Tools to Bind to Python

Henry Schreiner

PyHEP 2018

This talk is interactive, and can be run in SWAN. If you want to run it manually, just download the repository: github.com/henryiii/pybindings_cc (https://github.com/henryiii/pybindings_cc).

Open in SWAN

(https://cern.ch/swanserver/cgi-bin/go?projurl=https://github.com/henryiii/pybindings_cc.git)

Either use the menu option `CELL -> Run All` or run all code cells in order (don't skip one!)

Focus

- What Python bindings do
- How Python bindings work
- What tools are available

Caveats

- Will cover C++ and C binding only
- Will not cover every tool available
- Will not cover `cppyy` in detail (but see Enric's talk)
- Python 2 is dying, long live Python 3!
    - but this talk is Py2 compatible also

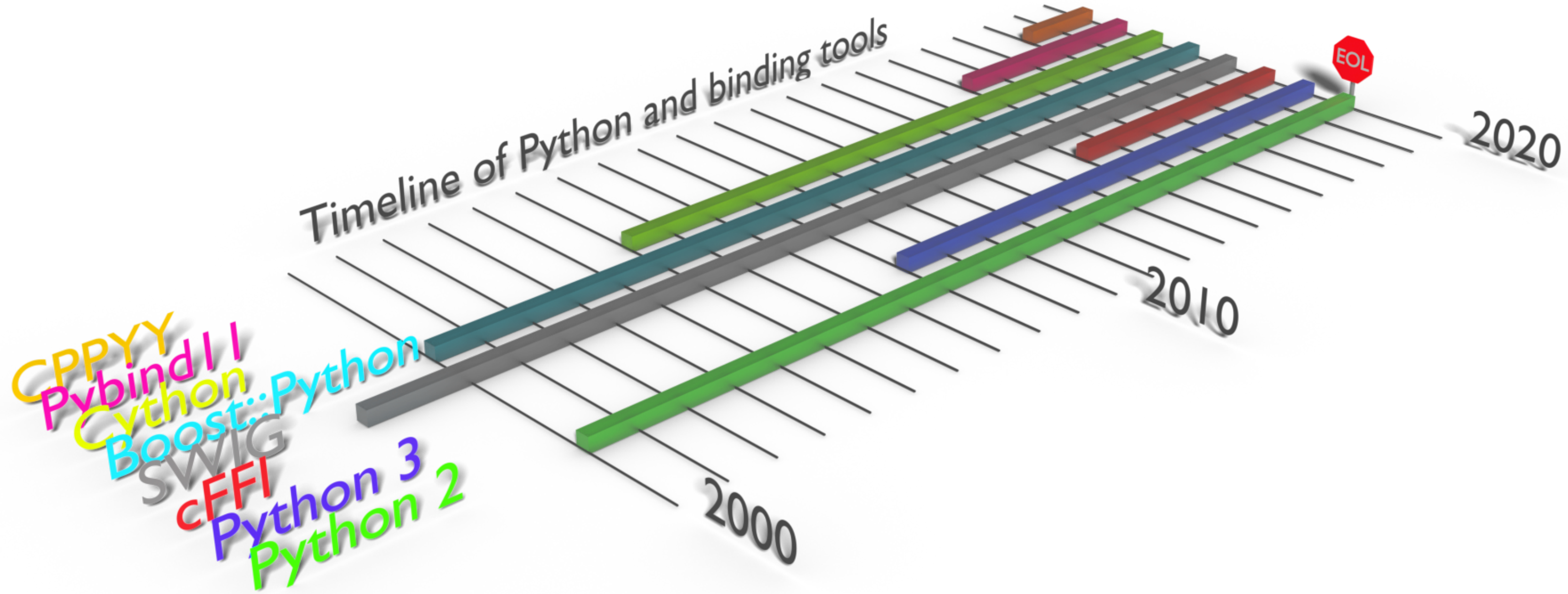
Overview:

Part one

- ctypes, CFFI : Pure Python, C only
- CPython: How all bindings work
- SWIG: Multi-language, automatic
- Cython: New language
- Pybind11: Pure C++11
- CPPYY: From ROOT's JIT engine

