# LABORATORIO DI PROGRAMMAZIONE 2
## Corso di laurea in matematica

## Introduzione al linguaggio PYTHON

Marco Lapegna
Dipartimento di Matematica e Applicazioni
Universita' degli Studi di Napoli Federico II

wpage.unina.it/lapegna



## Python

**Python** e' un linguaggio di programmazione ad alto livello sviluppato negli anni '90 da Guido Van Rossum (collaborazioni con Google e Dropbox)

Guido Van Rossum

E' **semplice** da imparare
- sintassi pulita e semplice
- linguaggio ridotto al minimo

E' estremamente **versatile**
- facile interazione con altri linguaggi
- numerose librerie aggiuntive disponibili

E' **utilizzato** in numerosi campi e aziende
- data science, intelligenza artificiale, web programming
- Google, Firaxis, National Weather Service, Dropbox…

Il logo di Python

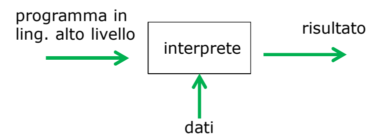


## compilatore vs interprete

**L'interprete:**

- traduce un'istruzione del programma in linguaggio macchina
- esegue immediatamente l'istruzione, prelevando dalla memoria i dati necessari

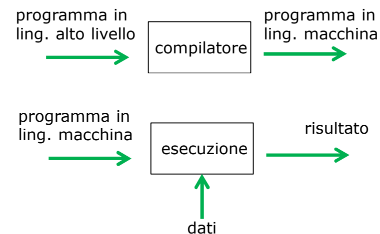
traduzione e esecuzione del programma sono **contestuali**




**Il compilatore**

- traduce l'intero programma in linguaggio macchina che viene conservato nel file system
- Viene eseguito il programma compilato

Traduzione e esecuzione del programma sono **separate**






## Python?

Python **e' un linguaggio interpretato**
**due modalita' di esecuzione**

**1) modalita interattiva**

- viene mandato in esecuzione l'interprete (python3)
- vengono fornite interattivamente le istruzioni una alla volta
- si termina con quit()

```
lapegna@lapemax:~/PYTHON$
lapegna@lapemax:~/PYTHON$ python3
Python 3.5.2 (default, Jul 10 2019, 11:58:48)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> a = 3
>>> b = 5
>>> c = a+b
>>> print(c)
8
>>> quit()
lapegna@lapemax:~/PYTHON$
```

utile in fase di sviluppo o piccoli progetti

## Python?

Python **e' un linguaggio interpretato**
**due modalita' di esecuzione**

**2) modalita' script**

- viene mandato in esecuzione l'interprete fornendo, sulla linea di comando, il file con le istruzioni (con suffisso .py)

```
lapegna@lapemax:~/PYTHON$
lapegna@lapemax:~/PYTHON$
lapegna@lapemax:~/PYTHON$ ls
funz.py                                      prova.py      sommaF.o
libsumC.so                                   __pycache__   somma.py
libsumF.cpython-35m-x86_64-linux-gnu.so      sommaC.o
main.py                                      SOMMAFeC
lapegna@lapemax:~/PYTHON$
lapegna@lapemax:~/PYTHON$ cat somma.py
a = 3
b = 5
c = a+b
print (c)

lapegna@lapemax:~/PYTHON$
lapegna@lapemax:~/PYTHON$ python3 somma.py
8
lapegna@lapemax:~/PYTHON$
```

utile in fase di produzione e grandi progetti



## struttura di un programma e commenti

- In Python **non e' necessario specificare inizio e fine del main**
- main e' di default il programma che viene mandato in esecuzione dalla linea di comando dall'interprete
- Le istruzioni del programma, una di seguito all'altra, costituiscono il main

```
begin programma
....
....
# questo e' un commento
....


end programma
```

struttura di un programma in Pascal-like

- In Python **le linee di commento iniziano con #**

```
....
....
# questo e' un commento
....
```

Equivalente PYTHON



## Dichiarazione e definizione delle variabili

- In PYTHON **le variabili non hanno un tipo specifico**
  - non e' necessario dichiararle
  - e' possibile assegnare ad esse qualunque tipo di dato
- Il PYTHON **e' Case Sensitive** (fa differenza tra maiuscole e minuscole)
- L'operatore **= indica una operazione di assegnazione** di un valore ad una variabile
- Tradizionali regole per rappresentare i tipi di dati

```
var: a    integer
var: x    float
var: c    character


a = 5
x = 5.9
c = 'A'
```

le variabili in Pascal-like

```
a = 5
x = 5.9
c = 'A'
```

Equivalente PYTHON



## Lettura e stampa delle variabili

- In python le operazioni di lettura e stampa avvengono con le funzioni input( ) e print( )
- **input( )**
  - ritorna **un solo valore di tipo stringa**
  - il valore di ritorno **deve essere converetito** in valore numerico con funzioni del tipo int( ) o float( )
  - ha come **unico argomento una stringa** di caratteri (che viene visualizzata sullo schermo)
- **print( )**
  - **stampa l'elenco dei valori passati come argomenti**

```
begin area_rettangolo
var: b, h, a   float

print " dammi base"
read b
print "dammi altezza"
read h
a = b*h
print "area = ", a

end area_rettangolo
```

lettura e stampa in Pascal-like

```
b = float(input("dammi base"))

h = float(input("dammi altezza"))

a = b*h

print("area = ", a)
```

Equivalente PYTHON

## struttura di selezione

- Il costrutto if-else: permette di eseguire istruzioni o gruppi di istruzioni diverse a seconda del verificarsi di una condizione.
- la condizione termina con **:**
- i gruppi di istruzioni da eseguire in alternativa **sono indentati ! (carattere 'TAB' obbligatorio)**
- e' possibile **innestare piu' strutture** di selezione indentando in maniera opportuna le istruzioni
- Disponibile anche senza il ramo else

```
…
if (condizione) then
…
…
else
…
…
endif
```

struttura di selezione in Pascal-like

```
…
if condizione:
        …
        …
else:
        …
        …
…
```

