A Comprehensive Walkthrough of Python/C++ Binding Creation

Kapilan Ramasamy · Follow

8 min read · Aug 28, 2023

Introduction

This article provides a comprehensive guide to creating Python bindings for C++ code using the pybind11 library. Python bindings enable seamless integration between C++ and Python code, allowing you to leverage C++ functionalities within your Python applications. We'll walk through the process of setting up the project, writing the necessary code, and building the bindings. By the end of this guide, you'll have a fully functional Python binding that allows you to use C++ functionalities from Python.

Navigating the Integration

Setting Up the Development Environment

Configure your development environment for both C++ and Python and make sure there are no compile or runtime errors in your C++ project.

Install the necessary tools and libraries, such as a C++ compiler and the chosen Python binding library.

Necessary dependencies

- cmake

- python3
- pip

For this integration we will use pybind11. Go to the pybind11 GitHub and download the library and place it in your c++ project folder.

Writing Python Wrapper Code

Main.cpp

This file contains the main C++ code that we want to bind to Python. It includes the necessary headers, defines functions, and uses the pybind11 library to create bindings for these functions. Notable sections:

- `printName` : A function that takes two strings as input and returns a combined message.
- `multiply` : A function that takes two ints to perform multiplication.
- `divide` : A function that utilizes a C++ `Calculator` class to perform division.

```
// main.cpp
#include <pybind11/pybind11.h>

#include "./src/calculations.h"

#include <iostream>

using namespace std;

#define STRINGIFY(x) #x
#define MACRO_STRINGIFY(x) STRINGIFY(x)

string printName(string name, string lastname) {
    return "This function works " + name + " " + lastname + "!";
}

double divide(double num1, double num2) {

     Calculator calculator;
```

```
     double result;

     result = calculator.divide(num1, num2);

     return result;
}

namespace py = pybind11;

PYBIND11_MODULE(testing, m) {
    m.doc() = R"pbdoc(
        Pybind11 example plugin
        -----------------------

        .. currentmodule:: testing

        .. autosummary::
           :toctree: _generate

           add
           subtract
    )pbdoc";

    m.def("printName", &printName, R"pbdoc(
        Print first and last name

        Some other explanation about the printName function.
    )pbdoc");

    m.def("multiply", [](int i, int j) { return i * j; }, R"pbdoc(
        Multiply two numbers

        Some other explanation about the multiply function.
    )pbdoc");

      m.def("divide", &divide, R"pbdoc(
        Divide two numbers

        Some other explanation about the divide function.
    )pbdoc");

#ifdef VERSION_INFO
    m.attr("__version__") = MACRO_STRINGIFY(VERSION_INFO);
#else
    m.attr("__version__") = "dev";
#endif
}
```

If you want to import files or libraries into main.cpp such as “calculations.h”(which is in the src folder). Here is the sample code:

Calculations.cpp and .h:

```
// calculations.cpp

#include "calculations.h"

double Calculator::add(double a, double b) {
    return a + b;
}

double Calculator::subtract(double a, double b) {
    return a - b;
}

double Calculator::times(double a, double b) {
    return a * b;
}

double Calculator::divide(double a, double b) {
    if (b == 0) {
        throw "Division by zero is not allowed.";
    }
    return a / b;
}
```

```
// calculations.h
#ifndef CALCULATIONS_H
#define CALCULATIONS_H

class Calculator {
public:
    double add(double a, double b);
    double subtract(double a, double b);
    double times(double a, double b);
    double divide(double a, double b);
};

#endif // CALCULATIONS_H
```

CMakeLists.txt:

This file is used by CMake to manage the project build process. It sets up the project, includes necessary subdirectories, and links the C++ code to the bindings. It also handles passing version information to C++ code. This file also makes the calculations.cpp visible and therefore able to be used in main.cpp

```
cmake_minimum_required(VERSION 3.4...3.18)
project(binding_example)

add_subdirectory(pybind11)
pybind11_add_module(testing main.cpp)

add_library(worker1 STATIC src/calculations.cpp)
set_target_properties(worker1 PROPERTIES POSITION_INDEPENDENT_CODE ON)

# EXAMPLE_VERSION_INFO is defined by setup.py and passed into the C++ code as a
# define (VERSION_INFO) here.
target_compile_definitions(testing
                           PRIVATE VERSION_INFO=${EXAMPLE_VERSION_INFO})


target_link_libraries(testing PRIVATE worker1)
```

