moduli_e_package

February 19, 2020

1 Moduli e Package

1.1 Moduli

Un modulo contiene al suo interno definizione di oggetti e metodi predisposti per essere riutilizzati da altri nomi. Un modulo definisce un namespace proprio.

Normalmente un programma in python si divide in: * scrip, codice del programma * modulo funzioni * modulo classi

Gli script contengono il flusso principale del programma, i moduli sono considerati come librerie di oggetti (classi, funzioni, strutture dati, etc.) che utilizziamo tramite importazione nello script od in altri moduli.

[3]:
```
import modulo2, modulo3
```

“‘ from modulo2 import myFunc as fc

[4]:
```
modulo2.myFunc(4)
```

```
4
```

Module search Path: * la home della directory del programma * la directory PYTHONPATH, variabile d’ambiente (facoltativa) * la directory della standard library * la directory site-packages * generato ERRORE per modulo non trovato

1.2 il bytecode di python

Prima di essere eseguiti i moduli devono essere compilati nel bytecode di python; i moduli compilati productono un file del bytecode con suffisso pyc e memorizzato nella cartella *__pycache__* solo nel momento in cui vengono importati.

Il file avrà come nome modulo.cpython-38.pyc.

Sono 2 i casi in cui python genera ed archivia questi files: * il file non è presente nella cache * il file presente nella cache ha DATA precedente a quella del sorgente da cui proviene (si presume che il sorgente sia cambiato e quindi viene ricreato il bytecode)

1.3 Lo Statement IMPORT

Se chiediamo a python che tipo di oggetto è il modulo importato, la risposta sarà *classe module*.

La funzione diventa un attributo della classe modulo importato; per invocarla sarà quindi necessario accedervi attraverso la dot notation; per velocizzare la gestione spesso si usa l'annotazione from miomodulo import miafunzione as fc. In questo caso ci riferiremo alla nostra funzione attraverso l'alias fc.

Le direttive pythoniche indicano che il modo più chiaro di utilizzare attributi di funzione è quello di avere: * import modulo as m * m.miafunzione(20)

In questo modo sapremo sempre da dove viene l'oggetto che stiamo utilizzando, anche nel caso in cui i moduli importati presentino elementi con lo stesso nome.

Ergo da evitare la sintassi from modulo importo * (perdiamo la provenienza degli oggetti)

1.4 Lo Statement FROM

Da utilizzare SOLO nel caso in cui il modulo sia molto corposo e per il nostro lavoro servano solo pochi oggetti. In questo modo eviteremo di sprecare allocazione di memoria per cose che non ci servono. Come visto precedentemente in questo caso potremo accedere all'oggetto senza richiamare il nome del modulo; come visto precedentemente è deprecata la pratica del from modulo import *.

1.5 Disambiguazione degli oggetti

Un caso particolare lo abbiamo quando i 2 moduli importati presentino entrambi l'attributo myFunc; in questo caso potremo utilizzare: * from modulo1 import myFunc as f1 * from modulo2 import myFunc as f2

ottenendo come risultato che gli attributi aventi lo stesso nome (myFunc) verranno poi utilizzati attraverso due alias diversi (f1 ed f2).

1.6 attributo name (attributo predefinito)

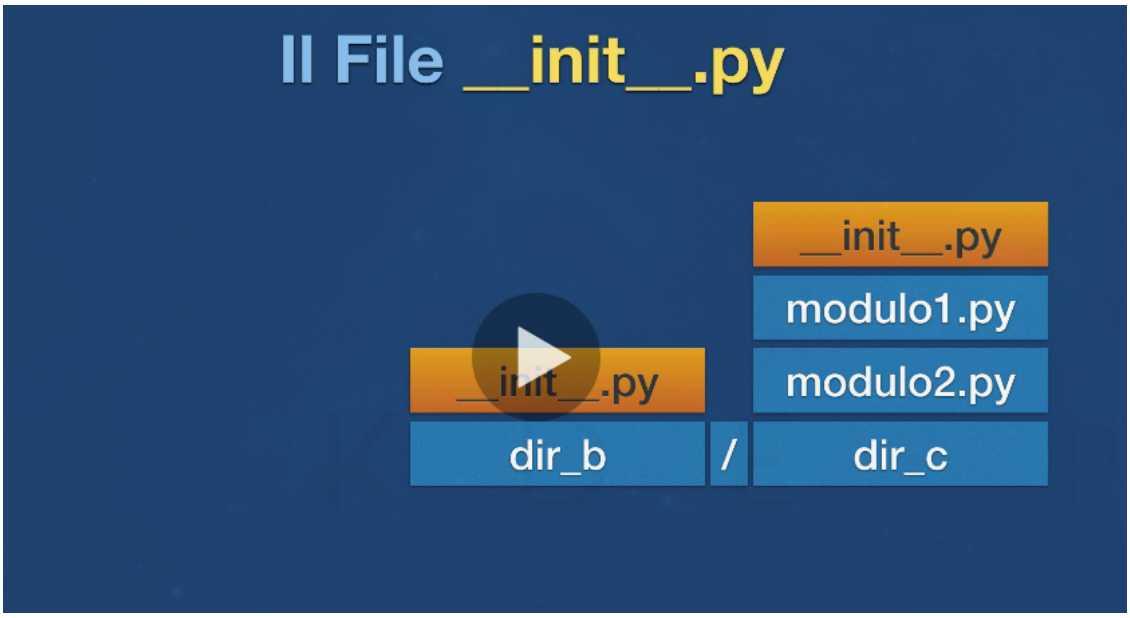
A volte è necessario eseguire un modulo direttamente da riga di comando, ergo risulta importante sapere se un modulo è eseguito dopo essere importato o da riga di comando. Quando un modulo viene importato il suo *__name__* sarà il nome del modulo; se eseguito da riga di comando il *__name__* sarà *__main__*

~~~ if **name** = '**main**': print("il modulo è stato eseguito da riga di comando") else: print("il modulo è stato importato")

1.7 I Package: la directory che contengono i moduli del programma

[8]:
```
from IPython.display import Image
```

[9]:
```
display(Image(filename="./package_python.png"))
```
~~~



Il file dundle init indica a python che questa struttura di directory è un package. la sintassi sarà: > import dir_b.dir_c.modulo1

zeroI file init possono contenere del codice; normalmente si inserisce del codice per inizializzare il package.

Se nel file dunder init inseriamo una riga di codice: > * *__all__* = [calcolatrice,] metterà a disposizione per l'importazione solo il modulo calcolatrice gli altri rimarranno nascosti!

[ ]: