Introduzione al Python per il calcolo scientifico

Felice Iavernaro

Dipartimento di Matematica
Università di Bari

Ottobre 2015

Calcolo Numerico
INFORMATICA L.T.

Cos'è Python? (1/3)

Da http://www.python.it/ (sito ufficiale della comunità italiana):

- Python è un linguaggio di programmazione ad alto livello, rilasciato pubblicamente per la prima volta nel 1991 dal suo creatore Guido van Rossum, programmatore olandese che ha lavorato per Google e attualmente operativo in Dropbox.
- Comodo, ma anche semplice da usare e imparare, Python è nato per essere un linguaggio immediatamente intuibile. La sua sintassi è pulita e snella così come i suoi costrutti, decisamente chiari e non ambigui.
- Python è un linguaggio pseudocompilato: un interprete si occupa di analizzare il codice sorgente (semplici file testuali con estensione .py) e, se sintatticamente corretto, di eseguirlo. In Python, non esiste una fase di compilazione separata (come avviene in C, per esempio) che generi un file eseguibile partendo dal sorgente. Ciò rende Python un linguaggio portabile. Una volta scritto un sorgente, esso può essere interpretato ed eseguito sulla gran parte delle piattaforme attualmente utilizzate, siano esse di casa Apple (Mac) che PC (Microsoft Windows, Linux).

Cos'è Python? (2/3)

- Python supporta diversi paradigmi di programmazione, come quello object-oriented e la programmazione strutturata. Rappresenta una delle tecnologie principali del core business di colossi come Google (YouTube è pesantemente basato su Python) e ILM (Industrial Light & Magic, una delle più importanti aziende nel campo degli effetti speciali digitali, oggi parte della più ampia LucasFilm).
- Python è un software free: non solo il download dell'interprete per la propria piattaforma, così come l'uso di Python nelle proprie applicazioni, è completamente gratuito; ma oltre a questo Python può essere liberamente modificato e così ridistribuito, secondo le regole di licenza open-source. Attualmente, lo sviluppo di Python (grazie e soprattutto all'enorme e dinamica comunità internazionale di sviluppatori) viene gestito dall'organizzazione no-profit *Python Software Foundation*.
- Curisità: Python deriva il suo nome dalla commedia *Monty Python's Flying Circus* dei celebri Monty Python, in onda sulla BBC nel corso degli anni 70.

Cos'è Python? (3/3)

Python è un linguaggio *General Purpose*, particolarmente appropriato per le seguenti applicazioni:

- Sviluppo web
- Accesso ai database
- Applicazioni desktop
- Giochi e grafica 3D
- Calcolo numerico e scientifico

Strumenti di lavoro (1/2)

Prerogativa di Python è un numero limitato di istruzioni e una sintassi minimale. Questo non significa che Python è limitato. Al contrario:

1. ciò garantisce semplicità di utilizzo (anche da parte di non esperti) ed alta leggibilità del codice.
2. sono disponibili tantissime librerie (packages, pacchetti) realizzate da terzi per la gestione di problematiche nelle diverse scienze applicate.

L'installazione di Python comprende sempre la libreria standard considerata elemento fondamentale. Altre librerie possono essere scaricate e installate all'occorrenza.

Strumenti di lavoro (2/2)

Ogni libreria è una cartella che racchiude uno o più moduli. Un modulo è un insieme di funzioni che concorrono alla gestione di un particolare problema. Di particolare interesse per le nostre applicazioni sono:

1. il modulo numpy: è un pacchetto fondamentale per il calcolo scientifico in Python
2. il pacchetto scipy: è una collezione di metodi numerici e tool che coprono diverse aree del calcolo scientifico quali: fitting di dati, algebra lineare, visualizzazione e manipolazione di dati, ottimizzazione, risoluzione di equazioni differenziali, metodi nell'ambito della statistica, ecc.

In questo corso utilizzeremo la distribuzione anaconda di Python che, oltre a includere diversi pacchetti, dispone di un'interfaccia grafica. La distribuzione anaconda è particolarmente utile per il calcolo scientifico e l'analisi dei dati. È scaricabile (gratuitamente) dal sito:

https://www.continuum.io

IDE

Durante questo corso utilizzeremo la IDE Spyder (inclusa in anaconda) quale interfaccia grafica all'interno della quale utilizzare Python:

IDE $\equiv$ Integrated Development Environment

All'avvio del programma compare un'interfaccia contenente diverse finestre, tra cui un editor di testo e una finestra detta *Python shell* (o console) con il prompt dei comandi:

`>>>` (console classica di Python)
`In [1]:` (Ipython: Interactive console di Python, consigliata)

a destra del quale è possibile inserire e gestire espressioni ed istruzioni.

L'utilizzo più semplice di Python è quello di una calcolatrice scientifica. Si scrive un'espressione numerica e la si valuta premendo il tasto invio. Esempio:

```
>>> 3+2*(9-4)
13
```

Tipi di dati in Python

Gli elementi principali per la costruzione delle espressioni numeriche elementari sono:[1]

- i numeri (int, long, float, complex);
- le stringhe (str);
- gli array (list, tuple, dictionary).

Ingredienti fondamentali nella composizione di un'espressione matematica sono:

- le variabili;
- gli operatori elementari;
- le funzioni matematiche elementari.

[1] qui e nel seguito, per sintetizzare, faremo riferimento alle proprietà principali degli oggetti che descriviamo. La trattazione risulta quindi incompleta ma di più semplice comprensione.

Numeri interi

Sono di due tipi

1. int (memorizzati in campi di 32 bit)
2. long (a precisione illimitata)

Gli interi di tipo int sono quelli che vanno da $-2^{31}+1$ a $2^{31}-1$, cioè quelli compresi nell'intervallo $[-2147483647, 2147483647]$.
Gli interi che fuoriescono da questo intervallo necessitano di più memoria per la loro rappresentazione. Essi saranno di tipo long e possono avere lunghezza illimitata (compatibilmente con la memoria fisica).

ESEMPI

```
>>> 2**30
1073741824
>>> type(2**30)
int
```

```
>>> x=5
>>> x/2
2.5
```

```
>>> x//2
2
>>> 5-4/2
3.0
```

Numeri floating point

Python usa la notazione decimale convenzionale per la rappresentazione dei numeri reali (non interi), con un punto per separare la parte intera da quella decimale, ad esempio: $1.23, -324.758$

Python permette anche la notazione scientifica o esponenziale (con mantissa ed esponente). Per specificare una potenza di 10 si utilizza la lettera e, ad esempio -3×10^8 lo si rappresenta digitando

```
>>> -3e8
```

mentre il numero 2.34×10^{-12} lo si rappresenta digitando

```
>>> 2.34e-12
```

ESEMPI

```
>>> 2.0**30
1073741824.0
>>> type(2.0**30)
float
```

```
>>> 3/2
1.5
>>> 3.0//2.0
1.0
```

```
>>> 5.0-2//3
5.0
>>> 5-2/3
4.333333333333333
```

Aritmetica di macchina

Python rappresenta i numeri reali in base binaria utilizzando la *doppia precisione*.

Ciascun numero è memorizzato in un campo da 64 bit[2] di cui:

- 1 bit identifica il segno (+ o -);
- 52 bit sono dedicati alla memorizzazione della mantissa Ciò corrisponde, in base 10, a circa 16 cifre significative.
- 11 bit sono dedicati alla memorizzazione dell'esponente.

Ad esempio i numeri 1.23456789012345678**90** e 1.23456789012345**66** verranno rappresentati dal medesimo numero di macchina.

```
>>> 1.2345678901234567890 == 1.2345678901234566
True
```

Inoltre non potranno essere rappresentati numeri né troppo grandi né troppo piccoli (in valore assoluto). Esempio: digitare

$$2.0 * *1023 \qquad 2.0 * *1024 \qquad 2.0 * * - 1022 \qquad 2.0 * * - 1075$$

[2] bit $\equiv$ binary digit, unità di informazione elementare

Le variabili: assegnazione

L'assegnazione è un'operazione utilizzata in informatica per inserire un valore in una variabile. Ad esempio con l'istruzione

```
>>> a=12.345
```

si assegna il valore scalare 12.345 alla variabile *a* che, da questo punto in poi, sarà disponibile nel command window per un successivo utilizzo, come ad esempio:

```
>>> b=a+1
```

Python è *case sensitive*, cioè fa differenza tra variabili scritte in maiuscolo ed in minuscolo:

```
>>> A=3.1


>>>a+A
15.445
```

Le variabili: proprietà (1/2)

- Le variabili sono sovrascrivibili, cioè, se digitiamo ora la stringa

  ```
  >>> A = 0.5
  ```

  il precedente valore 3.1 viene definitivamente perso.
- È possibile effettuare più assegnazioni sulla stessa riga, separate dal simbolo ;

  ```
  >>>w1=2.3; w2=-3.5; t=0.12;
  ```

  Alternativamente si può anche scrivere (assegnazione multipla)

  ```
  >>>w1, w2, t  = 2.3, -3.5, 0.12
  ```
- una variabile può essere una qualsiasi combinazione di caratteri alfanumerici. Il nome di una variabile non può iniziare con un numero; inoltre vi sono alcuni caratteri non ammessi, poiché hanno diverso significato (* + / - . , = ecc.).

Più avanti esamineremo più in dettaglio il significato delle variabili in Python. Al momento pensiamole come contenitori di valori.

Le variabili: proprietà (2/2)

Nota tecnica: Le variabili definite all'interno della shell, risiedono in una zona di memoria chiamata *main namespace*. Il modulo *__main__* è l'ambiente all'interno del quale viene eseguito un interprete di Python, come ad es. IPython.

- Per visualizzare il contenuto di una variabile, si digita la variabile e si preme invio o, alternativamente, si digita *print(nomevariabile)*
- Per visualizzare tutte le variabili definite (all'interno dello spazio dei nomi principale) dall'inizio della sessione di lavoro:

```
>>> import __main__
>>> dir(__main__)
```

- Per cancellare una variabile dal main namespace si utilizza il comando *del*. Esempi:

```
>>> del A
>>> del w1, w2
>>> dir(__main__)
```

Operazioni elementari

$+$	addizione;
$-$	sottrazione;
$*$	moltiplicazione;
$/$	divisione;
$**$	elevamento a potenza;

Esempi:

```
>>> 2**3
8
>>> 2**-3
0.125
>>> 2**0.5
1.4142135623730951
```

```
>>> 2**3/4
2.0
>>> 2**(3/4)
1.681792830507429
```

Funzioni elementari (1/4)

Proviamo a calcolare la radice quadrata mediante la funzione sqrt:

```
>>> sqrt(2)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
```

Le funzioni matematiche elementari sono disponibili dopo aver caricato il modulo numpy:

```
>>> import numpy
```

Questo comando carica in memoria tutte le funzioni presenti nel modulo numpy e le rende accessibili all'utente mediante la sintassi:

```
>>> numpy.nome_funzione
```

Ad esempio:

```
>>> numpy.sqrt(2)
1.4142135623730951
```

Funzioni elementari (2/4)

Funzioni trigonometriche:

sin	seno
cos	coseno
tan	tangente
arcsin	arcoseno
arccos	arcocoseno
arctan	arcotangente

sinh	seno iperbolico
cosh	coseno iperbolico
tanh	tangente iperbolica
arcsinh	arcoseno iperbolico
arccosh	arcocoseno iperbolico
arctanh	arcotangente iperbolica

Funzioni esponenziali e logaritmiche:

exp	esponenziale in base e
log2	logaritmo in base 2

log	logaritmo naturale
log10	logaritmo in base 10

Altre funzioni:

abs	valore assoluto o modulo
fix	parte intera

sqrt	radice quadrata
round	arrotondamento

Funzioni elementari (3/4)

Alternativamente, scrivendo

```
>>> import numpy as np
```

le funzioni del modulo numpy sono richiamabili mediante la sintassi

```
>>> np.nome_funzione
```

In questo caso utilizziamo l'alias `np` (generalmente adottato in letteratura) per contrarre la sintassi.

Esempi:

```
>>> np.sqrt(2)
1.4142135623730951
>>> np.log(2)
0.6931471805599453
>>> np.round(np.log(2))
1.0
```

```
>>> 4*np.arctan(1)
3.1415926535897931
>>> np.sinh(2)
3.6268604078470186
>>> (np.exp(2)-np.exp(-2))/2
3.6268604078470191
```

Funzioni elementari (4/4)

L'uso di un alias è importante poiché evita una possibile sovrapposizione tra funzioni aventi il medesimo nome ma appartenenti a moduli differenti. Se invece vogliamo usare le funzioni di numpy senza l'uso di un alias è possibile scrivere

```
>>> from  numpy import *
```

Esempi:

```
>>> sqrt(2)
1.4142135623730951
>>> log(2)
0.6931471805599453
>>> round(log(2))
1.0
```

Variabili predefinite

Il modulo numpy mette anche a disposizione alcune costanti matematiche tra cui:

- np.pi: il numero pi greco 3.1415927....;
- np.e: il numero di Nepero 2.7182818;
- np.inf, infinito;
- np.nan, not-a-number;

In Python le variabili logiche sono

- False, oppure 0: variabile logica *false*;
- True, oppure 1: variabile logica *true*.

Operatori logici e di relazione

Operatori logici: and, or, not.
Operatori di relazione:

$>$	maggiore	$>=$	maggiore o uguale
$<$	minore	$<=$	minore o uguale
$==$	uguale	$!=$ oppure $<>$	diverso

Esempi

```
>>> 2>=1
True
>>> not 2>1
False
>>> 2>1 and 0>1
False
>>> 2>1 or 0>1
True
```

```
>>> a=np.log(3)>1
>>> print(a)
True
>>> b=not a
>>> print(b)
False
```

```
>>> (np.sqrt(2))**2==2
False
>>> (np.sqrt(2))**2
2.0000000000000004
>>> np.cos(np.pi/2)==0
False
>>> np.cos(np.pi/2)
6.123233995736766e-17
```

Stringhe (1/2)

Una stringa è un testo racchiuso tra apici o doppi apici.

