**Building Portable Native Python Extensions With Rust, PyO3, And Maturin**

[[](https://medium.com/@kudryavtsev_ia?source=post_page-----3c1a1634d324--------------------------------)](https://medium.com/@kudryavtsev_ia?source=post_page-----3c1a1634d324--------------------------------)

[[Inside In-Sight](https://blog.savant-ai.io/?source=post_page-----3c1a1634d324--------------------------------)](https://blog.savant-ai.io/?source=post_page-----3c1a1634d324--------------------------------)

[Ivan Kud](https://medium.com/@kudryavtsev_ia?source=post_page-----3c1a1634d324--------------------------------)

·

Follow

Published in

[Inside In-Sight](https://blog.savant-ai.io/?source=post_page-----3c1a1634d324--------------------------------)

·

9 min read

·

Jun 20

31



When you develop native Python extensions in Rust, you must address several aspects to make them work for various environments. The article discusses those aspects in detail and demonstrates recipes for building more complex extensions using additional dependencies.

*The article concerns only extensions built with PyO3; if you make the integration with #[no\_mangle] extern unsafe "C" functions, you may need to modify the approach, but I don’t know how to deal with that because I rely on PyO3/Maturin.*

Let us understand what is required to make an extension that works (almost) everywhere:

1. build for all hardware platforms supported (x86, x86–64, aarch64, etc);
2. build for all operating systems (Linux, Windows, MacOS);
3. build for various releases for a selected OS;
4. build for various Python versions like 3.7–3.11.

It looks like a lot of work! But, it is not as bad as it looks at first sight: Maturin can generate a CI manifest for GitHub Actions, which builds all the matrix! However, if you don’t use GitHub Actions in a project, it doesn’t help you, requiring you to support the builds manually (which I demonstrate in the last part of the article). As for now, let us assume you use them because the chances that you are doing that are high!

**Maturin With GitHub Actions**

Maturin has a subcommand to generate a working CI manifest for GitHub Actions. Let us imagine that extension contains a rust library with PyO3; you can generate CI for this library with the following Maturin command:

maturin generate-ci -m extension/Cargo.toml

You can limit platforms and operating systems, but by default without limiting, you will get:

# This file is autogenerated by maturin v0.15.1  
# To update, run  
#  
# maturin generate-ci -m extension/Cargo.toml github  
#  
name: CI  
  
on:  
 push:  
 branches:  
 - main  
 - master  
 tags:  
 - '\*'  
 pull\_request:  
 workflow\_dispatch:  
  
permissions:  
 contents: read  
  
jobs:  
 linux:  
 runs-on: ubuntu-latest  
 strategy:  
 matrix:  
 target: [x86\_64, x86, aarch64, armv7, s390x, ppc64le]  
 steps:  
 - uses: actions/checkout@v3  
 - uses: actions/setup-python@v4  
 with:  
 python-version: '3.10'  
 - name: Build wheels  
 uses: PyO3/maturin-action@v1  
 with:  
 target: ${{ matrix.target }}  
 args: --release --out dist --find-interpreter --manifest-path extension/Cargo.toml  
 sccache: 'true'  
 manylinux: auto  
 - name: Upload wheels  
 uses: actions/upload-artifact@v3  
 with:  
 name: wheels  
 path: dist  
  
 windows:  
 runs-on: windows-latest  
 strategy:  
 matrix:  
 target: [x64, x86]  
 steps:  
 - uses: actions/checkout@v3  
 - uses: actions/setup-python@v4  
 with:  
 python-version: '3.10'  
 architecture: ${{ matrix.target }}  
 - name: Build wheels  
 uses: PyO3/maturin-action@v1  
 with:  
 target: ${{ matrix.target }}  
 args: --release --out dist --find-interpreter --manifest-path extension/Cargo.toml  
 sccache: 'true'  
 - name: Upload wheels  
 uses: actions/upload-artifact@v3  
 with:  
 name: wheels  
 path: dist  
  
 macos:  
 runs-on: macos-latest  
 strategy:  
 matrix:  
 target: [x86\_64, aarch64]  
 steps:  
 - uses: actions/checkout@v3  
 - uses: actions/setup-python@v4  
 with:  
 python-version: '3.10'  
 - name: Build wheels  
 uses: PyO3/maturin-action@v1  
 with:  
 target: ${{ matrix.target }}  
 args: --release --out dist --find-interpreter --manifest-path extension/Cargo.toml  
 sccache: 'true'  
 - name: Upload wheels  
 uses: actions/upload-artifact@v3  
 with:  
 name: wheels  
 path: dist  
  
