Python Parte 3: Moduli, I/O, Eccezioni

Parte del ciclo di seminari su

Programmazione Orientata agli Oggetti
e
Scripting in Python

a cura di:
Giancarlo Cherchi

Introduzione

- Moduli
- Input/Output
- Eccezioni

Moduli

- Un *modulo* è un file contenente definizioni e istruzioni Python
- Il file ha lo stesso nome del modulo con in più l'estensione ".py"
- Il nome del modulo è disponibile (come stringa) nella variabile globale __*name*__
- Le definizioni di un modulo possono essere importate in un altro modulo

Import

- Per importare un modulo all'interno di un altro modulo si usa:

import *nome_modulo*

- Attenzione! L'istruzione non dà accesso diretto ai simboli definiti nel modulo: occorre invece utilizzare *nome_modulo* come prefisso

Import

- Ad esempio, creando il file “modulo.py”:

```
def somma(a, b):
    return a+b
```

- Per utilizzare la funzione occorre scrivere:

```
import modulo
modulo.somma(3,4) #somma(3,4) è errato
```

Import

- Se si intende utilizzare spesso una definizione importata, è possibile assegnarla ad una variabile locale:

  prova = modulo.prova

- In questo modo si può evitare di ripetere ogni volta il prefisso “modulo.”

Import

- Digitare l'istruzione *import* nell'interprete equivale a immettere tutte le righe contenute nel file che si sta importando (inserendo però i simboli in un namespace che ha lo stesso nome del file)
- Saranno quindi eseguite delle eventuali istruzioni presenti nel file (che hanno generalmente il fine di inizializzare il modulo)

Import

- Ad esempio, se il file modulo.py contiene:

```
print "Carico il modulo"
def f():
    return 0
```

- Si ottiene:

```
>>> import modulo
Carico il modulo
>>>
```

Import

- L'istruzione *import* all'interno di un file, importa le definizioni di un modulo in altro modulo
- E' possibile nidificare le importazioni ed avere importazioni "incrociate"
- A differenza del C / C++ non è necessaria l'inclusione condizionale con #ifndef ma ogni modulo è importato una volta sola!

Import

- #file1
  a = 1
  import file2
  print file2.a

- #file2
  a = 2
  import file1
  print file1.a

- E’ possibile, ad esempio digitare:
  >>> import file1
  1
  2
  >>>

Import

- Eventuali istruzioni all'interno di un modulo saranno eseguite soltanto la prima volta che un modulo è importato
- Ogni modulo ha la sua "symbol table" che contiene i simboli "globali" per tutte le funzioni definite nel modulo
- E' possibile accedere ai simboli globali di un altro modulo utilizzando come prefisso il nome del modulo desiderato

Import

- E’ possibile importare direttamente nella symbol table corrente i simboli di un modulo:

  from *modulo* **import** *simb1*, *simb2*

- In questo modo *simb1* e *simb2* saranno visibili senza usare il prefisso “modulo.”

Import

- E’ possibile anche importare nella symbol table corrente tutti i simboli definiti all’interno un modulo:

  from *modulo* **import** *

- In questo modo saranno visibili direttamente nel namespace corrente tutti i simboli tranne quelli che iniziano per “_”

Path

- Dove vengono cercati i file da importare dall'interprete?
- Prima nella directory corrente, poi nei percorsi definiti nella variabile d'ambiente PYTHONPATH (se definita)
- La variabile *sys.path* contiene tutti i percorsi in cui l'interprete cercherà i file da importare

Moduli Standard

- Ogni interprete Python ha alcuni moduli definiti internamente
- Uno dei più importanti è il modulo *sys*, che dà accesso ad alcune variabili/funzioni che permettono di interagire con l'interprete
- Esempio:

```
import sys
sys.ps1 = 'Nuovo Prompt! '
```

Funzione dir()

- La funzione

  dir(*modulo*)

  rende la lista ordinata dei simboli esportati da un modulo

- Se non si specifica il nome del modulo saranno visualizzati i simboli al momento definiti

Funzione dir()

