Bandung 18 October 2023

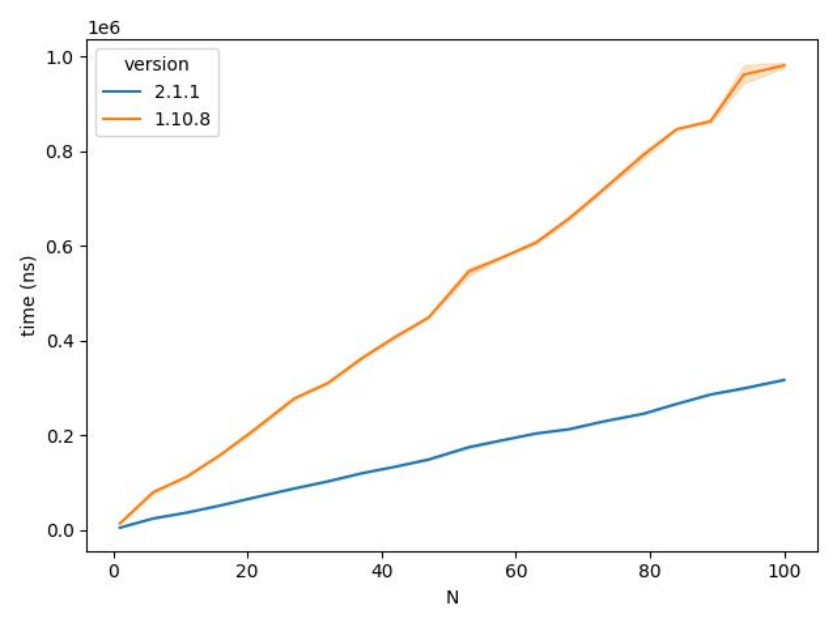
Boosting Python Performance with Rust

Build Python extensions using Rust

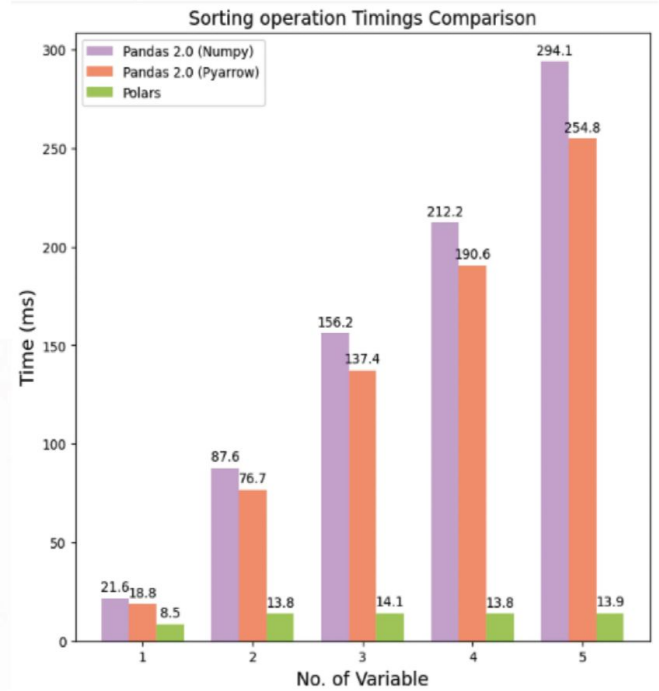
Tirtadwipa Manunggal

Technical Product Manager at Dcentric Health

🔥 Intro



Benchmarking **Pydantic** v1.10.8 vs v2.1.1 on instantiating huge data class[1]



Benchmarking **Pandas** vs Polars on sorting huge dataset [2]

[1] https://janhendrikewers.uk/pydantic-1-vs-2-a-benchmark-test.html
[2] https://medium.com/cuenex/pandas-2-0-vs-polars-the-ultimate-battle-a378eb75d6d1

👀 What's behind this?

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

raphlinus • 9 yr. ago

There are a lot of answers to this question, and of course a lot has to do with the skill of the programmer and their motivation to optimize for speed. Here's a very partial list of what the language provides:

1. It uses LLVM to generate assembly code, which is a state of the art optimizer, comparable to best-in-class C and C++ compilers.
2. It makes it easy to lay out data structures in a cache-efficient manner. See Why (most) High Level Languages are Slow for more detail on what this means and why it's important.
3. Its type system lets you choose a pointer/reference strategy that does the least amount of work at runtime while still being safe. The default in Swift, for example, is atomic reference counting. If you're not sharing across threads, Rust lets you replace `Arc<T>` with a less expensive `Rc<T>`. Or, in many cases, use a borrowed pointer with no runtime cost at all.
4. Garbage collection has some cost, even though modern garbage collectors are pretty good. This is even more true if RAM is tight and you can't budget a lot of it
5. The iterators in the static library (mostly) compile to very good code, eliminating even the bounds checks that would ordinarily be needed.
6. Its approach to generics (monomorphization) does all the work at compile time, with no runtime cost to figuring out the types. (This is a tradeoff, with potential negative consequences for compile time and code size).
7. Its runtime is very compatible with C, with no stack switching or copying needed for FFI when calling C libraries or system functions.