Part two

- An advanced binding in Pybind11


Timeline of Python and binding tools
EOL
2000
2010
2020
CPPYY
Pybind11
Cython
Boost::Python
SWIG
cFFI
Python 3
Python 2

Since this talk is an interactive notebook, *no code will be hidden*. Here are the required packages:

```
In [1]: !pip install --user cffi pybind11 numba
        # Other requirements: cython cppyy (SWIG is also needed but not a python module)
        # Using Anaconda recommended for users not using SWAN
```

```
Requirement already satisfied: cffi in /eos/user/h/hschrein/.local/lib/pytho
n3.6/site-packages
Requirement already satisfied: pybind11 in /eos/user/h/hschrein/.local/lib/p
ython3.6/site-packages
Requirement already satisfied: numba in /cvmfs/sft-nightlies.cern.ch/lcg/vie
ws/dev3python3/Wed/x86_64-slc6-gcc62-opt/lib/python3.6/site-packages
Requirement already satisfied: pycparser in /eos/user/h/hschrein/.local/lib/
python3.6/site-packages (from cffi)
Requirement already satisfied: llvmlite in /eos/user/h/hschrein/.local/lib/p
ython3.6/site-packages (from numba)
Requirement already satisfied: numpy in /cvmfs/sft-nightlies.cern.ch/lcg/vie
ws/dev3python3/Wed/x86_64-slc6-gcc62-opt/lib/python3.6/site-packages (from n
umba)
You are using pip version 9.0.3, however version 10.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

- Not on SWAN: cython, cppyy
- SWIG is also needed but not a python module
- Using Anaconda recommended for users not using SWAN

And, here are the standard imports. We will also add two variables to help with compiling:

In [2]:

```
from __future__ import print_function
import os
import sys
from pybind11 import get_include
inc = '-I ' + get_include(user=True) + ' -I ' + get_include(user=False)
plat = '-undefined dynamic_lookup' if 'darwin' in sys.platform else '-fPIC'
print('inc:', inc)
print('plat:', plat)
```

```
inc: -I /eos/user/h/hschrein/.local/include/python3.6m -I /cvmfs/sft-nightli
es.cern.ch/lcg/nightlies/dev3python3/Wed/Python/3.6.5/x86_64-slc6-gcc62-opt/
include/python3.6m
plat: -fPIC
```

What is meant by bindings?

Bindings allow a function(alitiy) in a library to be accessed from Python.

We will start with this example:

```
In [3]: %%writefile simple.c

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

```
Overwriting simple.c
```

Desired usage in Python:

```
y = square(x)
```

ctypes (https://docs.python.org/3.7/library/ctypes.html)

C bindings are very easy. Just compile into a shared library, then open it in python with the built in ctypes (https://docs.python.org/3.7/library/ctypes.html) module:

In [4]:

```
!cc simple.c -shared -o simple.so
```

In [5]:

```
from ctypes import cdll, c_float
lib = cdll.LoadLibrary('./simple.so')
lib.square.argtypes = (c_float,)
lib.square.restype = c_float
lib.square(2.0)
```

Out[5]:

```
4.0
```

- This may be all you need! Example: AmpGen (https://gitlab.cern.ch/lhcb/Gauss/blob/LHCBGAUSS-1058.AmpGenDev/Gen/AmpGen/options/ampgen.py) Python interface.
- In Pythonista (http://omz-software.com/pythonista/) for iOS, we can even use ctypes to access Apple's public APIs!

CFFI (http://cffi.readthedocs.io/en/latest/overview.html)

- The *C Foreign Function Interface* for Python
- Still C only
- Developed for PyPy, but available in CPython too

The same example as before:

```
In [6]:  from cffi import FFI
         ffi = FFI()
         ffi.cdef("float square(float);")
         C = ffi.dlopen('./simple.so')
         C.square(2.0)

Out[6]:  4.0
```

CPython (python.org)

- Let's see how bindings work before going into C++ binding tools
- This is how CPython itself is implemented

C reminder: `static` *means visible in this file only*

In [7]:

```
%%writefile pysimple.c
#include <Python.h>

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

static PyObject* square_wrapper(PyObject* self, PyObject* args) {
  float input, result;
  if (!PyArg_ParseTuple(args, "f", &input)) {return NULL;}
  result = square(input);
  return PyFloat_FromDouble(result);}

static PyMethodDef pysimple_methods[] = {
 { "square", square_wrapper, METH_VARARGS, "Square function" },
 { NULL, NULL, 0, NULL } };

#if PY_MAJOR_VERSION >= 3
static struct PyModuleDef pysimple_module = {
    PyModuleDef_HEAD_INIT, "pysimple", NULL, -1, pysimple_methods};
PyMODINIT_FUNC PyInit_pysimple(void) {
    return PyModule_Create(&pysimple_module); }
#else
DL_EXPORT(void) initpysimple(void) {
  Py_InitModule("pysimple", pysimple_methods); }
#endif
```

```
Overwriting pysimple.c
```

Build:

```
In [8]: !cc {inc} -shared -o pysimple.so pysimple.c {plat}
```

Run:

```
In [9]: import pysimple
        pysimple.square(2.0)

Out[9]: 4.0
```

C++: Why do we need more?

- Sometimes simple is enough!
- `export "C"` allows C++ backend
- C++ API can have: overloading, classes, memory management, etc...
- We could manually translate everything using C API

Solution:

C++ binding tools!

This is our C++ example:

In [10]:

```
%%writefile SimpleClass.hpp
#pragma once

class Simple {
    int x;
  public:
    Simple(int x): x(x) {}
    int get() const {return x;}
};
```

```
Overwriting SimpleClass.hpp
```

SIMPLIFIED WRAPPER AND INTERFACE GENERATOR

THE CURE FOR THE COMMON CODE

(swig.org)

- SWIG: Produces "automatic" bindings
- Works with many output languages
- Has supporting module built into CMake
- Very mature

Downsides:

- Can be all or nothing
- Hard to customize
- Customizations tend to be language specific
- Slow development

In [11]:

```
%%writefile SimpleSWIG.i

%module simpleswig
%{
/* Includes the header in the wrapper code */
#include "SimpleClass.hpp"
%}

/* Parse the header file to generate wrappers */
%include "SimpleClass.hpp"
```

```
Overwriting SimpleSWIG.i
```

In [12]:

```
!swig -swiglib
```

```
/build/jenkins/workspace/install/swig/3.0.12/x86_64-slc6-gcc62-opt/share/swi
g/3.0.12
```

SWAN/LxPlus only:

We need to fix the `SWIG_LIB` path if we are using LCG's version of SWIG (such as on SWAN)

In [13]:

```
if 'LCG_VIEW' in os.environ:
    swiglibold = !swig -swiglib
    swigloc = swiglibold[0].split('/')[-3:]
    swiglib = os.path.join(os.environ['LCG_VIEW'], *swigloc)
    os.environ['SWIG_LIB'] = swiglib
```

```
In [14]: !swig -python -c++ SimpleSWIG.i
```

```
In [15]: !c++ -shared SimpleSWIG_wrap.cxx {inc} -o _simpleswig.so {plat}
```

```
In [16]: import simpleswig
         x = simpleswig.Simple(2)
         x.get()
```

```
Out[16]: 2
```

Cython

(http://cython.org)

- Built to be a Python+C language for high performance computations
- Performance computation space in competition with Numba
- Due to design, also makes binding easy
- Easy to customize result
- Can write Python 2 or 3, regardless of calling language

Downsides:

- Requires learning a new(ish) language
- Have to think with three hats
- *Very* verbose

