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

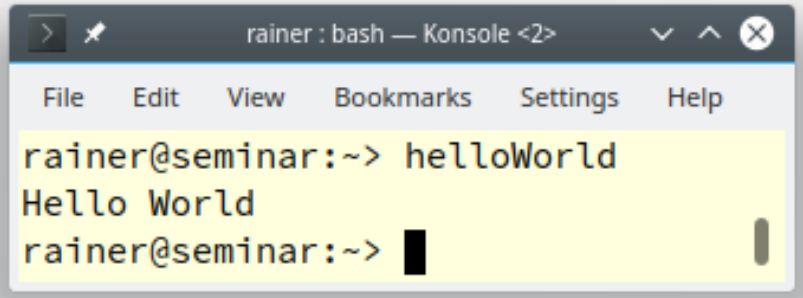
```
gcc -c -fpic helloWorld.c
```

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
```



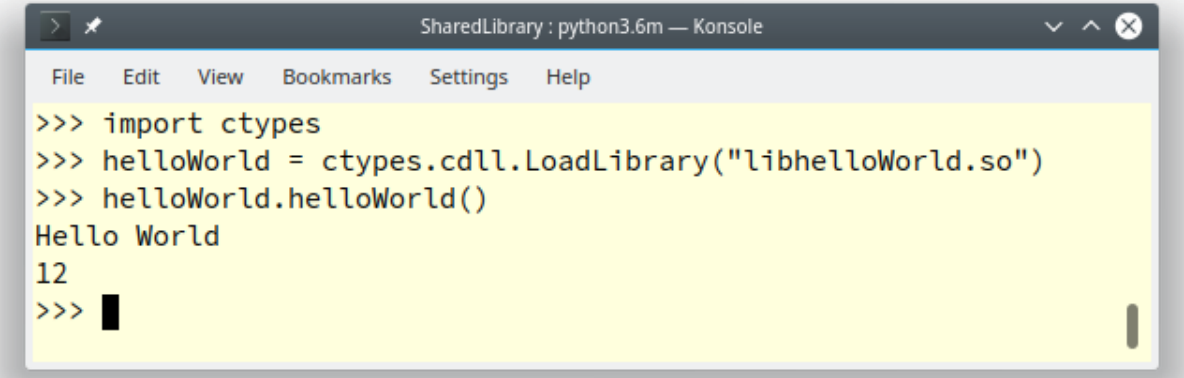
Extend Python

- Shared Library
- Ctypes
- Native
