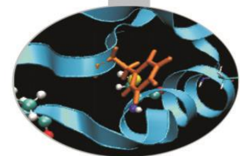
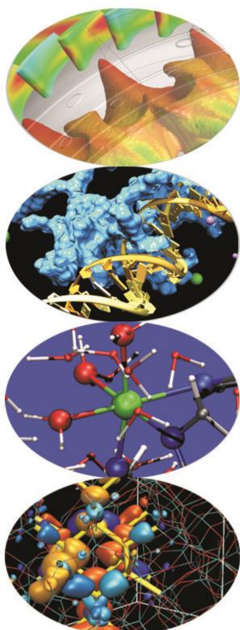
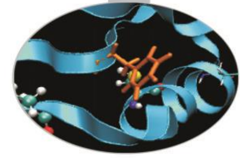
CINECA SCAI
SuperComputing Applications and Innovation



Introduzione

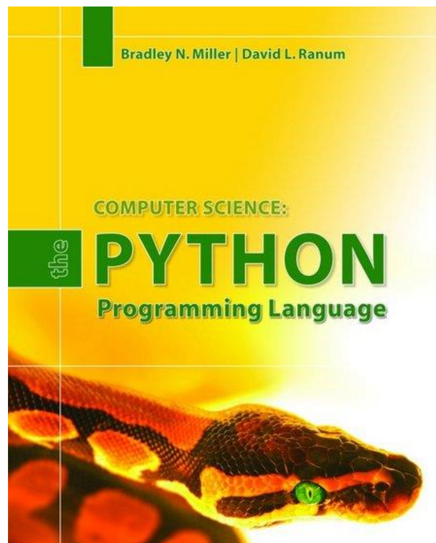


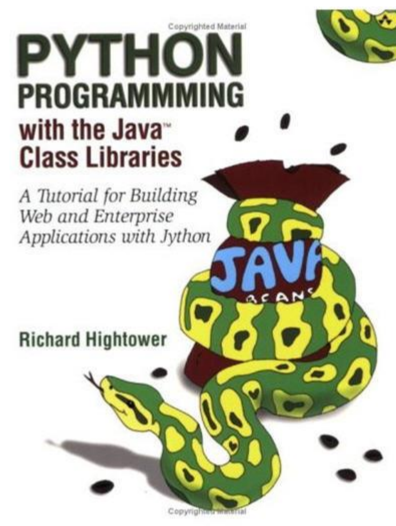
CINECA

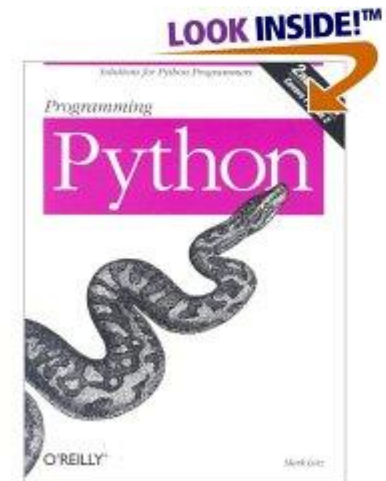


Python sta per Pitone?

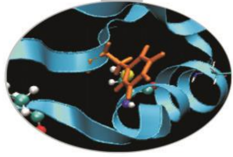
- Il nome deriva da “Monty Python's Flying Circus” (famoso gruppo di comici inglese di cui il padre di Python, GuidoVan Rossum, è grande fan)
- Ciononostante, numerosi libri e pubblicazioni/progetti di e su Python usano il pitone come animale simbolo







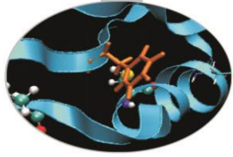
Cosa è python



*“Python è un linguaggio di programmazione di **alto livello, interpretato**, **orientato agli oggetti** e con una **semantica dinamica**. [...] molto interessante per lo **sviluppo rapido di applicazioni**, così come per l’utilizzo come **linguaggio di scripting** o come **linguaggio collante** per connettere assieme componenti esistenti. La sintassi semplice e facile da apprendere di Python enfatizza la **leggibilità** e riduce il costo di mantenimento dei programmi. [...] **programmazione modulare** ed il riutilizzo del codice. L’interprete Python e l’estesa libreria standard sono disponibili sia come sorgente che in forma binaria, **senza costo per le maggiori piattaforme, possono inoltre essere ridistribuiti liberamente**.”*

TRATTO DA: Manuale di riferimento di Python versione 2.3.4

http://docs.python.it/paper-a4/ref.pdf



In dettaglio

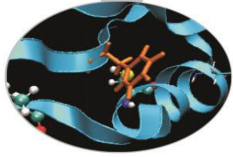
Temi trattati in maniera approfondita:

- ***linguaggio di alto livello***
- ***interpretato (e relazione con la performance)***
- ***semantica dinamica***
- ***sviluppo rapido di applicazioni***
- ***linguaggio di scripting***
- ***leggibilità***
- ***programmazione modulare***

Temi non trattati in maniera approfondita:

- ***orientato agli oggetti (solo cenni)***
- ***linguaggio collante***

Breve storia del linguaggio

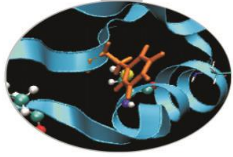


- Python è stato creato agli inizi degli anni 90 da Guido van Rossum al Centro di Matematica di Stichting (CWI, si veda http://www.cwi.nl/) nei Paesi Bassi, come successore di un linguaggio chiamato ABC. Guido rimane l'autore principale di Python, anche se molti altri vi hanno contribuito.
- Nel 1995, Guido continua il suo lavoro su Python presso il Centro Nazionale di Ricerca (CNRI, si veda http://www.cnri.reston.va.us/) in Reston, Virginia, dove ha rilasciato parecchie versioni del software.

TRATTO DA: Manuale di riferimento di Python versione 2.3.4

http://docs.python.it/paper-a4/ref.pdf

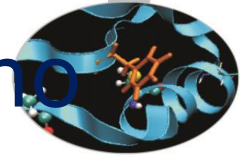
Breve storia del linguaggio



- Nell’ottobre dello stesso anno, la squadra di PythonLabs diventa la Digital Creations (ora Zope Corporation; si veda http://www.zope.com/). Nel 2001 viene fondata la Python Software Foundation (PSF, si veda http://www.python.org/psf/), un’organizzazione senza scopo di lucro creata specificamente per detenere la proprietà intellettuale di Python. Zope Corporation è un membro sponsorizzato dalla PSF.

- Tutti i rilasci di Python sono Open Source (si veda http://www.opensource.org/ per la definizione di “Open Source”).

