DOKUMENTACJA PROJEKTU W RAMACH PRZEDMIOTU PROGRAMOWANIE W JĘZYKU PYTHON.

Autor:
Filip Maciborski

Tytuł projektu:
Vocabulary tester

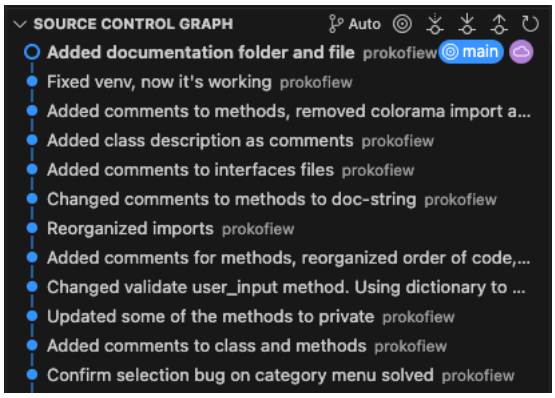
Tematy uwzględnione w projekcie:

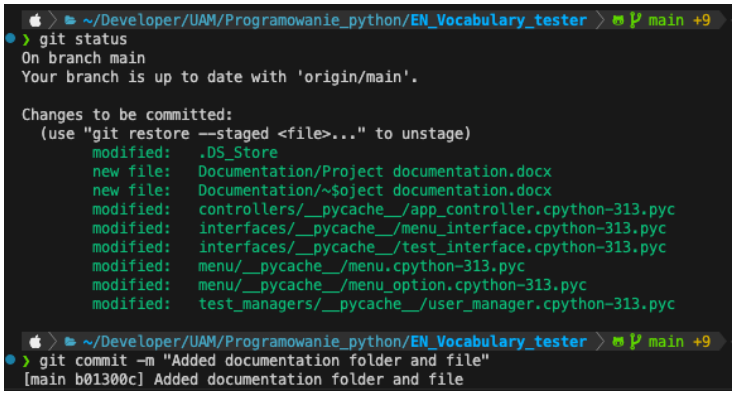
1. Git.
2. Czytelny kod w Pythonie – PEP8.
3. Zaawansowane struktury danych – moduł Pandas.
4. Obsługa dat i czasu – moduł DateTime.
5. Programowanie funkcyjne.
6. Programowanie obiektowe.

1. Git i GitHub

Projekt od pewnego momentu przechowywany był na GitHubie i także za jego pośrednictwem został przekazany wykładowcy. W trakcie pracy nad projektem, choć dopiero od pewnego momentu stosowany był Git.

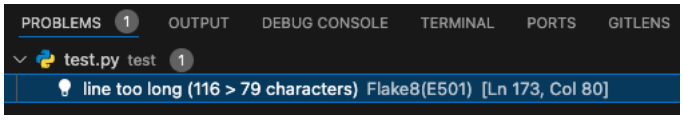
Git został zainstalowany oraz odpowiednio skonfigurowany. Został także połączony poprzez klucz SSH z GitHubem.





2. Czytelny kod w Pythonie – PEP8.

Nad projektem pracowałem z wykorzystaniem Visual Studio Code. Jako dodatek zainstalowałem *Flake8* podkreślający wszystkie możliwe błędy jakie występują w kodzie niezgodnym z *PEP8* np.:



Zastosowałem się do wszystkich wskazań dodatku i dostosowałem kod do *PEP8*.

