Министерство просвещения Российской Федерации

Полное наименование учебного заведения

VIII Международный конкурс исследовательских работ школьников
"Research start" 2025/26

Проектная работа

Генератор проверочных работ по неправильным глаголам

Выполнил: Езжов Леонтий Сергеевич

ученик 8 класса

Руководитель: Мартынова Мария Игоревна

учитель информатики

2026 год

Введение

В современном образовании цифровые инструменты становятся главными помощниками в упрощении рутинных задач. Это особенно актуально для предметов, требующих регулярной практики и проверки, таких как английский язык. Здесь одной из важных и одновременно сложной для запоминания является тема «Неправильные глаголы».

Однако часто процесс отработки их с учениками является сложным для преподавателя: подготовка разнообразных и уникальных проверочных работ отнимает время.

Именно для решения этих задач и был создан данный проект — «Генератор проверочных работ по неправильным глаголам».

Цель проекта: это создание программы, которая автоматизирует генерацию проверочных работ по теме «Неправильные глаголы».

Задачи:

1. Изучить библиотеку python-docx
2. Спроектировать и реализовать графический интерфейс программы.
3. Написать модуль генерации заданий и сохранения в Word-файл.
4. Реализовать генерацию двух вариантов одновременно.
5. Провести тестирование и подготовить информационный файл о программе.

Суть проекта: это программа, которая автоматизирует создание проверочной работы по теме “Неправильные глаголы”. Оно позволяет

учителю настраивать параметры будущей работы — выбирать количество глаголов, делать несколько вариантов и мгновенно получать готовый уникальный вариант. Результатом станет программа, генерирующая материалы в формате Word (.doc).

Библиотека python-docx

Язык программирования Python, известный своей простотой и обширной экосистемой библиотек, предлагает элегантное решение для этой задачи — библиотеку `python-docx`. Это мощный инструмент, который позволяет разработчикам и аналитикам программно создавать и изменять файлы `.docx` без необходимости установки Microsoft Office на сервере или компьютере.

`python-docx` — это библиотека с открытым исходным кодом для чтения, создания и обновления документов Microsoft Word 2007+ (.docx) файлов. Важно понимать, что она работает не с бинарным старым форматом `.doc`, а с современным форматом Office Open XML (OOXML), который представляет собой zip-архив, содержащий XML-файлы со структурой документа.

Библиотека спроектирована таким образом, чтобы абстрагировать разработчика от сложностей разбора XML и предоставить чистый, интуитивно понятный объектно-ориентированный Python API. Вместо того чтобы писать сложные XML-теги, вы работаете с объектами `Document`, `Paragraph`, `Run`, `Table` и так далее.

Установка библиотеки тривиальна и выполняется стандартным менеджером пакетов pip:

```bash
pip install python-docx
```

``

Основные концепции и объекты

Чтобы эффективно работать с `python-docx`, необходимо понимать его внутреннюю логику и иерархию объектов.

1. Document (Документ): Это корневой объект. Он представляет собой весь документ Word. С него все начинается: вы либо создаете новый пустой документ (`doc = Document()`), либо открываете существующий (`doc = Document('example.docx')`).

2. Paragraph (Абзац): Текст в Word организован в абзацы. В `python-docx` абзац представлен объектом `Paragraph`. Абзац — это блок текста, который заканчивается символом новой строки (при нажатии Enter). Вы можете добавлять новые абзацы в документ с помощью метода `add_paragraph()`.

3. Run (Фрагмент): Это самая важная и часто вызывающая путаницу у новичков концепция. Один абзац может состоять из нескольких "ранов" (Runs). Run — это непрерывная последовательность символов с одинаковым форматированием. Если в одном абзаце часть текста написана жирным, а часть — курсивом, то это два разных объекта `Run`. Это позволяет очень гибко управлять стилями на микроуровне.

4. Table (Таблица): Объект для работы с таблицами. Содержит строки (`rows`) и ячейки (`cells`). Ячейка, в свою очередь, может содержать абзацы.

