DOKUMENTACJA PROJEKTU W RAMACH PRZEDMIOTU PROGRAMOWANIE W JĘZYKU PYTHON.

Autor:

Filip Maciborski

Tytuł projektu:

Vocabulary tester

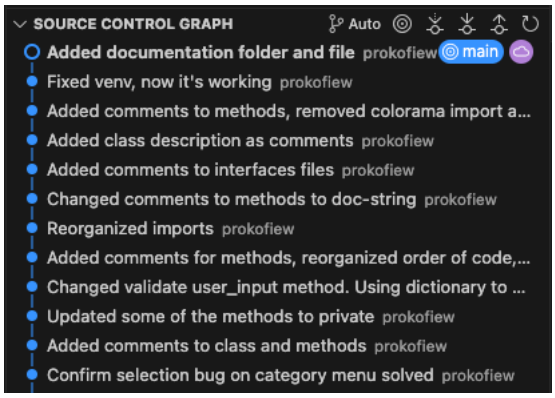
Tematy uwzględnione w projekcie:

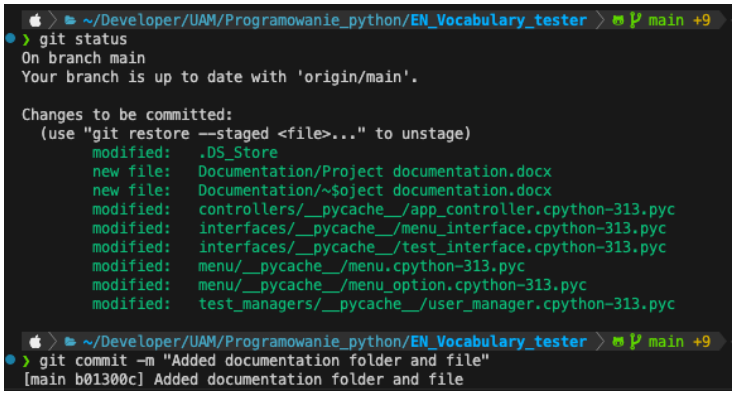
1. Git.
2. Czytelny kod w Pythonie – PEP8.
3. Zaawansowane struktury danych – moduł Pandas.
4. Obsługa dat i czasu – moduł DateTime.
5. Programowanie funkcyjne.
6. Programowanie obiektowe.

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.

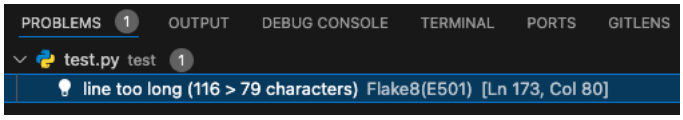
Git został zainstalowany oraz odpowiednio skonfigurowany. Został także połączony poprzez klucz SSH z GitHubem.





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:

```
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 (klasa *ResultManager*):

```
96      def analyze_answers(self, correct_answers, user_answers, expressions):
97          """ gets questions, correct answers
98          and user answers into a DataFrame, normalizes text,
99          compares test data and points wrong and correct answers,
100         modifies DataFrame to display text information (map),
101         calculates points and percentage"""
102
```

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

```
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()
```

⇒ 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:

```python
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

        def choose_all():
            self.selected_category = "All categories"
            self.question_manager.set_category(self.selected_category)
            self.__get_questions_amount()

        def back_to_language():
            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:

```python
    def __data_load(self):
        """ Loads database file
        and establishes sheets as properties """
        try:
            file_data = pd.ExcelFile(TEST_DATABASE)
            self.dictionaries = file_data.parse(sheet_name="categories")
            self.vocabluary = file_data.parse(sheet_name="vocabluary")
            self.data = pd.merge(
                self.vocabluary,
                self.dictionaries,
                left_on="category",
                right_on="category_id")
        except FileNotFoundError:
            print("Database file: tester_database.xlsx not found.")
            sys.exit()
```

Zapis danych do pliku:

```python
    def __save_to_database(self, data_frame, sheet_name):
        """ Updating database file with new vocabulary"""
        with pd.ExcelWriter(
            TEST_DATABASE, mode="a", if_sheet_exists="overlay"
        ) as writer:
            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 *ResultManager*):
- **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

