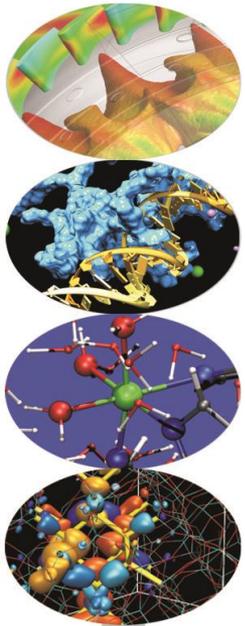
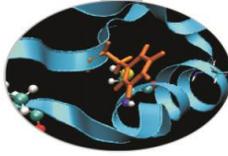


Funzioni



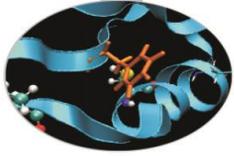


Funzioni: Definizione

Una funzione è un blocco organizzato di codice che viene utilizzato per eseguire un preciso task.

L'utilizzo di funzioni garantisce maggiore modularità al codice e una maggiore riutilizzabilità dello stesso.

Python dispone di funzioni *built-in*, p.e. *print()*, *help()*, *dir()* etc, ma è ovviamente possibile definire nuove funzioni: *user-defined-function*.



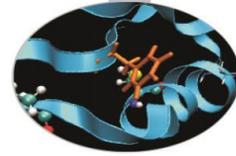
Funzioni: Definizione

La definizione di una funzione in python ha la seguente sintassi:

```
def NameFunction(arg1,arg2,...,argN):
```

La *keyword def* è seguita dal nome della funzione e dalla lista dei suoi argomenti.

Al prototipo della funzione segue il corpo della definizione.



Funzioni: Definizione

E' possibile aggiungere una stringa di documentazione antecedente il corpo della funzione per spiegare l'utilizzo della funzione e i parametri:
docstrings.

Esempio:

```
def stampa(stringa): ← Prototipo della funzione  
    """La funzione stampa stampa a video la stringa stringa """ ← Docstring  
    print stringa ← Corpo della funzione
```

```
help(stampa) ← Visualizza la descrizione  
stampa('Ciao') ← Chiamata della funzione
```

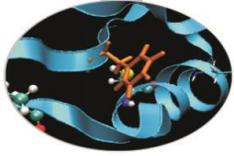
Output

Help on function stampa in module `__main__`:

```
stampa(stringa)
```

La funzione *stampa* stampa a video la stringa *stringa*.

```
Ciao
```



Funzioni: def statement

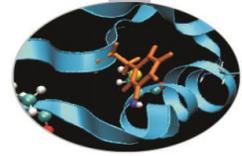
Lo statement *def* è uno statement esecutivo a tutti gli effetti.

Quando viene eseguito crea una nuova *function object* e le assegna un nome.

In Python è perfettamente legale dichiarare una funzione incapsulata in altri statements.

```
if test1:
    def f1():
        ...
else:
    def f2():
        ...

def void():
    def void2()
        ...
    return something
```



Funzioni: def statement

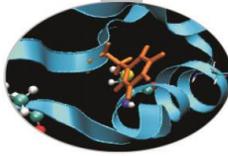
```
if test1:  
    def f1():  
        ...  
else:  
    def f2():  
        ...
```

La funzione *f1* non viene creata finchè lo statement *def* non viene raggiunto ed eseguito: in Python tutto viene valutato a *runtime* al contrario dei linguaggi compilati.

Esempio: `dyn_def.py`

Nota Nell'esecuzione dell'import di un modulo tutte le *function object* in esso definite vengono create.

Argomenti di una Funzione



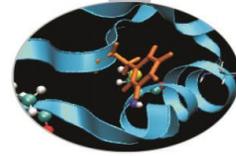
Passaggio di argomenti

In Python è importante distinguere tra oggetti *mutabili* e *immutabili* per capire il passaggio di argomenti ad una funzione.

