[Software Development]

Python – Optimization and Integration

Davide Balzarotti

Eurecom – Sophia Antipolis, France

When Python is not Enough

- Python is great for rapid application development
- Many famous examples...
  - Fermilab (scientific applications)
  - Google (in many services)
  - Industrial Light & Magic (everything)
  - …
- But nothing is perfect
  - Sometimes it is just too slow
  - Sometimes libraries are available only in C/C++
  - Sometimes you need to interact with some low-level functionality (system calls, I/O ports..)

The Best of Two Languages

- Two approaches to combine Python with C/C++
- Extending Python
  - Write the application in python and import external plugins to perform functions that are otherwise unavailable or just too slow
  - Best solution when Python is good enough for the rest of the application. The plugins provide the missing pieces
  - Examples: python imaging library (PIL), python numeric

The Best of Two Languages

- Two approaches to combine Python with C/C++
- Extending Python
  - Write the application in python and import external plugins to perform functions that are otherwise unavailable or just too slow
  - Best solution when Python is good enough for the rest of the application. The plugins provide the missing pieces
  - Examples: python imaging library (PIL), python numeric
- Embedding Python
  - Write the application in C, but run a Python interpreter in it
  - Best approach when Python is not the best solution, but you still want to support configuration or scripting in Python
  - Examples: Civilization IV, Gimp, IDA Pro, GDB

Step I: Find the Bottleneck in the code

- Usually a program spends 90% of its time in 10% of its code
- If the program is too slow, check where the bottleneck is
  - Do not try to guess which is the slow part
    (it’s not always as easy or as obvious as it may seem)

Step I: Find the Bottleneck in the code

- Usually a program spends 90% of its time in 10% of its code
- If the program is too slow, check where the bottleneck is
  - Do not try to guess which is the slow part
    (it’s not always as easy or as obvious as it may seem)
- Adopt a more systematic way:
  - Write the entire program prototype in Python
  - Run the program with a profiler (or manually time the different functions)
  - Identify the points that consume most of the CPU time
- Solutions:
  - Check the algorithm and the data structures
    (if it is slow, it is often because of a bad design.. i.e., it's your fault)
  - If no further optimization is possible... it may be time to get back to C

Timeit

- Simple benchmarking module
  - Useful to measure the performance of small bits of Python code
  - It executes the code multiple times (default is one million) and it takes the total time

```
>>> import timeit
>>> t = timeit.Timer(setup='x=[9,8,7,6,5,4,3,2,1]*100',
                  stmt='x.sort()')
>>> t.timeit(10000)
0.1711

>>> t = timeit.Timer(setup='x=[9,8,7,6,5,4,3,2,1]*100',
                  stmt='y=sorted(x)')
>>> t.timeit(10000)
0.6711
```

Python Profiler

- A profiler is a program that measures the runtime performance of a program, and provides a number of statistics
- Two approaches:
  - Deterministic: instrument all the function calls to get precise timing
  - Statistical: randomly sample the execution point to estimate where the time is spent in the code

