Python in High-Performance Computing

Jussi Enkovaara

Martti Louhivuori

Outline

- NumPy – fast array interface for Python
- MPI4Py – MPI interface for Python
- Extending Python with C

Numpy

Numpy – fast array interface

- Standard Python is not well suitable for numerical computations
  - lists are very flexible but also slow to process in numerical computations
- Numpy adds a new **array** data type
  - static, multidimensional
  - fast processing of arrays
  - some linear algebra, random numbers

Numpy arrays

- All elements of an array have the same type
- Array can have multiple dimensions
- The number of elements in the array is fixed, shape can be changed

Creating numpy arrays

- From a list

```
>>> import numpy as np
>>> a = np.array((1, 2, 3, 4), float)
>>> a
array([ 1.,  2.,  3.,  4.])
>>> list1 = [[1, 2, 3], [4,5,6]]
>>> mat = np.array(list1, complex)
>>> mat
array([[ 1.+0.j,  2.+0.j,  3.+0.j],
       [ 4.+0.j,  5.+0.j,  6.+0.j]])
>>> mat.shape
(2, 3)
>>> mat.size
6
```

Creating numpy arrays

- More ways for creating arrays

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b = np.linspace(-4.5, 4.5, 5)
>>> b
array([-4.5 , -2.25,  0.  ,  2.25,  4.5 ])
>>> c = np.zeros((4, 6), float)
>>> c.shape
(4, 6)
>>> d = np.ones((2, 4))
>>> d
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

Indexing and slicing arrays

- Simple indexing

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat[0,2]
3
>>> mat[1,-2]
>>> 5
```

- Slicing is possible over all dimensions

```
>>> a = np.arange(10)
>>> a[1:7:2]
array([1, 3, 5])
>>> a = np.zeros((4, 4))
>>> a[1:3, 1:3] = 2.0
>>> a
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  2.,  2.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Views and copies of arrays

- Simple assignment creates references to arrays
- Slicing creates “views” to the arrays
- Use copy() for real copying of arrays

```
a = np.arange(10)
b = a # reference, changing values in b changes a
b = a.copy() # true copy

c = a[1:4] # view, changing c changes elements [1:4] of a
c = a[1:4].copy() # true copy of subarray
```

Array manipulation

- reshape : change the shape of array

```
>>> mat = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat
array([[1, 2, 3],
       [4, 5, 6]])
>>> mat.reshape(3,2)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

- ravel : flatten array to 1-d

```
>>> mat.ravel()
array([1, 2, 3, 4, 5, 6])
```

Array manipulation

- concatenate : join arrays together

```
>>> mat1 = np.array([[1, 2, 3], [4, 5, 6]])
>>> mat2 = np.array([[7, 8, 9], [10, 11, 12]])
>>> np.concatenate((mat1, mat2))
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> np.concatenate((mat1, mat2), axis=1)
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

- split : split array to N pieces

```
>>> np.split(mat1, 3, axis=1)
[array([[1],
       [4]]), array([[2],
       [5]]), array([[3],
       [6]])]
```

Array operations

- Most operations for numpy arrays are done element-wise
  - +, -, *, /, **

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
>>> a + b
array([ 3.,  4.,  5.])
>>> a * a
array([ 1.,  4.,  9.])
```

Array operations

- Numpy has special functions which can work with array arguments
  - sin, cos, exp, sqrt, log, ...

```
>>> import numpy, math
>>> a = numpy.linspace(-pi, pi, 8)
>>> a
array([-3.14159265, -2.24399475, -1.34639685, -0.44879895,
         0.44879895, 1.34639685,  2.24399475,  3.14159265])
>>> math.sin(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: only length-1 arrays can be converted to Python scalars
>>> numpy.sin(a)
array([ -1.22464680e-16,  -7.81831482e-01,  -9.74927912e-01,
        -4.33883739e-01,   4.33883739e-01,   9.74927912e-01,
         7.81831482e-01,   1.22464680e-16])
```

Vectorized operations

- for loops in Python are slow
- Use “vectorized” operations when possible
- Example: difference

```
arr = np.arange(1000)
dif = np.zeros(999, int)
for i in range(1, len(arr)):
    dif[i-1] = arr[i] - arr[i-1]
```

vs.