As I said, there are many more, these are just the main ones I've experienced.

[3] https://dl.acm.org/doi/10.1145/3136014.3136031
[4] https://www.reddit.com/r/rust/comments/39eyn9/why_rust_so_fast/

🤔 Wait wait, but C/C++ binding has been there since long time ago...

- **Memory safety:** C/C++ completely gives the developer responsibility to manage the memory which opens up the possibility to unsafe memory handling, while Rust introduces a strong ownership system and a borrow checker that enforces memory safety at compile-time.

```c
// Allocating memory for a string
char *c_string = (char *)malloc(10 * sizeof(char));

// Writing more characters than allocated
strcpy(c_string, "Hello, World!");

// Print the string
printf("%s\n", c_string);

// Freeing the memory
free(c_string);

// Accessing freed memory (undefined behavior)
printf("%s\n", c_string);
```

```rust
// Creating a String in Rust
let mut rust_string = String::from("Hello, World!");

// Using the string
println!("{}", rust_string);

// Uncommenting the line below would result in a compile error.
// println!("{}", rust_string);
```

- **Package Manager and Tooling:** C and C++ traditionally do not have a standardized package manager like some other modern programming languages. While in Rust, we have Cargo that simplifies building and dependency management.

- **BONUS.** For the seventh year in a row, Rust is the most loved language with 87% of developers saying they want to continue using it based on Stackoverflow Developer Survey.

[5] https://thenewstack.io/stack-overflow-rust-remains-most-favored-but-clojure-pays-the-most/

💻 Stop talking. Show me the code!

🍕 Prerequisites

- **Python 3.7+**

- **Rust compiler and tooling**

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

- **Poetry (`pip` also fine):** Package manager

- **Maturin:** Rust-based Python packages building tool

```
poetry add maturin
```

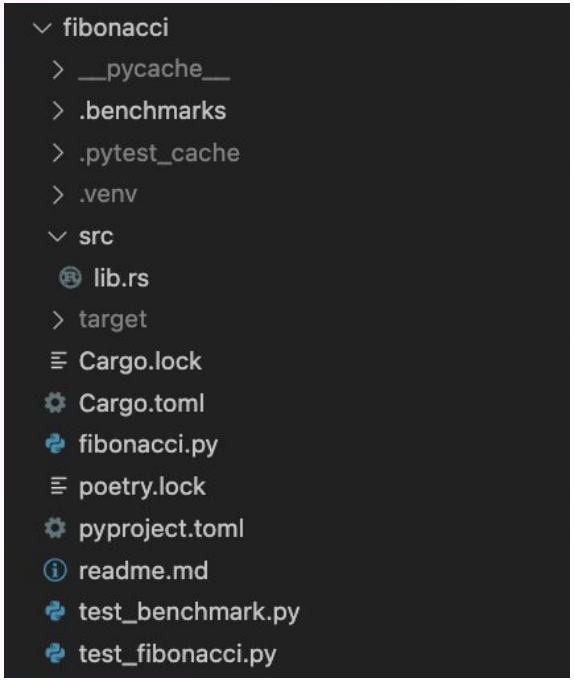
- **PyO3:** Rust bindings and tooling for Python

```
cargo add pyo3
```

🍰 Initialize Rust-Python project

```
maturin init
```

🥞 Project structure



🛠️ Think of a module and function...

The `#[pyfunction]` attribute is used to define a Python function from a Rust function. Once defined, the function needs to be added to a module using the `wrap_pyfunction!` macro.

The following example defines a function called double in a Python module called `my_extension`:

```rust
use pyo3::prelude::*;

#[pyfunction]
fn double(x: usize) -> usize {
    x * 2
}

#[pymodule]
fn my_extension(py: Python<'_>, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(double, m)?)?;
    Ok(())
}
```

Basic structure of **Rust**-based Python extension

```python
from my_extension import double

double(10)
```

How the extension is used in **Python**

Exercise 1: Types

When writing a Python extension in Rust, we must always pay attentions to the data type. While Python is a dynamically typed language, we must understand the how data type is converted from Rust to Python or vice-versa. The following types are basic type conversions between Rust and Python.

