Extending the cppyy support in Numba

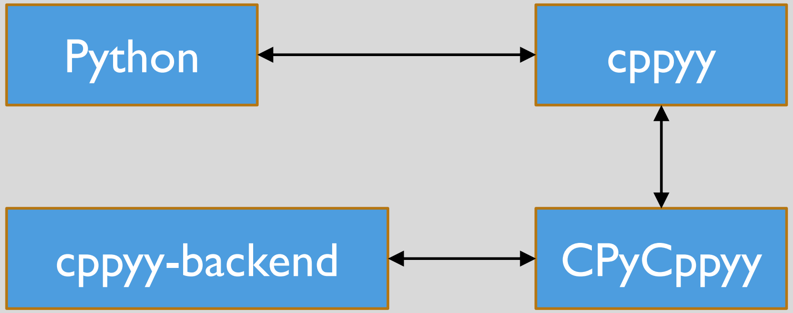
Aaron Jomy

cppyy

Numba

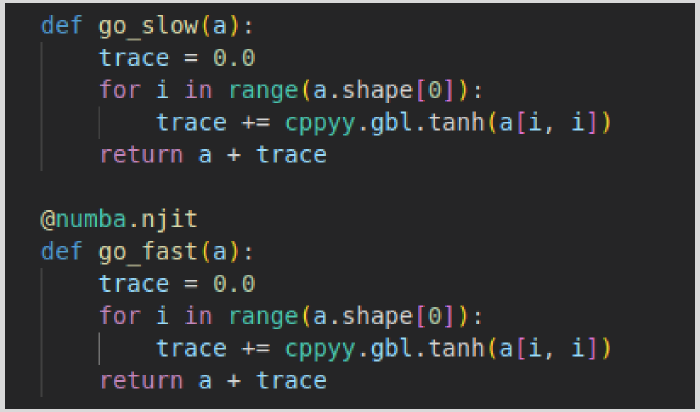
INTRODUCTION

- **Cppyy**:
  An automatic, run-time, Python-C++ bindings generator
- **Cling**
  is used in backend since an interactive C++ interpreter provides a runtime exec approach to C++ code
- **Numba**
  JIT compiler that translates Python and NumPy code into fast machine code.



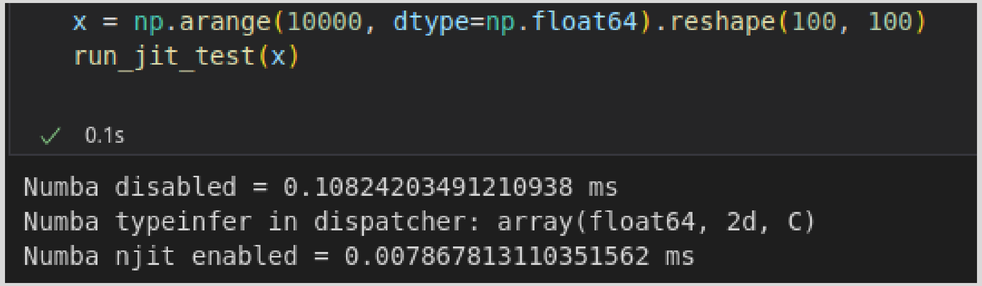
WHY USE NUMBA?

- The compute time overhead while switching between languages accumulates in loops with cppyy objects.



```python
def go_slow(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += cppyy.gbl.tanh(a[i, i])
    return a + trace

@numba.njit
def go_fast(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += cppyy.gbl.tanh(a[i, i])
    return a + trace
```

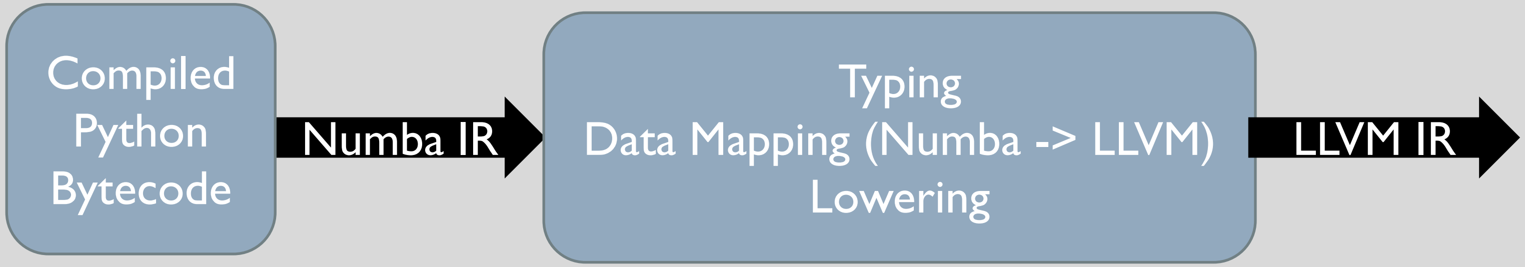
- Numba optimizes the loop and compiles it into machine code which crosses the language barrier only once



```
x = np.arange(10000, dtype=np.float64).reshape(100, 100)
run_jit_test(x)

✓ 0.1s

Numba disabled = 0.10824203491210938 ms
Numba typeinfer in dispatcher: array(float64, 2d, C)
Numba njit enabled = 0.007867813110351562 ms
```