5. Style (Стиль): Объекты стилей (для абзацев, символов, таблиц) позволяют применять предопределенные наборы параметров форматирования, что обеспечивает единообразие оформления документа.

Создание и форматирование текста

Рассмотрим базовый пример создания документа и добавления текста с различным форматированием.

```python

from docx import Document

from docx.shared import Pt, Cm, RGBColor

from docx.enum.text import WD_ALIGN_PARAGRAPH

1. Создаем новый документ

doc = Document()

2. Добавляем заголовок (по умолчанию стиль 'Heading 1')

doc.add_heading('Отчет о продажах за 2023 год', level=1)

3. Добавляем обычный абзац

paragraph = doc.add_paragraph('Это вводный текст отчета. ')

Добавляем текст с форматированием в тот же абзац

run = paragraph.add_run('Этот текст жирный и красный.')
```

```
run.bold = True

run.font.color.rgb = RGBColor(255, 0, 0)

run.font.size = Pt(14)

 Добавляем еще один абзац

paragraph = doc.add_paragraph('Здесь мог быть ваш текст. ')

 Добавляем подчеркнутый фрагмент

run = paragraph.add_run('А это подчеркнутый текст.')

run.underline = True

 4. Добавляем абзац, выровненный по центру

centered_paragraph = doc.add_paragraph('Этот текст будет по центру')

centered_paragraph.alignment = WD_ALIGN_PARAGRAPH.CENTER

 Сохраняем документ

doc.save('my_first_report.docx')
```

В этом примере мы видим, как создавать заголовки, абзацы и применять форматирование к отдельным фрагментам текста. Для задания размеров и цветов мы импортируем вспомогательные классы из `docx.shared`.

Работа со стилями

Использование прямого форматирования (жирный, цвет) для каждого фрагмента текста может быть утомительным. Гораздо эффективнее использовать стили. `python-docx` поддерживает встроенные стили Word (Normal, Heading 1, List Bullet и т.д.), а также позволяет создавать свои.

```python

from docx import Document

from docx.enum.style import WD_STYLE_TYPE

doc = Document()

Применение встроенного стиля к абзацу

doc.add_paragraph('Это обычный текст', style='Normal')

doc.add_paragraph('Это маркированный список', style='List Bullet')

Создание собственного стиля

custom_style = doc.styles.add_style('MyCustomStyle', WD_STYLE_TYPE.PARAGRAPH)

Настройка шрифта стиля

custom_style.font.name = 'Arial'

custom_style.font.size = Pt(12)

custom_style.font.bold = True
```

custom_style.paragraph_format.space_after = Pt(10) Отступ после абзаца

Применение созданного стиля

doc.add_paragraph('Текст с пользовательским стилем', style='MyCustomStyle')

doc.save('styled_document.docx')

```

Использование стилей делает код чище, а документы — профессиональнее.

Добавление таблиц

Таблицы — неотъемлемая часть многих документов, особенно отчетов. `python-docx` предоставляет удобные методы для их создания и заполнения.

```python

from docx import Document

doc = Document()

Добавляем заголовок

doc.add_heading('Динамика продаж', level=2)

Создаем таблицу: 3 строки, 4 столбца

```
table = doc.add_table(rows=3, cols=4)
```

Включаем отображение границ (по умолчанию они могут быть не видны)

```
table.style = 'Table Grid'
```

Заполняем шапку таблицы (первая строка)

```
header_cells = table.rows[0].cells
header_cells[0].text = 'Товар'
header_cells[1].text = 'Январь'
header_cells[2].text = 'Февраль'
header_cells[3].text = 'Март'
```

Заполняем данные

```
data = [
    ('Ноутбуки', '150', '165', '180'),
    ('Мыши', '300', '310', '295')
]
row_index = 1
for item in data:
```

```
    row_cells = table.rows[row_index].cells
    row_cells[0].text = item[0]
    row_cells[1].text = item[1]
    row_cells[2].text = item[2]
    row_cells[3].text = item[3]
    row_index += 1
doc.save('report_with_table.docx')
```

Этот код создает аккуратную таблицу с данными. При необходимости вы можете программно добавлять или удалять строки и столбцы, а также получать доступ к любой ячейке для форматирования текста внутри нее.