Gli oggetti *immutabili* (stringhe, tuple, numeri) vengono passati per *valore*: variazioni di questi parametri non hanno effetti sul programma chiamante.

Gli oggetti *mutabili* (liste, dictionary) vengono passati per *referenza*: variazioni di questi parametri si propagano al programma chiamante.

Inoltre le variabili definite nel corpo di una funzione hanno *scope* locale a quella funzione.



Argomenti di una Funzione

Esempio

```
lista=[1,2,3,3,4,4,7,8]
```

```
def remove_duplicate(vect):
```

```
    print "Vect inside function, before: ", vect
```

```
    for i in vect:
```

```
        c=vect.count(i)
```

```
        if c>1:
```

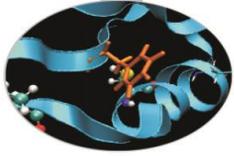
```
            vect.remove(i)
```

```
    print "Vect inside function after: ", vect
```

```
remove_duplicate(lista)
```

```
print "Vect outside function ",lista
```

Argomenti di una Funzione

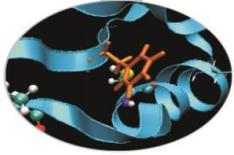


Output

Vect inside function, before: [1,2,3,3,4,4,7,8]

Vect inside function after: [1,2,3,4,7,8]

Vect outside function: [1,2,3,4,7,8]



Argomenti di una Funzione

Esempio2

```
lista=[1,2,3,3,4,4,7,8]
```

```
def change_vect(vect):
```

```
    print "Vect inside function, before: ", vect
```

```
    vect[2]=-99; vect=[4,5,6,6,7]
```

```
    print "Vect inside function, after: ", vect
```

```
change_vect(lista)
```

```
print "Vect outside function, after:", lista
```

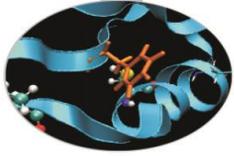
Output

```
Vect inside function, before: [1,2,3,3,4,4,7,8]
```

```
Vect inside function after: [4,5,6,6,7]
```

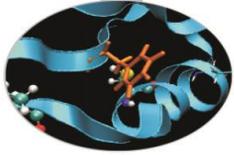
```
Vect outside function: [1,2,-99,3,4,4,7,8]
```

Argomenti di una Funzione



Argomenti di una funzione

- Il numero di parametri (*required parameters*) passati ad una funzione deve essere conforme al prototipo della funzione.
- E' possibile definire degli argomenti di *default* per una funzione, associando al nome del parametro un valore di default.
- I parametri di default seguono quelli obbligatori. Specificando dei parametri di default la chiamata a funzione può essere eseguita specificando meno parametri di quelli presenti nel prototipo.



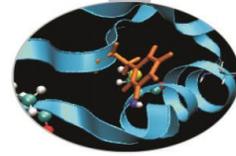
Argomenti di una Funzione

E' possibile chiamare una funzione utilizzando dei *keyword arguments* della forma *keyword=value*.

L'utilizzo dei keyword arguments permette di specificare i parametri senza seguire l'ordine posizionale.

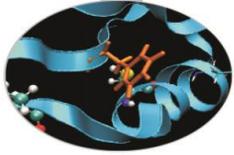
Esempio ordina.py

```
def insertion_sort(a):  
    for i in range(1,len(a)):  
        val=a[i]  
        j=i-1  
        while (j>=0 and a[j]>val):  
            a[j+1]=a[j]  
            j=j-1  
        a[j+1]=val  
    return a
```



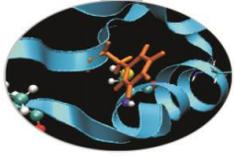
Argomenti di una Funzione

```
def bubble_sort(a):  
    n=len(a)  
    while(n>0):  
        for i in range(0,n-1):  
            if(a[i]>a[i+1]):  
                tmp=a[i]  
                a[i]=a[i+1]  
                a[i+1]=tmp  
  
        n=n-1  
    return a
```



