∼

Next steps with python

Scientific Computing Winter 2016/2017

Lecture 17

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

made wit pandoc

Python

- Scripting languages provide high flexibility and quick ways for parameter modification
- Python: de facto standard for scripting interfaces in scientific computing and glueing codes together
- But how to create an interface ?

Python and C++

- Architecture of python
  - Interpreter
  - Application programming interface (API) for interaction of C/C++ with python interpreter
    - register wrapper function with name
    - wrapper function knows how to fetch parameters/return values from/to python interpreter
    - wrapper function calls C++ function
  - Wrapper code with code to be wrapped linked to a “shared object” (UNIX), “dylib” (Mac), “DLL” (Windows)
  - Import of wrapper code makes it available in python
- Automatic tools for accessing API

SWIG

- Several possibilities
  - Cython (python dialect with possible inclusion of C)
  - pybind11 (C++11 classes for wrapping python)
  - **SWIG** (“classical tool” for wrapping interpreter)
- Simplified Wrapper and Interface Generator:
  - Tool to automatically create wrapper code from C++ style description
  - Create wrapper code in C++ which is linked together with library to be wrapped

Trying this out

```
$ cp -r /net/wir/numcxx/examples/part4 .
$ cd part 4
$ make
```

- mymodule.hxx: some code to be equipped with python interface
- mymodule.i: interface description for SWIG. This could be automatically generated from mymodule.hxx with some markup

```
swig -c++ -python  -o swigwrap_mymodule.cxx mymodule.i
```

- swigwrap_mymodule.cxx: C++ code generated (not for human consumption. . .)
- mymodule.py: accompanying python module

```
$(CXX)  $(PYTHON_LIBS) $(LINALG_LIBS) $(SHLDFLAGS) swigwrap_mymodule.cxx -o _mymodule.so
```

- _mymodule.so: shared object (dylib,dll) containing compiled code loaded into python

```
import mymodule
# this actually imports mymodule.py which loads _mymodule.so
```

dlopen, dlsym

```
SYNOPSIS
       #include <dlfcn.h>
       void *dlopen(const char *filename, int flags);
       Link with -ldl.
DESCRIPTION
   dlopen()

       The function  dlopen() loads the dynamic  shared object (shared
       library) file named by  the null-terminated string filename and
       returns an opaque "handle" for  the loaded object.  This handle
       is employed  with other  functions in the  dlopen API,  such as
       dlsym(3), dladdr(3), dlinfo(3), and dlclose().
```

```
SYNOPSIS
       #include <dlfcn.h>
       void *dlsym(void *handle, const char *symbol);
       Link with -ldl.
DESCRIPTION
    dlsym()

       The  function dlsym()  takes  a "handle"  of  a dynamic  loaded
       shared   object   returned   by    dlopen(3)   along   with   a
       null-terminated symbol name, and returns the address where that
       symbol is loaded  into memory.  If the symbol is  not found...
```

```c
void (*init)();

void *handle=dlopen("_mymodule.so");
init=dlsym(handle,"mymodule_init");
init();
```