

# **Integración de Rust y Python con PyO3**

Hermilo

4 de Septiembre 2025

# Introducción

- Nos gustaría mejorar el desempeño y diseño del software disponible.
- Tendencias tecnológicas y lenguajes de programación novedosos hace tentador a las empresas **reescribir** sus productos.
- Si algunas empresas han mostrado los beneficios en desempeño, seguridad y mantenibilidad de **reescribir** su software en un lenguaje nuevo, ¿Por qué no hacer lo mismo?
- **Reescribir software desde cero es una tarea difícil y riesgosa.**
- **El software existente puede tener años de experiencia en producción y monitoreo, así como una base de conocimiento para identificar bugs.**

## ¿Qué es Refactorizar?

- **Refactorizar** es “reescritura en una menor escala” (Mara & Holmes, 2025).
- Es una técnica más controlada para mejorar el diseño y performance del software existente.
- En lugar de reemplazar al sistema actual por completo, nos gustaría detectar **las partes más críticas de nuestro código** que necesitan ser refactorizadas.

¿Qué es Refactorizar?



## Python's bottleneck is inevitably performance (Hewitt, 2024)

- Muchas bibliotecas de Python tiene performance porque están parcial o totalmente implementadas en lenguajes de bajo nivel como C, C++, Fortran, Cython o bien con implementaciones alternativas de Python como PyPy, IronPython, Jython, etc (Rodrigues, 2023).
- Si existen alternativas para mejorar el desempeño, ¿Por qué una alternativa adicional? ¿Por qué Rust?

# ¿Por qué Rust?

- Rust es un lenguaje de programación que enfatiza tiempo de ejecución rápido, alta confiabilidad, memory safety y fearless concurrency.
  - **Confiabilidad** : Control de errores. Patrón que combina `Result<T, E>` Type con Pattern Matching donde el desarrollador se encarga de manejar los errores como parte del desarrollo del programa.
  - **Memory Safety** : Sin la existencia de un recolector de basura. Concepto de **borrow checker** que verifica que los accesos a datos son legales, lo que le permite a Rust prevenir problemas de seguridad sin imponer costos durante el tiempo de ejecución (McNamara, 2021). Tres conceptos importantes :
    - **lifetimes**
    - **ownership**
    - **borrowing**
  - **Concurrency** : Al escribir código multithreaded, la desarrolladora tendrá la confianza que no producirá **data races**.

¿Por qué Rust?

Rust offers power and precision to go beyond Python's limits (Hewitt, 2024)

A promotional poster for the Rust Nation UK event. The background is purple with faint, stylized gear and star patterns. At the top, the event name 'Rust Nation UK' is displayed in a stylized font with 'Rust' in a red box, 'Nation' in a yellow box, and 'UK' in a green box. Below this, the name 'DAVID HEWITT' is written in large, bold, white capital letters. To the right of the name is a circular portrait of David Hewitt, a man with glasses and a beard, smiling. Below the name and portrait, the text 'Rust & Python Developer - Pydantic' is written in a smaller, yellow font. At the bottom, the event dates '19th & 20th Feb', the website 'www.rustnationuk.com', and the location 'London, UK' are listed in white text.

**Rust Nation UK**

**DAVID HEWITT**

Rust & Python Developer - Pydantic

19<sup>th</sup> & 20<sup>th</sup> Feb    [www.rustnationuk.com](http://www.rustnationuk.com)    London, UK

# ¿Qué es PyO3?

PyO3 es un proyecto desarrollado en Rust que permite la integración con Python.

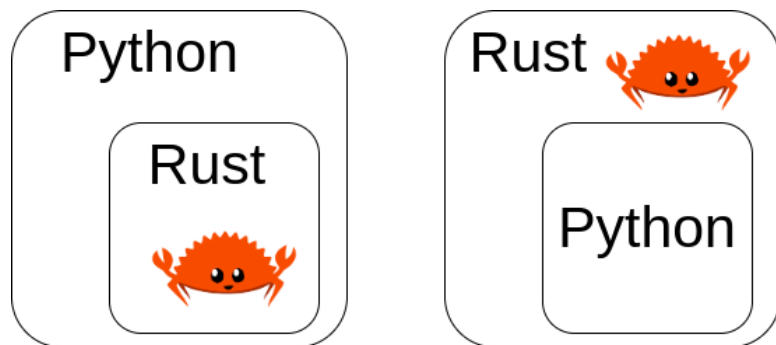


Figure 3: Tomado de Hewitt 2025

## Proyectos de soporte

- **maturin** : CLI build backend.
  - **setuptools-rust** : adiciona Rust a proyectos de setuptools.
  - **rust-numpy** : numpy interoperability.
- Guía para el desarrollador : <https://pyo3.rs>
  - Documentación : <https://docs.rs/pyo3>
  - Github : <https://github.com/pyo3/pyo3>



