Rust in the data science and machine learning stack

c@pgdm.ch

May 14, 2025

PyData Zürich Meetup

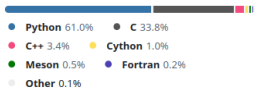
Contents

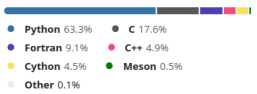
Introduction

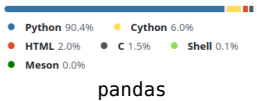
Compiled languages in the Python ecosystem

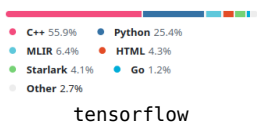
"I thought this was **Py**Data!"

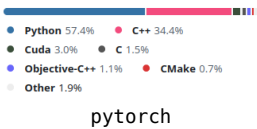
- The Python scientific and numerical ecosystem relies heavily on compiled languages for performance-critical parts as well as interfacing with hardware.

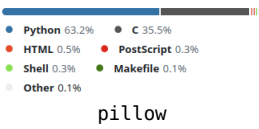












(line-of-code count for illustrative purposes)

Introduction

Compiled languages in the Python ecosystem

Python as a convenient interface / "glue language" to native code.

> *"Program development using Python is 5-10 times faster than using C/C++, and 3-5 times faster than using Java.*
>
> *In many cases, a prototype of an application can be written in Python without writing any C/C++/Java code. Often, the prototype is sufficiently functional and performs well enough to be delivered as the final product, saving considerable development time. [...]*
>
> *The best approach is often to write only the performance-critical parts of the application in C++ or Java, and use Python for all higher-level control and customization."*
>
> — *Guido van Rossum,*
> *"Glue It All Together With Python", Workshop on Compositional Software Architecture in Monterey, California, January 6-8, 1998.*

Introduction

Rust in the Python ecosystem

The past few years have seen the emergence of:

- High-quality (performance, API) Python packages using Rust rather than C or C++, e.g.
  - `polars` DataFrame (pandas alternative; 2021).
  - HuggingFace `tokenizers` (2020).
  - Rust rewrite for version 2 of `pydantic` (2023).
  - `orjson`, fast JSON serialization/deserialization (2018).
- New development tooling written in Rust, with rapid adoption thanks to their performance and UX.
  - `ruff` (formatter; 2022), `uv` (package/project manager; 2024), and upcoming type checker, from Astral.
  - `py-spy` (profiler; 2019).

Introduction

Goals of the talk

- Give a brief overview of the Rust programming language.
- Study the cases of `polars`, `tokenizers`, `pydantic`, `ruff` and `uv`, analyzing the main reasons for their success.
  - Rust is a great alternative to C, C++ or Cython for easily writing high-performance Python modules!
- Demonstrate how to write a Python module in Rust, through a practical example.
  - Simple BPE tokenizer.
- Discussion, e.g. good or less good use cases for Rust, especially in the ML/DS stack.

Contents

The Rust programming language

https://rust-lang.org

- First version in 2012, now at 1.86.0 (release every 6 weeks).
- Backed by the Rust foundation and its members
- MIT & Apache 2.0 license (like most of the ecosystem).
- Emphasis on performance, type safety, memory safety, *and* developer experience.

- Wide and expanding adoption:
  - Industry: Amazon, Google, Meta, Microsoft, Cloudflare, Github, Apple, Huawei, Discord, Dropbox...
  - Open source projects: Servo, uv, Ripgrep, Wezterm, Typst, Zed, Helix, ruff, Sccache, Hyperfine, Alacritty, Polars, InfluxDB, Meilisearch, Deno, Linux kernel...

The Rust programming language

> *Rust found a sweet spot: it is just as low-level as C or C++ with all the advantages of these (e.g. control, size, speed, etc.).*
> *At the same time, it is as high-level as Haskell with an amazing amount of functional heritage.*
> *It is still imperative, so quite accessible to most people, and it is just as flexible as Python.*
>
> — *Peter Varo*
> *on https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/*

In my experience*, writing Rust can be as fast and enjoyable** as writing Python, with the massive added benefits of performance and compile-time checks.

