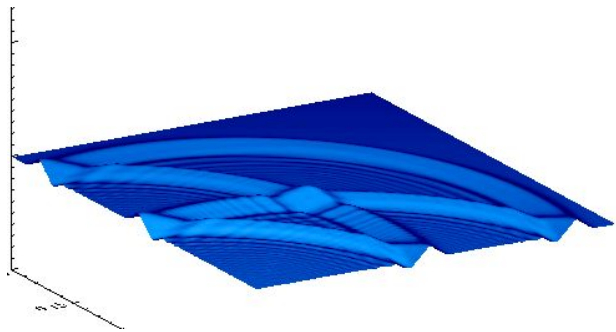
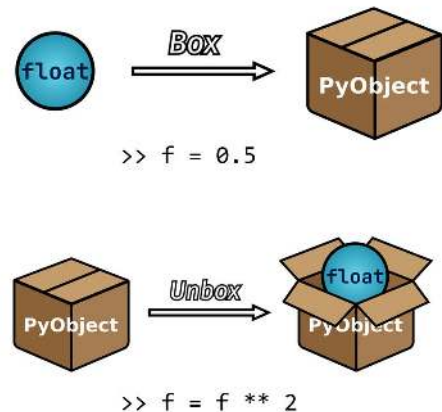
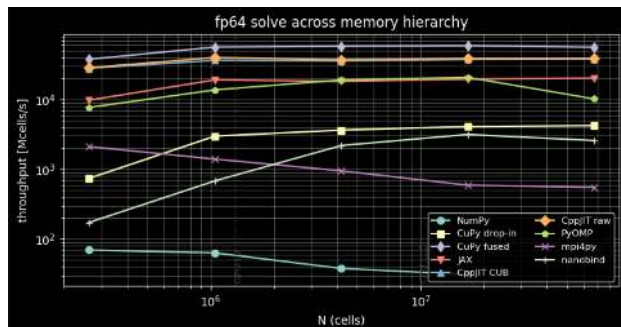




Programming Models, Libraries and Python/C++ Interoperability

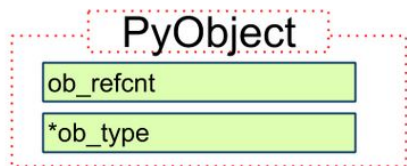
Hands-On Mini-App and Tutorial



Why is python slow?

Dynamic typing

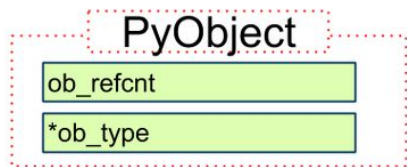
- Variables get a type only during runtime: Values(PyObjects) assigned to them.



Why is python slow?

Dynamic typing

- Variables get a type only during runtime: Values(PyObjects) assigned to them.

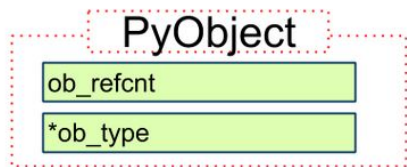


Python has only one data type, PyObject with a pointer to its runtime type, which is yet another PyObject*.*

Why is python slow?

Dynamic typing

- Variables get a type only during runtime: Values(PyObjects) assigned to them.



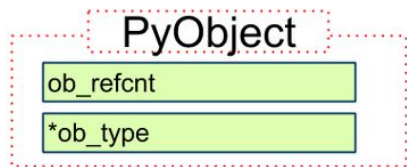
Python has only one data type, PyObject with a pointer to its runtime type, which is yet another PyObject*.*

- Hard for the interpreter to optimize the execution.

Why is python slow?

Dynamic typing

- Variables get a type only during runtime: Values(PyObjects) assigned to them.

