

# Python avanzato per uso astronomico

## I - Introduzione

Breve carrellata su alcuni aspetti avanzati del linguaggio Python, con particolare riguardo per i packages di uso scientifico e astronomico.

- Introduzione
- Programmazione ad oggetti
- Package per algebra lineare: `numpy`
- Package “scientifico”: `scipy`
- Package grafico: `matplotlib`
- Package “astronomico”: `astropy`

## Python:

- Linguaggio scripting
  - ... ma non solo
    - Applicazioni CGI
    - Calcolo numerico (!!)
    - GUI
    - Gaming
    - Automazione
- Supporta stili di programmazione diversi
  - Procedurale
  - Ad oggetti
  - Funzionale (... un po')
- Semi-interpretato
- Tipizzato e dinamico
  - Oggetti tipizzati
  - Variabili (nomi) non tipizzate
  - Gestione memoria automatica
  - Vasta libreria standard
    - Largamente portabile
  - Uso interattivo
    - Standard
    - Con ipython

- Perl
- Php
- Ruby
- R
- Java
- Matlab (\$)
- Octave
- IDL (\$)
- ....

- Versioni
  - 2.7.x - La più diffusa. Ora congelata
  - 3.5.x - La versione “attuale”<sup>1</sup>
- Piattaforme
  - Linux (preinstallato)
  - Windows e .NET
    - Installer da [www.python.org](http://www.python.org)
    - Active State (Commerciale)
    - IronPython
  - Mac
    - Package da [www.python.org](http://www.python.org)
    - Installabile con “homebrew”
  - Android
    - python-for-android + kivy<sup>2</sup>
    - buildozer<sup>2</sup>
    - SLA4

<sup>1</sup>Possono mancare alcuni packages

<sup>2</sup>Per sviluppare “App”

Nel seguito faremo riferimento principalmente a Python 2.7 con qualche indicazione quando python 3.x presenta significative differenze.

- Linguaggio semi-interpretato
  - Gestione p-code automatica
  - `program.py` → `program.pyc`
  - `program.pyc` → `program.pyo`
  - python3: directory `__pycache__`
- Linguaggio “Object Oriented”
  - Tutti gli elementi sono oggetti
- Linguaggio “Tipizzato”
  - Oggetti tipizzati
  - Identificatori (nomi) non tipizzati
- Linguaggio dinamico
  - Non dichiarazioni, ma “creazione di oggetti”
  - “garbage collection” automatico

## Creazione dinamica di oggetti

---

```
>>> a=3      # crea l'oggetto "numero intero di valore 3"
              # e gli assegna il nome "a"

>>> a="tre"   # crea l'oggetto "stringa di caratteri 'tre'"
              # e riassume il nome "a". L'oggetto
              # "intero di valore 3" viene distrutto

>>> p=[4,2,3,1] # Crea una lista di quattro elementi e le
                # assegna il nome "p"

>>> k=p       # Assegna anche il nome k alla stessa lista
```

---

## Introspezione

---

```
>>> type(a)
<type 'str'>
>>> type(p)
<type 'list'>

>>> dir(p)    # Elenca gli attributi dell'oggetto di nome "p"
...
>>> p.sort()  # Ordina la lista di nome "p"
>>> k
[1, 2, 3, 4]
>>> del p[2]
>>> k
[1, 2, 4]
>>> help(k)
....
```

---

## Collezioni

```
>>> atuple = (1, 2, 4, "cinque", 6.0, -7, "VIII") # Collezione ‘‘immutabile’’
>>> alist = [1, "due", 3, "cinque", 7.96]          # Collezione mutabile
>>> adict = {1:"uno", 2:2, 3:3.1415926, "cinque":5} # memoria associativa
```

## Dal manuale:

Operazione	Risultato
<code>x in s</code>	<i>True</i> se un elemento di <b>s</b> è uguale a <b>x</b> , altrimenti <i>False</i>
<code>x not in s</code>	<i>False</i> se un elemento di <b>s</b> è uguale a <b>x</b> , altrimenti <i>True</i>
<code>s + t</code>	concatenazione di <b>s</b> e <b>t</b>
<code>s * n, n * s</code>	<b>s</b> + <b>s</b> + <b>s</b> + ... <b>n</b> volte
<code>s[i]</code>	i-esimo elemento di <b>s</b> (base 0)
<code>s[i:j]</code>	porzione di <b>s</b> da <b>i</b> a <b>j</b>
<code>s[i:j:k]</code>	porzione di <b>s</b> da <b>i</b> a <b>j</b> con passo <b>k</b>
<code>len(s)</code>	lunghezza of <b>s</b>
<code>min(s)</code>	il più piccolo elemento di <b>s</b>
<code>max(s)</code>	il più grande elemento di <b>s</b>
<code>s.index(x)</code>	indice della prima occorrenza di <b>x</b> in <b>s</b>
<code>s.count(x)</code>	numero di occorrenze di <b>x</b> in <b>s</b>