* of writing a *lot* of Rust since 2021, and leading a team of up to 15 doing so.
** after some adaptation period, admittedly sometimes a bit frustrating

The Rust programming language

Features

- Basic syntax relatively similar to C++.
- Selected features:
  - Strongly and statically typed
  - Type inference
  - Algebraic data types
  - Immutability by default
  - Pattern matching
  - Move semantics
  - Expression-orientation
  - Trait-based OOP
  - Functional programming
  - Zero-cost abstractions
  - Ergonomic error handling
  - Asynchronous programming
  - Generics

The Rust programming language

Features

- Compilation to machine code via LLVM (or gcc) backend, with many supported targets.
- Excellent built-in tooling:
  - `cargo` (build system and dependency management)
  - `clippy` (linter)
  - `fmt` (formatter)
  - `rust-analyzer` (LSP)
  - `rustdoc` (HTML documentation)
  - `rustc` (compiler), with *excellent* errors

  Nice consequence: uniformity and compatibility throughout the ecosystem.
- Rich and well-documented standard library (via `rustdoc`), e.g. standard library docs.
- Interoperability with other languages (e.g. C, Python via pyo3, C++ via cxx).
- High-quality libraries, centralized on https://crates.io (akin to pypi).
- Plenty of excellent learning resources (see later).

The Rust programming language

First example

```rust
struct Mascot {
    name: String,
}
impl Mascot {
    fn greet(&self) -> String {
        format!("Hello {}, what's up?", self.name)
    }
}

fn main() {
    let mascot = Mascot {
        name: "Ferris".into(),
    };
    println!("{}", mascot.greet());
}
```

```
$ cargo init
$ cargo run -r
Hello Ferris, what's up?
```

The Rust programming language

Second example

```rust
use serde::Serialize;

#[derive(Serialize)]
struct Mascot {
    name: String,
}
impl Mascot {
    fn json(&self) -> String {
        serde_json::to_string_pretty(self).unwrap()
    }
    fn yaml(&self) -> String {
        serde_yaml::to_string(self).unwrap()
    }
}
fn main() {
    let mascot = Mascot {
        name: "Ferris".into(),
    };
    println!("{}", mascot.json());
    println!("{}", mascot.yaml());
}
```

```
$ cargo run -r
```

```
{
  "name": "Ferris",
}
name: Ferris
```

(the serde crate is fantastic!)

The Rust programming language

Third example

Functional programming is great for analytical applications.

```rust
struct User {
    age: u8,
    industry: String,
}
impl User {
    /// Compute the average age from a series of users, grouped by industry.
    fn mean_age_by_industry<'a>(users: impl Iterator<Item = &'a Self>) -> HashMap<String, f64> {
        users
            .filter(|u| !u.industry.is_empty())
            .map(normalize_industry)
            .into_group_map_by(|u| u.industry.clone())
            .into_iter()
            .map(|(industry, users)| {
                let total_age: u64 = users.iter().map(|u| u.age as u64).sum();
                let mean_age = total_age as f64 / users.len() as f64;
                (industry, mean_age)
            })
            .collect()
    }
}
```

The Rust programming language

Learning more about Rust

- The Rust Book
- Rust by Example
- Rust users forum
- A more detailed Rust introductory talk, in 40 slides, with a practical example: https://c.pgdm.ch/notes/brief-tour-rust-talk/
- More advanced:
  - Effective Rust (thanks to Kyrylo for the recommendation!)
  - Rust Performance Book

Contents

Case studies

What advantages for a tool or package written in Rust over Python?

- Performance:
  - No interpreter startup time (10-100 ms, possibly much more with many imports).
  - Compiled and optimized code (LLVM is amazing); 10-100x faster `release` builds.
  - Fine memory management (in particular no garbage collection).
  - Parallel processing without the Global Interpreter Lock (GIL).
    In Python, no parallelism for CPU-bound tasks without spawning a new process and paying the IPC cost.
- Single-binary/library deployment (with easy cross-compilation).
- Compile-time checks (types, static analysis...), avoiding *many* bugs.
- In case of a rewrite or port, insights gathered from the original implementation.
  This should not be neglected in judging a rewrite as much better and performant than the original!