Equivalente PYTHON



## Operatori aritmetici, logici e relazionali

- operatori aritmetici
  **+ - * /**
  **//** (div. intera) **%** (resto)
- operatori relazionali
  **> < == != >= <=**
- operatori logici
  **not and or**

```
…
if ( a > b   and   x != Y) then
…
…
else
…
…
endif
```

struttura di selezione in Pascal-like

```
…
if a > b   and   x != y:
        …
        …
else:
        …
        …
…
```

Equivalente PYTHON



## struttura di iterazione for

- e' necessario definire i valori dell'indice di iterazione con la funzione range

  **range(start, end, step)**

- itera da **start** a **end -1** con passo **step**
- dopo la funzione range sono necessari i **:**
- **istruzioni da ripetere indentate (obbligatorio) con il carattere TAB**
- piu' strutture innestate richiedono **indici distini** e **indentazioni differenti**
- campo **step opzionale** (default = 1)
- e' possibile iterare anche su dati di tipo misto esempio:

```
for k in (10, "gatto", 8.45):
        print (k)
```

```
…
for K = 1 to N step P
…
…
endfor
```

struttura di selezione in Pascal-like

```
…
for K in range(1,N,P):
        …
        …
…
```

Equivalente PYTHON



## Struttura di iterazione while

- Ripete piu' volte un gruppo di **istruzioni mentre una condizione e' vera**
- la condizione termina con **:**
- **istruzioni da ripetere indentate (obbligatorio)**
- Quanto la **condizione** risulta **falsa** la struttura di iterazione **termina**
- traduzione diretta del pascal-like
- non esiste l'equivalente della struttura repeat-until

```
…
while (condizione) do
…
…
endwhile
```

while-endwhile in Pascal-like

```
…
while condizione:
        …
        …
…
```

Equivalente PYTHON

## esempio 1: il massimo di N numeri

```
n = int( input("inserisci il numero di elementi") )

max = 0
for i in range(0, n) :
        num = int( input("inserisci un numero") )
        if num > max :
                max = num


print ("il massimo = ", max)
```

**osservare**:
1) conversione dei dati di input da stringa ad intero
2) la doppia indentazione delle strutture di controllo for e if



## esempio 2: il MCD

```
# calcolo del massimo comun divisore
a = int(input("dammi A"))
b = int(input("dammi B"))

while b != 0 :
        r = a%b
        a = b
        b = r

print ("il MCD = ", a)
```

**osservare**:
1) l'istruzione iniziale di commento
2) utilizzo della struttura while con indentazione
3) l'operazione "resto" ( % )



## Strutture dati in Python

- **stringa**:
  - sequenza di caratteri delimitati da doppio apice " (oppure singolo apice ' )
  - esempio:
    `S = "questa e' una stringa"`
- **tuple**:
  - sequenza di elementi non omogenei (numeri, stringhe, variabili,… ) , separati da virgole e opzionalmente racchiusi da parentesi tonde ( )
  - esempio:
    `T = ( 5, 6, "ciao", 8.45, x, y)` oppure `T = 5, 6, "ciao", 8.45, x, y`
- **lista**:
  - sequenza di elementi non omogenei (numeri, stringhe, variabili,… ), separati da virgole e racchiusi da parentesi quadre [ ]
  - esempio:
    `L = [ 5, 6, "ciao", 8.45, x, y ]`
- **array**
  - in Python non esiste il tipo array (ma ci torneremo dopo)



## stringa, tupla, lista

**caratteristica comune:**
e' possibile accedere alle singole componenti mediante un indice racchiuso tra parentesi quadre [ ]

Esempio:

```
S = "Gennaro Esposito"
print( S )
print( S[2] )
print ()

T = (1, 2, "ciao", 8.45)
print( T )
print( T[2] )
print ()

L = [1, 2, "ciao", 8.45]
print( L )
print( L[2] )
```

Risultato

```
Gennaro Esposito
n

(1, 2, 'ciao', 8.45)
ciao

[1, 2, 'ciao', 8.45]
ciao
```

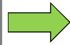
nota: l'indice parte sempre da 0

## stringa, tupla, lista

**Stringhe e Tuple non consentono la modifica di singoli elementi**

Esempi:

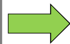
```
T = (1, 2, "ciao", 8.5)
print (T)
T[1] = 99
print (T)
```



Risultato

```
(1, 2, 'ciao', 8.5)
Traceback (most recent call last):
  File "prova.py", line 3, in <module>
    T[1] = 99
TypeError: 'tuple' object does not support item
assignment
```

```
S = "Gennaro Esposito"
print (S)
S[1] = 'K'
print (S)
```



```
Gennaro Esposito
Traceback (most recent call last):
  File "prova.py", line 3, in <module>
    S[1] = 'K'
TypeError: 'str' object does not support item
assignment
```

**Stringhe e Tuple sono oggetti immutabili**
**Le liste sono oggetti mutabili**



## operazioni comuni su stringa, tupla, lista

Le strutture dati in Python costituiscono un aspetto particolarmente importante,
e su di esse sono definite numerose operazioni

**operazioni comuni a tutte le strutture**

- = (assegnazione): copia una struttura su un'altra struttura (dello stesso tipo)
- in (appartenenza): ritorna True oppure False
- [ i : j ] (porzione): ritorna la porzione di struttura tra gli indici i e j-1
- + (concatenazione): unisce 2 o piu' strutture dati (dello stesso tipo)
- * (replicazione): replica i dati esistenti in una struttura dati

**funzione comune a tutte le strutture**

- **len( )** ritorna il numero di elementi della struttura (es.: **N = len(L)** )
- **type ( )** ritorna il tipo di struttura



## stringa, tupla, lista

```
S = "Gennaro Esposito"
R = S
print (R, "\n")

R = 'E' in S
print (R, "\n")

T = (5, 6, "ciao", 8.5)
R = T[2:4]
print (R, "\n")

x = (25, "hello", 5e-1)
R = T + x
print (R, "\n")

L = [5, 6, "ciao", 8.5]
R = 2 * L
print (R, "\n")
```