Argomenti di una Funzione

```
def ordina(a,method='bubble',copy=False):  
    if method=='bubble':  
        if copy==False:  
            b=bubble_sort(a)  
        else:  
            b=bubble_sort(a[:])  
    elif method=='insert':  
        if copy==False:  
            b=insertion_sort(a)  
        else:  
            b=insertion_sort(a[:])  
    else:  
        print "No valid method"  
    return b
```



Argomenti di una Funzione

```
v=[5,8,2,4,6,7,2,7,1,9,8]; print "v=",v
```

```
l=ordina(v,copy=True); print "l=",l; print "v=",v
```

```
l2=ordina(v); print "l2=",l2; print "v=",v
```

```
l3=ordina(v,copy=True,method='insert'); print "l3=",l3
```

Output:

```
v= [5, 8, 2, 4, 6, 7, 2, 7, 1, 9, 8]
```

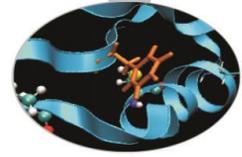
```
l= [1, 2, 2, 4, 5, 6, 7, 7, 8, 8, 9]
```

```
v= [5, 8, 2, 4, 6, 7, 2, 7, 1, 9, 8]
```

```
l2= [1, 2, 2, 4, 5, 6, 7, 7, 8, 8, 9]
```

```
v= [1, 2, 2, 4, 5, 6, 7, 7, 8, 8, 9]
```

```
l3= [1, 2, 2, 4, 5, 6, 7, 7, 8, 8, 9]
```

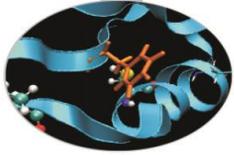


Argomenti di una Funzione

Python consente di utilizzare come parametro di funzione una lista di lunghezza variabile di argomenti. Un asterisco * preposto al nome del parametro indica una lista di argomenti di lunghezza variabile.

```
def func_var_arg(a,b,c,..., *args):
```

Questi argomenti vengono mantenuti in una tupla.

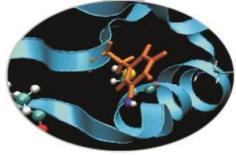


Argomenti di una Funzione

Esempio

```
def min_max_average(*args):  
    avg = 0  
    n = 0  
    for el in args:  
        n += 1  
        avg += el  
    avg /= float(n)  
    min_e = min(args)  
    max_e = max(args)  
    print "Media ", avg  
    print "Min ", min_e  
    print "Max ", max_e
```

Argomenti di una Funzione



Ouput

```
>>min_max_average(3,2,1)
```

Media 2.0

Min 1

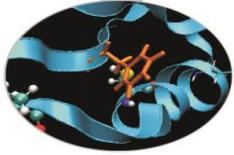
Max 3

```
>>min_max_average(2)
```

Media 2.0

Min 2

Max 2

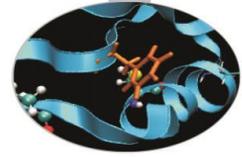


Argomenti di una Funzione

Un doppio asterisco preposto al nome del parametro indica una lista di lunghezza variabile di keyword arguments.

```
def func_var_arg(a,b,c,...,**args):
```

Questi argomenti vengono mantenuti in un *dictionary*.



Argomenti di una Funzione

Esempio

```
def function(a,**kw):  
    print a  
    print kw
```

Output

```
function(3)
```

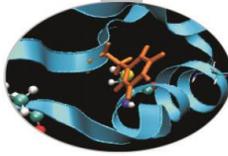
```
3
```

```
{}
```

```
>>function(3,b=2,c=4)
```

```
3
```

```
{'c':4,'b':2}
```

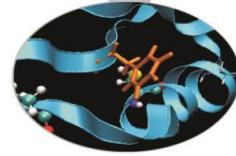


Argomenti di una Funzione

Python permette di utilizzare una funzione come argomento di un'altra funzione. Questa situazione è abbastanza tipica. Data una funzione $f(x)$ è necessaria una funzione per:

- Calcolare gli zeri di $f(x)$
- Calcolare l'integrale di $f(x)$ in $[a,b]$
- Calcolare un'approssimazione della derivata prima
- Etc

La funzione argomento viene trattata allo stesso modo di qualsiasi altro argomento: un riferimento ad un object, in questo caso ad un *function-object*.