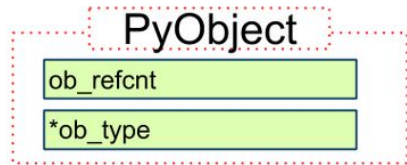


Python has only one data type, PyObject with a pointer to its runtime type, which is yet another PyObject*.*

- Hard for the interpreter to optimize the execution.
- Compilers can make extensive analysis and optimization before the execution.

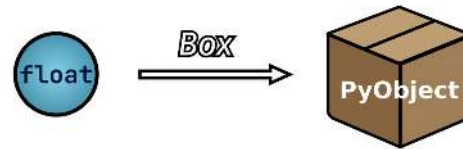
Why is python slow?

Dynamic typing

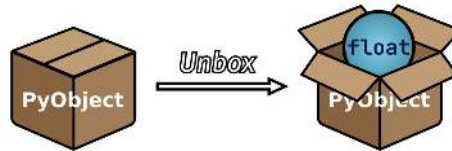


Python has only one data type, `PyObject*` with a pointer to its runtime type, which is yet another `PyObject*`.

*Python is a dynamically typed language (Duck Typing).
It wraps and later unwraps objects (referred to as boxing/unboxing)*



```
>> f = 0.5
```

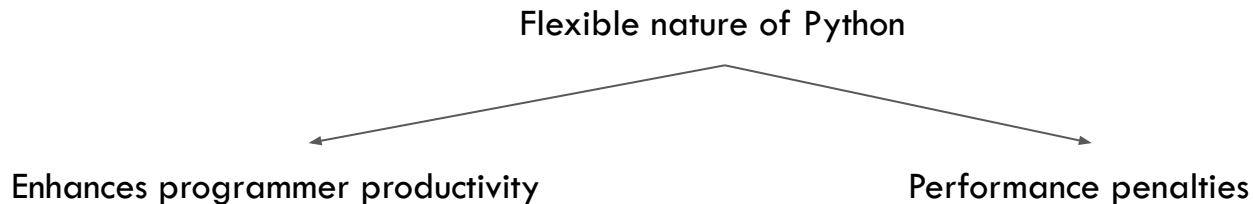


```
>> f = f ** 2
```

Why is python slow?

Flexible Data Structures

- Python builtins (lists, dictionaries, etc) -> Flexible
Downsides: Very generic, hence not well suited for extensive numerical computations.
- Data structure implementations efficient when processing diverse data
- Heavy overhead when processing only a single type of data.



Python is only slow if you are using it for the wrong task. We're going to look into ways to speed up simple Python workflows, either with native libraries or binding to low level code.

Why not just compile Python?

Several projects push Python toward native performance while keeping you (mostly) in Python:

- [Numba](#): JIT-compiles a numerical subset of Python (CPU/GPU)
- [Cython](#): ahead-of-time compiler for a Python superset
- [JAX](#): array programming with composable transforms (jit, grad, vmap), compiled via XLA to CPU/GPU/TPU
- etc.

Why not just compile Python?

Several projects push Python toward native performance while keeping you (mostly) in Python:

- [Numba](#): JIT-compiles a numerical subset of Python (CPU/GPU)
- [Cython](#): ahead-of-time compiler for a Python superset
- [JAX](#): array programming with composable transforms (jit, grad, vmap), compiled via XLA to CPU/GPU/TPU
- etc.

Each keeps your code close to Python — at the cost of a supported subset, a new mental model, or an extra toolchain. (and others: PyPy, Mojo, Codon, CPython's own JIT/free-threading...)

Today we take a different route: reuse the existing C/C++ world directly: **ctypes**, **cffi**, **nanobind**, **cppjit**.

Beyond Python....



In Python



Beyond Python - Bindings

A robust approach to speedups is by binding to faster compiled extensions (whether manual or automatic), like C/C++.