- SWIG
- pybind11

`ctypes` (Linux)

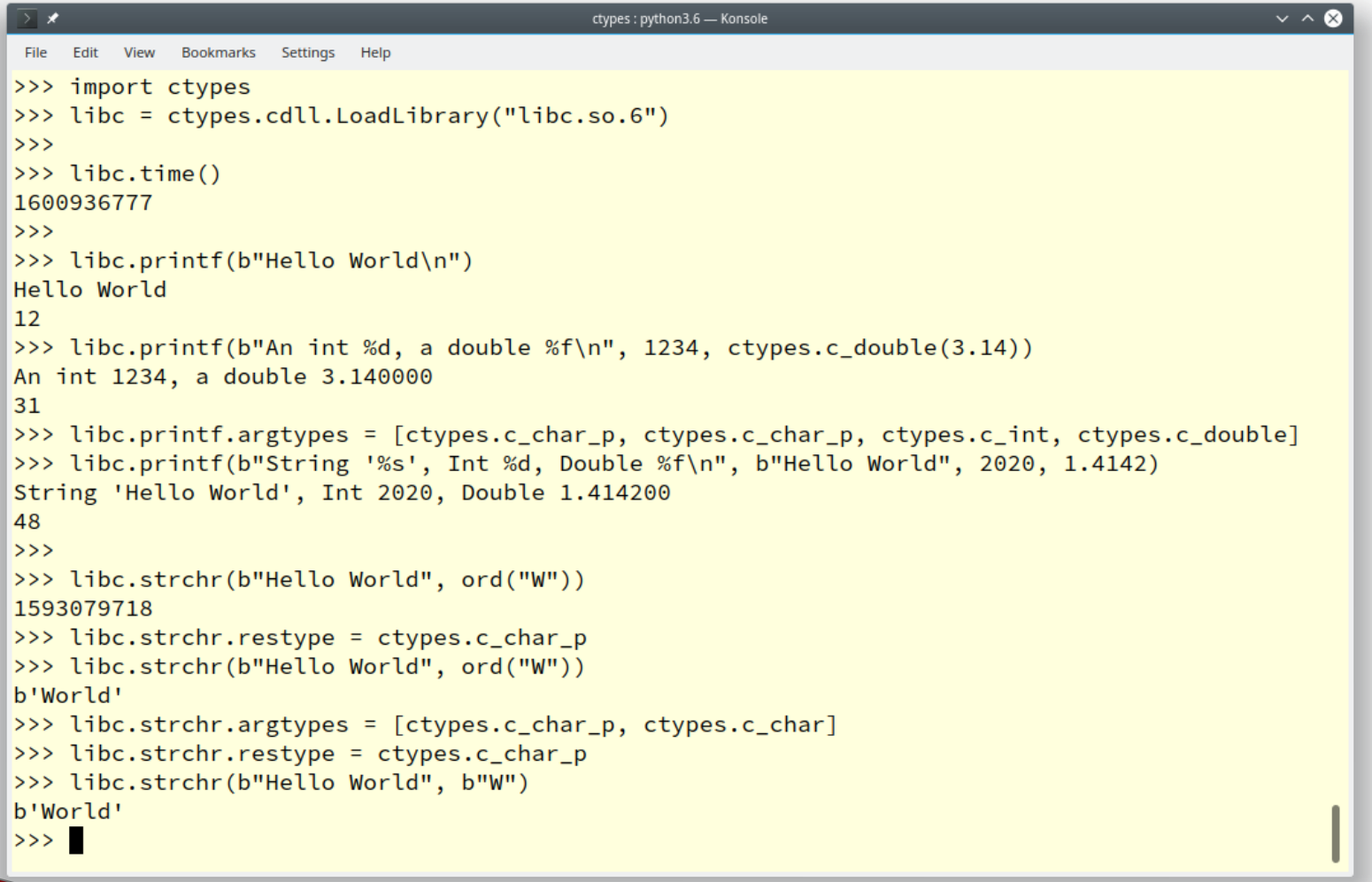
The library `ctypes` allows to call functions in shared libraries.

- Calling the shared library `libhelloWorld.so`.



ctypes (Linux)

The library ctypes enables to use libc.



