

Erweitern und Einbetten von Python

Rainer Grimm

Training, Coaching und
Technologieberatung

Erweitern und einbetten

Erweitern

1. Konvertiere die Werte von Python nach C/C++.
2. Verwende die konvertierten Werte um die C/C++-Funktionalität auszuführen.
3. Konvertiere die Ergebnisse von C/C++ nach Python zurück.

Einbetten

1. Konvertiere die Werte von C/C++ nach Python.
2. Verwende die konvertierten Werte um die Python-Funktionalität auszuführen.
3. Konvertiere die Ergebnisse von Python nach C/C++ zurück.

Erweitern und einbetten

Python erweitern

Python einbetten

Python erweitern

Shared Library

Ctypes

Native

SWIG

pybind11

Erzeugen einer Shared Library (Linux)

Die Shared Library soll aus den folgenden Dateien bestehen.

helloWorld.h

```
#include <stdio.h>

void helloWorld();
```

helloWorld.c

```
#include "helloWorld.h"

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

Erzeugen einer Shared Library (Linux)

Schritte zum Erzeugen und Verwenden einer Shared Library:

1. Positionsunabhängigen Code erzeugen

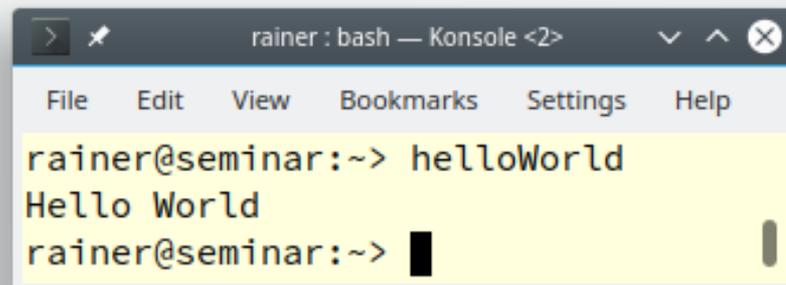
```
gcc -c -fpic helloWorld.c
```

2. Shared Library erzeugen

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

3. Linker und Runtime die Pfade bekannt machen

```
gcc -LPathToSharedLib -Wl,-rpath=PathToSharedLib -o  
helloWorldShared main.c -lhelloWorld
```



The screenshot shows a terminal window titled "rainer : bash — Konsole <2>". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content is as follows:

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

Python erweitern

Shared Library

Ctypes

Native

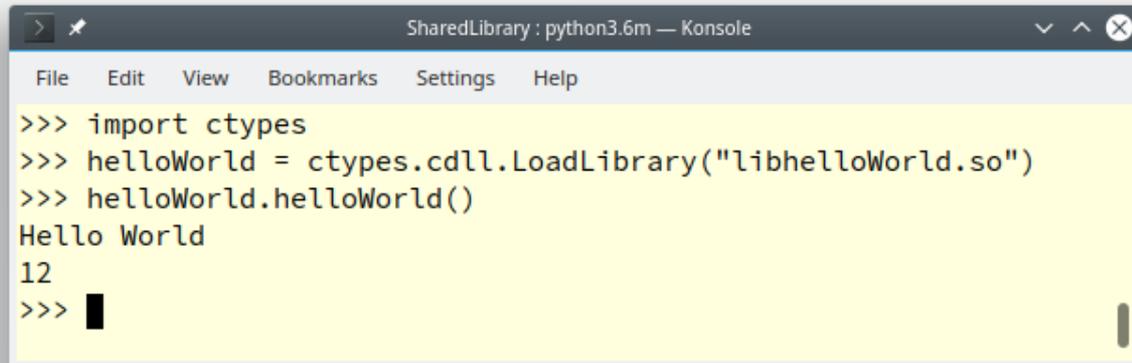
SWIG

pybind11

ctypes (Linux)

Die Library [ctypes](#) erlaubt es, Funktionen in Shared Libraries aufzurufen.

- Aufruf der Shared Library `libhelloWorld.so`.



```
SharedLibrary : python3.6m — Konsole
File Edit View Bookmarks Settings Help
>>> import ctypes
>>> helloWorld = ctypes.cdll.LoadLibrary("libhelloWorld.so")
>>> helloWorld.helloWorld()
Hello World
12
>>> █
```

ctypes (Linux)

Die Library [ctypes](#) erlaubt es, die [libc](#) zu verwenden.