Argomenti di una Funzione

Esempio Approssimazione della derivata seconda in un punto.

```
def f(x):
```

```
    return x**3
```

```
def diff2(f,x,h=1e-6):
```

```
    r=(f(x-h)-2*f(x)+f(x+h))/float(h*h)
```

```
    return r
```

```
if __name__ == '__main__':
```

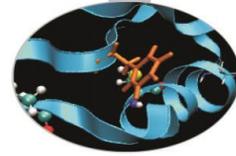
```
    print "Diff2 di x^3 per x=", 3, " ", diff2(f,3)
```

```
    print "Diff2 di sin(x) per x=", pi, " ", diff2(sin,pi)
```

Output

Diff2 di x^3 per x=3 18.001600210482138

Diff2 di sin(x) per x=pi 0.0



Argomenti di una funzione

Le funzioni in Python sono strutture molto flessibili: sono oggetti a tutti gli effetti. Questo consente a una funzione di essere argomento di un'altra funzione, return value, di essere assegnata ad una variabile etc.

Esempio:

```
def f1(arg):
```

```
    print "Sono la funzione 1"
```

```
    if (arg):
```

```
        def f2(x):
```

```
            print "Sono la funzione 2"
```

```
            print "x-value", x
```

```
            return x
```

```
        return f2
```

```
    else: return None
```

Output

```
a=f1
```

```
k=a(2)
```

```
Sono la funzione 1
```

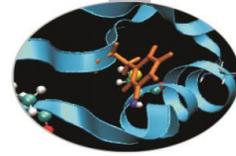
```
xx=k(3)
```

```
Sono la funzione 2
```

```
x-value 3
```

```
print "xx value ", xx
```

```
xx value 3
```



Return Value

La keyword *return* permette di specificare dei valori di ritorno di una funzione al programma chiamante.

In Python è possibile specificare più parametri di ritorno.

Esempio:

```
def swap(a,b):  
    tmp=b  
    b=a  
    a=tmp
```

Output:

```
m=2, n=3
```

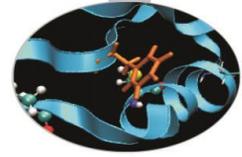
```
print "Prima m=",m," n=",n
```

```
Prima m=2 n=3
```

```
swap(m,n)
```

```
print "Dopo m=",m," n=",n
```

```
Dopo m=2 n=3
```



Return Value

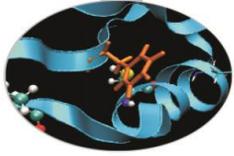
Esempio:

```
def swap(a,b):  
    tmp=b  
    b=a  
    a=tmp  
    return a,b
```

Output:

```
m=2, n=3  
print "Prima m=",m," n=",n  
Prima m=2 n=3  
  
m,n=swap(m,n)  
print "Dopo m=",m,"n=",n  
Dopo m=3 n=2
```

Scoping di Variabili



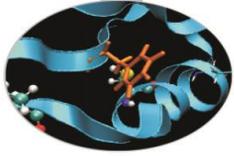
Con *scope* di variabili intendiamo la visibilità di una variabile all'interno di un codice.

Con *namespace* si intende una collezione di nomi di entità. In Python viene implementato tramite un dizionario che associa ad ogni variabile il suo valore.

Ciascuna funzione definisce un proprio namespace locale. Anche i metodi di una classe seguono le regole di scoping delle normali funzioni.

Ciascun modulo definisce un proprio namespace.

