

```
#include <iostream>
```

```
int main(){
```

```
    std::cout << "Hello World!\n";
```

```
    std::vector<int> myVec(10);
```

```
    std::iota(myVec.begin(), myVec.end(), 0);
```

```
    std::cout << "myVec: ";
```

```
    for ( auto i: myVec )
```

```
        std::cout << i << " ";
```

```
    std::function< bool(int)> myBindPred = bind( std::less<int>(),
```

```
        myVec.begin(), myVec.end() );
```

```
    std::cout << "myVec: ";
```

```
    for ( auto i: myVec ) std::cout << i << " ";
```

```
    std::cout << "\n\n";
```

```
    std::vector<int> myVec2(20);
```

```
    std::iota(myVec2.begin(), myVec2.end(), 0);
```

```
    std::cout << "myVec2: ";
```

```
    for ( auto i: myVec2 ) std::cout << i << " ";
```

Extend and Embed Python

Rainer Grimm

Training, Coaching, and
Technology Consulting

Extend and Embed

Extend

1. Convert the values from Python to C/C++.
2. Use the converted values to execute the C/C++ functionality.
3. Convert the results from C/C++ back to Python.

Embed

1. Convert the values from C/C++ to Python.
2. Use the converted values to execute the Python functionality.
3. Convert the results from Python back to C/C++.

Extend and Embed

Advantages

- Don't repeat yourself (DRY)
- Optimization of performance critical parts of the application
- Overcome the Global Interpreter Lock (GIL)

Extend and Embed

Extend Python

Embed Python

Extend Python

Shared Library

Ctypes

Native

SWIG

pybind11

Creating a Shared Library (Linux)

The shared library should consist of the following files.

helloWorld.h

```
#include <stdio.h>
```

```
void helloWorld();
```

helloWorld.c

```
#include "helloWorld.h"
```

```
void helloWorld() {  
    printf("Hello World\n");  
}
```

Creating a Shared Library (Linux)

Steps to create and use a shared library:

1. Generate position-independent code

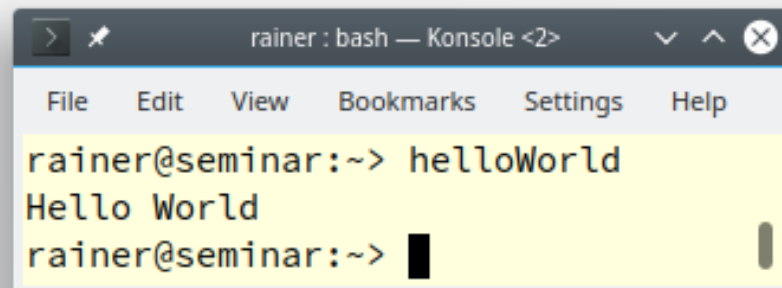
```
gcc -c -fpic helloWorld.c
```

2. Create the Shared Library

```
gcc -shared helloWorld.o -o libhelloWorld.so
```

3. Let the linker and runtime know the paths

```
gcc -L<PathToSharedLib> -Wl,-rpath=<PathToSharedLib>  
main.c -lhelloWorld -o helloWorld
```

A terminal window titled 'rainer : bash — Konsole <2>' with a menu bar (File, Edit, View, Bookmarks, Settings, Help). The terminal shows the command 'helloWorld' being executed, resulting in the output 'Hello World'. The prompt 'rainer@seminar:~>' is visible before and after the command.

```
rainer@seminar:~> helloWorld  
Hello World  
rainer@seminar:~> █
```

Extend Python

Shared Library

Ctypes

Native

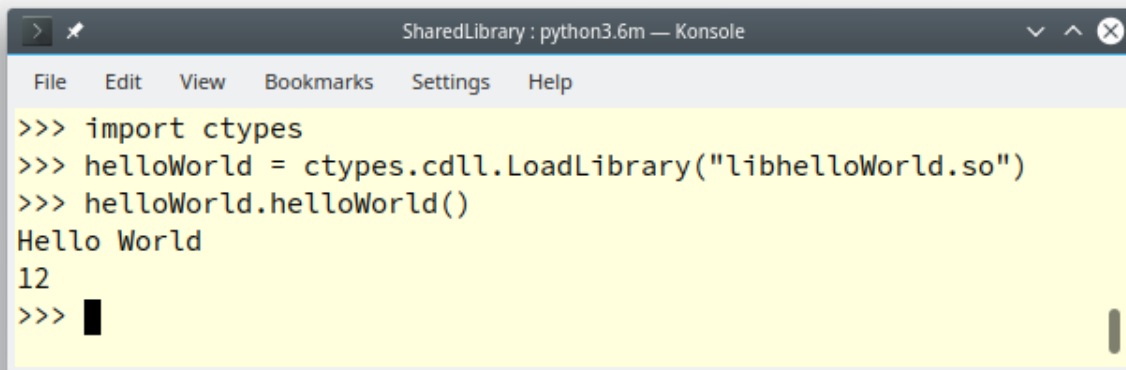
SWIG

pybind11

ctypes (Linux)

The library [ctypes](#) allows to call functions in shared libraries.