Python Profiler

- A profiler is a program that measures the runtime performance of a program, and provides a number of statistics
- Two approaches:
  - Deterministic: instrument all the function calls to get precise timing
  - Statistical: randomly sample the execution point to estimate where the time is spent in the code
- Python `cProfile` module: deterministic profiler implemented in C with a reasonable overhead
- Two simple way to use it:
  - From the command line: `python -m cProfile python-script`
  - From the code:

```
import cProfile;
cProfile.run('instruction')
```

```
def distance(p1,p2):
    x1,y1,z1 = p1
    x2,y2,z2 = p2
    tmp = (x2-x1)**2+(y2-y1)**2+(z2-z1)**2
    return math.sqrt(tmp)

def get_close(points, target, dist):
    return [p for p in points if distance(target,p) < dist]

points = get_random_points()
get_close(points, [5,5,5], 20)
```

```python
import cProfile
def distance(p1,p2):
    x1,y1,z1 = p1
    x2,y2,z2 = p2
    tmp = (x2-x1)**2+(y2-y1)**2+(z2-z1)**2
    return math.sqrt(tmp)

def get_close(points, target, dist):
    return [p for p in points if distance(target,p) < dist]

points = get_random_points()
cProfile.run('get_close(points, [5,5,5], 20)')
```

```python
import cProfile
def distance(p1,p2):
    x1,y1,z1 = p1
    x2,y2,z2 = p2
    tmp = (x2-x1)**2+(y2-y1)**2+(z2-z1)**2
    return math.sqrt(tmp)

def get_close(points, target, dist):
    return [p for p in points if distance(target,p) < dist]

points = get_random_points()
cProfile.run('get_close(points, [5,5,5], 20)')
```

```
           200003 function calls in 0.186 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.186    0.186 <string>:1(<module>)
   100000    0.111    0.000    0.136    0.000 test.py:13(distance)
        1    0.050    0.050    0.186    0.186 test.py:20(get_close)
   100000    0.024    0.000    0.024    0.000 {math.sqrt}
```

```python
import cProfile
def distance(p1,p2):
    x1,y1,z1 = p1
    x2,y2,z2 = p2
    tmp = (x2-x1)**2+(y2-y1)**2+(z2-z1)**2
    return math.sqrt(tmp)

def get_close(points, target, dist):
    return [p for p in points if distance(target,p) < dist]

points = get_random_points()
cProfile.run('get_close(points, [5,5,5], 20)')
```

```
            200003 function calls in 0.186 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.186    0.186 <string>:1(<module>)
   100000    0.111    0.000    0.136    0.000 test.py:13(distance)
        1    0.050    0.050    0.186    0.186 test.py:20(get_close)
   100000    0.024    0.000    0.024    0.000 {math.sqrt}
```

Total time inside the function (excluding the called functions)