Scoping di Variabili

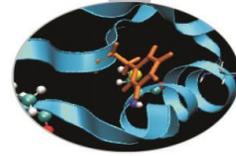


Possiamo distinguere tra differenti namespace:

- *Built-in namespace*
- *Module namespace* definito in un modulo o in un file
- *Local namespace*, definito da una funzione o da una classe

L'interprete durante l'esecuzione ricerca i nomi nell'ordine:

Local namespace → *Global namespace* → *Built-in namespace*



Scoping di Variabili

Esempio:

```
x=int(raw_input("Inserisci un numero: "))
```

Built-in

Module/global

```
def square(x):
```

Module/global

```
    x=x*x
```

Local

```
    print "Inside function: x è uguale a ",x
```

```
square(x)
```

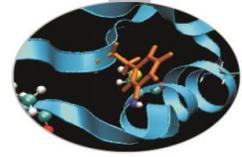
```
print "Outside function: x è uguale a ", x
```

Output:

Inserisci un numero: 5

Inside function: x è uguale a 25

Outside function: x è uguale a 5



Scoping di Variabili

ESEMPIO

```
>>>val=4
```

```
>>> if (val>5):
```

```
    def func3(stringa):
```

```
        print 'Sono la funzione func3',stringa+str(3)
```

```
else:
```

```
    def func1(stringa):
```

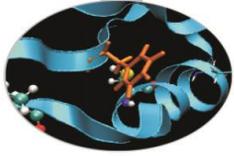
```
        def func2(stringa):
```

```
            print 'Sono la funzione func2',stringa+str(2)
```

```
        print 'Sono la funzione func1', stringa+str(1)
```

```
    func2(stringa)
```

Scoping di variabile



```
>>> func3()
```

Traceback (most recent call last):

File "<pyshell#195>", line 1, in <module>

```
func3()
```

NameError: name 'func3' is not defined

```
>>> func1('Ciao')
```

Sono la funzione func1 Ciao1

Sono la funzione func2 Ciao2

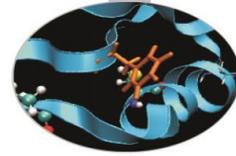
```
>>> func2('Ciao')
```

Traceback (most recent call last):

File "<pyshell#197>", line 1, in <module>

```
func2('Ciao')
```

NameError: name 'func2' is not defined



Scoping di Variabili

Il nome di una variabile globale può essere sovrascritto da una variabile locale. Per evitare ambiguità è utile fare ricorso allo statement *global* con la sintassi *global var1[,var2[,...,varN[*.

Esempio

```
globalVar='abc'  
def stampa():  
    globalVar = 'def'  
    print 'Inside stampa()', globalVar  
print 'Outside gg:', globalVar
```

Output

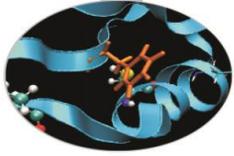
```
Inside stampa() def  
Outside gg: abc
```

```
globalVar='abc'  
def stampa():  
    global globalVar  
    globalVar='def'  
    print 'Inside stampa()',globalVar  
print 'Outside globalVar', globalVar
```

Output

```
Inside stampa() def  
Outside gg: def
```

Scoping di Variabili



NOTA1:

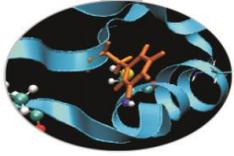
Il caricamento di un modulo avviene tramite la keyword *import*.
Esistono però due modalità differenti:

`import module`

`from module import`

Nel primo caso viene importato il modulo ma non viene modificato il namespace corrente. Nel secondo caso vengono importate funzioni e attributi da *module* al *namespace* corrente.