- dir() non elenca le variabili e/o funzioni definite internamente

  E’ però possibile ottenere questo risultato, digitando:

  import __builtin__
  dir (__builtin__)

Package

- I package consentono l'organizzazione gerarchica dei moduli in sottomoduli separandone i nomi con dei punti
- Ad esempio, si potrebbe progettare un insieme di moduli per la gestione/conversione di/tra vari formati audio, costituito da un modulo Sound di base e una serie di sottomoduli

Sound Package

- Sound/
  - __init__.py
  - Format/
    - __init__.py
    - wavread.py
    - wavwrite.py
    - mp3read.py
    - mp3write.py
  - Effects/
    - __init__.py
    - echo.py
    - reverber.py
    - chorus.py
  - Filters/
    - __init__.py
    - 3bandeq.py
    - karaoke.py

Sound Package

- I file __init__.py indicano all'interprete che le directory contengono dei package e possono essere vuoti o contenere istruzioni di inizializzazione
- E' possibile importare singoli moduli da un package:

  import Sound.Effects.echo

  in questo modo le funzioni si referenziano:

  Sound.Effects.echo.echofilter(delay=0.7)

Sound Package

- Un modo alternativo per importare il sottomodulo è:

  from Sound.Effects **import** echo

  in questo modo la funzione si referenzia:

  echo.echofilter(delay=0.7)

Sound Package

- Un’altra possibilità è quella di importare direttamente i simboli dal sottomodulo:

  from Sound.Effects.echo **import** echofilter

  così la chiamata di funzione è diretta:

  echofilter(delay=0.7)

Package

- In definitiva, nell'istruzione

  from *package* **import** *item*

  item può essere o un sottomodulo (o sottopackage) di un package o un simbolo definito nel package
- *import* controlla se *item* è definito, altrimenti assume che è un modulo e cerca di caricarlo

Package

- Invece, nell'istruzione

  import *item.subitem.subitem*

  a parte l'ultimo *subitem,* che può essere o un modulo o un package (ma non una funzione, variabile o classe), tutti gli altri devono essere dei package

Importare tutto un Package

- L'istruzione

  from *package* **import** *

  non funziona come dovrebbe a causa del file system che non distingue tra maiuscole/minuscole
- La soluzione è dare un'indicizzazione esplicita del package, mediante l'uso della variabile (di tipo lista) __*all*__

Importare tutto un Package

- La variabile __*all*__, se presente nel file __init__.py contiene i nomi dei moduli che verranno caricati quando viene incontrata un'istruzione "import *"
- Ad esempio, se il file Sounds/Effects/__init__.py contiene:

  __all__ = ["echo", "surround", "reverse"]

  l'istruzione from Sound.Effects import * importerà i tre sottomoduli del Sound package aventi i nomi specificati

Input & Output

- Per ottenere dell'output formattato, è possibile utilizzare l'operatore % del Python che ha come operandi due stringhe:

  s = *string1* **%** *valori*

  l'argomento *string1* è interpretato come una stringa di formato da applicare a *valori* (tipicamente una tupla)

Input & Output

- Per convertire dei valori in stringhe è possibile utilizzare le due funzioni *str*() e *repr*()
- La differenza tra le due è che *str* genera delle stringhe più leggibili, mentre *repr* genera stringhe con rappresentazione più simile a quella interna dell'interprete

Input & Output

- Esempio:
  >>> str(0.1)
  '0.1‘
  >>> repr(0.1)
  '0.10000000000000001'
  >>> import math
  >>> print “Il valore di pi greco è circa %5.3f' % math.pi
  Il valore di pi greco è circa 3.142

Input & Output

- Esempio:

```
tab = {‘Tizio': 4127, ‘Caio': 4098,
‘Sempronio': 7678}
for name, phone in tab.items():
   print '%-10s -> %10d' % (name, phone)
```

- Tizio -> 4127
  Caio -> 4098
  Sempronio -> 7678

Lettura e scrittura File

- Per aprire un file, si utilizza l'istruzione
  open(*nome*, *modo='r'*)

