Extending Python with Rust: a hands-on introduction to PyO3

Ivan Carvalho

Agenda

- Introduce the tools
- Build an extension using PyO3
- Understand what problems Python extensions in Rust solve

About me

- Maintainer of `rustworkx` (peaked at the Top 1% of PyPI packages)
- User of PyO3 since 2021
- Casually contributed features I needed to PyO3 upstream

Why this presentation

Rust has taken over the Python ecosystem!

- Popular libraries like `pydantic` and `cryptography` use Rust
- Python tooling also uses Rust like:
  - Astral's uv
  - Microsoft's Python Environment Tools
  - Facebook's Pyrefly

But why? Hopefully you'll understand by the end.

Tools we are going to be using

We don't assume you'll be familiar with all of them. Rust users will know Rust tools, Python users will known Python tools.

- PyO3
- Maturin
- pip
- PyPI
- Cargo
- Crates.io

Motivating Problem

To give us a concrete goal, this presentation will build an extension that can decode JPEG XL images.

As of 2025, iPhones can now take photos and save in the JPEG XL file format. It's feasible you'd find this file format in the wild!

Maturin

Maturin is a Python build tool provided by the PyO3 developers. It helps building Rust code as extensions.

Maturin can be installed with `pip install maturin`.

The initial draft of the repository was the output of `maturin new`.

`maturin develop` installs the extension locally for development.

`maturin build` is used to package a wheel.

Manifest files

These are the two files from `maturin new`, edited by me.

Cargo.toml

```toml
[package]
name = "jxl_demo"
version = "0.1.0"
edition = "2021"

[lib]
name = "jxl_demo"
crate-type = ["cdylib"]

[dependencies]
ndarray = "0.16"
pyo3 = {
    version = "0.26.0",
    features = [
        "abi3",
        "extension-module"
    ]
}
numpy = "0.26"
jxl-oxide = "0.11.4"
```

pyproject.toml

```toml
[build-system]
requires = ["maturin>=1.9,<2.0"]
build-backend = "maturin"

[project]
name = "jxl_demo"
requires-python = ">=3.10"
classifiers = [
    # ommited
]
dynamic = ["version"]
dependencies = [
    "pillow>=10.0",
    "numpy>=2.0",
]

[tool.maturin]
features = ["pyo3/extension-module"]
```

Dependencies

This demo is only possible thanks to `jxl-oxide` and `pyo3` being easily available on crates.io.

Rust arguably has more friendly dependency management than Python. We'll not discuss the Python packaging ecosystem.

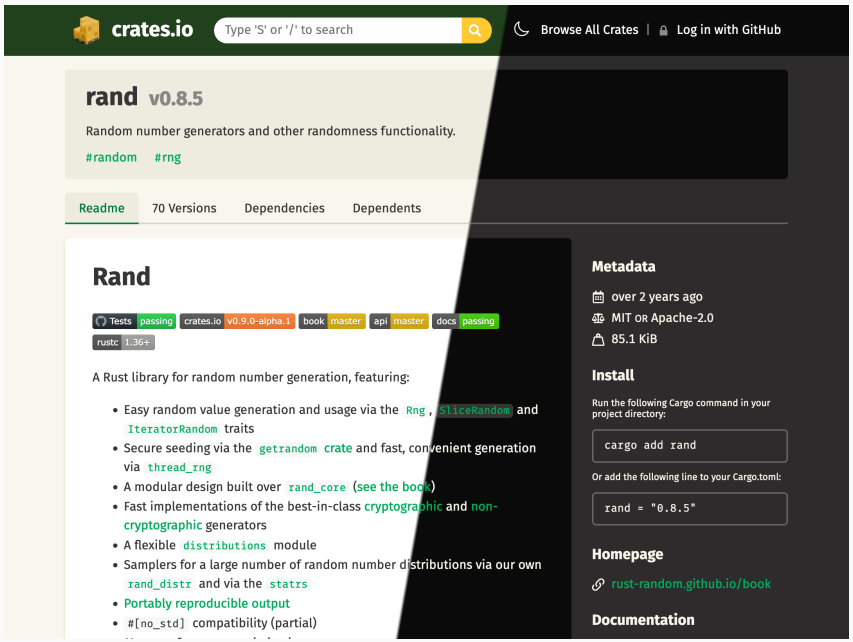

Figure 1: crates.io

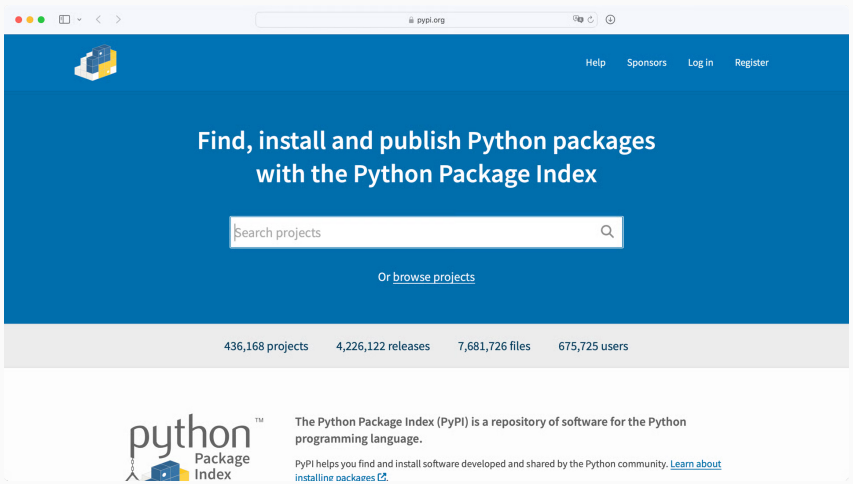

Figure 2: PyPI

PyO3

The glue between Python and Rust.