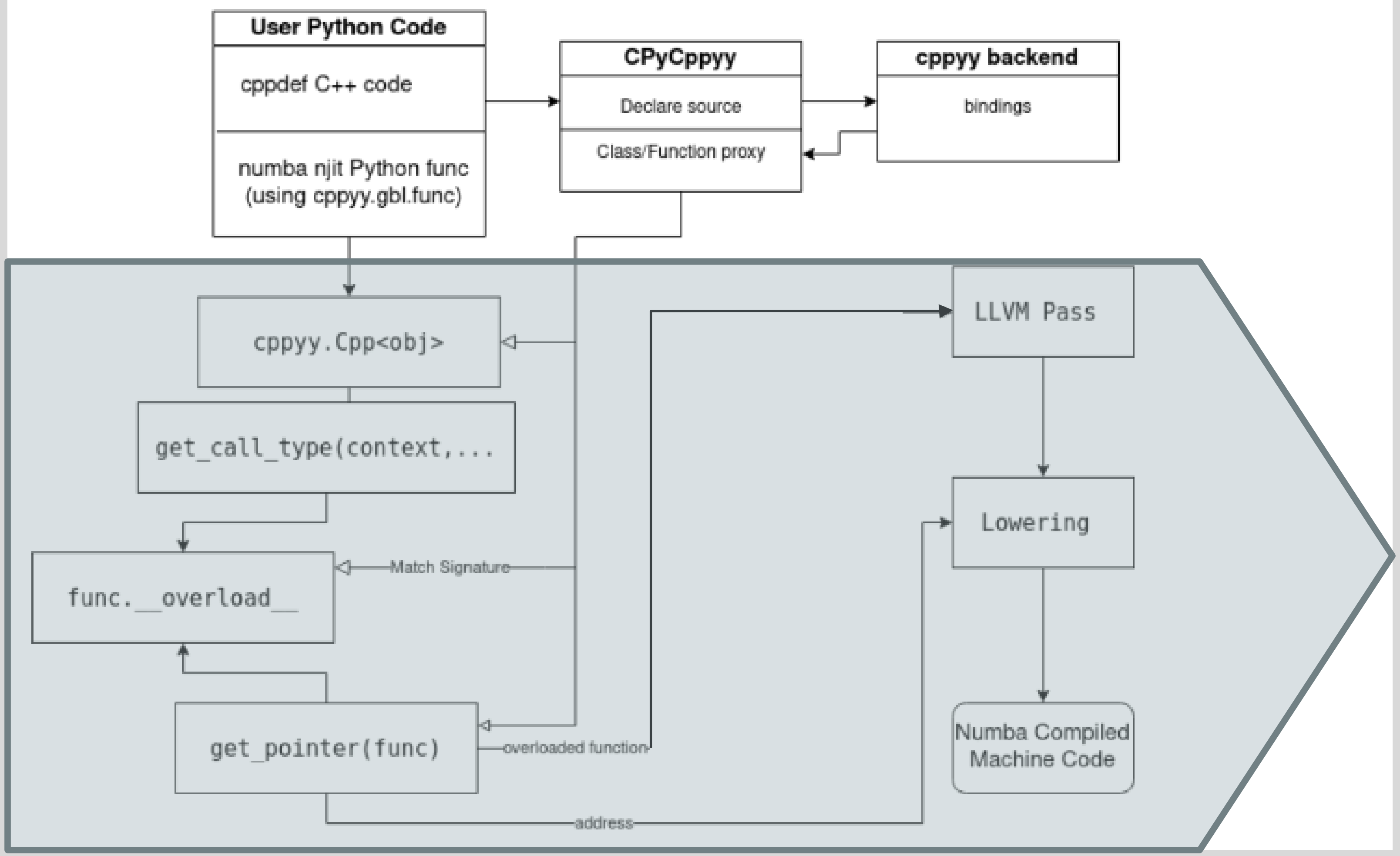
NUMBA PIPELINE

- **Typing**
  Numba core has a type inference algorithm which assigns a nb_type for a variable
- **Lowering**
  high-level Python operations into low-level LLVM code.
  Exploits typing to map to LLVM type
- **Boxing and unboxing**
  convert PyObject* ‘s into native values, and vice-versa.