Rust type	Resulting Python Type
`String`	`str`
`&str`	`str`
`bool`	`bool`
Any integer type (`i32` , `u32` , `usize` , etc)	`int`
`f32` , `f64`	`float`
`Option<T>`	`Optional[T]`
`(T, U)`	`Tuple[T, U]`
`Vec<T>`	`List[T]`
`Cow<[u8]>`	`bytes`
`HashMap<K, V>`	`Dict[K, V]`
`BTreeMap<K, V>`	`Dict[K, V]`
`HashSet<T>`	`Set[T]`
`BTreeSet<T>`	`Set[T]`
`&PyCell<T: PyClass>`	`T`
`PyRef<T: PyClass>`	`T`
`PyRefMut<T: PyClass>`	`T`

```rust
#[pyfunction]
pub fn integer_addition(left: i32, right: i32) -> i32 {
    left + right
}

#[pyfunction]
pub fn float_subtraction(left: f32, right: f32) -> f32 {
    left - right
}

#[pyfunction]
pub fn mixed_multiplication(left: i32, right: f32) -> f32 {
    left as f32 * right
}
```

```python
from type_conversion import mixed_multiplication

left: int = 1
right: float = 2
result = mixed_multiplication(left, right)
```

🏋️ Exercise 2: `for` loops and `if-else` condition

In Rust, the basic for `loop` is often used to iterate over collections like arrays, vectors, or ranges. In the first example, we use a range `(0..n)` to iterate from 0 to n (exclusive).

Rust's `if-else` conditions are similar to those in many other languages, but they come with Rust's ownership and borrowing concepts.

```rust
#[pyfunction]
pub fn integer_sequence(n: usize) -> Vec<usize> {
    let mut res = Vec::new();

    for i in 0..n {
        res.push(i);
    }

    res
}
```

```rust
#[pyfunction]
pub fn fizz_buzz(n: u32) -> String {
    let res: String;
    let is_divisible_by_three = (n % 3) == 0;
    let is_divisible_by_five = (n % 5) == 0;

    if is_divisible_by_three && is_divisible_by_five {
        res = "FizzBuzz".to_string();
    } else if is_divisible_by_three {
        res = "Fizz".to_string();
    } else if is_divisible_by_five {
        res = "Buzz".to_string();
    } else {
        res = n.to_string();
    }

    res
}
```

```python
from loops_and_conditions import integer_sequence, fizz_buzz

integer_sequence(10)
fizz_buzz(15)
```

🏋️‍♀️ Exercise 3: Exceptions

In Rust, errors are typically represented using the Result type, which is an enum with variants Ok for a successful result and Err for an error. Handling errors is an integral part of writing robust and safe Rust code. In PyO3, we can conveniently use `PyResult` to work with error.

```rust
use pyo3::exceptions::PyValueError;
use pyo3::prelude::*;

#[pyfunction]
pub fn division(left: i32, right: i32) -> PyResult<i32> {
    match left.checked_div(right) {
        Some(result) => Ok(result),
        None => Err(PyValueError::new_err(
            "Division by zero or overflow occurred",
        )),
    }
}
```

```python
import pytest

from error_in_rust import division

def test_positive_case():
    assert division(10, 1) == 10

def test_negative_case():
    with pytest.raises(Exception):
        division(10, 0)
```

🏋️‍♀️ Exercise 4: using Rust library

Rust's extensive libraries and performance benefits make it an ideal choice for certain tasks. Instead of reinventing the wheel when writing Python extension, we can use Rust libraries.

```rust
use jsonschema::JSONSchema;
use pyo3::prelude::*;
use serde_json::{Result, Value};

#[pyfunction]
pub fn validate_json(schema_str: &str, data_str: &str) -> bool {
    let maybe_schema = serde_json::from_str(schema_str);
    let maybe_data = serde_json::from_str(data_str);

    if maybe_schema.is_err() || maybe_data.is_err() {
        return false;
    }

    let schema: Value = maybe_schema.unwrap();
    let data: Value = maybe_data.unwrap();

    let compiled = JSONSchema::compile(&schema).unwrap();
    let res = compiled.is_valid(&data);

    res
}
```

```python
from use_rust_library import validate_json
from json import dumps

def test_use_rust_library():
    schema = dumps(
        {
            "type": "object",
            "properties": {
                "price": {"type": "number"},
                "name": {"type": "string"},
            },
        }
    )
    value = dumps({"price": 34.99, "name": "Eggs"})

    assert validate_json(schema, value) == True
```