## Comprehension

```
>>> another = [x*x for x in atuple if type(x) is int]
>>> atuple
(1, 2, 4, "cinque", 6.0, -7, "VIII")
>>> another
[1, 4, 16, 49]
```

## file: funny.py

---

```
def funny(a, b, default=3.1415926, toint=False):  
    if toint:  
        cvt = lambda x: int(x)  
    else:  
        cvt = lambda x: x  
    if a > b:  
        return cvt(a)  
    elif a < b:  
        return cvt(b)  
    else:  
        return cvt(default)
```

---

## Qualche esempio di uso:

---

```
>>> from funny import funny  
>>> funny(1.0, 2.0)  
2.0  
>>> funny(1.0, 1.0)  
3.1415926  
>>> funny(1.0, 1.0, 6.2831852, toint=True)  
6  
>>> funny(1.0, 1.0, toint=True, default=6.2831852)  
6
```

---

## Da notare:

- Argomenti posizionali nella definizione della funzione
- Argomenti opzionali nella definizione della funzione
- Gli argomenti opzionali hanno un nome ed un valore di default
- Argomenti posizionali nella chiamata di funzione
- Argomenti opzionali nella chiamata di funzione
- Gli argomenti opzionali possono essere specificati per posizione o per nome.



file: `showarg.py`

---

```
def showarg(*pos, **opt):  
    print "Argomenti posizionali:", pos  
    print "Argomenti opzionali:", opt
```

---

## Nella definizione di funzione:

---

```
>>> from showarg import showarg  
>>> showarg(1,2,3, pippo=0, pluto="zero")  
Argomenti posizionali: (1, 2, 3)  
Argomenti opzionali: {'pippo': 0, 'pluto': 'zero'}
```

---

- `*pos`: *pos* è una tupla che “riceve” i valori degli argomenti posizionali
- `**opt`: *opt* è un dizionario che “riceve” i valori degli argomenti opzionali

## Nella chiamata di funzione:

---

```
>>> a = [1., 2.]  
>>> funny(*a)  
2.0  
>>> b = {"toint": True, "default": .999999}  
>>> funny(*a, **b)  
2
```

---

- `*pos`: *pos* è una lista o una tupla i cui elementi diventano, nell'ordine, gli argomenti posizionali della funzione.
- `**opt`: *opt* è un dizionario i cui elementi diventano i valori dei corrispondenti argomenti opzionali della funzione.

**Modulo:** blocco di codice contenuto in un file

file: `fibonacci.py`

---

```
"Modulo per il calcolo della serie di Fibonacci"
```

```
MAXFIBO=1000
```

```
def fibo(n):  
    "Riporta la serie di Fibonacci fino ad n"  
    if n > MAXFIBO: return []  
    result = []  
    a, b = 0, 1  
    while b < n:  
        result.append(b)  
        a, b = b, a+b  
    return result
```

```
def fibo_print(n):  
    "Scrive la serie di Fibonacci fino ad n"  
    serie = fibo(n)  
    print "Serie di Fibonacci fino a %d:" % n  
    print serie
```

```
if __name__=='__main__':  
    fibo_print(14)
```

---

**Package:** una gerarchia di moduli

Avremo modo di vedere le caratteristiche e le modalità di uso dei package.

