def most\_common\_word(self):

"""

Find the most common word in the text.

:return: The most frequent word

"""

Определяется метод most\_common\_word внутри класса (например, класса Text).

Он ищет самое часто встречающееся слово в тексте, который хранится в self.text.

В строке документации (docstring) описывается назначение метода.

python

Копировать код

words = self.text.split()

self.text — это строка, в которой хранится текст.

Метод .split() разбивает строку по пробелам и возвращает список слов.

Например:

python

Копировать код

"hello world hello".split() → ["hello", "world", "hello"]

python

Копировать код

frequency = {}

Создаётся пустой словарь frequency.

В нём будем хранить количество вхождений каждого слова:

ключ — слово

значение — сколько раз оно встретилось

python

Копировать код

for word in words:

frequency[word] = frequency.get(word, 0) + 1

Цикл перебирает все слова из списка words.

frequency.get(word, 0) — ищет слово в словаре:

если слово уже есть → возвращает его текущее количество;

если слова ещё нет → возвращает 0.

К этому числу прибавляется 1 и сохраняется обратно.

Пример:

python

Копировать код

words = ["hello", "world", "hello"]

# после первого слова

{"hello": 1}

# после второго слова

{"hello": 1, "world": 1}

# после третьего слова

{"hello": 2, "world": 1}

python

Копировать код

most\_common = max(frequency, key=frequency.get)

Функция max() ищет максимум.

frequency.get передаётся как ключ для сравнения → значит, max() будет искать слово, у которого самое большое значение (частота).

Например, для словаря {"hello": 2, "world": 1} результат будет "hello".

python

Копировать код

return most\_common

Метод возвращает самое частое слово.

✅ Итог:

Метод most\_common\_word() ищет и возвращает слово, которое встречается чаще всего в тексте.

📌 Пример работы:

python

Копировать код

self.text = "apple banana apple orange apple banana"

→ words = ["apple", "banana", "apple", "orange", "apple", "banana"]

→ frequency = {"apple": 3, "banana": 2, "orange": 1}

→ most\_common = "apple"

def unique\_words(self):

"""

Get a list of unique words in the text.

:return: List of unique words

"""

Определяется метод unique\_words, который возвращает список уникальных слов из текста, хранящегося в self.text.

В docstring написано: метод возвращает список уникальных слов.

python

Копировать код

words = self.text.split()

Берём текст (self.text) и разбиваем его методом .split() на список слов.

По умолчанию .split() разделяет по пробелам и символам перевода строки.

Например:

python

Копировать код

"cat dog cat".split() → ["cat", "dog", "cat"]

python

Копировать код

unique = set(words)

Создаётся множество (set) из списка слов.

Особенность множества: оно хранит только уникальные элементы.

Повторяющиеся слова автоматически убираются.

python

Копировать код

["cat", "dog", "cat"] → {"cat", "dog"}

python

Копировать код

return list(unique)

Множество преобразуется обратно в список.

Возвращается список уникальных слов.

Порядок может быть любым, так как множества не сохраняют порядок.

✅ Итог:

Метод unique\_words() возвращает список слов из текста, но каждое слово встречается только один раз.

Разберём метод построчно и объясним, **зачем здесь @classmethod** и как это работает.

@classmethod

def from\_file(cls, file\_path):

"""

Create a Text instance from a file.

:param file\_path: Path to the text file

:return: Text object

"""

with open(file\_path, "r") as file:

content = file.read()

return cls(content)

## Что происходит

1. @classmethod  
   Декоратор делает метод **методом класса**. Вместо привычного self (экземпляр) он получает первый аргумент cls — **сам класс**.
   * Это удобно для **альтернативных конструкторов**: метод создаёт объект не напрямую через Text(...), а через cls(...).
   * Важно: если вы вызовете SubText.from\_file(...) у подкласса, то вернётся **экземпляр подкласса**, потому что используется cls, а не жёстко прописанный Text.
2. def from\_file(cls, file\_path):  
   Объявление метода. file\_path — путь к текстовому файлу.
3. Docstring  
   Коротко описывает назначение: создать объект Text (или его подкласса) из файла.
4. with open(file\_path, "r") as file:  
   Открываем файл в текстовом режиме чтения ("r"). Конструкция with — это **контекстный менеджер**: файл гарантированно закроется, даже если произойдёт ошибка.
5. content = file.read()  
   Читаем **весь файл целиком** в одну строку content. Если файл большой, это будет одна большая строка.
6. return cls(content)  
   Возвращаем новый объект, вызывая конструктор класса (cls) с прочитанным содержимым. Эквивалентно Text(content), но работает корректно и для наследников.

## Почему не @staticmethod?

@staticmethod не получает cls, и внутри пришлось бы писать Text(content). Тогда вызов у подкласса всё равно вернёт Text, а не подкласс. @classmethod решает эту проблему.

