

# Python Intermedio

Settembre 2006



## Funzioni

- Se non si sono mai viste si possono considerare il modo piu' semplice per poter creare del codice riutilizzabile.

## Classi

- Senza voler entrare in termini tecnici si possono considerare come dei blocchi di codice indipendente.  
In altre parole contengono sia le variabili (attributi)  
sia il codice per elaborarle (metodi)

## Eccezioni

- Gestione degli errori



Venezia Free Software

Loris Michielutti  
Loris Michielutti

Users Group





### Codice normale:

- Immaginiamo di dover calcolare i numeri di Fibonacci minori di 10

```
a, b = 0, 1
while b < 10:
    print b
    a, b = b, a+b
```

### Programmazione funzionale

- se volessimo adesso ricalcolare i numeri di Fibonacci minori di 20 dovremmo riscrivere lo stesso codice.

- allora riscriviamo il tutto per poter utilizzare lo stesso codice piu' volte

```
def Fib(n):
    """ stampa la serie di Fibonacci """ # questa e' una docString
    a, b = 0, 1 # (help Fib) ritorna la docString
    while b < n:
        print b
        a, b = b, a+b
```

```
# se omettiamo l'istruzione di return verra' ritornato il valore di default none
# (si usa definire procedura una funzione che non torna valore !)
```

- mettiamo in uso la funzione

```
Fib(20)
```



### Funzionale con ritorno di valori

- se volessimo ritornare una lista basta cambiare il codice in:

```
def Fib(n):  
    """ stampa la serie di Fibonacci  
    ma ritorna la lista dei valori """  
    lis = [] # questa e' la lista vuota  
    a, b = 0, 1  
    while b < n:  
        lis.append(b) # qui aggiungiamo i valori nella lista  
        # print b # commentato  
        a, b = b, a+b  
    return lis
```

- mettiamo in uso la funzione

```
f = Fib(50)
```

- si possono definire funzioni con un numero variabile di parametri  
e ci sono tre modi di farlo:



### Funzioni con piu' parametri in ingresso

- una cosa molto utile e' definire dei parametri di default:

```
def richiesta_conferma(prompt, num=4, ris='Si o No, prego!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('s', 'si', 'S', 'Si?'): return True
        if ok in ('n', 'no', 'N', 'No'): return False
        num -= 1
        if num < 0:
            raise IOError('errore Utente')
        print ris
```

- mettiamola in uso

```
richiesta_conferma('Vuoi realmente uscire?')
```

- attenzione si puo' operare sui parametri della funzione:  
o per posizione o per nome!!
- attenzione che i valori di default vengono valutati solo una volta quando la funzione e' creata!!!
- i valori di default devono essere definiti per ultimi!!!



### Funzioni con numero di parametri arbitrari in ingresso

- questo e' l'ultimo modo ma non per questo il meno usato

```
def fprintf(file, format, *args):          # *args non e' altro che una tupla
    file.write(format % args)
```

- se si vuol passare una lista di default: che si accumula:

```
def f(a, L=[]):
    L.append(a)                            # print f(1) risponde con [1]
    return L                               # print f(2) risponde con [1, 2]
```

- se invece si vuol passare una lista di default: che non venga accumulata:

```
def f(a, L=None):
    if L is None:                          # print f(1) risponde con [1]
        L = []                             # print f(2) risponde con [2]
    L.append(a)
    return L
```

- se si vuol passare un dizionario:

```
def f(**b):
    print(b)                               # la mettiamo in uso f(pippo=10)
```



### Fondamenti:

- un oggetto ha due parti fondamentali

#### Attributi

- contengono i dati dell'oggetto (variabili)

#### Metodi

- sono le funzioni con accesso implicito

- una **Classe** definisce una struttura (“Attributi, Metodi”) e permette di creare oggetti
- un oggetto della stessa classe condivide i Metodi e la stessa Lista di Attributi (ma non gli stessi valori, se non sono di tipo statico)

### Definizione:

```
class Rettangolo(object):  
    """ Questa e' la solita DocString """  
    def __init__(self, base, altezza):  
        """ questo e' il costruttore """  
        self.base = base  
        self.altezza = altezza  
    def area(self):  
        """ calcola l'area """  
        return self.base * self.altezza
```

