C++ and Python: Building Robust Applications by Offloading Compute-Heavy Workloads

Hariharan Ragothaman
conf42 Python2025
Track: Data

About Me

- Software Engineer at AMD
- Previously Lead Software Engineer at athenahealth
  - Also worked at Bain Capital and Bose Corporation
- Areas of Interest: DevSecOps, Distributed Systems, Applied Artificial Intelligence, Embedded Systems
- LinkedIn: https://www.linkedin.com/in/hariharanragothaman/
- GitHub: https://github.com/hariharanragothaman

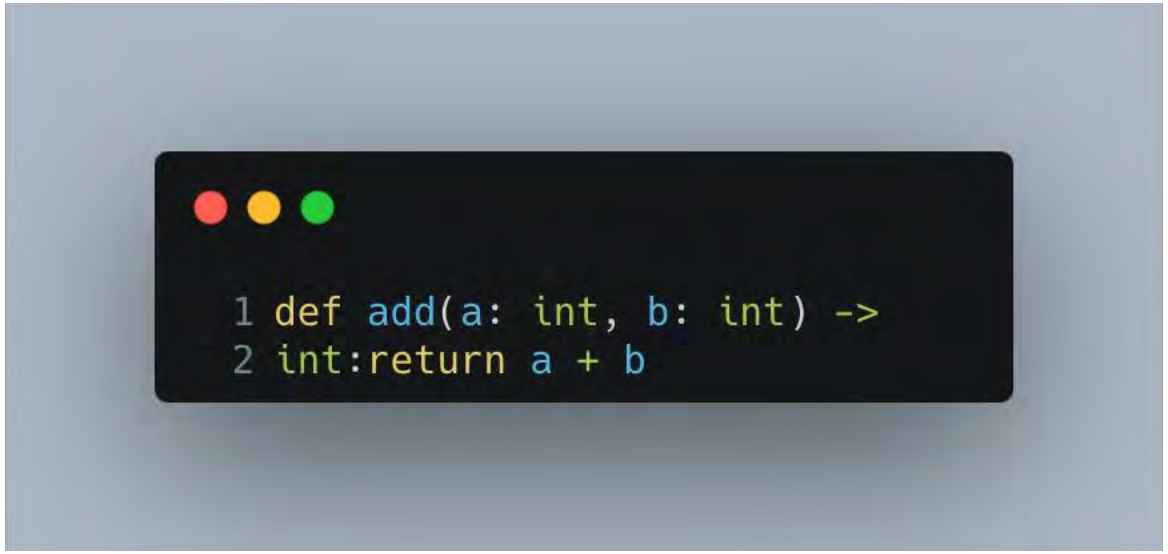
AGENDA

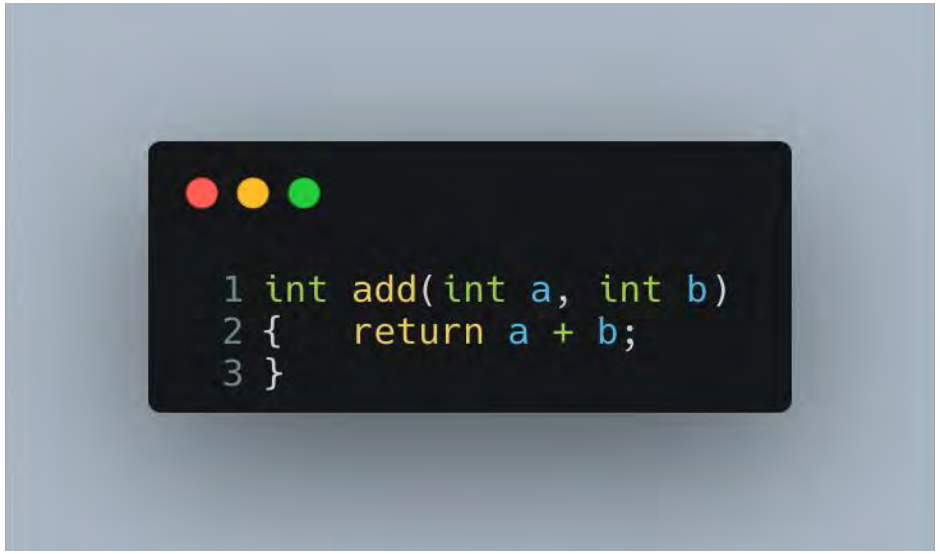
- Introduction
- Similarities and Difference b/w Python and C++
- Overview of Boost.Python and pybind11
- Project Setup
- Minimal Examples
- Demo
- Advanced Topics
- Distribution and Packaging
- Top Takeaways

Similarities b/w C++ and Python | Problem Statement

Why Python?