```
arr = np.arange(1000)
dif = arr[1:] - arr[:-1]
```

  - for loop is ~80 times slower!

Broadcasting

- If array shapes are different, the smaller array may be broadcasted into a larger shape

```
>>> from numpy import array
>>> a = array([[1,2],[3,4],[5,6]], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b = array([[7,11]], float)
>>> b
array([[  7.,  11.]])
>>> a * b
array([[  7.,  22.],
       [ 21.,  44.],
       [ 35.,  66.]])
```

Advanced indexing

- Numpy arrays can be indexed also with other arrays (integer or boolean)

```
>>> x = np.arange(10,1,-1)
>>> x
array([10,  9,  8,  7,  6,  5,  4,  3,  2])
>>> x[np.array([3, 3, 1, 8])]
array([7, 7, 9, 2])
```

- Boolean “mask” arrays

```
>>> m = x > 7
>>> m
array([ True,  True,  True, False, False, ...
>>> x[m]
array([10,  9,  8])
```

- Advanced indexing creates copies of arrays

Masked arrays

- Sometimes datasets contain invalid data (faulty measurement, problem in simulation)
- Masked arrays provide a way to perform array operations neglecting invalid data
- Masked array support is provided by numpy.ma module

Masked arrays

- Masked arrays can be created by combining a regular numpy array and a boolean mask

```
>>> import numpy.ma as ma
>>> x = np.array([1, 2, 3, -1, 5])
>>> m = x < 0
>>> mx = ma.masked_array(x, mask=m)
>>> mx
masked_array(data = [1 2 3 -- 5],
             mask = [False False False True False],
       fill_value = 999999)
>>> x.mean()
2.0
>>> mx.mean()
2.75
```

I/O with Numpy

- Numpy provides functions for reading data from file and for writing data into the files
- Simple text files
  - numpy.loadtxt
  - numpy.savetxt
  - Data in regular column layout
  - Can deal with comments and different column delimiters

Random numbers

- The module numpy.random provides several functions for constructing random arrays
  - random: uniform random numbers
  - normal: normal distribution
  - poisson: Poisson distribution
  - ...

```
>>> import numpy.random as rnd
>>> rnd.random((2,2))
array([[ 0.02909142,  0.90848   ],
       [ 0.9471314 ,  0.31424393]])
>>> rnd.poisson(size=(2,2))
```

Polynomials

- Polynomial is defined by array of coefficients p

  $p(x, N) = p[0]\, x^{N-1} + p[1]\, x^{N-2} + \ldots + p[N-1]$
- Least square fitting: numpy.polyfit
- Evaluating polynomials: numpy.polyval
- Roots of polynomial: numpy.roots
- ...

```
>>> x = np.linspace(-4, 4, 7)
>>> y = x**2 + rnd.random(x.shape)
>>>
>>> p = np.polyfit(x, y, 2)
>>> p
array([ 0.96869003, -0.01157275,  0.69352514])
```

Linear algebra

- Numpy can calculate matrix and vector products efficiently
  - dot, vdot, ...
- Eigenproblems
  - linalg.eig, linalg.eigvals, …
- Linear systems and matrix inversion
  - linalg.solve, linalg.inv