PyO3 exposes:

- Python types to Rust e.g. `PyAny`
- Conversion between Rust and Python types
- Rust functions to Python via `#[pyfunction]`
- Rust structs to Python via `#[pyclass]`

Python	Rust	Rust (Python-native)
`object`	-	`&PyAny`
`str`	`String`, `Cow<str>`, `&str`	`&PyUnicode`
`bytes`	`Vec<u8>`, `&[u8]`	`&PyBytes`
`bool`	`bool`	`&PyBool`
`int`	Any integer type (`i32`, `u32`, `usize`, etc)	`&PyLong`
`float`	`f32`, `f64`	`&PyFloat`
`complex`	`num_complex::Complex`[1]	`&PyComplex`
`list[T]`	`Vec<T>`	`&PyList`
`dict[K, V]`	`HashMap<K, V>`, `BTreeMap<K, V>`	`&PyDict`
`tuple[T, U]`	`(T, U)`, `Vec<T>`	`&PyTuple`

Figure 3: Incomplete conversion list for PyO3

PyO3 In Practice

Most of the work is to bridge the data between Python and Rust.

Once that is settled, it becomes Rust code.

```rust
#[pyfunction]
fn decode_jxl_as_array<'py>(
    py: Python<'py>,
    jxl_bytes: &Bound<'py, PyBytes>,
) -> PyResult<Bound<'py, PyArrayDyn<u8>>> {
    // Convert PyBytes to something Rust understands.
    let bytes = jxl_bytes.as_bytes();
    let cursor = std::io::Cursor::new(bytes);

    // Implementation details are hidden on purpose.
    let array = decode_jxl_core(cursor).map_err(|e| PyValueError::new_err(e))?;

    // Convert to a NumPy array and return.
    Ok(PyArrayDyn::from_array(py, &array))
}
```

Modules

We need to export our functions and classes in modules.

The good news: it is still much faster than regular Python imports.

```rust
#[pymodule]
fn jxl_demo(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(decode_jxl_as_array, m)?)?;
    m.add_function(wrap_pyfunction!(decode_jxl, m)?)?;
    Ok(())
}
```

Elaborate workflows with Python

One thing we haven't explored yet is that PyO3 has access to the Python interpreter. We can interact with the Python objects.

To give a concrete example, let's write some that interacts with `PIL`. Our demo will bridge the Rust and Python ecosystems.

Elaborate workflows in practice

```rust
#[pyfunction]
fn decode_jxl<'py>(
    py: Python<'py>,
    jxl_bytes: &Bound<'py, PyBytes>,
) -> PyResult<Bound<'py, PyAny>> {
    // Import PIL.Image module
    let pil_image = py.import("PIL.Image")?;
    let fromarray_fn = pil_image.getattr("fromarray")?;

    // Get the NumPy array from our existing function
    let np_array = decode_jxl_as_array(py, jxl_bytes)?;

    let shape = np_array.shape();
    // Create Pillow Image from NumPy array
    let pil_img = if shape[2] == 3 || shape[2] == 4 {
        // a.k.a. RGB case
        fromarray_fn.call1((np_array,))?
    } else { /* omitted */};

    Ok(pil_img)
}
```

Handling Errors

Python code raises exceptions when it finds errors.

Rust code returns a `Result<T, E>` enum.

PyO3 provides a `PyResult<T>` type:

- If your method returns the `Ok(_)` case, it means no error happened
- If it returns the `Err(_)` case, PyO3 will raise an exception on the Python side

Three main categories for Rust extensions

Now that we introduced Rust extensions, we can categorize three main groups of extensions:

- Simple Rust bindings
- Shared implementation between languages
- Blazingly Fast Python Extensions

Simple Rust Bindings

We want to reuse existing Rust code.

Examples include:

- This demo
- rpds-py: bindings to the `rpds` crate from Rust

Shared Implementation Between Languages

Write a Rust core and re-use it everywhere:

- Python bindings with PyO3
- Compile to WebAssembly (WASM) and use in JavaScript
- Possible to extend to more languages, see UniFFI

tiktoken is an example library that does that.

Blazingly Fast Python Extensions

Purposefully write libraries in Rust to leverage the qualities of Rust:

- Compiled extensions are faster than interpreted Python code

Blazingly Fast Python Extensions (continued)

Purposefully write libraries in Rust to leverage the qualities of Rust:

- Compiled extensions are faster than interpreted Python code
- Easy-to-use parallelism with low-overhead
  - Rayon makes multi-threading easy
  - std::thread ships with the language

Blazingly Fast Python Extensions (continued)

Purposefully write libraries in Rust to leverage the qualities of Rust:

- Compiled extensions are faster than interpreted Python code
- Easy-to-use parallelism with low-overhead
  - Rayon makes multi-threading easy
  - std::thread ships with the language
- Feasible to use SIMD
  - core::arch makes platform-specific instructions available
  - std::simd is the unstable API that will democratize SIMD even more

Blazingly Fast Python Extensions (continued)

Purposefully write libraries in Rust to leverage the qualities of Rust:

- Compiled extensions are faster than interpreted Python code
- Easy-to-use parallelism with low-overhead
  - Rayon makes multi-threading easy
  - std::thread ships with the language
- Feasible to use SIMD
  - core::arch makes platform-specific instructions available
  - std::simd is the unstable API that will democratize SIMD even more
- Re-use optimized solutions via dependencies
  - Easy to leverage the work of others
  - Example: ryu, the crate for writing floats to strings

Let users install your library

Rust tooling makes extensions easy to distribute:

- Build for Linux, macOS, Windows
- Target x86-64, ARM, PowerPC
- Cross-compilation is feasible

No one cares if your extension is fast it they can't install it!