Documentazione ufficiale e in italiano



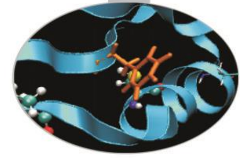
- La documentazione ufficiale del linguaggio la si può scaricare da:

http://www.python.org/doc/current/

- Esistono numerosi documenti tradotti in Italiano consultabili da:

http://docs.python.it/

Ordine del corso



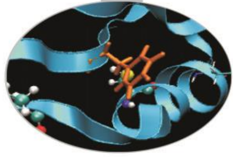
Corso introduttivo al linguaggio, strutturato in 11 moduli:

- Presentazione del linguaggio: modulo 1
- IDE e installazioni: modulo 2
- Tipi di variabili: modulo 3
- Cicli e Costrutti: modulo 4
- Dati strutturati: modulo 5
- Funzioni: modulo 6
- File e Directory: modulo 7
- Classe: modulo 8
- Moduli scientifici: modulo 9
- Debug, Profiling e Testing: modulo10
- Mixed Programming Language: modulo 11
- Approfondimenti: modulo 12

Ogni modulo prevede una parte di esercizi o di hands-on

E' previsto lo sviluppo di una applicazione finale di media dimensione sviluppabile a gruppi

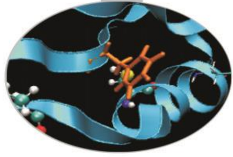
CINECA SCAI
SuperComputing Applications and Innovation



Modulo 1: introduzione al linguaggio

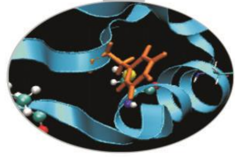
CINECA

Ordine del modulo



1. Linguaggi interpretati e compilati
2. Caratteristiche generali di python
3. Applicazioni scientifiche: motivazioni
4. Python VS Matlab ®
5. Esempi e considerazioni

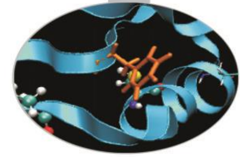
Due famiglie di linguaggi

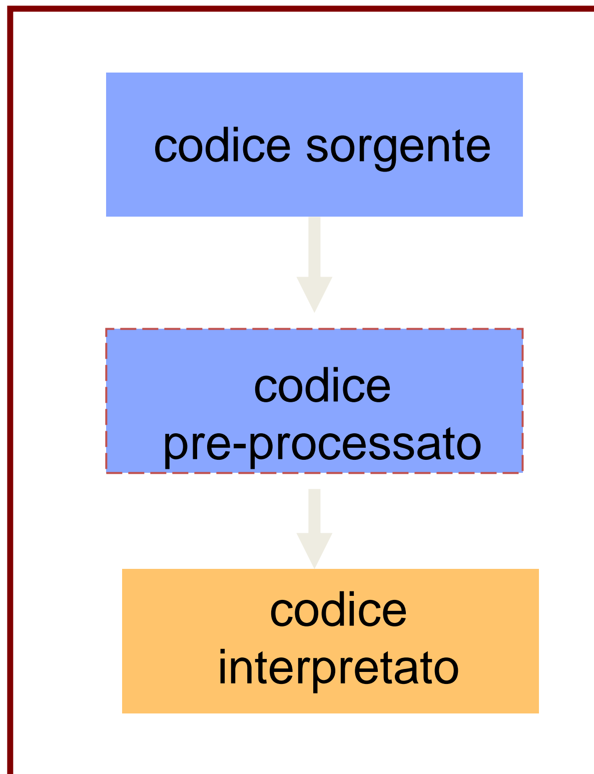


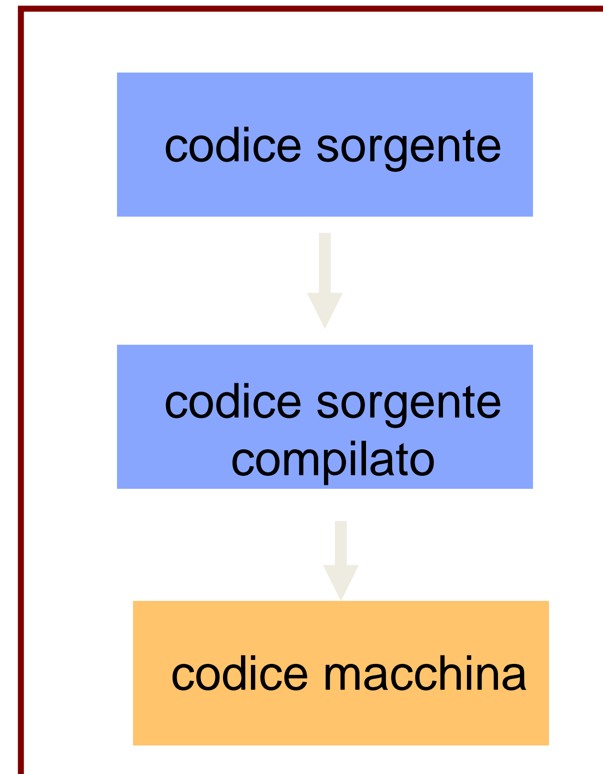
- Linguaggi interpretati vs linguaggi compilati

Interpretato	Compilato
Python	C/C++
Matlab ®	Fortran
Perl	Java

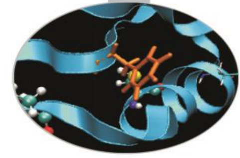
Due famiglie di linguaggi





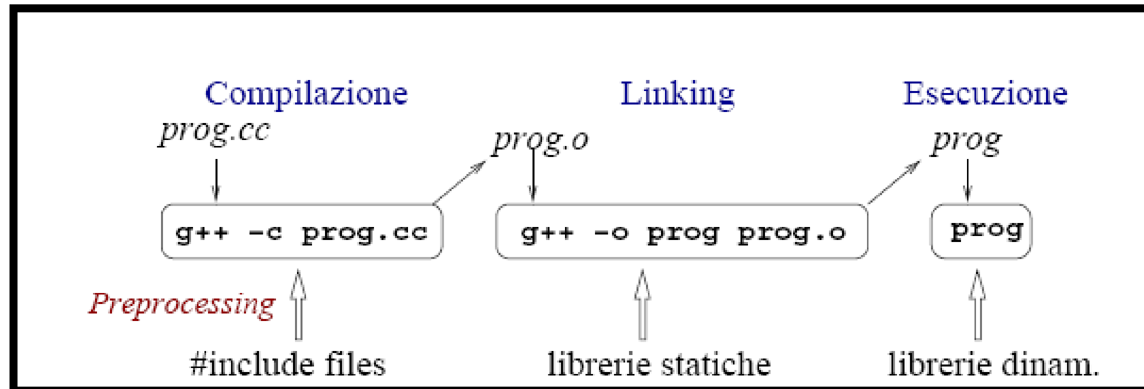


Interprete VS Compilatore



COMPILATORE

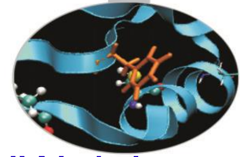
- Il processo di compilazione è costituito da diverse fasi



- La prima fase genera un file object .o a partire dai codici sorgenti .c
- La seconda fase genera l'eseguibile a partire dai file object .o
- Eventuali librerie sono linkate staticamente o dinamicamente a run-time.