Case studies

In all fairness, most of the advantages above also apply to packages or tools writen in other compiled languages (C, C++, Go...).

However, Rust makes this *easy* and *safe* for the developer:

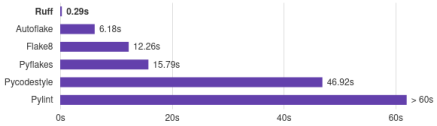
- Built-in tooling, e.g. linter (`clippy`), build system (`cargo`)
- Libraries (crates), accessible directly from `cargo` (à la `pip`).
- Modern language features (error handling, data types, functional programming...).
- Memory safety (borrow checker: lifetimes and mutability), especially useful for writing parallel code.
  If the code compiles, it is usually correct! (modulo deadlocks)
- Easy integration with Python via `pyo3` and `maturin`.

Friction is one of the major factors holding developers back from offloading performance-critical code to a compiled language.

Case studies

ruff

ruff by Astral is a linter and formatter acting as drop-in replacement for black, isort, flake8.
Released in 2022, it is now used by many projects and companies (PyTorch, HuggingFace, Pandas, SciPy...).



Source: https://astral.sh/blog/the-ruff-formatter

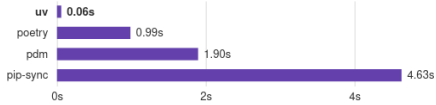
Why is it so much faster?

- No startup latency (Python interepreter, imports...): see the need for blackd.
- Parallel processing (formatting and linting is embarassingly parallel).
- Custom native parser (vs Python lib2to3 in black; C ast for flake8).
- Caching.

Case studies

uv

uv, released in 2024 by Astral (as well!), is a package and project manager for Python, usable as a drop-in replacement for `pip`, `pyenv`, `poetry`, `virtualenv`, etc.



Source: https://docs.astral.sh/uv/

What led to its rapid adoption?

- No startup latency.
- Fast dependency solver (a NP-hard problem), via `pubgrub-rs`.
- Parallel downloads and processing.
- Single binary, able to bootstrap Python.
- 10-100x faster than `pip`.
- Great UX.

(unfair advantage in direct speed comparison benchmarks: bytecode compilation is disabled by default.)

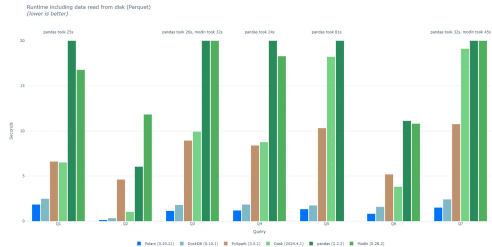
Case studies

`polars` DataFrames

`polars` is a dataframe library released in 2021, with significantly better performance than pandas, and a nice(r) API.

Performance features:

- Apache Arrow columnar in-memory data format.
- Parallelization (threads, SIMD instructions).
- Query planning/optimization.



Source: https://pola.rs/posts/benchmarks/

See:

- https://pola.rs/posts/i-wrote-one-of-the-fastest-dataframe-libraries/
- https://wesmckinney.com/blog/apache-arrow-pandas-internals/
  (*Apache Arrow and the "10 Things I Hate About pandas"* from the creator of pandas)

Case studies

pydantic

Widely used data validation package, first released in 2017. In 2022, the package's core was rewritten in Rust.

The motivations outlined by the authors in https://pydantic.dev/articles/pydantic-v2 are:

- Performance:

  "As a result of the move to Rust for the validation logic (and significant improvements in how validation objects are structured) pydantic V2 will be significantly faster than pydantic V1.
  pydantic V2 is between 4x and 50x faster than pydantic V1.9.1. In general, pydantic V2 is about 17x faster than V1 when validating a model containing a range of common fields."