## Come si usa un modulo:

---

```
>>> import fibo
>>> fibo.fibo(400)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
>>> dir(fibo)
['MAXFIBO', '__builtins__', '__doc__', '__file__', '__name__', '__package__', 'fibo']

>>> help(fibo)
Help on module fibo:

NAME
    fibo - Modulo per il calcolo della serie di Fibonacci

FILE
    /home/lfini/Personale/CorsiSeminari/2017-06-Python/varie/fibo.py

FUNCTIONS
    fibo(n)
        Riporta la serie di Fibonacci fino ad n

    fibo_print(n)
        Scrive la serie di Fibonacci fino ad n

DATA
    MAXFIBO = 1000

>>> fibo.__doc__
'Modulo per il calcolo della serie di Fibonacci'
>>> fibo.__file__
'fibo.pyc'
>>> fibo.MAXFIBO
1000
>>> fibo.fibo
<function fibo at 0x7fe98eab88c0>

>>> fibo.fibo_print(123)
Serie di Fibonacci fino a 123:
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

---

Cosa abbiamo nella pagina precedente:

- ❶ Istruzione `import`
- ❷ Esecuzione della funzione `fibonacci()` definita nel modulo `fibonacci`
- ❸ Esplorazione degli elementi del modulo `fibonacci`
  - Funzione *built-in*: `dir()`
  - Funzione *built-in*: `help()`
  - Altri elementi del modulo `fibonacci`: `__doc__`, `__file__`, ecc.
- ❹ Esecuzione della funzione `fibonacci_print` del modulo `fibonacci`

- Il *namespace* è un “contenitore” di nomi.
- All'interno di un programma esiste una gerarchia di namespace.

file: `prcubo.py`

---

```
nota = "Il cubo di %d e': %d"
```

```
cubo = 3.1415926
```

```
def print_cubo(n):  
    cubo = n*n*n  
    print nota % (n, cubo)
```

---

## Che usiamo lanciando python in modo interattivo

---

```
lfini@lfini-laptop:~/Personale/CorsiSeminari/2017-06-Python$ python
```

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
```

```
[GCC 5.4.0 20160609] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> cubo = "La terza potenza"
```

```
>>> import prcubo
```

```
>>> prcubo.print_cubo(11)
```

```
Il cubo di 11 e': 1331
```

```
>>> cubo
```

```
'La terza potenza'
```

```
>>> prcubo.cubo
```

```
3.1415926
```

```
>>> dir()
```

```
['__builtins__', '__doc__', '__name__', '__package__', 'cubo', 'prcubo']
```

```
>>> dir(prcubo)
```

```
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'cubo', 'nota',
```

---

Cosa abbiamo nella pagina precedente:

- ❶ Il namespace di livello 0 (python)
  - cubo
  - pcubo
- ❷ Il namespace del modulo pcubo
  - nota
  - cubo
  - print\_cubo
- ❸ Il namespace della funzione print\_cubo()
  - n
  - cubo

- “Scope”: ambito in cui un nome è valido
- Regole di scope:
  - Creazione di un nome: nel namespace corrente
  - Riferimento ad un nome: a partire dal namespace corrente, risalendo la gerarchia

## Rivediamo l'esempio precedente:

---

```
1. nota = "Il cubo di %d e': %d"
2. cubo = 3.1415926
3. def print_cubo(n=1):
4.     cubo = n*n*n
5.     print nota % (n, cubo)
6. ....
```

---

- Sequenza di definizione dei nomi
  - **nota** viene creato alla linea 1
  - **cubo<sub>1</sub>** viene creato alla linea 2
  - **n** viene creato alla linea 3 ed utilizzato alle linee 4 e 5
  - **cubo<sub>2</sub>** viene creato alla linea 4 e “nasconde” **cubo<sub>1</sub>**
  - **nota** alla linea 5 fa riferimento al nome creato alla linea 1
  - **n** e **cubo<sub>2</sub>** cessano di esistere alla linea 6
  - **cubo<sub>1</sub>**, definito alla linea 2, “ricompare” alla linea 6

## file: scope1.py

---

```
DEBUG = True

def main():
    print "Inizio programma ..."
    if DEBUG:
        print "Esecuzione in modo debug"

if __name__ == "__main__":
    main()
```

---

## file: scope2.py

---

```
DEBUG = False

def main():
    print "Inizio programma ..."
    if DEBUG:
        print "Esecuzione in modo debug"

if __name__ == "__main__":
    main()
```

---

scope1.py e scope2.py sono due versioni dello stesso programma. (L'unica differenza è il il valore della variabile locale DEBUG).