Esempi:

```
>>> nome='Elvira'
>>> type(nome)
<type 'str'>
>>> nome
'Elvira'
>>> cognome='D'Angelo'
SyntaxError: invalid syntax
>>> cognome="D'Angelo"
>>> cognome
"D'Angelo"
>>> nome_completo=nome+" "+cognome
>>> nome_completo
"Elvira D'Angelo"
>>> print(nome_completo)
Elvira D'Angelo
>>> len(nome)   <-- numero di
6                caratteri
```

```
>>> nome[0]
'E'
>>> nome[1]
'l'
>>> nome[5]
'a'
>>> nome[-1]
'a'
>>> nome[0:3]
'Elv'
>>> cognome[2:]
'Angelo'
>>> "r" in nome
True
>>> "e" in nome
False
```

Stringhe (2/2)

Se il testo occupa più righe, conviene utilizzare come delimitatore i doppi apici ripetuti tre volte. Esempio:

```
>>> lettera="""Caro Fabio,
ecco gli estremi del mio volo:

VOLO n. AZ1609
partenza da Roma, ore 09:20
arrivo a Bari, ore 10:30"""
>>> lettera
'Caro Fabio,\n ecco gli estremi del mio volo:\n VOLO n. AZ1609
\n partenza da Roma, ore 09:20\n arrivo a Bari, ore 10:30'

>>> print(lettera)
Caro Fabio,
ecco gli estremi del mio volo:

VOLO n. AZ1609
partenza da Roma, ore 09:20
arrivo a Bari, ore 10:30
```

Più avanti tratteremo più esaustivamente delle operazioni sulle stringhe.

ELEMENTI DI PROGRAMMAZIONE STRUTTURATA

Python mette a disposizione un linguaggio di programmazione di semplice utilizzo.

Esso dispone delle tre strutture fondamentali

- sequenza
- selezione
- iterazione

che consentono di tradurre in programma un qualsiasi algoritmo.

Sequenza

La *sequenza* non è altro che un blocco ordinato di istruzioni che verranno eseguite una dopo l'altra in successione.

Esempio

```
>>> a,b,c=1,-5,6
>>> delta=b**2-4*a*c
>>> x1=(-b-np.sqrt(delta))/(2*a)
>>> x2=(-b+np.sqrt(delta))/(2*a)
>>> print("le radici sono:  x1=", x1,"  x2=",x2)
```

Selezione (1/3): definizione di base

La selezione, nella sua forma più semplice, consente di eseguire un blocco di istruzioni a seconda che sia verificata o meno una data condizione.

```
if condizione :
    istruzioni
```

Da ricordare:

1. condizione rappresenta l'*argomento* del costrutto if. È una qualsiasi proposizione la cui veridicità viene esaminata;
2. istruzioni rappresenta il blocco delle istruzioni che formano il corpo del costrutto if. Tali istruzioni dovranno essere *indentate*, inserendo un certo numero di spazi. Comunemente si inserisce un carattere di tabulazione (tasto TAB);
3. se l'argomento dell'if risulta vero, vengono eseguite le istruzioni corrispondenti al corpo dell'if. Viceversa, se l'argomento dell'if è falso, viene effettuato un salto alla riga successiva alla fine del costrutto if.

Selezione (2/3): Esempio

Calcolo del valore assoluto di un numero x, supponendo di aver assegnato a x un valore.

```
>>> if x<0:
        x=-x
```

È possibile inserire queste istruzioni direttamente da riga di comando nella shell. Alternativamente possiamo inserirle in un file di testo con estensione .py (utilizzando, ad esempio, l'editor di Spyder) ed eseguire il file mediante il comando run. Opteremo per questa seconda scelta, che in seguito perfezioneremo. Creiamo il file valore_assoluto.py, contenente il codice:

```
x=input("inserisci un numero: x=")
x=float(x)
if x<0:
    x=-x
print("il valore assoluto di x e':",x)
```

N.B. L'output della funzione `input` è una stringa, che viene poi convertita in numero mediante la funzione `float`

Selezione (3/3): definizione generale

L'uso completo del costrutto **if** è:

if prima condizione **:**
    istruzioni
elif seconda condizione **:**
    istruzioni
    ⋮
else :
    istruzioni

Esempio: calcolo del segno di un numero reale x. Scriviamo il seguente codice in uno script file che denomineremo segno.py

```
x=float(input("inserisci un numero: x="))
if x>0:
    s=1
elif x==0:
    s=0
else:
    s=-1
print("il segno di x e':",s)
```

Un primo esempio di funzione

Normalmente è meglio evitare l'uso dell'istruzione *input* per l'ingresso da tastiera dei dati da elaborare, e del comando print per la stampa a video dei dati di output. Risulta invece più vantaggioso riguardare il codice all'interno di una funzione. Ecco la sintassi per il calcolo del segno di un numero:

```
def segno(x):
    if x>0:
        s=1
    elif x==0:
        s=0
    else:
        s=-1
    return s
```

- Notare l'indentazione del corpo della funzione.
- Salvare il file con il nome *segno.py*
- Confrontare con la function predefinita *np.sign*

Esercizio (sull'uso dell'if e sulla definizione di funzione)

Scrivere la function Python *eq2* che abbia in input tre coefficienti reali a, b, c, e in output le radici dell'equazione di secondo grado $ax^2 + bx + c = 0$, solo nel caso che siano reali (controllo sulla positività del discriminante).

1. Salvare il file con il nome programmi.py
2. dal prompt di Python scrivere

```
>>> import programmi      <-- programmi.py e' formalmente un modulo
>>> type(programmi)           che potra' contenere diverse funzioni
>>> dir(programmi)        <-- restituisce i nomi degli oggetti
>>> programmi.eq2(1,-5,6)     definiti nel modulo
>>> del programmi
>>> import programmi as pr
>>> pr.eq2(1,-5,6)
```

3. Sotto la prima riga aggiungere il testo

```
"""
Calcola le radici (reali) di un'equazione di secondo grado
"""
```

4. dal prompt di Python scrivere

```
>>> import programmi as pr     <-- ricarica il modulo aggiornato
>>> help(pr.eq2)                   nel namespace corrente
>>> pr.eq2.__doc__
```

Ripetizione

La ripetizione o iterazione permette l'esecuzione di un blocco di istruzioni un numero di volte prestabilito.La forma più semplice è:

```
for ind in range(int1,int2,step) :
        istruzioni
```

che esegue in maniera ripetuta il gruppo di istruzioni interne al ciclo per un numero di volte prestabilito, uguale al numero di volte in cui varia l'indice *ind* (detto contatore) fra il valore *int*1 e il valore $int2 - 1$ con un incremento pari a *step*.

- *int*1, *int*2 e *step* devono essere numeri interi
- Se l'incremento *step* non è specificato esplicitamente esso viene scelto per default uguale a $+1$.
- la funzione predefinita range(int1,int2,step) genera una lista di interi:

```
>>> range(-3,6)      --> [-3, -2, -1, 0, 1, 2, 3, 4, 5]
>>> range(-3,6,2)    --> [-3, -1, 1, 3, 5]
```

ESEMPI

- Calcolo della somma dei primi dieci numeri interi.

```
>>> somma=0
>>> for i in range(1,11):
        somma=somma+i

>>> print(somma)
55
```

- Stampa i primi 20 numeri della successione di Fibonacci

```
>>> a,b=1,1
>>> for k in range(2,15+2):
        print(a)
        c=a+b
        a=b
        b=c

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

ESERCIZIO

- Si scriva una funzione di nome *inverti* che abbia in input una stringa *s* e in output una stringa *r* che contenga gli stessi caratteri di *s* ma in ordine inverso

  Esempio:

$$s = \text{“enoteca”} \qquad \Longrightarrow \qquad r = \text{“acetone”}$$

Ripetizione condizionata

Un altro costrutto che implementa l'iterazione è il seguente:

```
while condizione :
        istruzioni
```

Questa struttura di controllo esegue in maniera ripetuta il gruppo di istruzioni interne al ciclo fino a quando la condizione posta come argomento del while resta verificata. Ovviamente, qualora la condizione risulti non verificata in partenza, Python salta tutto il blocco delle istruzioni, deviando il flusso del programma alla linea successiva al blocco.

Esempio: quanti numeri interi successivi occorre sommare per ottenere un numero maggiore di 354?

```
>>> somma=0; i=0
>>> while somma<=354:
        i=i+1
        somma=somma+i
>>> print(i)
27
```

Esercizio. Calcolo del M.C.D. tra due numeri. (1/2)

Siano m ed n due numeri positivi ed $\mathrm{MCD}(m,n)$ il loro Massimo Comune Divisore. Vale la seguente proprietà:

$$\mathrm{MCD}(m,n) = \begin{cases} \mathrm{MCD}(m-n,n), & \text{se } m > n, \\ \mathrm{MCD}(m,n-m), & \text{se } n > m, \\ m, & \text{se } n = m. \end{cases} \qquad (*)$$

Cioè: tutti e soli i divisori comuni di m ed n sono i divisori comuni di $m-n$ ed n, e quindi anche il massimo comune divisore di queste due coppie di numeri dovrà coincidere.

ESEMPIO:

$\mathrm{MCD}(30,18) = \mathrm{MCD}(12,18) = \mathrm{MCD}(12,6) = \mathrm{MCD}(6,6) = 6.$

I vantaggi della (*) sono:

- la (*) non fa uso della scomposizione in fattori primi dei due interi m ed n;
- la (*) induce un semplice algoritmo implementabile in Python.

Esercizio. Calcolo del M.C.D. tra due numeri. (2/2)

All'interno del modulo *programmi*, scrivere una funzione di nome MCD che abbia in input due interi positivi m ed n e restituisca in output il loro M.C.D.

```
def MCD(m,n):
    """
    Calcola il Massimo Comune Divisore tra due numeri
    """
    while m!=n:
        if m>n:
            m=m-n
        else:
            n=n-m
    return m
```

Funzioni: motivazione

Spesso una data sequenza di istruzioni che risolvono un dato problema deve essere ripetuta più volte, ad esempio quando:

(a) vogliamo risolvere il problema in corrispondenza di diversi dati.

(b) la risoluzione del problema in questione viene richiesta più volte durante lo svolgimento di un problema più ampio.

Lavorando dal prompt saremmo costretti, ogni volta, a rieseguire ciascuna delle istruzioni (si pensi che gli algoritmi più sofisticati si traducono in centinaia di righe di codice). Inoltre se chiudiamo e riapriamo la sessione di lavoro, tutte le variabili che avevamo definito vanno perse.

Converrà, in tal caso, riscrivere le stesse istruzioni all'interno di un file che, una volta salvato sul disco rigido, potrà essere richiamato ed eseguito all'occorrenza.

Funzioni Python

Una funzione Python ammette dei dati (o parametri) di input (ingresso) e restituisce dei dati di output (uscita).

Informalmente possiamo pensare che i parametri di input rappresentino i dati del problema che vogliamo risolvere, mentre i parametri di output rappresentino le soluzioni del nostro problema.

Ad esempio, nel caso dell'equazione di secondo grado, i parametri di input saranno i coefficienti dell'equazione, mentre i parametri di output saranno le radici della stessa.

Funzioni: sintassi

Una funzione Python ha la seguente struttura

```
def nome_funzione(a,b,...) :
        istruzioni
        :
        return c,d,...
```

Riconosciamo:

- la parola chiave **def** che identifica l'inizio della funzione;
- la parola chiave **return** che identifica la fine della funzione;
- il nome della funzione;
- le variabili di input a,b,...;
- le variabili di output c,d,...;

IMPORTANTE: Tutte le variabili usate all'interno di una function sono variabili locali, cioè esistono solo durante l'esecuzione della funzione e non modificano il main namespace. Ad esempio, la variabile *a* usata all'interno di una function sarà diversa dalla variabile *a* usata in un'altra function o nella shell.

Funzioni: come scriverle.

Un qualsiasi editor di testo può essere utilizzato per scrivere il codice che definisce una funzione. Spyder possiede un proprio editor di testo che ci aiuta durante la scrittura di un programma, mediante l'uso di colori, l'indentazione automatica, il check della correttezza semantica del codice, ecc.

Una volta scritta la funzione, la si salva in un file, mediante la barra degli strumenti dell'editor.

- Il nome del file non deve necessariamente coincidere con quello della funzione.
- Ogni file dovrà avere l'estensione .py per essere riconosciuto da Python.

OSSERVAZIONE: un file *.py* è formalmente un modulo[3]. Pertanto il file può ospitare più funzioni al suo interno.

[3] da discutere in seguito

Funzioni: come caricarle (1/2)

Per poter essere eseguita, la funzione dovrà essere caricata in memoria (più precisamente nel namespace del modulo main).

Poiché la nostra funzione risiede in un modulo, per poterla caricare occorre far riferimento al nome del modulo stesso.

Per fissare le idee, supponiamo che il file che identifica il modulo si chiami programmi, e che questo contenga le funzioni MCD e eq2.

Analizziamo le diverse possibilità.

Funzioni: come caricarle (2/2)

1. >>> import programmi
   In tal caso, le funzioni MCD e eq2 non sono accessibili direttamente, ma attraverso il nome del modulo, secondo la sintassi[4]

   >>> x=programmi.MCD(18,24) >>> [z1,z2]=programmi.eq2(1,-5,6)

2. >>> import programmi as pr
   In tal caso, mediante l'alias pr la sintassi di chiamata a ciascuna funzione risulta semplicficata

   >>> x=pr.MCD(18,24) >>> [z1,z2]=pr.eq2(1,-5,6)

3. >>> from programmi import MCD (oppure: >>> from programmi import MCD, eq2)
   In tal caso le funzioni sono accessibili mediante i loro nomi:

   >>> x=MCD(18,24) >>> [z1,z2]=segno(1,-5,6)

4. >>> from programmi import *
   Carica tutte le funzioni presenti nel modulo.

   >>> x=MCD(18,24) >>> [z1,z2]=eq2(1,-5,6)

[4] per maggiore chiarezza, riportiamo esplicitamente possibili valori di input

Liste (definizione)

Un lista è un insieme ordinato di oggetti non necessariamente omogenei. Per definire una lista, si elencano i suoi oggetti separandoli da virgole e racchiudendoli tra parentesi quadre.

Vediamo qualche esempio:

```
>>>x=[2, 1, 0, -3]                        <-- lista di numeri
>>>y=["a","b","c","ciao","python"]        <-- lista di stringhe
>>>z=["a",-7,"b",np.log,[1,-2,2**(-3)]] <-- lista mista
```

Come per le stringhe, per accedere agli elementi di una lista si indicano gli indici tra parentesi quadre:

```
>>> z[0]      <-- l'indice del primo
'a'                elemento della lista \'e 0
>>> type(z[0])
<type 'str'>
>>> z[3]
<ufunc 'log'>
>>> z[3](2)
0.6931471805599453
```

```
>>> z[-1]
[1, -2, 0.125]
>>> type(z[-1])
<type 'list'>
>>> z[-1][1]
-2
>>> z[-1][-1]
0.125
```

Liste: modificarne gli elementi

Le liste, al contrario delle stringhe, sono oggetti dinamici, cioè possiamo modificarle, introducendo, eliminando o ridefinendo degli elementi. Più propriamente, si dice che le liste sono oggetti mutabili, al contrario delle stringhe che sono immutabili.

Esempi:

```
>>> w=["abc", 23, 4.56]
>>> w
['abc', 23, 4.56]
>>> w[2]=7.8
>>> w
['abc', 23, 7.8]
>>> w[1]=w[1]-4*w[2]
>>> w
['abc', -8.2, 7.8]
>>> w=w+["ciao",2**3]
>>> w
['abc', -8.2, 7.8, 'ciao', 8]
>>> w[1:3]=[w[4]/2, w[1]+w[2]]
>>> w
['abc', 4, -0.4, 'ciao', 8]
```

```
>>> w[3]
'ciao'
>>> w[3:4]
['ciao']
>>> w[3:4]=[]          <-- elimina
>>> w                      un elemento
['abc', 4, -0.4, 8]
>>> w[1:1]=["salve"] <-- aggiunge
>>> w                      un elemento
['abc', 'salve', 4, -0.4, 8]
>>> w[2:]=[]
>>> w
['abc', 'salve']
>>> w[1][4]="o"
TypeError: 'str' object does not
           support item assignment
```

Liste: operazioni e funzioni elementari

- Lunghezza: Il numero di elementi di una lista si chiama *lunghezza* della lista: in Python si ottiene mediante la funzione predefinita *len*:

```
>>> L1=[1,2,3,"a","b","c"]
>>> len(L1)
6
```

- Concatenazione:

```
>>> L2=["d","e",4,5]
>>> L=L1+L2
>>> L
[1, 2, 3, 'a', 'b', 'c', 'd', 'e', 4, 5]
```

- Moltiplicazione:

```
>>> L3=["bla","bla"]
>>> 2*L3
['bla', 'bla', 'bla', 'bla']
>>> 3*L3
['bla', 'bla', 'bla', 'bla', 'bla', 'bla']
```

Liste: metodi (1/3)

Le liste, come ogni oggetto Python, dispongono di alcune funzioni dedicate, dette metodi. Per visualizzare i metodi disponibili per un dato oggetto si utilizza la funzione dir. Definire una lista L e provare: >>> dir(L). Elenchiamo alcuni metodi associati alle liste, mediante degli esempi. Sia $L = [1, 2, 3, 4]$.

- Aggiungere un elemento in coda a una lista:

```
>>> L.append(5)
>>> L
[1, 2, 3, 4, 5]
```

- Estendere una lista:

```
>>> L.extend([6,7,8])
>>> L
[1, 2, 3, 4, 5, 6, 7, 8]
```

- Inserire un elemento in una lista:

```
>>> L.insert(3,"abc")            <-- il primo argomento
>>> L                                e' l'indice dell'elemento
[1, 2, 3, 'abc', 4, 5, 6, 7, 8]     prima del quale inserire
>>> L.insert(0,"def")
>>> L
['def', 1, 2, 3, 'abc', 4, 5, 6, 7, 8]
```

Liste: metodi (2/3)

- Rimuovere un dato elemento:

```
>>> L=["a",1,"b",2,"c",3,"b",4]
>>> L
['a', 1, 'b', 2, 'c', 3, 'b', 4]
>>> L.remove("b")
>>> L
['a', 1, 2, 'c', 3, 'b', 4]
```

- Rimuove un elemento di dato indice:

```
>>> L.pop(3)
'c'
>>> L
['a', 1, 2, 3, 'b', 4]
```

- Individua l'indice di un dato elemento:

```
>>> L.index("b")
4
```

Liste: metodi (3/3)

- Conta il numero di ricorrenze di un dato elemento:

```
>>> L=["C","C","G","G","A","A","G","A","G"]
>>> L.count("G")
4
```

- Inverte l'ordine degli elementi:

```
>>> L.reverse()
>>> L
['G', 'A', 'G', 'A', 'A', 'G', 'G', 'C', 'C']
```

- ordina gli elementi:

```
>>> L.sort()
>>> L
['A', 'A', 'A', 'C', 'C', 'G', 'G', 'G', 'G']
```

NOTA TECNICA: Un metodo modifica l'oggetto "in place", cioè la variabile a cui viene applicato continua a puntare alla stessa locazione di memoria.

```
>>> L=[1,2,3]
>>> L=L+[4,5,6]   <-- crea una
>>> L                nuova lista
[1, 2, 3, 4, 5, 6]
```

```
>>> L=[1,2,3]
>>> L.extend([4,5,6]) <-- modifica la
>>> L                      lista L
[1, 2, 3, 4, 5, 6]
```

Esercizi (sull'uso delle liste numeriche e del costrutto for)

1. Scrivere una funzione di nome somma che abbia in input una lista numerica e in output la somma degli elementi della lista. Applicare la funzione al vettore $x = [1.5,\ -0.2,\ -3.1,\ 2.6]$. Confrontare con la funzione predefinita sum del modulo numpy.
2. Scrivere una funzione di nome media che abbia in input una lista numerica e in output la media degli elementi della lista. Tale funzione dovrà richiamare la funzione somma. Confrontare con la funzione predefinita mean del modulo numpy.
3. Scrivere una funzione di nome varianza che abbia in input una lista numerica e in output la varianza degli elementi della lista. Tale funzione dovrà richiamare la funzione media. Confrontare con la funzione predefinita var del modulo numpy.

Tuple (definizione)

Un tupla è un insieme ordinato immutabile di oggetti non necessariamente omogenei.
Dunque una tupla ha la stessa struttura di una lista ma, al contrario di quest'ultima, non possiamo modificarne gli elementi.
Per definire una tupla, si elencano i suoi oggetti separandoli da virgole e racchiudendoli (opzionalmente) tra parentesi tonde.

```
>>>x=(2, 1, 0, -3)                        <-- tupla di numeri
>>>y=("a","b","c","ciao","python")        <-- tupla di stringhe
>>>z=("a",-7,"b",np.log,[1,-2,2**(-3)]) <-- tupla mista
```

L'accesso agli elementi di una tupla segue la medesima sintassi vista per le stringhe e per le liste:

```
>>> z[0]
'a'
>>> z[-1]
[1, -2, 0.125]
>>> z[-1][1]
-2
>>> z[:3]
('a', -7, 'b')
```

```
>>> z[0]="c"

Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    z[0]="c"
TypeError: 'tuple' object does not support
           item assignment
>>> z[-1][1]=3
>>> z
('a', -7, 'b', <ufunc 'log'>, [1, 3, 0.125])
```

Stringhe, Liste e Tuple: operatori in, + e *

Per tutte e tre i tipi, gli operatori

- in : appartenenza
- + : concatenazione
- * : moltiplicazione

hanno il medesimo significato. Esempi:

```
>>> x=("a","b",1,2,3)
>>> y="Elvira"
>>> z=[4,5,6,"c","d"]
>>> "a" in x
True
>>> "a" in y
True
>>> "a" in z
False
>>> x+("c","d",-3)
('a', 'b', 1, 2, 3, 'c', 'd', -3)
```

```
>>> w=z+["e",(1,2)]
>>> w
[4, 5, 6, 'c', 'd', 'e', (1, 2)]
>>> 2*x
('a', 'b', 1, 2, 3, 'a', 'b', 1, 2, 3)
>>> 2*z
[4, 5, 6, 'c', 'd', 4, 5, 6, 'c', 'd']
>>> 3*y
'ElviraElviraElvira'
```

Liste e Tuple: confronti e conversioni

- La gestione delle liste è più lenta ma più potente rispetto a quella delle tuple.
- Le liste hanno il grande vantaggio di poter essere modificate sul posto. Inoltre per esse sono definite numerose operazioni utili per la gestione dei dati.
- Le tuple sono immutabili (dunque non modificabili) e per esse sono disponibili meno metodi.

Le funzioni predefinite list e tuple consentono la conversione da un tipo all'altro.
Esempi:

```
>>> x=("a","b",1,2,3)
>>> type(x)
tuple
>>> y=list(x)
>>> y
['a', 'b', 1, 2, 3]
>>> type(y)
list
```

Approfondimento: variabili, oggetti e referenze (1/5)

Come abbiamo visto, Python adotta la tipizzazione dinamica: i tipi vengono determinati automaticamente durante l'esecuzione delle istruzioni. Ad esempio:

```
>>> type(18)
int
>>> type(np.sqrt(3))
numpy.float64
>>> type([1,2,3])
list
>>> type("ciao")
str
```