```
ctypes : python3.6 — Konsole
File Edit View Bookmarks Settings Help
>>> 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'
>>> █
```

Python erweitern

Shared Library

Ctypes

Native

SWIG

pybind11

Native

Python um das Modul `helloWorld.c` erweitern.

 Die Datei `helloWorld.c` muss als Erweiterungsmodul (Shared Library) erzeugt werden.

Native

- Implementieren des Erweiterungs-Moduls

```
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) {...
```

- Zugriff auf die Python-API (1)
- Deklaration der C-Funktion (3)
- Definition der Methodentabelle (5)
- Definition des Moduls (10)
- Definition der C-Funktion (18)
- Initialisierung des Modules (27)

Native

- Definition Methodentabelle

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

- "helloWorld" : Name der Python Methode
- method_helloWorld: Name der C-Funktion
- METH_VARARGS: Aufrufkonvention für C-Funktion
- "Hello": Dokumentationsstring
- ...: weitere Methoden
- {NULL, NULL, 0, NULL}: Sentinel

Native

- Definition des Moduls

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

- "helloWorld": Name des Moduls
- "Hello World message": Dokumentation des Moduls
- -1: Größe des Interpreter-Zustands (Zustand wird in einer globalen Variable gespeichert)
- HelloWorld: Namen der Methodentabelle

Native

- Initialisierung des Modules

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

- `PyMODINIT_FUNC`: gibt ein `PyObject*` zurück
- `PyInit_helloWorld`: Initialisierungsfunktion
 - `PyInit_<Name des Modules>`
 - wird beim Laden des Modules automatisch aufgerufen
- `PyModule_Create(&helloWorldModule)`
 - erzeugt das neue Module
 - gibt es an den Aufrufer zurück

Native

Erzeugung des Moduls mit dem Python Modul [distutils](#)

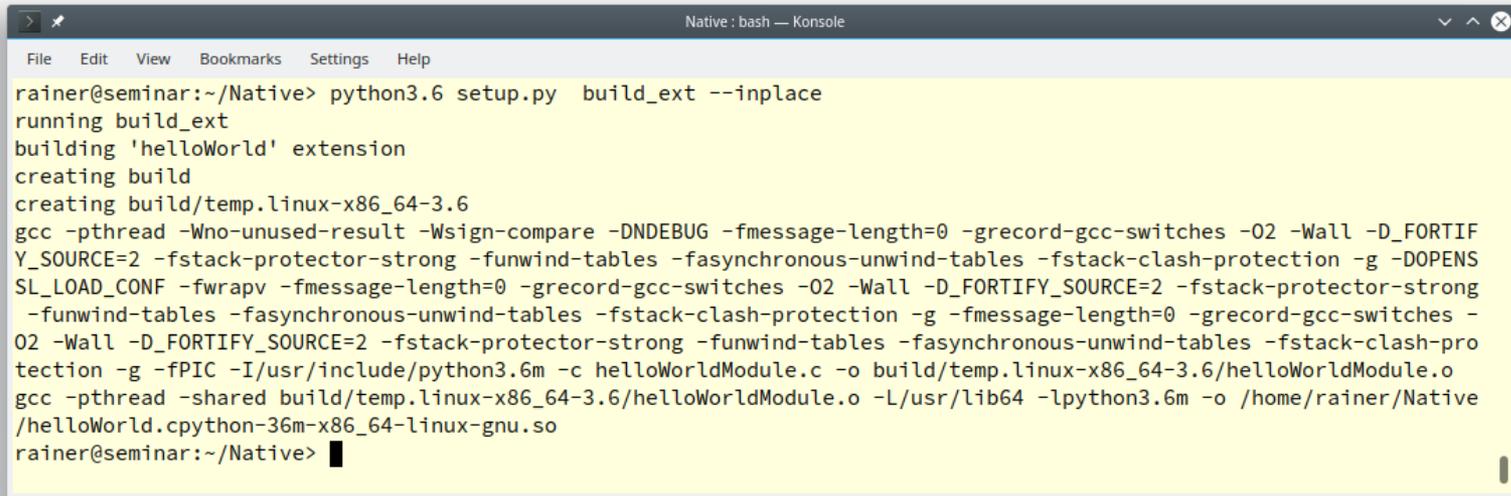
```
from distutils.core import setup, Extension

