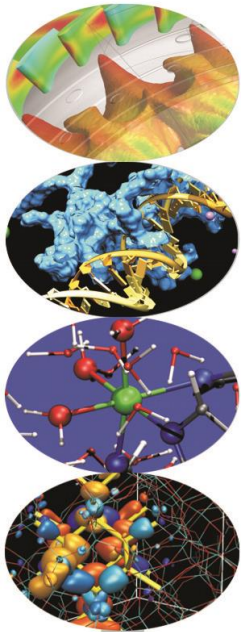
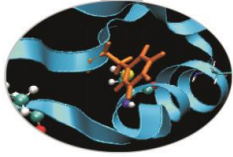


# Dato Strutturato



# Tipo di Dato Strutturato



Oltre ai tipi di dato nativi del linguaggio come stringhe, interi, numeri in virgola mobile o complessi, comuni anche ad altri linguaggi di programmazione, Python fornisce delle strutture dati native del linguaggio.

Tali strutture si suddividono in sequenze ordinate di dati o collezioni di dati.

Ma si possono anche classificare come strutture mutevoli o immutabili.

Le built-in data structures in Python sono:

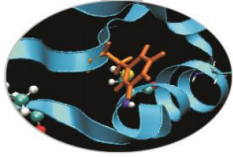
• Liste

• Tuple

• Dizionari

• Set

• Frozenset



# Liste

La *list* è un tipo di dato strutturato nativo del linguaggio. Gli elementi che costituiscono una lista possono essere istanze di oggetti nativi del linguaggio o istanze di oggetti definiti dall'utente. Una lista può contenere contemporaneamente elementi di diverso tipo.

Per istanziare una lista non è necessario specificare lunghezza o tipo di dato. Gli elementi di una lista sono contenuti tra [].

Per accedere al singolo elemento della lista si ricorre all'operatore [].

## Esempio:

```
>>>l=[]
```

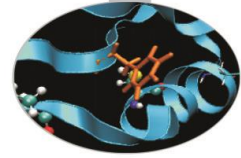
```
>>>m=['Lista','di',4,'elementi']
```

```
>>>print m[2],m[0]
```

```
>>>4 Lista
```

L'ordinamento viene mantenuto. Una lista non è una collezione di dati ma è una sequenza ordinata di dati.

# Liste



- Le liste supportano l'operatore di slicing [start:stop:step]

```
>>>a=[0,1,2,3,4,5,6,7]
```

```
>>>a[0:6]
```

```
[0,1,2,3,4,5]
```

```
>>>a[1:6:2]
```

```
[1,3,5]
```

```
>>>a[1::2]
```

```
[1,3,5,7]
```

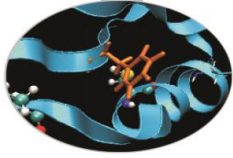
```
>>>a[::2]
```

```
[0,2,4,6]
```

Lo slicing può essere anche negativo

```
>>>a[6:0:-2]
```

```
[6,4,2]
```



# Liste

- La funzione `range()` serve per generare liste di numeri interi
- La funzione `range(start,stop,step)` genera una lista di interi da `start` a `stop` con passo `step`.

```
>>>a=range(3)
```

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

```
>>>type(a)
```

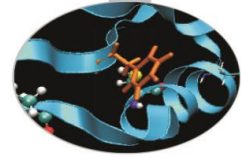
```
<type 'list'>
```

```
>>>a=range(1,10)
```

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

```
>>>a=range(1,10,2)
```

```
[1,3,5,7,9]
```



# Liste

- Sulle liste è possibile eseguire operazioni complesse.

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

```
>>>b=[el*2 for el in a]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
>>> l=[1,2]
```

```
>>> l2=['a','b']
```

```
>>> l3=[4,5]
```

```
>>> f=[(e1,e2,e3) for e1 in l for e2 in l2 for e3 in l3]
```

```
[(1, 'a', 4), (1, 'a', 5), (1, 'b', 4), (1, 'b', 5), (2, 'a', 4), (2, 'a', 5), (2, 'b', 4), (2, 'b', 5)]
```

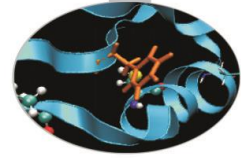
Equivalente a :

```
>>> for e1 in l:
```

```
    for e2 in l2:
```

```
        for e3 in l3:
```

```
            f.append((e1,e2,e3))
```



# Liste

Le liste sono contenitori dati modificabili. Gli oggetti di tipo lista contengono metodi `built_in` per manipolare e modificare i dati membro.

