Corso di **Python**

Lezione 9

Moduli e packages

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Moduli

- Un tipico programma Python è tipicamente costituito da più file sorgenti, detti anche **moduli**
- Ogni modulo può avere la necessità di utilizzare funzioni, classi o variabili globali definite in altri moduli
- Per gestire queste dipendenze in altri moduli si usano i costrutti *import* e *from-import*
- I moduli possono essere organizzati e raggruppati gerarchicamente in pacchetti (***packages***)
- Python supporta anche delle estensioni (***extensions***) scritte in altri linguaggi come C, C++, Java o C#, che possono essere importate come fossero dei moduli Python

Istruzione **import**

- Per importare un file sorgente come modulo si usa l'istruzione **import**:

```
import modname [as varname][,...]
```

  - *modname* è una sequenza di identificatori separati da punti, che individuano un percorso che punta ad un modulo in uno specifico package
  - *varname* è il nome con cui il modulo importato deve essere accessibile nel modulo che lo importa
  - Si possono effettuare gli import di più moduli con una sola istruzione, usando la virgola come separatore

- Esempio:

```
>>> import os, django.db.models
>>> os.getenv("PATH")
'/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin'
>>> django.db.models.BLANK_CHOICE_DASH
[(u'', u'---------')]
```

- L'accesso agli oggetti del modulo importato avviene con la stessa notazione (*punto*) che si usa per accedere agli attributi di generici oggetti Python

Istruzione **import - as**

- Talvolta è utile ridenominare i moduli che si importano utilizzando l'opzione ***as***:
  - Per evitare conflitti con altri moduli importati
  - Per ridurre la verbosità delle chiamate

- Esempio:

```
>>> import configparser as MyParser
>>> MyParser.__name__
'configparser'
```

- Si tratta comunque di un *alias* ad oggetto strutturato che potrebbe essere definito anche in maniera diretta:

```
>>> import configparser
>>> MyParser = configparser
>>> MyParser.__name__
'configparser'
```

- In generale posso importare più volte la stessa libreria con nomi diversi:

```
>>> import os, os as os2
>>> id(os.urandom) == id(os2.urandom)   # Nome diverso ma stesso modulo ...
True
```

Istruzione **from - import**

- Per importare solo alcuni *attributi* dal modulo specificato si usa il costrutto *from...import*:

```
from modname import attname [as varname][,...]
from modname import *
```

- Esempi:

```
>>> from collections import namedtuple
>>> from UserDict import IterableUserDict as UserDict
```

- In questo caso posso modificare non il nome del modulo ma dell'attributo che viene importato dal modulo
- La forma con l'asterisco permette di importare tutti i nomi esportati dal modulo:
  - Se nel modulo è definito un attributo **`__all__`** importa tutti e soli gli attributi elencati
  - Se `__all__` non è definito allora importa tutti gli attributi del modulo ad eccezione di quelli che iniziano con '`_`' (*underscore*)
  - La forma con **`*`** è da evitare ove possibile perché una modifica del modulo esterno può sovrascrivere nomi del modulo che effettua l'importazione

Evitiamo i **from ... import ***

- Con un modulo che definisce alcuni oggetti:

```
$ cat mymodule.py
PUBLIC_VALUE = 1000
_RESERVED_VALUE = 2000
__PRIVATE_VALUE = 3000

def f1():
    print("Chiamata funzione f1 del modulo %s" % __name__)
```

- Facendo delle importazioni dall'interprete posso sovrascrivere inavvertitamente degli oggetti esistenti:

```
>>> def f1():
...     print("Chiamata funzione f1 del modulo %s" % __name__)
...
>>> f1()
Chiamata funzione f1 del modulo __main__
>>> from mymodule import *
>>> f1()
Chiamata funzione f1 del modulo mymodule
```

import VS **from-import**

- In generale la forma *import* sarebbe da preferire rispetto alla forma *from-import*
- La seconda può essere preferibile quando è necessario importare solo pochi attributi dal modulo, per non appesantire lo spazio dei nomi
- Sempre la forma form-import può essere più conveniente quando è necessario importare solo moduli specifici da un determinato *package*