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()
```

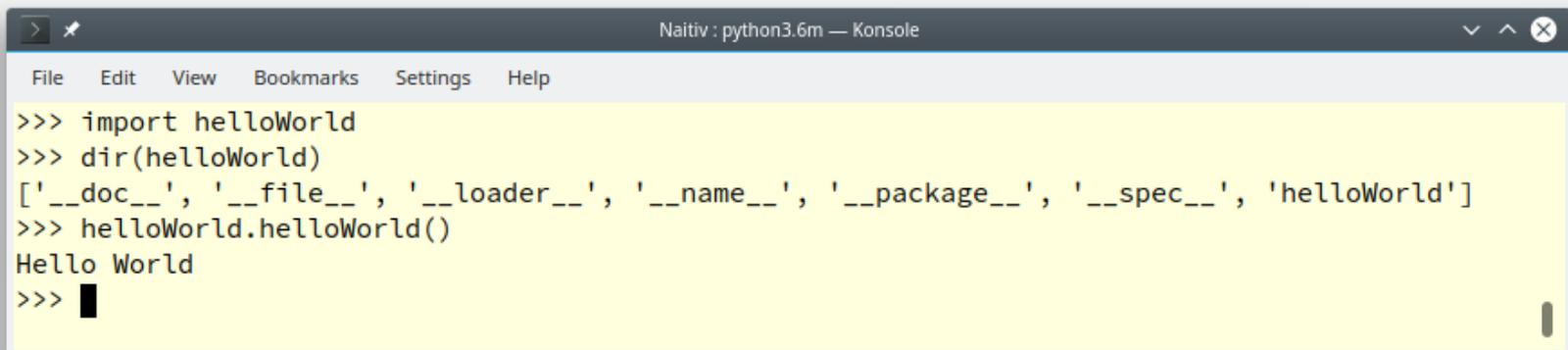
Native

- Bauen des Erweiterungsmoduls



```
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>
```

- Verwenden des Erweiterungsmoduls



```
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
>>>
```

Python erweitern

Shared Library

Ctypes

Native

SWIG

pybind11

SWIG

SWIG erzeugt Interfaces, sodass C/C++ mit anderen Programmiersprachen interagieren kann.

SWIG

- unterstützt C99 und C++98 bis C++17
- erzeugt Wrapper für die folgenden Programmiersprachen
 - C#
 - D
 - Java
 - Javascript
 - Perl
 - Python
 - PHP
 - Ruby

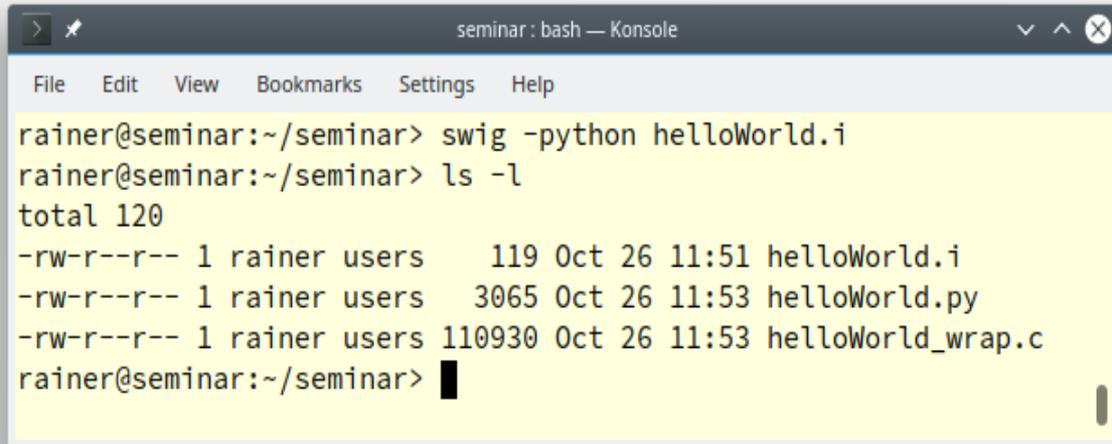
SWIG

- Definition des Interfaces

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

SWIG

- Erzeugen der Wrapper für Python



```
seminar: bash — Konsole
File Edit View Bookmarks Settings Help
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, der mit der restlichen Applikation gelinkt werden muss
- `helloWorld.py`
 - High-level Code, der in Python importiert wird

SWIG

- Implementierung der C-Funktionalität

- helloWorld.h

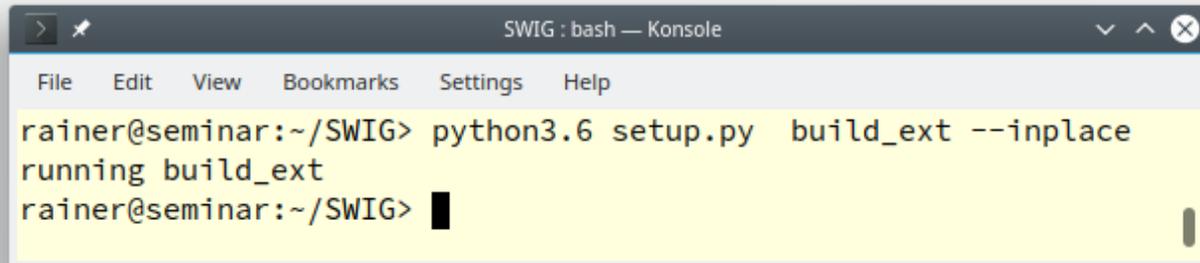
```
1  #include <stdio.h>
2
3  void helloWorld();
```