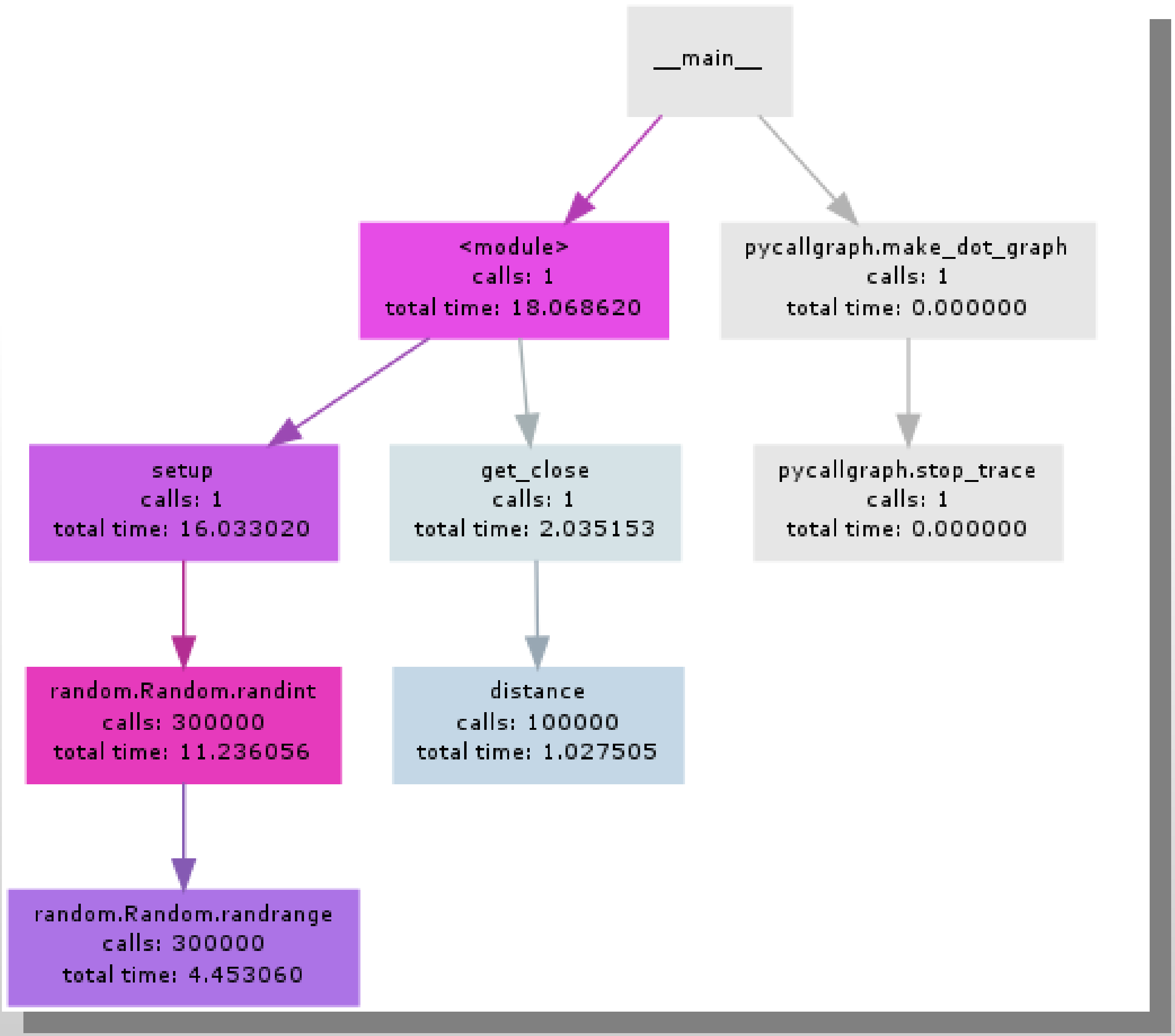
```python
import cProfile
def distance(p1,p2):
    x1,y1,z1 = p1
    x2,y2,z2 = p2
    tmp = (x2-x1)**2+(y2-y1)**2+(z2-z1)**2
    return math.sqrt(tmp)

def get_close(points, target, dist):
    return [p for p in points if distance(target,p) < dist]

points = get_random_points()
cProfile.run('get_close(points, [5,5,5], 20)')
```

```
         200003 function calls in 0.186 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.186    0.186 <string>:1(<module>)
   100000    0.111    0.000    0.136    0.000 test.py:13(distance)
        1    0.050    0.050    0.186    0.186 test.py:20(get_close)
   100000    0.024    0.000    0.024    0.000 {math.sqrt}
```

Total time inside the function (including the called functions)

PyCallGraph

```
> pycallgraph myprogram
[… a while later...]
```



Python C Extensions

- Most of python builtins and libraries are already implemented in highly optimized C (e.g., `math.sqrt()`) and are quite fast
- Python can be extended by building C/C++ modules that can be imported inside the Python code using the `import` statement
  - Most programs can be greatly optimized in term of speed by rewriting only very small part of them in C
  - Extensions are generally used to provide access to existing C libraries
- A C-coded extension is guaranteed to run only with the version of Python it is compiled for
- A Python extension module named 'foo' generally lives in a dynamic library with the same filename (`foo.so` in Unix)

Distance Module

```
#include "Python.h"

float distance(int x1, int y1, int z1, int x2, int y2, int z2){
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+...));
}

static PyObject
*distance_wrapper (PyObject *self, Pyobject *args){
   double xi,yi,zi,xj,yj,zj,dist;
   PyArg_ParseTuple(args,"dddddd",&xi,&yi,&zi,&xj,&yj,&zj);
   dist = distance(xi,yi,zi,xj,yj,zj);
   return Py_BuildValue("d",dist);
}

static PyMethodDef distance_methods[]={
 {"cdistance", distance_wrapper, METH_VARARGS},
 {NULL,NULL}
};

void initdistance(){
    (void) Py_InitModule("distance", distance_methods);
}
```