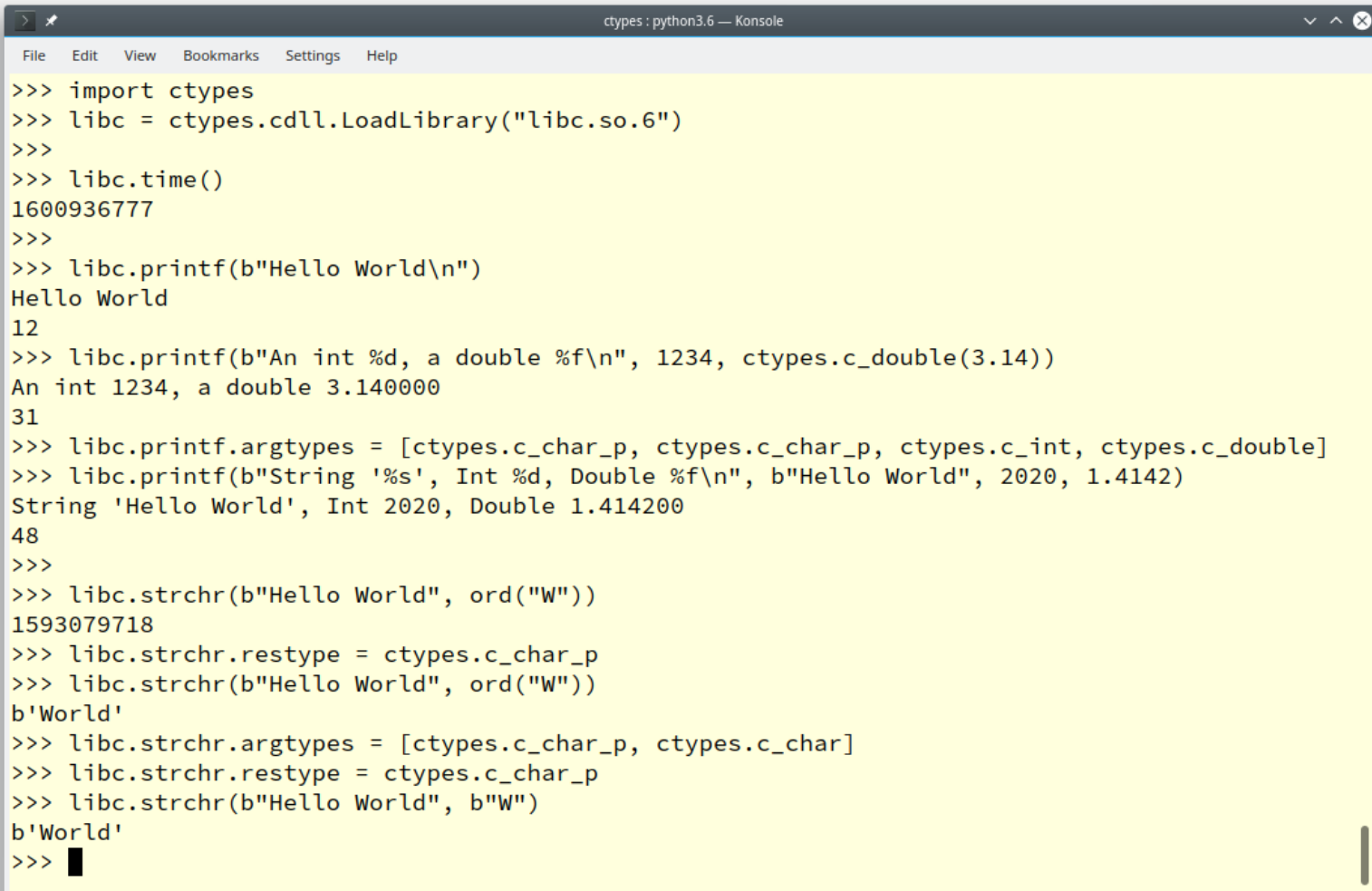
- Calling the shared library `libhelloWorld.so`.

A screenshot of a terminal window titled "SharedLibrary : python3.6m — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows a Python session where the ctypes module is imported, a shared library "libhelloWorld.so" is loaded using ctypes.cdll.LoadLibrary(), and the helloWorld function is called, resulting in the output "Hello World". The prompt "12" is visible below the output, and the session ends with a blank prompt line.

```
>>> import ctypes
>>> helloWorld = ctypes.cdll.LoadLibrary("libhelloWorld.so")
>>> helloWorld.helloWorld()
Hello World
12
>>> 
```

ctypes (Linux)

The library [ctypes](#) enables to use [libc](#).