L'eseguibile generato è machine-dependent

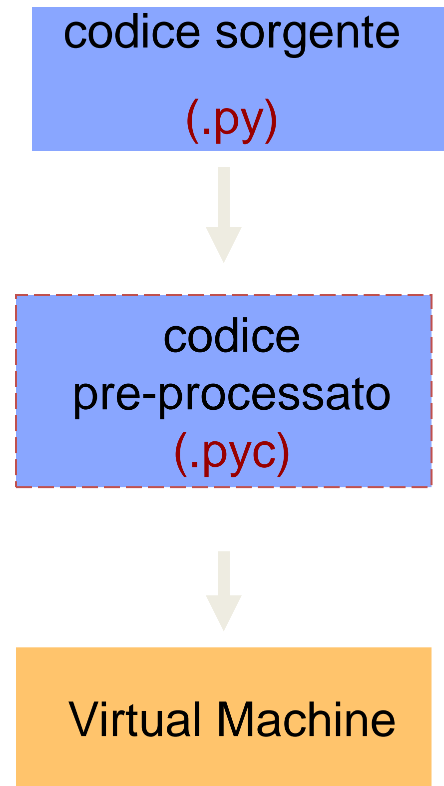
Python Virtual Machine



L'interprete è uno strato di software che si pone a metà tra il codice utente e l'HW del processore che esegue i comandi

L'installazione base di python prevede infatti almeno 2 componenti fondamentali:

- l'interprete
- set di librerie base

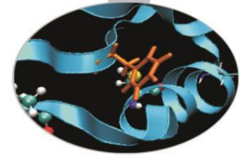


Python userà I pre-compilati per ogni eseguzione succesiva se non avvengono modifiche al codice sorgente (time-stamp)

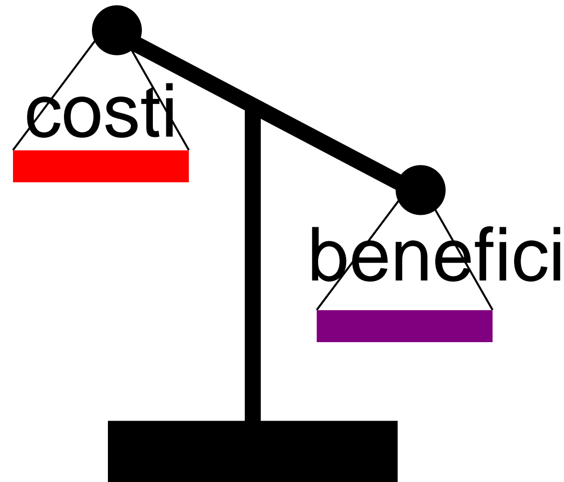
La PVM è di fatto la componente che esegue a livello macchina le singole istruzioni contenute nel .pyc.

Di fatto è una componente interna al linguaggio che tramite un grande ciclo itera lungo tutte le istruzioni

Costi e Benefici

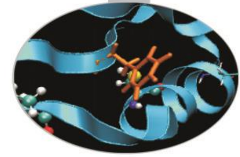


- Come visto l'interprete python (con PVM annessa) lavora in buona sostanza come un compilatore classico. La sola vera differenza risiede nel fatto che il codice, dopo essere stato pre-processato viene eseguito immediatamente; tuttavia I files .pyc sono diversi dai file .exe del c/c++.
- Il compilatore infatti richiede uno step aggiuntivo e il tempo di compilazione può variare considerevolmente in base al livello di ottimizzazione richiesto.
- Il codice eseguibile nel caso di un compilatore risulta generalmente più performante.
- Python in questo senso grazie al meccanismo della PVM si pone a metà via tra i linguaggi compilati (C/C++) e quelli interpretati classici (senza PVM).



SuperComputing Applications and Innovation

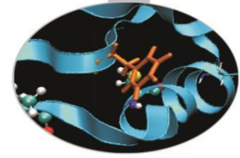
Costi e Benefici



- Velocità di esecuzione: i linguaggi compilati hanno in generale prestazioni migliori dei linguaggi interpretati (compilazione ottimizzata)
- Velocità di sviluppo: i linguaggi interpretati sono di alto livello, la messa a punto del codice è più semplice, sia per semplicità di programmazione, sia perché gli interpreti permettono di correggere gli errori non appena vengono scoperti, senza necessità di ricompilazione
- Portabilità del codice: su piattaforme hardware/software diverse da quelle su cui il codice è stato sviluppato.
  - Per un codice compilato:
    - Portabilità dell'eseguibile su piattaforme con hardware, software simile e quello su cui il codice è stato compilato
    - Portabilità tramite ricompilazione: esistenza di compilatori e librerie.
  - Per un codice interpretato:
    - Esistenza di un interprete sulla nuova piattaforma.

CINECA SCAI
SuperComputing Applications and Innovation

Due famiglie di linguaggi



Tiobe Programming Paradigm Index: Type system

Language Paradigms Used (%)

72.5
70.0
67.5
65.0
62.5
60.0
57.5
55.0
52.5
50.0
47.5
45.0
42.5
40.0
37.5
35.0
32.5
30.0
27.5

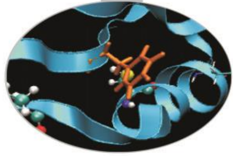
2002 2003 2004 2005 2006 2007 2008 2009

Dates

statically typed — dynamically typed

CINECA

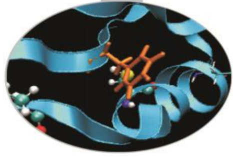
Definizioni alternative



Possibile definire I linguaggi distinguendoli anche per utilizzo o per caratteristiche macroscopiche. In questo caso si potrebbe parlare di linguaggi:

- *Type-safe (Fortran C/C++) VS dynamically typed (Python, Perl, PHP):* stando ad indicare che la dichiarazione delle variabile deve seguire regole opposte nei due casi
- *System VS scripting*: stando ad indicare la natura dell'utilizzo tipico
- *High level VS low level:* stando ad indicare livelli di vicinanza/lontananza dalla macchina e conseguenti livelli di astrazione

Tipizzazione



- Tipizzazione statica:

Alcuni linguaggi come il C/C++ o il Fortran adottano una tipizzazione statica.

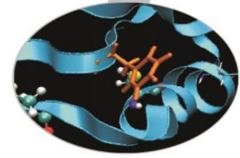
A ciascuna variabile è associato un tipo nell'atto della dichiarazione e non può cambiare nell'ambito dello scope di quel nome.

Una variabile può essere associato a diversi oggetti nell'ambito del suo scope, purché questi oggetti abbiano tutti lo stesso tipo.

Esempio:

```
void prova(){
        float a=5.0;
}
int main(){
        int a=0;
        int b=5;
        b=a;
}
```

Tipizzazione



- Tipizzazione dinamica:

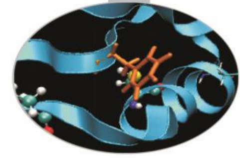
  Nella maggior parte dei linguaggi interpretati la tipizzazione è dinamica.

  Non c'è necessità di dichiarare il tipo associato ad una variabile.

  Una variabile può essere associata a più dati differenti.

```
>>> a = 4
>>> print type(a)
<type 'int'>
>>> a = 4.5
>>> print type(a)
<type 'float'>
>>> a = "sono una stringa"
>>> print type(a)
<type 'str'>
>>> a = 1, 2
>>> print type(a)
<type 'tuple'>
```

Tipizzazione

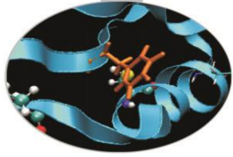


- Tipizzazione Forte

Le espressioni a cui un oggetto può prendere parte dipendono dal tipo dell’oggetto. Un tipizzazione forte fa sì che un’operazione effettuata con tipi di dato non coerenti non possa essere svolta.

```
>>> a=5
>>> b=4
>>> a+b
9
>>> c=5.6
>>> a+c
10.6
>>> s='9'
>>> a+s
Traceback (most recent call last):
  File "<pyshell#195>", line 1, in <module>
    a+s
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Tipizzazione



- <u>Tipizzazione Debole:</u>

  Linguaggi in cui il tipo di dato può essere ignorato. Operazioni tra dati diversi sono consentiti.

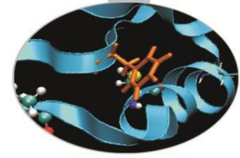
PERL

Print ’3’’+4

PYTHON

