CPPYY, mmap, ctypes, keystone, & running away from the compiler:

Compile the interpreter once, and bind instead

https://cppyy.readthedocs.io/en/latest/#

cppyy is an automatic, run-time, Python-C++ bindings generator, for calling C++ from Python and Python from C++. 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, and interactive exploration of C++ libraries. cppyy delivers this without any language extensions, intermediate languages, or the need for boiler-plate hand-written code. For design and performance, see this PyHPC'16 paper, albeit that the CPython/cppyy performance has been vastly improved since.

cppyy is based on Cling, the C++ interpreter, to match Python's dynamism, interactivity, and run-time behavior. Consider this session, showing dynamic, interactive, mixing of C++ and Python features (there are more examples throughout the documentation and in the tutorial):

So, what is the benefit of using something like this?

- Reduced compiling to a “one and done compiling the interpreter”
- Less bindings/wrappers
- Ability to use python with c++ project files
- A way to use c/c++ code without compiling it
- A way to bridge between other languages, c/c++, & python
- Direct access to c/c++ functions
- Direct access to python annotated “c/c++ functions” (lambdas or python functions
- A ground up approach to optimization that doesn’t rely on compiling a specific way

What do I need to do such myself?

- Python
- Cppyy
- c/c++/assembly code you’re planning to use
- Other needed files your project uses
- A c/c++ compiler you don’t mind removing after install
- Docker (maybe)

Example 1:

```python
import cppyy, numba
from ctypes import CFUNCTYPE, c_int

cppyy.cppdef("""
extern "C" {
    int (*m)(int);
}
int m2(int c){
    return m(c);
}
""")

m_data=0x1000

@CFUNCTYPE(c_int,c_int)
@numba.jit(nogil=True, nopython=True)
def add_int_py(i):
    return m_data + i

cppyy.gbl.m = add_int_py

print(cppyy.gbl.m2(-3))
```

```
(Re-)building pre-compiled headers (options: -O2 -march=native); this may take a minute ...
/home/runner/cppyy2numba2ctypesforreals/venv/lib/python3.8/site-packages/cppyy_backend/loader.py:139: UserWarning:
No precompiled header available (failed to build); this may impact performance.
  warnings.warn('No precompiled header available (%s); this may impact performance.' % msg)
4093
```

- Types are “matching”....
- Things can be assigned from python to c++?....?
- C++ interpreter in python
- Function & variable passing

Example 2: (https://code.activestate.com/recipes/579037-how-to-execute-x86-64-bit-assembly-code-directly-f/)

```python
import subprocess, os, tempfile
from ctypes import *

PAGE_SIZE = 4096

class AssemblerFunction(object):

  def __init__(self, code, ret_type, *arg_types):
    # Run Nasm
    fd, source = tempfile.mkstemp(".S", "assembly", os.getcwd())
    os.write(fd, code)
    os.close(fd)
    target = os.path.splitext(source)[0]
    subprocess.check_call(["nasm",source])
    os.unlink(source)
    binary = file(target,"rb").read()
    os.unlink(target)
    bin_len = len(binary)

    # align our code on page boundary.
    self.code_buffer = create_string_buffer(PAGE_SIZE*2+bin_len)
    addr = (addressof(self.code_buffer) + PAGE_SIZE) & (~(PAGE_SIZE-1))
    memmove(addr, binary, bin_len)

    # Change memory protection
    self.mprotect = cdll.LoadLibrary("libc.so.6").mprotect
    mp_ret = self.mprotect(addr, bin_len, 4)   # execute only.
    if mp_ret: raise OSError("Unable to change memory protection")

    self.func = CFUNCTYPE(ret_type, *arg_types)(addr)
    self.addr = addr
    self.bin_len = bin_len

  def __call__(self, *args):
    return self.func(*args)

  def __del__(self):
    # Revert memory protection
    if hasattr(self,"mprotect"):
      self.mprotect(self.addr, self.bin_len, 3)


if __name__ == "__main__":
  add_func = """ BITS 64
                 mov rax, rdi    ; Move the first parameter
                 add rax, rsi    ; add the second parameter
                 ret                 ; rax will be returned
                 """

  Add = AssemblerFunction(add_func, c_int, c_int, c_int)
  print Add(1, 2)
```

- Compile less, annotate & bind more
- A bit cluntier but still works
- Manual page & memory protection
- More potential for error
- Anonymous file creation on disk, thrice
- “Execute only”?
-

Example 3 (https://stackoverflow.com/questions/6040932/executing-assembler-code-with-python)

```python
import cppyy
from keystone.keystone import Ks
from keystone.keystone_const import KS_ARCH_X86, KS_MODE_64
from mmap import mmap, PAGESIZE, PROT_READ, PROT_WRITE, PROT_EXEC
from ctypes import c_int, c_void_p, addressof, CFUNCTYPE

def AssemblyFunction(i):
    ks=Ks(KS_ARCH_X86, KS_MODE_64)
    encoding, count = ks.asm(i)
    del ks, count, i
    return bytes(encoding)

cppyy.cppdef("""
extern "C" {
    int (*f)(int);
}

int main(void) {
    return f(42);
}
""")

buf = mmap(-1, PAGESIZE, prot=PROT_READ|PROT_WRITE|PROT_EXEC)

buf.write(AssemblyFunction(b"""
  mov eax, edi
  add eax, 1
  ret
"""))

cppyy.gbl.f=CFUNCTYPE(c_int, c_int)(addressof(c_void_p.from_buffer(buf)))
print(cppyy.gbl.main())
buf.close()
```

```
43

real    0m1.750s
user    0m0.777s
sys 0m0.107s
```

- Why would you compile something that you can interpret & execute?
- Why add the unneeded extra steps?
- What happened to “compile once and run anywhere”?
-

What else can we use it for besides using c++ & python?

- Why not the JVM? Embedding it in c/c++ & python?
- Why not include the JRE code so you can run it in python?

https://github.com/openjdk/jdk/blob/13158cb52db723be4932d815bdb0a17245259c84/src/java.base/share/native/libjli/java.c
https://github.com/openjdk/jdk/blob/13158cb52db723be4932d815bdb0a17245259c84/src/java.base/share/native/launcher/main.c

- Why not use a copy of the windows & linux kernels' source code with it?

https://github1s.com/zhuhuibeishadiao/ntoskrnl/blob/master/Init/initos.c
https://github1s.com/torvalds/linux