# La filosofía de PyO3 en el ecosistema de Python

PyO3 agrega el poder y precisión de Rust al ecosistema de Python. No es una sustitución. Es complementario.

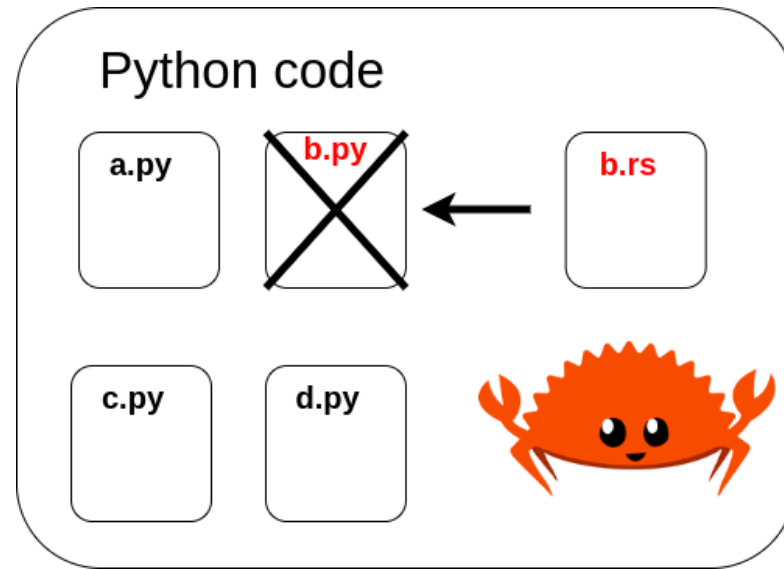


Figure 4: Tomado de Hewitt 2025

# Cómo funciona PyO3

PyO3 usa macros procedurales (“proc macro”) las cuales son una forma de metaprogramación<sup>1</sup> que permite manipular, generar código adicional, o agregar nuevas capacidades a nuestro programa<sup>2</sup>.

La macro expande el código de Rust para llamar la API de C de Python para definir funciones, clases y módulos.

```
1 #[pyfunction]
2 fn my_rust_function(){...}
```



---

<sup>1</sup>**Metaprogramación** es código que usará otro código como insumo

<sup>2</sup>En Rust existen dos tipos de macros : **declarativas** y **procedurales**

# Cómo funciona PyO3


PyO3 usa macros procedurales (“proc macro”) las cuales son una forma de metaprogramación<sup>3</sup> que permite manipular, generar código adicional, o agregar nuevas capacidades a nuestro programa<sup>4</sup>.

La macro expande el código de Rust para llamar la API de C de Python para definir funciones, clases y módulos.

```
1 #[pyfunction]
2 fn my_rust_function(){...}
```

 Rust

```
1 unsafe extern "C" fn __wrap(){...}
2 PyMethodDef{
3     ml_meth: __wrap as *mut c_void,
4     ...
5 }
```

 Rust

---

<sup>3</sup>Metaprogramación es código que usará otro código como insumo

<sup>4</sup>En Rust existen dos tipos de macros : **declarativas** y **procedurales**

# Cómo funciona PyO3

PyO3 usa macros procedurales (“proc macro”) las cuales son una forma de metaprogramación que permite manipular, generar código adicional, o agregar nuevas capacidades a nuestro programa.

