УДК 004

Сравнение производительности Python и Rust на примере задачи обработки строк

Гиренко Александр Евгеньевич
Приамурский государственный университет им.Шолом-Алейхема
Студент

Аннотация
В статье проводится сравнительный анализ производительности двух языков программирования — Python и Rust — на примере задачи подсчёта частоты слов в текстовом файле размером 20 МБ. Рассматриваются особенности работы интерпретируемого и компилируемого языков, измеряется время выполнения. Эксперимент показал, что Rust выполняет задачу в 1.8 раза быстрее Python (2.38 с против 4.33 с). Полученные результаты объясняются различиями в архитектуре языков, а также тем, что задача является преимущественно I/O-bound (зависит от скорости чтения файла), а не CPU-bound. На основе анализа даются рекомендации по выбору языка для задач обработки данных.
Ключевые слова: Python, Rust, производительность, обработка строк, сравнение языков, I/O-bound, бенчмарк.

Performance comparison of Python and Rust on a string processing task

Girenko Aleksandr Evgen'evich
Sholom-Aleichem Priamursky State University
Student

Abstract
The article provides a comparative performance analysis of two programming languages — Python and Rust — on the task of counting word frequency in a 20 MB text file. The features of interpreted and compiled languages are examined, and execution time is measured. The experiment showed that Rust performs the task 1.8 times faster than Python (2.38 s vs 4.33 s). The obtained results are explained by differences in language architecture, as well as the fact that the task is predominantly I/O-bound (depends on file reading speed) rather than CPU-bound. Based on the analysis, recommendations for choosing a language for data processing tasks are given.
Keywords: Python, Rust, performance, string processing, language comparison, I/O-bound, benchmark.

1 Введение

1.1 Актуальность

Выбор языка программирования — один из ключевых этапов разработки, влияющий на производительность, скорость разработки и поддержку проекта. Python остаётся самым популярным языком для анализа данных и быстрого прототипирования благодаря простому синтаксису и богатой экосистеме библиотек. Rust — современный компилируемый язык, который позиционируется как безопасная и высокопроизводительная альтернатива C/C++. Вопрос о том, насколько Rust быстрее Python на реальных задачах, остаётся предметом дискуссий. Исследование необходимо для выработки практических рекомендаций по выбору языка в зависимости от типа задачи.

1.2 Обзор исследований

В работе Э. Шуберта и Л. Ленссена рассматривается реализация алгоритма кластеризации k-medoids на языках Rust и Python, анализируется производительность двух реализаций и демонстрируются преимущества Rust при работе с большими объёмами данных [1]. С.М. Окулов в учебном пособии систематизирует алгоритмы обработки строк, рассматривая классические подходы к поиску, сортировке и анализу текстовых данных [2]. В работе Д.А. Кайшева рассматривается новое поколение языков программирования на примере Rust. Автор анализирует мультипарадигменность языка, поддержку функционального, обобщённого и императивного программирования, а также преимущества системы владения и безопасности памяти, обеспечивающих высокую производительность без сборщика мусора [3]. И. Бузения, Б.П. Кришан и М. Прадель представляют PyBench — бенчмарк исполняемого программного обеспечения на Python, описывая методику оценки и сравнения различных Python-проектов [4]. С.Б. Озгюр и А. Кой рассматривают примеры алгоритмов на финансовых рынках, включая анализ доходности активов, бенчмаркинг и бэктестинг с использованием языка Python [5].

1.3 Цель исследования

Целью работы является сравнительный анализ производительности Python и Rust на задаче подсчёта частоты слов в текстовом файле, а также объяснение полученных результатов с точки зрения архитектуры языков.

2 Материалы и методы

2.1 Тестовые данные

Для эксперимента сгенерирован текстовый файл размером 20 МБ, содержащий 2 000 000 строк (20 000 000 слов). Словарь состоял из 1000 уникальных слов, что обеспечило равномерное распределение частот и позволило корректно оценить скорость обработки. В таблице 1 представлены характеристики тестового файла.

Таблица 1 — Характеристика тестового файла

Параметр	Значение
Размер файла	~20 МБ
Количество строк	2 000 000
Слов в строке	10
Длина слова	3–10 символов
Всего слов	20 000 000
Уникальных слов	~1000

2.2 Задача

Задача заключается в подсчёте частоты встречаемости каждого слова в файле. Алгоритм:

1. Открыть файл.
2. Прочитать строку.
3. Привести строку к нижнему регистру.
4. Разбить строку на слова.
5. Для каждого слова увеличить счётчик в словаре.

