INDICE

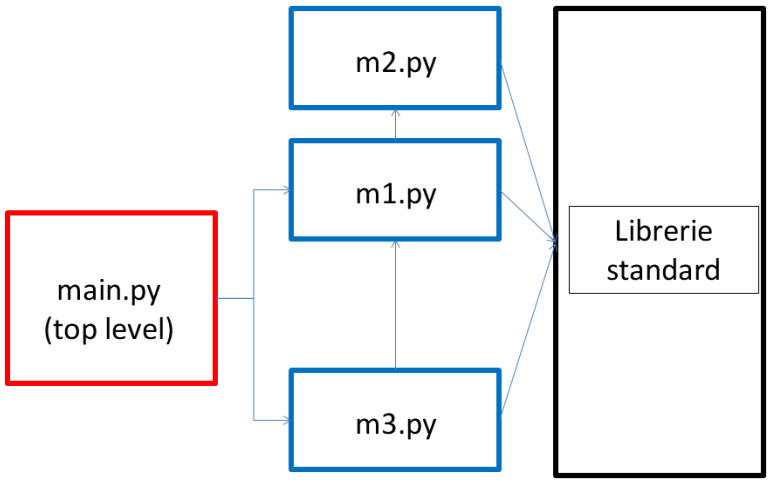
ancora sui moduli e sulla loro importazione

perche' organizzare il codice in moduli?

- perche' il codice risulta più ordinato
- perche' l'organizzazione in moduli riduce il rischio di conflitti di nomi
- perche' più programmi, anche in contesti diversi potrebbero avere bisogno delle stesse funzionalita' e non ha senso replicare il codice.

lo statement import

Facciamo riferimento ad un generico programma strutturato come segue:



In main.py compariranno le seguenti righe:

import m1 import m3 .... m1.funzione1() m2.funzione2()

abbiamo gia' ampiamente usato lo statement import. Adesso vediamo cosa fa l'interprete quando si prova ad importare un modulo.

- cerca il modulo
- compila il modulo (se necessario)
- esegue il codice necessario a rendere disponibili gli oggetti definiti nel modulo stesso

NOTA BENE: questi 3 passi sono eseguiti solo la prima volta che il modulo e' importato (vedi paragrafo import e reload)!!

ricerca del modulo

la ricerca dei moduli avviene secondo la seguente lista ordinata:

I. nella cartella dove risiede il programma principale (main.py nello schema sopra)
II. nelle cartelle specificate nella variabile di ambiente PYTHONPATH (se impostata)
III. nelle cartelle dove risiedono le librerie standard (dipendenti dalla particolare installazione, ad es C:\Python32\Lib)
IV. nelle cartelle specificate in eventuali file con estensione pth (se presenti, una cartella per riga, utile ad esempio, se si hanno installate più versioni dell'interprete)

Alcune osservazioni:

- Per i piccoli progetti, in cui tutto sta nella stessa cartella ci si ferma alla prima opzione.
- Visto che le cartelle dei moduli standard non sono le prime della lista attenzione a non 'nasconderne' qualcuno definendo un modulo con lo stesso nome
- L'insieme delle cartelle in cui l'interprete cerca i moduli e' consultabile nella lista di stringe sys.path (e' possibile modificare via codice questa lista)

In [3]:
```
import sys
for s in sys.path:
    print(s)
```

```
C:\Python32\lib\site-packages\distribute-0.6.28-py3.2.egg
C:\Python32\lib\site-packages\pyreadline-2.0_dev1-py3.2-win32.egg
C:\Python32\lib\site-packages\ipython-0.13-py3.2.egg
C:\Python32\lib\site-packages\nose-1.2.1-py3.2.egg
C:\Python32\lib\site-packages\pygments-1.5-py3.2.egg
C:\Python32\lib\site-packages\pyzmq-2.2.0.1-py3.2-win32.egg
C:\Python32\lib\site-packages\cython-0.17-py3.2-win32.egg
C:\Python32\lib\site-packages\tornado-2.4-py3.2.egg
C:\Python32\lib\site-packages\bitstring-3.0.2-py3.2.egg
C:\WINDOWS\system32\python32.zip
```

```
C:\Python32\DLLs
C:\Python32\lib
C:\Python32
C:\Python32\lib\site-packages
C:\python\00mymodule3
C:\Python32\Lib\site-packages\nbconvert
C:\Python32\Lib\site-packages\nbconvert\converters
C:\python\corso\lezioni\esercizi\dispense
C:\Python32\lib\site-packages\setuptools-0.6c11-py3.2.egg-info
C:\Python32\lib\site-packages\ipython-0.13-py3.2.egg\IPython\extensions
C:\Python32\lib\site-packages\PIL
```

A proposito di file eggs.

compilazione del modulo

Durante l'importazione, per ragioni di efficienza, l'interprete non carica direttamente il sorgente (.py) del modulo. Piuttosto compila quello che si chiama usualmente byte code, un file binario (con estensione pyc) che e' una via di mezzo tra il sorgente e un vero programma compilato. La compilazione del modulo da py a pyc (byte code) avviene solo se necessario cioe' se il codice (py) e' stato modificato più recentemente del byte code.
Altrimenti l'interprete carica direttamente il byte code.

importazione vera e propria del modulo

Al momento dell'importazione:

- sono definite (ma non eseguite) tutte le funzioni e più in generale tutti gli statement python che l'interprete incontra nel file importato
- viene definito un oggetto di tipo 'modulo' ed sono definiti tutti i suoi attributi
- questo oggetto e' quello che sara' utilizzato nel momento in cui nel file principale si richiameranno le funzionalita' definite nel modulo stesso (m1.funzione1()).

import cosa?

Non c'e' tempo per approfondire molto la questione ma, non a caso nello statement si omette l'estensione py del modulo da importare.
Questo accade non solo per una questione di sintassi ma soprattutto perche' con *import a* e' possibile importare:

- un file sorgente python a.py
- un file bytecode a.pyc
- una cartella a (package import)
- una libreria 'linkata' dinamicamente b.dll
- una libreria 'linkata' staticamente b.lib
- ....

import e reload

l'importazione effettiva di un modulo avviene effettivamente solo la prima volta che un modulo viene importato.
Questo vale sia (1) all'interno di un programma sia (soprattutto) (2) in una sessione interattiva dell'interprete.

Quanto detto sopra ha particolare rilevanza nell'uso interattivo di Python, specie in fase di sviluppo dei moduli:

- se si sta lavorando ad un modulo
- lo si sta testando in modalita' interattiva

- lo si modifica dopo la prima importazione

Affinche' le modifiche abbiano effette nella sezione e' necessario forzare la nuova importazione con la funzione reload del modulo standard imp.

esercizio (import vs reload)

Eseguire il seguente test (da fare in aula):

- creare un semplice modulo 'a0.py' con una funzione f() che stampa una stringa
- da idle eseguire il modulo con F5 in modo da impostare la cartella dove sta a come quella di lavoro
- controllare con dir() cosa è stato aggiunto allo scope corrente
- eseguire la funzione f (f())
- modificare la stringa restituita da f()
- importare a0 (import a0) e eseguire f() (a0.f())
- modificare nuovamente la stringa restituita da f()
- importare nuovamente a (import a0) e rieseguire f() (a0.f())
- importare il modulo imp e ricaricare il modulo a0 con imp.reload(a0)
- rieseguire f() (a.f())

Altre forme oltre al semplice *import nomemodulo*

from ... import ...

piuttosto che importare un intero modulo e' possibile importare solo alcuni nomi in esso definiti:

```
In [2]: from os.path import abspath
        abspath('.')
```

```
Out[2]:  'C:\\python\\corso\\lezioni'
```

nell'esempio sopra ho importato solo la funzione abspath del modulo os.path e la uso senza preporre il nome del modulo prima della funzione stessa.

equivalenza (import modulo) e (from modulo import nomi)

l'espressione:

```
In [ ]: from modulo import nome1,nome2
```

e' equivalente a:

```
In [ ]: import modulo
        nome1 = modulo.nome1
        nome2 = modulo.nome2
        del modulo
```

quando import e' strettamente necessario ...

quando due moduli definiscono lo stesso nome ....

from ... import *

quando uso *import modulo* per usare un nome definito nel modulo devo usare la sintassi *modulo.nome* a meno che non abbia usato lo statement from.

con from posso anche importare tutti i nomi definiti in un modulo che potranno quindi essere richiamati senza riferimento al modulo stesso:

vediamo un esempio: il modulo os.path definisce molte funzioni tra le quali abspath che restituisce il path completo di una cartella:

In [1]:
```
#accedo ad abspath attraverso il nome del modulo
import os.path
os.path.abspath('.')
```

Out[1]:
```
'C:\\python\\corso\\lezioni'
```

In [2]:
```
#abspath non è raggiungibile senza specificare il nome del modulo
abspath('.')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-2a90611801d8> in <module>()
----> 1 abspath('.')

NameError: name 'abspath' is not defined
```

ma se uso la sintassi from ... import * :

In [3]:
```
from os.path import *
abspath('.')
```

Out[3]:
```
'C:\\python\\corso\\lezioni'
```

questa sintassi puo' sembrare utile e comoda ma se possibile va evitata perche' viola la divisione in ***namespace*** introdotta con la strutturazione del codice in moduli

inoltre questa sintassi rende più difficile individuare la 'provenienza' di una variabile o di una funzione e di conseguenza rende meno 'leggibile' il codice.

import ... as ...

e' possibile cambiare nome a un modulo (ad esempio per abbreviarlo) con la seguente sintassi:

In [3]:
```
import math as m
m.cos(m.pi/6)
```

Out[3]:
```
0.8660254037844387
```

Esercizi sui moduli

operazioni preliminari

Nella cartella work o in una generica cartella di lavoro:

- definire un modulo `a.py` che contiene 3 funzioni con nome `fa1` , `fa2` e `f3` ciascuna delle quali stampa una stringa contenente i nomi del modulo e della funzione stessa

- definire un modulo `b.py` che contiene 3 funzioni con nome `fb1` , `fb2` e `f3` ciascuna delle quali stampa una stringa contenente i nomi del modulo e della funzione stessa

es 1 (import ... VS from ... import ...)

creare uno script s1.py in modo che:

- importi a e b con lo statement import e invochi le funzioni nel seguente ordine: a.fa1 , a.fa2 , a.f3 , b.fb1 , b.fb2 , b.f3

creare uno script s2.py in modo che:

- importi a e b con lo statement *from modulo import* * e invochi le funzioni nel seguente ordine: fa1 , fa2 , f3 , fb1 , fb2 , f3 (FARE ATTENZIONE ALL?OUTPUT ... COSA SUCCEDE?)

if __name__=='__main__':

- in genere in un modulo si definiscono funzioni o classi.
- c'e' un modo per includere nel modulo anche una sessione di test o di esempi delle funzioni create senza che queste siano eseguite quando il modulo e' importato da altri moduli.
- questa sessione di test/esempi sara' eseguita solo quando il modulo viene eseguito come script.

facciamo una prova con un modulo a.py come quello dell'esercizio precedente, nel quale si è inclusa una sezione di test:

In [ ]:
```
if __name__=='__main__':
    print('-'*10)
    fa1()
    print('-'*10)
    fa2()
    print('-'*10)
    f3()
    print('-'*10)
```

proviamo ad importare il modulo:

In [5]:
```
import sys
sys.path.append('C:/python/corso/lezioni/work')
import a
```

nulla di fatto, la sezione di test non viene eseguita

Se eseguiamo lo script da terminale invece la sezione di test viene eseguita

```
C:\python\corso\lezioni\work>python a.py
----------
modulo a, funzione fa1
----------
modulo a, funzione fa2
----------
modulo a, funzione f3
----------

C:\python\corso\lezioni\work>
```

Script e argomenti da riga di comando

Quando eseguiamo uno script da terminale possiamo anche passargli dei parametri da linea di comando. Vediamo due modi per gestirli:

- usando il modulo sys (sys.argv)
- usando il modulo getopt

sys.argv

il modulo sys mette a disposizione una variabile `argv` che è una lista di stringhe:

- `argv[0]` contiene il nome o il path completo dello script in esecuzione (dipende dal contesto e dal sistema operativo)
- `argv[1:]` sono i parametri passati da riga di comando

vediamo come cio' può essere sfruttato con un esempio

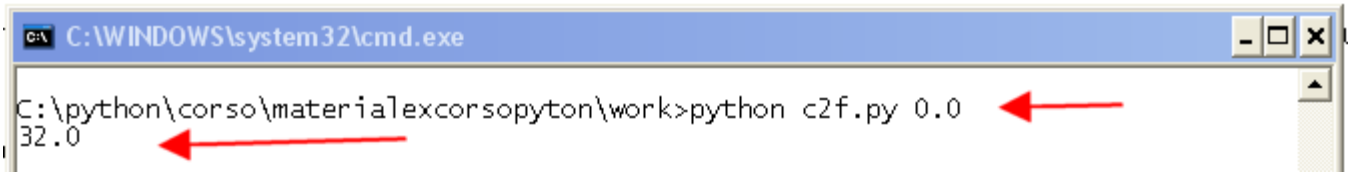
esempio: gestione di un singolo parametro da riga di comando

Facciamo un esempio:
Vogliamo uno script `c2f` che accetti come parametro da riga di comando un numero float che indica la temperatura Celsius e che restituisca su standard output la temperatura convertita in Fahrenheit F = 9.*C/5+32.

In altre parole vogliamo che se eseguiamo lo script da terminale nel seguente modo:

python c2f.py 0.0

sul terminale stesso, alla riga successiva compaia il numero 32.0:



Creiamo uno script c2f.py nella catella work che contiene la seguente funzione di trasformazione da gradi Celsius a Fahrenheit:

In [ ]:
```
def c2f(C):
    return 9.*C/5+32.
```

subito dopo la definizione della funzione inseriremo il seguente codice:

if __name__=='__main__': import sys C = float(sys.argv[1]) #NB, no argv[0] print(c2f(C))

il risultato dovrebbe essere quello sperato

esercizio: gestione di un numero imprecisato di parametri da riga di comando

a partire dallo script del precedente esempio, creare `c2f_01.py` che gestisca un numero imprecisato di numeri float passati da riga di comando e che applichi la conversione definita dalla funzione c2f() a ciascuno di essi.

```
C:\python\corso\materialexcorsopyton\work>python c2f_list.py 0.0 1 2 3 4 5 6
0.0 C -> 32.0 F
1.0 C -> 33.8 F
2.0 C -> 35.6 F
3.0 C -> 37.4 F
4.0 C -> 39.2 F
5.0 C -> 41.0 F
6.0 C -> 42.8 F
```

coppie opzione,valore sulla riga di comando, il modulo getopt

Il modulo standard getopt permette di specificare le opzioni da riga di comando assegnando a ciascun parametro un nome (opzione).
la sintassi supportata è

-nomeopzione valore

in particolare si usa la funzione `getopt.getopt` che accetta come `parametri principali`:

- una lista contenente i valori degli argomenti passati da riga di comando (in genere `sys.argv[1:]`) che conterra' sia i nomiopzione sia i valori)
- una stringa che definisce i nomi delle opzioni valide

la funzione `restituisce`:

- una lista di touple ciascuna delle quali contiene una coppia di stringhe ('-' + nomeopzione , valore)
- una lista che contiene tutti gli altri parametri passati senza opzione

esempio sull'uso di getopt

consideriamo la formula che esprime la posizione di un punto uniformemente accelerato in funzione dell'istante t, dell'accelerazione a e dell condizioni iniziali v0 e s0 per t=0:

In [ ]:
```
def posiz(t,a,v0,s0):
    return s0 + v0*t + 0.5*a*t**2
```

voglio creare uno script `posiz.py` che accetti il tempo, l'accelerazione, la velocità e la posizione iniziale come parametri nel seguente modo:

-T -A -V -S

dove l'ordine non conti e dove se eventualmente non definisco uno dei parametri valgano dei valori di default da impostare a piacere

```
C:\python\corso\materialexcorsopyton\work>python posiz.py -T 2 -V 1 -S 3 -A 6
posiz(t=2.0 , a=6.0 , v0=1.0 , s0=3.0) = 17.0
```

per fare cio' definisco il corpo principale dello script come segue

if __name__=='__main__': import sys,getopt #definisco i valori di default a = t = 1 s0 = v0 = 0 options,args = getopt.getopt(sys.argv[1:], #gli argomenti 'T:S:V:A:') #le lettere riconosciute come opzione seguite da : se associate ad un valore #restituisce una print('options = {0}'.format(options)) print('args = {0}'.format(args)) for option,value in options: if option == '-T': t = float(value) elif option == '-S': s0 = float(value) elif option == '-V': v0 = float(value) elif option == '-A': a = float(value) print('posiz(t={0} , a={1} , v0={2} , s0={3}) = {4}'.format(t,a,v0,s0,posiz(t,a,v0,s0)))

Cenni alla creazione di packages

- Quando i progetti diventano grandi il numero dei nuovi moduli creati puo' essere grande ed e' spesso opportuno organizzarli in modo strutturato.
- I moduli possono essere strutturati in cartelle (packages) e, piuttosto che aggiungere decine di cartelle ai percorsi di ricerca dei moduli, le cartelle possono essere importate come se fossero moduli con la possibilita' di accedere a tutti i moduli che esse contengono.

per creare un package:

- affinche' la cartella sia importabile come package deve essere situata in uno dei 'luoghi' dove l'interprete ricerca i moduli (quindi, per una prima prova, va bene la cartella di lavoro corrente, vedi paragrafi precedenti)

- la cartella deve contenere un file '`init.py`' che viene eseguito alla prima importazione del package e che puo' essere anche vuoto
- i moduli all'interno delle cartelle sono accessibili con la sintassi `nomepackage.nomemodulo` mediante gli statement di importazione usati con i moduli semplici
- i packages possono essere annidati

esercizio (packages, s3.py)

- creare una cartella mypack in work
- creare al suo interno un file vuoto con nome __init__.py
- copiare nella cartella mypack i moduli a e b dell'esercizio sui moduli e modificare le stringhe restituite dalle funzioni in modo che si riferiscano anche al package oltre che al modulo
- creare in work tre versioni di uno script in cui si usano le 6 funzioni definite in a e b in modo che:
  - a) si importino a e b con lo statement *import ...* (`s3_01.py`)
  - b) si importano a e b con lo statement *import mypack.x as x* (`s3_02.py`)
  - c) si importino a e b con lo statement *from package import modulo1,modulo2* (`s3_03.py`)

esercizio (packages annidati, s4.py)

- creare una package subpack annidato in mypack
- copiare nella cartella subpack i moduli a e b dell'esercizio precedente e modificare le stringhe restituite dalle funzioni in modo che si riferiscano anche al subpackage oltre che al package ed al modulo
- creare un nuovo script `s4.py` nel quale si usano tutte le 12 funzioni, avendo cura di verificare di averle invocate tutte