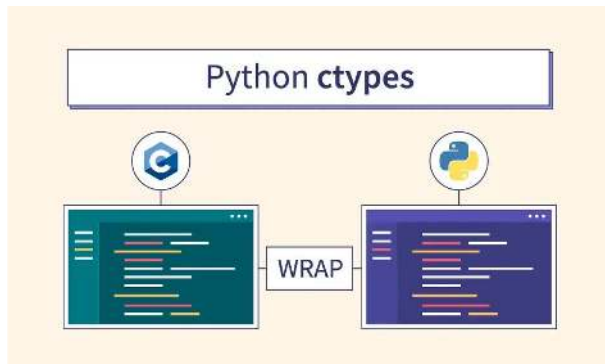
- Performance lives in small array kernels, not in end-to-end C++ workflows
- Just bounded loops, borrowed memory, that are callable from opaque interfaces
- With micro-kernels, maintaining large C++ codebases is not necessary
- That's why the C++ here is approachable, even more so with an LLM in the loop.

Beyond Python - Bindings

A robust approach to speedups is by binding to faster compiled extensions (whether manual or automatic), like C/C++.

How do we do that?

ctypes

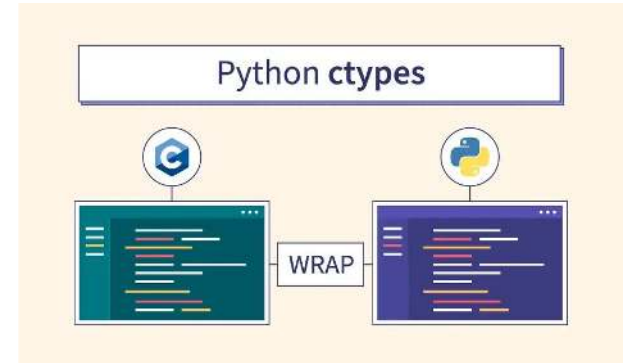


Ctypes

ctypes is a foreign function library for Python

It provides C compatible data types, and allows calling functions in DLLs or shared libraries

Allows wrapping these libraries in pure Python



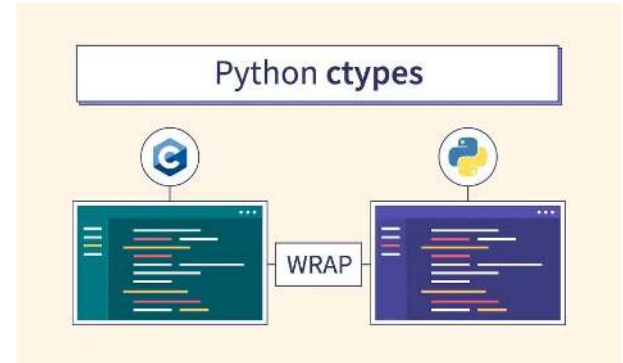
Ctypes

ctypes is a foreign function library for Python

It provides C compatible data types, and allows calling functions in DLLs or shared libraries

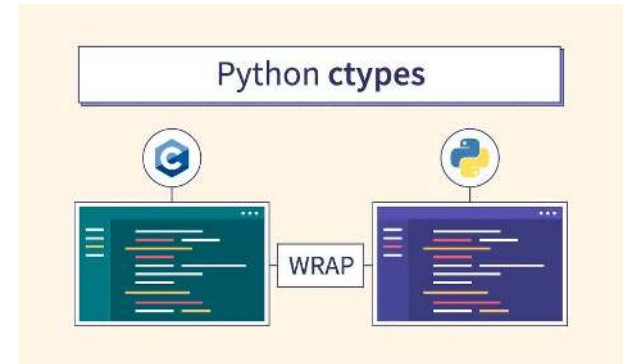
Allows wrapping these libraries in pure Python

Let's take a look at how we can compile, store and access these extensions later in the notebook.



Ctypes

What are the “compiled extensions” we are referring to?