2.3 Инструментарий

Для Python использовался интерпретатор CPython 3.12. Для Rust — компилятор rustc с флагом оптимизации -O. Оба языка запускались в Google Colab (Linux, 2 vCPU, 12 ГБ ОЗУ).

2.4 Методика измерения

Время выполнения замерялось с помощью стандартных средств: time.time() в Python и Instant::now() в Rust. Замер охватывал только этап обработки файла (без учёта генерации).

3 Результаты

3.1 Время выполнения

Результаты сравнения времени выполнения представлены в таблице 2.

Таблица 2 — Сравнение времени выполнения

Язык	Время выполнения (с)	Уникальных слов
Python	4.33	~1000
Rust	2.38	~1000
Разница	1.8×	—

Python выполнил задачу за 4.33 секунды, Rust — за 2.38 секунды. Python оказался медленнее в 1.8 раза.

На рисунке 1 представлена столбчатая диаграмма сравнения времени выполнения.

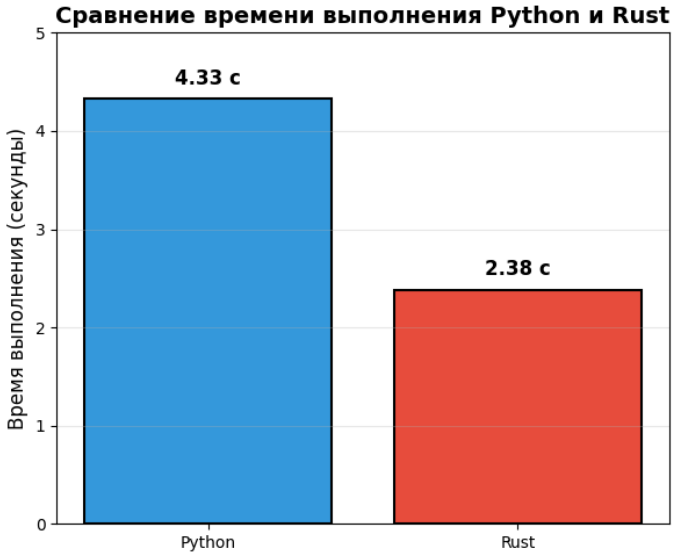


Рисунок 1 — Сравнение времени выполнения Python и Rust

3.2 Анализ результатов

Полученный разрыв (1.8×) существенно меньше, чем можно ожидать от сравнения интерпретируемого и компилируемого языков (обычно 10–50×). Это объясняется тем, что задача является I/O-bound, а не CPU-bound:

- Основное время тратится на чтение файла с диска, а не на вычисления.
- Python и Rust используют одинаковые системные вызовы для чтения файлов.
- Компилируемый код Rust не даёт преимущества на этапе ожидания данных от диска.

Однако даже на такой задаче Rust показал почти двукратное преимущество за счёт:

1. Отсутствия сборщика мусора — Rust не тратит время на управление памятью в рантайме.
2. Эффективной работы со строками — итераторы Rust компилируются в оптимизированный машинный код.
3. Отсутствия накладных расходов интерпретатора — Python выполняет байт-код, что медленнее нативной компиляции.

3.3 Листинг программной реализации

Для обеспечения воспроизводимости результатов на рисунках 2 и 3 представлен код на Python и Rust. На рисунке 4 представлен результат программы.

```python
import random
import string

# Генерируем словарь из 1000 случайных слов
words = [''.join(random.choices(string.ascii_lowercase, k=random.randint(3, 10))) for _ in range(1000)]

# Создаём файл ~20 МБ
with open('large_text.txt', 'w') as f:
    for _ in range(2_000_000):
        f.write(' '.join(random.choices(words, k=10)) + '\n')

print("Файл создан!")
```

```
Файл создан!
```

```python
import time

def count_words_python(filename):
    word_count = {}
    with open(filename, 'r') as f:
        for line in f:
            for word in line.lower().split():
                word_count[word] = word_count.get(word, 0) + 1
    return word_count

start = time.time()
result_py = count_words_python('large_text.txt')
end = time.time()

print(f"Python: {end - start:.2f} секунд")
print(f"Уникальных слов: {len(result_py)}")
```