## • Inserimento

- `append(object)` inserimento di *object* in coda
- `insert(index,object)` inserimento di *object* in posizione `index`
- `extend(iterable)` concatenazione di liste

## • Ricerca

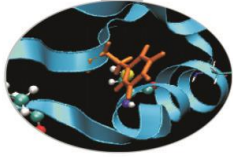
- `index(value, [start, [stop]])` ricerca di *value* e ritorno della prima occorrenza
- `count(value)` calcolo delle occorrenze di *value*

## • Eliminazione

- `remove(value)` eliminazione della prima occorrenza di *value*
- `pop([index])` eliminazione elemento in posizione *index*







# Liste

Gli operatori + e \* possono essere applicati anche sulle liste.

L'operatore + effettua una concatenazione tra due liste.

L'operatore \* effettua una ripetizione.

Python supporta per le liste anche l'operatore += .

L'operatore + e la funzione extend hanno le stesse funzionalità. I tempi di esecuzione sono però differenti.

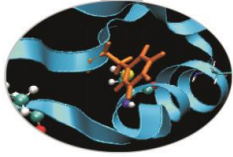
## Esempio:

```
import time
```

```
l=range(100000000)
```

```
v=range(1000000)
```

# Liste



```
T1=time.clock()
```

```
s=l+v
```

```
T2=time.clock()
```

```
print "Tempo di esecuzione di + :", T2-T1 , "s"
```

```
T3=time.clock()
```

```
l.extend(v)
```

```
T4=time.clock()
```

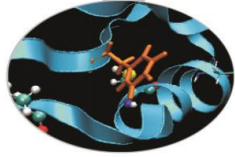
```
print "Tempo di esecuzione di extend :", T4-T3 , "s"
```

## OUTPUT:

Tempo di esecuzione di +: 2.81 s

Tempo di esecuzione di extend: 0.033 s

# Liste



I metodi implementati nella lista la rendono particolarmente adatta ad essere utilizzata come *stack* o come *queue*.

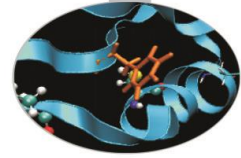
I metodi *pop* e *append* possono essere usati per implementare la logica LIFO tipica degli stack.

I metodi *pop* con indice 0 e *append* possono essere usati per implementare la logica FIFO tipica della queue.

## Esempio:

```
# file stack_queue.py
stack=[1, 2, 3, 4]
print 'Stack iniziale:', stack
for i in range(5,7):
    stack.append(i)
print "Append: ", stack
stack.pop()
print "Pop:" , stack
```

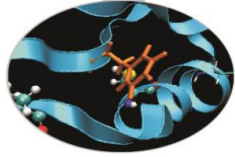
# Liste



```
queue=[ 'a','b','c','d' ]  
print "Queue iniziale:", queue  
queue.append('e')  
queue.append('f')  
print "Append:", queue  
queue.pop(0)  
print "Pop:", queue
```

## OUTPUT:

```
Stack iniziale: [1, 2, 3, 4]  
Append: [1, 2, 3, 4, 5, 6]  
Pop: [1, 2, 3, 4, 5]  
Queue iniziale: ['a', 'b', 'c', 'd']  
Append: ['a', 'b', 'c', 'd', 'e', 'f']  
Pop: ['b', 'c', 'd', 'e', 'f']
```



# Tuple

Le *tuple* sono sequenze ordinate di dati racchiusi tra (). Le tuple sono liste molto particolari i cui elementi sono immutabili.

L'accesso al singolo elemento avviene attraverso l'operatore [].

Lo slicing [start:end] è ammesso anche sulle tuple.