Risultato

```
"Gennaro Esposito"

True

('ciao', 8.5)

(5, 6, 'ciao', 8.5, 25, 'hello', 0.5)

[5, 6, 'ciao', 8.5, 5, 6, 'ciao', 8.5]
```



## oggetti e metodi

In Python, tutti i dati prendono il nome generico di

**OGGETTO**

Un **oggetto** puo' essere visto come un'area di memoria in cui sono conservati **valori**
e su cui sono definite le **operazioni** che e' possibile eseguire su di ssi

in Python esistono **oggetti predefiniti** (variabili, liste, stringhe, tuple ed altri)
e **oggetti definiti dall'utente**

Su ogni tipo di oggetto sono definite particolari operazioni chiamate

**METODI**

hanno la forma:

```
risultato = nome_oggetto.nome_metodo( eventuali argomenti)
```

## metodi sulle liste

Se **L** e' una lista:

**L.append(x)**
aggiunge l'elemento **x** alla fine della lista

**L.insert(i,x)**
inserisce l'elemento **x** prima dell' **i**-mo elemento

**L.remove(x)**
rimuove l'elemento **x** dalla lista

**L.pop(i)**
rimuove e ritorna l'elemento in posizione **i**. **L.pop()** rimuove e ritorna l'ultimo elemento

**L.clear()**
rimuove tutti gli elementi della lista

**L.sort()**
ordina la lista (in place)

**L.reverse()**
dispone gli elementi in ordine inverso (in place)



## esempio 1: usare una lista come una pila

```
L = [1, 2, 3, 4, 5]
print(L)

L.append(6)
L.append(7)
print(L)

R = L.pop()
print (R)

R = L.pop()
print(R)

print(L)

L.reverse()
print(L)

L.clear()
print()
```

```
[1, 2, 3, 4, 5]

[1, 2, 3, 4, 5, 6, 7]

7

6

[1, 2, 3, 4, 5]

[5, 4, 3, 2, 1]

[]
```



## esempio 2: usare una lista come una coda

```
L = [1, 2, 3, 4, 5]
print(L)

L.insert(0,6)
L.insert(0,7)
print(L)

R = L.pop()
print (R)

R = L.pop()
print(R)

print(L)

L.sort()
print(L)

L.clear()
print()
```

```
[1, 2, 3, 4, 5]

[7, 6, 1, 2, 3, 4, 5]

5

4

[7, 6, 1, 2, 3]

[1, 2, 3, 6, 7]

[]
```



## metodi sulle tuple

Se **T** e' una tupla:

**T.index(x)**
ritorna la posizione del primo **x** in **T** (es.: **pos = T.index( x )** ). Errore se argomento assente

**T.count(x)**
conta quante volte **x** e' presente in **T** (es.: **cnt = T.count( x )** )

```
T = 'a', 'b', 10, 11, 12, 'a'
print(T)

P = T.index('a')
C = T.count('a')

print ('P=', P, 'C=', C)
```

```
'a', 'b', 10, 11, 12, 'a'

P=0

C=2
```

## metodi sulle stringhe

decine di metodi !

Se `S` e' una stringa:

`S.upper()` `S.lower()`
trasformano la stringa in caratteri tutti maiuscoli/minuscoli

`S.alpha()` `S.alnum()` `S.isnumeric()` `S.isspace()`
ritornano True/False a secondo del tipo di stringa

`S.split('c')`
crea una lista dividendo la stringa in base al separatore `'c'`

`S.replace('old','new')`
crea una nuova stringa sostituendo un gruppo di caratteri (`old`) con un altro gruppo (`new`)

tanti altri…



## file

I file sono **collezioni di dati**, memorizzati nel file system del sistema operativo, a cui si puo' accedere attraverso un **nome.**

`F = open('nomefile', 'modo')`

e' una funzione che rende disponibile il file al programma e ritorna un oggetto, attraverso il quale si fa successivamente riferimento al file

l'apertura e' sempre la prima azione da compiere per utilizzare un file

`F` e' il nome dell'oggetto ritornato

`'nomefile'` e' il nome del file nel file system

`'modo'` e' la modalita' di apertura

- 'r' e' possibile solo leggere dal file
- 'w' e' possibile solo scrivere sul file. Elimina dati in eventuale file gia' esistente
- 'a' e' possibile aggiungere elementi al file
- 'r+' e' possibile leggere e scrivere sul file



## metodi sui file

Se `F` e' l'oggetto ritornato dalla funzioine `open`:

`F.read(N)`
funzione che legge `N` caratteri e li ritorna in una stringa. Senza argomento ritorna l'intero file (es.: `S = F.read()` )

`F.readline()`
funzione che legge una riga del file e la ritorna in una stringa (es.: `S = F.readline()` )

`F.write(S)`
scrive la stringa `S` nel file. Sovrascrive eventuali caratteri presenti !!

`F.seek(N)`
si posiziona sull'`N`-mo carattere del file (es.: `F.seek(0)` si posiziona all'inizio del file)

`F.close()`
chiude il file. Impossibile eseguire altre operazioni su di esso



## esempio

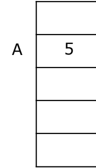
```
F = open("testo", "r+")
S = F.read()
print (S)

S2 = "perche' la retta via era smarrita"
F.write(S2)

F.seek(0)
S = F.read()
print (S)

F.close()
```

```
nel mezzo del cammin di nostra vita
mi ritrovai in una selva oscura

nel mezzo del cammin di nostra vita
mi ritrovai in una selva oscura
perche' la retta via era smarrita
```

## Il modello delle variabili in Python

Nei linguaggi come Fortran e C c'e' una stretta relazione tra
- nome della variabile
- valore della variabile

In memoria il nome della variabile e' il nome della
locazione di memoria
che contiene il valore della variabile stessa




Ad esempio l'istruzione in linguaggio C

```
A = 5;
```

assegna alla locazione di memoria chiamata A il valore 5

- In Python questi due concetti sono separati
- Le variabili sono etichette che possono essere trasferite dinamicamente a valori anche di tipo diverso

**modello di tipizzazione dinamica**



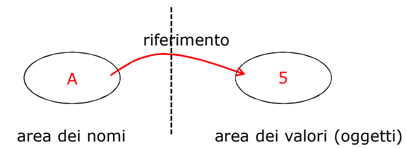
## modello di tipizzazione dinamica

Nel modello di memoria in Python,
l'area dei nomi delle variabili e' logicamente separata dall'area dei valori delle variabili.

