzania danych.
- **Obsługa plików i serializacja danych**: Ugruntowano umiejętności pracy z plikami, w tym zapisywania i odczytywania danych w formacie JSON, z kompleksową obsługą błędów.
- **Testowanie oprogramowania**: Znacząco rozwinięto kompetencje w zakresie pisania i uruchamiania testów jednostkowych, funkcjonalnych i integracyjnych, w tym zaawansowanych technik mockowania i profilowania.
- **Kontrola wersji (Git/GitHub)**: Praktyczne zarządzanie projektem z wykorzystaniem systemu kontroli wersji, w tym tworzenie opisowych commitów i zarządzanie repozytorium.
- **Rozwiązywanie problemów**: Umiejętność diagnozowania i rozwiązywania złożonych problemów (np. błędy w testach, problemy z importami, konfiguracja środowiska).
- **Wizualizacja danych**: Podstawowe umiejętności w zakresie tworzenia graficznych reprezentacji danych za pomocą matplotlib.