```
>>> A =  np.array(((2, 1), (1, 3)))
>>> B =  np.array(((-2, 4.2), (4.2, 6)))
>>> C = np.dot(A, B)
>>> b = np.array((1, 2))
>>> np.linalg.solve(C, b)  # solve C x = b
array([ 0.04453441,  0.06882591])
```

Numpy performance

- Matrix multiplication
  - C = A * B
  - matrix dimension 200
  - pure python: 5.30 s
  - naive C: 0.09 s
  - numpy.dot: 0.01 s

Summary

- Numpy provides a static array data structure
- Multidimensional arrays
- Fast mathematical operations for arrays
- Arrays can be broadcasted into same shapes
- Tools for linear algebra and random numbers

Parallel programming with Python using mpi4py

Outline

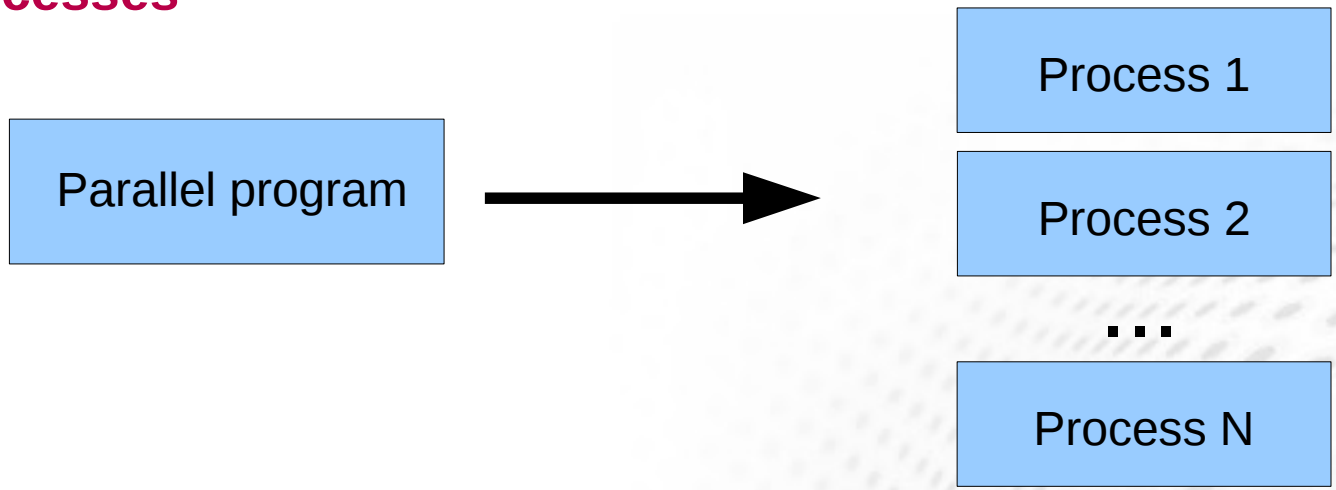
- Brief introduction to message passing interface (MPI)
- Python interface to MPI – mpi4py
- Performance considerations

Message passing interface

- MPI is an application programming interface (API) for communication between separate processes
- The most widely used approach for distributed parallel computing
- MPI programs are portable and scalable
  - the same program can run on different types of computers, from PC's to supercomputers