A screenshot of a terminal window titled 'ctypes : python3.6 — Konsole'. The window has a menu bar with 'File', 'Edit', 'View', 'Bookmarks', 'Settings', and 'Help'. The terminal shows a series of Python commands using the ctypes module to load the libc library and call various functions like time(), printf(), and strchr(). The output of these calls is displayed in the terminal, including memory addresses, formatted strings, and character pointers. The terminal background is yellow, and the text is black. A vertical scrollbar is visible on the right side of the terminal window.

```
>>> import ctypes
>>> libc = ctypes.cdll.LoadLibrary("libc.so.6")
>>>
>>> libc.time()
1600936777
>>>
>>> libc.printf(b"Hello World\n")
Hello World
12
>>> libc.printf(b"An int %d, a double %f\n", 1234, ctypes.c_double(3.14))
An int 1234, a double 3.140000
31
>>> libc.printf.argtypes = [ctypes.c_char_p, ctypes.c_char_p, ctypes.c_int, ctypes.c_double]
>>> libc.printf(b"String '%s', Int %d, Double %f\n", b"Hello World", 2020, 1.4142)
String 'Hello World', Int 2020, Double 1.414200
48
>>>
>>> libc.strchr(b"Hello World", ord("W"))
1593079718
>>> libc.strchr.restype = ctypes.c_char_p
>>> libc.strchr(b"Hello World", ord("W"))
b'World'
>>> libc.strchr.argtypes = [ctypes.c_char_p, ctypes.c_char]
>>> libc.strchr.restype = ctypes.c_char_p
>>> libc.strchr(b"Hello World", b"W")
b'World'
>>> █
```

Extend Python

Shared Library

Ctypes

Native

SWIG

pybind11

Native

Extend Python with the `helloWorld.c` functionality.



The `helloWorld.h`, **and** `helloWorld.h` files are used to create an extension module (shared library).

Native

- Implementing the extension module "helloWorld"

```
1  #include <Python.h>
2
3  static PyObject* method_helloWorld(PyObject*, PyObject*);
4
5 > static PyMethodDef HelloWorld[] = { ...
9
10 > static struct PyModuleDef helloWorldModule = { ...
17
18 > static PyObject* method_helloWorld(PyObject* self, PyObject* args) { ...
26
27 > PyMODINIT_FUNC PyInit_helloWorld(void) { ...
```

- Accessing the Python API (1)
- Declaration of the C function (3)
- Definition of the method table (5)
- Definition of the module (10)
- Definition of the C function (18)
- Initialization of the module (27)

Native

- Access to the Python API thanks to `<Python.h>`.
 - `<Python.h>`
 - must be the first header file.
 - contains the header files `<stdio.h>`, `<string.h>`, `<errno.h>` and `<stdlib.h>`.
- All visible symbols start with `Py` or `PY`

Native

- Definition method table

```
static PyMethodDef HelloWorld[] = {  
    {"helloWorld", method_helloWorld, METH_VARARGS, "Hello"},  
    ...  
    {ZERO, ZERO, 0, ZERO}  
};
```

- "helloWorld" : name of the Python method
- method_helloWorld: name of the C function
- METH_VARARGS: calling convention for C function
- "Hello": documentation string
- ...: other methods
- {NULL, NULL, 0, NULL}: Sentinel

Native

- Definition of the module

```
static struct PyModuleDef helloWorldModule = {  
    PyModuleDef_HEAD_INIT,  
    "helloWorld",  
    "Hello World message",  
    -1,  
    HelloWorld  
};
```

- "helloWorld": name of the module
- "Hello World message": documentation of the module
- -1: size of the interpreter state (state is stored in a global variable).
- HelloWorld: names of the method table

Native

- Definition of the c function

```
static PyObject* method_helloWorld(PyObject* self, PyObject* args) {  
  
    printf("Hello World\n");  
    Py_INCREF(Py_None);  
    return Py_None;  
}  
  
}
```

- "PyObject": basic Python type that contains the reference counter and the pointer to the type
- "self": module
- "args": Python function arguments
- Py_None: Python's None type

Native

- Initialization of the module

```
PyMODINIT_FUNC PyInit_helloWorld(void) {  
    return PyModule_Create(&helloWorldModule);  
}
```

- `PyMODINIT_FUNC`: returns a `PyObject*`.
- `PyInit_helloWorld`: initialization function
 - `PyInit_<name of the module>`
 - is called automatically when loading the module
- `PyModule_Create(&helloWorldModule)`
 - creates the new module
 - returns it to the caller

Native

Creation of the module with the Python module [disutils](#)

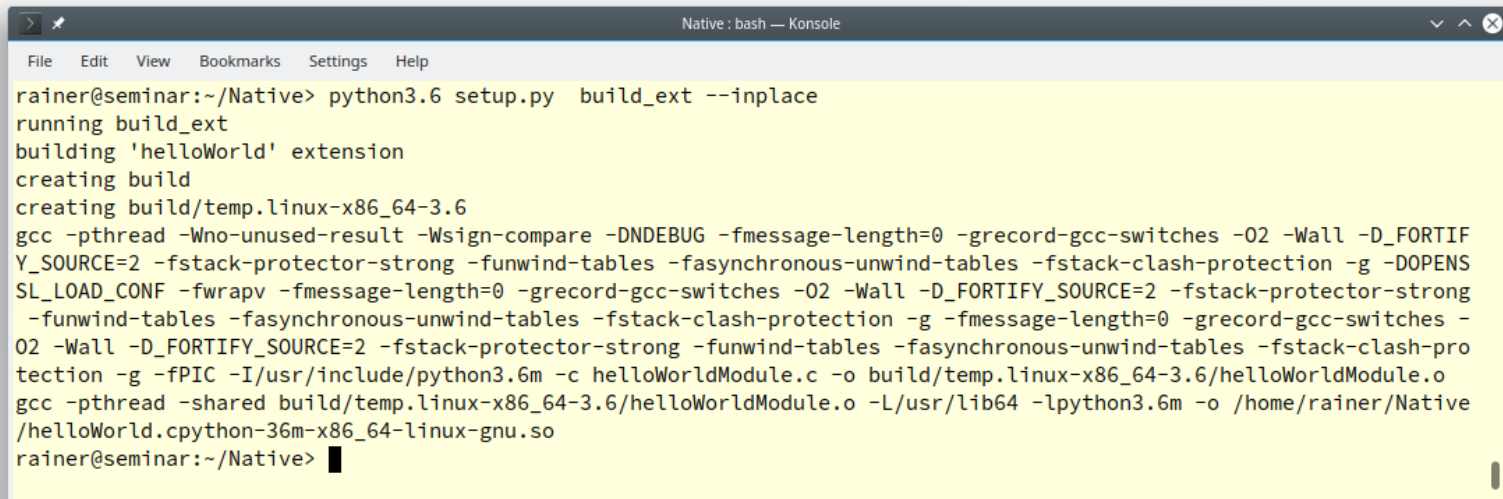
```
from distutils.core import setup, Extension

def main():
    setup(name = "helloWorld",
          version = "1.0.0",
          description = "Python extension to the hello world C-function.",
          author = "Rainer Grimm",
          author_email = "schulung@ModernesCpp.de",
          ext_modules=[Extension("helloWorld", ["helloWorldModule.c"])]))

if __name__ == "__main__":
    main()
```