Добавление изображений

Визуальная информация часто бывает полезна. `python-docx` позволяет легко вставлять изображения, управляя их размером.

```python
from docx import Document
from docx.shared import Cm
doc = Document()
```

doc.add_heading('Визуализация данных', level=1)

Добавляем абзац перед картинкой

doc.add_paragraph('График продаж по месяцам:')

Вставляем изображение. Важно: файл 'chart.png' должен существовать.

Указываем ширину картинки (высота подстроится пропорционально)

doc.add_picture('chart.png', width=Cm(15))

doc.add_paragraph('Рисунок 1. Динамика продаж.')

doc.save('report_with_image.docx')

```

Чтение и модификация существующих документов

Возможности `python-docx` не ограничиваются созданием новых файлов. Вы можете открыть существующий документ и использовать его как шаблон. Это, пожалуй, самая мощная функция для генерации однотипных документов (договоров, актов, справок).

Представьте, что у вас есть файл `template.docx` с текстом: "Уважаемый {Имя}! Ваш баланс составляет {Баланс} рублей."

```python

from docx import Document

```
Открываем документ-шаблон
doc = Document('template.docx')
Данные для подстановки
client_name = "Иван Петрович"
client_balance = "15 000"
Проходим по всем абзацам документа
for paragraph in doc.paragraphs:
    Заменяем плейсхолдеры в тексте абзаца
    if '{Имя}' in paragraph.text:
        paragraph.text = paragraph.text.replace('{Имя}', client_name)
    if '{Баланс}' in paragraph.text:
        paragraph.text = paragraph.text.replace('{Баланс}', client_balance)
```

Замена в inline-части текста (run) может быть сложнее, так как текст разбит на runs. Простейший способ выше работает, но сбрасывает форматирование абзаца. Более продвинутый метод - итерироваться по runs.

```
Сохраняем результат как новый документ
doc.save('final_letter.docx')
```

```

Прямая замена `paragraph.text` сбрасывает форматирование абзаца. Для сохранения форматирования при замене нужно работать на уровне `runs`, что требует более аккуратного кода, но принцип остается тем же.

Библиотека `python-docx` является незаменимым инструментом в арсенале Python-разработчика, которому приходится иметь дело с офисными документами. Она предоставляет чистый и понятный интерфейс для создания сложных структурированных документов, будь то ежемесячные отчеты, коммерческие предложения, дипломы или накладные.

Основные преимущества использования `python-docx`:

- Экономия времени: Автоматизация рутинных операций по созданию и форматированию.
- Исключение ошибок: Программная генерация исключает человеческий фактор при подсчете итогов или подстановке данных.
- Гибкость: Возможность использовать шаблоны, что позволяет разделить дизайн документа и логику его наполнения.
- Кроссплатформенность: Не требует установленного Microsoft Office, работает на Windows, Linux и macOS.
```

Описание программы

Основная идея программы — помочь учителям английского языка сэкономить время на составление проверочных работ по неправильным глаголам.

Вместо ручного подбора слов и форматирования документа учитель получает готовые варианты за пару кликов. Освободившееся время можно потратить на проверку тетрадей или другие важные задачи.

Этапы разработки программы:

1. Написание скрипта генерации глаголов
2. Генерация двух вариантов
3. Настройка интерфейса и главного меню (рабочее окно, кнопки, навигация)
4. Разработка основного рабочего окна
5. Тестирование и подготовка файла с описанием программы

Функционал программы: Пользователь открывает программу и появляется приветственное окно. В нём он может, как начать работу с программой нажав кнопку "Начать" как и посмотреть информацию о создателе (мне), нажав на кнопку с человечком и буквой i. После нажатия на кнопку начать приветственное окно закрывается и открывается основное рабочее окно программы где он может непосредственно сгенерировать работу указав параметры генерации и нажав на кнопку "Сгенерировать". Указав количество глаголов (до 120 шт. включительно) пользователь может поставить галочку рядом с надписью "2 варианта"

чтобы программа сгенерировала две работы разных вариантов. После указания всех параметров генерации пользователь нажимает кнопку "Сгенерировать" и программа предлагает выбрать в открывшемся окне путь сохранения сгенерированной работ и дать названия файлов. После выбора места сохранения пользователь может найти эти работы, там где он указал.