We utilise the runtime numba compilation process to lower C++ code cppdef’ed in Python
How? →

NUMBA LOW LEVEL EXTENSION API

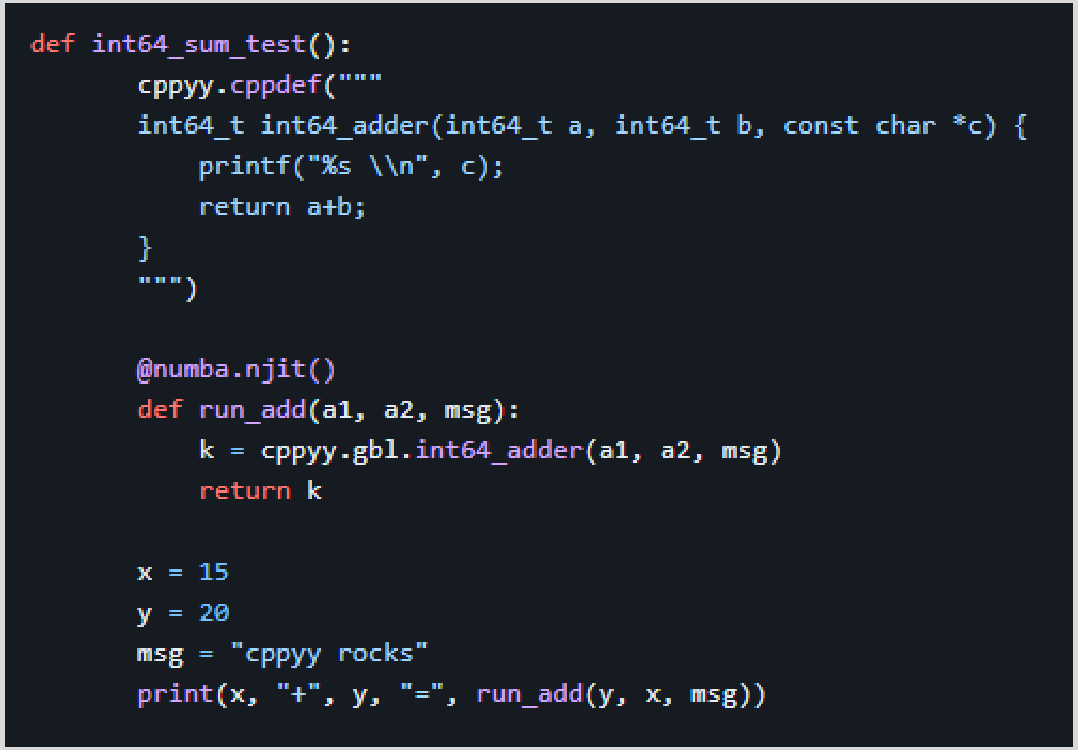


CPPYY NUMBA SUPPORT
cppyy/numba_ext.py

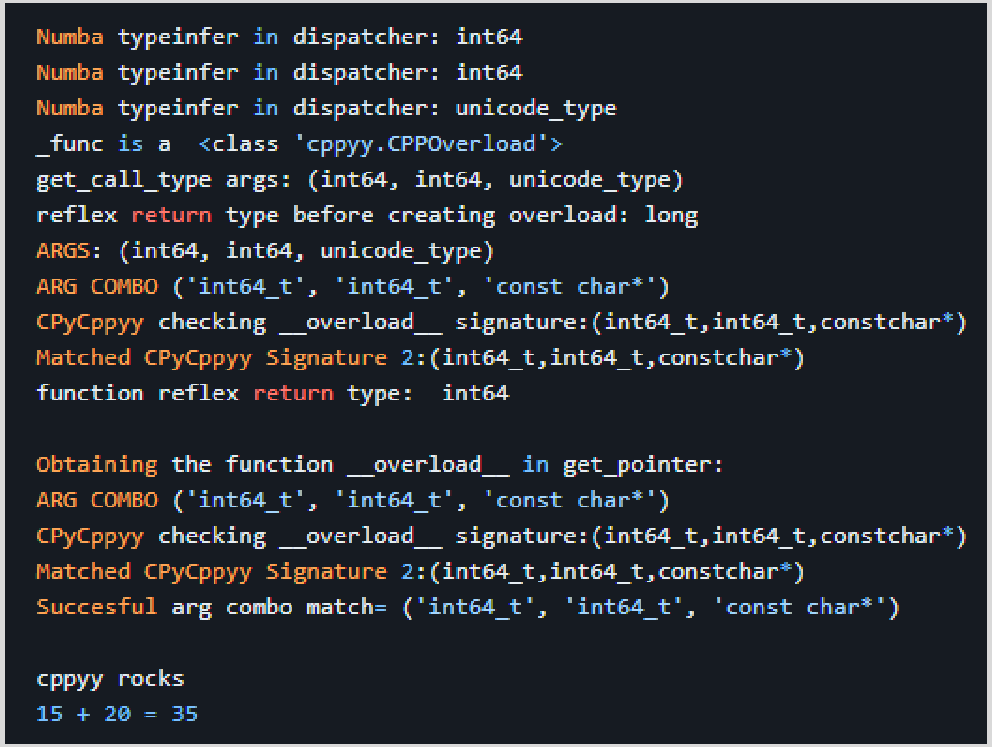
CHALLENGES

- Typing is one of the largest problems posed: Template function utilization, reference types and correct function matching depend on the type resolution system
- Type Inference solution:
  A mechanism to handle implicit casting based on propagated type info and the cppyy reflection layer.
- Note: Typing does not backtrack since the numba extension will only ever obtain the numba type inference result.

Python	Numba Type	LLVM Type used in Numba lowering
3 (int)	int64	i64
3.14 (float)	float64	double
(1, 2, 3)	UniTuple(int64, 3)	[3 x i64]
(1, 2.5)	Tuple(int64, float64)	{i64, double}
np.array([1, 2], dtype=np.int32)	array(int64, 1d, C)	{i8*, i8*, i64, i64, i32*, [1 x i64]}
“Hello”	unicode_type	{i8*, i64, i32, i32, i64, i8*, i8*}



```python
def int64_sum_test():
    cppyy.cppdef("""
    int64_t int64_adder(int64_t a, int64_t b, const char *c) {
        printf("%s \\n", c);
        return a+b;
    }
    """)

    @numba.njit()
    def run_add(a1, a2, msg):
        k = cppyy.gbl.int64_adder(a1, a2, msg)
        return k

    x = 15
    y = 20
    msg = "cppyy rocks"
    print(x, "+", y, "=", run_add(y, x, msg))
```