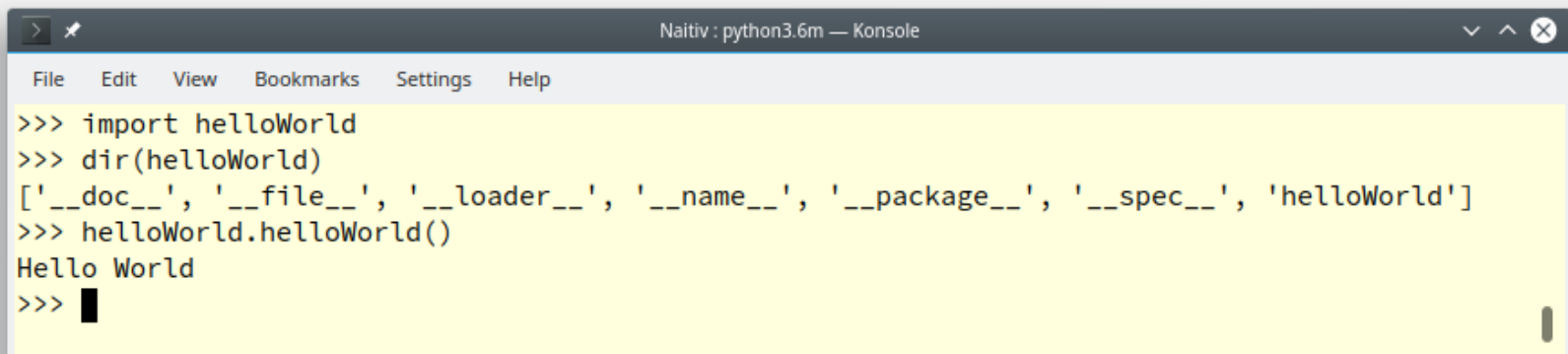
Native

- Building the expansion module



```
Native : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~/Native> python3.6 setup.py build_ext --inplace
running build_ext
building 'helloWorld' extension
creating build
creating build/temp.linux-x86_64-3.6
gcc -pthread -Wno-unused-result -Wsign-compare -DNDEBUG -fmessage-length=0 -grecord-gcc-switches -O2 -Wall -D_FORTIFY_SOURCE=2 -fstack-protector-strong -funwind-tables -fasynchronous-unwind-tables -fstack-clash-protection -g -DOPENSSL_LOAD_CONF -fwrapv -fmessage-length=0 -grecord-gcc-switches -O2 -Wall -D_FORTIFY_SOURCE=2 -fstack-protector-strong -funwind-tables -fasynchronous-unwind-tables -fstack-clash-protection -g -fmessage-length=0 -grecord-gcc-switches -O2 -Wall -D_FORTIFY_SOURCE=2 -fstack-protector-strong -funwind-tables -fasynchronous-unwind-tables -fstack-clash-protection -g -fPIC -I/usr/include/python3.6m -c helloWorldModule.c -o build/temp.linux-x86_64-3.6/helloWorldModule.o
gcc -pthread -shared build/temp.linux-x86_64-3.6/helloWorldModule.o -L/usr/lib64 -lpython3.6m -o /home/rainer/Native/helloWorld.cpython-36m-x86_64-linux-gnu.so
rainer@seminar:~/Native>
```

- Using the expansion module



```
Nativ : python3.6m — Konsole
File Edit View Bookmarks Settings Help
>>> import helloWorld
>>> dir(helloWorld)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'helloWorld']
>>> helloWorld.helloWorld()
Hello World
>>>
```

Extend Python

Shared Library

Ctypes

Native

SWIG

pybind11

SWIG

SWIG (**S**implified **W**rapper and **I**nterface **G**enerator) generates interfaces so that C/C++ can interact with other programming languages.