Distance Module

```
#include "Python.h"

float distance(int x1, int y1, int z1, int x2, int y2, int z2){
    return sqrt((x2-x1)*(x2
}

static PyObject
*distance_wrapper (PyObjec
   double xi,yi,zi,xj,yj,zj,dist;
   PyArg_ParseTuple(args,"dddddd",&xi,&yi,&zi,&xj,&yj,&zj);
   dist = distance(xi,yi,zi,xj,yj,zj);
   return Py_BuildValue("d",dist);
}

static PyMethodDef distance_methods[]={
 {"cdistance", distance_wrapper, METH_VARARGS},
 {NULL,NULL}
};

void initdistance(){
    (void) Py_InitModule("distance", distance_methods);
}
```

Includes the Python API
Must be included **before** any
other standard headers.

Distance Module

```
#include "Python.h"

float distance(int x1, int y1, int z1, int x2, int y2, int z2){
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+...));
}

static PyObject
*distance_wrapper (PyObject *self, Pyobject *args){
   double xi,yi,zi,xj,yj,zj,dist;
   PyArg_ParseTuple(args,"ddd
   dist = distance(xi,yi,zi,x
   return Py_BuildValue("d",d
}

static PyMethodDef distance_methods[]={
 {"cdistance", distance_wrapper, METH_VARARGS},
 {NULL,NULL}
};

void initdistance(){
    (void) Py_InitModule("distance", distance_methods);
}
```

The distance function implemented in C.

Distance Module

```
#include "Python.h"

float distance(int x1, int y1,
z2){
    return sqrt((x2-x1)*(x2-x1)+
}

static PyObject
*distance_wrapper (PyObject *self, PyObject *args){
   double xi,yi,zi,xj,yj,zj,dist;
   PyArg_ParseTuple(args,"dddddd",&xi,&yi,&zi,&xj,&yj,&zj);
   dist = distance(xi,yi,zi,xj,yj,zj);
   return Py_BuildValue("d",dist);
}

static PyMethodDef distance_methods[
                                     MET

                                     e", distance_methods);
```

Wrapper for the Python code
`self` is only used in method
`args` is a tuple containing the arguments

`PyArg_ParseTuple` extract values from a Python tuple and convert them to C values (in this case 6 decimals)

`Py_BuildValue` converts C values to Python objects. If you have a function that does not return any value, use the `Py_RETURN_NONE` macro

Distance Module

```
#include "Python.h"

float distance(int x1, int y1, int z1, int x2, int y2, int
z2){
   return sqrt((x
}

static PyObject
*distance_wrapper
   double xi,yi,zi
   PyArg_ParseTupl
   dist = distance
   return Py_BuildValue("d",dist);
}

static PyMethodDef distance_methods[]={
 {"cdist", distance_wrapper, METH_VARARGS, 'fast distance'},
 {NULL,NULL,0,NULL}
};

void initdistance(){
    (void) Py_InitModule("distance", distance_methods);
}
```

Method Table:

- python method name
- corresponding function
- calling convention (VARARGS or KEYWORDS)
- docstring

Distance Module

```
#include "Python.h"

float distance(int x1, int y1, int z1, int x2, int y2, int
z2){
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+...));
}

static PyObject
*distance_wrapper (PyObject *self, Pyobject *args){
   double xi,yi,zi,xj,yj,zj,dist;
   PyArg_ParseTuple(args,"dddddd",&xi,&yi,&zi,&xj,&yj,&zj);
   dist = distance(xi,yi,zi,xj,yj,zj);
   return Py_Bu
}

static PyMethoc
 {"cdist", dist                                              ,
 {NULL,NULL,0,NULL}
};

void initdistance(){
    (void) Py_InitModule("distance", distance_methods);
}
```

Initialization Function (kind of the main of the module):
Initialize the python module using the method table
(It's the only non-static function defined in the module)