- Improving architecture and code quality
- Minimizing the likelihood of bugs, especially with respect to error handling

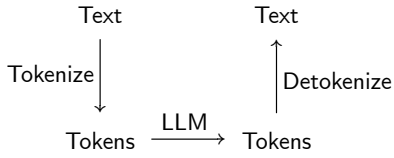
See also the *How Pydantic V2 leverages Rust's Superpowers* 2023 talk from the author of `pydantic`:
https://slides.com/samuelcolvin/deck-0e6306 https://www.youtube.com/watch?v=YynpIOnGcto

Case studies

Tokenizers

In general, tokenizers transform:

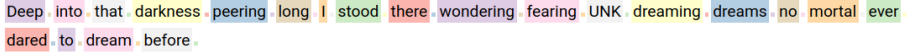
- Text (sequence of words/characters/bytes) into;
- A sequence of numbers (tokens), for use in ML algorithms.



For example, at the two extremes:

- Assign every *character* to a different token:

- Assign every *word* to a different token (larger vocabulary size; smaller sequence lengths):

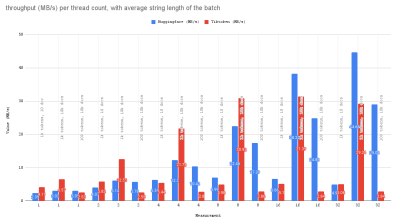


Case studies

Tokenizers

HuggingFace's `tokenizers` package:

> *Extremely fast (both training and tokenization), thanks to the Rust implementation. Takes less than 20 seconds to tokenize a GB of text on a server's CPU.*
>
> — *https://github.com/huggingface/tokenizers*



The other package is `tiktoken`, also written in Rust.

Good use case for implementing the core in a lower-level systems language like Rust or C++:

- Maximize throughput of single-core processing.
- Parallelization over threads with minimal overhead.

(similarly to a dataframe library like `polars`)

Contents

Writing a tokenizer package in Rust

For fun and demonstration purposes, let's now:

- Look at the Byte Pair Encoding (BPE) algorithm for tokenization.
  In a very simplified way, e.g. disregard punctuation, split on spaces...
- Implement a naive version in Python.
- Port the naive version to Rust, and create bindings for Python.

Out of scope: Algorithmically optimal implementations. We will use mostly the same naive algorithm from Python and Rust.

Writing a tokenizer package in Rust

Byte Pair Encoding for tokenization

Byte Pair Encoding (BPE) algorithm, originally described in 1994:

Philip Gage, *A New Algorithm for Data Compression*, The C User Journal, 1994.

- Allows creating a vocabulary with a balanced size, capturing e.g. suffixes, prefixes and stems.
  token ization, wonder ing, doubt ing, that
- Used for example in GPT models.

Two phases:

1. Training from a text corpus and a target vocabulary size
   Can be seen as a greedy algorithm to find a text compression scheme minimizing encoded sequence length.
2. Tokenization of new text.
   Apply rules learnt during training.

Writing a tokenizer package in Rust

Byte Pair Encoding for tokenization

Phase 1 (Training):

1. Start with a small vocabulary: all characters observed in a training corpus.
2. Until the target vocabulary size is reached grow it by:
   - Creating a new token by merging the consecutive pairs of tokens that happen most frequently in corpus words, taking into account only word frequency.
     Example: In English, we will likely start by merging:
     t and h into th, i and n into in, and later th and e into the.
   - The two tokens from the merge may or may not subsist in the vocabulary, depending whether they are still used in other pairs.