```
print '3'+4
Traceback (most recent call last):
  File "<pyshell#196>", line 1, in <module>
    print '3'+4
TypeError: cannot concatenate 'str' and 'int' objects
```

Duck typing/Nominal Typing



Nominal Typing

- Buona parte dei linguaggi compilati eseguono un typing nominale.

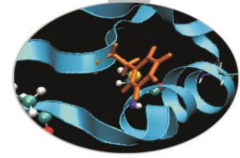
Due oggetti si riferiscono allo steso tipo se le loro dichiarazioni sono identiche. Il tipo è noto a tempo di compilazione.

Duck Typing

- "if it walks like a duck, and quacks like a duck, then it is a duck"
- Nel duck – typing il controllo sul tipo è fatto solo a tempo di esecuzione

```
>>> l = [1, 2, 3]
>>> d = { 0: 0, 'a': 2, 'b': 4 }
>>> i = 4
>>> print l[0] + 3, d[0]+3
(4,3)
>>> print i[0] + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is unsubscriptable
```

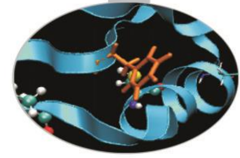
The Zen of Python



- Python è un linguaggio di programmazione con molti aspetti positivi:
  - Grande semplicità d'uso
  - Grande semplicità di apprendimento (assomiglia alla pseudocodifica)
  - Grande leggibilità (c'è un solo modo per fare qualsiasi cosa)
  - Grande portabilità.
- Python è un linguaggio multiparadigma
  - Imperative
  - Objectoriented
  - Functional
  - Structural
  - Aspectoriented

NON E' UN SEMPLICE LINGUAGGIO DI SCRIPTING!!

The Zen of Python



```
import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

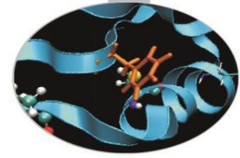
Although never is often better than *right* now.

If the implementation is hard to explain, it's a bad idea.

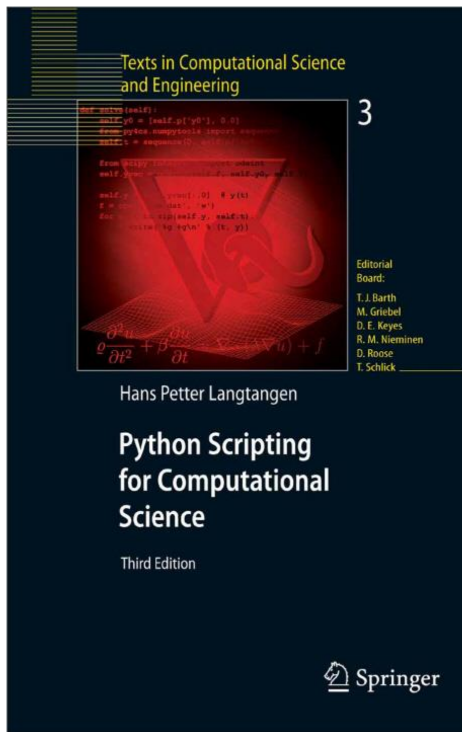
If the implementation is easy to explain, it may be a good idea.

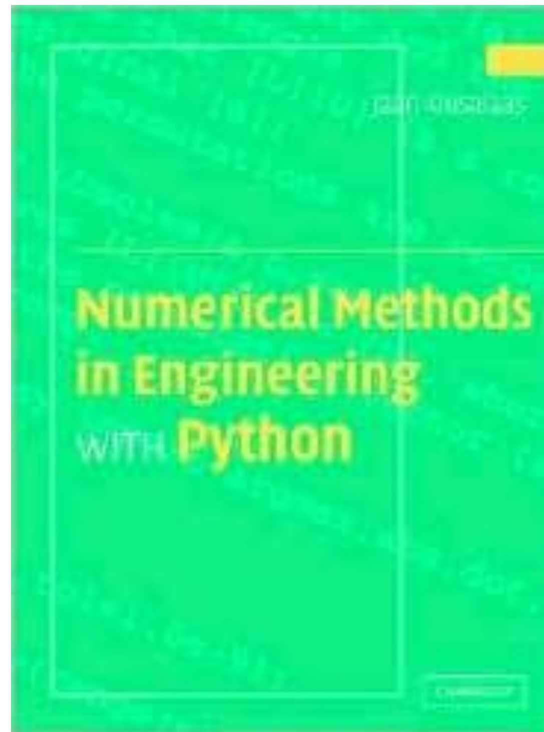
Namespaces are one honking great idea -- let's do more of those!

Python in ambito scientifico

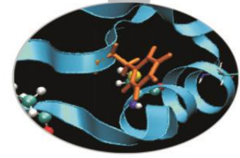


- Questo corso tratta di script come strumento di lavoro nella programmazione in ambito scientifico tecnico attraverso Python.

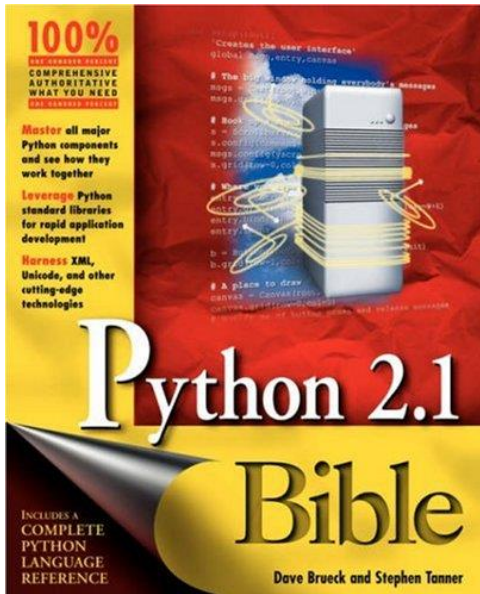


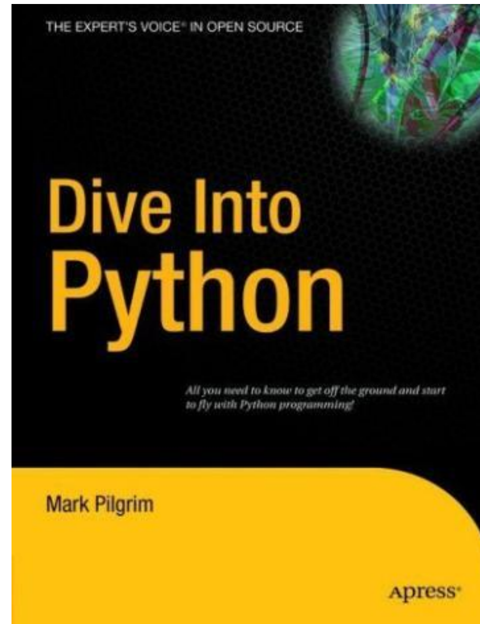


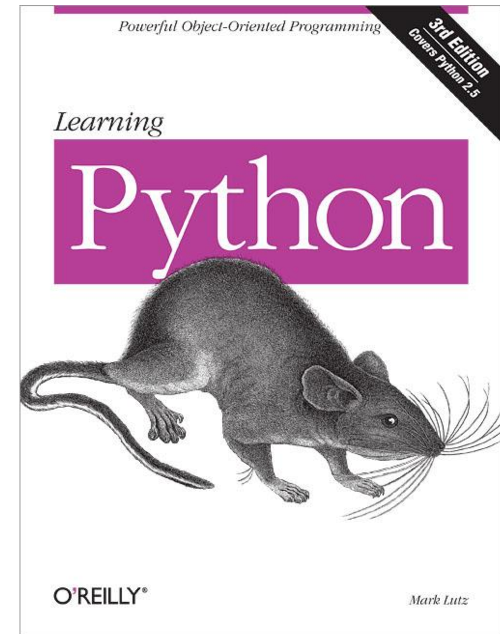
Python in ambito scientifico



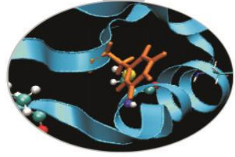
- Necessita di una conoscenza di base del linguaggio