La macro expande el código de Rust para llamar la API de C de Python para definir funciones, clases y módulos.

```
1 #[pyfunction]
2 fn my_rust_function(){...}
```



Herramientas como maturin y setuptools-rust se encargan de la tarea de compilar el código de Rust a un módulo para que Python pueda consumirlo.

```
1 unsafe extern "C" fn __wrap(){...}
2 PyMethodDef{
3     ml_meth: __wrap as *mut c_void,
4     ...
5 }
```



b.so



b.pyd



## ¿Cómo consume Python las extensiones?

- Comunmente la declaración de Python se usa para cargar un archivo (módulo) de Python.

```
1 import b
```

 Python




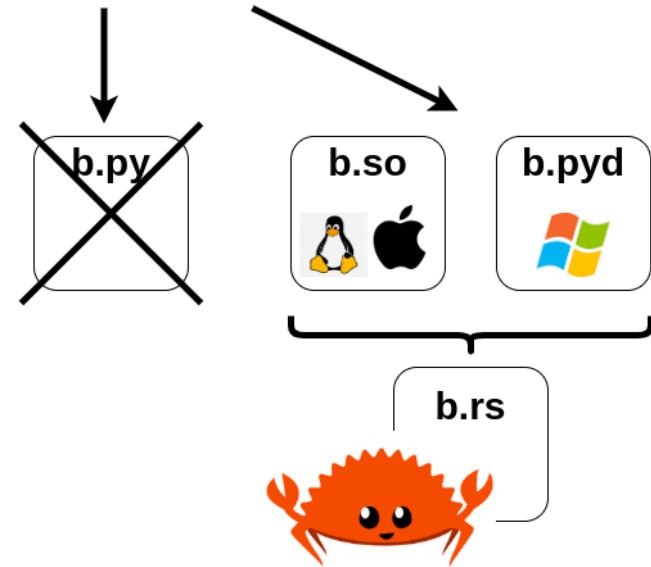
**b.py**

# ¿Cómo consume Python las extensiones?

- Comunmente la declaración de Python se usa para cargar un archivo (módulo) de Python.
- También puede carga un **extension module** de una biblioteca compatible con el ABI<sup>5</sup> de Python.
- Esto es apliamente usado, e.g. Cython, C, C++, y Rust.

```
1 import b
```

 Python



---

<sup>5</sup>Application Binary Interface

# Cython Compiler

- El compilador Cython convierte el código fuente de Cython a código de C optimizado.
- Posteriormente se compilará durante el proceso de construcción del paquete.