Perché è utile l'aliasing **as**

```
try:
  from lxml import etree
  print("running with lxml.etree")
except ImportError:
  try:
    # Python 2.5
    import xml.etree.cElementTree as etree
    print("running with cElementTree on Python 2.5+")
  except ImportError:
    try:
      # Python 2.5
      import xml.etree.ElementTree as etree
      print("running with ElementTree on Python 2.5+")
    except ImportError:
      try:
        # normal cElementTree install
        import cElementTree as etree
        print("running with cElementTree")
      except ImportError:
        try:
          # normal ElementTree install
          import elementtree.ElementTree as etree
          print("running with ElementTree")
        except ImportError:
          print("Failed to import ElementTree from any known place")
```

Docstring del modulo

- La docstring del modulo è definita con un letterale stringa (solitamente stringa multilinea) nella prima *linea logica* del file:

```
$ head -4 cmdb/urls.py
# -*- coding: utf-8 -*-
"""
Definizione delle url per il progetto 'cmdb'.
"""
```

- Se la prima linea logica non è una stringa allora la docstring sarà la stringa vuota
- La docstring viene mappata sull'attributo `__doc__` del modulo:
- Esempio:

```
>>> import os
>>> os.path.__doc__
'Common operations on Posix pathnames.\n\n ... the pathname
component of URLs.\n'
```

Attributi di un modulo

- Un modulo ha una serie di attributi specifici:

  `__name__` : Nome del modulo

  `__doc__` : Stringa di documentazione

  `__file__` : Nome del file che contiene il modulo (opzionale)

  `__package__` : Nome del package che contiene il modulo

  `__builtins__` : Funzioni built-in accessibili dal modulo

- In Python 3 sono stati aggiunti altri 3 attributi:

  `__cached__` : Path alla versione compilata del modulo (opzionale)

  `__loader__` : Nome del loader utilizzato per caricare il modulo (usato per introspezione)

  `__spec__` : Istanza `ModuleSpec` (PEP 0451) con informazioni di importazione (usato per l'introspezione)

- Questi attributi sono utilizzabili direttamente nel codice del modulo, mentre esternamente bisogna riferirsi con il prefisso del modulo:

```
__file__        # internamente al codice del modulo a.py

>>> import a    # in altro file o nell'interprete interattivo
>>> a.__file__
a.py
```

Mappa dei simboli

- L'attributo speciale `__dict__` contiene la mappa dei simboli del modulo (nome --> valore)
- Non è possibile riassegnare l'intero dizionario:

```
>>> import mymodule
>>> mymodule.__dict__ = {}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: readonly attribute
```

- Ma è possibile modificare le singole voci (operazione comunque da evitare) o aggiungerne di nuove:

```
>>> mymodule.__dict__['__PRIVATE_VALUE']
3000
>>> mymodule.__dict__['__PRIVATE_VALUE'] = 1   # Equivale a
mymodule.__PRIVATE_VALUE = 1
>>> mymodule.__dict__['__PRIVATE_VALUE']
1
>>> mymodule.__dict__['new_value'] = 255
>>> mymodule.__dict__['new_value']
255
```

- Non usare direttamente `__dict__`, meglio usare la notazione *punto* o al limite `setattr`

Caricamento dei moduli

- Il caricamento dei moduli è affidato alla funzione built-in `__import__`
- Come opera questa questa funzione:

  1) Viene consultato il dizionario dei moduli di sistema *sys.modules*

  2) Se non presente cerca di caricare il modulo cercandolo secondo l'ordine dei path della lista *sys.path* (possono essere path di directory o di archivi ZIP)