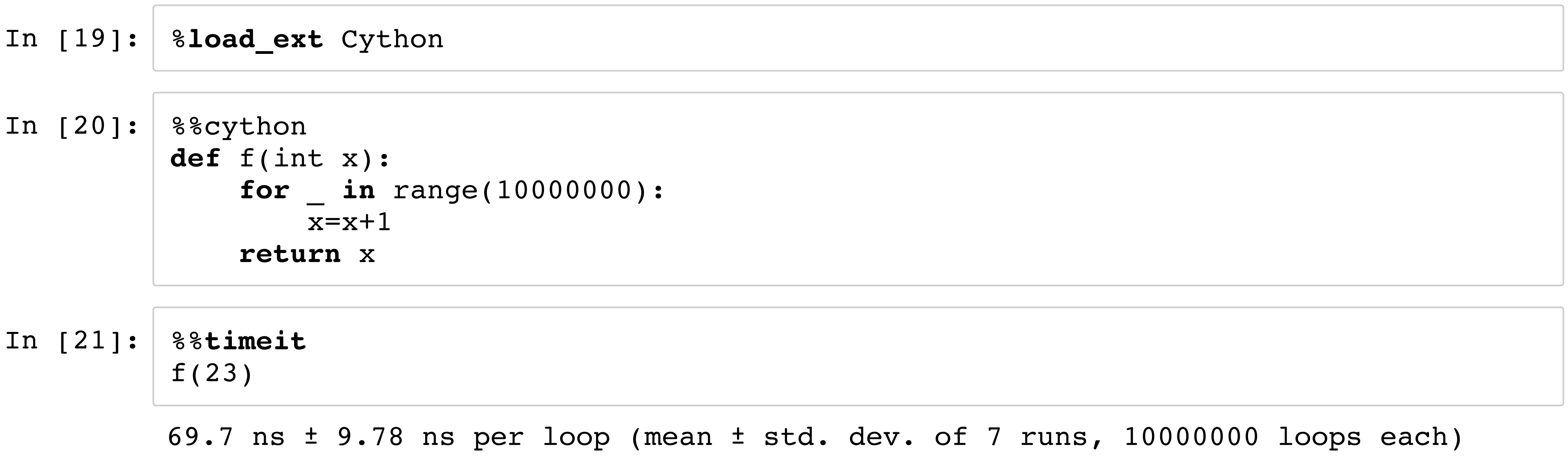
Aside: Speed comparison Python, Cython, Numba (https://numba.pydata.org)

```
In [17]: def f(x):
             for _ in range(100000000):
                 x=x+1
             return x
```

```
In [18]: %%time
         f(1)
```

```
CPU times: user 6.88 s, sys: 0 ns, total: 6.88 s
Wall time: 6.88 s
```

```
Out[18]: 100000001
```



In [19]:
```
%load_ext Cython
```

In [20]:
```
%%cython
def f(int x):
    for _ in range(10000000):
        x=x+1
    return x
```

In [21]:
```
%%timeit
f(23)
```

```
69.7 ns ± 9.78 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

In [22]:

```
import numba
@numba.jit
def f(x):
    for _ in range(10000000):
        x=x+1
    return x
```

In [23]:

```
%time
f(41)
```

```
CPU times: user 0 ns, sys: 11 µs, total: 11 µs
Wall time: 56.3 µs
```

Out[23]:

```
10000041
```

In [24]:

```
%%timeit
f(41)
```

```
268 ns ± 12.9 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Binding with Cython (https://cython.org)

In [25]:

```
%%writefile simpleclass.pxd
# distutils: language = c++

cdef extern from "SimpleClass.hpp":
    cdef cppclass Simple:
        Simple(int x)
        int get()
```

```
Overwriting simpleclass.pxd
```

In [26]:

```
%%writefile cythonclass.pyx
# distutils: language = c++

from simpleclass cimport Simple as cSimple

cdef class Simple:
    cdef cSimple *cself

    def __cinit__(self, int x):
        self.cself = new cSimple(x)

    def get(self):
        return self.cself.get()

    def __dealloc__(self):
        del self.cself
```

```
Overwriting cythonclass.pyx
```

In [27]:

```
!cythonize cythonclass.pyx
```

```
Compiling /eos/user/h/hschrein/SWAN_projects/pybindings_cc/cythonclass.pyx b
ecause it changed.
[1/1] Cythonizing /eos/user/h/hschrein/SWAN_projects/pybindings_cc/cythoncla
ss.pyx
```