1. Faster Development
2. Relatively Simpler Interfaces and Usage
3. Best Option for Prototyping





Similarities b/w C++ and Python

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

void greet_all(const vector<string>& names)
{   for (const string& name : names) {
        cout << "Hello, " << name << endl;
    }
}

```

```python
from typing import List

def greet_all(names: List[str]) ->
None:for name in names:
        print(f"Hello, {name}")
```

Similarities b/w C++ and Python

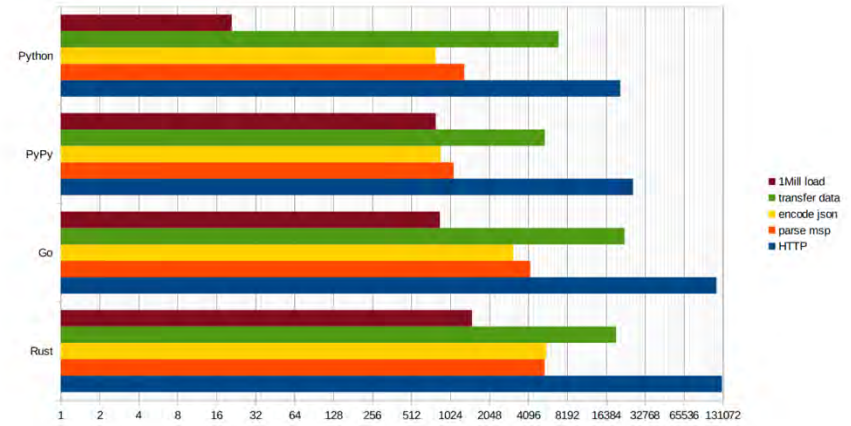
```python
from typing import TypeVar, Generic

T = TypeVar('T')

class Box(Generic[T]):
    def __init__(self, content: T) ->
None:   self.content = content

    def get_content(self) -> T:
        return self.content

# Usage examples:
int_box = Box(123)
str_box = Box("Hello")

```

```cpp
template<typename T>
class Box {
public:
    Box(const T& content) : content(content)
{}
    T getContent() const {
        return content;
    }

private:
    T content;
};