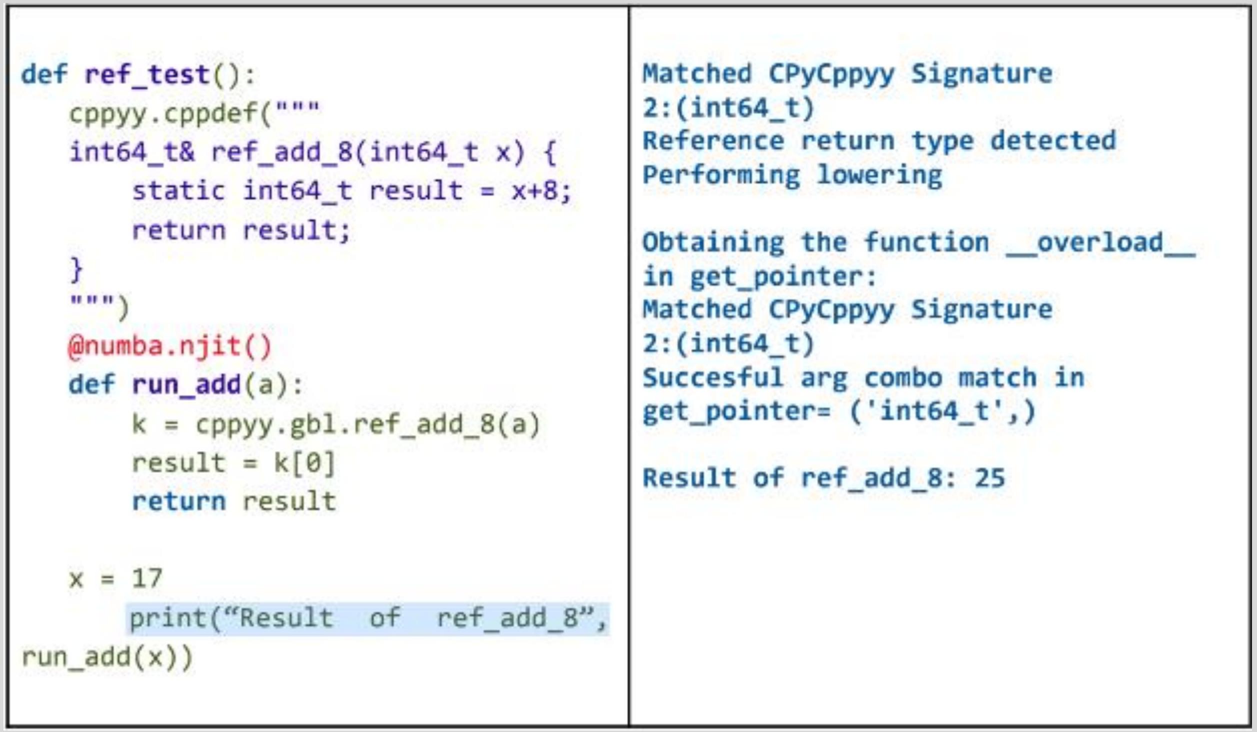


```
Numba typeinfer in dispatcher: int64
Numba typeinfer in dispatcher: int64
Numba typeinfer in dispatcher: unicode_type
_func is a  <class 'cppyy.CPPOverload'>
get_call_type args: (int64, int64, unicode_type)
reflex return type before creating overload: long
ARGS: (int64, int64, unicode_type)
ARG COMBO ('int64_t', 'int64_t', 'const char*')
CPyCppyy checking __overload__ signature:(int64_t,int64_t,constchar*)
Matched CPyCppyy Signature 2:(int64_t,int64_t,constchar*)
function reflex return type:  int64

Obtaining the function __overload__ in get_pointer:
ARG COMBO ('int64_t', 'int64_t', 'const char*')
CPyCppyy checking __overload__ signature:(int64_t,int64_t,constchar*)
Matched CPyCppyy Signature 2:(int64_t,int64_t,constchar*)
Succesful arg combo match= ('int64_t', 'int64_t', 'const char*')

cppyy rocks
15 + 20 = 35
```

PRIMARY DELIVERABLES:

- Add general support for C++ templates in Numba through Cppyy
- Add support for C++ reference types in Numba through Cppyy

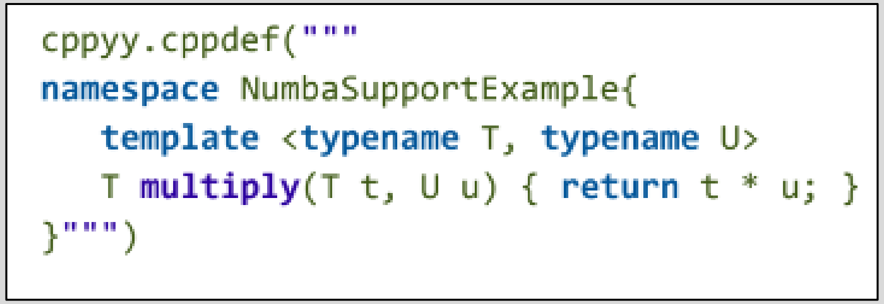
SOME EXAMPLES



```
def ref_test():
   cppyy.cppdef("""
   int64_t& ref_add_8(int64_t x) {
       static int64_t result = x+8;
       return result;
   }
   """)
   @numba.njit()
   def run_add(a):
       k = cppyy.gbl.ref_add_8(a)
       result = k[0]
       return result

   x = 17
       print(“Result  of  ref_add_8”,
run_add(x))
```

```
Matched CPyCppyy Signature
2:(int64_t)
Reference return type detected
Performing lowering

Obtaining the function __overload__
in get_pointer:
Matched CPyCppyy Signature
2:(int64_t)
Succesful arg combo match in
get_pointer= ('int64_t',)

Result of ref_add_8: 25
```