Przykłady (na podstawie klasy *NewTest*)**:**

⇒ Dzielenie długich stringów na linie i stosowanie odpowiednich wcięć(klasa *NewTest*):

```
155     def start_test(self):
156         """ Starts the actual test,
157         displays summary of users choices,
158         displays importnt information about test"""
159 
160         if self.test_language_version and self.selected_category:
161             Menu.clear_console()
162             print(f"Starting test.\nChoosen options:\n"
163                   f"1. Translating from: {self.test_language_version}.\n"
164                   f"2. Category: "
165                   f"{self.selected_category if isinstance(
166                       self.selected_category, str) else ', '.join(
167                           map(str, self.selected_category))}.\n"
168                   f"3. Number of questions: {self.questions_amount}.\n"
169                   f"4. Test time limit: "
170                   f"{self.test_time_limit_in_seconds // 60} min.\n")
```

⇒ ‘Zawijanie’ długich wywołań funkcji z zachowaniem wcięć:

```
85          # creating list of categories sorted by category_id
86          categories = (
87              self.data[["category_id", "category_name"]]
88              .drop_duplicates()
89              .sort_values("category_id")
90              .values
91              .tolist())
92
```

⇒ Komentarze opisujące zadania/działanie metod jako multiline doc-string:

```
210     def get_results(self, correct_answers, user_answers, expressions):
211         """ gets questions, correct answers
212         and user answers into a DataFrame, normalizes text,
213         compares test data and points wrong and correct answers,
214         modifies DataFrame to display text information (map),
215         calculates points and percentage"""
216
```

⇒ Prawidłowe uporządkowanie argumentów funkcji w formie listy (klasa *NewTest*):

```
251         def end_test(self, test_data):
252             """ passing data to result manger """
253             result_manager = ResultManager(
254                 self.test_datetime,
255                 self.user_name,
256                 self.point_score,
257                 self.questions_amount,
258                 self.percentage_score,
259                 self.test_duration,
260                 self.test_time_limit_in_seconds,
261                 test_data
262             )
```

⇒ Uporządkowane importy (klasa *NewTest*):

```
1   import datetime
2   import pandas as pd
3
4   from colorama import Fore
5   from interfaces.test_interface import Test
6   from menu.menu import Menu
7   from menu.menu_option import MenuOption
8   from test_managers.text_formatter import TextFormatter
9   from test_managers.question_manager import QuestionManager
10  from test_managers.result_manager import ResultManager
11  from test_managers.time_manager import TimeManager
12  from test_managers.file_manager import FileManager
13  from test_managers.user_manager import UserManager
14
```

⇒ Prawidłowe nazewnictwo klas i metod:

```
16  class NewTest(Test):
17      def __init__(self, data_file, data, main_menu):
18          self.main_menu = main_menu  # allowing user to use main menu
19          self.data_file = data_file  # database file path
20          self.data = data  # Data from database as DataFrame
21          self.test_language_version = None
22          self.selected_category = None
```

```
76
77          def choose_all():
78              self.selected_category = "All categories"
79              self.question_manager.set_category(self.selected_category)
80              self.__get_questions_amount()
81
82          def back_to_language():
83              self.__set_language()
```

3. Zaawansowane struktury danych – moduł Pandas.

W projekcie został wykorzystany moduł *Pandas*. W największym zakresie wykorzystana została *DataFrame* w oparciu, o który program funkcjonuje:

⇒ Praca z plikiem Excel i arkuszami pliku w których została utworzona baza danych programu (klasa *AppController*):

Pobranie danych z pliku:

```
112     def __data_load(self):
113         """ Loads database file
114         and establishes sheets as properties """
115         try:
116             file_data = pd.ExcelFile(TEST_DATABASE)
117             self.dictionaries = file_data.parse(sheet_name="categories")
118             self.vocabluary = file_data.parse(sheet_name="vocabluary")
119             self.data = pd.merge(
120                 self.vocabluary,
121                 self.dictionaries,
122                 left_on="category",
123                 right_on="category_id")
124         except FileNotFoundError:
125             print("Database file: tester_database.xlsx not found.")
126             sys.exit()
127
```

Zapis danych do pliku:

```
143     def __save_to_database(self, data_frame, sheet_name):
144         """ Updating database file with new vocabulary"""
145         with pd.ExcelWriter(
146             TEST_DATABASE, mode="a", if_sheet_exists="overlay"
147         ) as writer:
148             data_frame.to_excel(writer, sheet_name=sheet_name, index=False)
```

⇒ Wykorzystanie *DataFrame* do zarządzania przepływem danych, podczas tworzenia pytań i zbierania odpowiedzi użytkownika, dodawania nowych danych do bazy danych.