## Esempio:

```
>>>t=(1,2,3,4)
```

```
>>>el=t[1]
```

```
>>>t2=t[0:2]
```

```
>>>print el, t2
```

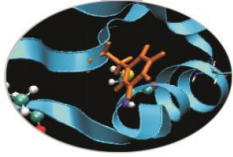
```
2 (1,2)
```

I dati contenuti in una tupla possono essere di tipo eterogeneo.

Le tuple sono oggetti immutabili. Non contengono pertanto metodi di:

- eliminazione
- inserimento

# Tuple



## Esempio:

```
>>>t1=(1,2,3,4,'ciao','mondo',[2,3])
```

```
>>>t1[3]='jkjk'
```

Traceback (most recent call last):

File "<pyshell#26>", line 1, in <module>

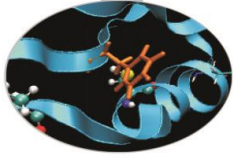
```
t1[1]=3
```

TypeError: 'tuple' object does not support item assignment

Sulle tuple sono definiti gli operatori *in* e *not in*.

Le tuple sono indicate per definire set di valori costanti. Per alcune operazioni risultano più efficienti delle liste.

# Tuple



## Esempio:

```
import timeit
```

```
sum=0
```

```
t0=timeit.Timer('t=(1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8)')
```

```
t00=t0.timeit()
```

```
print 'Tuple creating time ',t00, '\n'
```

```
t1=timeit.Timer('l=[1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8,1,2,3,4,5,6,7,8]')
```

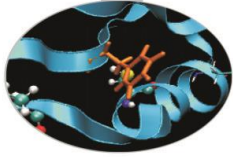
```
t11=t1.timeit()
```

```
print 'List creating time ',t11, '\n'
```

```
t=tuple( range(9000))
```

```
l=range(9000)
```

# Tuple



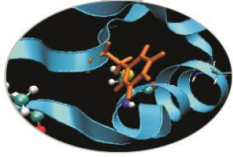
```
x=timeit.Timer('sum1(t)','from __main__ import sum1,t')
t1=x.timeit()
print 'Time accessing a tuple '+str(t1)+ '\n'
```

```
y=timeit.Timer('sum1(l)','from __main__ import sum1,l')
t2=y.timeit()
print 'Time accessing a list '+str(t2)+' \n'
```

```
def sum1(l):
    s=0
    for i in xrange(l):
        s+=i
```



# Tuple



## OUTPUT:

Time creating a tuple 0.0415067049835

Time creating a list 1.07190083708

Time accessing a tuple 0.0719978947727

Time accessing a list 0.0729975422291

E' possibile convertire un dato tupla in un dato lista e viceversa  
rispettivamente attraverso le funzioni *list* e *tuple*.

## Esempio:

```
>>>l=[1,2,1,2]
```

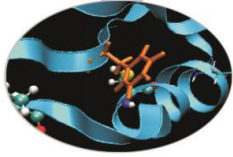
```
>>>t=tuple(l)
```

```
>>>l2=list(t)
```

```
>>>type(t); type(l2)
```

```
<type 'tuple'>
```

```
<type 'list'>
```



# Dictionary

Un *dictionary* è un insieme non ordinato di oggetti. Ogni oggetto di un dictionary è identificato univocamente da una *key*.

Ogni elemento di un dictionary è rappresentato da una coppia *key – value*.

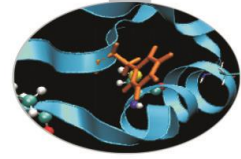
In Python le *keys* possono essere solo oggetti immutabili: le tuple, per esempio, possono essere usate come chiavi di indicizzazione di un *dictionary*.

Come le tuple e le liste anche i dizionari sono tipi built-in.

I dati sono racchiusi tra { }, l'accesso al singolo elemento avviene attraverso [ ].

## Esempio:

```
>>>d={ } # dizionario vuoto
>>>d={1: 'Hello', 'due': 'World'} # dizionario con due elementi
>>>d[1]
'hello'
```



# Dictionary

Le chiavi in Python sono univoche e sono inoltre case-sensitive.  
I dati contenuti in un dizionario possono essere eterogenei.