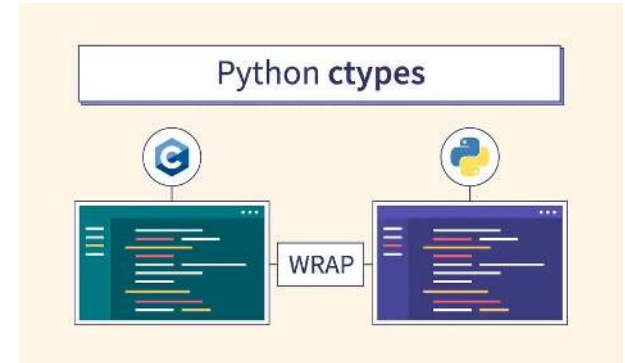
Ctypes

What are the “compiled extensions” we are referring to?

.so/.dll files

Dynamically linked libraries

Can be loaded at runtime by the OS



Ctypes

Example

- Library code in C
- Shared object creation
- Load library from Python
- Annotate signature
- Use the function

```
float square(float x) {  
    return x*x;  
}
```



```
gcc -shared -fPIC -o simple.so simple.c
```

```
import ctypes  
  
# Open library  
lib = ctypes.cdll.LoadLibrary(file_to_load)  
  
# Annotate signature  
lib.square.argtypes = (ctypes.c_float,)  
lib.square.restype = ctypes.c_float  
  
# Use function  
lib.square(x)
```



nanobind

nanobind

Lightweight C++17 library that exposes C++ types in Python and vice versa

nanobind

Lightweight C++17 library that exposes C++ types in Python and vice versa

Creates Python bindings to existing C++ code

nanobind

[Nonobind](#) is a Lightweight C++17 library that exposes C++ types in Python and vice versa, in the spirit of Boost.Python and Pybind11

Creates Python bindings to existing C++ code

Provides core C++ features in Python:

- Functions with value, reference, or pointer parameters and return types
- Instance methods and static methods
- Overloaded functions
- Callbacks

nanobind

Provides core C++ features in Python:

- Functions with value, reference, or pointer parameters and return types
- Instance methods and static methods
- Overloaded functions
- Callbacks
- Iterators and ranges
- Single and multiple inheritance
- STL data structures
- Smart pointers with reference counting like `std::shared_ptr`
- And more...

nanobind

Example

- Write Python extension module in C++ with nanobind
- Compile Python extension
- Import it from Python and use without further boilerplate

```
#include <nanobind/nanobind.h>
```

```
namespace nb = nanobind;
```

```
float square(float x) {  
    return x*x;  
}
```

```
NB_MODULE(pysimple, m) {  
    m.def("square", &square);  
}
```

```
c++ -std=c++17 pysimple.cpp -shared -o pysimple.so  
...
```

```
import pysimple
```

```
pysimple.square(3)
```



Automatic bindings - **cppjit**

cppjit: Yet another Python – C++ binder?

Automatic bindings - **cppjit**

cppjit: Yet another Python – C++ binder?

Yes, but bindings are **runtime**

Automatic bindings - **cppjit**

cppjit: Yet another Python – C++ binder?

Yes, but bindings are **runtime**

Python is all runtime, so runtime is more natural

Automatic bindings - **cppjit**

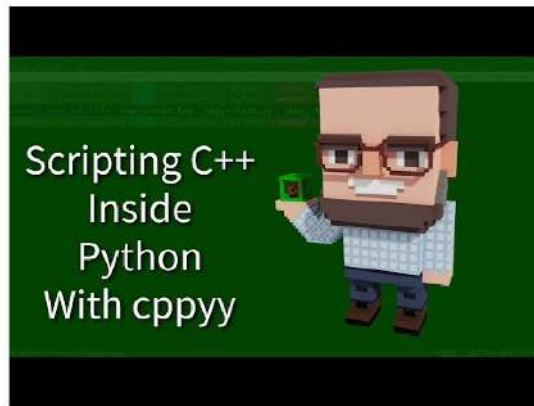
cppjit: Yet another Python – C++ binder?