- *nome* è una stringa contenente il nome del file che si vuole aprire
- *modo* può essere 'r', 'w', 'a', 'r+', per indicare se si vuole accedere in sola lettura, sola scrittura, in modo 'append' o 'lettura/scrittura', rispettivamente.
  Sono ammessi anche i modi binari ('rb'…)

Alcuni metodi di File

- Dopo aver creato un oggetto file, è possibile richiamare su di esso diversi metodi
- read(*dim*) legge i primi *dim* byte del file restituendoli in una stringa; se *dim* non è specificato sarà letto l'intero file
- Se è stata raggiunta la fine del file, sarà restituita una stringa vuota

Alcuni metodi di File

- Il metodo *readline* legge una singola linea, aggiungendo il carattere ‘\n’ alla fine della stringa restituita (tranne nel caso in cui sia l’ultima linea)
- Se il file è terminato viene restituita una stringa vuota
- Una linea vuota viene invece rappresentata con un solo carattere ‘\n’

Alcuni metodi di File

- Il metodo *readlines* rende una lista i cui elementi sono le linee del file
- E' possibile precisare un parametro intero che limita le dimensioni del file letto (arrotondate però in modo da completare l'ultima linea letta)

Alcuni metodi di File

- write(*dati*) scrive la stringa *dati* sul file e rende *None*
- tell() rende un intero corrispondente alla posizione corrente nel file, misurata in byte dall'inizio del file
- seek(*offset*, *startref*) cambia la posizione aggiungendo *offset* byte da un particolare riferimento *startref* (=0 inizio file, =1 posizione corrente, =2, fine del file)

Alcuni metodi di File

- close() permette la chiusura di un file e di liberare le risorse allocate per esso dal sistema operativo
- chiaramente, dopo la close non è possibile effettuare operazioni sull'oggetto file:

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Esempio uso di File

- >>> f = open('/tmp/workfile', 'r+')
  >>> f.write('0123456789abcdef')
  >>> f.seek(5) # sesto byte del file
  >>> f.read(1)
  '5‘
  >>> f.seek(-3, 2) # terzultimo carattere
  >>> f.read(1)
  'd'
  >>> f.close()

Modulo pickle

- E’ semplice leggere/scrivere stringhe da/su file
- Per quanto riguarda i numeri, occorre un passaggio in più, in quanto read() rende delle stringhe, che poi possono essere convertite in numeri tramite int(), float(), complex()
- Per leggere/scrivere dizionari, liste e oggetti più complessi?

Modulo pickle

- Il modulo *pickle* fornisce un supporto notevole in questa direzione
- Consente di rappresentare gli oggetti in formato stringa (processo noto come "*pickling*")
- E' successivamente possibile ricostruire un oggetto a partire da una stringa ("*unpickling*")

Modulo pickle

- Dato un oggetto x e un file f, aperto in scrittura, è sufficiente una riga…

  pickle.dump (x, f)

- Per riottenere l'oggetto salvato (se f è un file aperto in lettura):

  x = pickle.load (f)

Eccezioni

- Anche se sintatticamente corrette, alcune istruzioni possono provocare degli errori durante la loro esecuzione
- Le *eccezioni* sono degli errori che avvengono durante l'esecuzione e possono essere gestite all'interno di un programma Python
- Nell'interprete sono generalmente segnalate con dei messaggi

Eccezioni

- >>> 10 * (1/0)
  Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  ZeroDivisionError: integer division or modulo by zero

  >>> 4 + spam*3
  Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  NameError: name 'spam' is not defined

  >>> '2' + 2
  Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  TypeError: cannot concatenate 'str' and 'int' objects

Gestire le Eccezioni

- Esempio:

```
while True:
    try:
       x = int(raw_input(“Numero? "))
       break
    except ValueError:
       print "Oops! Non è un numero!"
```

Gestire le Eccezioni

- L'istruzione *try* cerca di eseguire le istruzioni contenute nel blocco che segue i due punti (sino ad *except*)
- Se non si verificano eccezioni, il blocco dopo *except* viene saltato
- Se si verifica un'eccezione, l'esecuzione salta immediatamente al blocco *except* corrispondente al tipo di eccezione avvenuta

Gestire le Eccezioni

- Se non esiste un blocco *except* in grado di gestire il tipo di eccezione che è stata lanciata, l'esecuzione del programma termina con un errore di "*unhandled exception*"
- Un'istruzione *try* può quindi essere seguita da diverse istruzioni *except*, ognuna in grado di gestire un tipo particolare di eccezione

Gestire le Eccezioni

- Una clausola except può gestire più eccezioni contemporanemente:

  except (RuntimeError, TypeError):

- L'ultima clausola *except* può non avere parametri (attenzione… si potrebbero mascherare errori di programmazione!)