```
>>> a=18
>>> type(a)
int
>>> a=np.sqrt(3)
>>> type(a)
numpy.float64
>>> a=[1,2,3]
>>> type(a)
list
```

- Ogni oggetto possiede un suo tipo, indipendentemente dal fatto che vi sia un'assegnazione.
- Se assegnamo un oggetto a una variabile, essa assume il tipo dell'oggetto a cui si riferisce.

Dunque, la nozione di tipo risiede nell'oggetto e non nella variabile che si riferisce all'oggetto.

Approfondimento: variabili, oggetti e referenze (2/5)

Vediamo più in dettaglio come interpretare correttamente un'assegnazione, ad esempio

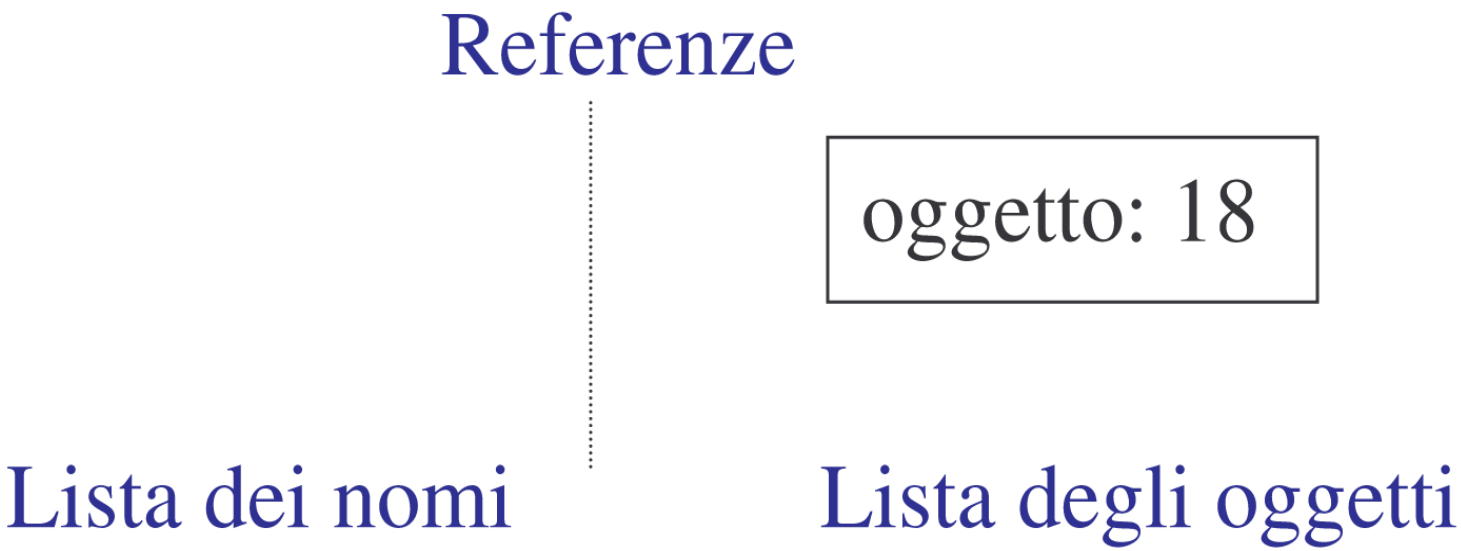
$$>>> a = 18$$

Approfondimento: variabili, oggetti e referenze (2/5)

Vediamo più in dettaglio come interpretare correttamente un'assegnazione, ad esempio

$$>>> a = 18$$

1. Viene creato un oggetto (che nel nostro caso coincide con l'intero 18) e conservato in memoria;

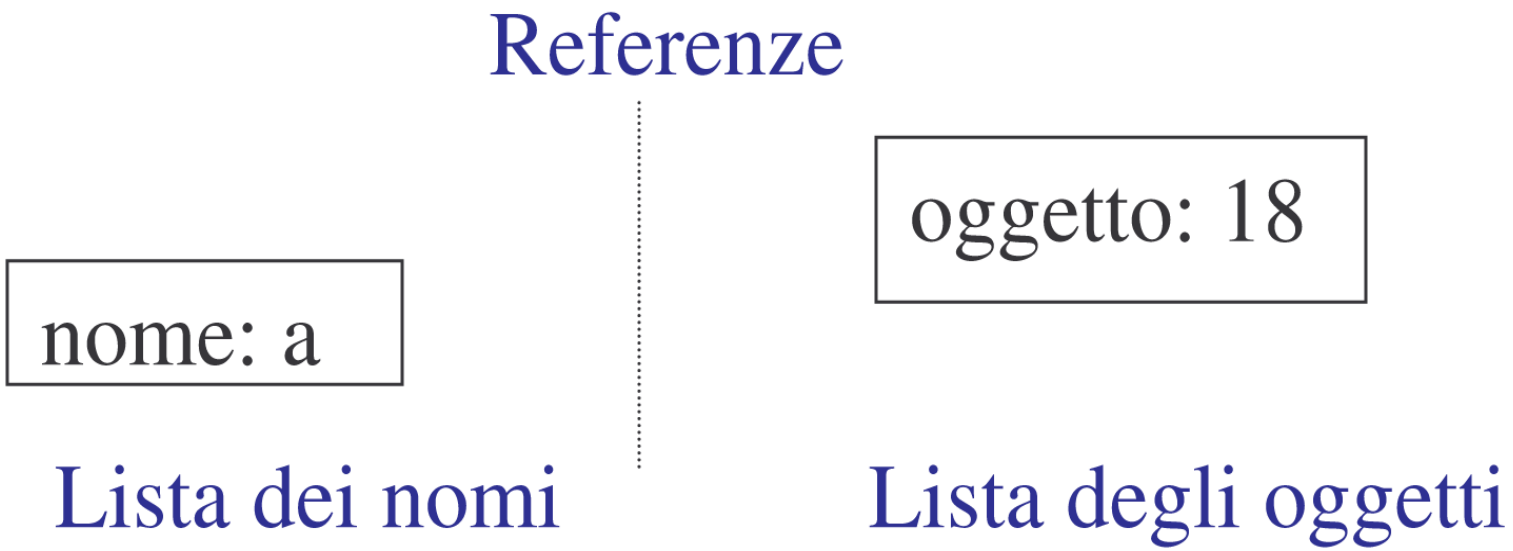


Approfondimento: variabili, oggetti e referenze (2/5)

Vediamo più in dettaglio come interpretare correttamente un'assegnazione, ad esempio

$$>>> a = 18$$

1. Viene creato un oggetto (che nel nostro caso coincide con l'intero 18) e conservato in memoria;
2. viene creato il nome *a* (nel caso in cui non esista già);

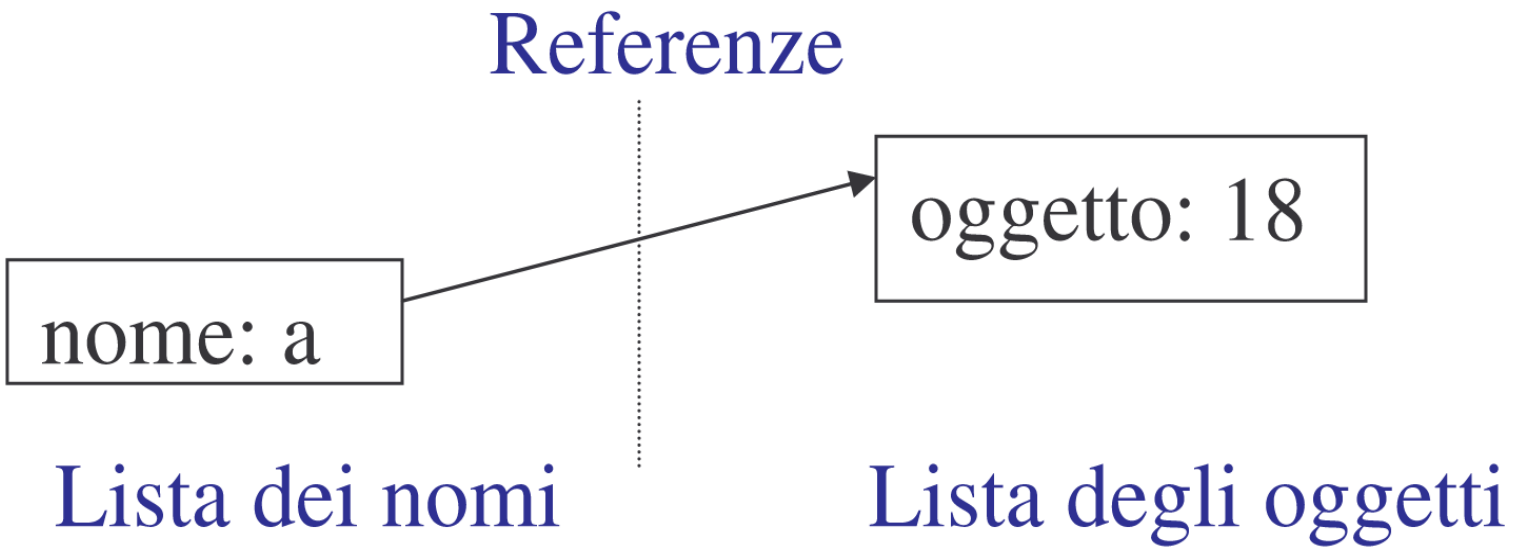


Approfondimento: variabili, oggetti e referenze (2/5)

Vediamo più in dettaglio come interpretare correttamente un'assegnazione, ad esempio

$$>>> a = 18$$

1. Viene creato un oggetto (che nel nostro caso coincide con l'intero 18) e conservato in memoria;
2. viene creato il nome *a* (nel caso in cui non esista già);
3. viene creato un collegamento (referenza) che partendo da *a*, punta alla locazione di memoria che contiene l'oggetto 18.



Approfondimento: variabili, oggetti e referenze (3/5)

Interpretiamo ora le seguenti assegnazioni aggiuntive

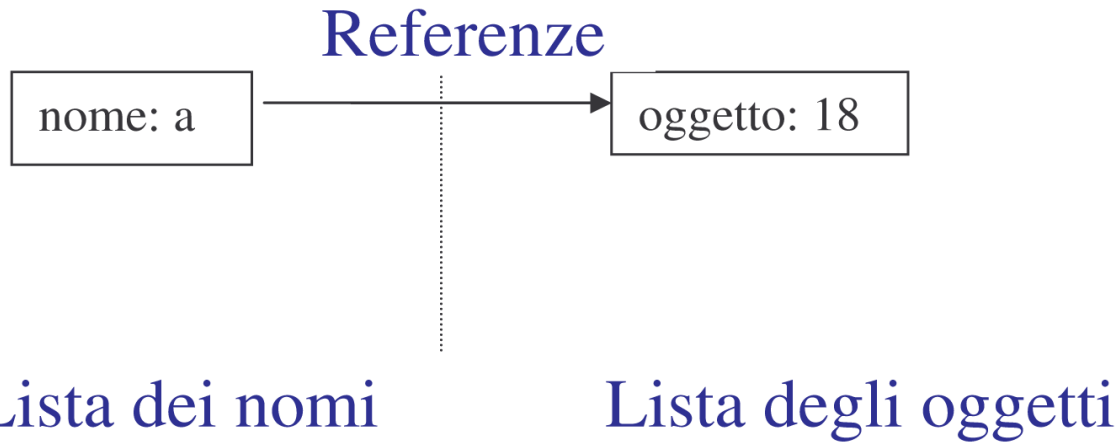
```
>>> b = a
>>> a = 13
```

Approfondimento: variabili, oggetti e referenze (3/5)

Interpretiamo ora le seguenti assegnazioni aggiuntive

$$>>> b = a$$
$$>>> a = 13$$

1. Viene ricercato l'oggetto associato al nome *a* (cioè l'intero 18);

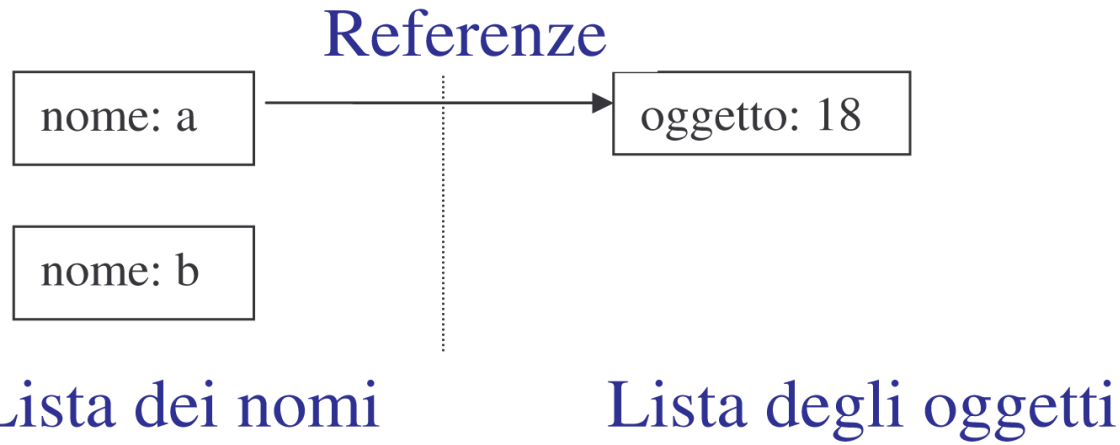


Approfondimento: variabili, oggetti e referenze (3/5)

Interpretiamo ora le seguenti assegnazioni aggiuntive

```
>>> b = a
>>> a = 13
```

1. Viene ricercato l'oggetto associato al nome *a* (cioè l'intero 18);
2. viene creato il nome *b* (nel caso in cui non esista già);

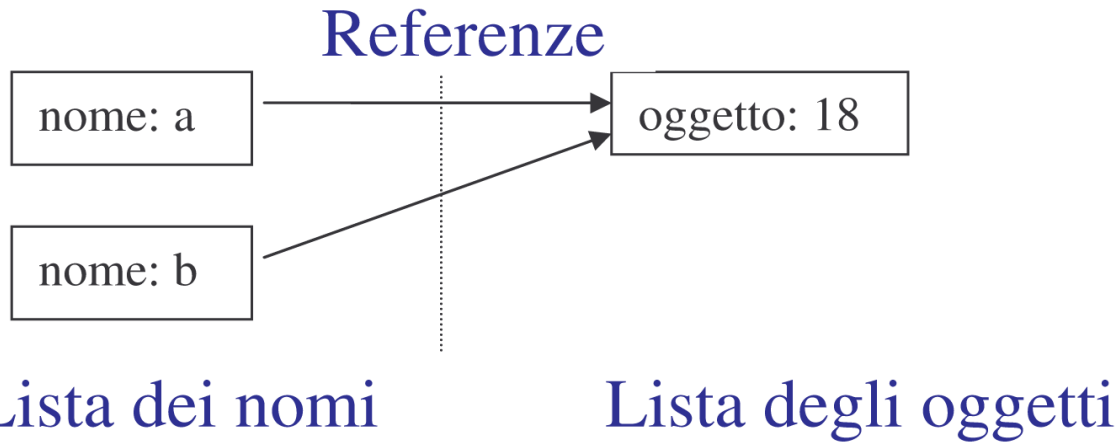


Approfondimento: variabili, oggetti e referenze (3/5)

Interpretiamo ora le seguenti assegnazioni aggiuntive

$$>>> b = a$$
$$>>> a = 13$$

1. Viene ricercato l'oggetto associato al nome *a* (cioè l'intero 18);
2. viene creato il nome *b* (nel caso in cui non esista già);
3. viene creato un collegamento (referenza) che partendo da *b*, punta alla locazione di memoria che contiene l'oggetto 18. A questo livello, i due nomi *a* e *b* puntano alla stessa area di memoria!

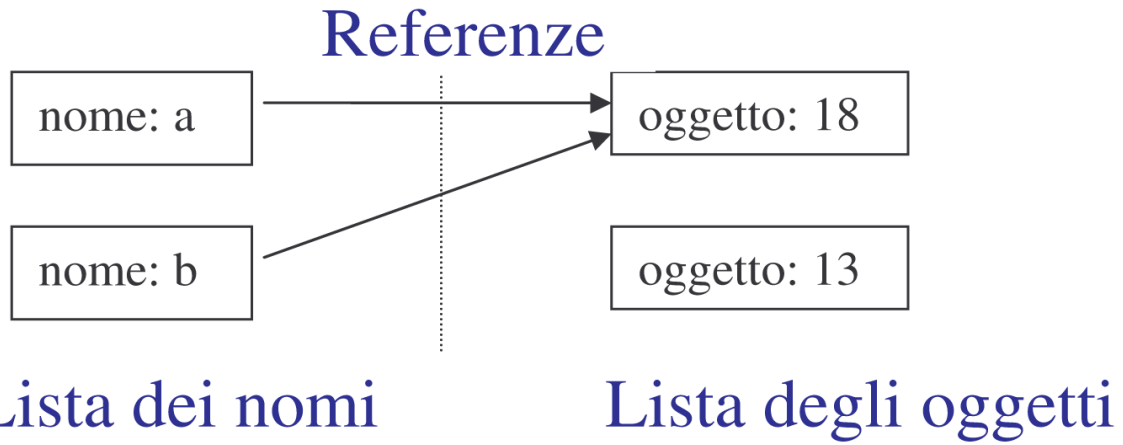


Approfondimento: variabili, oggetti e referenze (3/5)

Interpretiamo ora le seguenti assegnazioni aggiuntive

$$>>> b = a$$
$$>>> a = 13$$

1. Viene ricercato l'oggetto associato al nome *a* (cioè l'intero 18);
2. viene creato il nome *b* (nel caso in cui non esista già);
3. viene creato un collegamento (referenza) che partendo da *b*, punta alla locazione di memoria che contiene l'oggetto 18. A questo livello, i due nomi *a* e *b* puntano alla stessa area di memoria!
4. viene creato e conservato in memoria l'intero 13;

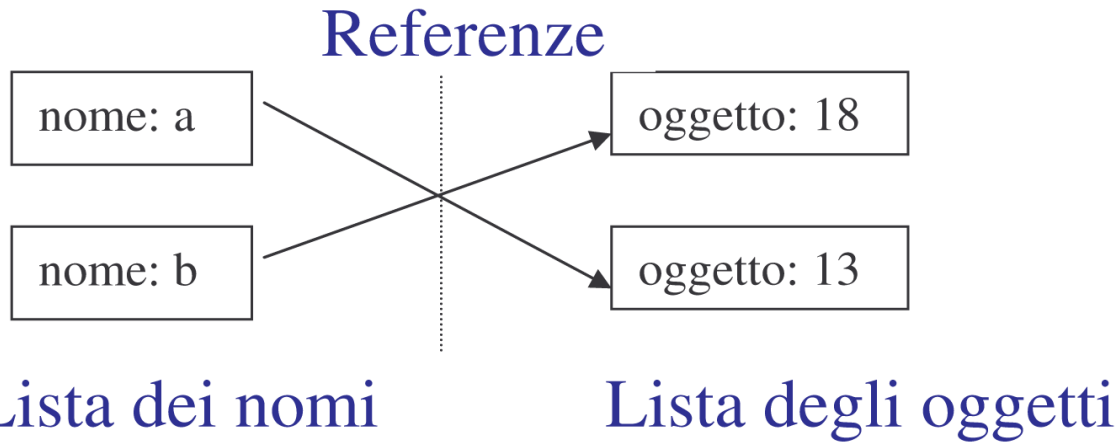


Approfondimento: variabili, oggetti e referenze (3/5)

Interpretiamo ora le seguenti assegnazioni aggiuntive

$$>>> b = a$$
$$>>> a = 13$$

1. Viene ricercato l'oggetto associato al nome *a* (cioè l'intero 18);
2. viene creato il nome *b* (nel caso in cui non esista già);
3. viene creato un collegamento (referenza) che partendo da *b*, punta alla locazione di memoria che contiene l'oggetto 18. A questo livello, i due nomi *a* e *b* puntano alla stessa area di memoria!
4. viene creato e conservato in memoria l'intero 13;
5. al nome *a* viene assegnata una referenza alla locazione di memoria che contiene il nuovo oggetto 13.



Approfondimento: variabili, oggetti e referenze (4/5)

Quando lavoriamo con oggetti mutabili (quali le liste), vi sono operazioni che li modificano "sul posto" anziché creare nuovi oggetti (ad es. un assegnazione a un elemento di una lista). In tal caso occorre fare particolare attenzione agli oggetti condivisi da due o più variabili. Consideriamo il seguente esempio

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1[1] = 4
```