Ad esempio l'istruzione Python: `A = 5`

consiste dei seguenti 3 passi

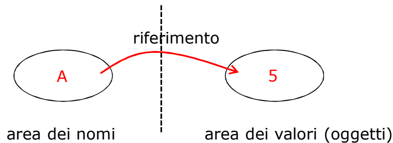
- viene creato un oggetto (una struttura in memoria) che rappresenta il valore 5
- viene creata (se non esiste gia') una variabile di nome A
- viene creato un riferimento (un puntatore) tra la variabile A e il valore 5

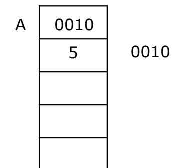





## modello di tipizzazione dinamica

In pratica la variabile A contiene l'indirizzo della locazione di memoria che contiene il valore 5




modello logico

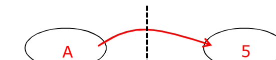
modello fisico

- il tipo di dato e' quindi una caratteristica dell'oggetto e non della variabile
- la variabile puo' riferirsi a oggetti di tipo differente nel corso dell'esecuzione
- le variabili puntano sempre ad oggetti, e mai ad altre variabili
- oggetti come liste e/o tuple possono puntare ad altri oggetti (ad esempio agli oggetti che contengono)

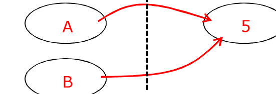


## modello di tipizzazione dinamica: esempio1

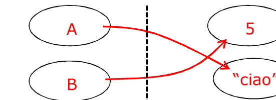
**A = 5**
viene creato in memoria l'oggetto 5, che viene collegato alla variabile A




**B = A**
viene copiato il contenuto di A (cioe' l'indirizzo di 5) in B. Ora entrambe le variabili si riferiscono allo stesso oggetto



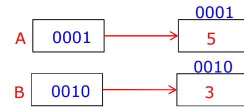

**A = "ciao"**
viene creato il nuovo oggetto "ciao", che viene collegato alla variabile A



## un esempio interessante

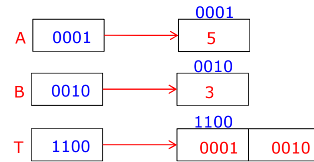
**A = 5**
**B = 3**

vengono creati due oggetti (valori), due nomi di variabili A e B, e i riferimenti ai rispettivi oggetti




**T = A , B**

viene creata un nuovo oggetto tupla con i riferimenti di A e B, un nuovo nome T, e il relativo riferimento

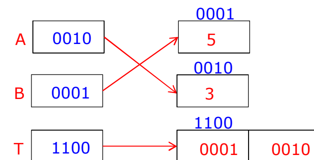



**B , A = T**

vengono copiati i valori della tupla T in B e A

**SCAMBIO DI VARIABILI A E B**

equivalentemente
**B , A = A , B**



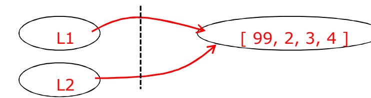



## riferimenti condivisi su oggetti mutabili

```
L1 = [1, 2, 3, 4]
L2 = L1
L1[0] = 99
print ("L1 = ", L1)
print ("L2 = ", L2)
```

```
L1 = [99, 2, 3, 4]

L2 = [99, 2, 3, 4]
```

L'istruzione **L2 = L1** non crea un nuovo oggetto **L2** uguale a **L1**, ma solo un collegamento condiviso al medesimo oggetto




Una **modifica** sui dati di L1 **si ripercuote** anche su L2 !

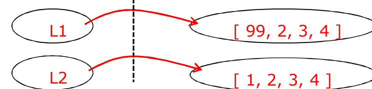


## riferimenti condivisi su oggetti mutabili

se occorre e' possibile fare una copia fisica di **L1** con il metodo **copy()**

```
L1 = [1, 2, 3, 4]
L2 = L1.copy()
L1[0] = 99
print ("L1 = ", L1)
print ("L2 = ", L2)
```

```
L1 = [99, 2, 3, 4]

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




Una **modifica** sui dati di L1 **non si ripercuote** su L2 !



## garbage collection

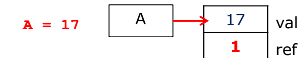
fisicamente, un oggetto in memoria e' una struttura dati che contiene
- valore che si vuole rappresentare (val)
- numero di variabili che fanno riferimento a tale valore (ref)

Quando ref == 0
(cioe' non e' referenziato da nessuna variabile)
l'oggetto diventa irraggiungibile e viene rimosso, rendendo la memoria di nuovo disponibile per nuovi oggetti

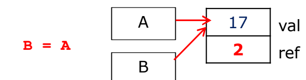
**garbage collection**

**Esempio**

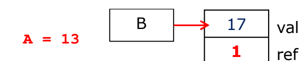
**A = 17**




**B = A**




**A = 13**




**B = 13**

## I moduli

- possono ritornare al programma chiamante piu' di un valore sotto forma di tupla
- Gli argomenti della function sono tutti argomenti di input e sono passati per valore
- le funzioni sono inserite in un file con estensione .py (chiamato modulo) che deve essere importato nel file contenente il programma chiamante (nell'esempio **funzioni.py** )
- un insieme di una o piu' funzioni costituisce un modulo
- un modulo e' a tutti gli effetti un oggetto (come liste, stringhe,..) per cui le funzioni in esso contenute sono i metodi associati al modulo

sintassi **modulo.metodo (…)**