 sdist:  
 runs-on: ubuntu-latest  
 steps:  
 - uses: actions/checkout@v3  
 - name: Build sdist  
 uses: PyO3/maturin-action@v1  
 with:  
 command: sdist  
 args: --out dist --manifest-path extension/Cargo.toml  
 - name: Upload sdist  
 uses: actions/upload-artifact@v3  
 with:  
 name: wheels  
 path: dist  
  
 release:  
 name: Release  
 runs-on: ubuntu-latest  
 if: "startsWith(github.ref, 'refs/tags/')"  
 needs: [linux, windows, macos, sdist]  
 steps:  
 - uses: actions/download-artifact@v3  
 with:  
 name: wheels  
 - name: Publish to PyPI  
 uses: PyO3/maturin-action@v1  
 env:  
 MATURIN\_PYPI\_TOKEN: ${{ secrets.PYPI\_API\_TOKEN }}  
 with:  
 command: upload  
 args: --skip-existing \*

The pipeline generates wheels for Linux (x86\_64, x86, aarch64, armv7, s390x, ppc64le), Windows (x86, x86–64), MacOS (x86\_64, aarch64), and sdist (I remove this step instantly because I don’t need sdist for Rust code usually).

The final step of the manifest (release) uploads the artifacts on PyPi (you need to configure the MATURIN\_PYPI\_TOKEN secret for your repo); the stage triggers only for tagged versions (releases).

Just save the manifest as .github/workflows/CI.yml , commit changes, push to remote, and you are all set.

The action’s build command uses the --find-interpreter flag, which instructs Maturin to build the extension for all available Python interpreters.

Probably you want to ask: “All available where?”. It is a good point: let us take a look at the stages to find the answer. It is obvious for Windows and MacOS:

windows:  
 runs-on: windows-latest  
 strategy:  
 matrix:  
 target: [x64, x86]  
 steps:  
 - uses: actions/checkout@v3  
 - uses: actions/setup-python@v4  
 with:  
 python-version: '3.10'  
 architecture: ${{ matrix.target }}

It installs Python 3.10; if you need other versions, you have to add steps using other versions. Let us look at the MacOS step:

macos:  
 runs-on: macos-latest  
 strategy:  
 matrix:  
 target: [x86\_64, aarch64]  
 steps:  
 - uses: actions/checkout@v3  
 - uses: actions/setup-python@v4  
 with:  
 python-version: '3.10'

The same picture here: it installs 3.10.

**Linux Builds**

The build process works differently for Linux: it uses [Manylinux](https://github.com/pypa/manylinux" \t "_blank) docker images to build wheels for multiple Python interpreters.

*ManyLinux is a Linux distribution compatibility standard that aims to provide a consistent and reliable environment for running Python software on different Linux distributions. It was initially developed by the Python Packaging Authority (PyPA) to address the challenge of distributing binary Python packages that can be installed and run across various Linux distributions without the need for recompilation.*

*The name “ManyLinux” stems from the fact that there are many different Linux distributions with varying library versions and configurations. These differences can lead to compatibility issues when trying to distribute binary packages, as they may rely on specific versions of system libraries that are not available or have different names on different distributions.*

*The ManyLinux standard specifies a set of rules and guidelines for building binary wheels (pre-compiled packages) that are compatible with a wide range of Linux distributions. It defines a specific set of system libraries that must be present and prescribes a consistent naming scheme for those libraries, regardless of the underlying distribution.*

*By adhering to the ManyLinux standard, package maintainers can create binary wheels that can be installed on many different Linux distributions without the need for source code compilation or modification. This greatly simplifies the distribution process and makes it easier for Python developers to provide pre-compiled packages that work across a broad range of Linux distributions.*

*ManyLinux wheels are identified by a naming convention that includes a “manylinux” tag followed by a version number, such as “manylinux2014” or “manylinux2010”. This tag indicates the specific version of the ManyLinux standard that the wheel complies with, allowing users to select the appropriate version for their target distribution.*

*Overall, ManyLinux helps improve the compatibility and ease of distribution for Python software on Linux, reducing the burden of managing different library dependencies and enabling developers to reach a broader audience of Linux users.*

Now that you have read the note above let me formulate simply what is a Manylinux Docker image: it is a container based on ancient Linux releases like CentOS7 and Almalinux 8 packed with all possible Python versions placed into the /opt/python directory, e.g., /opt/python/cp37-cp37m/bin/python. Several popular Python versions, from 3.7 to 3.12, exist in the current manylinux\_2\_28 container. Aside from CPython, there are also versions for PyPy, optimizing JIT.