Scoping di Variabili

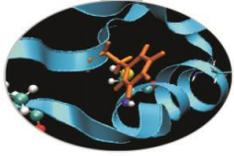


Esempio:

```
import os  
os.path.basename(os.getcwd())  
'Python25'
```

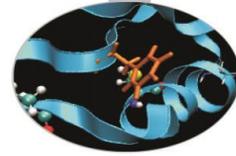
```
from os import *  
path.basename(getcwd())  
'Python25'
```

Scoping di Variabili



NOTA2:

La funzione *locals()* e la funzione *globals()* restituiscono rispettivamente un dizionario con le variabili locali e globali dello scope corrente.



Scoping di Variabili

```
>>> globals()
```

```
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, '__package__': None}
```

```
>>> import math
```

```
>>> globals()
```

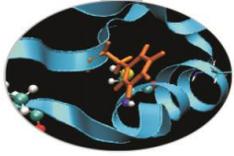
```
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', '__doc__': None, 'math': <module 'math' (built-in)>, '__package__': None}
```

```
>>> from math import *
```

```
>>> globals()
```

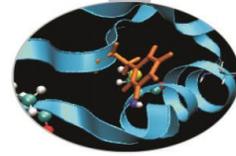
```
{'pow': <built-in function pow>, 'fsum': <built-in function fsum>, 'cosh': <built-in function cosh>, 'ldexp': <built-in function ldexp>, 'hypot': <built-in function hypot>, 'acosh': <built-in function acosh>, 'tan': <built-in function tan>, 'asin': <built-in function asin>, 'isnan': <built-in function isnan>, 'log': <built-in function log>, 'fabs': <built-in function fabs>, 'floor': <built-in function floor>, 'atanh': <built-in function atanh>, 'sqrt': <built-in function sqrt>, '__package__': None, 'frexp': <built-in function frexp>, 'factorial': <built-in function factorial>, 'degrees': <built-in function degrees>, 'pi': 3.141592653589793, 'log10': <built-in function log10>, '__doc__': None, 'math': <module 'math' (built-in)>, 'asinh': <built-in function asinh>, 'fmod': <built-in function function gamma>}
```

Scoping di Variabili



```
>>> def myf():  
    x=5  
    a=3  
    print locals()
```

```
>>> myf()  
{'a': 3, 'x': 5}
```



Lambda Function

Python permette la creazione di funzioni *inline anonime*, tramite la keyword *lambda*, con la sintassi:

lambda <args> : <expression>

equivalente ad un funzione con argomenti *args* e con *return-value expression*.

Sebbene Python non sia un linguaggio funzionale, l'utilizzo delle *lambda function* congiuntamente all'utilizzo di alcune funzioni *built-in* permette la creazione di costrutti tipici del paradigma funzionale.

Esempio:

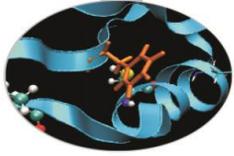
```
def power(x): return x**2
```

```
print "f:" , power(4)
```

```
g=lambda x: x**2
```

```
print "g:" , g(4)
```

Lambda Function

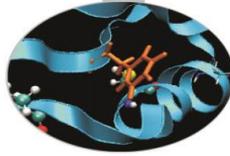


Le funzioni *built-in*:

- *filter(function or None, sequence) -> list*
- *map(function, sequence,[sequence,...]) -> list*
- *reduce(function, sequence[, initial]) -> value*

Sono utilizzate spesso in congiunzione con le funzioni lambda.

Lambda Function



- *filter* : applica un filtro ad una sequenza e ritorna tutti gli elementi della sequenza dove è True la funzione

```
>>> l=range(10)
```

```
>>> filter(lambda x: x < 3, l)
```

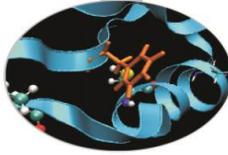
```
[0,1,2]
```

```
>>>def f(x):
```

```
    if(x<3): return True
```

```
>>>filter(f,l)
```

Lambda Function



- *map* : applica una funzione a tutti gli elementi di una sequenza e ritorna gli elementi modificati secondo la funzione

