DIEE - Università degli Studi di Cagliari

Programmazione Orientata agli Oggetti e Scripting in Python

Python & C/C++

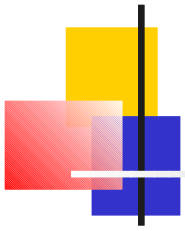
Alessandro Orro – DIEE Univ. di Cagliari

Introduzione

- In questa lezione vedremo come è possibile far interaggire il linguaggio Python con un linguaggio a basso livello come il C/C++.
  In particolare:
  - *Embedding*
    incorporare il Python in un programma C/C++
  - *Extending*
    creare nuovi moduli Python in C/C++
  - *Wrapping*
    interfacciare elementi C++ con elementi Python

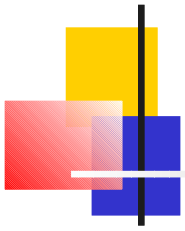
Introduzione

- Difficoltà:
  - Tipi
  - Chiamate di funzione
  - Sintassi: parole chiave (in) e operatori (=, ++)
  - Oggetti/Riferimenti/Puntatori
- Strumento:
  - Tutto questo è facilitato dal fatto che l'interprete Python è scritto in C ed esporta sia le dichiarazioni (*API - Application Programming Interface*) che le librerie.



Header Python.h

- Il file **`Python.h`** contiene le dichiarazioni di tipi/funzioni che vengono utilizzate per richiamare le funzionalità dell'interprete Python.
  - Rappresenta l'interfaccia software fra Python e C
  - E' indipendente dalla piattaforma e dal sistema operativo (nei limiti del possibile!)
  - L'header **`Python.h`** ha sempre questo nome e si trova nella directory **`'PythonPath\include'`**

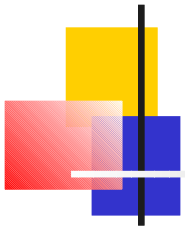


Libreria libpython

- La libreria del Python contiene in forma binaria (compilata) tutte le funzionalità dell'iterprete
  - Rappresenta tutto l'ambiente Python
  - E' dipendente dalla piattaforma e dal sistema operativo
  - E' distribuita in forme differenti a seconda della versione del Python, del sistema operativo e del tipo di libreria (statica, dinamica)
    - Libreria dinamica: **`python23.dll`** (windows), **`libpython23.so`** (linux)
    - libreria statica: **`python23.lib`**
    - directory **`'PythonPath\libs'`**

Embedding

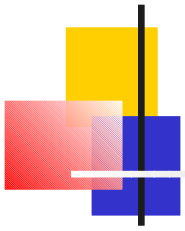
- **`Py_Initialize();`**
  - Inizializza l'interprete creando i moduli fondamentali del Python (**`__builtin__`**, **`__main__`**, **`sys`**).
- **`Py_Finalize();`**
  - Chiude l'interprete dellocando la memoria per tutti i moduli e gli oggetti eventualmente creati.
- **`PyRun_SimpleString(char *source);`**
  - Manda una stringa contenente sorgente python all'interprete che la esegue.
- **`PyRun_SimpleFile(FILE *f, char *fname);`**
  - Manda un file contenente sorgente python all'interprete che lo esegue.



Embedding

```
#include <Python.h>

int main(int argc, char *argv[]) {
  Py_Initialize();
  PyRun_SimpleString(
    "import re\n"
    "L=[1,2,3]\n"
    "L+=[4]\n"
    "print L" );
  Py_Finalize();
  return 0;
}
```



Embedding

- L'esempio va compilato con un normale compilatore C (es. mingw per windows)
  - includere `'...\Python23\include'`
  - includere `'...\Python23\libs'`
  - linkare con `'python23.dll'`

```
cc -L"c:\Python23\libs" -lPython23
  -I"c:\Python23\include" prova.cpp -o
  prova.exe
```

Embedding

- Il programma prova.exe appena creato utilizza la libreria esterna (dinamica) **`python23.dll`**
  - **`python23.dll`** deve essere in un path visibile per esempio in **`'c:\WINDOWS\system32'`**

```
> prova.exe
[1, 2, 3, 4]
```

  - Questo metodo può essere usato anche per creare un eseguibile da un programma python anche se per questo esistono tool specifici.