SWIG

- supports C99 and C++98 to C++17.
- can create wrappers for the following programming languages:
 - C#
 - D
 - Java
 - Javascript
 - Perl
 - Python
 - PHP
 - Ruby

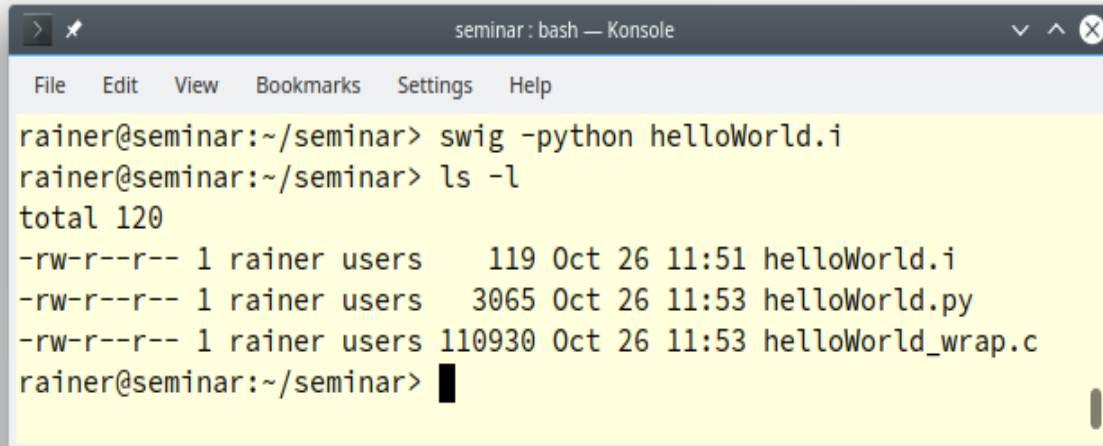
SWIG

- Interface definition

```
/* hello.i */  
  
%module helloWorld  
{  
#include "helloWorld.h"  
}  
  
external void helloWorld();
```

SWIG

- Creating the wrappers for Python

A terminal window titled 'seminar: bash — Konsole' with a menu bar (File, Edit, View, Bookmarks, Settings, Help). The terminal shows the following commands and output:

```
rainer@seminar:~/seminar> swig -python helloWorld.i
rainer@seminar:~/seminar> ls -l
total 120
-rw-r--r-- 1 rainer users 119 Oct 26 11:51 helloWorld.i
-rw-r--r-- 1 rainer users 3065 Oct 26 11:53 helloWorld.py
-rw-r--r-- 1 rainer users 110930 Oct 26 11:53 helloWorld_wrap.c
rainer@seminar:~/seminar>
```

- `helloWorld_wrap.c`
 - Low-level wrapper that must be linked to the rest of the application
- `helloWorld.py`
 - High-level code imported into Python

SWIG

- Implementation of the C functionality

- helloWorld.h

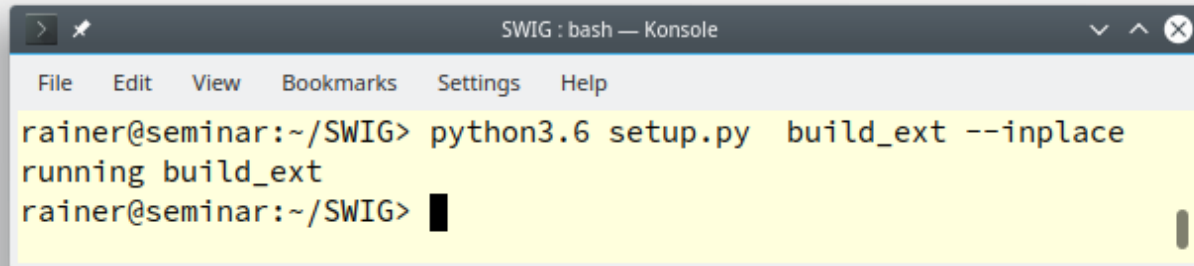
```
1  #include <stdio.h>
2
3  void helloWorld();
```

- helloWorld.c

```
1  #include "helloWorld.h"
2
3  void helloWorld() {
4      printf("Hello World\n");
5  }
```