Scripting interpretati e applicazioni scientifiche: motivazioni



Esistono tratti comuni a molti linguaggi/ambienti di sviluppo in ambito scientifico (Maple, Mathematica, Matlab, S-Plus/R, Python) che hanno invogliato gli sviluppatori in ambito scientifico ad utilizzarli:

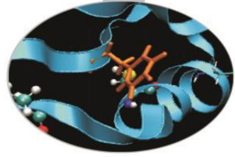
- sintassi più semplice e pulita
- connessione diretta tra simulazioni e visualizzazione

Tuttavia non tutti questi ambienti presentano sufficiente

facilità di comunicazione con altri linguaggi/ambienti in ambito scientifico.

Python sì

Python per applicazioni scientifiche



- [Python Success Stories]

AstraZeneca Uses Python for Collaborative Drug Discovery

MayaVi Uses Python for Scientific Data Visualization

ForecastWatch.com Uses Python To Help Meteorologists

Python in The Blind Audio Tactile mapping System

Python Streamlines Space Shuttle Mission Design

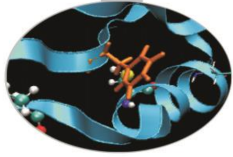
Carmanah Lights the Way with Python

Google

Youtube

… e tanti altri

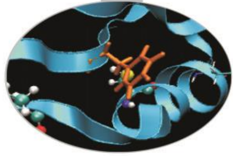
Python come collante



Python oltre ad offrire le caratteristiche sopracitate presenta anche le caratteristiche non banali di:

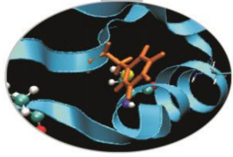
- essere un ottimo collante tra ambienti/librerie/codici sviluppati in altri linguaggi per l'ambito scientifico (C, C++ e Fortran) e/o la visualizzazione
- permettere di eseguire con semplicità la 'traduzione' tra diversi formati di grandi quantità di dati gestendo files e cartelle interagendo con il sistema operativo
- permettere di associare con relativa semplicità delle interfacce (GUI) alle applicazioni sviluppate
- essere altamente portabile (Unix, Linux, Mac, Windows)

Python Vs Matlab



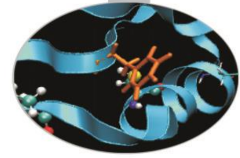
- Python per l'ambito scientifico presenta molte caratteristiche simili a Matlab; per i seguenti motivi è però superiore:
- maggiore potenza e flessibilità del linguaggio
- ambiente completamente open e integrabile con applicazioni esterne
- moduli compatti contenenti numerosi funzioni
- passaggio di funzioni come argomenti semplificato
- gestione semplificata di strutture dati annidate
- Object Oriented Programming (OOP) miglior supporto con C, C++ Fortran
- funzioni scalari che lavorano spesso anche su array senza modifiche
- sorgenti gratuiti e multipiattaforma

Python vs Matlab



Tuttavia va sottolineato che Matlab (essendo specifico al calcolo scientifico e alla visualizzazione):

- ha una documentazione superiore
- ha un maggior numero di routines per l'algebra lineare e la soluzione di ODE, l'ottimizzazione, l'analisi dei segnali, etc.
- ha degli strumenti per la visualizzazione 2D/3D migliori e più stabili.



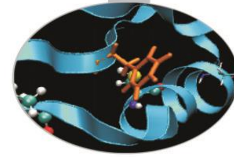
Statistiche di utilizzo

Position Oct 2012	Position Oct 2011	Delta in Position	Programming Language	Ratings Oct 2012	Delta Oct 2011	Status
1	2	↑	C	19.822%	+2.11%	A
2	1	↓	Java	17.193%	-0.72%	A
3	6	↑↑↑	Objective-C	9.477%	+3.23%	A
4	3	↓	C++	9.260%	+0.19%	A
5	5	=	C#	6.530%	-0.19%	A
6	4	↓↓	PHP	5.669%	-1.15%	A
7	7	=	(Visual) Basic	5.120%	+0.57%	A
8	8	=	Python	3.895%	-0.05%	A
9	9	=	Perl	2.126%	-0.31%	A
10	11	↑	Ruby	1.802%	+0.28%	A
11	10	↓	JavaScript	1.261%	-0.93%	A
12	12	=	Delphi/Object Pascal	1.097%	-0.01%	A
13	13	=	Lisp	0.947%	-0.08%	A
14	18	↑↑↑↑	Pascal	0.839%	+0.12%	A
15	16	↑	Lua	0.728%	-0.07%	A
16	20	↑↑↑↑	Ada	0.654%	+0.04%	B
17	15	↓↓	PL/SQL	0.630%	-0.27%	B
18	25	↑↑↑↑↑↑↑	Visual Basic .NET	0.599%	+0.12%	A--
19	21	↑↑	MATLAB	0.591%	+0.02%	B
20	19	↓	Assembly	0.568%	-0.05%	B

Tratto da: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

CINECA SCAI

SuperComputing Applications and Innovation



Trends

Tiobe Programming Community Index

Normalized fraction of total hits (%)

27.5
25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0

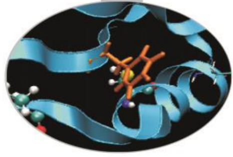
2002 2003 2004 2005 2006 2007 2008 2009

Dates

Java, C++, (Visual) Basic, C#, JavaScript, C, PHP, Python, Perl, Ruby

CINECA

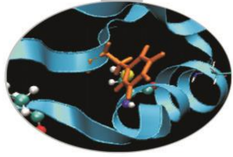
Prospettive nei prossimi 10 anni



Difficile fare previsioni, ma i trend dicono che chi sviluppa applicazioni scientifiche chiede naturalmente nuove caratteristiche di flessibilità oltre che di performance al linguaggio candidato:

- C++
- Fortran 2003 (OOP)
- Python
- Java
- PHP
- Perl
- Jython
- Ruby
- ….