Setup.py:

This is a Python script that configures the project build and packaging process. It sets up the project metadata, specifies build options, and integrates CMake into the build process. It also defines the build_ext class, which manages the build extension process.

```
# setup.py
import os
import re
import subprocess
import sys
from pathlib import Path

from setuptools import Extension, setup
from setuptools.command.build_ext import build_ext

# Convert distutils Windows platform specifiers to CMake -A arguments
PLAT_TO_CMAKE = {
    "win32": "Win32",
    "win-amd64": "x64",
    "win-arm32": "ARM",
    "win-arm64": "ARM64",
}

# A CMakeExtension needs a sourcedir instead of a file list.
# The name must be the _single_ output extension from the CMake build.
# If you need multiple extensions, see scikit-build.
class CMakeExtension(Extension):
    def __init__(self, name: str, sourcedir: str = "") -> None:
        super().__init__(name, sources=[])
        self.sourcedir = os.fspath(Path(sourcedir).resolve())

class CMakeBuild(build_ext):
    def build_extension(self, ext: CMakeExtension) -> None:
        # Must be in this form due to bug in .resolve() only fixed in Python 3.
        ext_fullpath = Path.cwd() / self.get_ext_fullpath(ext.name)
        extdir = ext_fullpath.parent.resolve()

        # Using this requires trailing slash for auto-detection & inclusion of
        # auxiliary "native" libs

        debug = int(os.environ.get("DEBUG", 0)) if self.debug is None else self
        cfg = "Debug" if debug else "Release"

        # CMake lets you override the generator - we need to check this.
```

```python
        # Can be set with Conda-Build, for example.
        cmake_generator = os.environ.get("CMAKE_GENERATOR", "")

        # Set Python_EXECUTABLE instead if you use PYBIND11_FINDPYTHON
        # EXAMPLE_VERSION_INFO shows you how to pass a value into the C++ code
        # from Python.
        cmake_args = [
            f"-DCMAKE_LIBRARY_OUTPUT_DIRECTORY={extdir}{os.sep}",
            f"-DPYTHON_EXECUTABLE={sys.executable}",
            f"-DCMAKE_BUILD_TYPE={cfg}",  # not used on MSVC, but no harm
        ]
        build_args = []
        # Adding CMake arguments set as environment variable
        # (needed e.g. to build for ARM OSx on conda-forge)
        if "CMAKE_ARGS" in os.environ:
            cmake_args += [item for item in os.environ["CMAKE_ARGS"].split(" ")

        # In this example, we pass in the version to C++. You might not need to
        cmake_args += [f"-DEXAMPLE_VERSION_INFO={self.distribution.get_version(

        if self.compiler.compiler_type != "msvc":
            # Using Ninja-build since it a) is available as a wheel and b)
            # multithreads automatically. MSVC would require all variables be
            # exported for Ninja to pick it up, which is a little tricky to do.
            # Users can override the generator with CMAKE_GENERATOR in CMake
            # 3.15+.
            if not cmake_generator or cmake_generator == "Ninja":
                try:
                    import ninja

                    ninja_executable_path = Path(ninja.BIN_DIR) / "ninja"
                    cmake_args += [
                        "-GNinja",
                        f"-DCMAKE_MAKE_PROGRAM:FILEPATH={ninja_executable_path}
                    ]
                except ImportError:
                    pass

        else:
            # Single config generators are handled "normally"
            single_config = any(x in cmake_generator for x in {"NMake", "Ninja"

            # CMake allows an arch-in-generator style for backward compatibilit
            contains_arch = any(x in cmake_generator for x in {"ARM", "Win64"})

            # Specify the arch if using MSVC generator, but only if it doesn't
            # contain a backward-compatibility arch spec already in the
            # generator name.
```

```
            if not single_config and not contains_arch:
                cmake_args += ["-A", PLAT_TO_CMAKE[self.plat_name]]

            # Multi-config generators have a different way to specify configs
            if not single_config:
                cmake_args += [
                    f"-DCMAKE_LIBRARY_OUTPUT_DIRECTORY_{cfg.upper()}={extdir}"
                ]
                build_args += ["--config", cfg]

        if sys.platform.startswith("darwin"):
            # Cross-compile support for macOS - respect ARCHFLAGS if set
            archs = re.findall(r"-arch (\S+)", os.environ.get("ARCHFLAGS", ""))
            if archs:
                cmake_args += ["-DCMAKE_OSX_ARCHITECTURES={}".format(";".join(a

        # Set CMAKE_BUILD_PARALLEL_LEVEL to control the parallel build level
        # across all generators.
        if "CMAKE_BUILD_PARALLEL_LEVEL" not in os.environ:
            # self.parallel is a Python 3 only way to set parallel jobs by hand
            # using -j in the build_ext call, not supported by pip or PyPA-buil
            if hasattr(self, "parallel") and self.parallel:
                # CMake 3.12+ only.
                build_args += [f"-j{self.parallel}"]

        build_temp = Path(self.build_temp) / ext.name
        if not build_temp.exists():
            build_temp.mkdir(parents=True)

        subprocess.run(
            ["cmake", ext.sourcedir, *cmake_args], cwd=build_temp, check=True
        )
        subprocess.run(
            ["cmake", "--build", ".", *build_args], cwd=build_temp, check=True
        )

# The information here can also be placed in setup.cfg - better separation of
# logic and declaration, and simpler if you include description/version in a fi
setup(
    name="binding_example",
    version="0.0.1",
    author="your_name",
    author_email="you@gmail.com",
    description="A test project using pybind11 and CMake",
    long_description="",
    ext_modules=[CMakeExtension("testing")],
    cmdclass={"build_ext": CMakeBuild},
    zip_safe=False,
```

```
    extras_require={"test": ["pytest>=6.0"]},
    python_requires=">=3.7",
)
```