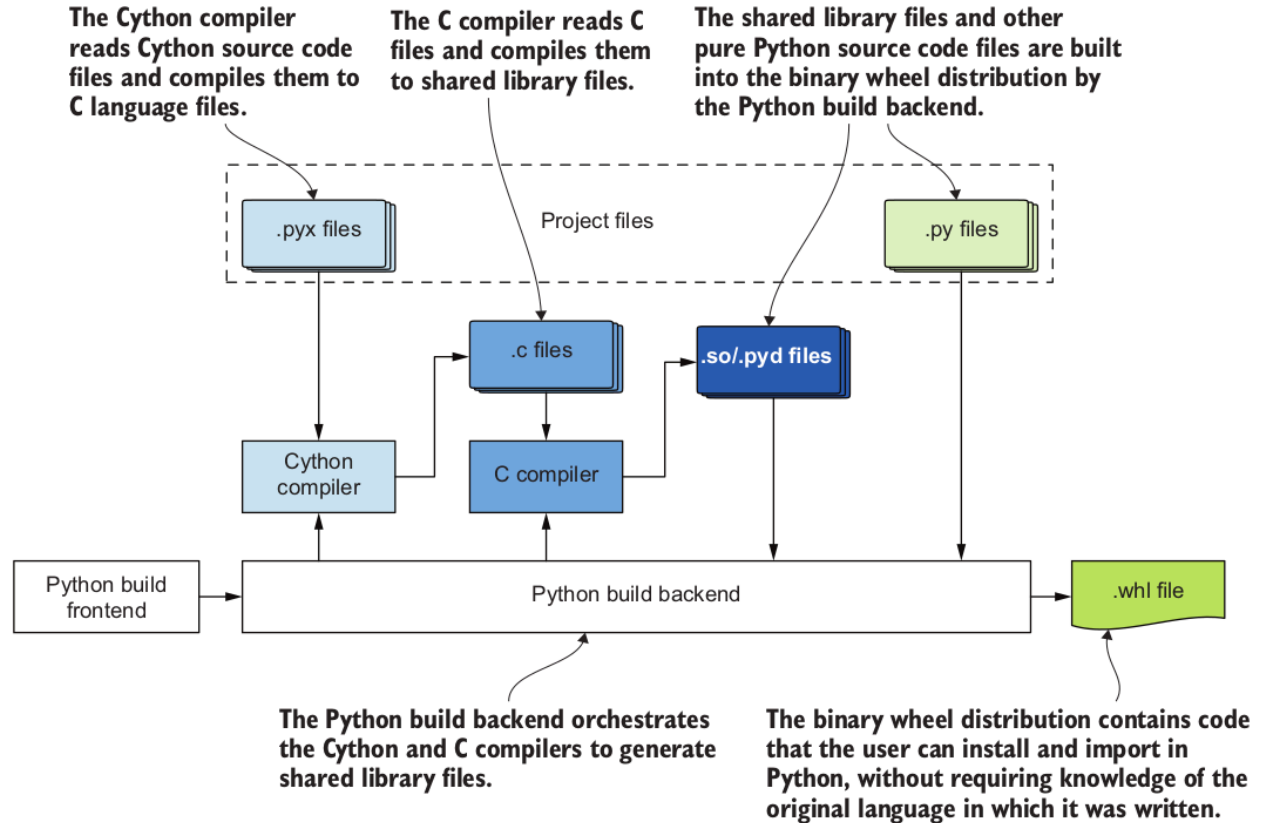


Figure 4.1 Extensions are compiled into shared libraries that are included in binary wheel distributions.

Figure 8: Tomado de (Hilliard, 2022)

# Portabilidad

- Cuando escribimos paquetes usando solo Python, este es **extremadamente portable**-cualquier sistema que tenga una versión compatible de Python puede ejecutar el paquete.
- Cuando incluimos código que debe ser compilado, **el código fuente típicamente debe ser compilado separadamente para cada arquitectura y SO donde será usado.**

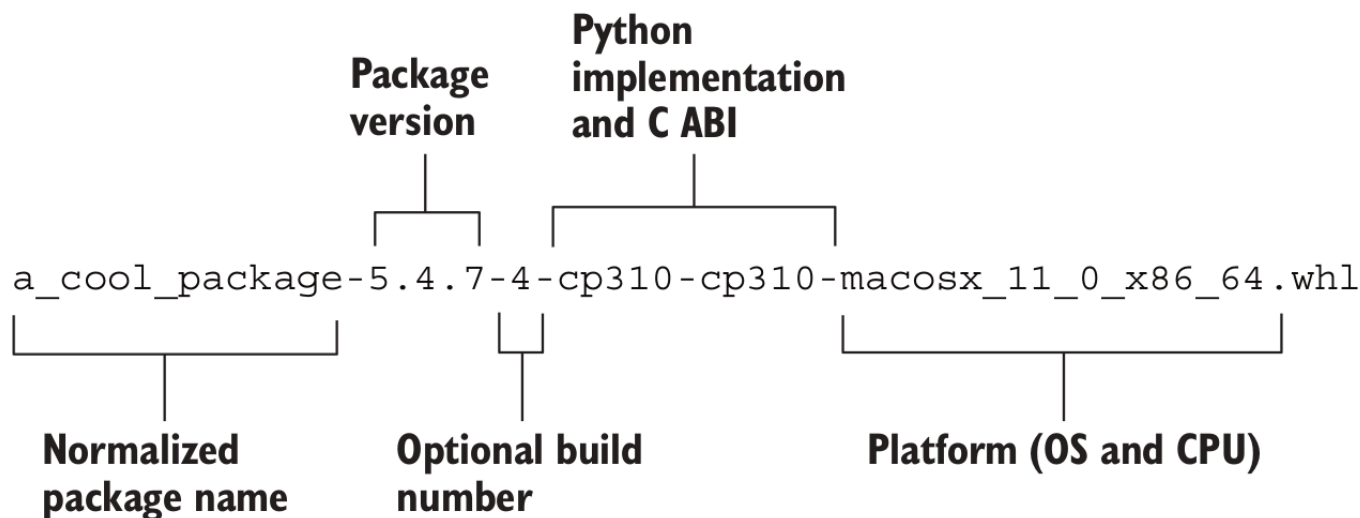


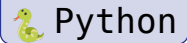
Figure 9: Anatomía de un archivo de distribución wheel. Tomado de (Hilliard, 2022)



# Ejemplo

Programa que cuenta ocurrencias en un texto:


```
1  def count_ocurrences(  
2      contents: str,  
3      needle: str,  
4      ) -> int:  
5  
6      total = 0  
7  
8      for line in contents.splitlines():  
9          for word in line.split():  
10             if word == needle:  
11                 total+=1  
12  
13      return total
```