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_helloword`: 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

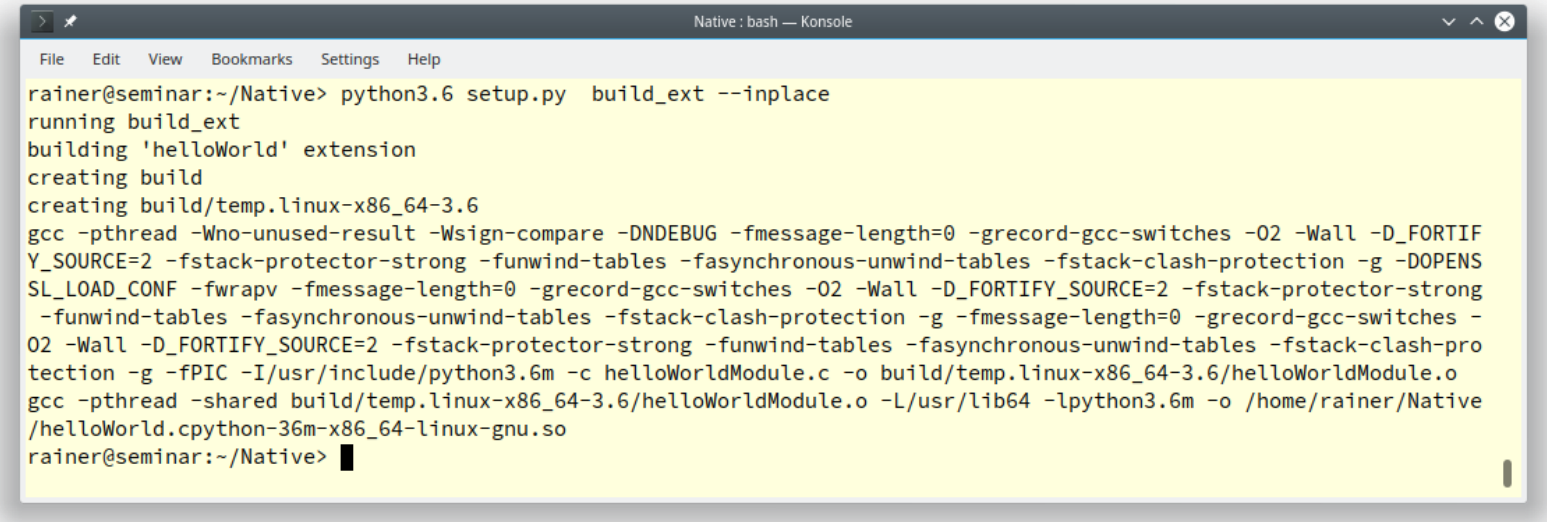
Native

Creation of the module with the Python module disutils

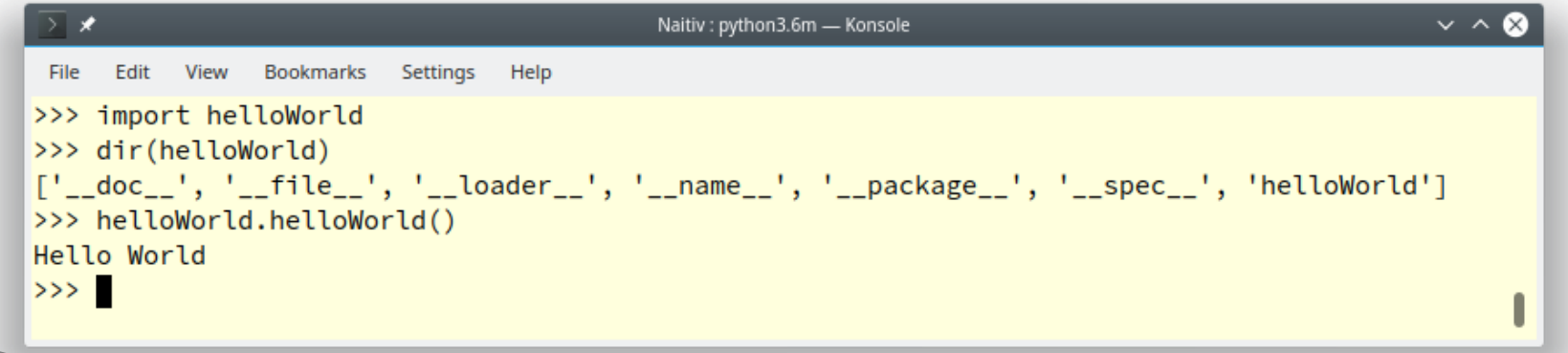
```
from distutils.core import setup, Extension