Pyproject.toml:

This TOML configuration file defines build-system requirements and tools used in the project. It specifies the required versions of setuptools, CMake, and other tools.

```
# pyproject.tom
[build-system]
requires = [
    "setuptools>=42",
    "wheel",
    "ninja",
    "cmake>=3.12",
]
build-backend = "setuptools.build_meta"

[tool.mypy]
files = "setup.py"
python_version = "3.7"
strict = true
show_error_codes = true
enable_error_code = ["ignore-without-code", "redundant-expr", "truthy-bool"]
warn_unreachable = true

[[tool.mypy.overrides]]
module = ["ninja"]
ignore_missing_imports = true


[tool.pytest.ini_options]
minversion = "6.0"
addopts = ["-ra", "--showlocals", "--strict-markers", "--strict-config"]
xfail_strict = true
filterwarnings = [
    "error",
    "ignore:(ast.Str|Attribute s|ast.NameConstant|ast.Num) is deprecated:Deprec
]
testpaths = ["tests"]

[tool.cibuildwheel]
test-command = "pytest {project}/tests"
test-extras = ["test"]
```

```
test-skip = ["*universal2:arm64"]
# Setuptools bug causes collision between pypy and cpython artifacts
before-build = "rm -rf {project}/build"

[tool.ruff]
extend-select = [
  "B",    # flake8-bugbear
  "B904",
  "I",    # isort
  "PGH",  # pygrep-hooks
  "RUF",  # Ruff-specific
  "UP",   # pyupgrade
]
extend-ignore = [
  "E501",   # Line too long
]
target-version = "py37"
```

File: Manifest.in

This file lists the additional files to be included when packaging the project for distribution. It ensures that necessary files like licenses, headers, and CMake configuration files are included in the distribution. This file is crucial for making the src folder included in the build process so the calculations.h and .cpp are seen.

```
include README.md LICENSE pybind11/LICENSE
graft pybind11/include
graft pybind11/tools
graft src
global-include CMakeLists.txt *.cmake
```

Recap:

To create Python bindings for C++ code using pybind11, follow these steps:

1. Go to the pybind11 GitHub and download the library and place it in your c++ project folder.

2. Write the C++ code in `main.cpp`, defining the functions you want to bind to

Python. Import any files or libraries, but make sure to adjust the code accordingly in CMakeLists.txt, and the Manifest.in

3. Use `CMakeLists.txt` to set up the project, include subdirectories, and link the C++ code to the bindings.
4. Configure `setup.py` with project metadata, build options, and integration with CMake.
5. Define `pyproject.toml` to specify build-system requirements and tools.
6. List additional files in `Manifest.in` to ensure they're included in the distribution.