Writing a tokenizer package in Rust

Byte Pair Encoding for tokenization

Phase 2 (Tokenization):

1. Split the text into words.
2. Apply the merge rules from the training phase in order
   e.g. Merge successive t and h into th, and then successive th and e into the.
3. Assign token IDs to the resulting strings, use the UNK token if necessary.

de ep into that dar k ness pe er ing long i sto od there wonder ing fe ar ing doub ting dre am ing dre am s no m ort al ever d ared to dre am before

Target vocabulary size: 1000 tokens

deep into that darkness pe ering long i stood there wonder ing fear ing doub ting dream ing dreams no mortal ever dared to dream before

Target vocabulary size: 5000 tokens

Writing a tokenizer package in Rust

Python implementation

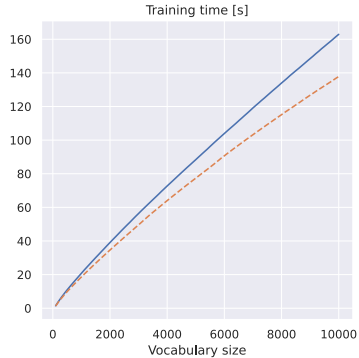
Naive implementation following https://huggingface.co/learn/llm-course/en/chapter6/5, in 150 lines.

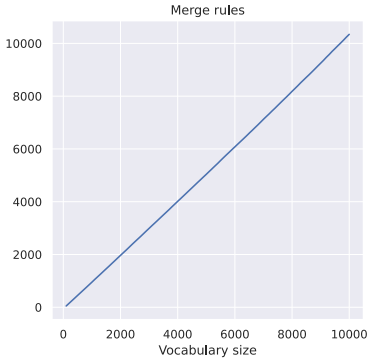
```python
class AbstractTokenizer(ABC):

    @abstractmethod
    def tokenize(self, line: str) -> list[TokenID]: ...

    @abstractmethod
    def tokens_as_strings(self, tokens: list[TokenID]) -> list[Token]: ...

    @staticmethod
    @abstractmethod
    def train(lines: Iterator[str], vocab_size: int) -> AbstractTokenizer: ...
```

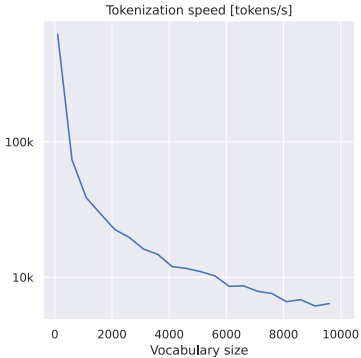
Train it on a small corpus of 500k words from the 5 most popular books on Project Gutenberg.

Writing a tokenizer package in Rust

Python implementation







On our small corpus of 500k words, it takes about 03:30 to train the tokenizer with a vocabulary size of 10k tokens.

With this vocabulary, tokenization speed is about 6k tokens/s (complexity linear in number of merge rules).

The orange curve in training uses an approximation of the vocabulary size to save having to recompute it at each iteration.

Writing a tokenizer package in Rust

Python implementation

Profiling (e.g. with `py-spy`) shows nothing unexpected:

- Training: most of the time is spent computing the frequencies of all pairs.
  (this is less interesting to optimize, because it is essentially a one-off operation).
- Tokenization: most of the time spent finding and applying merges.
  - Editing in the middle of a list of tokens is inefficient.
  - Applying the merges requires iterating over them and over the sequence.

How would you speed either up (out of scope here)?
Careful: the order of the merge rules matters! A word might be formed by composing tokens in multiple ways, but the correct tokenization is unique: programm ing or pro gramm ing?
See for example the HuggingFace `tokenizers` source or this blog post from GitHub or this blog post.

Writing a tokenizer package in Rust

Rust implementation

Let's implement the very same algorithms (training and tokenization) in Rust.

```rust
pub trait Tokenizer: Sized {
    fn tokenize(&self, line: &str) -> impl Iterator<Item = TokenID>;

    fn tokens_as_string<T: IntoIterator<Item = TokenID>>(
        &self,
        tokens: T,
    ) -> impl Iterator<Item = String>;

    fn train(lines: impl IntoIterator<Item = String>, vocab_size: usize) -> Self;
}
```

Writing a tokenizer package in Rust

Rust implementation

```rust
type Pair = (String, String);
type Merges = IndexMap<Pair, String>;

#[derive(Deserialize, Serialize)]
pub struct Tokenizer {
    merges: Merges,
    pub tokens: BiMap<TokenID, Token>,
}

// Implement the tokenizer trait
impl crate::Tokenizer for Tokenizer {
  // ...
}
```

Writing a tokenizer package in Rust

Rust implementation

Normalization with iterators and functional programming:

```rust
fn normalize_line(line: &str) -> String {
    line.chars()
        .filter(|c| c.is_alphanumeric() || c.is_whitespace())
        .flat_map(|c| c.to_lowercase())
        .collect()
}
```

Writing a tokenizer package in Rust

Rust implementation

```rust
fn tokenize(&self, line: &str) -> impl Iterator<Item = TokenID> {
    let mut splits: Vec<Vec<Token>> = normalize_line(line)
        .split_whitespace()
        .map(|word| {
            word.chars()
                .map(String::from)
                .chain(std::iter::once(String::from(" ")))
                .collect()
        })
        .collect();
    for (pair, merge) in &self.merges {
        for split in &mut splits {
            let mut i = 0;
            while i < split.len() - 1 {
                if split[i] == pair.0 && split[i + 1] == pair.1 {
                    split.splice(i..=i + 1, [merge.clone()]);
                } else {
                    i += 1;
                }
            }
        }
    }
    splits
        .into_iter()
        .flatten()
        .map(|token| self.tokens.get_by_right(&token).copied().unwrap_or(UNK))
}
```

(find what could be inefficient in this implementation and rewrite it…or wait to identify what the actual hot path is.)

Writing a tokenizer package in Rust

Rust implementation

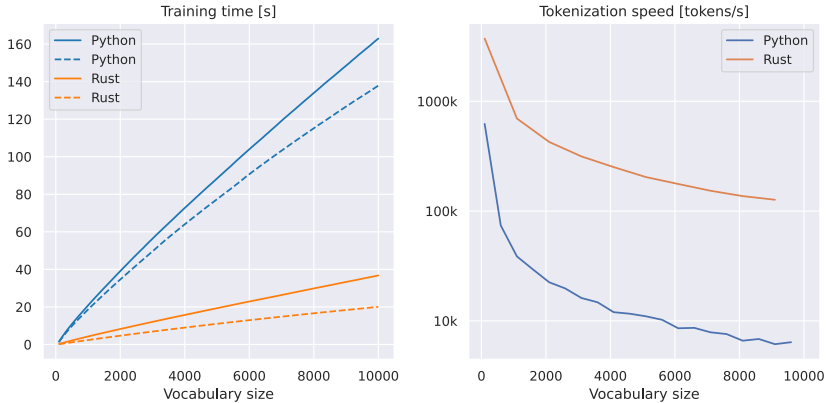
Using the tokenizer:

```rust
let tokenizer = tokenizers::bpe::Tokenizer::train(files, args.vocab_size);
tokenizer.save(file)?;

let doc = std::fs::read_to_string(doc)?;
let tokens: Vec<TokenID> = doc
    .lines()
    .flat_map(|line| tokenizer.tokenize(line))
    .collect();
println!("{:?}", tokens);
```

See the repository for the full code, including a CLI using the `clap` argument parsing crate.

Writing a tokenizer package in Rust

Rust implementation: performance

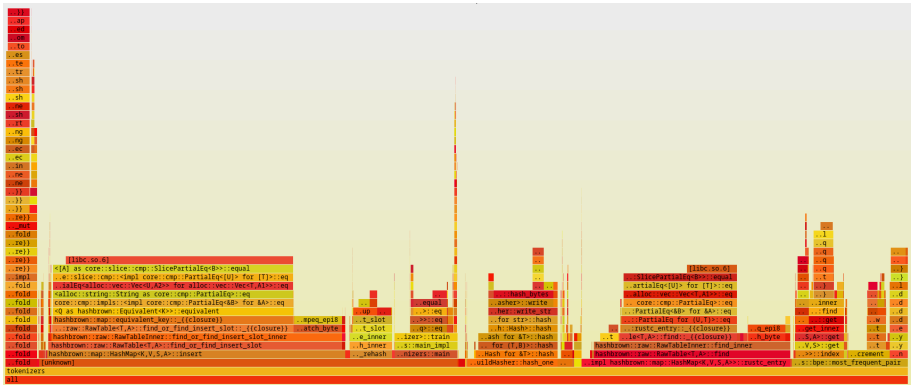


At 10k vocabulary size: 4.4x faster at training and 20x faster at tokenization (100k tokens/s).
(the dashed line in tokenization again use a vocabulary size estimation cheat).