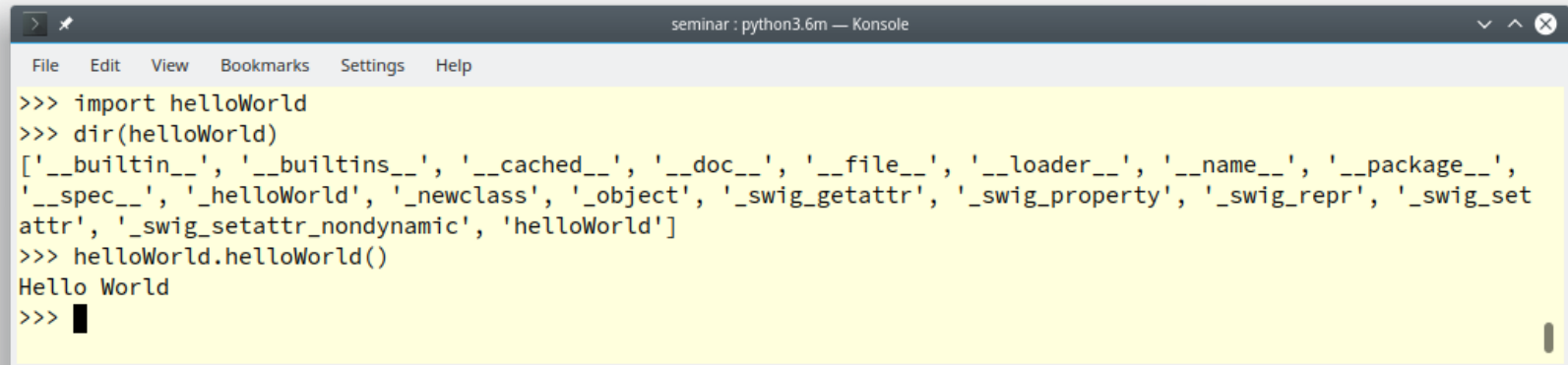
SWIG

- Building the expansion module



```
SWIG : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~/SWIG> python3.6 setup.py build_ext --inplace
running build_ext
rainer@seminar:~/SWIG>
```

- Using the extension module



```
seminar : python3.6m — Konsole
File Edit View Bookmarks Settings Help
>>> import helloWorld
>>> dir(helloWorld)
['__builtin__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__',
 '__spec__', '_helloWorld', '_newclass', '_object', '_swig_getattr', '_swig_property', '_swig_repr', '_swig_set
attr', '_swig_setattr_nondynamic', 'helloWorld']
>>> helloWorld.helloWorld()
Hello World
>>>
```

Extend Python

Shared Library

Ctypes

Native

SWIG

pybind11

pybind11

[pybind11](#) - Seamless operability between C++11 and Python

- Is fully implemented in header files
- Based on [Boost.Python](#)
- C++ data types can be used (extended) in Python
- Python data types can be used (embedded) in C++

pybind11

- Core feature
 - Lambda expressions
 - Functions
 - Accept arguments by value, reference, or pointers
 - Overload
 - Classes
 - Methods and attributes
 - Single and multiple inheritance
 - Virtuality
 - Library
 - STL
 - Smart pointer

pybind11

```
1 #include <pybind11/pybind11.h>
2
3 int add(int i, int j) {
4     return i + j;
5 }
6
7 PYBIND11_MODULE(function, m) {
8     m.def("add", &add, "A function which adds two numbers");
9 }
```

- `#include <pybind11/pybind11.h>`: C++11/Python binding
- `PYBIND11_MODULE`: called by import
- `function`: Name of the module
- `m`: variable of type `py::module_`
- `m.def`: makes the function known to Python

pybind11

- Convention

```
namespace py = pybind11;
```

- Functions

- Keyword arguments

```
m.def("add", &add, "A function which adds two numbers",  
      py::arg("i"), py::arg("j"));
```

- Default arguments

```
m.def("add", &add, "A function which adds two numbers",  
      py::arg("i") = 2000 , py::arg("j") = 11 );
```

pybind11

- Functions

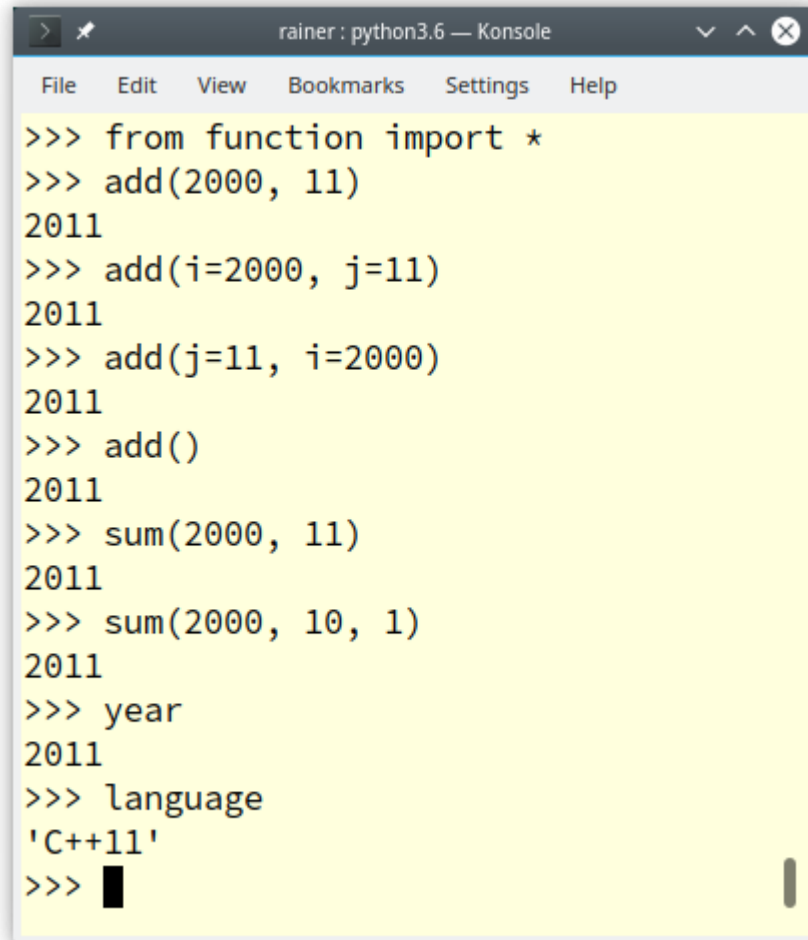
- Overload

```
m.def("sum", py::overload_cast<int, int>(&sum),  
      "Sum up two values");  
m.def("sum", py::overload_cast<int, int, int>(&sum),  
      "Sum up three values");
```

- Variables

```
m.attr("year") = 2011;  
m.attr("language") = "C++11";
```

pybind11

A screenshot of a terminal window titled "rainer : python3.6 — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows a series of Python commands and their outputs. The commands are: "from function import *", "add(2000, 11)", "add(i=2000, j=11)", "add(j=11, i=2000)", "add()", "sum(2000, 11)", "sum(2000, 10, 1)", "year", and "language". The outputs are: "2011", "2011", "2011", "2011", "2011", "2011", "2011", "2011", and "'C++11'". The cursor is at the end of the last line.

```
> ✂ rainer : python3.6 — Konsole
File Edit View Bookmarks Settings Help
>>> from function import *
>>> add(2000, 11)
2011
>>> add(i=2000, j=11)
2011
>>> add(j=11, i=2000)
2011
>>> add()
2011
>>> sum(2000, 11)
2011
>>> sum(2000, 10, 1)
2011
>>> year
2011
>>> language
'C++11'
>>> █
```

pybind11

- Object orientation

```
1 #include <pybind11/pybind11.h>
2 #include <string>
3
4 struct HumanBeing {
5     HumanBeing(const std::string& n) : name(n) { }
6     const std::string& getName() const { return name; }
7     std::string name;
8 };
9
10 namespace py = pybind11;
11
12 PYBIND11_MODULE(human, m) {
13     py::class_<HumanBeing>(m, "HumanBeing")
14         .def(py::init<const std::string &>())
15         .def("getName", &HumanBeing::getName);
16 }
```

- `class_`: creates a class
- `py::init`: requires the parameters of the constructor as template arguments

pybind11

- Special methods

```
def("__repr__", [](const HumanBeing& h) {  
    return "HumanBeing: " + h.name;  
}))
```

- Attributes

```
def_readwrite("familyName", &HumanBeing::familyName);
```

- Inheritance

```
py::class_<HumanBeing>(m, "HumanBeing")  
    .def(py::init<const std::string &>());  
py::class_<Woman, HumanBeing>(m, "Woman")  
    .def(py::init<const std::string &>())
```

pybind11

```
rainer: python3.6 — Konsole
File Edit View Bookmarks Settings Help
>>> from human import *
>>> bea = Woman("Beatrix")
>>> bea
HumanBeing: Beatrix
>>> dir(bea)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'familyName', 'gender', 'getName']
>>> bea.familyName
'Grimm'
>>> bea.getName()
'Beatrix'
>>> bea.gender
<bound method PyCapsule.gender of HumanBeing: Beatrix>
>>> bea.gender()
'female'
>>> print(bea)
Grimm Beatrix
>>> █
```

Extend and Embed

Extend Python

Embed Python

Execute a String directly

Execute a string

Run modules

Execute functions

Execute a String

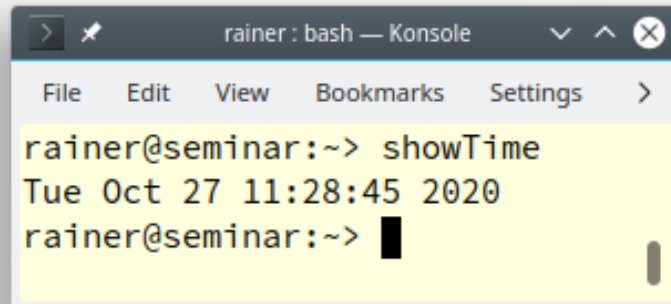
- Implementation of the C program

```
1 #include <Python.h>
2
3 int main(int argc, char* argv[]) {
4
5     Py_Initialize();
6     PyRun_SimpleString("import time\n"
7                        "print(time.ctime(time.time()))");
8     Py_Finalize();
9
10 }
```

- Initializes Python interpreter (5)
- Runs Python source code (6)
- Shuts down the interpreter (8)

Execute a String

- Running the program



A screenshot of a terminal window titled "rainer : bash — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", and "Settings". The terminal content shows the user "rainer" at host "seminar" in the directory "~" running the command "showTime". The output is "Tue Oct 27 11:28:45 2020". The prompt "rainer@seminar:~>" is followed by a black cursor block.

```
rainer@seminar:~> showTime
Tue Oct 27 11:28:45 2020
rainer@seminar:~> █
```

Execute a Module

Execute a string

Run a module

Execute a function

Run a Module

The module `showTime.py`

```
import time
```

```
print(time.ctime(time.time()))
```

Run Module

- Implementation of the C program

```
1 #include <Python.h>
2 #include <stdio.h>
3
4 int main(int argc, char* argv[]) {
5
6     Py_Initialize();
7     FILE* pyFile = fopen("showTime.py", "r");
8     if (pyFile) {
9         PyRun_SimpleFile(pyFile, "showTime.py");
10        fclose(pyFile);
11    }
12    Py_Finalize();
13 }
```

- Initializes Python interpreter (6)
- Runs Python source code (9)
- Shuts down the interpreter (12)