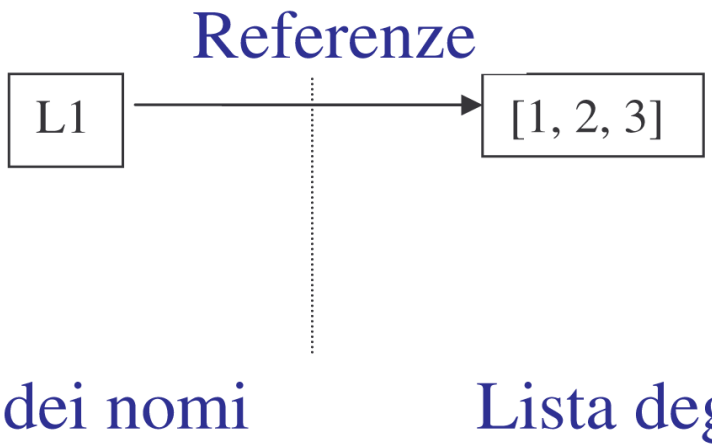
```
>>> L2
[1, 4, 3]
```

Approfondimento: variabili, oggetti e referenze (4/5)

Quando lavoriamo con oggetti mutabili (quali le liste), vi sono operazioni che li modificano "sul posto" anziché creare nuovi oggetti (ad es. un assegnazione a un elemento di una lista). In tal caso occorre fare particolare attenzione agli oggetti condivisi da due o più variabili. Consideriamo il seguente esempio

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1[1] = 4
>>> L2
[1, 4, 3]
```

1. Alla variabile $L1$ viene assegnata la lista $[1, 2, 3]$;



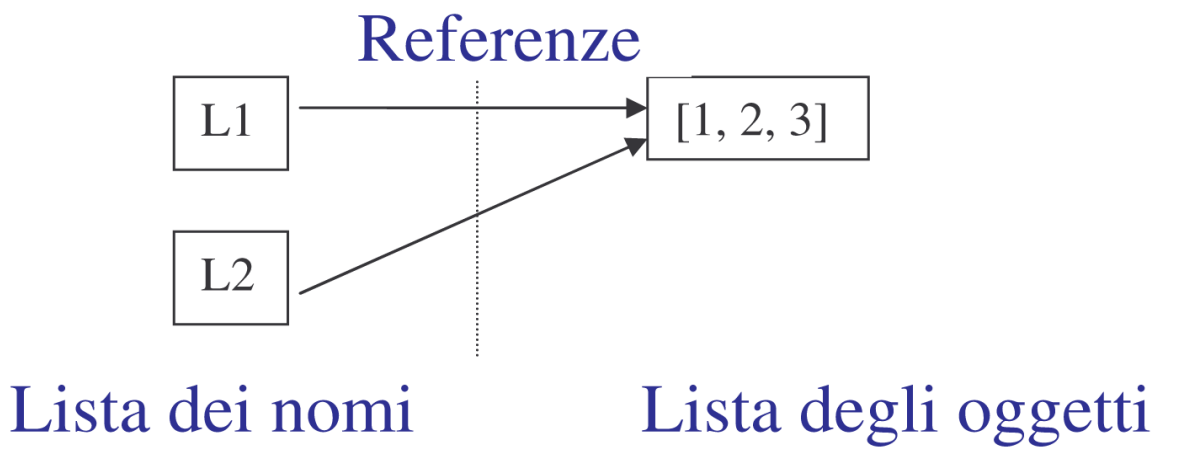
Approfondimento: variabili, oggetti e referenze (4/5)

Quando lavoriamo con oggetti mutabili (quali le liste), vi sono operazioni che li modificano "sul posto" anziché creare nuovi oggetti (ad es. un assegnazione a un elemento di una lista). In tal caso occorre fare particolare attenzione agli oggetti condivisi da due o più variabili. Consideriamo il seguente esempio

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1[1] = 4
```

```
>>> L2
[1, 4, 3]
```

1. Alla variabile $L1$ viene assegnata la lista $[1, 2, 3]$;
2. la variabile $L2$ referenzia lo stesso oggetto referenziato dalla variabile $L1$



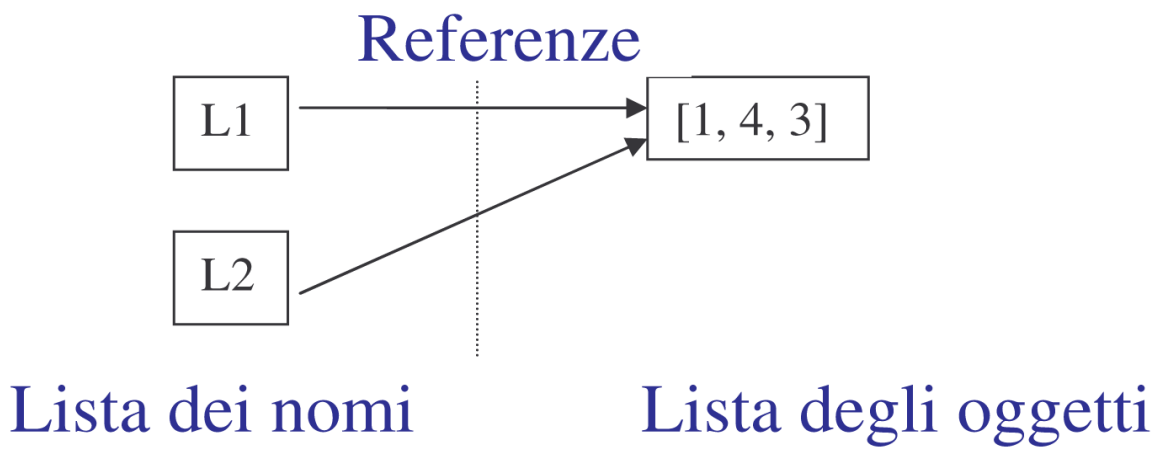
Approfondimento: variabili, oggetti e referenze (4/5)

Quando lavoriamo con oggetti mutabili (quali le liste), vi sono operazioni che li modificano "sul posto" anziché creare nuovi oggetti (ad es. un assegnazione a un elemento di una lista). In tal caso occorre fare particolare attenzione agli oggetti condivisi da due o più variabili. Consideriamo il seguente esempio

```
>>> L1 = [1, 2, 3]
>>> L2 = L1
>>> L1[1] = 4
```

```
>>> L2
[1, 4, 3]
```

1. Alla variabile $L1$ viene assegnata la lista $[1, 2, 3]$;
2. la variabile $L2$ referenzia lo stesso oggetto referenziato dalla variabile $L1$
3. Modifichiamo "in place" un elemento della lista. La modifica si riflette anche sulla variabile $L2$.



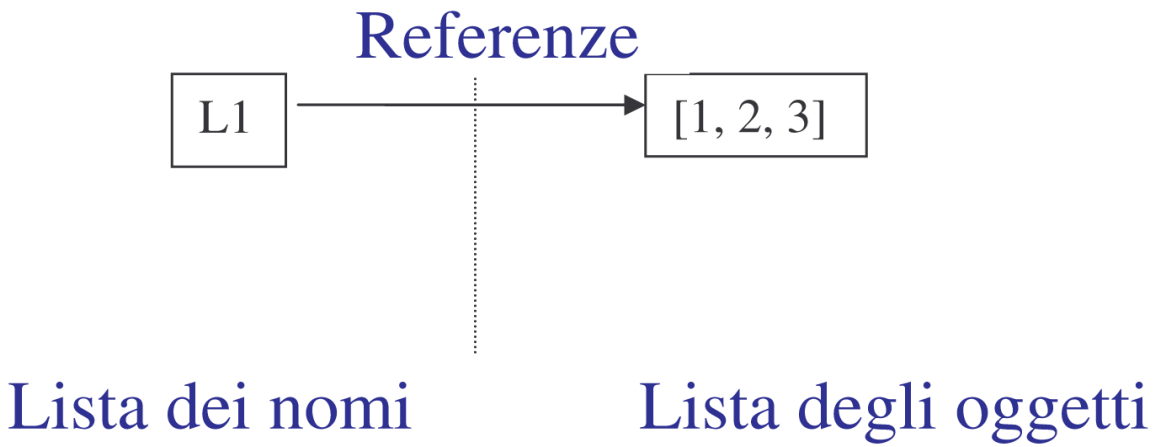
Approfondimento: variabili, oggetti e referenze (5/5)

Quando si vogliono evitare le referenze condivise agli oggetti mutabili, conviene, in fase di assegnazione, creare una copia dell'oggetto. Vediamo come riscrivere il precedente esempio.

Approfondimento: variabili, oggetti e referenze (5/5)

Quando si vogliono evitare le referenze condivise agli oggetti mutabili, conviene, in fase di assegnazione, creare una copia dell'oggetto. Vediamo come riscrivere il precedente esempio.

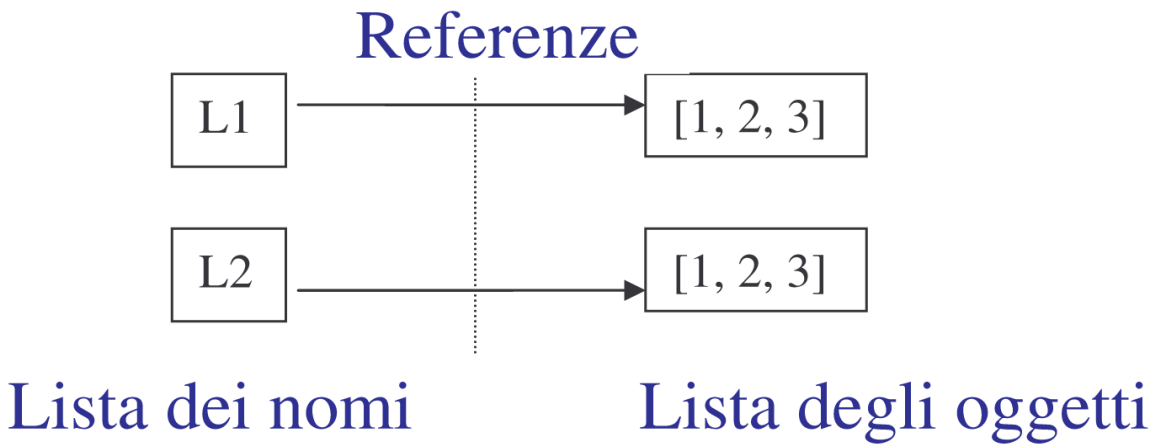
```
>>> L1 = [1, 2, 3]
```



Approfondimento: variabili, oggetti e referenze (5/5)

Quando si vogliono evitare le referenze condivise agli oggetti mutabili, conviene, in fase di assegnazione, creare una copia dell'oggetto. Vediamo come riscrivere il precedente esempio.