🎖️ Hands-on DIY Benchmarking

We will demonstrate benchmarking algorithms written in booth Rust and Python. Let's take two simple complex algorithms: **Fibonacci** and **Prime Factorization**.

```rust
#[pyfunction]
pub fn fibonacci(n: u32) -> u32 {
    if n == 0 {
        0
    } else if n == 1 {
        1
    } else {
        fibonacci(n - 1) + fibonacci(n - 2)
    }
}
```

```python
def fibonacci(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Hands-on DIY Benchmarking (cntd.)

```rust
use pyo3::prelude::*;

#[pyfunction]
pub fn factorize(n: u32) -> Vec<u32> {
    let mut factors = Vec::<u32>::new();
    let mut i = 2;
    let mut num = n;

    while i * i <= n {
        if num % i != 0 {
            i += 1;
        } else {
            num /= i;
            factors.push(i);
        }
    }
    if num > 1 {
        factors.push(num);
    }

    factors
}
```

```python
def factorize(n: int) -> list[int]:
    factors: list[int] = []
    i: int = 2

    while i * i <= n:
        if n % i:
            i += 1
        else:
            n //= i
            factors.append(i)

    if n > 1:
        factors.append(n)

    return factors
```

🎖️ Hands-on DIY Benchmarking (cntd.)

```python
import fibonacci
import rust_fibonacci


def test_python_prime_factorization(benchmark):
    result = benchmark(fibonacci.fibonacci, 32)

    assert result == 2178309


def test_rust_prime_factorization(benchmark):
    result = benchmark(rust_fibonacci.fibonacci, 32)

    assert result == 2178309
```

```python
import prime_factorization
import rust_prime_factorization


def test_python_prime_factorization(benchmark):
    result = benchmark(prime_factorization.factorize, 2147483647)

    assert result == [2147483647]


def test_rust_prime_factorization(benchmark):
    result = benchmark(rust_prime_factorization.factorize, 2147483647)

    assert result == [2147483647]
```

Do and Don’ts

Do...

- **Performance-Critical Operations:** use Rust for performance-critical operations, such as numeric calculations, data processing, or algorithms where Rust's performance benefits are crucial.
- **Reuse Existing Rust Libraries:** take advantage of existing Rust libraries for specific tasks. If there's a well-established Rust library that meets your needs, using it in your Python extension can save development time and benefit from the Rust ecosystem.
- **Wrap Complex C Libraries:** consider using Rust to wrap and simplify the usage of complex C libraries. Rust's safety features can help provide a more user-friendly interface for Python developers.
- **Write High-Performance Modules:** use Rust to write high-performance modules that can be seamlessly integrated into larger Python applications. This is especially beneficial when Python's performance is a bottleneck.

Don’ts...

- **Mix Rust and Python for Business Logic:** Avoid using Rust for the core business logic of your Python API unless there's a compelling reason. Mixing languages can introduce complexity and make the codebase harder to manage.
- **Assume All Python Developers Know Rust:** Avoid assuming that all Python developers working with your code are familiar with Rust. Provide clear documentation and examples to facilitate easy integration for Python developers.
- **Use Rust for Rapid Prototyping:** Rust may not be the best choice for rapid prototyping or when quick iterations are required. The compile times and strict ownership checks may slow down the development process in certain scenarios.
- **Don’t ride the trend waves:** Carefully evaluate whether Rust's performance benefits and safety features align with the specific needs and goals of your project, ensuring a practical and justified decision.

Finally, always remember it’s not Python versus Rust

- The performance comparison between Python and Rust should be viewed in the context of their intended **use cases and the specific problems** they are designed to solve.
- Both languages have **unique strengths**, and choosing between them often involves considering the broader tooling and ecosystem.
- Python is a high-level language known for its **readability, ease of use, and extensive ecosystem of libraries**. It excels in rapid development, scripting, and is widely used for tasks like data analysis, web development, and automation.
- Rust, on the other hand, is a systems programming language designed for **performance, safety, and concurrency.**
- Moreover, combining languages within a project is a common practice known as polyglot programming. This approach allows developers to utilize the best tool for each specific job, **maximizing the benefits** of different languages in a single project.

https://github.com/liberocks/pyconid23

🦀 🐍 That's all. Thank you.

Questions?

Github: @liberocks

Twitter: @liberocks

LinkedIn: https://www.linkedin.com/in/tirtadwipa-manunggal