- helloWorld.c

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

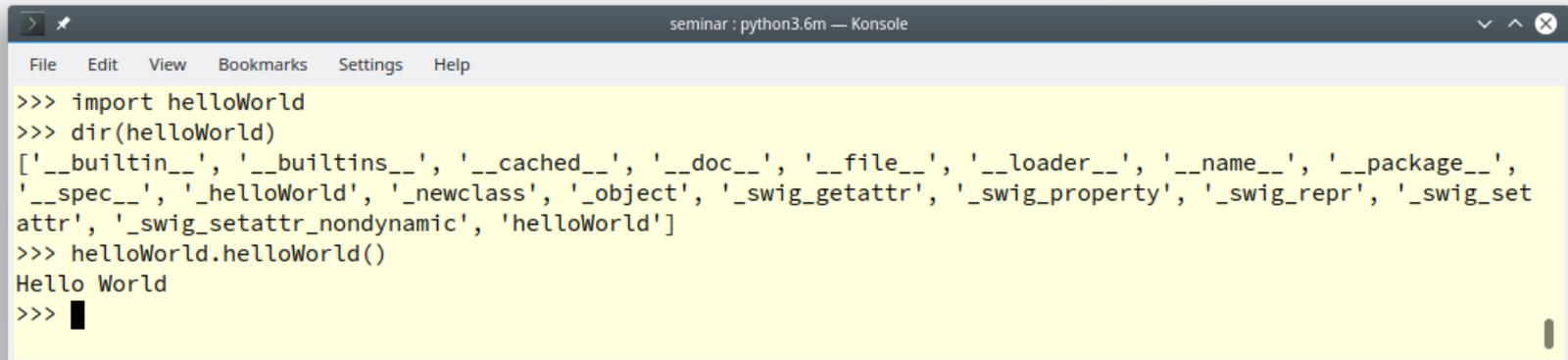
SWIG

- Bauen des Erweiterungsmoduls



```
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>
```

- Verwendung des Erweiterungsmoduls



```
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
>>>
```

Python erweitern

Shared Library

Ctypes

Native

SWIG

pybind11

pybind11

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

- Ist vollständig in Headerdateien implementiert
- Basiert auf [Boost.Python](#)
- C++ Datentypen lassen sich in Python verwenden (erweitern)
- Python Datentypen lassen sich in C++ verwenden (einbetten)

pybind11

- Kernfeature
 - Lambda Ausdrücke
 - Funktionen
 - Argumente per Value, Referenz oder Zeiger annehmen
 - Überladen
 - Klassen
 - Methoden und Attribute
 - Einfach- und Mehrfachvererbung
 - Virtualität
 - Bibliothek
 - 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 Bindung
- `PYBIND11_MODULE`: wird durch `import` aufgerufen
- `function`: Name des Moduls
- `m`: Variable vom Typ `py::module_`
- `m.def`: macht die Funktion Python bekannt

pybind11

- Konvention

```
namespace py = pybind11;
```

- Funktionen

- Schlüsselwortargumente

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

- Defaultargumente

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

pybind11

- Funktionen

- Überladen

```
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");
```

- Variablen

```
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

- Objektorientierung

```
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_`: erzeugt eine Klasse
- `py::init`: benötigt die Parameter des Konstruktors als Template-Argumente

pybind11

- **Spezielle Methoden**

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

- **Attribute**

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

- **Vererbung**

```
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
>>> █
```

Erweitern und einbetten

Python erweitern

Python einbetten

String direkt ausführen

String ausführen

Module ausführen

Funktionen ausführen

String ausführen

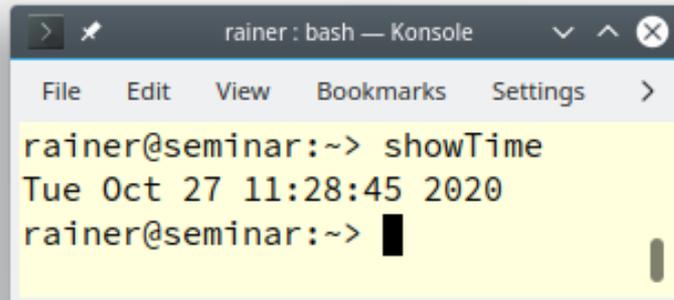
- Implementierung der C-Programms

```
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 }
```

- Python Interpreter initialisieren (5)
- Python Sourcecode ausführen (6)
- Interpreter herunterfahren (8)

String ausführen

- Ausführen des Programms



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

A terminal window titled "rainer : bash — Konsole" with a menu bar containing "File", "Edit", "View", "Bookmarks", and "Settings". The terminal content shows the command "showTime" being executed, resulting in the output "Tue Oct 27 11:28:45 2020". The prompt "rainer@seminar:~>" is followed by a cursor "█".