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



- Using the expansion module



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

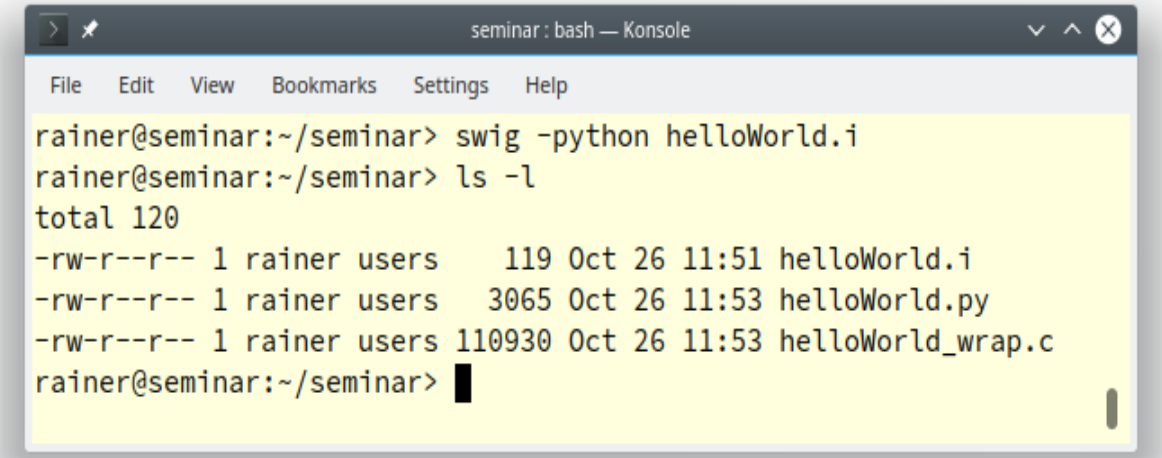
SWIG

- Interface definition

```
/* hello.i */

%module helloWorld
%{
#include "helloWorld.h"
%}

external void helloWorld();
```

SWIG

- Creating the wrappers for Python



  - `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`

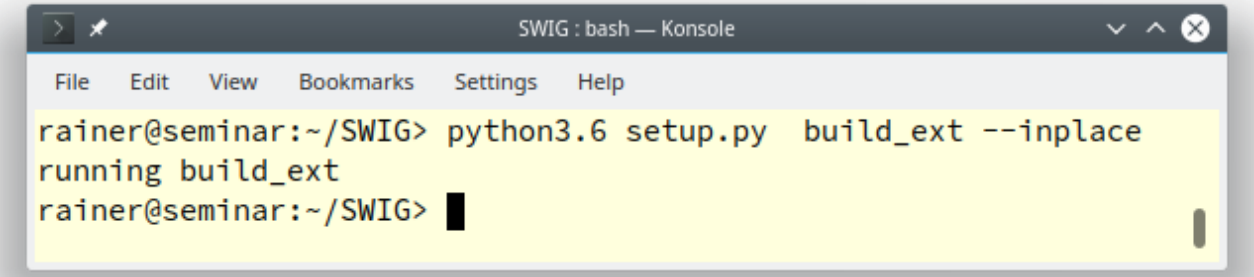
```
1   #include <stdio.h>
2
3   void helloWorld();
```

  - `helloWorld.c`

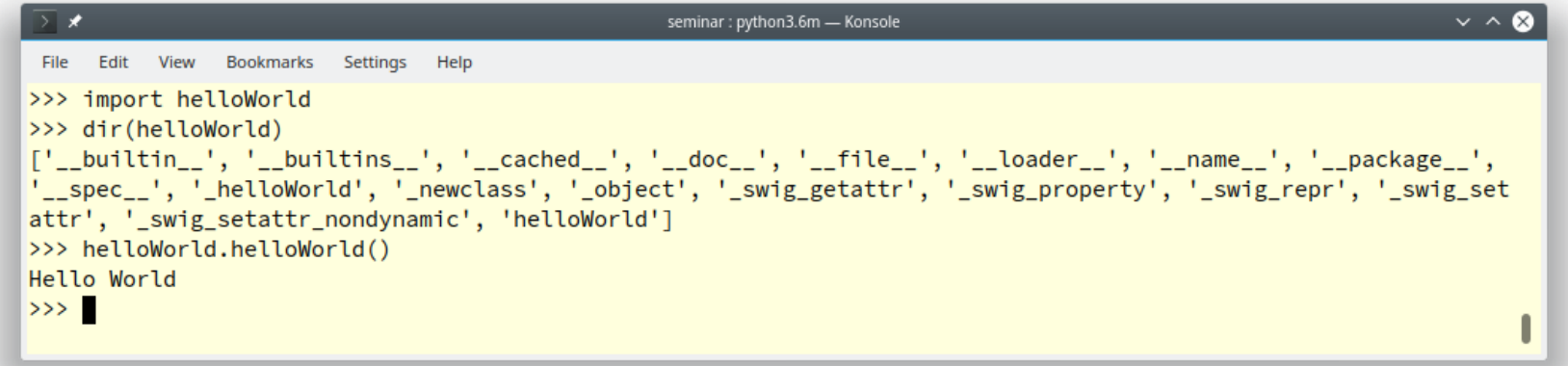
```
1   #include "helloWorld.h"
2
3   void helloWorld() {
4       printf("Hello World\n");
5   }
```