Maturin is smart: when --find-interpreter is set, it discovers all Python environments and builds for them. Thus, with Manylinux, you receive great automation out of the box.

Now, let us investigate what to do when the above-discussed approach doesn’t work.

**When It Does Not Work**

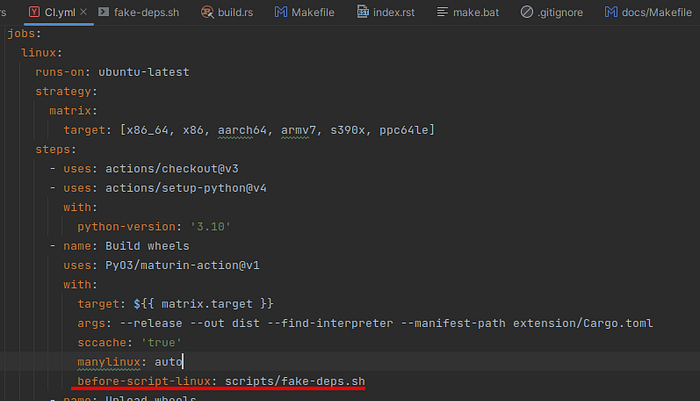
Up until this point, I found two cases where you may fail using the default approach:

* the first, obviously, is when you don’t use GitHub Actions;
* the second is when you need extra dependencies installed in the build container (the problem mostly concerns Manylinux builds) before you can build your package.

Let us first discuss the second problem related to dependencies.

**Installing Dependencies**

Fortunately, the Maturin PyO3/action provides you with the parameter before-script-linux which executes the script in the build container right before the build process:



The script must address the specifics of each potential Manylinux container candidate because Maturin may invoke a specific Manylinux container for certain architecture: you may meet the situation when in one container, the YUM package manager is available, in other DNF works, and so forth. So the script must be smart enough to determine the environment and install dependencies in a compatible way.

*Take a look at the*[*project*](https://github.com/insight-platform/pyo3-sphinx-documentation)*on GitHub to check how before-script-linux is used.*

**Custom Build Without GitHub Actions**

For simplicity's sake, imagine you want to build a portable Python extension locally. Let’s discuss how to do it.

*For those, who want just a look at the implementation, rather than read the rest, take a look at my*[*repo*](https://github.com/insight-platform/FFmpeg-Input)*, where it is implemented.*

Assume we want to build only for two modern environments: X86–64 and AARCH64. To do that, we will use Manylinux Docker images directly to make our custom-built images. I will demonstrate the solution on my repo FFmpeg-Input.

The plan is as follows:

1. take the Manylinux image as a source for our container;
2. install dependencies;
3. install Maturin;
4. install Rust;
5. build wheels;
6. copy wheels to the host system.

Create a Dockerfile with the following content:

FROM quay.io/pypa/manylinux\_2\_28\_x86\_64 as base

If you plan to use AARCH, it would be (to build it, you need the buildx Docker extension or ARM hardware):

FROM quay.io/pypa/manylinux\_2\_28\_aarch64 as base

Next, let us add dependencies:

FROM quay.io/pypa/manylinux\_2\_28\_aarch64 as base  
  
# system deps (in my case for FFmpeg)  
#  
RUN dnf install -y epel-release  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/free/el/rpmfusion-free-release-8.noarch.rpm  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/nonfree/el/rpmfusion-nonfree-release-8.noarch.rpm  
RUN dnf install -y ffmpeg ffmpeg-devel  
RUN dnf install -y clang clang-devel

By default, the PATH variable doesn’t include paths to Python interpreters available, so I add them manually:

FROM quay.io/pypa/manylinux\_2\_28\_aarch64 as base  
  
# system deps (in my case for FFmpeg)  
#  
RUN dnf install -y epel-release  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/free/el/rpmfusion-free-release-8.noarch.rpm  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/nonfree/el/rpmfusion-nonfree-release-8.noarch.rpm  
RUN dnf install -y ffmpeg ffmpeg-devel  
RUN dnf install -y clang clang-devel  
  
# add manylinux python interpreters we need  
#  
ENV PATH="/root/.cargo/bin:/opt/python/cp37-cp37m/bin:/opt/python/cp38-cp38/bin:/opt/python/cp39-cp39/bin:/opt/python/cp310-cp310/bin:/opt/python/cp311-cp311/bin:/opt/python/cp312-cp312/bin:$PATH"

Now it is time to install Maturin and its dependencies:

FROM quay.io/pypa/manylinux\_2\_28\_aarch64 as base  
  
# system deps (in my case for FFmpeg)  
#  
RUN dnf install -y epel-release  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/free/el/rpmfusion-free-release-8.noarch.rpm  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/nonfree/el/rpmfusion-nonfree-release-8.noarch.rpm  
RUN dnf install -y ffmpeg ffmpeg-devel  
RUN dnf install -y clang clang-devel  
  
# add manylinux python interpreters we need  
#  
ENV PATH="/root/.cargo/bin:/opt/python/cp37-cp37m/bin:/opt/python/cp38-cp38/bin:/opt/python/cp39-cp39/bin:/opt/python/cp310-cp310/bin:/opt/python/cp311-cp311/bin:/opt/python/cp312-cp312/bin:$PATH"  
  
# maturin installation  
#  
RUN pip3 install maturin==0.15 patchelf cffi ziglang sccache>=0.4.0

The next step is to install Rust. First, let us create an auxiliary script with Rust installation instructions in docker/install-basic-deps-manylinux.sh:

#!/usr/bin/env bash  
  
set -e  
  
curl -o rustup.sh --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs  
sh rustup.sh -y  
source $HOME/.cargo/env  
rustup update  
rustc -V  
  
cargo install cargo-chef --locked

The last command installs the cargo-chef extension — a very handy tool helping Docker layers remain intact when there is no real change. It speeds up builds by utilizing Docker layer caching.

Now, we may copy it to Docker, install it, and create a new image chef that is used to build the extension:

FROM quay.io/pypa/manylinux\_2\_28\_aarch64 as base  
  
# system deps (in my case for FFmpeg)  
#  
RUN dnf install -y epel-release  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/free/el/rpmfusion-free-release-8.noarch.rpm  
RUN dnf install -y --nogpgcheck https://download1.rpmfusion.org/nonfree/el/rpmfusion-nonfree-release-8.noarch.rpm  
RUN dnf install -y ffmpeg ffmpeg-devel  
RUN dnf install -y clang clang-devel  
  
# add manylinux python interpreters we need  
#  
ENV PATH="/root/.cargo/bin:/opt/python/cp37-cp37m/bin:/opt/python/cp38-cp38/bin:/opt/python/cp39-cp39/bin:/opt/python/cp310-cp310/bin:/opt/python/cp311-cp311/bin:/opt/python/cp312-cp312/bin:$PATH"  
  
# maturin installation  
#  
RUN pip3 install maturin==0.15 patchelf cffi ziglang sccache>=0.4.0  
  
# install Rust  
WORKDIR /opt  
COPY docker/install-basic-deps-manylinux.sh .  
RUN bash /opt/install-basic-deps-manylinux.sh  
  
# Our base image with Rust, Maturin, and system dependencies  
#  
FROM base as chef  
ENV PATH="/root/.cargo/bin:$PATH"  
RUN rustc -V

Great! We are heading straight to build steps, but first, here is a little chef magic:

# THE TOP IS SKIPPED  
# ...  
# ...  
  
# Our base image with Rust, Maturin, and system dependencies  
#  
FROM base as chef  
ENV PATH="/root/.cargo/bin:$PATH"  
RUN rustc -V  
  
# if chef detects that deps are changed, recipe.json will differ from  
# the one in cache and the next steps will be executed in full,  
# otherwise dependencies are used from cache  
#   
FROM chef AS planner  
WORKDIR /opt  
COPY . .  
RUN cargo chef prepare --recipe-path recipe.json  
  
# check if recipe.json is changed  
#   
FROM chef AS builder  
WORKDIR /opt  
COPY --from=planner /opt/recipe.json recipe.json # HERE is the magic! Changed/Not changed!  
RUN cargo chef cook --release --recipe-path recipe.json

Now, add build commands:

# THE TOP IS SKIPPED  
# ...  
# ...  
  
# build all possible wheels  
#  
maturin build -f --release --out dist  
  
# remove files to release disk space  
#  
RUN rm -rf target  
  
# copy wheels to a small image without unnecessary files  
#   
FROM alpine:3.18 as dist  
COPY --from=builder /opt/dist /opt/dist

Finally, run the build process:

docker build -t build-container -f Dockerfile .

Copy resulting wheels in the host system:

docker run --rm -it -v $(pwd)/distfiles:/tmp build-container \  
 cp -R /opt/dist /tmp

Now the $(pwd)/distfiles contains all built wheels.

**Conclusion**

You may find the complete code in my [FFmpeg-Input](https://github.com/insight-platform/FFmpeg-Input" \t "_blank) repo. Also, check [CI](https://github.com/insight-platform/FFmpeg-Input/blob/main/.github/workflows/ci.yml) demonstrating using custom Manylinux Docker images with GitHub Actions.