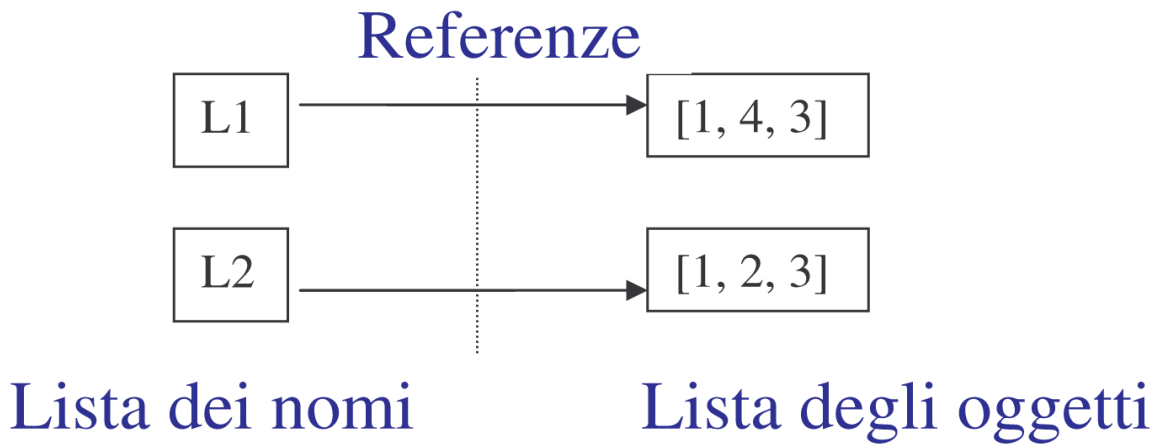
```
>>> L1 = [1, 2, 3]
>>> L2 = L1[:]    <-- crea una copia di L1
```



Approfondimento: variabili, oggetti e referenze (5/5)

Quando si vogliono evitare le referenze condivise agli oggetti mutabili, conviene, in fase di assegnazione, creare una copia dell'oggetto. Vediamo come riscrivere il precedente esempio.

```
>>> L1 = [1, 2, 3]
>>> L2 = L1[:]    <-- crea una copia di L1
>>> L1[1] = 4
>>> L2
[1, 2, 3]
```



Esercizi (sull'uso delle liste e dei costrutti for e if)

1. Scrivere una function di nome ordina che ha in input un vettore x ed in output il vettore che si ottiene da x ordinando le sue componenti in senso crescente. Confrontare con la funzione predefinita sort del modulo numpy.

ESERCIZIO

Congettura di Collaz. Sia x_0 un numero intero positivo. Per $n = 0, 1, 2, \ldots$, si consideri la successione $\{x_n\}$ definita ricorsivamente come segue:

$$x_{n+1} = \begin{cases} \dfrac{x_n}{2}, & \text{se } x_n \text{ è pari,} \\ 3x_n + 1, & \text{se } x_n \text{ è dispari.} \end{cases}$$

La congettura di Collaz[5] afferma che qualsiasi sia il punto iniziale x_0, l'algoritmo raggiunge sempre il valore 1 dopo un numero finito di passi, ovvero:

$$\text{per ogni intero positivo } x_0, \quad \text{esiste} \quad n \in \mathbb{N} \text{ tale che } x_n = 1.$$

Esercizio. Si scriva una funzione che abbia in input un numero intero positivo x_0 e restituisca in output

- il più piccolo intero n tale che $x_n = 1$;
- una lista x che contenga l'intera sequenza di valori $x_0, x_1, \ldots, x_n$.

Suggerimento. Per stabilire se un numero è pari o dispari, può essere utile la function *mod* del modulo *numpy*. Se x e y sono due numeri interi positivi, $mod(x, y)$ restituisce il resto della divisione tra x e y. Dunque se $mod(x, 2)$ è uguale a 0, x è pari, mentre se è uguale a 1, allora x è dispari.

[5] http://it.wikipedia.org/wiki/Congettura_di_Collatz.

Definizione di matrice reale

Definizione

Dati due interi positivi m ed n, una matrice A reale $m \times n$ è un array bidimensionale avente m righe ed n colonne così definito

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ \vdots & \vdots & & & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{pmatrix}$$

Più formalmente possiamo dire che una matrice è un'applicazione

$$A : \{1, 2, \ldots, m\} \times \{1, 2, \ldots, n\} \longrightarrow \mathbb{R}$$

tale che $A(i, j) = a_{ij} \in \mathbb{R}$.

Notazioni

Una matrice verrà solitamente denotata con le lettere maiuscole dell'alfabeto, mentre gli elementi di una matrice con le lettere minuscole; ad esempio la scrittura

$$A = \{a_{ij}\}_{\substack{i=1,\dots m \\ j=1,\dots n}}, \text{ ovvero } \quad A = \{a_{ij}\}, \quad i = 1, \dots m, \quad j = 1, \dots n,$$

denoterà una matrice ad m righe ed n colonne il cui generico elemento è a_{ij}. Denotiamo con $\mathbb{R}^{m\times n}$ l'insieme delle matrici con m righe ed n colonne.

Esempio ($A \in \mathbb{R}^{3\times 4}$)

$$A = \begin{pmatrix} 2 & 0 & -1 & \sqrt{3} \\ \pi & \log(3) & -1/3 & 1 \\ 2/3 & 0 & \sin(\pi/7) & 4/3 \end{pmatrix},$$

è una matrice 3×4*, ovvero a* 3 *righe e* 4 *colonne.*

Definire una matrice in Python (1/2)

In Python, una matrice può essere definita mediante la function matrix del modulo numpy. Nel seguito supporremo, per semplicità di notazione, di aver caricato il modulo numpy mediante l'istruzione

```
>>> from numpy import *
```

La funzione matrix converte una lista, una tupla o una stringa in un oggetto di tipo matrice. Esempi:

```
>>> A=matrix("1 2 3;-1 0 1")
>>> A
matrix([[ 1,  2,  3],
        [-1,  0,  1]])
```

In questo esempio A è una matrice 2×3. Equivalentemente, avremmo potuto scrivere:

```
>>> A=matrix([[1, 2, 3],[-1, 0, 1]])
>>> A=matrix(((1, 2, 3),(-1, 0, 1)))
```

Definire una matrice in Python (2/2)

In alternativa all'uso della funzione matrix è possibile utilizzare la funzione array del modulo numpy. Essa è più generale poiché può gestire array a uno o a più indici. Esempio:

```
>>> B=array([[1, 2, 3],[-1, 0, 1]])
>>> B
array([[ 1,  2,  3],
       [-1,  0,  1]])
```

I due tipi matrix e array condividono diversi metodi. Per ottenere la lista dei metodi associati ai due tipi, digitare, dal prompt dei comandi, dir(A) per la matrice definita in precedenza e dir(B) per l'array definito qui sopra.
Benché meno generale e meno diffuso rispetto all'array, l'oggetto matrix è più idoneo nel contesto dell'algebra lineare numerica e pertanto, insieme all'array, sarà utilizzato durante il presente corso. In ogni caso è sempre possibile:

- riguardare una variabile di tipo array quale matrice (e viceversa) mediante le funzioni asmatrix e asarray.
- trasformare un array in una matrice mediante le funzioni matrix e array.

Accedere agli elementi di una matrice (o di un array)

```
>>> A=matrix("1 2 3;4 5 6")
>>> A
matrix([[1, 2, 3],
        [4, 5, 6]])
>>> A[0,0]
1
>>> A[1,0]
4
>>> A[1,2]
6
>>> A[1,3]
IndexError: index 3 is out of
bounds for axis 1 with size 3
>>> A[0,:]
matrix([[1, 2, 3]])
>>> A[:,0]
matrix([[1],
        [4]])
```

```
>>> B=array(A)
>>> B
array([[1, 2, 3],
       [4, 5, 6]])
>>> B[0,2]
3
>>> B[1,:]
array([4, 5, 6])
>>> B[:,2]
array([3, 6])
>>> B[1,1:3]
array([5, 6])
>>> B[0:2,0:2]
array([[1, 2],
       [4, 5]])
```

ESERCIZIO

Si scriva una funzione che abbia

- in input un vettore (array o lista) x le cui componenti siano interi compresi tra 0 e 9;
- in output un'array (o matrice) A di due colonne così definita:
  - la prima colonna di A riporta le cifre distinte del vettore x
  - il generico elemento della seconda colonna di A riporta il numero di volte in cui la cifra corrispondente nella prima colonna compare nel vettore x.

ESEMPIO:

$$x = (6, 4, 0, 6, 8, 0, 0) \Longrightarrow A = \begin{pmatrix} 0 & 3 \\ 4 & 1 \\ 6 & 2 \\ 8 & 1 \end{pmatrix}$$

Matrici particolari (1/4)

- Se $m = n$, (num. di righe= num. di colonne), la matrice è detta quadrata di dimensione n (se $m \neq n$, la matrice è detta rettangolare);
- se $m = n = 1$, la matrice si riduce ad un unico elemento e dunque coincide con uno scalare: $A = (a_{11})$;
- se $m = 1$, la matrice possiede un'unica riga, pertanto si riduce ad un vettore riga: $A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \end{pmatrix}$
- se $n = 1$, la matrice possiede un'unica colonna, pertanto si riduce ad un vettore colonna:

$$A = \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} \equiv \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \end{pmatrix}^T.$$

Matrici particolari (2/4)

- La matrice $m \times n$ i cui elementi sono tutti nulli si chiama matrice nulla e si denota con $0_{m \times n}$ o più semplicemente con 0.

  Per ottenere una matrice nulla in Python, anziché elencare i suoi elementi, si può utilizzare la function predefinita zeros del modulo numpy:

```
>>>zeros(shape=(2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

  Se invece usiamo la function *ones*:

```
>>>ones(shape=(2,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

  È anche possibile utilizzare la forma contratta zeros((2,3)) e ones((2,3)). Come già osservato, possiamo convertire gli array in matrici mediante la funzione matrix.

Matrici particolari (3/4)

- Si chiama matrice identica, ogni matrice quadrata avente elementi diagonali uguali ad 1 ed elementi extra-diagonali nulli:

$$I = \begin{pmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & \cdots & \cdots & 0 \\ \vdots & \vdots & & & \vdots \\ 0 & 0 & \cdots & \cdots & 1 \end{pmatrix}$$

Per ottenere una matrice identica in Python si usa la function predefinita identity:

```
>>>identity(4)
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

Matrici quadrate particolari (4/4)

- Una matrice quadrata A è detta:
  - diagonale se tutti i suoi elementi extra-diagonali sono nulli: $a_{ij} = 0,\ \forall i, j = 1, \ldots n, \quad i \neq j$;
  - triangolare inferiore se tutti i suoi elementi al di sopra della diagonale principale sono nulli: $a_{ij} = 0,\ \forall i < j$;
  - triangolare superiore se tutti i suoi elementi al di sotto della diagonale principale sono nulli: $a_{ij} = 0,\ \forall i > j$;
- Esempi:

$$D = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 3 \end{pmatrix}, \quad T = \begin{pmatrix} 1 & 0 & 0 \\ -1 & -2 & 0 \\ 2 & -4 & 3 \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & -2 \\ 0 & 0 & 3 \end{pmatrix}.$$

```
>>>D=diag([1,-2,3])
>>>D
array([[1, 0, 0],
       [0,-2, 0],
       [0, 0, 3]])
```

Addizione tra matrici (1/2)

Definizione

Se $A = \{a_{ij}\}$ *e* $B = \{b_{ij}\}$ *sono matrici* $m \times n$ *si definisce somma tra* A *e* B *la matrice*

$$A + B = \{a_{ij} + b_{ij}\} \in \mathbb{R}^{m \times n}$$

Esempio

$$A = \begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 1 \\ 2/3 & 1 & -2 \end{pmatrix}, \; B = \begin{pmatrix} 1 & 0 & 1 \\ -1 & -2 & 0 \\ 1/3 & -4 & 3 \end{pmatrix}, \; A + B = \begin{pmatrix} 2 & 2 & 4 \\ -2 & -2 & 1 \\ 1 & -3 & 1 \end{pmatrix}.$$

Addizione tra matrici (2/2): Esempio Python

```
>>> A=matrix("1 2 3;-1 0 1")
>>> B=matrix("1 0 1; -1 -2 0")
>>> C=A+B
>>> print(C)
[[ 2  2  4]
 [-2 -2  1]]
```

Funziona anche riguardando A e B come array:

```
>>> asarray(A)+asarray(B)
array([[ 2,  2,  4],
       [-2, -2,  1]])
```

Moltiplicazione di uno scalare per una matrice (1/2)

Definizione

Se $A = \{a_{ij}\} \in \mathbb{R}^{m\times n}$ *e* $\lambda \in \mathbb{R}$, *si definisce prodotto di* λ *per* A *la matrice*

$$\lambda \cdot A = \{\lambda a_{ij}\} \in \mathbb{R}^{m\times n}$$

Esempio

$$A = \begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 1 \\ 2/3 & 1 & -2 \end{pmatrix}, \quad 2 \cdot A = \begin{pmatrix} 2 & 4 & 6 \\ -2 & 0 & 2 \\ 4/3 & 2 & -4 \end{pmatrix}.$$

Moltiplicazione di uno scalare per una matrice (2/2): Esempio Python

```
>>> A=matrix("1 2 3;-1 0 1")
>>> A
matrix([[ 1,  2,  3],
        [-1,  0,  1]])

