Marco Barisione

Estendere Python in C

Perché estendere Python?

- La maggior parte del tempo di esecuzione è passata ad eseguire poche funzioni
  - Queste funzioni scritte in Python potrebbero non essere abbastanza veloci
- Utilizzare codice preesistente o librerie esterne
- Interfacciarsi direttamente ad hardware

Perché non usare le API Python?

- Python fornisce una API completa ed estesa, perché non usarla direttamente?
  - Bassa produttività
  - Lavoro noioso e ripetitivo, quindi fonte di errori
  - Serve molto codice per cose stupide
  - Mai fare manualmente un lavoro che può essere fatto altrettanto bene (se non meglio) da un tool automatico

SWIG

- Cos'è SWIG?
  - SWIG = "Simplified Wrapper and Interface Generator"
  - Parte da dichiarazione in C per generare il codice di interfaccia con Python
  - Facile da usare
  - Quasi tutto automatico
  - Può essere direttamente richiamato da make (o da un programma analogo)
  - Si può scrivere codice C senza preoccupari dell'integrazione con Python
  - Parte da un file sorgente C (".c"), da un header (".h") o, meglio, da un file di interfaccia (".i")

Come lavora SWIG

- Si scrive la libreria in C (“modulo.c” ad esempio)
- Si scrive un file di interfaccia (“modulo.i”)
- Si esegue SWIG sul file di interfaccia (che produce “modulo_wrap.c” e “modulo.py”)

```
swig -python modulo.i
```

- Si compilano i sorgenti e si linkano in un file “_modulo.so” su Unix o “_modulo.pyd” su Windows
  - Il file sorgente “modulo.c” può essere linkato staticamente, cioè direttamente nel file di estensione
  - Dinamicamente, cioè in una altra libreria esterna.
    - Utile in particolare se la libreria deve essere usata da altre applicazioni non Python o non sono disponibili i sorgenti

Perché usare un file di interfaccia

- SWIG può essere egeguito direttamente su un header o un file sorgente C
  - Potrebbero essere esportate funzioni che non serve utilizzare da Python
  - SWIG non è un compilatore, potrebbe non gestire correttamente parte della sintassi del C
- In caso si decida comunque di utilizzarlo su un header o sorgente si possono inserire istruzioni specifiche

```
#ifdef SWIG /* SWIG definisce questa macro */
    /* Codice specifico di SWIG */
#endif
```

Un primo esempio (1)

- Supponiamo di avere un file sorgente C (“somma.c”) contenente solo una funzione somma

```
#include "somma.h"
int somma(int a, int b){return a + b;}
```

- E il relativo header (“somma.h”)

```
extern int somma(int a, int b);
```

- Ora è necessario scrivere il file di interfaccia

```
%module somma
%{
#include "somma.h"
%}
extern int somma(int a, int b);
/* Si poteva usare %include "somma.h" */
```

Un primo esempio (2)

- Le linee che iniziano con “%” sono comandi specifici di SWIG
- “%module somma” indica a SWIG che il nome del modulo (quello poi usato da Python) sarà “somma”
- Il codice racchiuso fra “%{” e “%}” viene incluso direttamente in “somma_wrap.c”
- Vista la semplicità del codice si poteva includere direttamente l’header con “%include”
- Vi è un’importante differenza fra i due “include”
  - “%include” include un file affinché sia interpretato da SWIG
  - “#include” serve per fare includere l’header dal compilatore, questo necessita, infatti, di conoscere il prototipo delle funzioni in “somma_wrap.c”

Un primo esempio (3)

- Ora si può procedere alla creazione del modulo, ad esempio utilizzando il seguente makefile

```
somma.so: somma.i
    swig -python somma.i
    gcc -shared -o _somma.so somma.c \
        somma_wrap.c
clean:
    rm somma_wrap.c somma.py _somma.so *.o
```

- Ovviamente il file necessari, come “Python.h”, devono essere trovabili da gcc.
- I moduli devono essere compilati con lo stesso compilatore usato per compilare l’interprete

Un primo esempio (4)

- Il file ".so" o ".pyd" generato contiene le interfacce di basso livello
- Il file ".py" può essere usato da Python come un normale modulo

```
>>> import somma
>>> print somma
<module 'somma' from 'somma.py'>
>>> dir(somma)
['__doc__', '__file__', '__name__', 'somma']
>>> print somma.somma(5, 3)
8
```

Variabili e costanti (1)

- È possibile accedere da Python anche a variabili globali C e tradurre “#define” ed enumerazioni in variabili
  - File sorgente “var.c”

```
#include "var.h"
int a = 42;
```

  - File header “var.h”

```
#define X 12
enum{ZERO, UNO, DUE};
int a;
```

  - File di interfaccia “var.i”

```
%module var
%{#include "var.h"%}
%include var.h /* SWIG esamina l’header */
```

Variabili e costanti (2)

- Ora è possibile compilare il file ed usare il modulo

```
>>> import var
>>> print var.ZERO, var.UNO, var.DUE
0 1 2
>>> print var.X
12
>>> print var.cvar.a
15
>>> var.ZERO = 42
>>> print var.ZERO
42
```

Variabili e costanti (3)

- Le enumerazioni e le “#define” diventano nuove variabili Python
  - Non costanti!
  - Modifica solo per Python, nel modulo C non cambiano
    - Infatti non sono variabili ma sono espanse dal preprocessore

- L’accesso alle variabili avviene attraverso “nome_modulo.cvar.nome_variabile”
  - In caso di modifica il loro valore è modificato anche per il codice C

Variabili immutabili

- Una variabile globale può essere resa costante dichiarandola come “const” o con “%immutable”

```
%immutable;    // Le variabili seguenti
               // sono costanti
int a;
%mutable;      // Le variabili seguento sono
               // non costanti
```

Oppure

```
const int a;  // Solo questa costante
```

- Un tentativo di assegnamento ad una variabile costante produce un’eccezione

Tipi base in SWIG

- I tipi base sono interpretati correttamente da SWIG e tradotti nel tipo Python equivalente
- I puntatori sono tradotti in stringhe contenenti anche informazioni sul tipo
  - Comunque Python non potrebbe usare direttamemente i puntatori
  - Il codice generato da SWIG può effettuare controlli sui tipi
  - Esempio:
    - int* → "_1008e124_p_int"
    - double** → "_f8ac _pp_double"
    - char** → "_10081012 _pp_char"
    - double*** → "_f8bc _ppp_double"

Tipi “complessi” in SWIG

- Tutti gli altri tipi (strutture, array ecc.) sono considerati puntatori
  - Nel file di interfaccia non è necessario definire le strutture
  - Avviene quasi sempre la cosa giusta
  - In alcuni casi è necessario usare “typedef”
    - C: `void foo(size_t num);`
    - Python: `foo(40)` → **Errore!** (si aspettava _p_size_t)
    - Per risolvere: `typedef unsigned int size_t;`
    - Come per “%include” anche “typedef” è un’istruzione solo per SWIG, il compilatore deve comunque sapere le eventuali “typedef”

Strutture (1)

- Se le strutture sono definite nell'interfaccia allora sono trasformate in classi Python

point.h	point.i
`struct Point{`	`%module point`
`    int x, y;`	`%{#include "point.h"%}`
`};`	`%include point.h`

```
>>> import point
>>> p = point.Point()
>>> p.x = 13
>>> print p.x, p.y
13 0
```

Strutture (2)

- In C non è possibile avere funzioni in una struct, in SWIG sì

```
%module point
%{
#include "point.h"
%}
%include point.h
%extend Point{
    struct Point __add__(struct Point *other){
        struct Point ret;
        ret.x = self->x + other->x;
        ret.y = self->y + other->y;
        return ret;
    }
};
```

Strutture (3)

- Il costruttore in SWIG (come in C++) ha il nome della struttura
- Il distruttore in SWIG ha il nome della struttura preceduto da “~”
- È frequente che esistano funzioni che svolgano la funzione di metodi. Queste possono essere aggiunte automaticamente se seguono determinate regole

```
%extend Strutt{
    Strutt();   /* Chiama "new_Strutt()" */
    ~Strutt();  /* Chiama "delete_Strutt()" */
    metodo();   /* Chiama "Strutt_metodo()" */
};
```

Parametri di ingresso ed uscita (1)

- In C è frequente usare degli argomenti come parametri di uscita o di ingresso

```
void add(int x, int y, int *r){
    *r = x + y;
}

int sub(int *x, int *y){
    return *x - *y;
}

void negate(double *x){
    *x = -(*x);
}
```

Parametri di ingresso ed uscita (2)

- Tutto ciò può essere reso più aderente al modo di gestire gli argomenti di Python

```
%module inout
%{#include "inout.h"}
%include "typemaps.i"
void add(int, int, int *OUTPUT);
int sub(int *INPUT, int *INPUT);
void negate(double *INOUT);
```

- Ed usato quindi come
  - `a = inout.add(3,4)` → a = 7
  - `b = inout.sub(7,4)` → b = 3
  - `c = inout.negate(4)` → c = - 4

Array

- Gli array sono considerati come puntatori

- Non vi è conversione automatica con le liste o un'altro tipo simile poiché:
  - Pesanti problemi di prestazioni se l'array è troppo grande
  - In C non c'è modo di conoscere le dimensioni di un array
  - Il C fa poca distinzione fra puntatori e array
    - La distinzione è nulla se gli array sono passati a funzioni

Unbounded C arrays

```
int sum(int a [], int n){
    int i, sum = 0;
    for (i = 0; i < n; ++i){sum += a[i];}
    return sum;
}

%include "carrays.i"
%array_class(int, array_interi);

a = array_interi(5)
for i in range(5): a[i] = i
sum(a, 5)                                  → 10
```

- Non c'è bound checking

Puntatori a char (char *)

- Se l'input di una funzione C è un "char *" allora si può passare una stringa Python come argomento

```
/* C */
void foo(char *s);
# Python
foo("Hello")
```

  - La stringa passata non può essere modificata e non può contenere "\0"

- Se una funzione C ritorna un "char *", questo viene copiato in una stringa Python
  - Il valore ritornato non può contenere "\0"
  - Attenzione ai memory leak

Passare stringhe binarie

```
/* C */
void foo(char *str, int len);


/* Interfaccia */
%apply (char *STRING, int LENGTH) {
    (char *str, int len)
};
int foo(char *str, int len);


# Python
foo("Hello\0World")
```

Rilasciare la memoria

- Se un oggetto ritornato è stata creato con “malloc” deve essere anche distrutto con “free”
- È possibile rendere l’operazione automatica con “%newobject”

```
/* C */
char *foo(){
    char *ret = (char *)malloc(...);
    return ret;
}

/* Interfaccia */
%newobject foo;
char *foo();
```

Rinominare le funzioni

- È possibile rinominare le funzioni in modo che il loro nome sia diverso in Python da quello C
  - Il nome della funzione C potrebbe essere una keyword in Python
  - Il nome C potrebbe avere un prefisso per evitare collisioni che in Python possono essere evitate con i moduli
    - È infatti tipico vedere funzioni C con nomi come “mia_libreriaStampa()”

```
/* Interfaccia */
%rename(my_print) print;
extern void print(char *);
```