Łączenie danych:

```
138     def __join_data_frames(self, base_data_frame, added_data_frame):
139         """ Private method to join DataFrames with pd.concat """
140         return pd.concat([
141             base_data_frame, added_data_frame], ignore_index=True)
```

Tworzenie nowej kategorii słów:

```
150     def __create_new_category(self, new_category):
151         """ Creating new category_id and
152         DataFrame for new vocabulary"""
153         new_category_id = int(self.dictionaries["category_id"].max() + 1)
154         new_category_data_frame = pd.DataFrame(
155             {
156                 "category_id": [new_category_id],
157                 "category_name": [new_category]
158             }
159         )
```

Operacje na kolumnach w celu pozyskania wyników testu (klasa *NewTest*):

- **map()** – zmiana wartości bazując w serii na podstawie podanej innej wartości
- **apply()** – stosowanie funkcji do wierszy lub kolumn
- **astype()** - zmiana typu danych w series
- **mean()** – średnia procentowa kolumny
- **sum()** – obliczenie sumy kolumny

```
def get_results(self, correct_answers, user_answers, expressions):
    """ gets questions, correct answers      You, 13 minutes ago • Unc
    and user answers into a DataFrame, normalizes text,
    compares test data and points wrong and correct answers,
    modifies DataFrame to display text information (map),
    calculates points and percentage"""

    test_data = pd.DataFrame({
        "Questions": expressions,
        "Correct answers": correct_answers,
        "Your answers": user_answers,
    })

    # Normalize text
    test_data["Normalized Correct"] = test_data[
        "Correct answers"].apply(TextFormatter.normalize_text)
    test_data["Normalized User"] = test_data[
        "Your answers"].apply(TextFormatter.normalize_text)

    # Comparing data
    test_data["Correct/Wrong"] = test_data[
        "Normalized Correct"] == test_data["Normalized User"]
    test_data["Points"] = test_data["Correct/Wrong"].astype(int)

    # Map to text
    test_data["Correct/Wrong"] = test_data[
        "Correct/Wrong"].map({True: "Correct", False: "Wrong"})

    # Calculating the results
    self.percentage_score = test_data["Points"].mean() * 100
    self.point_score = test_data["Points"].sum()

    return test_data
```

4. Obsługa dat i czasu – moduł DateTime.

Moduł *DateTime* został w programie wykorzystany do pobrania daty i godziny przeprowadzenia testu, a także zmierzenia czasu jego trwania. W oparciu o limit czasu ustawiany przez użytkownika w minutach, obliczany jest limit czasu na sekundy. Stworzony dekorator mierzy czas trwania testu. Na podstawie porównania wartości określane jest to czy użytkownik zmieścił się w limicie czasowym.

Przykłady (klasa NewTest)**:**

⇒ Pobranie daty i czasu podczas tworzenia instancji klasy *NewTest* – wybranie z menu głównego opcji -> *'Start New Test'*

```
class NewTest(Test):
    def __init__(self, data_file, data, main_menu):
        self.main_menu = main_menu  # allowing user to use main menu
        self.data_file = data_file  # database file path
        self.data = data  # Data from database as DataFrame
        self.test_language_version = None
        self.selected_category = None
        self.test_datetime = datetime.datetime.now()
```

⇒ Ustalenie limitu czasu dla testu – bez wykorzystania *DateTime,* ale ważne w późniejszym użyciu (klasa *TimeManager*)

```
    def set_test_time_limit(self):
        try:
            test_time_limit = int(input("Enter test time limit in minutes: "))
            self.test_time_limit_in_seconds = test_time_limit * 60
            return True
        except ValueError:
            message = "Invalid value. Enter a number"
            print(self.text_formatter.colorize(message, Fore.RED))
            self.display_sleep(1.5)
            return self.set_test_time_limit()
```