Embedding

```
#include <Python.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  char fname[]="prova2.py";
  FILE *f = fopen(fname,"r");
  if (f==NULL) {
    printf("impossibile trovare %s\n",fname);
    return -1;
  }
  Py_Initialize();
  PyRun_SimpleFile(f, fname);
  Py_Finalize(); fclose(f);
  return 0;
}
```

Extending

- Per creare nuovi moduli scritti in C si utilizza ancora l'interfaccia API del Python.
- Le funzioni che si usano per estendere il Python sono le stesse che si usano per incorporare il Python dentro il C.
  - Vedremo in dettaglio solo le estensioni
  - L'embedding è più raro.

Extending

Un'estensione può essere fatta in due modi:

- Creazione a mano
  - si ha un controllo maggiore del processo
  - è un lavoro noioso e ripetitivo
- Utilizzo di software specifici:
  - non si ha un controllo completo in quanto i tool hanno i loro limiti
  - gran parte del lavoro può essere automatizzato

Extending

- Vediamo prima un esempio “a mano”: modulo che implementa alcune funzioni (**`helloworld.c`**)
- Occorre compilare il file e creare una libreria dinamica (**`helloworld.dll`**) specificando le solite librerie e path del Python. Vedere il **`Makefile`**.
- A questo punto **`helloworld.dll`** può essere importato in un programma Python come se fosse un normale modulo Python.
  - Uso l'istruzione **`import helloworld`**
  - Eventualmente devo settare i path corretti

Extending

- Utilizzo del modulo

```
>>> import helloworld
>>> helloworld.saluto()
'Ciao!'
>>> helloworld.greeting()
'Hello World!'
>>> helloworld.saluto
<built-in function saluto>
>>> helloworld
<module 'helloworld' from 'helloworld.dll'>
```

Extending

- Tutte le funzioni del modulo **helloworld** vengono trattate come funzioni built-in e non come normali funzioni dichiarate nel sorgente
  - E' come se avessi aggiunto funzionalità all'interprete.
  - Non posso accedee al loro codice
  - Non posso aggiungere dinamicamente attributi

```
>>> import helloworld
>>> helloworld.saluto.A = 100  # errore
```

Extending

- Proviamo a implementare con le C API una funzione **`range`** che emula il comportamento dell'analoga funzione builtin e **`sqrange`** che genera una lista con il valore dei quadrati.
- Mettiamo tutto dentro un modulo **`fast.dll`**

```
>>> import fast
>>> L1 = fast.range(4)
[0,1,2,3]
>>> L1 = fast.sqrange(4)
[0.0,1.0,4.0,9.0]
```

Extending

- implementazione di **range**

```
static PyObject *
range(PyObject *self, PyObject *_N) {
    int N;
    if ( PyArg_ParseTuple( _N, "i", &N) == 0 )
      return NULL;
    PyObject* list = PyList_New(N);
    for(int i=0; i<N; ++i)
      PyList_SetItem(list,i,Py_BuildValue("i",i));
    return list;
}
```

Extending

- implementazione di **sqrange**

```
static PyObject *
sqrange(PyObject *self, PyObject *_N) {
    int N;
    if ( PyArg_ParseTuple( _N, "i", &N) == 0 )
      return NULL;
    PyObject* list = PyList_New(N);
    for(int i=0; i<N; ++i)
      PyList_SetItem(list,i,Py_BuildValue("f",1.0*i*i));
    return list;
}
```

Extending

- Velocità esecuzione:
  - **range** è praticamente equivalente alla builtins
    - Le builtins sono già molto ottimizzate
  - **sqrange** è il 2 volte più veloce rispetto ad una implementazione in “puro Python” e 5 volte più lenta di un'implementazione in “puro C”

```
float v[100000];
int i;
for(i=0; i<100000; ++i)
  v[i]=1.0*i*i;
```

Extending

- Rispetto al programma in puro C, una parte del tempo di esecuzione è usato per lanciare l'interprete e inizializzare l'ambiente
- ... una parte per chiamatare le funzioni
- ... e una parte per la conversione tipi in ingresso/uscita tramite le funzioni
  - **`PyArg_ParseTuple`**
  - **`PyList_New`**
  - **`Py_BuildValue`**

Extending

- Questo sovraccarico è inevitabile se si vogliono avere i vantaggi di un linguaggio ad alto livello come il Python e quelli di uno a basso livello come il C.
- Ci sarebbe lo stesso problema con qualsiasi altra “coppia” di linguaggi.
- All'aumentare della complessità di calcolo questo tempo incide sempre meno sul tempo complessivo di esecuzione.
  - Nelle applicazioni reali è molto conveniente implementare i moduli critici in C/C++.