Before we continue make sure your project directory looks something like this:

```
project_directory/
│
├── src/
│   ├── calculations.cpp
│   ├── calculations.h
│   └── ... (other C++ source/header files)
│
├── main.cpp
├── CMakeLists.txt
├── setup.py
├── pyproject.toml
├── Manifest.in
├── test.py
│
└── README.md
```

After setting up these files, navigate to the project directory in the terminal and run the following commands:

```
python setup.py sdist bdist_wheel
pip install .
```

After running `python setup.py sdist bdis_wheel` , there should be three new files in directory called build, dist, binding_exmaple.egg-info. So the logs the show might be confusing, but to simply whats going on in the terminal, the log provides a detailed overview of the steps involved in building a Python package with C++ bindings using the pybind11 library. The process includes creating source distributions, generating metadata, building C++ extensions, creating binary distribution wheels, and installing the package. The log also highlights warnings about missing files and potential os version compatibility issues.

The main important part that indicates success of binding is if the "**cpython-39-darwin.so" is built at 100%.** Only then can we proceed to run pip install .

Finally, run the `test.py` file to verify that the Python binding works correctly. Here is an example:

```
import testing as m


output = m.printName("Kapilan","Ramasamy")

print(output)

print("You are " + str(m.multiply(10,2)) + " years old!")
print("In 10 years you will be " + str(m.divide(90,3)) + " years old!")
```

If everything works as expected, congratulations! You've successfully created a Python binding for C++ code using the pybind11 library.

Here is a more advanced example of a project that uses python binding:

GitHub - AutomataLab/JSONSki_python

Contribute to AutomataLab/JSONSki_python development by creating an account on GitHub.

github.com

Remember that Python bindings offer a powerful way to combine the strengths of Python and C++, allowing you to create high-performance applications with the ease of Python development.

Python Binding C Programming

Follow

Written by Kapilan Ramasamy

1 Follower

life enthusiast

Recommended from Medium

Amazon.com Seattle, WA
Software Development Engineer Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by $25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)

- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with < 2.2s runtime

HeatMap (JavaScript)

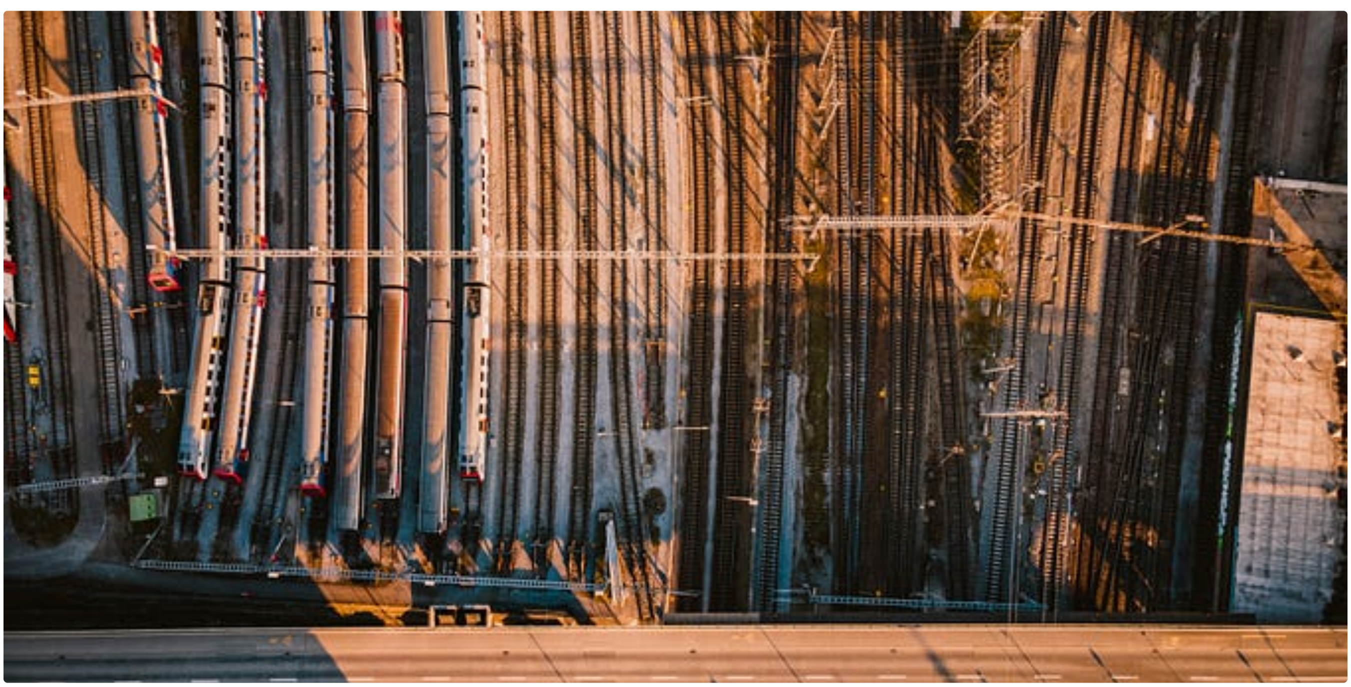
- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay

Alexander Nguyen in Level Up Coding

The resume that got a software engineer a $300,000 job at Google.

1-page. Well-formatted.

May 31 18.4K 299



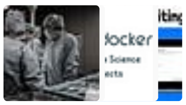
Andrew Zuo

Async Await Is The Worst Thing To Happen To Programming

I recently saw this meme about async and await.

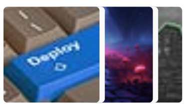
Jun 21 · 2.9K · 166

Lists



Coding & Development

11 stories · 752 saves

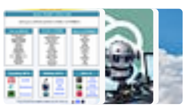


Predictive Modeling w/ Python

20 stories · 1458 saves

Practical Guides to Machine Learning

10 stories · 1776 saves



ChatGPT

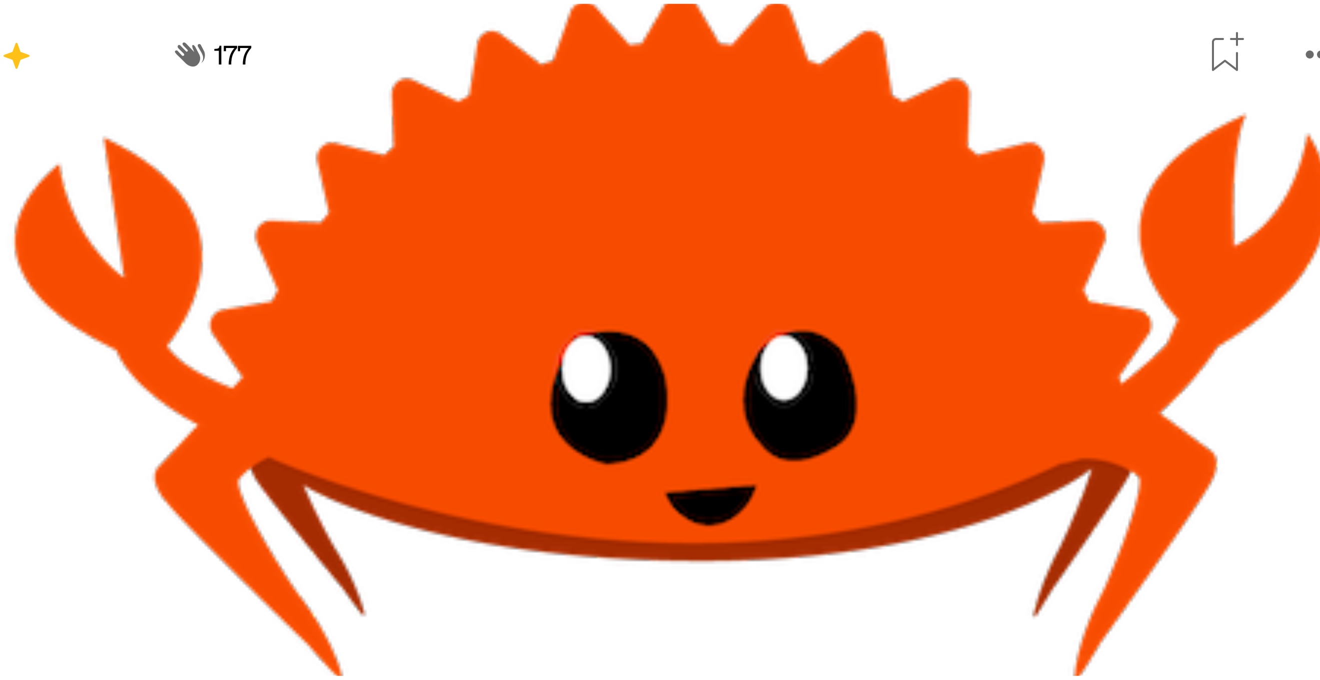
21 stories · 767 saves

Gajanan Rajput

How To Use ThreadPoolExecutor in Python 3