  3) Se viene trovato il modulo allora viene caricato e salvato nel dizionari dei moduli
- L'istruzione di import può essere ripetuta più volte senza danni
- Il primo caricamento del modulo può richiedere più tempo, mentre successivi *import* risultano immediati
- La lista **sys.path** contiene una serie di path prioritari e può essere estesa con la variabile PYTHONPATH:

```
>>> import sys; sys.path
['', '/usr/lib64/python27.zip', '/usr/lib64/python2.7', '/usr/lib64/python2.7/plat-
linux2', '/usr/lib64/python2.7/lib-tk', '/usr/lib64/python2.7/lib-old',
'/usr/lib64/python2.7/lib-dynload', '/usr/lib64/python2.7/site-packages',
'/usr/lib64/python2.7/site-packages/gtk-2.0', '/usr/lib/python2.7/site-packages',
'/usr/lib/python2.7/site-packages/gtk-2.0']
```

Caricamenti circolari

- Python ammette i caricamenti circolari: è possibile definire un modulo `m1.py` che contiene un `import m2` e un modulo `m2.py` che contiene `import m1`
- È una possibilità che si può usare sapendo che:
  - Quando effettuiamo un `import m1` il modulo viene caricato in `sys.modules['m1']` ed eseguito
  - Quando, eseguendo il modulo `m1`, si trova `import m2` il secondo modulo viene caricato in `sys.modules['m2']`, l'esecuzione di `m1` viene interrotta e inizia l'esecuzione di `m2`
  - Quando in `m2` si esegue `import m1`, quest'ultimo non riprende l'esecuzione comunque, perché `__import__` trova che c'è già `sys.modules['m1']`
  - Non posso accedere nel seguito di `m2` a valori di `m1` che non sono ancora definiti (per accedere si intende usare effettivamente, non istruzioni magari contenute in funzioni o classi definite ma al momento non chiamate)
  - L'esecuzione di `m1` riprende e termina dopo che quella di `m2` sarà stata completata
- Per queste limitazioni nel caricamento è meglio fare uno sforzo per eliminare i caricamenti circolari ove possibile

Test caricamento circolare

- Script `m1.py` e `m2.py` per testare il caricamento circolare:

```
# Modulo m1 per il test degli import
print("Definisco oggetti di m1")
stringa_di_m1 = "ABC"
print("Adesso importo il modulo m2")
import m2
print("Termino l'esecuzione di m1")
numero_di_m1 = 1
```

```
# Modulo m2 per il test degli import
print("Definisco oggetti per m2")
stringa_di_m2 = "XYZ"
print("Adesso importo il modulo m1")
import m1
print("Termino l'esecuzione di m2")
numero_di_m2 = 2
```

```
>>> import m1
Definisco oggetti di m1
Adesso importo il modulo m2
Definisco oggetti per m2
Adesso importo il modulo m1
Termino l'esecuzione di m2
Termino l'esecuzione di m1
```

<< Conclude prima l'importazione di m2 ...

Rimozione di moduli

- È possibile rimuovere moduli precedentemente caricati:

```
>>> import mymodule
>>> import sys
>>> del mymodule
>>> del sys.modules['mymodule']
```

- Allo stato attuale i moduli rimossi non vengono di norma eliminati dalla memoria:
  - Non è semplice per riferimenti ricorsivi nel caricamento dei moduli
  - La rimozione avviene solo per moduli che hanno un numero di riferimenti molto limitato
  - Previsto il meccanismo in Python 3 ma serve implementare un codice specifico per ogni modulo
  - http://bugs.python.org/issue9072

Reload di moduli

- È possibile ricaricare moduli già inseriti senza rimuoverli
- Si usa una funzione del pacchetto **imp**, una libreria di funzioni interne per gestire le importazioni
- Esempio:

```
>>> import imp
>>> imp.reload(mymodule)
```

- Il reload non rimuove il modulo dalla memoria
- In Python 2 imp.reload() implementata come funzione built-in
  - In Python 3 la funzione built-in `reload` è stata tolta e ridefinita come funzione di libreria `imp.reload()`
  - Python 2.6/2.7 includono entrambe le possibilità con un backport

Libreria **importlib**

- Da python 3.1 è stata definita la libreria **importlib**, che sostituisce la libreria *imp*:
  - Amplia le funzionalita, implementando una serie di proposal (PEP)
  - **imp** è deprecata dalla versione 3.4, ma una volta eliminata verrà mappata con un alias alla nuova libreria
  - https://docs.python.org/3/library/importlib.html

Si crea una nuova libreria quando il codice è strutturalmente diverso ...