Extending

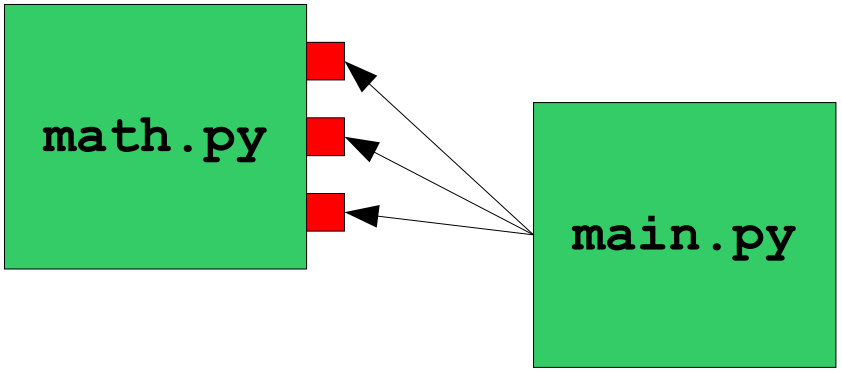
- Il fatto di avere un'interfaccia in Python anche per le funzioni a basso livello mi permette di utilizzare la progettazione prototipale:

  1. Realizzo tutto il sistema in Python (*prototyping*)
  2. Individuo le parti critiche (*profiling*)
  3. Reimplemento le funzioni critiche in C (*extending*)

- Il sistema nel complesso rimane inalterato (stessa interfaccia).
- La libreria standard contiene tutte le funzionalità necessarie per il testing e il profiling.

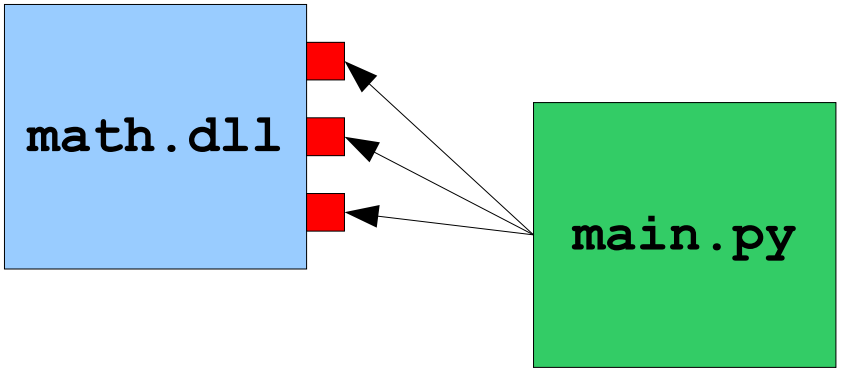
Extending

- Prototipo: tutto in Python



Extending

- Sistema: Python/C



Extending

- E' possibile creare dei moduli completi, cioè con funzioni, oggetti, classi, ecc...
  - Da un punto di vista tecnico le API del Python mettono a disposizione tutto quello che serve.
  - Da un punto di vista pratico la quantità di codice necessaria per creare una classe in Python dal C può essere notevole.
- E' possibile usare strumenti automatici
  - Lo vediamo direttamente con il Wrapping

Wrapping

- Wrapping: creazione dei moduli Python che esportano le funzionalità di moduli C già esistenti
- Si possono usare le API:
  - Nel sorgente C importo il modulo le definizioni del modulo da wrappare (moduloc.h).
  - Compilo e creo la dll come al solito

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

Wrapping

- Gran parte del lavoro può essere fatto con l'ausilio di strumenti specifici:
  - SWIG, Boost Python, PyCXX, ...
  - Automatizzano gran parte del lavoro
  - E' comunque necessaria una certa conoscenza delle API del Python se si vogliono fare cose avanzate.
  - Alcuni preferiscono lavorare a “mano”, anche se comunque vale la pena di impararne almeno uno.
  - In seguito vedremo qualche cenno a SWIG (uno dei migliori).

Wrapping - SWIG

- **S**implified **W**rapper and **I**nterface **G**enerator
  - Sito: *http://www.swig.org/index.html*
  - E' principalmente un generatore automatico di Wrapper ma può essere usato anche per Embedding e Extending
  - Probabilmente è lo strumento più potente (insieme a Boost) disponibile attualmente.
  - Non intrusivo!
  - Gratuito
  - Multipiattaforma: Windows, Unix, Mac
  - Multilingiaggio: Python, Java, Perl, PHP, Ruby

Wrapping - SWIG

- Come si usa:
  1. Parto da un modulo già esistente (sorgente **`mod.cpp`** + **`mod.hpp`** oppure oggetto **`mod.o`**)
  2. Creo un file di interfaccia (**`mod.i`**) che "spiega" al programma SWIG come creare il wrapper.
  3. Lancio SWIG per generare il modulo wrapper (**`mod.py`** e **`mod_wrap.cpp`**)

```
> swig.exe -c++ -python -o mod_wrap.cpp
  mod.i
```

continua ...