- MPI is flexible and comprehensive
  - large (over 120 procedures)
  - concise (often only 6 procedures are needed)
- MPI standard defines C and Fortran interfaces
- **mpi4py** provides (an unofficial) Python interface

Execution model in MPI

- Parallel program is launched as set of **independent, identical processes**



- All the processes contain the same program code and instructions
- Processes can reside in different nodes or even in different computers
- The way to launch parallel program is implementation dependent
  - mpirun, mpiexec, aprun, poe, ...
- When using Python, one launches N Python interpreters
  - mpirun -np 32 python parallel_script.py

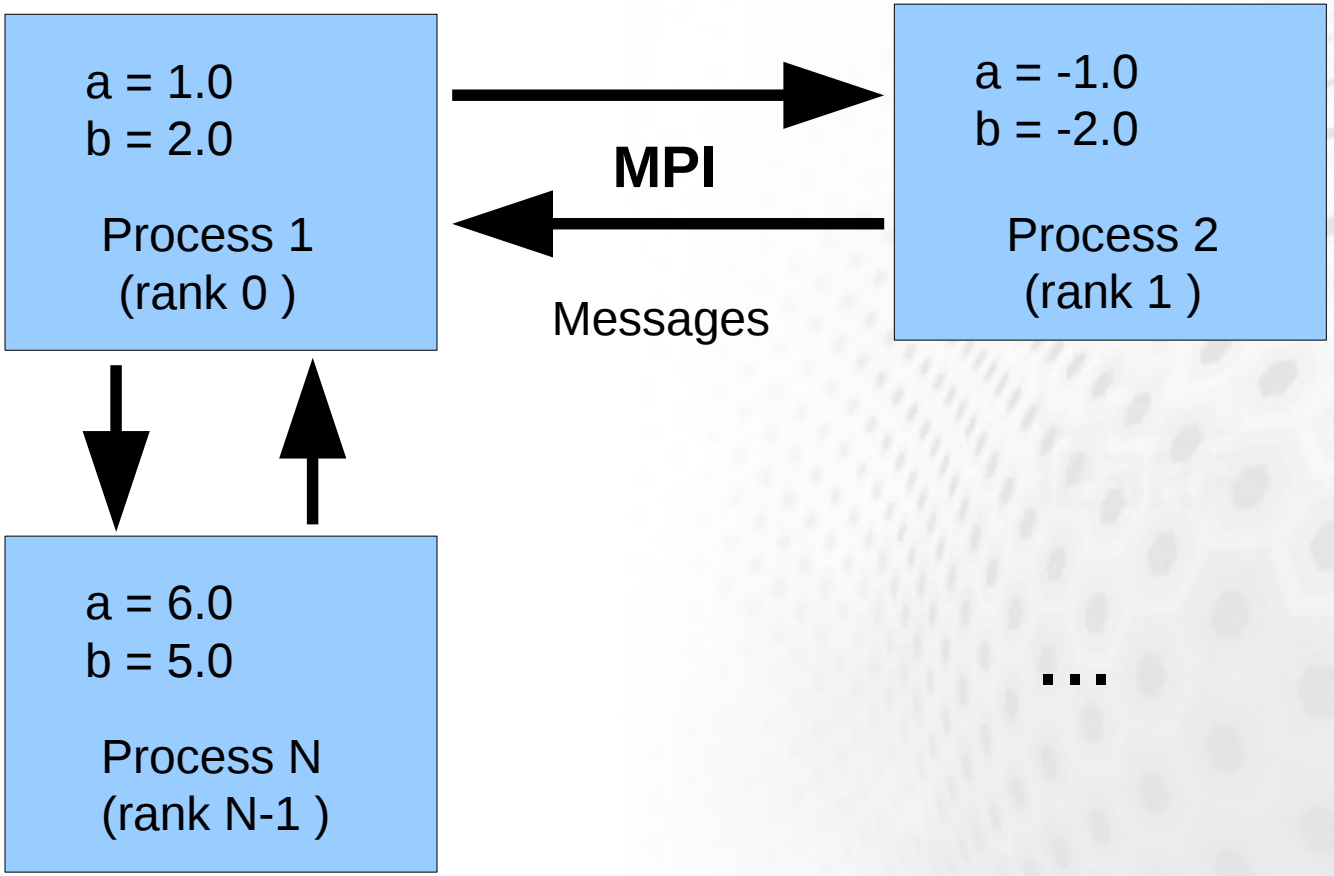
MPI Concepts

- rank: id number given to process
  - it is possible to query for rank
  - processes can perform different tasks based on their rank
- Communicator: group containing process
  - in mpi4py the basic object whose methods are called
  - **MPI_COMM_WORLD** contains all the process (MPI.COMM_WORLD in mpi4py)