SWIG

- Building the expansion module



- Using the extension module



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

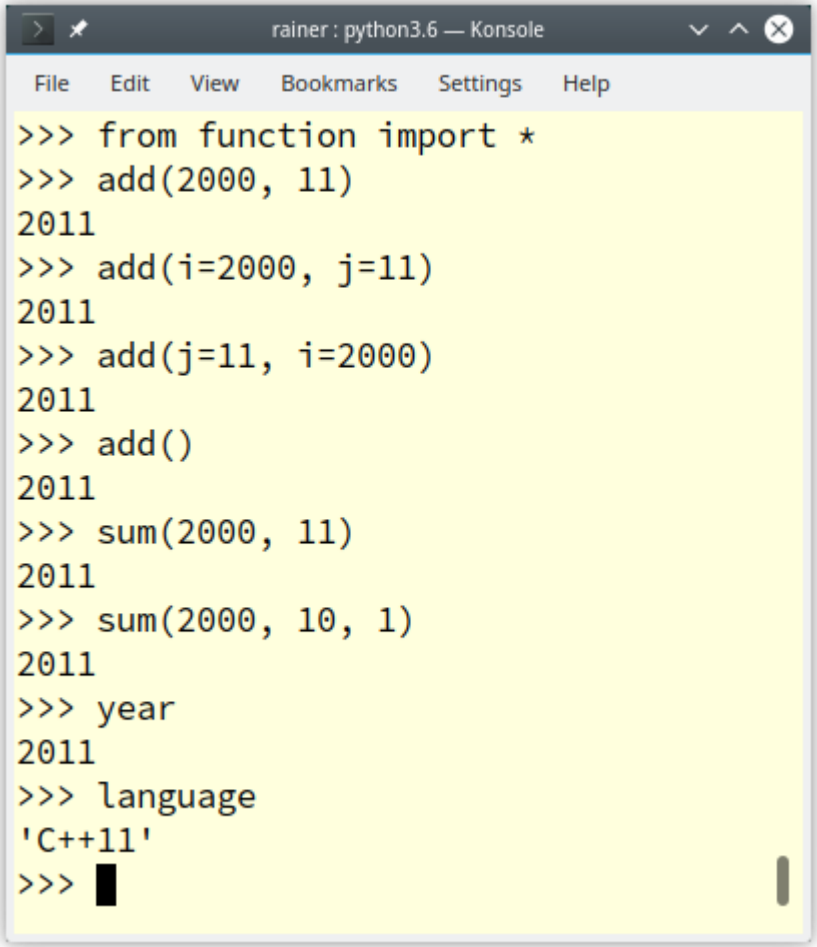
- Funtions
  - Overload

```
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


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

```cpp
#include <pybind11/pybind11.h>
#include <string>

struct HumanBeing {
    HumanBeing(const std::string& n) : name(n) { }
    const std::string& getName() const { return name; }
    std::string name;
};

namespace py = pybind11;

