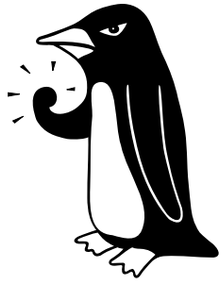
ctypes



Warwick RSE

27/3/2019

What is ctypes?

- Shared object files are standardised ways of providing compiled library code

- If you know the types that each function in the library expects then you can call the function

- Typically you call them from C code and use the header files from the source that generated the library to provide those types

- You don’t have to have anything to do with the original code to do so

What is ctypes?

- Python allows you to call these shared object files using the “ctypes” library

- On Windows you can also call Windows DLL files in the same way, but we’re going to talk about Linux shared objects here

- Not actually as daunting as it sounds, just have to describe the types that are used

- Similar, but different to doing it for Numba

What is ctypes?

- ```
  from ctypes import CDLL
  libc=CDLL('libc.so.6')
  #Can't use libc.so since symlink
  a=libc.printf("Example print %i\n",10)
  ```

- Output is "Example print 10"

- This is actually using the routine underlying "printf" in C code

- For printf Python's attempt to infer type from my parameters worked, but in general it won't
  ```

C code for gravity forces

```c
#include <math.h>

void pairforce(double x1, double y1, double x2, double y2, double m1,
    double m2 , double *f)
{
  double r = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))
      + 1.0e-9 ;
  f[0] = -m1*m2*(x1-x2)/(r*r*r);
  f[1] = -m1*m2*(y1-y2)/(r*r*r);
}
```

- Compile with “gcc -fPIC -shared pairforce.c -o pairforce.so”
- Doesn’t really matter why, but there’s lots of descriptions on the internet of how to build so files

What is ctypes?

- ```
  from ctypes import *
  so=CDLL('./pairforce.so')

  so.pairforce.argtypes=[c_double, c_double,
  c_double, c_double, c_double, c_double,
  POINTER(c_double)]

  result = (c_double*2)()
  so.pairforce(1.0,1.0,0.0,0.0,1.0,1.0, result)
  ```

- "argtypes" for a function in an SO specifies the types of arguments that are supplied when the function is called

- There is also restype which specifies the return type of a function (which pairforce doesn't have)
  ```

Common Types in ctypes

- c_char - Interprets as character
- c_char_p - C character pointer array
- c_double
- c_float
- c_int
- c_long
- c_long_long
- c_void_p - Void pointer

Common Types in ctypes

- POINTER - Describes a pointer to an item
  - POINTER(c_int) is pointer to a C integer
- Structure - Class which is used as the base for defining C structs

Creating C types in Python

- Once you have defined the argtypes Python will do most of the conversions for you, just supply things that it knows how to convert

- Have to be a bit more careful with the POINTER argument

- Since the return is a two element array we have to create this

  - This is done by the line **`result = (c_double * 2)()`**

  - If you wanted to give the items values you would put them in the second brackets comma separated

Creating C types in Python

- What about if you had a pointer to a single item rather than an array?

- You can do **`result = (c_double * 1)()`**

- Can also do

```
a = ctypes.c_double(1.0)
result = ctypes.pointer(a)
```

Creating C types in Python

- Can also go directly from numpy arrays if you want
- Have to create them specially so it's a bit of a faff

```
import numpy as np
import ctypes
array = np.zeros([10,10], dtype=ctypes.c_double)
ptr_to_double = ctypes.POINTER(ctypes.c_double)
c_array_ptr = array.ctypes.data_as(ptr_to_double)
```

- But it does work well for mixing numpy and C code

What's the problem?

- Oddly, performance

- The performance of the hybrid version of the planets code that calls the pair_force function is better than the native Python code but substantially worse than the Numba or C code

- Why?

- The overhead of calling the shared object function is quite high

- Have to be careful about where you switch from Python to C to get good performance

Where to switch?

- You want as much of your code as possible in C without compromising on ease of use or ease of writing

- In this case probably the best solution is to have the specification of initial conditions in Python but the integration in C

- The C code works using structs to represent planets.

    - Want to match in Python

Matching a C structure in Python

```
struct planet{
  double *x, *y;
  double *vx, *vy;
  double mass;
};
```

```
class planet_c(Structure):
    _fields_ = [('x', POINTER(c_double)),
                ('y', POINTER(c_double)),
                ('vx', POINTER(c_double)),
                ('vy', POINTER(c_double)),
                ('mass', c_double)]
    def __init__(self, count):
        self.x = (c_double * count)()
        self.y = (c_double * count)()
        self.vx = (c_double * count)()
        self.vy = (c_double * count)()
        self.mass = np.float64(1.0)
```

Creating the class

- Class descended from ctypes.Structure
- Set the _fields_ parameter to a list of tuples describing the fields in the C structure using ctypes type definitions
- Then use the __init__ function to actually create the fields to match
- Create instances in much the same way

```
plist = [planet_c(nits+1) for x in range(nplanets)]
planets = (planet_c*nplanets)(*plist)
```

How's performance now?

- Performance now is almost identical to the raw C code
- About 10x faster than the Numba result
- But all of the initial conditions etc. are specified in Python and you can use matplotlib to plot the graphs
- Fastest that you are going to get while retaining convenience

Isn’t this a lot of work?

- Not too bad really
- But a lot of it can be automated
- Various libraries for automating/semi automating generation of interfaces
- SWIG - Can generate interface files for linking C code to Python, TCL, JS, Perl, PHP, R and suchlike
    - Wrappers describe interfaces completely. Just call

Isn’t this a lot of work?

- Numpy F2Py - Link Fortran code to Python, generates interfaces automatically
    - Doesn’t support many modern Fortran Features
- F90Wrap - Extended version of F2Py by Warwick’s own James Kermode
    - Supports much more modern Fortran

Other options

- Can write actual Python C/Fortran libraries
    - Can manipulate all Python datatypes etc.
    - Can create own Python datatypes
    - Lot of work!
    - Doesn’t work outside of Python (mostly)

Other options

- Can embed Python interpreter in C/Fortran code
    - If your code is mostly in compiled code but you want the flexibility of Python to specify initial conditions etc. then you can write a normal C/Fortran code and put Python inside it
- Also a lot of work
- Python is quite hard to embed like this
    - Lua is much easier

Main limitation

- In all of these cases you still have to write your core code in a compiled language

- Depending on what you are doing this might be almost as easy as Python (compare most of the C and Fortran examples in this course with their native Python equivalents) or much harder than in Python

- But if you want speed there is no substitute for compiled code

    - Except for writing raw assembly code