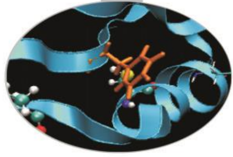
L’interprete



- Python è un linguaggio interpretato
- L'interprete esegue una compilazione del sorgente in bytecode, che viene poi eseguito su una virtual machine.
- L’interprete è molto utile anche usato in interattivo.
- Qualsiasi errore possa avvenire nel corso dell'esecuzione dell'interprete in interattivo, l'interprete sopravvive, anche in caso di SyntaxError:

```
>>> 50/0
Traceback (most recent call last):
  File "<pyshell#199>", line 1, in <module>
    50/0
ZeroDivisionError: integer division or modulo by zero
>>> def func()
SyntaxError: invalid syntax
```

Come passare gli argomenti

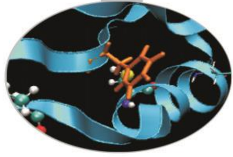


```
#!/usr/bin/env python
import math
infile='mydata.dat'
outfile='myout.dat'
indata = open( infile, 'r')
linee=indata.readlines()
indata.close()
processati=[ ]
x=[ ]
for el in linee:
	valori = el.split()
	x.append(float(valori[0])); y = float(valori[1])
	processati.append(f(y))

outdata = open(outfile, 'w')
i=0
for el in processati:
	outdata.write('%g %12.5e\n' % (x[i],el))
	i+=1
outdata.close()
```

```
def f(y):
	if y >= 0.0:
		return y**5*math.exp(-y)
	else:
		return 0.0
```

Come passare gli argomenti



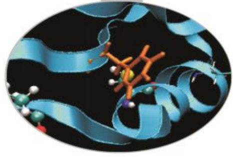
possibili varianti sono:

```
import sys
try:
        infile = sys.argv[1]; outfile = sys.argv[2]
except:
        print "Richieste di: ",sys.argv[0], "nome file IN e nome file OUT"    sys.exit(1)
```

oppure:

```
infile = raw_input( “inserire nome file IN: “ )
outfile = raw_input( “inserire nome file OUT: “ )
```

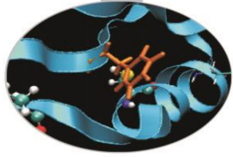
Esempio-1



Costruire N files di testo che servono da input per altri programmi

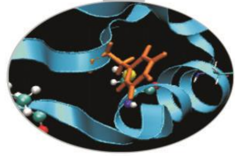
```
"""
questo script produce un journal files che legge un .cas e poi ciclicamente legge tutti i
    .dat presenti e fa scrivere le componenti di velocità ed i gradienti su testo
"""
start=401
stop=481
passo=1
nomejou = 'post_txt_rm_arco6M-400.jou'
nomecas= "./aorta_6M.cas"
nomedat = "/data/bkponzini/DAT-U-6M/aorta_6M-"
des=".dat.gz"
nometxt = "/data/bkponzini/TXT-U-6M/export_txt"
o=open(nomejou,'w')
o.write("/file/read-case/"+nomecas+'\n')
```

Esempio-1



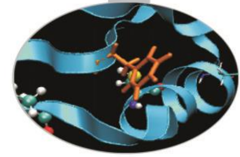
```
for i in range(start,stop+passo,passo):
    if i<10:
        o.write("/file/read-data/"+nomedat+"000"+str(i)+des+'\n')
        o.write("/file/export/ascii "+nometxt+str(i)+".txt"+'\n')
        o.write("default-interior \n \n no \n x-velocity y-velocity z-velocity
+\n dx-velocity-dx \n dy-velocity-dx \n dz-velocity-dx \
                \n dx-velocity-dy \n dy-velocity-dy \n dz-velocity-dy \n dx-velocity-dz
+\n dy-velocity-dz \n dz-velocity-dz \n q \n yes \n \n”)
        o.write('!rm '+nomedat+"000"+str(i)+des+'\n')
    elif (i>=10 and i<100):
        o.write("/file/read-data/"+nomedat+"00"+str(i)+des+'\n')
        o.write("/file/export/ascii "+nometxt+str(i)+".txt"+'\n')
        o.write("default-interior \n\n no \n x-velocity \n y-velocity \n z-velocity
+\n dx-velocity-dx \n dy-velocity-dx \n dz-velocity-dx \
                \n dx-velocity-dy \n dy-velocity-dy \n dz-velocity-dy \n dx-velocity-dz
+\n dy-velocity-dz \n dz-velocity-dz \n q \n yes \n \n”)
        o.write('!rm '+nomedat+"00"+str(i)+des+’\n’)
```

Esempio-1



```
elif(i>=100 and i<1000):
        o.write("/file/read-data/"+nomedat+"0"+str(i)+des+'\n')
        o.write("/file/export/ascii "+nometxt+str(i)+".txt \n')
        o.write("default-interior \n \n no \n x-velocity \n y-velocity \n z-velocity
+\n dx-velocity-dx \n dy-velocity-dx \n dz-velocity-dx
            \n dx-velocity-dy \n dy-velocity-dy \n dz-velocity-dy \n dx-velocity-dz
+\ndy-velocity-dz \n dz-velocity-dz \n q \n yes \n \n”)
        o.write('!rm '+nomedat+"0"+str(i)+des+'\n')
    elif(i>=1000 and i<10000):
        o.write("/file/read-data/"+nomedat+str(i)+des+'\n')
        o.write("/file/export/ascii "+nometxt+str(i)+".txt \n')
        o.write("default-interior \n\n no \n x-velocity \n y-velocity \n z-velocity
+\ndx-velocity-dx \n dy-velocity-dx \n dz-velocity-dx \
            \n dx-velocity-dy \n dy-velocity-dy \n dz-velocity-dy \n dx-velocity-dz
+\ndy-velocity-dz \n dz-velocity-dz \n q \n yes \n\n”)
        o.write('!rm '+nomedat+str(i)+des+'\n')
    else:
        print 'probably to much files, isnt??'
o.write("/exit y")
o.close()
```

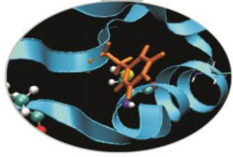
Import di moduli



- Un file con terminazione .py costituisce un modulo per Python
- Un modulo può contenere qualsiasi tipo di codice python

```
$ more my.py
def hello_world():
  print “Ciao, mondo!”
$ python
>>> import my
>>> dir(my)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
'hello_world']
>>> print my.__file__
my.py
>>> my.hello_world()
Ciao, mondo!
```

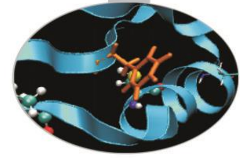
Import di moduli



- Python è costituito da una serie di moduli, forniti dalla installazione di base oppure aggiunti esternamente (tipico dei moduli scientifici e di visualizzazione)
- Per importare i moduli, siano essi di sistema o aggiunti in seguito, si usa il comando di sistema from/import con le 3 possibili sintassi:

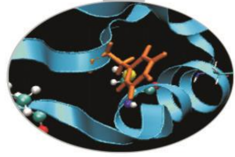
```
from os import *
from os import
from os import path as PP
import os
import os as O
```

Esempio-2.0

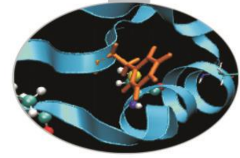


```
import os
#importo lo spazio dei nomi del modulo os
>>> os.curdir
'.'
>>> os.getenv('HOME')
'C:\\Documents and Settings\\ponzini.CILEA-DOM.000'
>>> os.listdir('.')
['aboutDialog.py', 'aboutDialog.pyc', 'AutoComplete.py', 'AutoComplete.pyc',
    'AutoCompleteWindow.py', 'AutoCompleteWindow.pyc', 'AutoExpand.py',
    'AutoExpand.pyc', 'Bindings.py', 'Bindings.pyc', 'CallTips.py', 'CallTips.pyc',
    'CallTipWindow.py', 'CallTipWindow.pyc', 'Clas….]
>>> os.defpath
'.;C:\\bin'
```