Override di funzioni built-in

- Il dizionario delle funzioni built-in a livello di modulo viene utilizzato per il *lookup dei nomi* dopo la verifica nelle variabili globali (se il nome non viene trovato viene generato un errore `NameError`)
- I nomi delle funzioni built-in non sono riservati perciò si può fare l'override di una funzione built-in a livello di modulo definendo una nuova funzione come simbolo globale del modulo
- Importando il modulo `__builtin__` (`builtins` in Python 3) si può arrivare a fare un override diretto della funzione stessa:

```
# Ridefinizione di reload() per accettare una stringa con nome
# del modulo (da 'Python in a Nutshell' di Alex Martelli)
import __builtin__

def reload(mod_or_name):
    if isinstance(mod_or_name, str):
        mod_or_name = __import__(mod_or_name)
    return _reload(mod_or_name)

__builtin__.reload = reload
```

Packages

- Un package è un modulo che contiene altri moduli, secondo una struttura gerarchica
- La struttura gerarchica è realizzata mediante sottodirectory o archivi ZIP
- Un pacchetto *pkg* risiede in una directory `pkg/` ed ha il modulo principale (*module-body*) definito nel file `pkg/__init__.py`
- Gli altri file *.py* che si trovano medesima directory `pkg/` sono moduli del pacchetto stesso
- Tutte le directory di `pkg/` che contengono un file `__init__.py` sono sottopacchetti di *pkg*
- Un modulo *mdl* definito in un pacchetto *pkg* può essere importato riferendosi ad esso come pkg.mdl
- `__init__.py` viene comunque caricato prima di un qualsiasi altro modulo del pacchetto

Esempio di package

```
sound/                          Top-level package
      __init__.py
      formats/                  Subpackage sound.format
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
      effects/                  Subpackage sound.effects
              __init__.py
              echo.py
              surround.py
              reverse.py
      filters/                  Subpackage sound.filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
```

Ruolo di `__init__.py`

- Il file `__init__.py` effettua l'inizializzazione del pacchetto:
  - deve essere usato come un analogo del metodo `__init__` delle classi
  - Mettere in `__init__.py` i caricamenti di funzioni base del pacchetto che devono essere viste dopo l'import del pacchetto
- Esempio (file `__init__.py` del pacchetto Django):

```
from django.utils.version import get_version
VERSION = (1, 8, 6, 'final', 0)
__version__ = get_version(VERSION)

def setup():
    from django.apps import apps
    from django.conf import settings
    from django.utils.log import configure_logging

    configure_logging(settings.LOGGING_CONFIG, settings.LOGGING)
    apps.populate(settings.INSTALLED_APPS)
```

Path assoluti e relativi

- Fino a Python 2.4 l'importazione aveva ambiguità nella specifica di path locali o assoluti, per esempio:

```
import foo
```

  poteva significare il modulo foo del package locale o del package di libreria

- Dalla versione 2.5 è stata implementata una sintassi più estesa che rende i vecchi path solo assoluti (rispetto ai path definiti in sys.path ...) e permette di specificare degli import con path esplicitamente relativi:

```
from . import echo    # importa echo da ./__init__.py
from .. import formats
from ..filters import equalizer
```

- La nuova specifica dei path per gli import è il default dalla versione 2.7 e in Python 3, mentre per la 2.5 e 2.6 si può attivarla all'inizio del modulo con:

```
from __future__ import absolute_import
```

Il **main** program

- Come in altri linguaggi usabili per lo scripting non è prevista una funzione *main*
- Ad essere main è il modulo stesso che inizialmente caricato
- Questo modulo viene mandato in esecuzione come ogni altro modulo, ma ha la peculiarità di essere l'unico ad avere '__main__' come nome del modulo (anche se il file si chiama diversamente):

```
>>> __name__    # E' __main__ anche nela shell interattiva ...
'__main__'
```