```
...
if ( rank == 0 )
   …

if ( rank == 1)
   …

...
```

Data model

- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving messages



Using mpi4py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD   # Communicator object containing all
                        # processes
```

- Basic methods of communicator object
  - Get_size()
    Number of processes in communicator
  - Get_rank()
    rank of this process

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

print “I am rank %d in group of %d processes” % (rank, size)
```

Sending and receiving data

- Sending and receiving a dictionary

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   data = {'a': 7, 'b': 3.14}
   comm.send(data, dest=1, tag=11)
elif rank == 1:
   data = comm.recv(source=0, tag=11)
```

Sending and receiving data

- Arbitrary Python objects can be communicated with the **send** and **receive** methods of communicator

  send(data, dest, tag)

  - **data** Python object to send
  - **dest** destination rank
  - **tag** id given to the message

  receive(source, tag)

  - **source** source rank
  - **tag** id given to the message
  - data is provided as return value

- Destination and source ranks as well as tags have to match

Communicating NumPy arrays

- Arbitrary Python objects are converted to byte streams when sending
- Byte stream is converted back to Python object when receiving
- Conversions give overhead to communication
- (Contiguous) NumPy arrays can be communicated with very little overhead with upper case methods:

  Send(data, dest, tag)

  Recv(data, source, tag)

  – Note the difference in receiving: the data array has to exist in the time of call

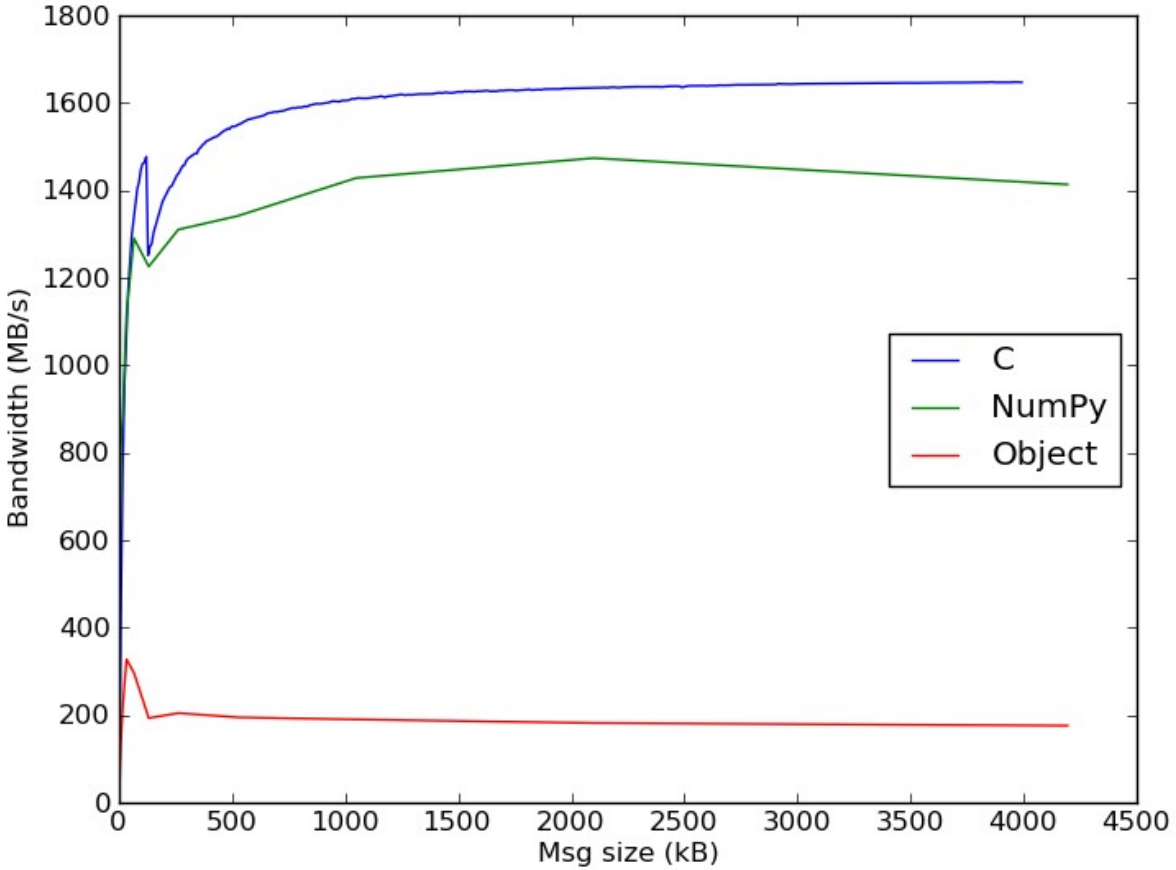
Communicating NumPy arrays

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   data = numpy.arange(100, dtype=numpy.float)
   comm.Send(data, dest=1, tag=13)
elif rank == 1:
   data = numpy.empty(100, dtype=numpy.float)
   comm.Recv(data, source=0, tag=13)
```

- Note the difference between upper/lower case!
  - send/recv: general Python objects, slow
  - Send/Recv: continuous arrays, fast

mpi4py performance

- Ping-pong test



Summary

- mpi4py provides Python interface to MPI
- MPI calls via communicator object
- Possible to communicate arbitrary Python objects
- NumPy arrays can be communicated with nearly same speed as from C/Fortran

C - extensions

C - extensions

- Some times there are time critical parts of code which would benefit from compiled language
- 90/10 rule: 90 % of time is spent in 10 % of code
  - only a small part of application benefits from compiled code
- It is relatively straightforward to create a Python interface to C-functions
  - data is passed from Python, routine is executed without any Python overheads

C - extensions

- C routines are build into a shared library
- Routines are loaded dynamically with normal import statements