// Usage examples:
Box<int> intBox(123);
Box<std::string> strBox("Hello");

```

Performance (or) other Needs?

1. Benchmarking
2. Cross-functional teams in an organization.
3. numpy? And other external libraries?
    1. Abstraction over contiguous multi-dimensional array
    2. Easy to use API for many mathematical function
    3. Core Implementation is in C and C++

Note: Python 3.13 is about 15% faster for module imports in large projects, and up to 10% faster for function calls in CPU-intensive tasks
Also now has new JIT and no-GIL modes.



Here are the actual results in table format: (req/s)

	HTTP	parse msp	encode json	transfer data	1Mill load
Rust	128747.61	5485.43	5637.20	19551.83	1509.84
Go	116672.12	4257.06	3144.31	22738.92	852.26
PyPy	26507.69	1088.88	864.48	5502.14	791.68
Python	21095.92	1313.93	788.76	7041.16	20.94

Project Setup and Minimal Examples

Some Pre-Requisites:

1. A working C++ compiler that supports at least C++11.
2. Python development headers and libraries, which you can install
    1. Example: apt-get install python3-dev.
3. A build system, often CMake, to coordinate compilation, linking, and library paths.

Note:

For Boost.Python specifically, you also need Boost installed.
For pybind11, you can install it via a package manager or just copy the headers into your project.

Overview of Boost.python and pybind11

If you've used Python for prototyping or data analysis, but you have a performance bottleneck that begs for C++, you need a bridge between them. Both Boost.Python and pybind11 do this.

Boost.Python: Part of the larger Boost ecosystem, actively maintained for many years. It's stable, well-documented, and ships with a Python module linking library.

pybind11: A more modern alternative, heavily inspired by Boost.Python, but it's a lightweight, header-only library. If you dislike additional library dependencies and want an easy install, pybind11 might be your choice.

In practice, they both achieve the same result: they let you write C++ code that can be called from Python as if it were a Python module. Let's explore how to set up our environment."

References

Boost.Python – https://www.boost.org/doc/libs/1_78_0/libs/python/doc/html/index.html
Pybind11 - https://github.com/pybind/pybind11

Advanced Topics

1. Exposing C++ Classes
2. Exception Translation
   - *Both libraries let us map C+ exceptions to Python* exceptions to Python exceptions so your users see Pythonic stack traces.
3. Containers and Iterators
   - You can easily pass std::vector, std::array, or other containers back and forth, especially in pybind11 with its STL container support.
4. Integration with numpy
   - **Boost.Python** has a numpy submodule for arrays.**pybind11** has pybind11/numpy.h to handle NumPy arrays with minimal overhead.
5. Performance Considerations
   - Minimizing boundary crossings can help keep your Python/C++ integration fast. For large data sets, consider moving more logic into C++.

Distribution and Packaging

Once bindings are ready, you can:

1. **Use them locally**: Just copy the compiled .so or .pyd next to your Python scripts, or set your PYTHONPATH appropriately.
2. **Make a pip-installable package**: Create a setup.py using either vanilla setuptools or scikit-build. Then, you can publish to PyPI or share it with other so “pip install my_module” just works.

What about binding Performance?

There are definitely overheads

1. Data needs to go back and forth between C++ objects and PyObjects
2. There is implementation difference b/w Boost.Python and pybind11
    1. For example - pybind11 relies on smartpointers / vectors – which can make things slower.

Hence we need to choose the right parts of our code / product that need to be optimized.

Platform	Time (seconds)
C++	0.0042
Python	1.2206
Boost.Python	1.5278
pybind11	5.74263

Counter-Measures:

1. pybind11 and Boost.Python have special types that directly bind with python and C++.
    1. Example; py::dict in C++ code (Use binding objects for C++ conversion)
2. Optimized type findings such– py::array_t<double>
3. Use smart pointers instead of regular pointers. (python is a garbage collected language)

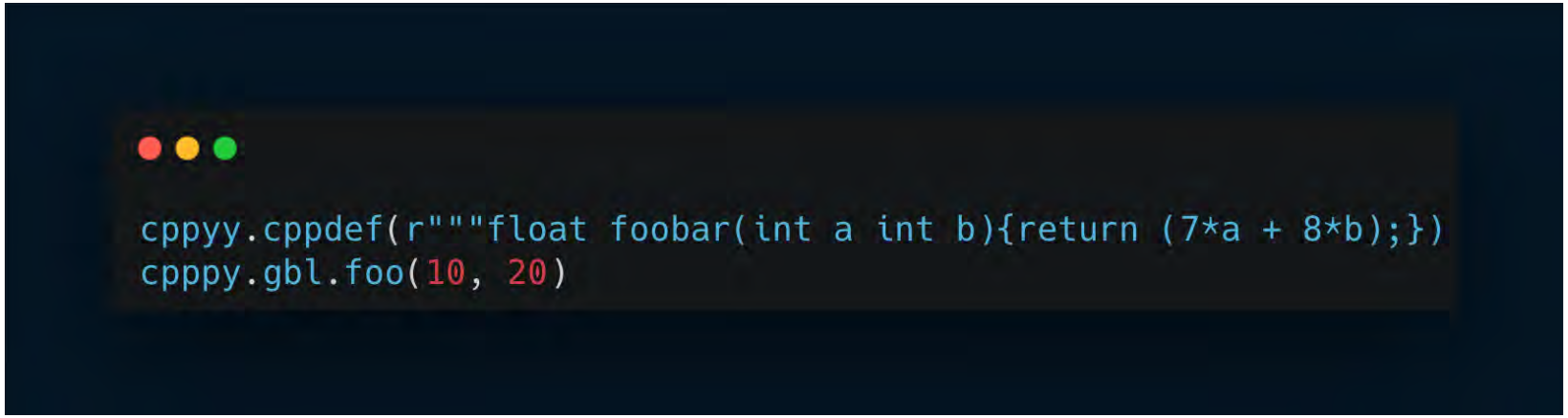
Dynamic Binding using Cppyy

1. Writes C++ directly into Python
2. Supports CPython and PyPy
3. Easy to Use – Much much faster than Pybind11 and Boost.python

Reference: https://github.com/root-project/cling - C++ Cling Interpreter

Advantages:

1. No need to install / compile
2. Full support for templates
3. Full support for inheritance
4. Full support for callbacks and lambdas



Top Takeaways

1. **Boost.Python** and **pybind11** are both excellent tools to blend the power of C++ with the expressiveness of Python.
2. Choose **Boost.Python** if you're already in the Boost ecosystem or need its robust set of features.
3. Choose **pybind11** if you want a lightweight, modern, header-only approach.
4. The basic pattern is always similar:
    1. Write a C++ function or class.
    2. Use a module-defining macro to expose it.
    3. Compile as a shared library.
    4. Import it in Python.
5. For broader usage, integrate with packaging tools like setuptools or scikit-build.
6. To dynamically bind – use **Cppyy** (that uses cling interpreter)

Note: Code in slide-deck is checked-in here: https://github.com/hariharanragothaman/conf42Python2025