- Ecco quindi che si può creare una funzione main per il modulo, che viene richiamata solo se questo è il main program del processo da eseguire:

```
import sys


def main(argv):
    n = int(argv[1])
    print(n + 1)


if __name__ == '__main__':
    sys.exit(main(sys.argv))
```

Distribuire pacchetti Python

- I packages possono essere distribuiti in vari formati:
  - .exe, .zip, .msi per **Windows**
  - .rpm, .srpms, .deb, .ZIP, .tar.gz per **Linux**
  - .app per **MacOS**
  - .egg (Python Eggs;-) un formato .ZIP installabile con *easy_install*
  - .whl (Python Wheels) sempre formato .ZIP che sostituisce i .egg
- Per creare e distribuire i pacchetti si può usare:
  - **distutils** : sistema nativo con la distribuzione, con alcune limitazioni
  - **setuptool** : sistema alternativo esteso, con più opzioni rispetto a *distutils*
- Con *distutils* non è ancora possibile creare dei pacchetti *wheels*, cosa che invece è possibile con *setuptools*
- *Sia distutils che setuptools permettono di pubblicare il pacchetto su PyPI (in formato .tar.gz)*

Libreria distutils

- E' più che sufficiente per gli usi standard
  - Creare automaticamente archivi *.tar* e *.zip*
  - Creare automativamente pacchetti *.rpm* e *.deb*
- Per utilizzarla basta definire uno script `setup.py` nella directory base del progetto
- Esempio minimale:

```
from distutils.core import setup
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Comandi di distutils

```
$ python setup.py sdist
```

- Crea una distribuzione sorgente (.tar su Linux e .zip su Windows)

```
$ python setup.py bdist_rpm
```

- Crea un pacchetto RPM su sistema Linux RPM-based
- Usando pacchetto aggiuntivo su Debian (alien) è possibile creare il file .rpm e poi convertirlo in automatico in .deb

```
$ python setup.py bdist_wininst
```

- Crea un pacchetto .exe per Windows

```
$ python setup.py bdist -help-formats
```

- Per vedere quali altri formati sono disponibili

```
$ python setup.py install
```

- Installa il pacchetto (richiede privilegi di root)

File opzionale `setup.cfg`

- E' un file di configurazione con sintassi identica a quella dei file *.INI* di Window
- Permette di definire parametri per i vari comandi disponibili su distutils
- Esempio:

```
# Setup Configuration file for Lograptor
[bdist_rpm]
packager = Davide Brunato <brunato@sissa.it>
doc-files = README LICENSE doc/Lograptor.pdf
vendor = SISSA
group = Applications/System
release = 10
```

Installazione di pacchetti

- Per installare software e librerie Python si possono usare ovviamente i pacchetti della distribuzione
- Python ha però il suo gestore di pacchetti **pip** ("Pip Installs Packages" o "Pip Installs Python"):
  - creato nel 2008 per migliorare *easy_install* un tool già disponibile dal 2004 nel package *setuptools*
  - permette di gestire le dipendenze, listare i pacchetti e disinstallare pacchetti installati
  - Installazione dei pacchetti del repo PyPi via SSL

Comandi di **pip**

```
Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
  freeze                      Output installed packages in requirements format.
  list                        List installed packages.
  show                        Show information about installed packages.
  search                      Search PyPI for packages.
  wheel                       Build wheels from your requirements.
  hash                        Compute hashes of package archives.
  help                        Show help for commands.
```

Ambiente virtuale e pip

- Avendo a disposizione Python 3, oppure Python 2 con il pacchetto *virtualenv*, si possono creare ambienti virtuali con pacchetti aggiuntivi, anche senza essere root ...
- Esempio:

```
$ pyvenv myenv
$ source myenv/bin/activate
(myenv) $ pip list
(myenv) $ pip list
pip (7.1.0)
setuptools (18.0.1)
$ (myenv)$ pip install django
Collecting django
  Downloading Django-1.9.1-py2.py3-none-any.whl (6.6MB)
    100% |████████████████████████████████| 6.6MB 121kB/s
Installing collected packages: django
Successfully installed django-1.9.1
```

- https://packaging.python.org/en/latest/installing/#virtual-environments