# Ejemplo

La traducción de esta función con PyO3 es mecánica:

```
1  def count_ocurrences(  
2      contents: str,  
3      needle: str,  
4      ) -> int:  
5  
6      total = 0  
7  
8      for line in contents.splitlines():  
9          for word in line.split():  
10             if word == needle:  
11                 total+=1  
12  
13      return total
```

 Python


```
1  #[pyfunction]  
2  fn count_ocurrences(  
3      contents: &str,  
4      needle: &str,  
5  ) -> usize {  
6      let mut total = 0;  
7      for line in contents.lines(){  
8          for word in line.split(" "){  
9              if word == needle{  
10                 total += 1;  
11             }  
12         }  
13     }  
14     total  
15 }
```

 Rust


# Ejemplo

La traducción de esta función con PyO3 es mecánica:

```
1 def count_ocurrences(  
2     contents: str,  
3     needle: str,  
4 ) -> int:  
5  
6     total = 0  
7  
8     for line in contents.splitlines():  
9         for word in line.split():  
10             if word == needle:  
11                 total+=1  
12  
13     return total
```

 Python

```
1 #[pyfunction]  
2 fn count_ocurrences(  
3     contents: &str,  
4     needle: &str,  
5 ) -> usize {  
6     let mut total = 0;  
7     for line in contents.lines(){  
8         for word in line.split(" "){  
9             if word == needle{  
10                 total += 1;  
11             }  
12         }  
13     }  
14     total  
15 }
```

 Rust

**~ 2-4X faster (Python 3.12)**

```
1  /// A Python module implemented in Rust
2  #[pyo3::pymodule]
3  mod hello_py3{
4      use pyo3::prelude::*;
5      /// Counts the number of occurrences of `needle` in
6      `contents`.
7      #[pyfunction]
8      fn count_ocurrences(contents: &str, needle: &str) -> usize{
9          let mut total = 0;
10         for line in contents.lines(){
11             for word in line.split(" "){
12                 if word == needle{
13                     total += 1;
14                 }
15             }
16         }
17         total
18     }
```



## Rust Source

```

1  /// A Python module implemented in Rust
2  #[pyo3::pymodule]
3  mod hello_py3{
4      use pyo3::prelude::*;
5      /// Counts the number of occurrences of `needle` in
6      `contents`.
7      #[pyfunction]
8      fn count_ocurrences(contents: &str, needle: &str) -> usize{
9          let mut total = 0;
10         for line in contents.lines(){
11             for word in line.split(" "){
12                 if word == needle{
13                     total += 1;
14                 }
15             }
16             total
17         }
18     }

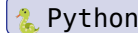
```



```

1  import hello_py3
2
3  contents = "a b c d"
4
5  hello_py3.count_ocurrences(contents, needle = "a")

```



## Python API

## Rust Source

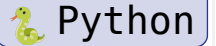
## ¿Qué hace el intérprete cuando llamamos a esta función?

```
1 import hello_py03
2
3 contents = "a b c d"
4
5 hello_py03.count_ocurrences(contents, needle = "a")
```



Python

# ¿Qué hace el intérprete cuando llamamos a esta función?



```
1  import dis
2
3  dis.dis('count_ocurrences("a b c d e", needle="a")')
4      0          0 RESUME          0
5
6      1          2 PUSH_NULL
7          4 LOAD_NAME          0 (count_ocurrences)
8          6 LOAD_CONST        0 ('a b c d e')
9          8 LOAD_CONST        1 ('a')
10         10 KW_NAMES          2 (('needle',))
11         12 CALL              2
12         20 RETURN_VALUE
```

# Bibliography

Hilliard, D. (2022). *Publishing Python Packages. Test, share, and automate your projects*. Manning. <https://www.manning.com/books/publishing-python-packages>

Mara, L., & Holmes, J. (2025). *Refactoring to Rust*. Manning. <https://www.manning.com/books/refactoring-to-rust>

McNamara, T. (2021). *Rust in Action. Systems programming concepts and techniques*. Manning. <https://www.manning.com/books/rust-in-action>

Rodrigues, T. (2023). *Fast Python. High performance techniques for large datasets*. Manning. <https://www.manning.com/books/fast-python>