# (object) nuovo modo per definire  
# una classe madre  
# questo e' un metodo speciale  
# viene chiamato ogni qualvolta si istanzia  
# un oggetto, serve ad inizializzare i  
# i parametri  
# questo e' un metodo della classe



### Ereditarieta':

- le Classi costituiscono una gerarchia e possono derivare da altre classi ereditando metodi e attributi. (La classe madre e' detta anche superclasse)
- si puo' ereditare anche da piu' classi contemporaneamente (eredita' multipla)

### Classe base:

```
class Geom(object):  
    """ classe base di geometria """  
    def desc(self):  
        """ descrivo i miei attributi """  
        print "la mia Area e' %s" % (self.area())
```

### Classe derivata:

```
class Rett(Geom):  
    """ Questa classe e' derivata dalla superclasse Geometria """  
    def __init__(self, base, altezza):  
        """ questo e' il costruttore """  
        self.base = base  
        self.altezza = altezza  
    def area(self):  
        """ calcola l'area """  
        return self.base * self.altezza
```

# Classi



altra Classe derivata:

```
class Cerchio(Geom):
```

```
    """ Questa classe e' derivata dalla superclasse Geometria """
```

```
    def __init__(self, raggio):
```

```
        self.raggio = raggio
```

```
    def area(self):
```

```
        """calcola l'area"""
```

```
        import math
```

```
        return math.pi * self.raggio * self.raggio
```

Usiamo le classi derivate:

```
>> import fGeom
```

```
# richiamo il file dove abbiamo definito le classi
```

```
>> r = fGeom.Rett(2,3)
```

```
# istanzio l'oggetto rettangolo
```

```
>> c = fGeom.Cerchio(10)
```

```
# istanzio l'oggetto cerchio
```

```
>> r.desc()
```

```
# richiamo il metodo derivato !!!
```

```
la mia Area e' 6
```

```
>> c.desc()
```

```
# richiamo il metodo derivato !!!
```

```
la mia Area e' 314.15.....
```



### Polimorfismo:

- la possibilita' che l'esecuzione di uno stesso metodo abbia effetti diversi in base all'oggetto che deve trattare.
- abbiamo visto che l'oggetto rettangolo richiama il calcolo dell'area nello stesso modo dell'oggetto cerchio!!!  
pero' ognuno opera in modo diverso

### Override (sovrapposizione):

- una Classe derivata (sottoclasse) puo' implementare il proprio metodo ed avere accesso anche al metodo originale

### Metodi Virtuali:

- quando si dichiara un metodo senza fornire l'implementazione l'implementazione e' affidata alla classe figlio.

### Metodi e Variabili Private:

- in python si definiscono privati tutti gli oggetti che iniziano con 2 underscore e finiscono senza (es: `__mioParametro`)  
se finiscono con 2 underscore sono considerati speciali (es: `__init__`)  
ma sono raggiungibili anche al di fuori della classe!!!



### Eccezioni :

- Condizioni particolari in cui si deve decider se interrompere l'esecuzione e informare l'utente.
- esempi:
  - si tenta di leggere un file inesistente
  - si cerca di dividere un numero per zero

### Costrutti:

`try:`

- si tenta di eseguire un blocco di codice

`except:`

- il blocco di codice che gestisce l'eccezione

`raise:`

- si solleva una eccezione o si riattiva quella intercettata



### Esempio

```
import sys
```

```
try:
```

```
    f = open("mioFile.txt")
```

```
    s = readline()
```

```
    i = int(s.strip())
```

```
except IOError(errno, strerror):
```

```
    print "Errore I/O (%s): %s" %(errno, strerror)
```

```
except ValueError:
```

```
    print "Non si puo' convertire il dato in un intero"
```

```
except:
```

```
    print "Errore inatteso:" sys.exc_info()[0]
```

```
    raise
```

```
# apro il file
```

```
# leggo una linea
```

```
# converto in intero il dato
```

```
# gestione di una particolare
```

```
# eccezione con parametri
```

```
# gestione di una particolare
```

```
# eccezione senza parametri
```

```
# gestione generica
```

```
# passo la gestione al sistema
```