Python Objects

- Most Python/C API functions have one or more arguments as well as a return value of type `PyObject*`
  - `PyObject*` is a pointer to an opaque data type representing an arbitrary Python object
  - `PyObject` cannot be kept on the stack, therefore only pointer variables of type `PyObject*` can be declared
- Memory Management:
  - Python handles memory management automatically by keeping track of all references to variables. This is transparent to the programmer
  - But C is not a memory-managed language !!
    As soon as a Python data structure ends up referenced from C, Python lose the ability to track the references automatically

Managing References

- The solution is to have the C code to handle all the reference counters manually
- Two macros:
  - `Py_INCREF()` - increments the reference counter
  - `Py_DECREF()` - decrements the reference counter and frees the object when the count reaches zero
- When a piece of code owns a reference to an object, it has to call `Py_DECREF()` when the reference is no longer needed
  - Forgetting to dispose of an owned reference creates a memory leak

Compiling the Extensions

- `python-config` tells you all you need to use in order to compile a Python module

```
balzarot> gcc `python-config --cflags`
 -shared -fPIC
 -o distance.so distance.c
```

- Now the C module can be used as any other python module

```
balzarot> python
>>> import distance
>>> distance.cdist(1,1,1,0,0,0)
1.732050
```

- By using the `cdist` function, the `close()` function is now 100% faster

The Pythonic Way: Distutils

- The preferred approach for building a Python extension module is to compile it with distutils
  - Distutils takes care of building the extension with all the correct flags, headers, ...
  - It can also handle the installation of the package
- The developer has to write a `setup.py` file

```
from distutils.core import setup, Extension

module1 = Extension('distance', sources=['distance.c'])

setup (name = 'distance',
       version = '1.0',
       description = 'This is the distance package',
       ext_modules = [module1])
```

The Pythonic Way: Distutils

- The preferred approach for building a Python extension module is to compile it with distutils
  - Distutils takes care of building the extension with all the correct flags, headers, ...
  - It can also handle the installation of the package
- The developer has to write a `setup.py` file

```
from distutils.core import setup, Extension

module1 = Extension('distance', sources=['distance.c'])

setup (name = 'distance',
```

```
balzarot> python setup.py build_ext --inplace

balzarot> python setup.py install
```

Using Existing Libraries

- Manual functions decoration can be cumbersome
  - It is appropriate when coding new built-in data types or core Python extensions...
  - ... but if you have a library with hundreds of functions written in C and you want to import it in Python, it's better to use a tool to simplify the job
- Examples:
  - OpenGL module
  - Subversion
  - Mapserver
  - Fifengine
  - ...

SWIG

- The *Simplified Wrapper and Interface Generator* (SWIG) is an interface compiler that connects programs written in C/C++ with scripting languages such as Perl, Python, Ruby, and Tcl (http://www.swig.org)
- SWIG decorates C source with the necessary Python markup
- Generates two files:
  - `extension.py` Python file that contains high-level support code. This is the file that you will import to use the module
  - `extension_wrap.c` C source file containing the low-level wrappers that need to be compiled and linked with the rest of the C library to obtain the Python module
- Limitations:
  - C++ support is not perfect.. check out `Boost.Python` for C++

SWIG

distance.c

```
#import "distance.h"
#import <math.h>

float distance(int x1, int y1, int z1, int x2, int y2, int
z2){
   return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+...));
}
```

distance.h

```
float distance(int x1, int y1, int z1, int x2, int y2, int z2);
```

SWIG

distance.c

```
#import "distance.h"
#import <math.h>

float distance(int x1, int y1, int z1, int x2, int y2, int
z2){
    return sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+...));
}
```

distance.h

```
float distance(int x1, int y1, int z1, int x2, int y2, int z2);
```

distance.i

```
%module distance

%{
#define SWIG_FILE_WITH_INIT
#include "distance.h"
%}

extern float distance(int x1, int y1, int z1, int x2, int
y2, int z2);
```

SWIG – Building the Module

```
balzarot> swig -python distance.i
balzarot> ls
distance.c        distance.h  distance.py
distance_wrap.c   distance.i
```