```
import hello

hello.hello()
```

- A library **hello.so** is looked for
- A function **hello** (defined in myext.so) is called

Creating C-extension

1) Include Python headers

```
#include <Python.h>
```

2) Define the C-function

```
...
PyObject* hello_c(PyObject *self, PyObject *args)
{
  printf("Hello\n");
  Py_RETURN_NONE;
}
```

- Type of function is always PyObject
- Function arguments are always the same
  (args is used for passing data from Python to C)
- A macro py_RETURN_NONE is used for returning “nothing

Creating C-extension

3) Define the Python interfaces for functions

```
...
static PyMethodDef functions[] = {
 {"hello", hello_c, METH_VARARGS, 0},
 {“func2”, func2, METH_VARARGS, 0},
 {0, 0, 0, 0}  /* “Sentinel” notifies the end of definitions */
};
```

- **hello** is the function name used in Python code, **hello_c** is the actual C-function to be called
- Single extension module can contain several functions (hello, func2, ...)

Creating C-extension

4) Define the module initialization function

```
...
PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", functions);
}
```

- Extension module should be build into **hello.so**
- Extension is module is imported as **import hello**
- Functions/interfaces defined in functions are called as hello.hello(), hello.func2(), ...

5) Compile as shared library

```
gcc -shared -o myext.so -I/usr/include/python2.6 -fPIC myext.c
```

- The location of Python headers (/usr/include/...) may vary in different systems

Full listing of hello.c

```
#include <Python.h>

PyObject* hello_c(PyObject *self, PyObject *args)
{
  printf("Hello\n");
  Py_RETURN_NONE;
}

static PyMethodDef functions[] = {
 {"hello", hello_c, METH_VARARGS, 0},
 {0, 0, 0, 0}
};

PyMODINIT_FUNC inithello(void)
{
    (void) Py_InitModule("hello", functions);
}
```

Passing arguments to C-functions

```
...
PyObject* my_C_func(PyObject *self, PyObject *args)
{
  int a;
  double b;
  char* str;
  if (!PyArg_ParseTuple(args, "ids", &a, &b, &str))
    return NULL;
  printf("int %i, double %f, string %s\n", a, b, str);
  Py_RETURN_NONE;
}
```

- **PyArg_ParseTuple** checks that function is called with proper arguments
  "ids" : integer, double, string
  and does the conversion from Python to C types

Returning values

```
...
PyObject* square(PyObject *self, PyObject *args)
{
  int a;
  if (!PyArg_ParseTuple(args, "i", &a))
    return NULL;
  a = a*a;
  return Py_BuildValue("i", a);
}
```

- Create and return Python integer from C variable a
  “d” would create Python double etc.
- Returning tuple:
  Py_BuildValue("(ids)", a, b, str);

Operating with NumPy array

```
#include <Python.h>
#define NO_IMPORT_ARRAY
#include <numpy/arrayobject.h>

PyObject* my_C_func(PyObject *self, PyObject *args)
{
  PyArrayObject* a;
  if (!PyArg_ParseTuple(args, "O", &a))
    return NULL;

  int size = PyArray_SIZE(a);  /* Total size of array */
  double *data = PyArray_DATA(a); /* Pointer to data */
  for (int i=0; i < size; i++)
  {
     data[i] = data[i] * data[i];
  }
  Py_RETURN_NONE;
}
```

- NumPy provides API also for determining the dimensions of array etc.

Tools for easier interfacing

- Cython
- SWIG
- pyrex
- f2py (for Fortran code)

Summary

- Python can be extended with C-functions relatively easily
- C-extension build as shared library
- It is possible to pass data between Python and C code
- Extending Python:
  http://docs.python.org/extending/
- NumPy C-API

  http://docs.scipy.org/doc/numpy/reference/c-api.html