## Esempio:

```
>>>d={'a':'Ciao','b':'Mondo'}
```

```
>>>d[a]='CIAO';
```

```
>>>d
```

```
{1: 'CIAO', 'due': 'world'}
```

```
>>>d['c']          # Errore la chiave non è presente
```

Traceback (most recent call last):

```
File "<pyshell#73>", line 1, in <module>
```

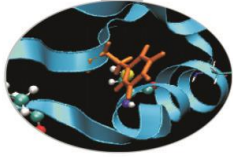
```
    d['c']
```

```
KeyError: 'c'
```

```
>>>(d.key(),d.value())
```

```
([1, 'due'], ['CIAO', 'world'])
```

Le funzioni *key()* and *value()* ritornano rispettivamente le chiavi e i valori di un dizionario.



# Dictionary

I dizionari vengono manipolati attraverso le funzioni di:

## •Update:

Un dizionario può essere modificato aggiungendo items o modificando gli esistenti. I metodi built-in preposti:

- update(E, \*\*F)

## Esempio:

```
>>>d={1:'Hello',2:'World' }
```

```
>>>d.update({3:'Ciao'}) # aggiunta tramite il metodo update
```

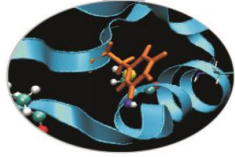
```
>>>d[4]='Mondo' # aggiunta di un item
```

```
>>>d[2]='world' # modifica di un item
```

```
>>>d
```

```
{1: 'Hello', 2: 'world', 3: 'Ciao', 4: 'Mondo'}
```

# Dictionary



## •Eliminazione:

Tramite cancellazione di un elemento o eliminazione dell'intero contenuto di un dizionario. I metodi preposti:

- clear()
- pop(k[,d])
- popitem()

## Esempio:

```
>>>d={1:'Mon',2:'Tue',3:'Wed',4:'Thu',5:'Fri',6:'Sat',7:'Sun'}
```

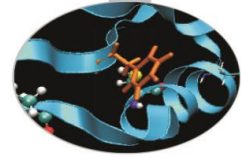
```
>>>l1=d.popitem()          # popitem
```

```
>>>d
```

```
{2: 'Tue', 3: 'Wed', 4: 'Thu', 5: 'Fri', 6: 'Sat', 7: 'Sun'}
```

```
>>>l2=d.pop(3)           # pop
```

```
{2: 'Tue', 4: 'Thu', 5: 'Fri', 6: 'Sat', 7: 'Sun'}
```



# Dictionary

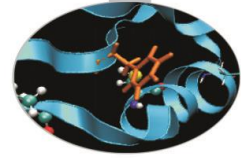
```
>>>del d[4]                #del
>>>d
{2: 'Tue', 5: 'Fri', 6: 'Sat', 7: 'Sun'}
>>>d.clear()              #clear
>>>d
{ }
```

## • Altri metodi:

- get(k[,d])
- has\_key(k)
- items()

## Esempio:

```
>>> d={1:'Mon',2:'Tue',3:'Wed',4:'Thu',5:'Fri',6:'Sat',7:'Sun'}
>>>d.has_key(8)
False
```



# Dictionary

```
>>>d.get(7)
```

```
'Sun'
```

```
>>>d.items()
```

```
[(1, 'Mon'), (2, 'Tue'), (3, 'Wed'), (4, 'Thu'), (5, 'Fri'), (6, 'Sat'), (7, 'Sun')]
```

Un utilizzo interessante dei dizionari o array associativi consiste nell'immagazzinamento di matrici sparse. Una struttura dati efficiente per l'immagazzinamento di matrici sparse consente di ridurre i costi computazionali e di ridurre l'occupazione in memoria.

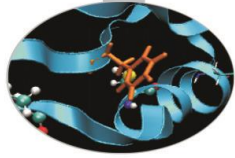
Nel formato *COordinate* si immagazzinano le coppie (i,j) corrispondenti ai valori  $a_{ij}$  non nulli.

**Esempio:**

$$a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{bmatrix}$$

$$d = \{(0,0) : 1, (1,2) : 1, (2,0) : 2\}$$

# Dictionary



Al posto delle tuple, si potrebbe usare come chiave un intero, secondo la mappatura (per righe):  $(i,j) \rightarrow j+i*n$  dove  $n$  è il numero di colonne.