Wrapping - SWIG

...

4. Compilo il wrapper **`mod_wrap.cpp`** per generare la libreria dinamica **`_mod.dll`**.

```
> cc -shared mod_wrap.cpp mod.o
  -lswigpy -lPython23 -o _mod.dll
```

per semplicità sono stati omessi tutti i path

- header e libreria del Python
- header e libreria di SWIG

Wrapping - SWIG

- Alla fine nella directory saranno presenti i seguenti file:

  `mod.cpp` **`mod.hpp`** **`mod.i`**
  `mod.o` **`mod.py`** **`_mod.dll`**
  `mod_wrap.cpp`

- Può sembrare troppo complicato ? In realtà no!
  - Posso automatizzare tutta la procedura con un Makefile
  - Sono tutti generati automaticamente, a parte il modulo il C++ (ovviamente) e **`mod.i`**.
  - Non c'e' stato bisogno di modificare il modulo originale né di scrivere dell'altro codice in C.

Wrapping - SWIG

- A questo punto posso importare il modulo **`mod.py`** da un programma Python.
- Rispetto all'approccio manuale non viene importata direttamente la dll **`mod.dll`** ma si passa per un'ulteriore interfaccia **`mod.py`**.
  - Questo permette di gestire alcuni aspetti particolari (puntatori)

Wrapping - SWIG

Esempio: **Point.hpp**

```
struct Point {
    int x,y;
    Point(int _x, int _y);
    void write() const;
};
```

Wrapping - SWIG

Esempio: **Point.cpp**

```
#include <iostream>
#include "mod.hpp"
Point::Point(int _x, int _y)
    : x(_x), y(_y) {}
void Point::write() const {
    std::cout << "(" << x << "," << y;
    std::cout << ")\n";
}
```

Wrapping - SWIG

Esempio: **test_Point.py**

```
from Point import *
point = Point(1,2)
point.write()  # (1,2)
pointlist=[Point(1,2), Point(2,3)]
for point in pointlist:
    point.write()
```

Wrapping - SWIG

- Il wrapper non genera solo la classe **`Point`** ma anche **`PointPtr`** che rappresenta **`Point*`** in C++.
- Quando accedo ad un membro di **`PointPtr`** viene fatto riferimento allo stesso membro di un **`Point`** sottostante

```
point = Point(1,2)
ptr = PointPtr(point)
ptr.write()
```

Wrapping - SWIG

- In questo modo si possono gestire senza problemi anche le funzioni che prendono in ingresso puntatori.
- Per esempio: in C++

```
struct Point {
   ...
   inline static void incr(Point *p) {
     p->x++; p->y++;
   }
};
```

Wrapping - SWIG

- in Python

```
from Point import *

point = Point(1,2)
point.write()   # 1,2
Point.incr(point)
point.write()   # 2,3
```

Wrapping - SWIG

- Le classi generate tramite wrapping possono essere estese dal Python:

```
from Point import *

class Point2(Point):
    def swap(self):
        self.x,self.y=self.y,self.x
Point.incr(point)
point.write()   # 2,3
```

Wrapping - SWIG

- Le regole che SWIG utilizza per generare i wrapper sono contenute nel file **`Point.i`**.
  - E' un file header C/C++ con qualche direttiva aggiuntiva specifica di SWIG.
  - Vediamo solo qualche esempio. Per i dettagli vedere il manuale di riferimento di SWIG.

Wrapping - SWIG

- L'esempio più semplice: wrapping per tutto il contenuto del modulo Point.hpp

Point.i

```
%module Point
%{
#include "Point.hpp"
%}
%include "Point.hpp"
```

Wrapping - SWIG

- **`%module Point`**
  indica il nome del modulo
- **`#include "Point.hpp"`**
  include il file nell'header del wrapper
- **`%include "Point.hpp"`**
  indica che l'header deve essere interpretato da SWIG in modo da generare i wrapper corrispondenti.
  - Al suo posto posso mettere l'elenco delle sole funzioni che voglio esportare (funzionalità avanzata)

Wrapping - SWIG

Altre funzionalità:

- Estensione dell'interfaccia
- Definizioni di template
- Typedef
- Rinominazione: utile per le parole chiave
- Gestione di array e stringhe (char*)
- STL parzialmente supportata

Vediamo solo qualche esempio ...