Yes, but bindings are **runtime**

Python is all runtime, so runtime is more natural

C++-side runtime-ness is provided by [Clang-Repl](#) and [CpplInterOp](#)

History in particle physics: preceded by [PyROOT](#) and [cppyy](#)



[Video on cppyy](#) on the C++ weekly channel

Automatic bindings - **cppjit**

Run-time generation enables:

- Detailed specialization for higher performance
- Lazy loading for reduced memory use in large scale projects
- Python-side cross-inheritance and callbacks for working with C++ frameworks
- Run-time template instantiation
- Automatic object downcasting
- Exception mapping

cppjit removes the boundary between C++ and Python

Let's look at **some examples** on how this can be used.

Interactive C++ from Python

- No wrapper code
- Readily use standard library and custom classes

```
import cppjit
```

```
cppjit.include("pricing.hpp")
```

```
# Use your C++ class as if it was a Python class
```

```
p = cppjit.gbl.Portfolio()
```

```
p.add(100.0)
```

```
p.add(-40.5)
```

```
print("Total:", p.total())
```

```
# Output:
```

```
# Total: 59.5
```

Python

Generate and run C++ at Python runtime

- We can JIT compile C++ code **on the fly**
- Great for **rapid prototyping** and iterative development

Researches can gradually spin off performance-critical code to C++ for their own benefit, with **production ready C++ code** as a byproduct

- no “big rewrite moment”



```
import cppjit

cppjit.cppdef("""
double exotic_payoff(double spot) {
    if (spot > 100)
        return (spot - 100) * 2.0;
    else
        return 0.0;
}
""")

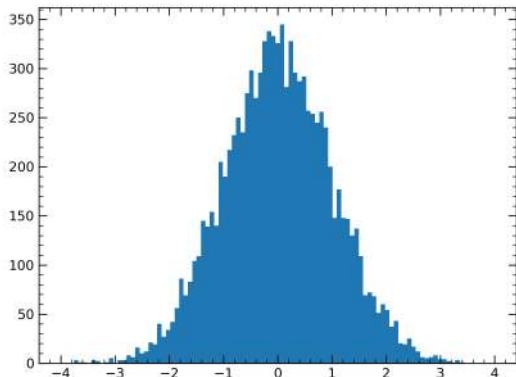
print(cppjit.gbl.exotic_payoff(120))

# Output:
# 40.0
```

Python as a C++ debugger: shining light into the “blackbox”

- Common C++ types are automatically “pythonized”, making them transparent to Python
- Example use case: great for exploratory debugging inside an existing C++ system!

```
std::vector<double> makePredictions() {  
    // ... implementation  
}  
  
void analyzePredictions(std::vector<double> const &) {  
    // ... implementation  
}
```



```
import cppjit  
import matplotlib.pyplot as plt  
  
preds = cppjit.gbl.makePredictions()  
  
plt.hist(preds, bins=np.linspace(4, 4, 100))  
plt.show()  
  
cppjit.gbl.analyzePredictions(preds)
```



Use C++ standard library like native Python Objects

You don't need adapters:

C++ containers **behave like Python containers.**

- **Template specialization** happens at runtime
- Your C++ data structures **don't need to be converted** to cross the language barrier

```
import cppjit
```

```
v = cppjit.gbl.std.vector["int"]()
```

```
v.push_back(10)
```

```
v.push_back(20)
```

```
v.push_back(30)
```

```
# Pythonic access
```

```
print("v[1]", v[1])
```

```
print("len ", len(v))
```

```
# Pythonic iteration
```

```
for x in v:
```

```
    print(x)
```



NumPy arrays as first-class citizens across languages

NumPy arrays are everywhere in scientific Python code.