Nell' esempio precedente:

## Esempio:

#a è una lista di liste  $a=[[a1, a2,...an], \dots, [m1,m2,...,mn]]$

#Matrice sparsa 100 X 100:

#Definizione del dizionario con tuple

d={}

```
for i in range(0,len(a)):
```

```
    for j in range (0,len(a[0])):
```

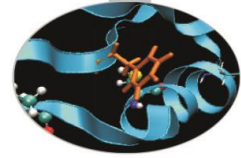
```
        if a[i][j]>0:
```

```
            t=(i,j)
```

```
            d.update({t:a[i][j]})
```



# Dictionary



**#Definizione del prodotto matrice-vettore con il metodi COordinate**

```
def prod_m_v_dict(d,v,l):
```

```
    b=[]
```

```
    for i in xrange(l):
```

```
        b.append(0)
```

```
    for k in d.keys():
```

```
        i=k[0]
```

```
        j=k[1]
```

```
        val=d[k]
```

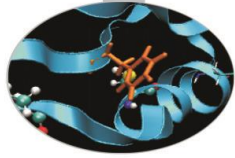
```
        b[i]+=val*v[j]
```

**#Definizione del prodotto matrice-vettore standard**

```
def prod_m_v_list(a,v):
```

```
    b=[]
```

# Dictionary



```
for i in range(0,len(a)):
    somma=0
    for j in range(0,len(a[0])):
        somma+=a[i][j]*v[j]
    b.append(somma)
```

## #Tempi di calcolo:

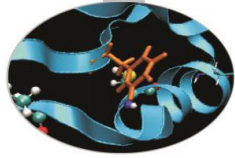
Prodotto matrice vettore liste: 400.151556209 microsec

Prodotto matrice vettore dict: 17.8503196928 microsec

## #Definizione moltiplicazione per uno scalare con array associativo

```
def prod_scalare_dict(d,s):
    dd={}
    for k in d.keys():
        dd.update({k:d[k]*s})
```

# Dictionary



**#Definizione della moltiplicazione per uno scalare caso std**

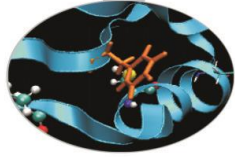
```
def prod_scalare_list(a,s):  
    for i in range (0, len(a)):  
        for j in range(0, len(a[0])):  
            a[i][j]=a[i][j]*s
```

**#Tempi di Calcolo**

Prodotto per uno scalare list: 467.861930815 microsec

Prodotto per uno scalare dict: 22.1639234568 microsec

# Switch Case e Dictionary



Il costrutto *switch-case* permette di controllare il flusso del programma in base al valore di una variabile o espressione.

In Python è possibile emulare il funzionamento del costrutto *switch-case* utilizzando come struttura dati dei *dictionary*.

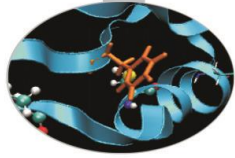
La forma generale è la seguente:

```
{  
Key_1: function_1  
Key_2: function_2  
Key_3: function_3  
...  
Key_N: function_N  
}
```

E' possibile gestire *case* diversi da interi.

Lo switch-case costruito con i dizionari risulta più efficiente del blocco if-else.

# Switch Case e Dictionary



## #definizione del dizionario

```
caso=random.randint(1,30)
```

```
d={1:f_1, 2:f_2, 3:f_3, 4:f_4, 5:f_5, 8:f_1, 9:f_1,10:f_1,11:f_1  
,12:f_2,13:f_2,14:f_3,15:f_3,16:f_4,17:f_4,18:f_5,19:f_5}.get(caso,default)()
```

## #definizione del blocco if-elif-else

```
if caso==1 or caso==8 or caso==9 or caso==10 or caso==11:
```

```
    f_1(y)
```

```
elif caso==2 or caso==12 or caso==13:
```

```
    f_2(y)
```

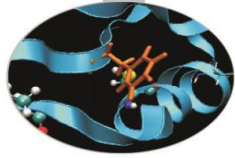
```
elif caso==3 or caso==14 or caso==15:
```

```
    f_3(y)
```

```
elif caso==4 or caso==16 or caso==17:
```

```
    f_4(y)
```

# Switch Case e Dictionary



```
elif caso==5 or caso==18 or caso==19:
```

```
    f_5(y)
```

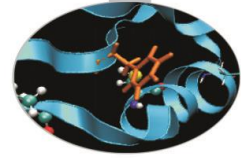
```
else:
```

```
    default(y)
```

## #OUTPUT

Tempo di calcolo utilizzando dict: 0.07 sec

Tempo di calcolo utilizzando if-elif: 1.46 sec



# Set – Frozenset

Python mette a disposizione due distinte strutture dati per la rappresentazione di insiemi di elementi.