Writing a tokenizer package in Rust

Rust implementation: performance

Profiling of the training step, using `cargo-flamegraph`:



During training, most of the time is spent working with hash maps.

Writing a tokenizer package in Rust

Rust implementation: performance

Profiling of tokenization:

During tokenization, we loop over merges and letters, check for matching pairs, and merge when necessary.

Writing a tokenizer package in Rust

Rust implementation: performance

The implementation has a lot of inefficiencies:

- Unnecessary allocations (`Clones`).
- Suboptimal algorithms.
- No explicit parallelism.

(on the other hand, we use some zero-brainpower tricks: `jemalloc`, enable instruction sets, use fast hashers)

Yet, we achieved a 20x speed gain at tokenization (resp. 4.5x at training), with roughly the same amount of code and complexity:

Language	**Files**	**Lines of code**
Python	4	210
Rust	4	291

Writing a tokenizer package in Rust

Python bindings

Things that Rust is not a great fit for:

- Quick prototyping or data analysis.
- Plotting.
- Designing and training neural networks.
- Rewriting the entire Python ecosystem.

To bridge this gap, let's make our tokenizer available from Python under the same interface as the Python implementation!

```
Tokenizer = bpe_py.Tokenizer if args.tokenizer == "py" else TokenizerRs
tokenizer = Tokenizer.train(iterator, vocab_size=args.vocab_size)
tokens = tokenizer.tokenize_text(f.read())
```

Writing a tokenizer package in Rust

Python bindings

- The `pyo3` crate provides easy-to-use bindings for to call Python from Rust and vice-versa.

  It is used by `pydantic`, HuggingFace `tokenizers`, `orjson`, `tiktoken`...

  – Ergonomic mapping of types (e.g. `list[T]` to `Vec<T>`, `str` to `&str` or `String`, etc.); quasi almost zero-cost in many cases.

  – The crate will be compiled to a shared library (`.so`) that the Python interpreter is able to load as a module. (same as for moduels written in other languages)

- `maturin` simplifies the build process, packaging the modules as Python wheels.

```
$ maturin build -r
  Building a mixed python/rust project
Compiling tokenizers_py v0.1.0
Finished `release` profile [optimized + debuginfo] target(s) in 3.82s
Built wheel for CPython 3.13 to target/wheels/tokenizers-0.1.0-cp313-cp313-manylinux_2_39_x86_64.whl
```

Writing a tokenizer package in Rust

Python bindings

```rust
#[pyclass]
struct Tokenizer {
    inner: tokenizers::bpe::Tokenizer,
}
impl Tokenizer {
  fn tokenize(&self, line: &str) -> Vec<TokenID> {
      self.inner.tokenize(line).collect()
  }
  fn tokens_as_strings(&self, tokens: &Bound<'_, PyAny>) -> PyResult<Vec<String>> {
      let tokens: Vec<TokenID> = tokens
          .try_iter()?
          .map(|t| PyResult::Ok(t?.extract::<TokenID>()?))
          .try_collect()?;
      Ok(self.inner.tokens_as_string(tokens).collect())
  }
 // ...
}

#[pymodule]
fn tokenizers(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_class::<TokenizerPy>()?;
    Ok(())
}
```

```python
from .tokenizers import Tokenizer
```

Development:

```
$ maturin develop -r
$ python -m tokenizers.main
```

Creating a wheel for distribution:

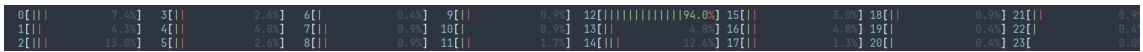
```
$ maturin build -r
$ unzip -l tokenizers-0.1.0.whl | awk
'{print $4}'
  tokenizers-0.1.0.dist-info/METADATA
  tokenizers-0.1.0.dist-info/WHEEL
  tokenizers/main.py
  tokenizers/utils.py
  tokenizers/__init__.py
  tokenizers/bpe.py
  tokenizers/tokenizers.so
  tokenizers-0.1.0.dist-info/RECORD
```

Writing a tokenizer package in Rust

Rust implementation: Parallelism

To conclude, let's go back to the Rust implementation.

Didn't we say that Rust enabled easy fearless concurrency?



A sad Ryzen 9 7900 being underutilized while tokenizing.

Writing a tokenizer package in Rust

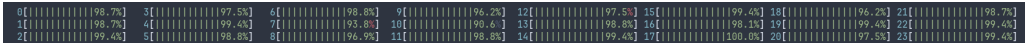
Rust implementation: Parallelism

Word-level tokenization is embarassingly parallel (esp. given we split on whitespace), so let's use a threadpool with the great rayon crate to fully utilize our hardware by adding 4 characters to our code:

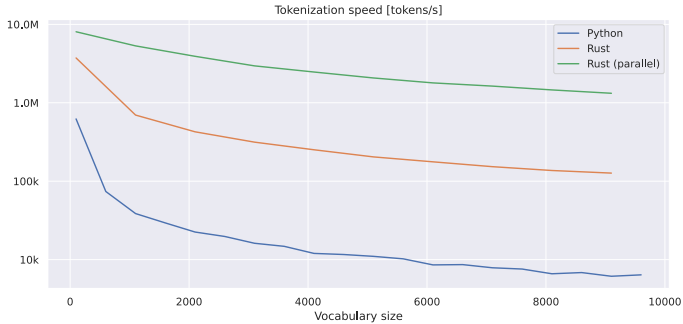
```
fn tokenize_text(&self, text: &str) -> Vec<TokenID> {
    text.par_lines() // note the `par_`
        .map(|line| self.tokenize(line))
        .flatten()
        .collect()
}
```

Writing a tokenizer package in Rust

Rust implementation: Parallelism



A happy Ryzen 9 7900 being properly utilized to reach 1M tokens/s on its 24 threads, with a single process.



Contents

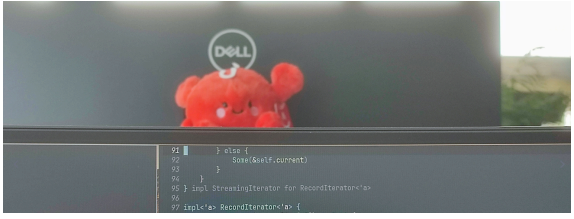
Conclusion

- Writing performance-critical code in a compiled language can bring significant performance improvements with no algorithmic effort, by taking advantage of compiler optimizations and parallelism (single- or multi-threaded).
  e.g. 6k tokens/s → 100k tokens/s → 1M tokens/s for our toy tokenizer
- Rust makes this *easy* and *safe*, including the integration with Python, reducing the barrier compared to writing extensions in C or C++.
- This is a great opportunity in machine learning and data science engineering, where CPU-intensive work is often present and can be a bottleneck.
  Not covered in this talk: `async` Rust
- We also saw multiple examples of tools written fully in Rust, which can be a very valid option beyond prototyping.

Any bottlenecks in your Python code where you think it would be worth trying Rust?

Conclusion

Thank you / Q&A



Personal page: https://c.pgdm.ch – slides and code uploaded by next Monday.

A more detailed Rust introductory talk: https://c.pgdm.ch/notes/brief-tour-rust-talk/
A talk on async Rust: https://c.pgdm.ch/notes/practical-async-rust-talk/

Connect: c@pgdm.ch or https://www.linkedin.com/in/corentinperretgentil/