```
96      def analyze_answers(self, correct_answers, user_answers, expressions):
97          """ gets questions, correct answers
98          and user answers into a DataFrame, normalizes text,
99          compares test data and points wrong and correct answers,
100         modifies DataFrame to display text information (map),
101         calculates points and percentage"""
102
103         test_data = pd.DataFrame({
104             "Questions": expressions,
105             "Correct answers": correct_answers,
106             "Your answers": user_answers,
107         })
108
109         # Normalize text
110         test_data["Normalized Correct"] = test_data[
111             "Correct answers"].apply(TextFormatter.normalize_text)
112         test_data["Normalized User"] = test_data[
113             "Your answers"].apply(TextFormatter.normalize_text)
114
115         # Comparing data
116         test_data["Correct/Wrong"] = test_data[
117             "Normalized Correct"] == test_data["Normalized User"]
118         test_data["Points"] = test_data["Correct/Wrong"].astype(int)
119
120         # Map to text
121         test_data["Correct/Wrong"] = test_data[
122             "Correct/Wrong"].map({True: "Correct", False: "Wrong"})
123
124         # Calculating the results
125         point_score = test_data["Points"].sum()
126         percentage_score = test_data["Points"].mean() * 100
127         self.test_instance.set_point_score(point_score)
128         self.test_instance.set_percentage_score(percentage_score)
129
130         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'*

```
15    class NewTest(Test):
16        def __init__(self, data_file, data, main_menu):
17            self._data_file = data_file  # path of Excel file for
18            self._user_name = None
19            self._test_time_limit_in_seconds = 0
20            self._test_language_version = None
21            self._selected_category = None
22            self._test_duration = 0
23            self._questions_amount = 0
24            self.__main_menu = main_menu  # allowing user to use m
25            self.__data = data  # Data from database as DataFrame
26            self.__test_datetime = self.set_test_datetime()
```

⇒ Pomiar czasu trwania testu (wrapper w *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.set_test_duration((end_time - start_time).total_seconds())
21            return result
22        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()*):

```
222     @TimeManager.measure_time  # TimeManager decorator to measure test time
223     def submit_answer(self, questions_data):
224         """ responsible for displaying questions and gathering answers,
225         checking if test was stopped"""
226         user_answers = []
227 
228         for idx, expression in enumerate(questions_data):
229             print(f"Question no.{idx + 1}")
230             answer = input(
231                 f"Enter translation of: \"{expression}\".\nAnswer -> ")
232             if answer.lower() == "stop test":
233                 return user_answers, True
234             else:
235                 user_answers.append(answer.strip())
236 
237         # Return the answers and flag whether the test was stopped
238         return user_answers, False
```

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

```
153         print(f"Test date: {test_date_time.strftime('%d-%m-%Y')}")
154         print(f"Test time: {test_date_time.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*):

```
9       @staticmethod
10      def normalize_text(text):
11          """
12          Removes diacritics, converts to lowercase, and strips whitespace.
13          """
14          if not isinstance(text, str):
15              return text
16
17          # Remove leading and trailing whitespace
18          stripped_text = text.strip()
19
20          # Normalize text to decompose diacritic characters
21          normalized_text = unicodedata.normalize("NFD", stripped_text)
22
23          # Remove diacritic marks
24          without_diacritics = "".join(
25              char for char in normalized_text
26              if unicodedata.category(char) != "Mn"
27          )
28
29          # Replace specific characters and convert to lowercase
30          final_text = without_diacritics.replace('ł', 'l').lower()
31
32          return final_text
```

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

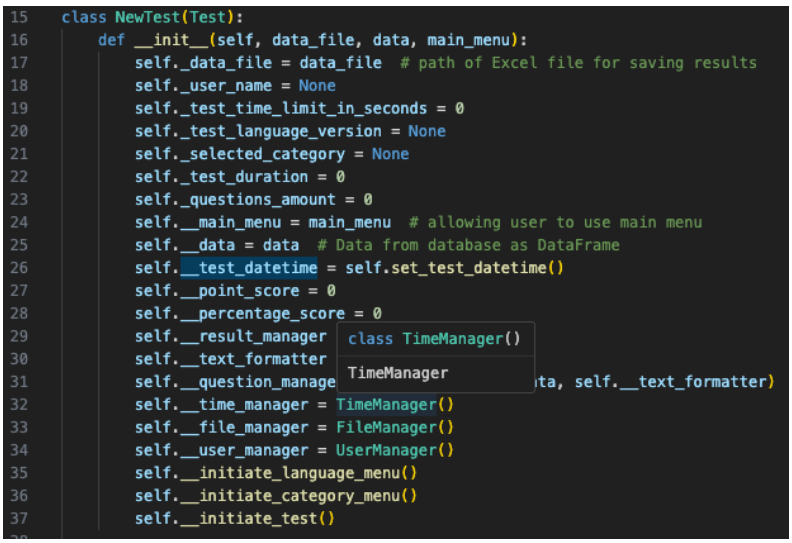
```
20      @staticmethod
21      def clear_console():
22          if os.name == "nt":
23              os.system("cls")
24          else:
25              os.system("clear")
```

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:

⇒ 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*):



Użytkownik nie ma w ogóle możliwości modyfikacji np. *__self.point_score* lub *self.__percentage_score*. Pola te są także chronione przed przypadkową modyfikacją wewnątrz klasy. Dostęp do nich można uzyskać jedynie za pomocą specjalnie do tego stworzonych metod *'gettera'* i *'settera'*:

```
46      def get_point_score(self):
47          return self.__point_score
48
49      def set_point_score(self, points_data):
50          self.__point_score = points_data
51
52      def get_percentage_score(self):
53          return self.__percentage_score
54
55      def set_percentage_score(self, percentage_data):
56          self.__percentage_score = percentage_data
```

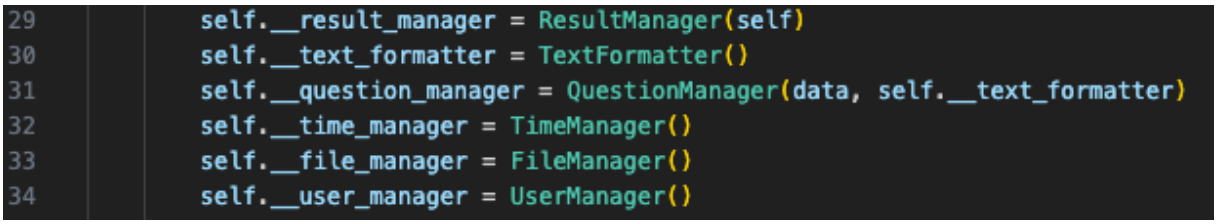
Ponieważ niektóre klasy są ze sobą mocno powiązane np. *NewTest* oraz *ResultManager*, aby *ResultManager* mógł obliczyć wynik oraz ustawić/zapisać punkty oraz procentowy wynik testu, wykorzystane jest przekazanie instancji testu to obiektu *ResultManager*. *ResultMenager* musi także wykorzystać *'getter'* oraz *'setter'* by wykonywać działania na polach klasy *NewTest*:

Metoda w klasie *ResultManger*

```
148     def display_test_outcome(self):
149         test_date_time = self.test_instance.get_test_datetime()
150         points_score = self.test_instance.get_point_score()
151         percentage_score = self.test_instance.get_percentage_score()
152         questions_amount = self.test_instance.get_questions_amount()
153         print(f"Test date: {test_date_time.strftime('%d-%m-%Y')}")
154         print(f"Test time: {test_date_time.strftime('%H:%M:%S')}")
155         print(f"User: {self.test_instance._user_name}")
156         print(f"Points: {points_score}/{questions_amount}")
157         print(f"Percentage: {percentage_score:.2f}%")
158         test_duration_str = self.__format_test_duration()
159         print(f"Test time limit: {
160             self.test_instance._test_time_limit_in_seconds // 60} min.")
161         print(f"Your time is: {test_duration_str}\n")
162         self.__determine_test_outcome()
```

- Niektóre klasy korzystają z metod jedynie 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 po uruchomieniu nowego tesru. 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.
6. **ResultManager** -odpowiada za porównanie odpowiedzi użytkownika i prawidłowych odpowiedzi z bazy danych. Zlicza punkty i oblicza procentowy wynik testu. Wyświetla także wynik testu w formie podsumowania.