In [28]:

```
!g++ cythonclass.cpp -shared {inc} -o cythonclass.so {plat}
```

In [29]:

```
import cythonclass
x = cythonclass.Simple(3)
x.get()
```

Out[29]:

```
3
```

pybind11

(http://pybind11.readthedocs.io/en/stable/)

- Similar to Boost::Python, but easier to build
- Pure C++11 (no new language required), no dependencies
- Builds remain simple and don't require preprocessing
- Easy to customize result
- Great Gitter community
- Used in GooFit 2.1+ (https://goofit.github.io) for CUDA too [CHEP talk] (https://indico.cern.ch/event/587955/contributions/2938087/)

Downsides:

In [30]:

```
%%writefile pybindclass.cpp

#include <pybind11/pybind11.h>
#include "SimpleClass.hpp"

namespace py = pybind11;

PYBIND11_MODULE(pybindclass, m) {
    py::class_<Simple>(m, "Simple")
        .def(py::init<int>())
        .def("get", &Simple::get)
    ;
}
```

```
Overwriting pybindclass.cpp
```

In [31]:

```
!c++ -std=c++11 pybindclass.cpp -shared {inc} -o pybindclass.so {plat}
```

In [32]:

```
import pybindclass
x = pybindclass.Simple(4)
x.get()
```

Out[32]:

```
4
```

CPPYY (http://cppyy.readthedocs.io/en/latest/)

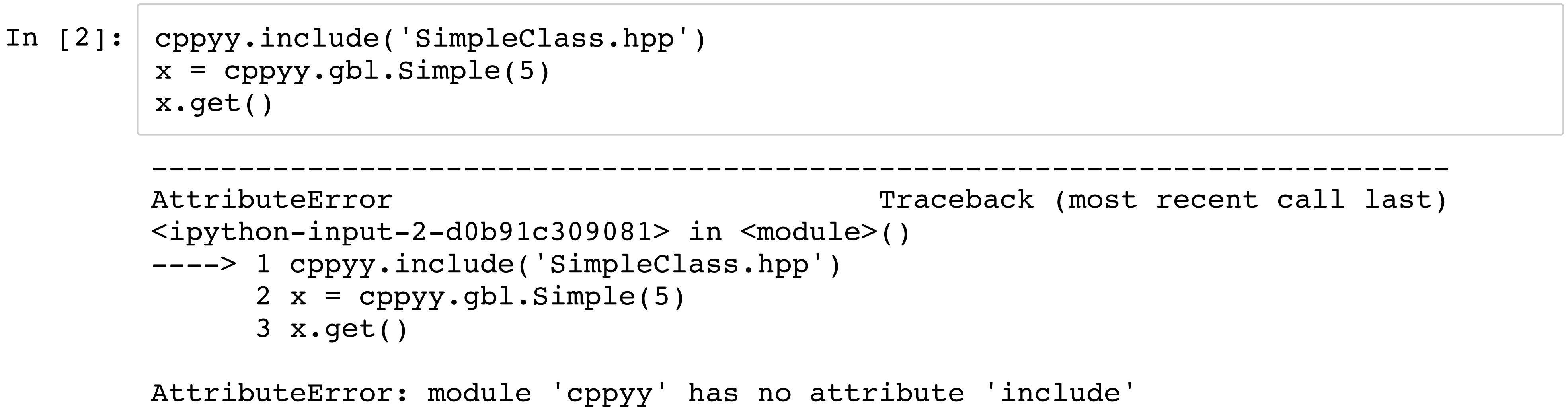
- Born from ROOT bindings
- Built on top of Cling
- JIT, so can handle templates
- See Enric's talk for more

Downsides:

- Header code runs in Cling
- Heavy *user* requirements (Cling)
- ROOT vs. pip version
- Broken on SWAN (so will not show working example here)

In [1]:

```
import cppyy
```



In [2]:

```
cppyy.include('SimpleClass.hpp')
x = cppyy.gbl.Simple(5)
x.get()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-2-d0b91c309081> in <module>()
----> 1 cppyy.include('SimpleClass.hpp')
      2 x = cppyy.gbl.Simple(5)
      3 x.get()

AttributeError: module 'cppyy' has no attribute 'include'
```

Continue to part 2