⇒ Pomiar czasu trwania testu (wrapper w *TimeManager*):

```
    @staticmethod
    def measure_time(func):
        @wraps(func)
        def wrapper(instance, *args, **kwargs):
            start_time = datetime.datetime.now()
            result = func(instance, *args, **kwargs)
            end_time = datetime.datetime.now()
            instance.test_duration = (end_time - start_time).total_seconds()
            return result
        return wrapper
```

⇒ Pomiar czasu uruchomiony dla funkcji, która wyświetla pytania i pobiera odpowiedzi użytkownika, czyli w momencie, gdy użytkownik przechodzi do rozwiązywania testu (klasa *NewTest*, metoda *def submit_answer()* wywołana w metodzie def *start_test()*):

```
    @TimeManager.measure_time  # TimeManager decorator to measure test time
    def submit_answer(self, questions_data):
        """ responsible for displaing questions and gathering answers,
        checking if test was stoped"""
        user_answers = []
        print(self.text_formatter.colorize(
            "*** To stop test, enter: \"Stop test\" ***\n",
              Fore.LIGHTYELLOW_EX))
```

⇒ Wyświetlanie poszczególnych elementów *DateTime*, osobno daty i czasu w odpowiednim formacie (klasa *ResultManager* metoda *def display_test_outcome()*):

```
44      def display_test_outcome(self):
45          print(f"Test date: {self.test_datetime.strftime('%d-%m-%Y')}")
46          print(f"Test time: {self.test_datetime.strftime('%H:%M:%S')}")
```

5. Programowanie funkcyjne.

Program wykorzystuje paradygmat programowania funkcyjnego. Jest to poniekąd efekt tego, że zastosowałem podejście obiektowe.

⇒ Funkcja wyższego rzędu – funkcja przyjmująca inną funkcję jako argument (wrapper w klasie *TimeManager*):

```
13      @staticmethod
14      def measure_time(func):
15          @wraps(func)
16          def wrapper(instance, *args, **kwargs):
17              start_time = datetime.datetime.now()
18              result = func(instance, *args, **kwargs)
19              end_time = datetime.datetime.now()
20              instance.test_duration = (end_time - start_time).total_seconds()
21              return result
22          return wrapper
```

⇒ Funkcja wyższego rzędu, przyjmuje tekst i kolor jako argument, a następnie zwraca zmodyfikowany tekst – przetwarzanie danych w stylu funkcyjnym (klasa *TextFormatter*).

```
22      def colorize(self, text, color):
23          """
24          Applies the specified color to the text.
25          """
26          return f"{color}{text}{Style.RESET_ALL}" if color else text
```

⇒ Funkcja czysta, nie modyfikuje stanu obiektu, wynik funkcji zależy wyłącznie od przekazanych argumentów, zwraca ten sam wynik dla tych samych danych wejściowych (klasa *TextFormatter*):

```python
    @staticmethod
    def normalize_text(text):
        """
        Removes diacritics, converts to lowercase, and strips whitespace.
        """
        if not isinstance(text, str):
            return text

        # Remove leading and trailing whitespace
        stripped_text = text.strip()

        # Normalize text to decompose diacritic characters
        normalized_text = unicodedata.normalize("NFD", stripped_text)

        # Remove diacritic marks
        without_diacritics = "".join(
            char for char in normalized_text
            if unicodedata.category(char) != "Mn"
        )

        # Replace specific characters and convert to lowercase
        final_text = without_diacritics.replace('ł', 'l').lower()

        return final_text
```

⇒ Funkcja czysta, jedynym efektem jest wykonanie polecenia systemowego, nie zależy od stanu obiektu i go nie modyfikuje (klasa *Menu*):

```python
    @staticmethod
    def clear_console():
        if os.name == "nt":
            os.system("cls")
        else:
            os.system("clear")
```

6. Programowanie obiektowe.

Program wykorzystuje kilka cech programowania obiektowego. M.in.:

⇒ Interfejsy i ich implementacja. Wykorzystane zostały dwa interfejsy (klasy abstrakcyjne), które wymuszą implementacje konkretnych metod podczas tworzenia klasy implementującej interfejs:

```python
from abc import ABC
from abc import abstractmethod

# Test iterface with abstract methods


You, 3 days ago | 1 author (You)
class Test(ABC):

    @abstractmethod
    def start_test(self):
        pass

    @abstractmethod
    def get_questions_and_answers_data(self):
        pass

    @abstractmethod
    def submit_answer(self, answer):
        pass

    @abstractmethod
    def get_results(self):
        pass

    @abstractmethod
    def end_test(self):
        pass

    @abstractmethod
    def save_results(self):
        pass

```

```python
16  class NewTest(Test):
```

⇒ Enkapsulacja. Zarówno niektóre metody oraz pola klas ukrywane są przed użytkownikiem i dostęp do nich jest ograniczony, gdyż z punktu widzenia użytkownika szczegóły implementacji nie są istotne lub ważne jest by użytkownik lub inne klasy nie mogły ich swobodnie modyfikować (klasa *NewTest*):

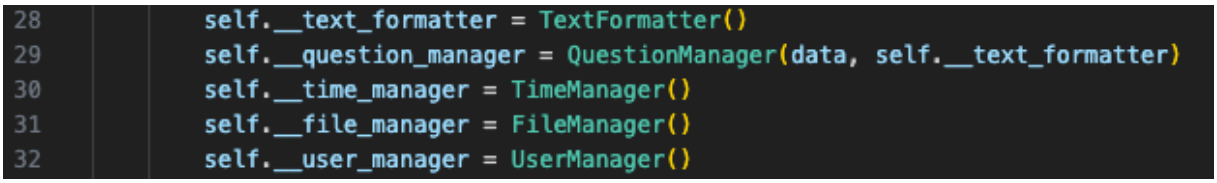
```python
        self.__test_language_version = None
        self.__selected_category = None
        self.__test_datetime = datetime.datetime.now()
        self.__point_score = 0
        self.__percentage_score = 0
        self.__user_name = None
        self.__test_time_limit_in_seconds = None
```

Użytkownik nie ma w ogóle możliwości modyfikacji np. *__self.point_score* lub *self.__percentage_score*. Pola te modyfikowane są przez odpowiednie metody w klasie.

Użytkownik ma możliwość modyfikowania pól *self.__user_name* oraz *self.__test_language* ale odbywa się to za pośrednictwem odpowiednich metod, które maja dostęp do tych pól. Same metody są również prywatne.

- W tym przypadku niektóre klasy korzystają z metod wewnętrznie i nie ma potrzeby by były dostępne spoza klasy. Metoda *def __initiate_language_menu()* wywoływana jest w konstruktorze i służy do utworzenia menu wyboru języka. Żadna inna klasa nie musi z tej metody korzystać.

⇒ Kompozycja. Najlepszym przykładem jest klasa *NewTest*, która korzysta z kilku innych klas w celu obsługi najważniejszych elementów testu:



Klasa ta w konstruktorze inicjuje kilka obiektów które realizują:

1. **TextFormatter** – kolorowanie i normalizacja tekstu odpowiedzi, tak aby odpowiedzi mogły być udzielane w języku polskim bez ‘polskich znaków’
2. **Questionmanager** – odpowiada za określenie ilości pytań i stworzenie list pytań i odpowiedzi.
3. **TimeManager** – odpowiada za ustalanie limitu czasu testu, mierzenie czasu przebiegu testu, odliczanie do rozpoczęcia testu i utrzymanie uśpienia ekranu np. podczas wyświetlania komunikatów o błędach
4. **FileManager** – odpowiada za zapis danych do pliku
5. **UserManager** – odpowiada za interakcję z użytkownikiem, ustawienie imienia, walidację wprowadzanych przez niego wartości w terminalu.