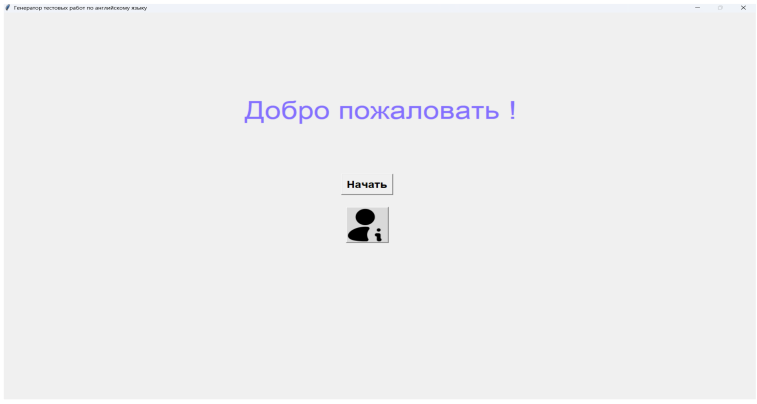

Рис.1. Приветственное окно

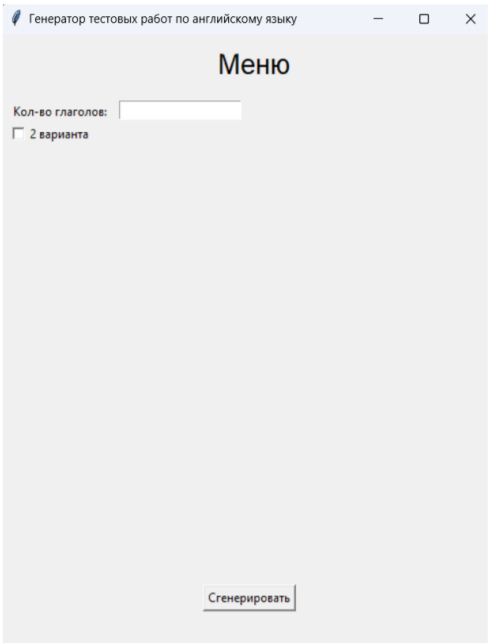

Рис.2. Основное рабочее окно

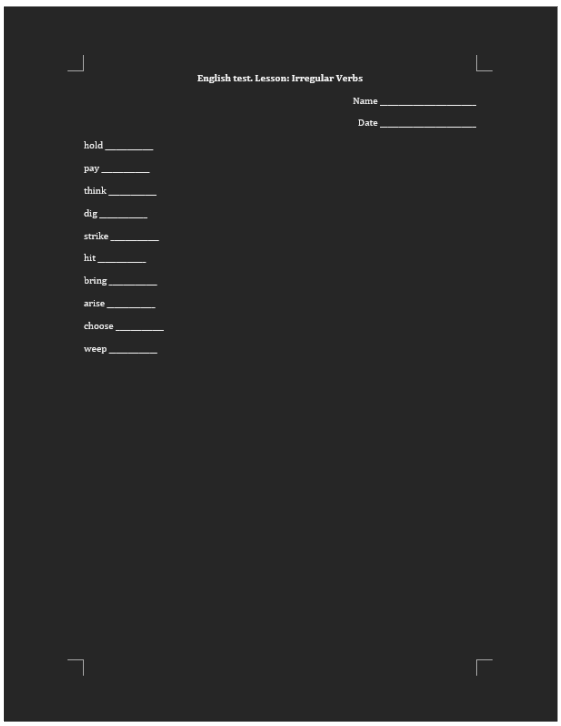

Рис.3. Результат программы

Текст программы

```
from tkinter import*

from tkinter.messagebox import *

import os

from datetime import datetime

import tkinter as tk

from docx import Document

from tkinter.filedialog import *

from verbs_database import irregular_verbs

import random

from tkinter import messagebox

# Приветствие в консоли

username = os.getlogin()

now = datetime.now()

print('Доброго времени суток', username,'!', '\n')

print(f'Московское время:', now.strftime('%H:%M:%S'))

print(f'Дата:', now.strftime('%Y-%m-%d'), '\n')
```

```
print('ВНИМАНИЕ !\nВы используйте BETA-версию приложения. В этой версии могут быть ошибки !')

# Начальный экран

# Настройка окна

root = Tk()

root.title("Генератор тестовых работ по английскому языку")

root.state('zoomed')

root.resizable(False, False)

# Код начального экрана

#1. Переменные

#TODO Сделать чтобы автоматом определал где изображение

info_image =
PhotoImage(file=r"C:\Users\leont\OneDrive\Документы\Прогаммы и коды\Python\School Project\images\info.png")

#2. Функции

def information():

    showinfo('Информация', 'Название приложения: "Генератор тестовых работ по \nанглийскому языку"\n \nРазработчик: Леонтий Ежов/Leo\n \nВерсия: BETA')
```

```
start_menu = None

def start():
    #Настройка окна
    global start_menu
    start_menu =  Toplevel(root)
    root.withdraw()
    start_menu.geometry('500x600')
    #Поля ввода
    koli4estvo = Entry(start_menu)
    koli4estvo.place(x=120, y=65)
    # Флажок "Несколько вариантов"
    neskilko_variantov = StringVar()
    neskolko_variantov = Checkbutton(start_menu, text="2 варианта")
    neskolko_variantov.place(x=7, y=85)
    # Надпсиси
    nadpis_menu = Label(start_menu, text = "Меню", font="Arial 20")