- They **seamlessly convert** to C-style arrays and STL types (std::vector)
- Very handy for **bi-directional data transfer between the C++ and Python side**
 - NumPy arrays are essentially a view on contiguous memory

```
import cppjit
import numpy as np
```

```
a = np.array([10, 20, 30], dtype=np.int32)
```

```
# NumPy arrays can be converted to STL types:
```

```
std = cppjit.gbl
```

```
v = std.vector[int](a)
```

```
cppjit.cppdef("""
```

```
void inspect_arr(int const *arr, size_t n){
```

```
    for (size_t i = 0; i < n; ++i) {
```

```
        std::cout << arr[i] << std::endl;
```

```
    }
```

```
}
```

```
""")
```

```
# NumPy arrays implicitly convert
```

```
# to C-style arrays in function calls (zero copy):
```

```
cppjit.gbl.inspect_arr(a, len(a))
```



More function callbacks

Python logic injected into C++ pipelines
without wrapper glue or virtual inheritance:

- `std::function` callbacks to Python
 - Second callback mechanism besides virtual dispatching

We can plug Python-defined strategies into a high-performance C++ engine.

```
import cppjit

cppjit.cppdef("""
#include <functional>

double apply(double x, std::function<double(double)> f)
{
    return f(x);
}
""")

def my_func(x):
    return x * x + 1

print(cppjit.gbl.apply(3.0, my_func))

# Output:
# 10.0
```



Full example - zero-copy callback to Python for predictions

Advanced use case that ties together:

- NumPy array support
- std::function callbacks
- Python proxies of C++ types

Use case: call into Python-side AI inference from C++.

```
using Func = std::function<void(  
    size_t, double const*, double *)>;
```

```
void run_inference(Func func, std::vector<double> const &x)  
{  
    double out{};  
    func(x.size(), x.data(), &out);  
    std::cout << out << std::endl;  
}
```



```
def predict(n, x, out):  
    # zero-copy conversion from double * to numpy array  
    x = np.frombuffer(x, dtype=np.float64, count=n)  
  
    # Setting elements of output array just works!  
    out[0] = np.sum(np.square(x))  
  
x = np.array([1., 2., 3.])
```



```
cppjit.gbl.run_inference(predict, x) # Outputs 14
```

Error handling

Errors in multi-language systems can often be difficult and verbose, so we focus on the **essential information**.

Example on right: Error in Python callback.

C++ exceptions are also automatically wrapped in Python exceptions (e.g. `std::runtime_error`)

```
void run_callback(  
    std::function<void()> func) {  
    func();  
}
```



```
def func():  
    raise RuntimeError("Hello Python error")  
  
cppjit.gbl.run_callback(func)
```



```
Traceback (most recent call last):  
  File "error.py", line 12, in <module>  
    cppjit.gbl.run_callback(func)  
~~~~~  
  File "error.py", line 10, in func  
    raise RuntimeError("Hello Python error")  
RuntimeError: Hello Python error
```

Conclusions

- Combining **Python for steering** and **C/C++ for performance code** is an established pattern in HPC
- Solid **language interoperability** crucial for good developer experience
- We have seen some tools that operate on different levels:
 - **ctypes** for basic foreign function support in Python
 - **nanobind** for generating Python extensions that statically wrap your C/C++ code
 - **cppjit** for dynamic bindings and a **unified Python-C++ runtime**

Let's look at the notebook `07_cpp_interop.ipynb` to try these technologies

Mini App: Shallow Water Equation

Hands On

Next: Shallow Water Equation Mini App

- So far we have seen powerful array-oriented libraries in Python for GPU, and learnt how to profile and optimise your code.
- Now let's look at another aspect of the HPC Python landscape: Reaching for a different programming model or library, or dropping into a C/C++ kernel to integrate with an existing codebase.
- Some familiarity with C/C++ helps, but the syntax is not the focus.
- During this exercise, we solve the same problem with different approaches and compare the throughput each reaches on the same hardware.

Next: Shallow Water Equation Mini App

- We will start now (Monday afternoon) and wrap-up at the end of the Tuesday morning session
- Form teams of two. This exercise is involved, but provides of a lot of existing scaffolding (`swe_core`) that provides utilities and abstractions, so your focus can be on implementing the best possible algorithm, computing and memory model.