Рисунок 2 — Листинг на Python

```
rust_code = '''
use std::collections::HashMap;
use std::fs::File;
use std::io::{BufRead, BufReader};
use std::time::Instant;

fn count_words_rust(filename: &str) -> HashMap<String, u32> {
    let file = File::open(filename).unwrap();
    let reader = BufReader::new(file);
    let mut word_count = HashMap::new();

    for line in reader.lines() {
        let line = line.unwrap();
        for word in line.to_lowercase().split_whitespace() {
            *word_count.entry(word.to_string()).or_insert(0) += 1;
        }
    }
    word_count
}

fn main() {
    let start = Instant::now();
    let result = count_words_rust("large_text.txt");
    let duration = start.elapsed();
    println!("Rust: {:?}", duration);
    println!("Уникальных слов: {}", result.len());
}
'''

with open('main.rs', 'w') as f:
    f.write(rust_code)

print("Rust-код записан в main.rs")
```

```
Rust-код записан в main.rs
```

```
!rustc -O main.rs -o rust_program
print("Компиляция завершена!")
```

```
Компиляция завершена!
```

```
!./rust_program
```

Рисунок 3 — Листинг на Rust

```
Python: 4.33 секунд
Уникальных слов: 1000
```

```
Rust: 2.382622699s
Уникальных слов: 1000
```

Рисунок 4 — Результат выполнения

4 Обсуждение

4.1 Интерпретация результатов

Разрыв в 1.8 раза — это значимое преимущество, но не настолько драматичное, как в классических бенчмарках (например, вычисление чисел Фибоначчи, где Rust быстрее в десятки раз). Это подтверждает важный принцип: выбор языка должен зависеть от типа задачи.

Для задач, где основное время тратится на ввод-вывод (чтение файлов, работа с сетью, базами данных), разница между Python и Rust сокращается.

Для задач, требующих интенсивных вычислений (математика, обработка изображений, криптография), Rust даёт многократное преимущество.

4.2 Сравнение с альтернативными подходами

На рынке существуют и другие подходы к ускорению Python:

- PyPy — JIT-компилятор Python, который может ускорить код в 2–5 раз без переписывания.
- Cython — компиляция Python-кода в C, требует аннотаций типов.
- Numba — JIT-компиляция для числовых вычислений.

Rust выигрывает у этих подходов за счёт безопасности памяти и отсутствия накладных расходов на рантайм, но проигрывает в скорости разработки.

4.3 Влияние типа задачи на выбор языка

На основе полученных результатов можно сформулировать следующее правило (табл.3).

Таблица 3 — Выбор языка в зависимости от типа задачи

Тип задачи	Рекомендуемый язык
Быстрое прототипирование, анализ данных	Python
I/O-bound задачи (чтение файлов, API)	Python или Rust
CPU-bound задачи (вычисления, обработка сигналов)	Rust
Системное программирование, встраиваемые системы	Rust
Машинное обучение (обучение моделей)	Python (с Rust-расширениями)

5 Выводы

В ходе проведённого исследования реализован полный цикл сравнительного анализа производительности Python и Rust на задаче подсчёта частоты слов в текстовом файле размером 20 МБ.

Установлено, что Rust выполняет задачу в 1.8 раза быстрее Python (2.38 с против 4.33 с). Разрыв оказался меньше ожидаемого из-за того, что задача является I/O-bound: основное время тратится на чтение файла, а не на вычисления. Тем не менее, Rust показал преимущество за счёт отсутствия сборщика мусора, эффективной работы со строками и нативной компиляции.

Полученные результаты позволяют сделать вывод: выбор языка программирования должен определяться типом задачи. Python остаётся оптимальным выбором для быстрого прототипирования и I/O-bound задач, тогда как Rust даёт значительное преимущество на CPU-bound задачах и в системном программировании.

Библиографический список

1. Schubert E., Lenssen L. Fast k-medoids clustering in Rust and Python // Journal of Open Source Software. 2022. Vol. 7. № 75. P. 4183. DOI: 10.21105/joss.04183. URL: https://www.elibrary.ru/item.asp?id=57164536
2. Окулов С.М. Алгоритмы обработки строк: учебное пособие. М.: БИНОМ. Лаборатория знаний, 2012. 413 с. URL: https://www.elibrary.ru/item.asp?id=20241346
3. Кайшев Д.А. Новое поколение языков программирования – Rust // StudNet. 2021. Т. 4. № 6. С. 147. URL: https://www.elibrary.ru/item.asp?id=46318022
4. Bouzenia I., Krishan B.P., Pradel M. PyBench: A benchmark of executable Python software // Proceedings of the ACM on Software Engineering. 2024. Vol. 1. № FSE. P. 338–358. DOI: 10.1145/3643742. URL: https://www.elibrary.ru/item.asp?id=73610903
5. Ozgur S.B., Koy A. Algorithm examples in financial markets: assets income analyzing, benchmark and backtesting with Python programming // Journal of Business, Economics and Finance. 2020. Vol. 12. № 1. P. 95–97. DOI: 10.17261/pressacademia.2020.1364. URL: https://www.elibrary.ru/item.asp?id=75640217