String ausführen

String ausführen

Modul ausführen

Funktionen ausführen

Modul ausführen

Das Modul `showTime.py`

```
import time
```

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

Modul ausführen

- Implementierung der C-Programms

```
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 }
```

- Python Interpreter initialisieren (6)
- Python Sourcecode ausführen (9)
- Interpreter herunterfahren (12)

String direkt ausführen

String ausführen

Modul ausführen

Funktionen ausführen

Funktionen ausführen

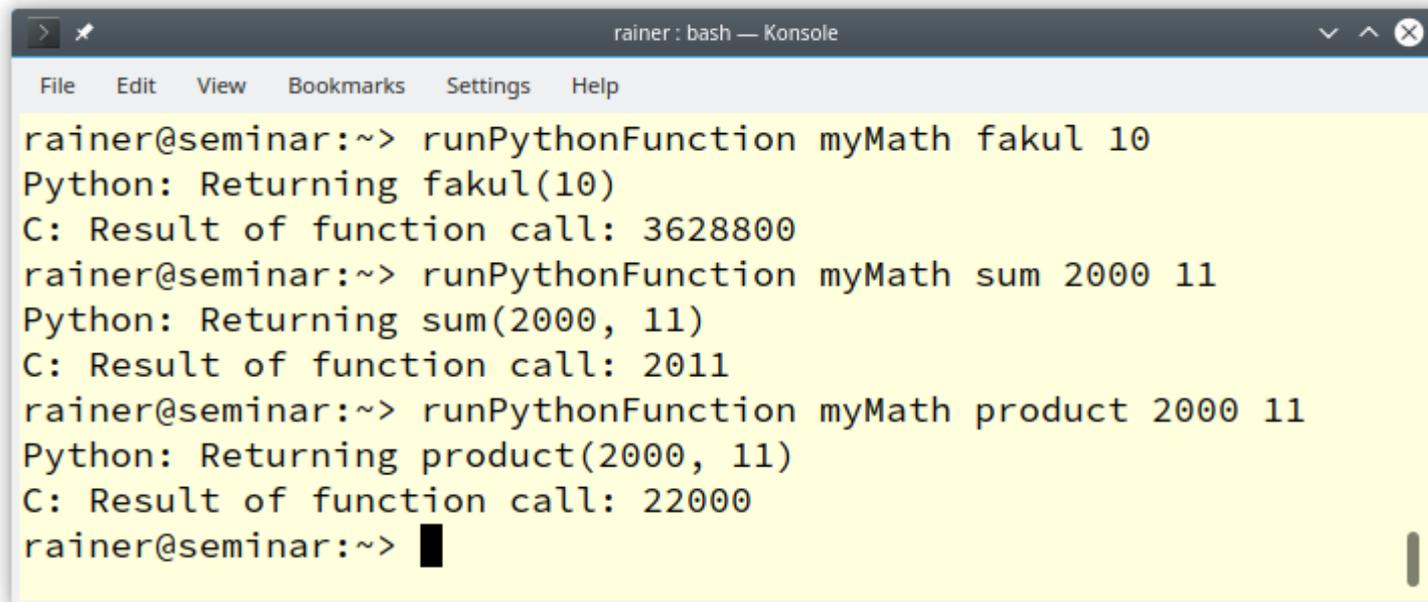
Das Modul `math.py`

```
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
```

Funktionen ausführen

Das C-Programm `runPythonFunction.c` erlaubt es, eine Funktion eines Python Moduls auszuführen.

`runPythonFunction Modul Funktion Argumente`



```
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:~> █
```

Funktionen ausführen

Folgende Schritte führt die Datei `runPythonFunction.c` aus.

- Lesen der Kommandozeile
- `sys.path` um das lokale Verzeichnis erweitern
- Importieren des Python-Moduls
- Parsen der Funktionsargumente
- Aufruf der Python-Funktion
- Das Ergebnis der Python-Funktion in C verwenden