```
begin main

var: a, b, sum, dif integer

read a, b

SeD(in: a, b; out: sum, dif)

print sum, dif

end main
```

chiamata di una procedura in Pascal-like

```
import funzioni

read(a, b)

sum, dif = funzioni.SeD(a, b)

print (sum, dif)
```

Equivalente PYTHON (file main.py)



## I moduli

- la function agisce su proprie variabili locali
- la testata della function contiene i valori di input, mentre l'istruzione return, ritorna i valori di output al programma chiamante sotto forma di tupla
- Per gli argomenti di input/output e' possibile utilizzare nomi diversi nelle function e nel programma chiamante
- le istruzioni del corpo della function sono indentate (obbligatorio) con il carattere TAB

```
procedure SeD(in x, y; out z, w)

var: x, y, z, w integer

z = x + y

w = x - y

end main

end procedure
```

chiamata di una procedura in Pascal-like

```
def SeD (x, y)
        z = x+ y
        w = x - y
        return z, w
```

Equivalente PYTHON (file funzioni.py)



## la libreria standard

Python ha un set di istruzioni molto ristretto, ma dispone di una libreria che ne aumenta le funzionalita composta da centinaia di moduli

**libreria standard**

- accesso al sistema operativo
- sviluppo di interfacce grafiche
- web programming
- esecuzione multithreading
- …
- matematica
  - **math** — funzioni matematiche
  - **cmath** — numeri complessi
  - **decimal** — aritmetica f.p.
  - **random** — generatore numeri casuali
  - **statistics** — funzioni statistiche

e' necessario importare i moduli

```
import math

pi = math.pi
R = math.sin(pi)
print (pi, R)

e = math.e
R = math.log(e)
print(e, R)
```

esempio di uso del modulo math

```
3.141592653589793 1.2246467991473532e-16

2.718281828459045 1.0
```

risultato



# FINE

Questo lucido, ed i successivi, non fanno parte della lezione

## C contro Python

due programmi equivalenti

```c
int main(){
int a, s, i, h;
a = 1;  s = 0;

for (i = 0; i < 1000; i++){
for (j = 0; j < 1000; j++){

         s = s + a;
         a = -a;
}}

}
```

prova.c

```python
s = 0
a = 1

for i in range(0, 1000):
        for j in range(0, 1000):

                s = s + a
                a = -a
```

prova.py

```
lapegna@lapemax:~/PYTHON$ cc prova.c
lapegna@lapemax:~/PYTHON$ time ./a.out

real    0m0.003s
user    0m0.000s
sys     0m0.000s
lapegna@lapemax:~/PYTHON$
lapegna@lapemax:~/PYTHON$
lapegna@lapemax:~/PYTHON$ time python3 prova.py

real    0m0.204s
user    0m0.204s
sys     0m0.000s
lapegna@lapemax:~/PYTHON$
```



## C contro Python

- Un codice Python puo' risultare da **decine a centinaia di volte piu' lento** di un equivalente codice C o Fortran
- Cio' e' dovuto alla sua natura **interpretativa**
- Python **non e' utilizzato direttamente per implementare metodi** per il calcolo tecnico/scientifico
- I punti di forza di Python sono altri
  - **set ristretto di istruzioni (facilita' di utilizzo e di apprendimento)**
  - disponibilita' di centinaia di moduli che ne **estendono la funzionalita'** nei campi piu' disparati (data science, machine learning, web programming, GUI development, calcolo scientifico, …)
  - **facile integrazione** con altri strumenti e linguaggi di programmazione che permettono di raggiungere elevate prestazioni

**Python as a glue language**



## Numpy

**Numpy** (numerical python) e' il principale modulo che estende le funzionalita' di Python con strutture dati e funzioni per il **calcolo tecnico scientifico**

- array multidimensionali
- funzioni matematiche di base

principali vantaggi:

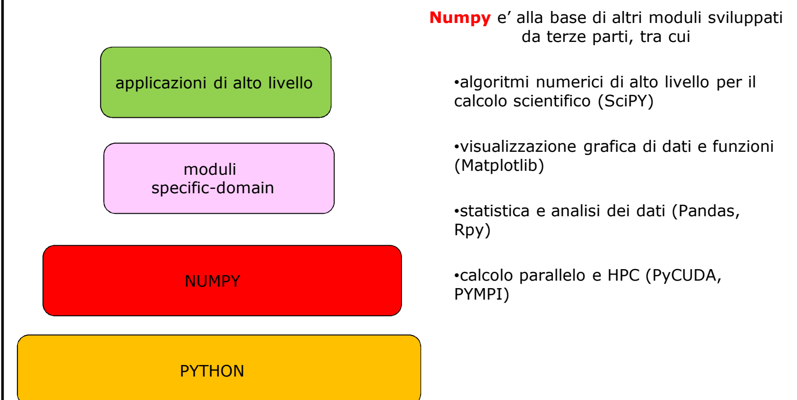
- gli array sono gestiti in maniera piu' efficiente rispetto alle liste (allocazione contigua in memoria e accesso attraverso indici, come in C e Fortran)
- molti metodi sono scritti in C e/o Fortran con notevole miglioramento delle prestazioni
- operazioni di I/O (anche su file) sono piu' efficienti

Numpy non e' utile per applicazioni al di fuori del contesto tecnico scientifico (web programming, GUI development,…)

tutta la documentazione disponibile su
www.numpy.org



## software stack per il calcolo scientifico in Python




**Numpy** e' alla base di altri moduli sviluppati da terze parti, tra cui

- algoritmi numerici di alto livello per il calcolo scientifico (SciPY)
- visualizzazione grafica di dati e funzioni (Matplotlib)
- statistica e analisi dei dati (Pandas, Rpy)
- calcolo parallelo e HPC (PyCUDA, PYMPI)

## importare il modulo Numpy

Prima di ogni utilizzo di Numpy,
e' **necessario importare il relativo modulo** mediante il comando **`import`**

```
import numpy
```

importa il modulo Numpy.
Di seguito si fara' riferimento alla sintassi del tipo

**`numpy.metodo ( )`**

oppure