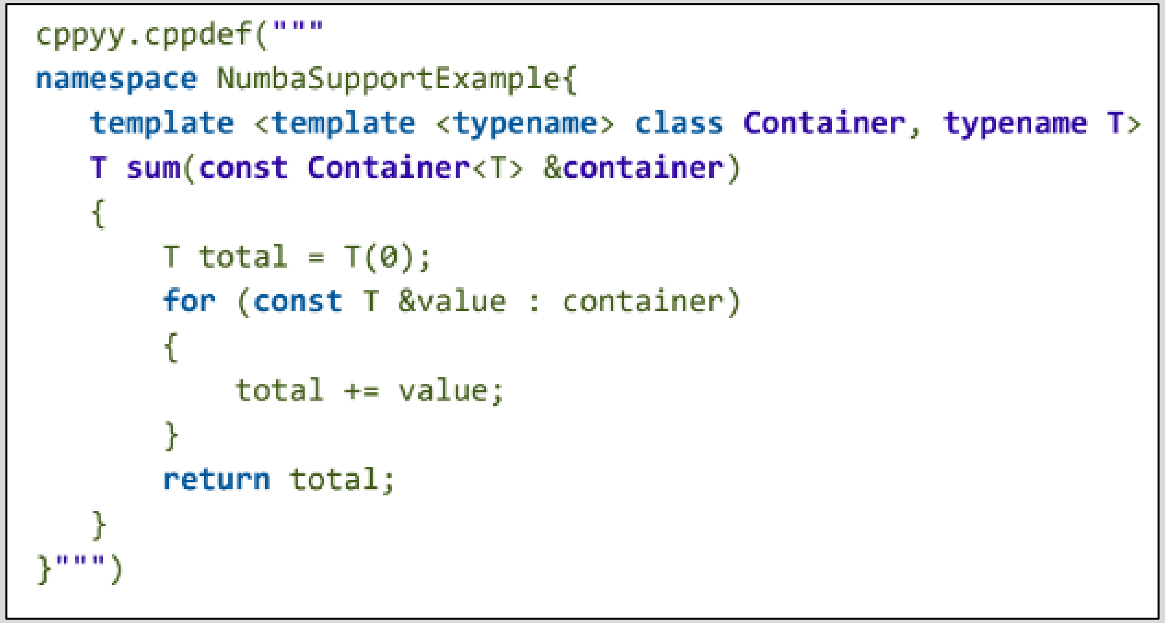
reference types



```
cppyy.cppdef("""
namespace NumbaSupportExample{
   template <typename T, typename U>
   T multiply(T t, U u) { return t * u; }
}""")
```

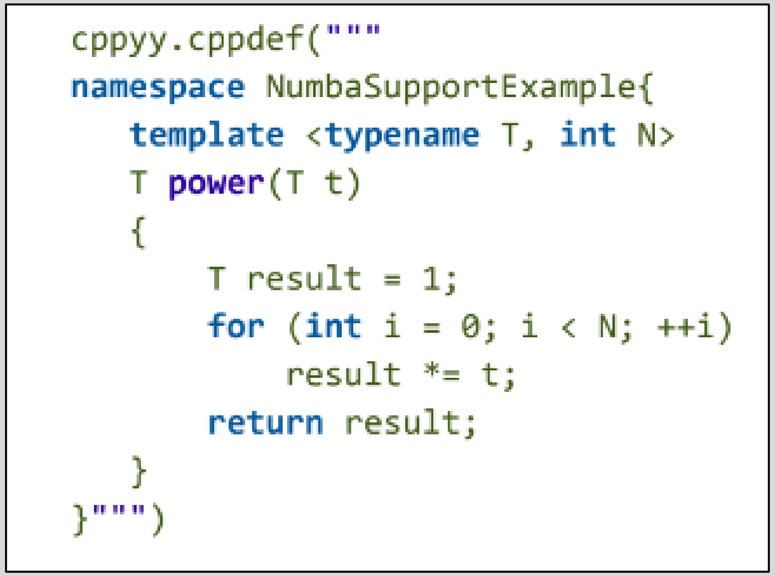
multiple template parameters

SOME EXAMPLES



```
cppyy.cppdef("""
namespace NumbaSupportExample{
   template <template <typename> class Container, typename T>
   T sum(const Container<T> &container)
   {
       T total = T(0);
       for (const T &value : container)
       {
           total += value;
       }
       return total;
   }
}""")
```

Template template parameters



```
cppyy.cppdef("""
namespace NumbaSupportExample{
   template <typename T, int N>
   T power(T t)
   {
       T result = 1;
       for (int i = 0; i < N; ++i)
           result *= t;
       return result;
   }
}""")
```

Non-type template parameters

Thank You!