► Run swig to generate Python wrappers

SWIG – Building the Module

```
balzarot> swig -python distance.i
balzarot> ls
distance.c         distance.h  distance.py
distance_wrap.c  distance.i

balzarot> gcc distance.c distance_wrap.c
`python-config --cflags` -shared -fPIC
 -o _distance.so
```

Compile the library and the Python wrapper
Important: the name of the output file has to match the name of the module prefixed by an underscore

SWIG – Building the Module

```
balzarot> swig -python distance.i
balzarot> ls
distance.c        distance.h  distance.py
distance_wrap.c   distance.i

balzarot> gcc distance.c distance_wrap.c
`python-config --cflags` -shared -fPIC
 -o _distance.so

balzarot> python
>>> import distance
>>> distance.distance
<built-in function distance>
```

► Use the module.. the usual way

SWIG – Global Variables

- There is no direct way to map variable assignment in C to variable assignment in Python
  - In Pyhton: `"x=3; y=x"` x and y are names for the same object containing the value 3
  - In C: `"x=3; y=x;"` x and y refer to different memory locations that store the value 3
- SWIG can generate wrappers to access global C variable in Python as attributes of a special object called cvar (this name can be changed using the -globals option)

```
%module example
%inline %{
extern int x;
%}
```

```
>>> import example
>>> example.cvar.x = 3
```

SWIG – Structures

- SWIG automatically wraps C structures in Python classes

```
%module example
struct Vector {
   double x,y,z;
};
```

```
>>> import example
>>> v = example.Vector()
>>> v.x = 2.3
>>> v.y = 3.1
```

PyPy

- PyPy is a fast, compliant alternative implementation of the Python language (2.7.10 and 3.2.5 beta)
- Advantages:
  - Increases speed thanks to its Just-in-Time compilation
  - Slightly reduced memory usage due to optimized GC
- Check features and compatibility on `http://pypy.org`
- Very easy to use: just run `pypy` instead of `python`

Embedding Python

- Include Python code in a C/C++ application
- Very useful for:
  - User defined routine
  - Extension scripts
- And it is much simpler than you may think

Embedding Python

- Include Python code in a C/C++ application
- Very useful for:
  - User defined routine
  - Extension scripts
- And it is much simpler than you may think

```
#include "Python.h"

void hello_python()
{
	Py_Initialize();
	PyRun_SimpleString("print 'Hello World'");
	Py_Finalize();
}

void main(){
  hello_python();
}
```

A (slightly) More Sophisticated Example

- Executing Python code is useful when it can interact with the rest of the application

- For example:
  - A Python script that modify an image in Gimp
  - A Python script that implement the artificial intelligence of a monster in a game
  - …

- This can be done by combining what we already saw about Python extensions with an embedded interpreter

```c
#include "Python.h"

static PyObject*
emb_hello(PyObject *self, PyObject *args)
{
    printf("Hello From C\n");
    return Py_BuildValue("i", 1);
}

static PyMethodDef EmbMethods[] = {
        {"emb_hello", emb_hello, METH_VARARGS, ""},
        {NULL, NULL, 0, NULL}
};

void hello_python()
{
      FILE* f;
      f = fopen("script.py","r");
      Py_Initialize();
      Py_InitModule("emb", EmbMethods);
      PyRun_SimpleFile(f, "script.py");
      Py_Finalize();
}

void main(){
  hello_python();
}
```

Python extension to allow the script to interact with the rest of the application

Embed the Python interpreter to execute external scripts

```c
#include "Python.h"

static PyObject*
emb_hello(PyObject *self, PyObject *args)
{
    printf("Hello From C\n");
    return Py_BuildValue("i", 1);
}

static PyMethodDef EmbMethods[] = {
        {"emb_hello", emb_hello, METH_VARARGS, ""},
        {NULL
};

void hello_py
{
      FILE* f
      f = fop
      Py_Init
      Py_InitModule("emb", EmbMethods);
      PyRun_SimpleFile(f, "script.py");
      Py_Finalize();
}

void main(){
  hello_python();
}
```

```python
#script.py

import emb;

print 'Hello From Python'
emb.emb_hello()
```