```
import numpy as np
```

importa il modulo Numpy e lo rinomina come **`np`** (standard di fatto in molte applicazioni).
Di seguito si fara' riferimento alla sintassi del tipo

di seguito useremo questa notazione **`np.metodo ( )`**

## gli array

La principale caratteristica di Numpy e' quella di definire **nuovi oggetti di tipo array**

Gli array sono strutture dati **omogenee**,
di **dimensione fissata** e allocate in locazioni di memoria **contigue**

il metodo **`array`** crea un array a partire da una lista o da una tupla

```
import numpy as np
A = np.array([1, 2, 3, 4],  dtype = np.float32)
```

crea un array A a partire da una lista di valori e li trasforma in float

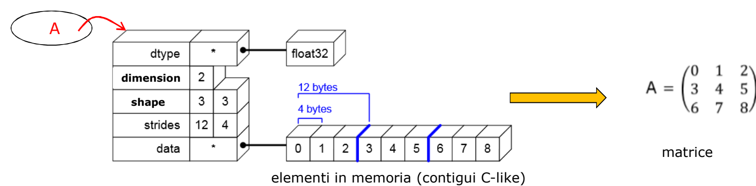
```
import numpy as np
B = np.array([[1, 2] , [3, 4], [5, 6]],  dtype = np.float32,  order = 'F')
```

crea un array 2D (matrice) B da una lista (di liste), li trasforma in float e il memorizza Fortran-like (column order)

gli argomenti dtype e order sono opzionali, ma fortemente consigliati

## gli array

Il nome dell'array punta ad un oggetto che lo descrive in memoria




elementi in memoria (contigui C-like)

definizioni

- dimensione: la dimensionalita' dell'array (numero di indici)
- forma (shape): una tupla contenente il numero di elementi in ogni dimensione (righe e colonne).
- stride : una tupla contenente la distanza (in byte) degli elementi lungo le righe e le colonne (analogo della leading dimension)

## dimensione, forma e stride

Tali caratteristiche sono ritornati dai metodi **`ndim`**, **`shape`** e **`strides`**

```
B = np.array([[1, 2] , [3, 4], [5, 6]], order = "C")
D = B.ndim
F = B.shape
S = B.strides

print (D, "\n", F, "\n", S)
```

```
2
(3, 2)
(16, 8)
```

S[0] = 16 (dist. tra righe)

| B = | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|

S[1] = 8 (dist. tra colonne)

ordinamento "row-major" (C-like)

```
B = np.array([[1, 2] , [3, 4], [5, 6]], order = "F")
D = B.ndim
F = B.shape
S = B.strides

print (D, "\n", F, "\n", S)
```

```
2
(3, 2)
(8, 24)
```

S[0] = 8 (dist. tra righe)

| B = | 1 | 3 | 5 | 2 | 4 | 6 |
|---|---|---|---|---|---|---|

S[1] = 24 (dist. tra colonne)

ordinamento "column-major" (Fortran-like)

## indicizzazione e partizionamento

come per liste e tuple e' possibile

- accedere agli elementi dell'array attraverso indici in parentesi quadre []
- utilizzare il partizionamento con la notazione **[ start : end : step ]**

```
B = np.array([ [1, 2, 3] , [4, 5, 6], [7, 8, 9] ])

print ("B[1,1] = ", B[1,1])

D = B[0:3:2 , 0:3:2]       # [ start : end : step ]
print ("D = ", D )
```

```
B[1,1] = 5

D = [[1, 3]
     [7, 9]]
```

N.B. gli indici iniziano da 0



## osservazione

**gli array sono oggetti mutabili !**

```
B = np.array([ [1, 2, 3] , [4, 5, 6], [7, 8, 9] ])

D = B[0:3:2 , 0:3:2]       # [ start : end : step ]
```

oppure

```
B = np.array([ [1, 2, 3] , [4, 5, 6], [7, 8, 9] ])

D = B
```

creano solo un **nuovo riferimento** tra **l'oggetto array** preesistente e il **nuovo nome D**

eventuali **modifiche** agli elementi di D si **ripercuotono** anche su B e viceversa
(analogamente a quanto accade per le liste e' possibile utilizzare il metodo **copy()** per creare una copia fisica dell'array)



## passaggio di una matrice ad una function

Esempio: Trasposta di una matrice

```
import numpy as np
import funzioni

N = 3
B = np.array([ [1, 2, 3] , [4, 5, 6], [7, 8, 9] ])

B = funzioni.trasp(B,N)
print(B)
```

file funzioni.py

```
def trasp(Y, N):
        for i in range(0,N):
                for j in range(0,i):
                        t = Y[i,j]
                        Y[i,j] = Y[j,i]
                        Y[j,i] = t
        return Y
```



## matrici particolari

opportuni metodi di Numpy creano matrici particolari predefinite

- il metodo **zeros(shape)** crea una matrice nulla di forma shape (numero di righe e colonne)
  es.: **A = np.zeros(3,4)**
- il metodo **ones(shape)** crea una matrice di forma shape di elementi uguali a 1
  es.: **A = np.ones(3,4)**
- il metodo **empty(shape)** crea una matrice di forma shape non inizializzata
  es.: **A = np.empty(3,4)**
- il metodo **identity(N)** crea una matrice identita' di ordine N
  es.: **A = np.identity(4)**

## funzioni arange e linspace

**arange**

genera un array mediante l'espressione

`C = arange (a,b,h)`

i cui elementi sono `a, a+h, a+2h, … a+mh`
(con `m` piu' grande intero tale che **a+mh < b**)

esempio:

`C = np.arange(0, 1, 0.1)`

genera l'array

`C = [0, 0.1, 0.2, …, 0.9]`

N.B. il numero di elementi e' (b-1)/h

**linespace**

genera un array mediante l'espressione

`C = linspace (a,b,n)`

composto da n elementi compresi tra `a` e `b`
(inclusi)

esempi:

`C = np.linspace(0, 1, 11)`

genera l'array

`C = [0, 0.1, 0.2, …, 1]`

N.B. la distanza tra gli elementi e' (b-a)/(n-1)



## operazioni "elemento per elemento" su array

Dati due array 2-dimensionali `A` e `B`

- somma "elemento per elemento" di array (1D e 2D)
  es.: `C = A + B`
- prodotto "elemento per elemento" di array (1D e 2D)
  es.: `C = A * B`
- incremento (1D e 2D)
  es.: `A = A + 1`
- generiche espressioni "elemento per elemento" di array (1D e 2D)
  es.: `C = 2**(A+1)-B`
- generiche espressioni aritmetiche e logiche "elemento per elemento" di array (1D e 2D)
  es.: `C = 2**(A+1)-B`
  es.: `C = A > B`

tali operazioni sono circa 10 volte piu' efficienti rispetto a quelle ottenute esplicitando le strutture di iterazione in Python



## altri metodi su su array

Dati due array 2-dimensionali `A` e `B`

- prodotto "righe per colonne" di matrici (colonne di A = righe di B)
  es.: `C = np.matmult(A, B)`
- trasposta di matrice
  es.: `C = np.transpose(A)`
- altri metodi applicabili ad array 1D e 2D
  - `C = np.sort(A)`
  - `C = np.max(A)`   `C = np.min(A)`
  - `C = np.sum(A)`   `C = np.prod(A)`
  - `C = np.mean(A)`   `C = np.var(A)`   `C = np.std(A)`
  - `C = np.sin(A)`   `C = np.sqrt(A)`   `C = np.log(A)`

  - tali metodi, quando sono utilizzati su array 2D, operano riga per riga (ad es. sort)

consultare http://numpy.org



## sottomodulo linalg

Numpy possiede vari sottomoduli per specifici ambiti applicativi, tra cui il sottomodulo linalg per le operazioni di algebra lineare.

E' basato su **BLAS** e **LAPACK**