Esempio-2.1



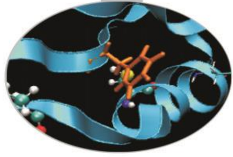
```
import os as O
#importo lo spazio dei nomi del modulo os come O
>>> O.curdir
'.'
>>> O.getenv('HOME')
'C:\\Documents and Settings\\ponzini.CILEA-DOM.000'
>>> O.listdir('.')
['aboutDialog.py', 'aboutDialog.pyc', 'AutoComplete.py', 'AutoComplete.pyc',
    'AutoCompleteWindow.py', 'AutoCompleteWindow.pyc', 'AutoExpand.py',
    'AutoExpand.pyc', 'Bindings.py', 'Bindings.pyc', 'CallTips.py', 'CallTips.pyc',
    'CallTipWindow.py', 'CallTipWindow.pyc', 'Clas....]
>>> O.defpath
'.;C:\\bin'
```



Esempio-2.3

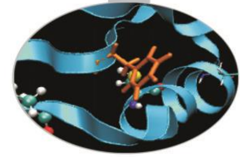
```
from os import *
#importo tutto dallo spazio dei nomi del modulo os
>>> curdir
'.'
>>> getenv('HOME')
'C:\\Documents and Settings\\ponzini.CILEA-DOM.000'
>>> listdir('.')
['aboutDialog.py', 'aboutDialog.pyc', 'AutoComplete.py', 'AutoComplete.pyc',
    'AutoCompleteWindow.py', 'AutoCompleteWindow.pyc', 'AutoExpand.py',
    'AutoExpand.pyc', 'Bindings.py', 'Bindings.pyc', 'CallTips.py', 'CallTips.pyc',
    'CallTipWindow.py', 'CallTipWindow.pyc', 'Clas….]
>>> defpath
'.;C:\\bin'
```

Esempio-2.4



```
from os import curdir as CC
#importo un elemento specifico con un alias
>>> CC
'.'
>>> from os import getenv as GG
>> GG ('HOME')
'C:\\Documents and Settings\\ponzini.CILEA-DOM.000'
```

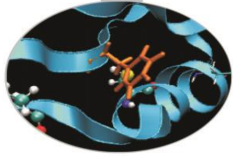
Import di moduli



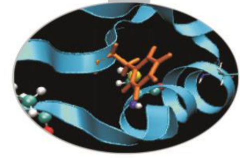
Moduli possono essere raggruppati in pacchetti, che hanno una struttura gerarchica rappresentata da directory. Una directory contenente un file __init__.py, eventualmente vuoto, è un pacchetto. Se la directory contiene altri pacchetti o moduli, essi sono accessibili come contenuto del pacchetto.

```
sound/                      Toplevel package
      __init__.py           Initialize the sound package
      formats/              Subpackage for file format conversions
              __init__.py wavread.py wavwrite.py
              aiffread.py aiffwrite.py auread.py
              auwrite.py  ...
      effects/              Subpackage for sound effects
              __init__.py echo.py surround.py
              reverse.py  ...
      filters/              Subpackage for filters
              __init__.py equalizer.py vocoder.py
              karaoke.py  ...
```

Import di moduli

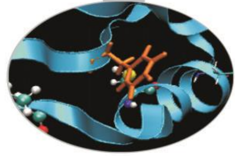


```
>>> import sound.effects.echo.echofilter
>>> sound.effects.echo.echofilter(...)
>>> import sound.effects.echo
>>> sound.effects.echo.echofilter(...)
>>> from sound.effects import echo
>>> echo.echofilter(...)
>>> from sound.effects.echo import echofilter
>>> echofilter(...)
```



Commenti

- Tutto è oggetto in python
  - cosa è un oggetto?



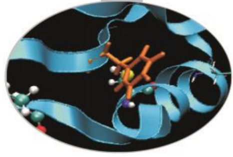
Sintassi di un oggetto

Alcune nozioni base sulla sintassi degli oggetti e sulle modalità di accesso ai membri, prima di procedere formalmente alla definizione delle classi:

- accesso alle funzioni membro:
  - *nome_oggetto.nome_funzione_membro()*
- accesso ai dati membro:
  - *nome_oggetto.nome_dato_membro*

```
>>> a
[1, 2, 3, 4]
>>> a.count(1)
1
>>> a.__doc__
"list() -> new list\nlist(sequence) -> new list initialized from sequence's items"
>>> a=numpy.array([1,2,3])
>>> a.shape
(3,)
>>> a.dtype
dtype('int32')
```

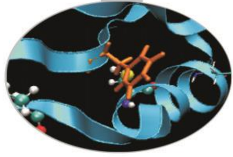
Commenti



Anche un modulo è un oggetto:

- quindi possiamo guardare nel modulo e interrogare I suoi dati membro:

```
>> import pickle as P
>> print P.__name__
pickle
>> print P.__version__
```



Commenti

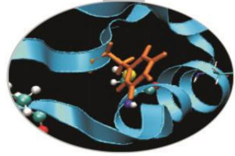
```
>>> print P.__doc__
Create portable serialized representations of Python objects.

See module cPickle for a (much) faster implementation.
See module copy_reg for a mechanism for registering custom picklers.
See module pickletools source for extensive comments.

Classes:

    Pickler
    Unpickler
```

Commenti



Functions:

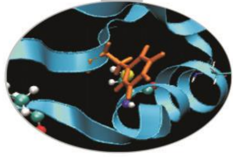
dump(object, file)
dumps(object) -> string
load(file) -> object
loads(string) -> object

Misc variables:

__version__
format_version
compatible_formats

```
>>> print P.__version__
$Revision: 38432 $
>>>
```

Introspezione



- L'introspezione è la capacità di un linguaggio di fornire varie

informazioni sugli oggetti runtime.

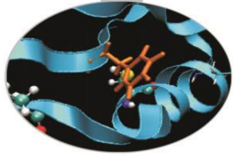
- Python ha un ottimo supporto per l'introspezione, a differenza di

linguaggi come Fortran o C che non ne hanno alcuno, o C++ che

ha un supporto estremamente limitato.

- Usare l'interprete in interattivo per fare introspezione facilita la comprensione del codice e del linguaggio.