PYBIND11_MODULE(human, m) {
    py::class_<HumanBeing>(m, "HumanBeing")
        .def(py::init<const std::string &>())
        .def("getName", &HumanBeing::getName);
}
```

- `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;
})
```

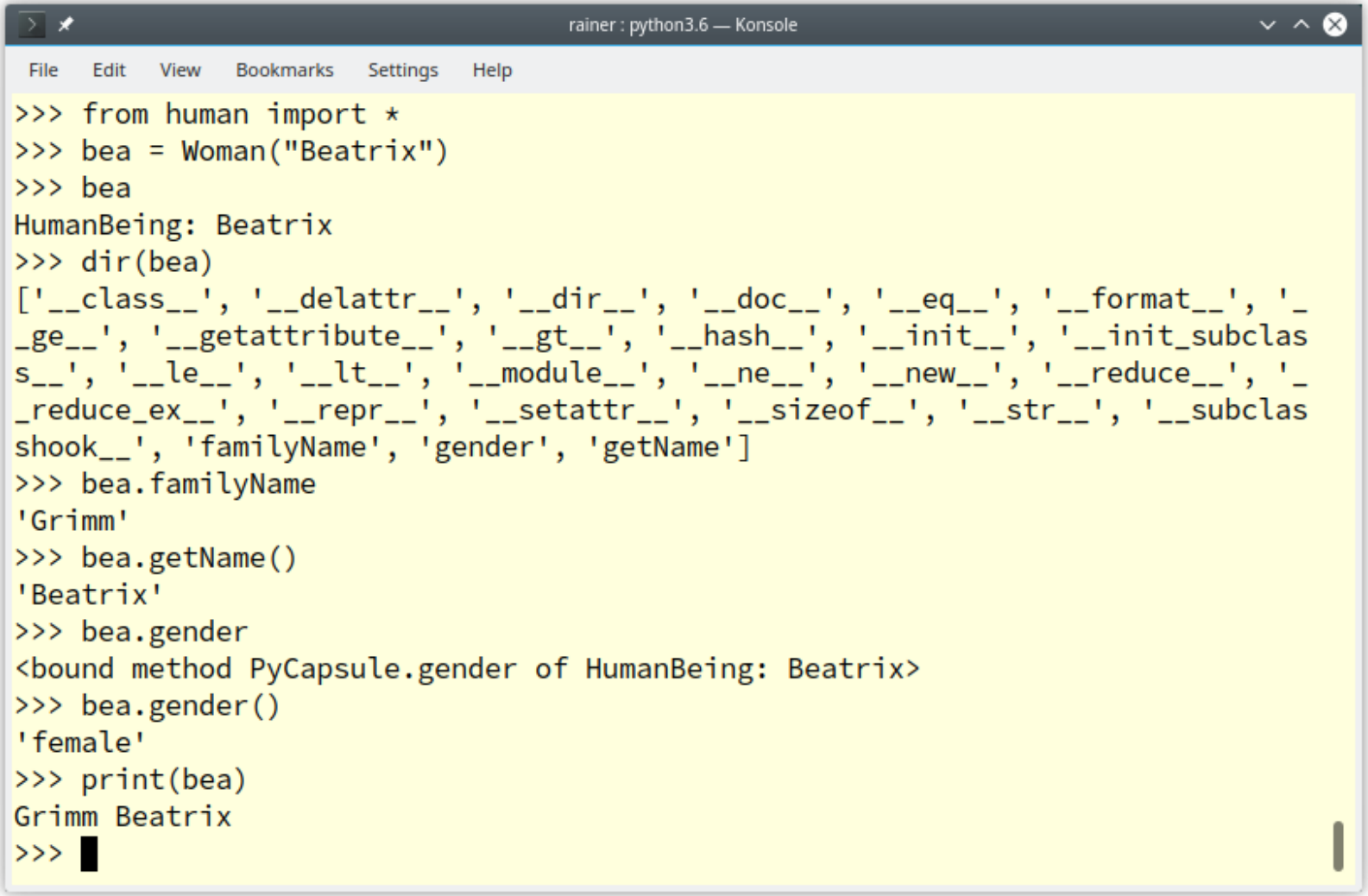
- Attributes

```
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__', '__getattribute__', '__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

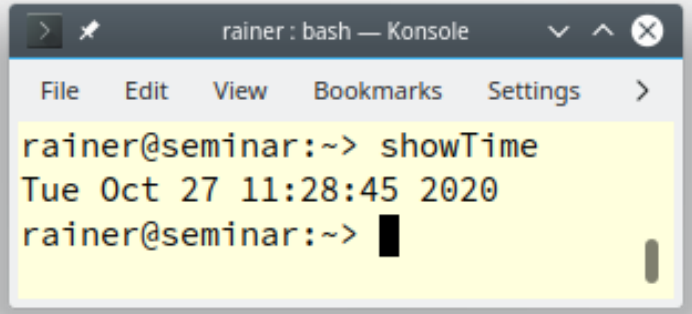
- Implementation of the C program