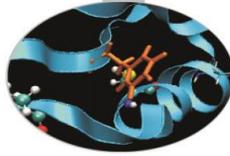
```
>>>a=range(10); print a
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>>b=map(lambda x:x*3,a); print a
```

```
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Lambda Function



- *reduce*: esegue un'operazione di riduzione su tutti gli elementi di una sequenza e ritorna un singolo valore ottenuto combinando gli elementi secondo quanto definito dalla funzione

```
>>>a=range(10); print a
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

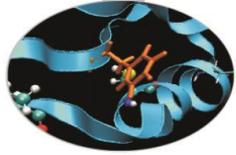
```
>>>b=reduce(lambda x,y: x+y,a); print b
```

```
45
```

```
>>>c=sum(a); print c
```

```
45
```

Lambda Function

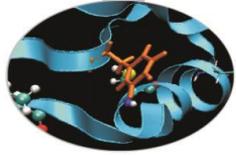


Esempio: Prodotto Matrice-Vettore:

-Metodo 1:

```
def prod_mat_vect1(m,v,dim,dim2):  
    res=[0 for i in xrange (len(v))]  
    for i in xrange(dim):  
        for j in xrange(dim2):  
            res[i]+=m[i][j]*v[j]  
    return res
```

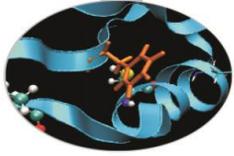
Lambda Function



-Metodo 2: eliminazione ciclo colonne

```
def prod_mat_vect2(m,v,dim):  
    res=[0 for i in xrange(dim)]  
    for i in xrange(dim):  
        res[i]=reduce(lambda x,y: x+y,map(lambda x,y: x*y, m[i][:], v))  
    return res
```

Lambda Function



-Metodo 3: eliminazione ciclo righe e colonne

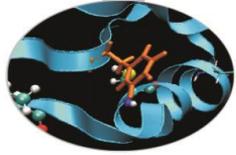
```
def prod_mat_vect3(m,v,dim):
```

```
    index=xrange(dim)
```

```
    res=map(lambda i: reduce(lambda x,y: x+y,map(lambda x,y: x*y, m[i][:], v)),index)
```

```
    return res
```

Lambda function



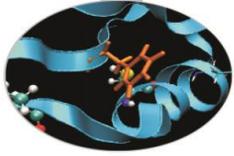
Sono solitamente utilizzate per definire piccole porzioni di codice al posto dello statement *def* o dove l'utilizzo degli statement non è consentito sintatticamente o dove si necessita l'utilizzo una sola volta.

Esempio (switch with dictionary):

```
def plus():  
    return 2+2  
def minus():  
    return 2-2  
def mult():  
    return 2*2  
d={'plus':plus(),'minus':minus(), 'mult': mult()}
```

Oppure risparmiando qualche riga di codice:

```
d={'plus': (lambda :2+2) ,  
'minus': (lambda: 2-2)  
'mult': (lambda: 2*2) }
```



Funzioni Ricorsive

Una funzione si definisce ricorsiva se contiene all'interno della sua definizione una chiamata a se stessa.

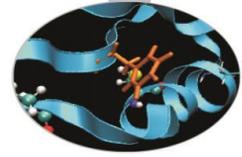
Esempio

```
def factorial(n):  
    if n==0 or n==1:  
        return 1  
    else:  
        return n*factorial(n-1)
```

factorial(5)

Soluzioni alternative al calcolo del fattoriale possono includere l'utilizzo di *lambda-function* e della funzione *reduce* come segue

Funzioni Ricorsive



Esempio:

```
def mult(x,y):  
    return x*y
```

```
def fact1_mult(n):  
    try:  
        return reduce(mult, xrange(2,n+1))  
    except TypeError:  
        return 1
```

```
def fact2_mult(n):  
    try:  
        return reduce(lambda x,y: x*y, xrange(2,n+1))  
    except TypeError:  
        return 1
```

```
f=fact1_mult(40)
```

```
f2=fact2_mult(40)
```