Dive into the world of Python concurrency with our comprehensive guide on ThreadPoolExecutor. Learn how to effectively manage threads for...

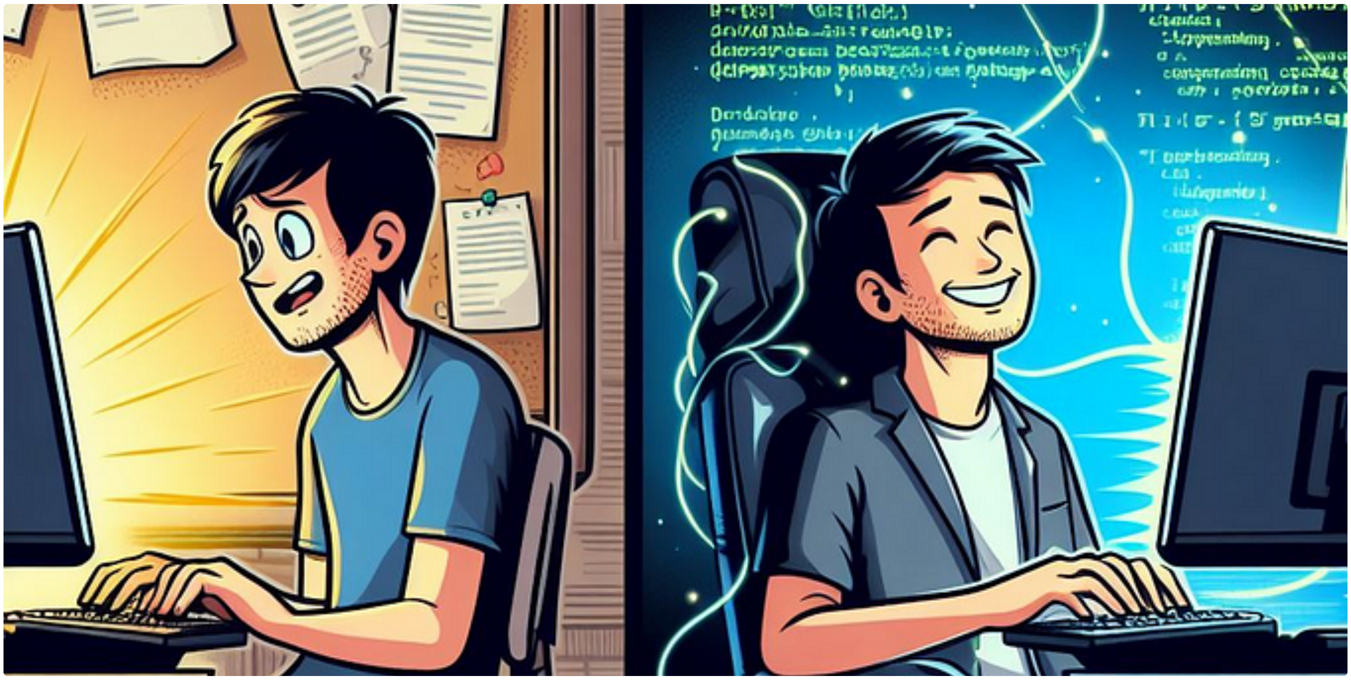
177



Austin Starks in CodeX

I spent 18 months rebuilding my algorithmic trading platform in Rust. I'm filled with regret.

Jun 27 3.8K 150



Abhay Parashar in The Pythoneers

17 Mindblowing Python Automation Scripts I Use Everyday

Scripts That Increased My Productivity and Performance

Jul 28 6.6K 56

Brother Nan

Practice 2: Implementing a Read/Write Lock

In this post, let's implement a read/write lock, a common interview question that tests your understanding of threading and synchronization...

Mar 4 1

See more recommendations