```
 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



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

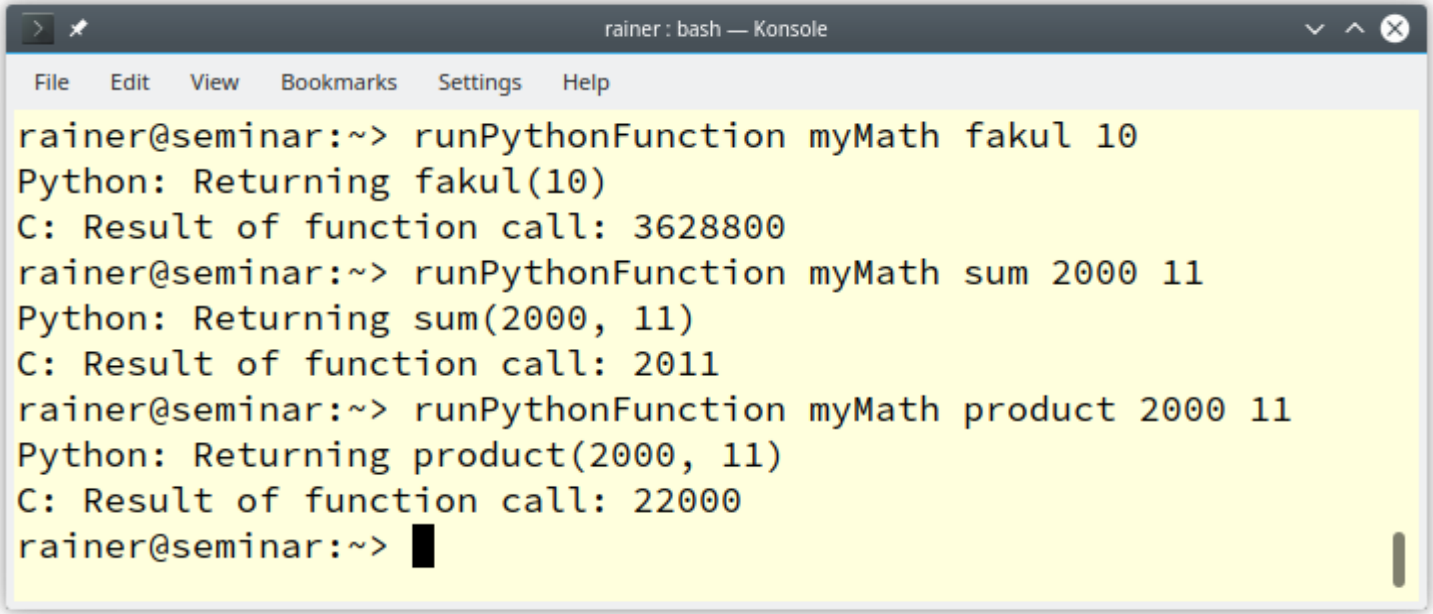
The `myMath.py` module

```
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
```



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