```
import numpy.linalg
```

importa il modulo numpy.linalg
Di seguito si fara' riferimento alla sintassi del tipo

`numpy.linalg.metodo ( )`

oppure

```
import numpy.linalg as lin
```

importa il modulo numpy.linalg e lo rinomina come `lin`
Di seguito si fara' riferimento alla sintassi del tipo

`lin.metodo ( )`

## sottomodulo linalg

Esempi. Data una matrice **A**

- determinante e inversa di A

```
D = np.linalg.det(A)        I = np.linalg.inv(A)
```

- autovalori e autovettori di A

```
vals, vecs = np.linalg.eig(A)
```

- fattorizzazioni e risoluzione di sistemi
  - `c = np.linalg.cholesky(A)`
  - `c = np.linalg.solve(A,b)`
  - `q, r = np.linalg.qr(A)`

**fare riferimento a www.numpy.org per altri metodi**

## le Universal Function (ufunc)

Numphy possiede un ampio insieme (quasi 100) di funzioni capaci di operare su tutti gli elementi di un array

**Le UFUNC sono dei wrapper a funzioni scritte tipicamente in C o Fortran**

- operazioni matematiche di base
- funzioni trigonometriche e logaritmiche
- manipolazione di bit
- operazioni su dati floating point

Esempio: se A e' un array

`C = np.cos(A)`    (calcola il coseno di tutti gli elementi di A)

N.B.
- le Universal Function sono particolarmente efficienti per lavorare su array, ma possono essere usate anche per singole variabili.
- Su singole variabili le corrispondenti funzioni del modulo math sono pero' piu' efficienti

## Scipy

**Scipy** e' una collezione di moduli di alto livello per il calcolo scientifico, che utilizza numpy

I metodi di Scipy sono scritti in linguaggi compilati (ad es. C e/o Fortran), appartengono a librerie consolidate di software matematico (ad es. LAPACK, FFTPACK, QUADPACK, ODEPACK,…) e sono particolarmente efficienti per il calcolo scientifico

Esempi di sottomoduli di Scipy

- **cluster** algoritmi di Clustering (es. K-means)
- **constants** costanti fisiche e matematiche
- **fftpack** Fast Fourier Transform
- **integrate** calcolo integrali e ODE
- **interpolate** interpolazione, approssimazione e splines
- **io** Input and Output
- **linalg** routine di algebra lineare
- **ndimage** N-dimensional image processing
- **optimize** ottimizzazione
- **signal** signal processing
- **sparse** matrici sparse
- **stats** funzioni per la statistica

## Scipy (esempio)

```
import scipy.fftpack
```

Importa il modulo scipy.fftpack
Di seguito si fara' riferimento alla sintassi del tipo

**scipy.fftpack.metodo ( )**

oppure

```
import scipy.fftpack as fpack
```

importa il modulo nscipy.fftpack e lo rinomina come **fpack**
Di seguito si fara' riferimento alla sintassi del tipo

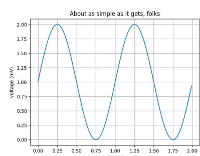
**fpack.metodo ( )**

fare riferimento a www.scipy.org per tutti i sottomoduli e i relativi metodi

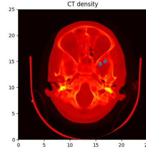
# matplotlib

**matplotlib** e' una collezione di moduli per la rappresentazione grafica di dati 2D e 3D, ed e' basata su numpy

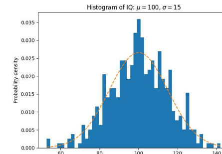
matplotlib contiene decine di sottomoduli. (consultare www.matplotlib.org)



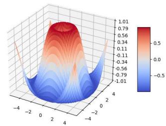
funzioni 2D

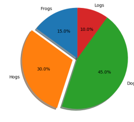


immagini

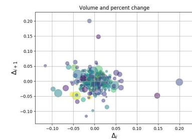


istogrammi


funzioni 3D



grafici a torte

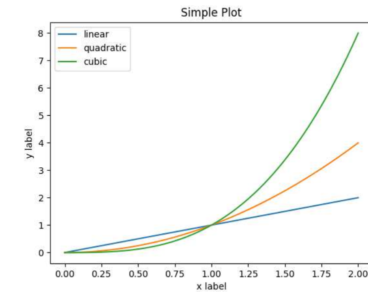


scattered plot

# matplotlib (esempio)