Si tratta di un esempio di una tecnica normalmente usata per visualizzare in modo condizionato valori utili per comprendere il funzionamento interno del programma che segue e lo scopo delle poche righe iniziali è quello di mostrare alla partenza il modo di esecuzione:

---

```
$ python scope1.py
Inizio programma ...
Esecuzione in modo debug
```

```
$ python scope2.py
Inizio programma ...
```

---



Supponiamo di voler determinare il modo di esecuzione al momento del lancio del programma senza dover modificare la variabile `DEBUG`, ad esempio lanciando il programma con un parametro opzionale: **-d**.

file: `scope3.py`

---

```
import sys

DEBUG = False

def main():
    if "-d" in sys.argv:
        DEBUG = True

    print "Inizio programma ..."
    if DEBUG:
        print "Esecuzione in modo debug"

if __name__ == "__main__":
    main()
```

---

Vediamo adesso i due modi di esecuzione:

---

```
$ python scope3.py -d
Inizio programma ...
Esecuzione in modo debug
```

---

Fin qui tutto bene, ma ...

---

```
$ python scope3.py
Inizio programma ...
Traceback (most recent call last):
  File "scope3.py", line 14, in <module>
    main()
  File "scope3.py", line 10, in main
    if DEBUG:
UnboundLocalError: local variable 'DEBUG' referenced before assignment
```

---

Come mai l'errore? ... alla pagina seguente

Ricordando le regole di *scope*:

- in `scope3.py` c'è l'assegnazione: `DEBUG = True`
- la variabile `DEBUG` viene quindi cercata nel namespace locale della funzione `main()`
- la variabile esiste solo se viene eseguita la parte condizionale del primo `if`.
- Nota: anche nel caso in cui non si verifica l'errore, la variabile `DEBUG` che interessa (quella globale) non viene modificata!

file: `scope4.py` – Versione corretta

---

```
import sys

DEBUG = False

def main():
    global DEBUG
    if "-d" in sys.argv:
        DEBUG = True

    print "Inizio programma ..."
    if DEBUG:
        print "Esecuzione in modo debug"

if __name__ == "__main__":
    main()
```

---

L'istruzione `global DEBUG` forza python a collocare (o cercare) il nome `DEBUG` nel namespace globale, quello “più in alto” nella gerarchia.

## La classe vuota:

---

```
class Empty(object):  
    pass
```

---

## Semplice classe (file: `numero.py`)

---

```
class Numero(object):  
    "Un esempio di classe: Numero()"  
    names=("zero","uno","due","tre","quattro",  
          "cinque", "sei","sette","otto","nove")  
  
    def __init__(self,n):          # Costruttore  
        self.name=Numero.names[n]  
        self.value=n  
  
    def __str__(self):             # Metodo "overloaded"  
        return self.name  
  
    def upper(self):               # Metodo  
        return self.name.upper()
```

---

## Come si usano le classi:

---

```
>>> from numero import Numero  
>>> a=Numero(2)  
>>> b=Numero(3)  
>>> a  
<numero.Numero object at 0x7f535c1c0050>  
>>> b  
<numero.Numero object at 0x7f535c1c02d0>  
>>> print a, b  
due tre  
>>> a.value  
2  
>>> a.upper()  
'DUE'  
>>>
```

---

Cosa abbiamo nella pagina precedente:

- ➊ Definizione di classe
- ➋ Costruttore della classe
- ➌ Attributi e metodi
- ➍ Variabili di classe e variabili di istanza
- ➎ Overload
- ➏ Istanze di una classe (oggetti)
  - Accesso agli attributi di un oggetto
  - Uso dei metodi di un oggetto

## Proseguendo l'esempio precedente:

---

```
>>> a+b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Numero' and 'Numero'
>>>
```

---

## Definiamo una "classe derivata" (file: `numeropiu.py`)

---

```
from numero import Numero

class NumeroPiu(Numero):
    "Numero con operazione di addizione"
    def __add__(self, x):          # Metodo
        return NumeroPiu(self.value+x.value)

    def upper(self):              # Metodo overloaded
        return self.name.capitalize()
```

---

## Usiamo la nuova classe:

---

```
>>> from numeropiu import NumeroPiu
>>> a=NumeroPiu(2)
>>> b=NumeroPiu(3)
>>> a
<numeropiu.NumeroPiu object at 0x7f535c149b10>
>>> b
<numeropiu.NumeroPiu object at 0x7f535c1b94d0>
>>> print a, "+", b, "=", a+b
due + tre = cinque
>>> a.upper()
'Due'
>>>
```

---

Cosa abbiamo nella pagina precedente:

- 1 Messaggio di errore
- 2 Import “selettivo”
- 3 Definizione di classe derivata
- 4 Overload dell'operatore “+”
- 5 Overload del metodo upper()
- 6 Come funziona la nuova classe

- Python è particolarmente ricco di moduli e packages distribuiti insieme al linguaggio.
- La prima cosa da fare nell'affrontare un problema di programmazione: cercare un modulo che lo tratta.
- Moduli standard: (⇒)
  - sys
  - os
  - os.path
  - math, cmath
  - random
  - ... e altri 280, più o meno
- Package di particolare interesse per noi:
  - numpy
  - scipy
  - matplotlib
  - astropy

Oggetti e funzioni in stretta relazione con l'interprete Python.