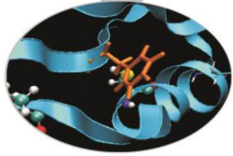
Introspezione



- Esiste un comando di sistema che permette di conoscere tutti I dati e le funzioni contenute in un modulo:

```
>> dir (P)
['APPEND', 'APPENDS', 'BINFLOAT', 'BINGET', 'BININT', 'BININT1', 'BININT2', 'BINPERSID', 'BINPUT',
 'BINSTRING', 'BINUNICODE', 'BUILD', 'BooleanType', 'BufferType', 'BuiltinFunctionType',
 'BuiltinMethodType', 'ClassType', 'CodeType', 'ComplexType', 'DICT', 'DUP', 'DictProxyType',
 'DictType', 'DictionaryType', 'EMPTY_DICT', 'EMPTY_LIST', 'EMPTY_TUPLE', 'EXT1', 'EXT2',
 'EXT4', 'EllipsisType', 'FALSE', 'FLOAT', 'FileType', 'FloatType', 'FrameType', 'FunctionType',
 'GET', 'GLOBAL', 'GeneratorType', 'GetSetDescriptorType', 'HIGHEST_PROTOCOL', 'INST', 'INT',
 'InstanceType', 'IntType', 'LIST', 'LONG', 'LONG1', 'LONG4', 'LONG_BINGET', 'LONG_BINPUT',
 'LambdaType', 'ListType', 'LongType', 'MARK', 'MemberDescriptorType', 'MethodType',
 'ModuleType', 'NEWFALSE', 'NEWOBJ', 'NEWTRUE', 'NONE', 'NoneType',
 'NotImplementedType', 'OBJ', 'ObjectType', 'PERSID', 'POP', 'POP_MARK', 'PROTO', 'PUT',
 'PickleError', 'Pickler', 'PicklingError', 'PyStringMap', 'REDUCE', 'SETITEM', 'SETITEMS',
 'SHORT_BINSTRING', 'STOP', 'STRING', 'SliceType', 'StringIO', 'StringType', 'StringTypes',
 'TRUE', 'TUPLE', 'TUPLE1', 'TUPLE2', 'TUPLE3', 'TracebackType', 'TupleType', 'TypeType',
 'UNICODE', 'UnboundMethodType', 'UnicodeType', 'Unpickler', 'UnpicklingError',
 'XRangeType', '_EmptyClass', '_Stop', '__all__', '__builtins__', '__doc__', '__file__',
 '__name__', '__version__', '_binascii', '_extension_cache', '_extension_registry',
 '_inverted_registry', '_keep_alive', '_test', '_tuplesize2code', 'classmap',
 'compatible_formats', 'decode_long', 'dispatch_table', 'dump', 'dumps', 'encode_long',
 'format_version', 'load', 'loads', 'marshal', 'mloads', 're', 'struct', 'sys', 'whichmodule']
```

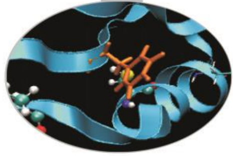
Commenti



ovviamente anche una classe di un modulo è un oggetto:

```
>> dir(P.Unpickler)
['__doc__', '__init__', '__module__', '_instantiate', 'dispatch', 'find_class',
    'get_extension', 'load', 'load_append', 'load_appends', 'load_binfloat',
    'load_binget', 'load_binint', 'load_binint1', 'load_binint2', 'load_binpersid',
    'load_binput', 'load_binstring', 'load_binunicode', 'load_build', 'load_dict',
    'load_dup', 'load_empty_dictionary', 'load_empty_list',
    'load_empty_tuple', 'load_eof', 'load_ext1', 'load_ext2', 'load_ext4',
    'load_false', 'load_float', 'load_get', 'load_global', 'load_inst', 'load_int',
    'load_list', 'load_long', 'load_long1', 'load_long4', 'load_long_binget',
    'load_long_binput', 'load_mark', 'load_newobj', 'load_none', 'load_obj',
    'load_persid', 'load_pop', 'load_pop_mark', 'load_proto', 'load_put',
    'load_reduce', 'load_setitem', 'load_setitems', 'load_short_binstring',
    'load_stop', 'load_string', 'load_true', 'load_tuple', 'load_tuple1',
    'load_tuple2', 'load_tuple3', 'load_unicode', 'marker']
```

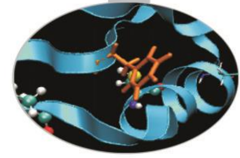
Uso dei dati membro built-in in main



Di questi dati membro propri di ogni modulo si fa largo uso in python soprattutto per quanto riguarda l'importazione di un modulo:

```
if __name__== '__main__':
        print 'Il chiamante sono proprio io'
else:
        print 'sono stato importato da un altro modulo'
```

help()



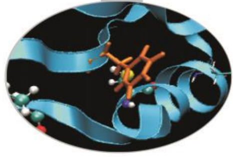
Analogamente a quanto mostrato per la funzione di sistema dir() che ha validità generale per ogni modulo/oggetto, è anche disponibile una funzione di sistema per conoscere la documentazione su una generica funzione:

```
import pickle
>>> help(pickle.dump)
Help on function dump in module pickle:

dump(obj, file, protocol=None)

>>> help(pickle.load)
Help on function load in module pickle:
```

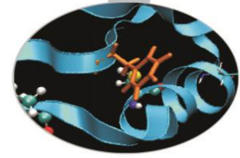
Type()



- Determinare il tipo di un'oggetto in Python è estremamente facile: basta usare il comando type.

```
>>>a=5
>>>type(a)
<type'int'>
>>> l = [1, "alfa", 0.9, (1, 2, 3)]
>>> print [type(i) for i in l]
[<type 'int'>, <type 'str'>, <type
'float'>, <type 'tuple'>]
>>>
```

pydoc

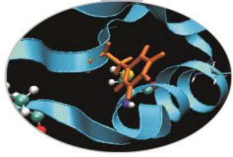


- Pydoc è un tool, ovviamente scritto in python, che utilizza l'introspezione per fornire le informazioni racchiuse in un modulo in maniera chiara e compatta.
- Pydoc utilizza le doc string __doc__ ed i vari altri attributi

standard che gli oggetti hanno (__name__, __file__, ...).

```
[alinve@lagrange ~]$ pydoc os
Help on module os:
NAME
    os - OS routines for Mac, DOS, NT, or Posix depending on what system we're on.
FILE
    /usr/lib64/python2.4/os.py
DESCRIPTION
    This exports:
      - all functions from posix, nt, os2, mac, or ce, e.g. unlink, stat, etc.
      - os.path is one of the modules posixpath, ntpath, or macpath
      - os.name is 'posix', 'nt', 'os2', 'mac', 'ce' or 'riscos'
      - os.curdir is a string representing the current directory ('.' or ':')
```

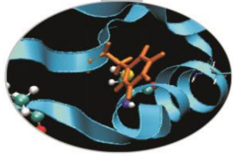
Moduli creati dall’utente



- Tutto quanto mostrato sulle funzioni di sistema per l’introspezione dei moduli, degli oggetti e delle funzioni vale anche per il codice prodotto dall’utente.

```
#file mymodule.py
“””this is the documentation section of my module
usually here I explain who is author; what the module does; the functions provided;
lincesing
“””
_eps=0.001
def myfunc(var1,var2):
“”” this function compare two values and print out the largest one:
    -IN: var1; var2
    -OUT: none
    -usage: myfunc(var1,var2)
“””
    if var1>var2:          print ‘largest value between input:’, var1
    elif var1==var2: print ‘the two value are equal’
    else:     print ‘largest value between input:’, var2
```

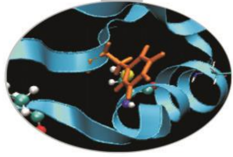
Moduli creati dall’utente



```
from mymodule import *
#import mymodule
## import mymodule as M
>> dir()
 >> ['__builtins__', '__doc__', '__name__', 'myfunc‘,’_eps’]
#dir(mymodule)
help(myfunc)
Help on function myfunc in module __main__:

myfunc(var1, var2)
    this function compare two values and print out the largest one:
    -IN: var1; var2
    -OUT: none
    -usage: myfunc(var1,var2)
#help(mymodule.myfunc)
##help(M.myfunc)
```

Test



- Dall’interprete interattivo importare il modulo math e scoprirne il contenuto.
- Usare le funzioni help e il tools pydoc per fare introspezione