```
import matplotlib.pyplot as plt

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

plt.show()
```

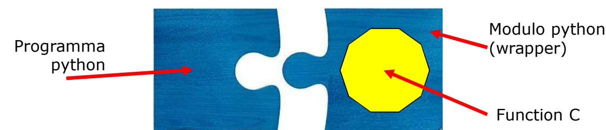
Sottomodulo pyplot



# Interazione con altri linguaggi

- Python dispone di numerosi strumenti per interagire con moduli e funzioni gia' disponibili, scritti in altri linguaggi
- L'idea e' quella di presentare una programma scritto in altro linguaggio (ad esempio una function C) come un modulo Python

Il modulo Python risultante e' noto come wrapper (involucro)




- Uno degli strumenti piu' utilizzati e semplici e' la libreria **`ctypes`**
- (https://docs.python.org/3/library/ctypes.html)



# Wrapper con la libreria ctypes

- La libreria (modulo) **`ctypes`** fornisce gli strumenti per convertire tipi di dato python in tipi di dato compatibili con il C

PYTHON ⟺ ctypes ⟺ C

- permette inoltre di chiamare le funzioni C come librerie dinamiche (shared library)
- Le librerie dinamiche non sono inserite all'interno del file eseguibile (ad es. a.out) al momento della compilazione, ma vengono collegate ad esso solo al momento dell'effettivo utilizzo
  - Risparmio di memoria: piu' processi in esecuzione possono usare contemporaneamente la stessa libreria dinamica presente in memoria
  - Facilita' di manutenzione del progetto: in caso di modifica della libreria dinamica, tutto il resto del progetto non deve essere modificato ne' ricompilato
- Per scrivere un wrapper con ctypes e' necessario
  - Importare nel wrapper la libreria ctypes
  - Importare nel wrapper la libreria dinamica che contiene le function scritte in C
  - Specificare e preparare gli argomenti da passare alla function C
  - Chiamare la function C come un modulo python

## Esempio 1

- Sia la seguente funzione C

```
float somma1(float a, float b){
        float c;
        c = a + b;
        return c;
}
```

Vogliamo richiamare tale funzione da un programma python

- La funzione C deve essere compilata e inserita in una libreria dinamica

```
cc -fPIC  -shared  functions.c -o functions.so
```

- Tale comando traduce in linguaggio macchina il contenuto del file **functions.c** (contenente la function **somma1**) e crea una libreria dinamica (shared) **functions.so**
- Eventualmente e' possibile inserire piu' file all'interno della stessa libreria dinamica



## Il wrapper

```
def somma1_w(a, b):

        import ctypes
        functions = ctypes.CDLL(“./functions.so”)

        functions.somma1.argtypes = [ctypes.c_float, ctypes.c_float ]
        functions.somma1.restype = ctypes.c_float
        c = functions.somma1(a, b)

        return c
```

- Il wrapper e' un comune modulo python (nell'esempio **somma1_w**), che al suo interno richiama opportunamente la function C
- Dopo aver importato la libreria **ctypes**, il metodo **CDLL** importa nel wrapper la libreria **functions.so** come un oggetto (nell'esempio **functions**)
- **argtypes** specifica il tipo degli argomenti di **somma1**
- **restype** specifica il tipo del valore di ritorno di **somma1** (**None** per funzioni void)
- Successivamente si richiama la function **somma1** come metodo dell'oggetto **functions**



## Il programma principale

```
import wrapper

A = 3.2
B = 6.3

print(“ A e B = “, A, B)

C = wrapper.somma1_w (A, B)

print(“ la somma = “, C)
```

- Semplicemente
  - Importa il file con il wrapper (**wrapper.py**)
  - Richiama il metodo **somma1_w**



## Esempio 2

- Sia la seguente funzione C

```
void somma2(float a, float b, float *c){

        *c = a + b;

}
```

Vogliamo richiamare tale funzione da un programma python

- In questo caso, la function C e' di tipo **void**, e come terzo argomento (il risultato c) si aspetta un **float ***
- Nel wrapper e' necessario passare il terzo argomento come puntatore

## Il wrapper

```
def somma2_w(a, b):

        import ctypes
        functions = ctypes.CDLL("./functions.so")
        c = ctypes.c_float()

        functions.somma2.argtypes = [ctypes.c_float, ctypes.c_float,
                                      ctypes.POINTER(ctypes.c_float) ]
        functions.somma2.restype = None

        c = functions.somma2(a, b, ctypes.byref(c) )

        return    c.value
```

- In questo caso e' necessario
  - Creare un oggetto **c** di tipo ctype, compatibile cioe' con il float C
  - Specificare che il terzo argomento di **somma2** e' un puntatore ad un float
  - Specificare che **somma2** non ritorna valori
  - Richiamare la function e passare il terzo argomento per indirizzo
  - Ritornare al programma principale il valore dell'oggetto c



## Esempio 3

- Sia la seguente funzione C

```
float  massimo(float *a, int n){
        int i;
        float mas;

        mas = a[0];
        for (i=1; i<n; i++){
                if (a[i] > mas) ) mas = a[i]
        }

        return mas;

}
```

Vogliamo richiamare tale funzione da un programma python

- In questo caso, il primo argomento della function e' un **array**
- Necessario convertire un array python (cioe' un oggetto numpy) in un array C (cioe' un float*)



## Il wrapper

```
def massimo_w(x, n):

        import ctypes
        functions = ctypes.CDLL("./functions.so")

        x_p = x.ctypes.data_as(ctypes.POINTER(ctypes.c_float))

        functions.massimo.argtypes = [ ctypes.POINTER(ctypes.c_float),
                                                         ctypes.c_int ]
        functions.massimo.restype = ctypes.c_float()

        mas = functions.massimo(x_p, n)

        return   mas
```

- In questo caso e' necessario
  - Creare un oggetto **x_p** convertendo l'array python **x** in un float* del C, mediante **ctypes.data_as**
  - Specificare che il primo argomento di **massimo** e' un puntatore ad un float



# FINE

Questo lucido, ed i successivi, non fanno parte della lezione