Gestire le Eccezioni

```
import sys
try:
        f = open('myfile.txt')
        s = f.readline()
        i = int(s.strip())
except IOError, (errno, strerror):
        print "I/O error(%s): %s" % (errno, strerror)
except ValueError:
        print “Impossibile convertire in intero"
except:
        print “Errore generico", sys.exc_info()[0]
raise
```

Gestire le Eccezioni

- Il blocco *try* / *except* ammette una clausola *else* opzionale che viene eseguita nel caso in cui nel blocco *try* non sia stata lanciata alcuna eccezione:
- ```
  for arg in sys.argv[1:]:
      try:
          f = open(arg, 'r')
      except IOError:
          print ‘impossibile aprire', arg
      else:
          print arg, ‘ha', len(f.readlines()), 'linee'
          f.close()
  ```
  ```

Gestire le Eccezioni

- Un'eccezione può avere associato un argomento
- Nella clausola *except* può essere specificata una variabile dopo il nome dell'eccezione. La variabile è associata ad un'istanza dell'eccezione avente gli argomenti in *istanza.args*.
- L'istanza definisce __*getitem*__ e __str__ in modo da accedere direttamente agli argomenti senza passare per .*args*.

Gestire le Eccezioni

- ```
  try:
      raise Exception('spam', 'eggs')
  except Exception, inst:
      print type(inst) # istanza dell'eccezione
      print inst.args   # argomenti di .args
      print inst         # __str__ lo consente
      x, y = inst       # __getitem__ lo permette
      print 'x =', x
      print 'y =', y
  ```

  ```
  <type 'instance'>
  ('spam', 'eggs')
  ('spam', 'eggs')
  x = spam
  y = eggs
  ```
  ```

Lanciare le Eccezioni

- Per lanciare un'eccezione si usa l'istruzione raise:

  raise *Nome*, '*argomenti*'

- Esempio:

```
>>>raise NameError, 'Attenzione'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: Attenzione
```

Lanciare le eccezioni

- Per capire se è stata lanciata un'eccezione che però non si vuole gestire, si può usare l'istruzione raise senza parametri:

```
try:
    raise NameError, 'Ciao'
except NameError:
    print 'Eccezione lanciata!'
    raise
Eccezione lanciata!!
Traceback (most recent call last):
File "<stdin>", line 2, in ?
NameError: Ciao
```

Eccezioni personalizzate

- E' possibile definire delle classi per gestire eccezioni particolari
- Normalmente, sono derivate dalla classe Exception. Esempio:

```
>>> class MyError(Exception):
                def __init__(self, value):
                  self.value = value
                def __str__(self):
                  return repr(self.value)
>>> try:
        raise MyError(2*2)
    except MyError, e:
        print 'Eccezione personale con valore:', e.value
```

Azioni di clean-up

- L'istruzione *try* ammette la clausola opzionale *finally* che introduce delle azioni di "clean-up" da eseguire sempre (sia nel caso in cui sono state lanciate eccezioni, sia nel caso in cui non ne sono state lanciate)
- Se è stata lanciata un'eccezione, essa è lanciata nuovamente dopo l'esecuzione di *finally*
- Nota: un'istruzione *try* può avere una o più clausole *except* o una *finally* ma non entrambe!

Azioni di clean-up

- **try**:
  raise KeyboardInterrupt
  finally:
  print ‘Addio mondo!’
  Addio mondo!
  Traceback (most recent call last):
  File "<stdin>", line 2, in ?
  KeyboardInterrupt