## Возможные исключения и нюансы

* FileNotFoundError — если файла нет.
* PermissionError — нет прав на чтение.
* UnicodeDecodeError — если кодировка файла не совпадает с системной по умолчанию.
* Чтение целиком (read()) может быть неэффективно для очень больших файлов.

## Болеe надёжный вариант

Добавим явную кодировку UTF-8 и обработку ошибок чтения:

from pathlib import Path

class Text:

def \_\_init\_\_(self, text: str):

self.text = text

@classmethod

def from\_file(cls, file\_path: str | Path) -> "Text":

path = Path(file\_path)

# Проверка существования и файла

if not path.is\_file():

raise FileNotFoundError(f"Файл не найден: {path}")

# Явно задаём кодировку и политику ошибок

with path.open("r", encoding="utf-8", errors="strict") as f:

content = f.read()

return cls(content)

Если файлы бывают в разных кодировках, можно использовать errors="replace" или попытаться определить кодировку заранее (например, через chardet/charset-normalizer — это уже доп. зависимость).

## Пример использования

text = Text.from\_file("data/article.txt")

print(len(text.text)) # длина прочитанного текста

Итого: это **альтернативный конструктор**, который читает содержимое файла и возвращает **экземпляр текущего класса** (или подкласса), благодаря @classmethod и использованию cls(...).

no\_punct = self.text.translate(str.maketrans("", "", string.punctuation))

1. string.punctuation

Это константа из модуля string.

Она содержит все стандартные знаки пунктуации:

!"#$%&'()\*+,-./:;<=>?@[\]^\_`{|}~

2. str.maketrans("", "", string.punctuation)

Функция str.maketrans() создаёт таблицу преобразования символов.

Синтаксис:

str.maketrans(замены, удаляемые\_символы)

В нашем случае:

первые два аргумента — пустые строки (""), значит никаких замен символов нет;

третий аргумент — string.punctuation, значит все эти символы нужно удалить.

👉 В итоге создаётся таблица перевода, которая говорит: «удалить все знаки пунктуации».

3. self.text.translate(...)

Метод .translate() применяет эту таблицу преобразования к строке.

Все знаки пунктуации удаляются, остальные символы остаются без изменений.

4. Присвоение результата

no\_punct = ...

В переменной no\_punct будет храниться строка без знаков пунктуации.

✅ Пример:

import string

text = "Hello, world! How's it going?"

no\_punct = text.translate(str.maketrans("", "", string.punctuation))

print(no\_punct)

Результат:

Hello world Hows it going

👉 Итого: эта строка кода удаляет все символы пунктуации из текста.

Хочешь, я покажу, как переписать этот код с использованием re.sub, чтобы можно было удалять не только пунктуацию, но и, например, цифры или спецсимволы?

**1. re.sub(pattern, repl, string)**

* Это функция из модуля **re** (регулярные выражения).
* Она заменяет все вхождения **шаблона pattern** в строке string на подстроку repl.

В нашем случае:

* pattern = r"[^A-Za-z0-9\s]", т.е. оставляем
* repl = "" (пустая строка, т.е. удаляем)
* string = self.text (наш текст).

**2. Регулярное выражение r"[^A-Za-z0-9\s]"**

Разберём символы внутри [...] (это класс символов):

* ^ (в начале) → отрицание: «всё, кроме перечисленного».
* A-Z → латинские заглавные буквы.
* a-z → латинские строчные буквы.
* 0-9 → цифры.
* \s → пробельные символы (пробелы, табуляции, переносы строк).

👉 В итоге шаблон означает:  
**«Любой символ, который НЕ является буквой (A-Z, a-z), цифрой (0-9) или пробелом»**.

**3. Замена**

"", self.text

* Все символы, которые соответствуют этому шаблону, заменяются на пустую строку "".
* То есть они удаляются.

**4. Результат**

cleaned\_text — это строка из self.text, но в ней остались только:

* латинские буквы,
* цифры,
* пробелы.

Все знаки пунктуации, спецсимволы, эмодзи и прочее удаляются.

✅ **Пример:**

import re - module (regular expressions) t

text = "Hello, world! 123 :)"

cleaned\_text = re.sub(r"[^A-Za-z0-9\s]", "", text)

print(cleaned\_text)

Результат:

Hello world 123

👉 Итого: эта строка кода очищает текст от **всех символов, кроме латинских букв, цифр и пробелов**.

**r"..." — это raw string (сырая строка)**

* Обычно в строках Python символ \ (обратный слеш) используется как управляющий символ:
* "\n" → перевод строки
* "\t" → табуляция
* Но в регулярных выражениях (модуль re) обратный слеш используется очень часто — например, \d, \s, \w.

Если писать обычную строку, придётся экранировать слеши:

"\\d" → это реально `\d` в regex

С r"..." всё проще:

r"\d" → это `\d` (не нужно писать двойной слеш)

r перед строкой говорит Python: **«воспринимай строку как есть, не обрабатывай \ как спецсимволы»**.(r = raw)