Esploriamo tramite `python` in particolare:

- `sys.path` (Vedi anche: `PYTHONPATH`)
- `sys.argv`
- `sys.exit()`
- `sys.maxint`
- `sys.float_info`
- `sys.maxsize`
- `sys.platform`
- `sys.stdin`
- `sys.stdout`
- `sys.stderr`



Oggetti e funzioni relativi al Sistema Operativo.

Esploriamo tramite `python` in particolare:

- `os.sep`
- `os.linesep`
- `os.defpath`
- `os.environ`
- `os.getenv("HOME")`
- `os.tmpfile()`
- `os.access("a.py", os.R_OK)`
- `os.curdir`
- `os.chdir("newdir")`
- `os.listdir("dirname")`
- `os.mkdir("/dir1/dir2/name")`
- `os.makedirs("/dir1/dir2/name")`
- `os.rename("old", "new")`
- `os.rename("old", "new")`
- `os.walk("topdir")`

Funzioni per la manipolazione dei nomi dei file in modo portabile

Esploriamo tramite `python` in particolare:

- `os.path.abspath(path)`
- `os.path.basename(path)`
- `os.path.dirname(path)`
- `os.path.split(path)`
- `os.path.commonprefix(list)`
- `os.path.exists(path)`
- `os.path.getatime(path)`
- `os.path.getmtime(path)`
- `os.path.getctime(path)`
- `os.path.join(path1, path2, ...)`

Funzioni matematiche su numeri reali (`math`) e complessi (`cmath`) e generazione di numeri e sequenze casuali (`random`)

Esploriamo tramite `python` e funzione `help()`:

- `help(math)`, notare:
  - `fsum`
  - `expm1`
  - `log1p`
- `help(cmath)`
- `help(random)`

Gli errori nei programmi python sono trattati usando il meccanismo delle **eccezioni**.

file: error.py

---

```
def division(a, b):  
    return a/b
```

---

Adesso usiamo il modulo per causare errori:

---

```
>>> from error import division  
>>> division(2.0, 4)  
0.5  
>>> division(2.0, 0)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
    File "error.py", line 2, in division  
      return a/b  
ZeroDivisionError: float division by zero
```

---

Le eccezioni possono essere intercettate:

file: error1.py

---

```
import error  
  
def division(a,b):  
    try:  
        error.division(a, b)  
    except:  
        print "Non si puo' dividere per zero!"
```

---

ed ecco il risultato:

---

```
>>> from error1 import division  
>>> division(2.33, 0)  
Non si puo' dividere per zero!
```

---

## Notiamo ancora:

---

```
>>> division("tre", 2)
Non si puo' dividere per zero!
>>>
```

---

Le eccezioni consentono un trattamento più accurato degli errori:

file: `error2.py`

---

```
import error

def division(a,b):
    try:
        error.division(a, b)
    except ZeroDivisionError:
        print "Non si puo' dividere per zero!"
```

---

## Questa versione funziona “meglio”:

---

```
>>> from error2 import division
>>> division(2, 0)
Non si puo' dividere per zero!
>>>
>>>
>>> division(2, "zero")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "error2.py", line 5, in division
    error.division(a, b)
  File "error.py", line 2, in division
    return a/b
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

---

Python è dotato di un debugger (**pdb**) che consente di controllare l'esecuzione di un programma in vari modi:

- definizione di "breakpoint"
- esecuzione passo passo
- esame di variabili
- esecuzione di funzioni

## Esempio di uso:

---

```
$ pdb fibo.py
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(1)<module>()
-> "Modulo per il calcolo della serie di Fibonacci"
(Pdb) b 9
Breakpoint 1 at /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py:9
(Pdb) c
Serie di Fibonacci fino a 14:
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(9)fibo()
-> a, b = 0, 1
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(10)fibo()
-> while b < n:
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(11)fibo()
-> result.append(b)
(Pdb) n
> /home/lfini/Personale/CorsiSeminari/2017-06-Python/fibo.py(12)fibo()
-> a, b = b, a+b
(Pdb) p result
[1, 1]
```

---