    nadpis_menu.place(x=217, y=10)
```

```
koli4estvo_nadpis = Label(start_menu, text="Кол-во глаголов:")
koli4estvo_nadpis.place(x=10, y=65)
# Делаем проверки
def check_generate():
    kolvo = koli4estvo.get()
    #Здесь было int (kolvo) > 120
    #TODO Доделать чтобы писалась конкретная ошибка
    if kolvo.isdigit() and int(kolvo) <= 120:
        save_path = asksaveasfilename(
            defaultextension=".doc",
            filetypes=[("Text files", "*.doc"), ("All files", "*.*")])
        doc = Document()  # Создание документа
        stroka_zagolovka = doc.add_paragraph()  # Добавление параграфа
        stroka_zagolovka.alignment = 1  # Выравивание текста
        zagolovok_doca_text = "English test. Lesson: Irregular Verbs"
        zagolovok = stroka_zagolovka.add_run(zagolovok_doca_text)
        zagolovok.bold = True
```

```
            all_verbs = list(irregular_verbs.keys())
            chosen_verbs_by_random = random.sample(all_verbs,
int(koli4estvo.get()))
            for verb in chosen_verbs_by_random:
                dans = irregular_verbs[verb]
                doc.add_paragraph(f"{verb}" " " "_________")
                doc.save(save_path)
        else:
            messagebox.showerror("Ошибка генерации", "Вы оставили поле
пустым или привысили максимальное количество глаголов (120)")
    #Кнопки
    Generate = Button(start_menu, text='Сгенерировать',
command=check_generate)
    Generate.place(x=205, y=540)
#3. Надпись Добро Пожаловать !
welcome_label = Label(root,text="Добро пожаловать !", fg='LightSlateBlue',
font="Arial 45")
welcome_label.place(x=490, y=200)
#4.Кнопка Начать
```

```
na4at_button = Button(root, text="Начать", font=("Arial", 18, "bold"),
command=start)

na4at_button.place(x=690, y=390)

#6. Кнопка информации

info_button = Button(root, image=info_image, width='81', height='81',
command=information)

info_button.place(x=700, y=470)

#width=50, height=10

root.mainloop()
```