Execute a String

Execute a string

Run a module

Execute functions

Execute Functions

The myMath.py module

```
print("Python", end = "")
```

```
def fakul(num):  
    from functools import reduce  
    print("Returning fakul({})".format(num))  
    return reduce(lambda x, y: x * y, range(1, num + 1 ))
```

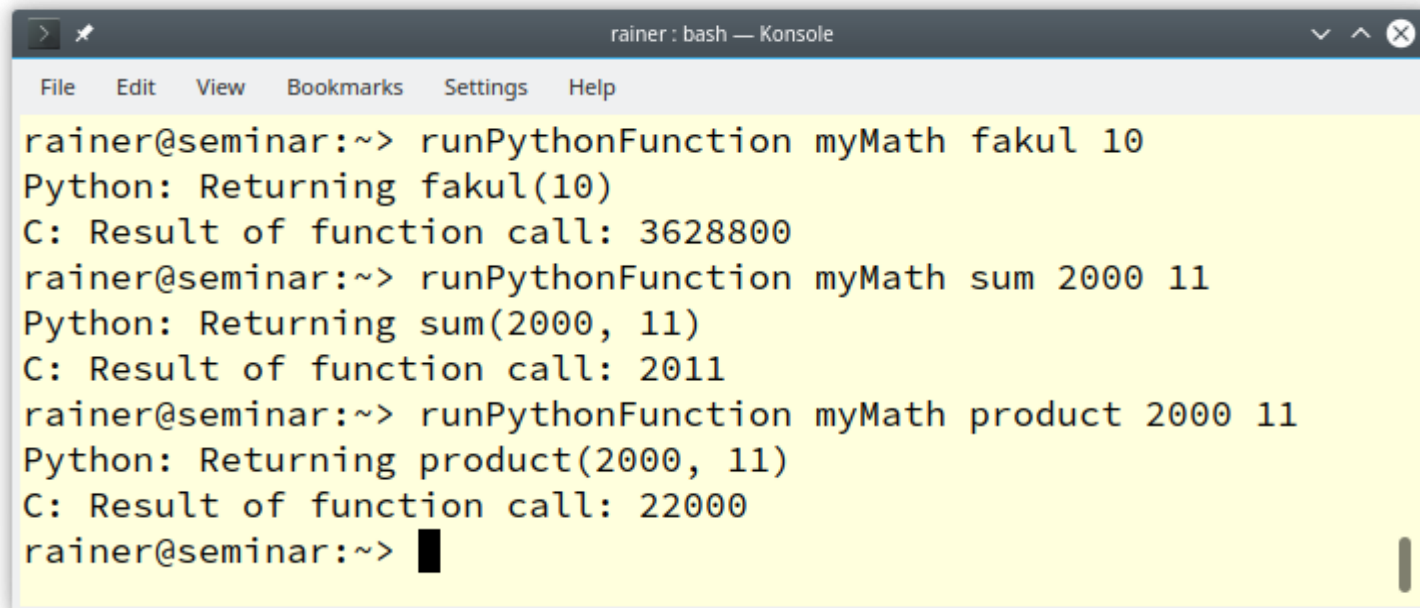
```
def sum(fir, sec):  
    print("Returning sum({}, {})".format(fir, sec))  
    return fir + sec
```

```
def product(fir, sec):  
    print("Returning product({}, {})".format(fir, sec))  
    return fir * sec
```

Execute Functions

The C program `runPythonFunction.c` allows to execute a function of a Python module.

`runPythonFunction` module function arguments



```
rainer : bash — Konsole
File Edit View Bookmarks Settings Help
rainer@seminar:~> runPythonFunction myMath fakul 10
Python: Returning fakul(10)
C: Result of function call: 3628800
rainer@seminar:~> runPythonFunction myMath sum 2000 11
Python: Returning sum(2000, 11)
C: Result of function call: 2011
rainer@seminar:~> runPythonFunction myMath product 2000 11
Python: Returning product(2000, 11)
C: Result of function call: 22000
rainer@seminar:~> █
```

Execute Functions

The following steps are performed by the `runPythonFunction.c` file.

- Read the command line
- Extend `sys.path` by the local directory
- Import the Python module
- Parse the function arguments
- Call the Python function
- Use the result of the Python function in C

Execute Functions

- Extend `sys.path` by the local directory

```
PyObject* sysmodule = PyImport_ImportModule("sys");  
PyObject* syspath = PyObject_GetAttrString(sysmodule, "path");  
PyList_Append(syspath, PyUnicode_FromString("."));
```

- Import the Python module

```
pName = PyUnicode_DecodeFSDefault(argv[1]);  
pModule = PyImport_Import(pName);
```

Execute Functions

- Parse the function arguments

```
pFunc = PyObject_GetAttrString(pModule, argv[2]);  
pArgs = PyTuple_New(argc - 3);  
for (i = 0; i < argc - 3; ++i) {  
    pValue = PyLong_FromLong(atoi(argv[i + 3]));  
    PyTuple_SetItem(pArgs, i, pValue);  
}
```

- Call the Python function

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

- Use the result of the Python function in C

```
printf("C: Result of function call: %ld\n", PyLong_AsLong(pValue));
```

Extend and Embed

Extend Python

Embed Python

Further Information

- [Python meets C/C++, Part 1: Extending Python with or embedding C/C++ in it](#)
- [Python meets C/C++, part 2: SWIG and pybind11](#)