>>> lam=2.5
>>> lam*A
matrix([[ 2.5,  5. ,  7.5],
        [-2.5,  0. ,  2.5]])
```

```
>>> B=array([[1, 2, 3],[-1, 0, 1]])
>>> B
array([[ 1,  2,  3],
       [-1,  0,  1]])

>>> lam=2.5
>>> lam*B
array([[ 2.5,  5. ,  7.5],
       [-2.5,  0. ,  2.5]])
```

Trasposta di una matrice (1/2)

Se $A \in \mathbb{R}^{m \times n}$, la trasposta di A, denotata con A^T, è la matrice ottenuta da A scambiando le righe con le colonne (o viceversa), ovvero

$$A^T = \{a_{ji}\}, \qquad i = 1, \ldots m, \; j = 1, \ldots n.$$

Pertanto $A^T \in \mathbb{R}^{n \times m}$.

Esempio

$$A = \begin{pmatrix} 1 & 2 & 3 \\ -1 & 0 & 1 \end{pmatrix} \Longrightarrow A^T = \begin{pmatrix} 1 & -1 \\ 2 & 0 \\ 3 & 1 \end{pmatrix}.$$

Trasposta di una matrice (1/2): Esempio Python

In Python per ottenere il trasposto si usa l'attributo .T oppure la funzione transpose (validi sia per le matrici che per gli array):

```
>>> A=matrix("1 2 3;-1 0 1")
>>> A
matrix([[ 1,  2,  3],
        [-1,  0,  1]])

>>> B=A.T
>>> B
matrix([[ 1, -1],
        [ 2,  0],
        [ 3,  1]])

>>> C=transpose(B)
>>> C
matrix([[ 1,  2,  3],
        [-1,  0,  1]])
```

Prodotto di matrici (righe per colonne)

- Ricordiamo che se $\mathbf{a}$ e $\mathbf{b}$ sono due vettori (colonna) di lunghezza n, il prodotto scalare di $\mathbf{a}$ e $\mathbf{b}$ denotato con $\mathbf{a}^T\,\mathbf{b}$ è così definito:

$$\mathbf{a}^T\,\mathbf{b} \equiv \sum_{k=1}^{n} a_k\, b_k.$$

- Siano $A \in \mathbb{R}^{m\times p}$ e $B \in \mathbb{R}^{p\times n}$. Si definisce prodotto (righe per colonne) tra A e B la matrice $C = A \cdot B \in \mathbb{R}^{m\times n}$ il cui elemento generico c_{ij} è il prodotto scalare tra la riga i-esima di A e la colonna j-esima di B:

$$c_{ij} = \mathbf{a}_i^T\,\mathbf{b}_j = \sum_{k=1}^{p} a_{ik}\, b_{kj}, \quad i = 1, \ldots m, \quad j = 1, \ldots, n.$$

$\mathbf{a}_i^T \rightarrow$ i-esima riga di A; $\quad \mathbf{b}_j \rightarrow$ j-esima colonna di B.

ESEMPIO

Osservazione

Il prodotto tra due matrici è possibile solo se il numero di colonne del primo fattore coincide con il numero di righe del secondo fattore.

$$A = \left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \hline -1 & 0 & 1 & 2 \end{array}\right) = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \end{pmatrix}, \quad B = \left(\begin{array}{c|c|c} 1 & -1 & 3 \\ -1 & 0 & 1 \\ 0 & 2 & -1 \\ 2 & 1 & -2 \end{array}\right) = \begin{pmatrix} \mathbf{b}_1, & \mathbf{b}_2, & \mathbf{b}_3 \end{pmatrix}$$

$$A \cdot B = \begin{pmatrix} \mathbf{a}_1^T \mathbf{b}_1 & \mathbf{a}_1^T \mathbf{b}_2 & \mathbf{a}_1^T \mathbf{b}_3 \\ \mathbf{a}_2^T \mathbf{b}_1 & \mathbf{a}_2^T \mathbf{b}_2 & \mathbf{a}_2^T \mathbf{b}_3 \end{pmatrix} = \begin{pmatrix} 7 & 9 & -6 \\ 3 & 5 & -8 \end{pmatrix}$$

$B \cdot A$ non è possibile.

Ulteriori esempi (1/2)

- $\left(\begin{array}{ccc} 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \end{array}\right) \cdot \begin{pmatrix} -1 \\ 1 \\ -2 \end{pmatrix} = \begin{pmatrix} -5 \\ -11 \\ -17 \end{pmatrix}$
- $\begin{pmatrix} -1 & 1 & -2 \end{pmatrix} \cdot \left(\begin{array}{c|c|c} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}\right) = \begin{pmatrix} -11 & -13 & -15 \end{pmatrix}$
-

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^2 = \left(\begin{array}{ccc} 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & 9 \end{array}\right) \cdot \left(\begin{array}{c|c|c} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}\right) = \begin{pmatrix} 30 & 36 & 42 \\ 66 & 81 & 96 \\ 102 & 126 & 150 \end{pmatrix}$$

Ulteriori esempi (2/2)

- $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 0 & -1 \\ -2 & 1 \end{pmatrix} = \begin{pmatrix} -4 & 1 \\ -8 & 1 \end{pmatrix}$
- $\begin{pmatrix} 0 & -1 \\ -2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} -3 & -4 \\ 1 & 0 \end{pmatrix}$

Osservazione

Dunque, se A e B sono quadrate dello stesso ordine, $A \cdot B$ e $B \cdot A$ sono ben definite, tuttavia, in generale A e B non sono permutabili cioè, in generale, $A \cdot B \neq B \cdot A$.

Ne segue che la moltiplicazione tra matrici non è commutativa.

Prodotto tra matrici: Esempio Python

```
>>> A=matrix("1 2 3; 4 5 6")
>>> A
matrix([[1, 2, 3],
        [4, 5, 6]])

>>> B=matrix("1 2 1 -1;1 -1 0 1;0 1 -2 1")
>>> B
matrix([[ 1,  2,  1, -1],
        [ 1, -1,  0,  1],
        [ 0,  1, -2,  1]])

>>> C=A*B
>>> C
matrix([[ 3,  3, -5,  4],
        [ 9,  9, -8,  7]])
```

N.B.: Definendo A e B come array l'operazione $*$ assume diverso significato. Per poter moltiplicare le due matrici utilizzare la funzione dot: `>>> dot(A,B)`

La funzione shape

La funzione shape applicata alla matrice A di dimensioni $m \times n$ restituisce una tupla di due elementi (m, n) contenente il numero m di righe e il numero n di colonne della matrice A. In Python:

```
>>> A=matrix("1 2 3;4 5 6")
>>> shape(A)
(2, 3)

>>> [m,n]=shape(A)
>>> print(m,n)
2 3
```

Esercizi

ESERCIZIO: Scrivere una function Python che ha in input una matrice A ed un vettore $\mathbf{x}$ ed in output il vettore $\mathbf{y} = A\mathbf{x}$:

$$y(i) = \sum_{j=1}^{n} A(i,j)x(j), \qquad i = 1, \ldots, m$$

essendo A di dimensioni $m \times n$ ed $\mathbf{x}$ di lunghezza n.

ESERCIZIO: Scrivere una function Python che ha in input due matrici A e B ed in output la matrice prodotto $C = A \cdot B$:

$$C(i,j) = \sum_{k=1}^{p} A(i,k)B(k,j), \qquad i = 1, \ldots, m, \quad j = 1, \ldots, n$$

essendo A di dimensioni $m \times p$ e B di dimensioni $p \times n$.

La funzione "arange" (1/3)

La funzione arange è frequentemente utilizzata in Python.

- Se $n1 \in \mathbb{N}$ ed $n2 \in \mathbb{N}$, con $n1 < n2$, mediante l'espressione arange(n1,n2) si ottiene un array che contiene tutti i numeri interi compresi tra $n1$ e $n2 - 1$. Esempi:

```
>>> arange(1:11)
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

>>> arange(2:3)
array([2])

>>> arange(2,2)
array([], dtype=int64)

>>> arange(10,1)
array([], dtype=int64)
```

In questa accezione arange assume lo stesso significato di range usato in precedenza nel ciclo for.

La funzione "arange" (2/3)

- Più in generale vale la seguente regola. Se $a \in \mathbb{R}$, $b \in \mathbb{R}$ e $h \in \mathbb{R}$, l'istruzione arange(a,b,h) restituisce un array i cui elementi sono

$$a,\ a+h,\ a+2h,\ \ldots\ ,a+mh$$

dove m è un numero intero tale che

$$a+mh<b \qquad \text{e} \quad a+(m+1)h \geq b.$$

Questo significa che gli elementi del vettore di output vanno da a a b con incremento h, arrestandosi al numero che non supera (o uguaglia) b. L'incremento h può essere un numero reale positivo o negativo.

La funzione “arange” (3/3) ESEMPI in Python

```
>>> arange(10,20,2)
array([10, 12, 14, 16, 18])

>>> arange(10,21,3)
array([10, 13, 16, 19])

>>> arange(100,78,-5)
array([100,  95,  90,  85,  80])

>>> arange(0,pi/4,0.1)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7])
```

La funzione *linspace*

Permette di ottenere lo stesso risultato della funzione arange, prefissando però il numero di punti anziché il passo.

La funzione linspace serve per costruire un vettore di punti equidistanti: mediante linspace(x1,x2) si ottiene un array di 50 punti equidistanti compresi tra $x1$ e $x2$, mentre con linspace(x1,x2,n) si ottiene un array di n elementi equidistanti compresi tra $x1$ e $x2$. Esempio

```
>>> linspace(0,1,11)
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,

>>> linspace(0,pi,8)
array([ 0.        ,  0.44879895,  0.8975979 ,  1.34639685,
        1.7951958 ,  2.24399475,  2.6927937 ,  3.14159265])
```

Le operazioni componentwise

Anziché effettuare la moltiplicazione nel senso righe per colonne tra due matrici (o vettori), l'operazione di moltiplicazione "$*$" eseguita su due array delle stesse dimensioni effettua la moltiplicazione *elemento per elemento*, restituendo una matrice i cui elementi sono il prodotto degli elementi omonimi dei due fattori.

Ad esempio, considerati $\mathbf{x} = [x_1,\ x_2,\ x_3]$ ed $\mathbf{y} = [y_1,\ y_2,\ y_3]$, avremo:

$$\mathbf{x} * \mathbf{y} = [x_1 y_1,\ x_2 y_2,\ x_3 y_3]$$

Analogamente, avremo:

$$\mathbf{x}/\mathbf{y} = [x_1/y_1,\ x_2/y_2,\ x_3/y_3]$$

e

$$\mathbf{x} * *\mathbf{y} = [x_1^{y_1},\ x_2^{y_2},\ x_3^{y_3}]$$

ESEMPIO in Python

```
>>> A=array([[1,2,3],[4,5,6]])
>>> A
array([[1, 2, 3],
       [4, 5, 6]])
>>> B=array([[-2, 4, 2],[-1, 3, -2]])
>>> B
array([[-2,  4,  2],
       [-1,  3, -2]])

>>> A*B
array([[ -2,   8,   6],
       [ -4,  15, -12]])

>>> A/B
array([[-0.5       ,  0.5       ,  1.5       ],
       [-4.        ,  1.66666667, -3.        ]])
```

Tabulare una funzione (1/2)

Consideriamo una funzione reale di variabile reale $y = f(x)$. Sia $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ un vettore di elementi appartenenti al dominio di f. Vogliamo costruire il vettore delle valutazioni di f, cioè

$$\mathbf{y} = [f(x_1),\ f(x_2), \ldots, f(x_n)].$$

Definiamo in Python il vettore

```
>>> x=linspace(0,pi,5)
>>> x
array([0., 0.78539816, 1.57079633, 2.35619449, 3.14159265])
```

e corrispondentemente valutiamo le seguenti funzioni.

- $y = \sin(x)$:

```
>>> y=sin(x)
>>> y
array([0.00000000e+00, 7.07106781e-01, 1.00000000e+00,
                       7.07106781e-01, 1.22464680e-16])
```

Tabulare una funzione (2/2)

- $y = \sin(x)\cos(x)$:

```
>>> y2=sin(x)*cos(x)
>>> y2
array([0.00000000e+00,   5.00000000e-01,  6.12323400e-17,
       -5.00000000e-01,  -1.22464680e-16])
```

- $y = x^2\, e^{-x}$:

```
>>> y=(x**2)*exp(-x)
>>> y
array([0., 0.2812455, 0.512922, 0.5261868, 0.4265042])
```

- $y = \dfrac{x}{\cos(x)}$:

```
>>> y=x/cos(x)
>>> y
array([ 0.00000000e+00,  1.11072073e+00, 2.56530508e+16,
       -3.33216220e+00, -3.14159265e+00])
```

Grafico di una funzione: esempio Python (1/2)

Si voglia rappresentare il grafico della funzione $y = \sin(x)e^{-x}$ nell'intervallo $[0, 2\pi]$. Per le istruzioni grafiche faremo uso del modulo pylab che caricheremo in memoria ad esempio mediante il comando

```
>>> from pylab import *
```

Le righe di codice per la costruzione del grafico della funzione sono:

```
>>> x=linspace(0,2*pi,100)
>>> y=sin(x)*exp(-x)
>>> plot(x,y)
```

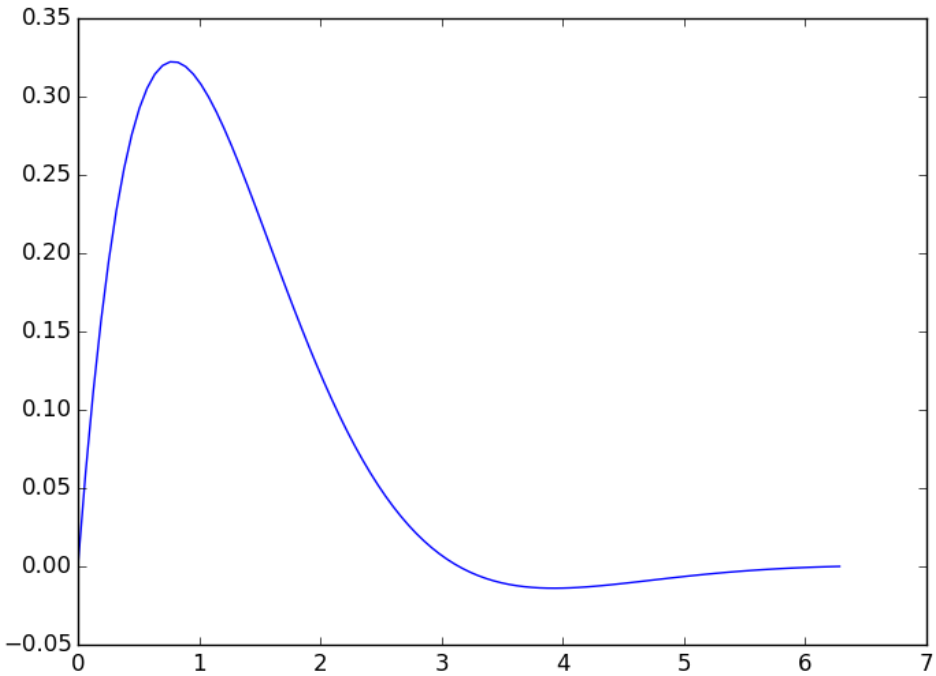
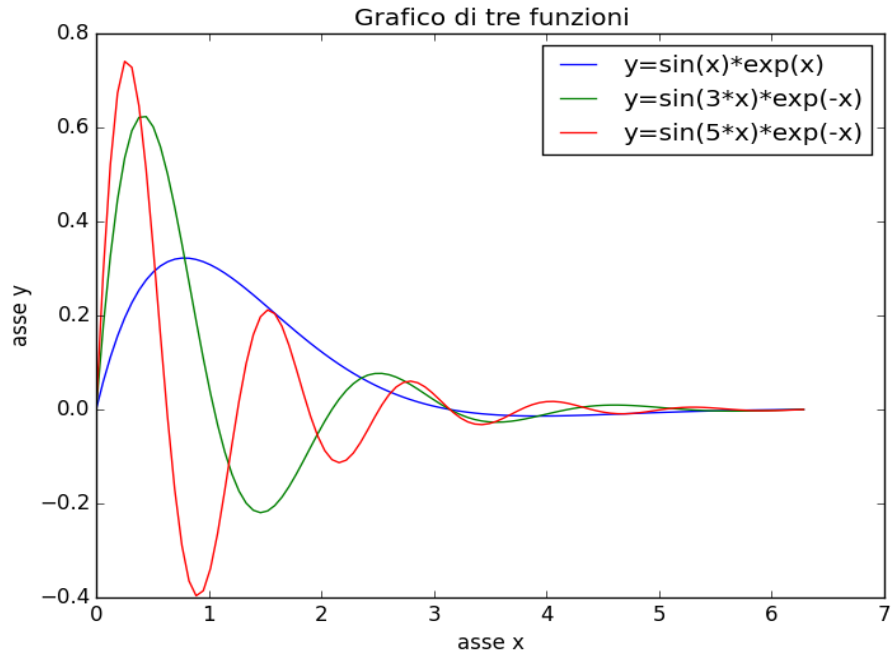


Grafico di una funzione: esempio Python (2/2)

Si vogliano rappresentare sugli stessi assi i grafici delle funzioni $y = \sin(x)e^{-x}$, $y = \sin(3x)e^{-x}$, $y = \sin(5x)e^{-x}$. Le righe di codice:

```
>>> x=linspace(0,2*pi,100)
>>> y=sin(x)*exp(-x)
>>> y1=sin(3*x)*exp(-x)
>>> y2=sin(5*x)*exp(-x)
>>> plot(x,y,label="y=sin(x)*exp(x)")
>>> plot(x,y1,label="y=sin(3*x)*exp(-x)")
>>> plot(x,y2,label="y=sin(5*x)*exp(-x)")
>>> legend(loc='upper right')
>>> xlabel("asse x")
>>> ylabel("asse y")
>>> title("Grafico di tre funzioni")
```

producono il grafico



APPENDICI

A completamento di questa nota introduttiva sull'uso di Python, facciamo un breve cenno ad alcuni argomenti fondamentali ma che hanno un'importanza marginale nell'ambito degli obiettivi del corso. Verranno discussi:

1. Dizionari
2. Insiemi
3. Scrittura su un file e lettura da file
4. Il modulo os di Python
5. Alcune generalizzazioni del costrutto for

Dizionari (definizione)

Un dizionario è una struttura non ordinata di elementi che realizza un'associazione tra due insiemi:

- l'insieme delle chiavi (keys)
- l'insieme dei valori (values)

Le chiavi possono essere di qualsiasi tipo, purché immutabile. I valori possono essere di qualsiasi tipo (anche mutabile). Esempi:

```
>>> rubrica={"Paolo" : "328364959","Giacomo" : "3371294839"}
>>> rubrica["Paolo"]
'328364959'
>>> rubrica["Giacomo"]
'3371294839'
>>> len(rubrica)
2
```

Dizionari (manipolazione)

I dizionari sono oggetti mutabili.

```
>>> rubrica.keys()                          <-- lista delle chiavi
['Paolo', 'Giacomo']
>>> rubrica.values()                        <-- lista dei valori
['328364959', '3371294839']
>>> rubrica.items()
[('Paolo', '328364959'), ('Giacomo', '3371294839')]
>>> rubrica
{'Paolo': '328364959', 'Giacomo': '3371294839'}
>>> rubrica["Paolo"]="3391234567"           <-- modifica un item
>>> rubrica
{'Paolo': '3391234567', 'Giacomo': '3371294839'}
>>> rubrica["Tiziana"]="3492837460"         <-- aggiunge un item
>>> rubrica
{'Paolo': '3391234567', 'Tiziana': '3492837460', 'Giacomo': '3371294839'}
>>> del rubrica["Giacomo"]                  <-- elimina un item
>>> rubrica
{'Paolo': '3391234567', 'Tiziana': '3492837460'}
>>> rubrica.clear()                         <-- elimina tutto
>>> rubrica
{}
```

Insiemi (definizione)

Un insieme (set) è una struttura non ordinata e non necessariamente omogenea di elementi. Viene creato partendo da altre strutture mediante la funzione predefinita set.

```
>>> pari=[0,2,4,6,2,0,8,0,8]
>>> s1=set(pari)
>>> s1
set([0, 8, 2, 4, 6])
>>> type(s1)
set
>>> dispari=(1,1,3,3,5,5,7,9)
>>> s2=set(dispari)
>>> s2
set([1, 3, 9, 5, 7])
>>> vocali="AEIOU"
>>> s3=set(vocali)
>>> s3
set(['A', 'I', 'E', 'U', 'O'])
>>> "O" in s3
True
>>> "B" in s3
False
```

Insiemi (manipolazione)

Gli insiemi sono oggetti mutabili.

```
>>> s1 | s2                          <-- unione
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> s1 & s2                          <-- intersezione
set([])
>>> s3
set(['A', 'I', 'E', 'U', 'O'])
>>> s4=set("CIAO")
>>> s4
set(['I', 'A', 'C', 'O'])
>>> s3 & s4
set(['I', 'A', 'O'])
>>> s4 - s3                          <-- differenza
set(['C'])
>>> s4.remove("C")
>>> s4
set(['I', 'A', 'O'])
>>> s4.issubset(s3)
True
>>> s4.add(17)
>>> s4
set(['I', 'A', 'O', 17])
```

Leggere da un file – Scrivere in un file

Nelle applicazioni, è importante avere uno strumento che consenta di:

1. poter salvare delle informazioni (risultato delle nostre elaborazioni) sul disco e renderle di conseguenza persistenti.
2. poter leggere delle informazioni da file, da utilizzare ad es. come input per le nostre elaborazioni.

Python permette di svolgere le comuni operazioni sui file: apertura e chiusura di un file, lettura e scrittura su un file, ecc.

L'istruzione per aprire un file è open. La sintassi completa è:

nomeoggetto=open("nomefile", modalità)

dove nomefile indica il nome del file da aprire, di solito un file di testo (ad es. dati.txt). Il secondo argomento indica l'operazione che si intende compiere sul file. Le possibilità sono:

"w" : scrittura (write)

"r" : lettura (read)

"a" : appende (append) il testo in coda a quello preesistente

"r+" : lettura e scrittura

La funzione open, restituisce un oggetto di tipo file che verrà assegnato al nome nomeoggetto. I metodi di tale oggetto permetteranno la manipolazione del file.

Scrivere in un file: un semplice esempio

ESERCIZIO: Aprire un nuovo file di nome prova.txt nella directory di lavoro, e scrivere il nostro nome e cognome.

Per fissare le idee supponiamo che il path che identifica la nostra directory sia

C:\python

Diversamente si dovrà far riferimento al path completo che identifichi la nostra cartella di lavoro.

```
>>> prova=open("C:\python\prova.txt","w")
>>> prova.write("Felice Iavernaro")
>>> prova.close()
```

Ora apriamo il file prova.txt (con un qualsiasi editor di testo) e leggiamone il contenuto.

Leggere da un file: un semplice esempio

ESERCIZIO: Aprire il file di nome prova.txt nella directory di lavoro (C:\python), leggerne e stamparne il contenuto.

```
>>> prova=open("C:\python\prova.txt","r")
>>> nominativo=prova.readline()
>>> prova.close()
>>> print(nominativo)
Felice Iavernaro
```

DOMANDA: C'e' un modo per evitare di riportare il path completo?

RISPOSTA: Vedi la prossima slide.

Il modulo os (1/2)

Il modulo os gestisce diverse funzionalità del sistema operativo. È spesso utile se si desidera rendere i programmi "platform-independent", ovvero renderli eseguibili indipendentemente dal sistema operativo utilizzato (Windows, Linux, Mac, ecc.). Essendo un modulo, va importato mediante il comando

```
>>> import os
```

Ecco alcune utili funzionalità:

- os.name: restituisce una stringa che specifica la piattaforma su cui stiamo lavorando (nt per Windows, posix per Linux/Unix,...)
- os.getcwd(): restituisce una stringa che indica la directory corrente di lavoro (cwd=current working directory)
- os.chdir(): cambia la directory corrente di lavoro
- os.sep: separatore utilizzato nella descrizione di un percorso
- os.listdir(): elenca il contenuto della directory specificata dal path in argomento
- os.remove(): rimuove il file indicato in argomento

Il modulo os (2/2)

- os.sys.path: lista di stringhe, ciascuna delle quali definisce un path all'interno del quale python esegue la ricerca dei file che vogliamo utilizzare
- os.sys.path.append("C:\python"): appende alla lista un nuovo path, ad esempio quello che identifica la directory entro cui salviamo i nostri file

I seguenti esempi sono stati riprodotti su un sistema Windows NT

```
>>> import os
>>> os.name
'nt'
>>> os.getcwd()
'C:\\Python27\\Lib\\idlelib'
>>> os.chdir("C:\python")
>>> os.getcwd()
'C:\\python'
>>> os.sep
'\\'
>>> os.listdir(".") <-- il punto indica la directory corrente
['prova.txt']
>>> os.remove("prova.txt")
>>> os.listdir(".")
[]
```

Esempi di scrittura e lettura (1/2)

```
>>> es=open("esempio.txt","w")
>>> es.write("Ciccio;3284354123\n")   <-- il ";"  e' un separatore tra
>>> es.write("Elvira;3391234567\n")       i due campi (nome e numero)
>>> es.write("Marco;3479876543\n")        \n indica l'andata a capo
>>> es.close()                            (su Mac: \r)
```

Controlliamo il contenuto del file esempio.txt. Aggiungiamo due nomi:

```
>>> es=open("esempio.txt","a")
>>> es.writelines(["Carla;3336543210\n","Paolo;3389753108\n"])
>>> es.close()
```

Il metodo writelines permette la scrittura di più stringhe, inserite come elementi di una lista (o una tupla).

Ricontrolliamo il contenuto del file esempio.txt. Ora leggiamone il contenuto e carichiamo la rubrica in un dizionario.

Esempi di scrittura e lettura (2/2)

```
>>> import string   <-- modulo che carica metodi per
>>> rubrica={}          la manipolazione di stringhe
>>> es=open("esempio.txt","r")
>>> riga=es.readline()
>>> riga
'Ciccio;3284354123\n'
>>> while riga!="":
      [nome,numero]=string.split(riga[0:-1],";")
      rubrica[nome]=numero
      riga=es.readline()

>>> rubrica
{'Paolo': '3389753108', 'Carla': '3336543210', 'Marco': '3479876543',
'Ciccio': '3284354123', 'Elvira': '3391234567'}
>>> es.close()
```

ESERCIZIO: Fare l'operazione contraria: memorizzare la rubrica nel file agenda.txt

Alcune generalizzazioni del costrutto for

Risoluzione dell'esercizio precedente

```
>>> agenda=open("agenda.txt","w")
>>> for nome in rubrica.keys():
        riga=nome+";"+rubrica[nome]+"\n"
        agenda.write(riga)

>>> agenda.close()
```

oppure

```
>>> agenda=open("agenda.txt","w")
>>> for nome,numero in rubrica.items():
riga=nome+";"+numero+"\n"
agenda.write(riga)

>>> agenda.close()
```

Altro esempio di for

```
>>> stringa="ciao"
>>> for k in stringa:
        print(k)
c
i
a
o
```