Un *set* è una collezione mutabile non ordinata di oggetti.

Un *frozenset* è una collezione immutabile non ordinata di oggetti.

In entrambi i casi gli elementi sono univoci.

## Esempio:

```
>>>s=set(('ciao',1,'Mondo'))
```

```
>>>fs=frozenset(('ciao',2))
```

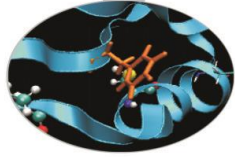
I *set* mettono a disposizione dei metodi per la modifica dell'insieme di dati.

## •Inserimento:

-add(obj)

-update(obj)

# Set – Frozenset



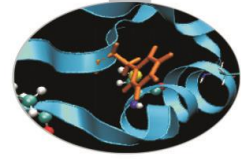
## •Eliminazione

- discard(x) remove(x)
- clear()
- pop()

## Esempio:

```
>>>s=set(('abc','def',1,2,3,'ghi'))
>>>s.add(4)
>>>s.update(('lmn',5))
>>>s
set([1, 2, 3, 4, 5, 'abc', 'lmn', 'ghi', 'def'])
>>>s.discard(4)
>>>s.remove(5)
>>>s.pop()
```





# Set - Frozenset

```
>>>s
```

```
set([2, 3, 'abc', 'lmn', 'ghi', 'def'])
```

```
>>>s.clear(); s
```

```
set([ ])
```

Entrambi i contenitori dati dispongono di metodi per gestire le operazioni tra insiemi:

*-union,*

*-intersection,*

*-difference,*

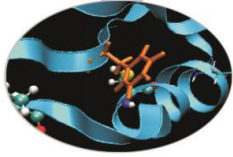
*-issubset,*

*-issuperset*

## Esempio:

```
>>>s=set((1,2))
```

```
>>>s2=frozenset((2,3,4))
```



# Set - Frozenset

```
>>>s3=s.union(s2)
```

```
>>>s4=s.difference(s2)
```

```
>>>s5=s2.intersection(s)
```

```
>>>s.issubset(s2)
```

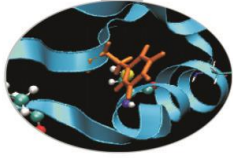
False

```
>>>print 's3', s3 , 's4', s4, 's5', s5
```

```
s3 set([1, 2, 3, 4]) s4 set([1]) s5 frozenset([])
```

In entrambi i casi i dati contenuti possono essere eterogenei.

Inoltre i *frozenset* in quanto immutabili possono essere utilizzati per l'indicizzazione dei dizionari.



# Sequenze dati e cicli

- Il ciclo for consente di iterare su oggetti iterabili come lista,tuple,stringhe, set, dizionari.

## LISTE

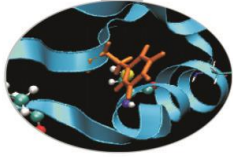
```
>>> a=[1,2,3,4,5]
>>> for el in a:
print el
1
2
3
4
5
```

## STRINGHE

```
>>> a="Ciao"
>>> for el in a:
print el
C
i
a
o
```

## SET

```
>>> a=set([1,2,3,4])
>>> for el in a:
print el
1
2
3
4
```



# Sequenze dati e cicli

## DIZIONARI chiave

```
>>> a={1:'a',2:'b'}  
>>> for el in a.keys():  
print el
```

```
1  
2
```

## DIZIONARI valori

```
>>> a={1:'a',2:'b'}  
>>> for el in a.values():  
print el
```

```
a  
b
```

## DIZIONARI chiave-valori

```
>>> a={1:'a',2:'b'}  
>>> for k,v in a.items():  
print k,v
```

```
1 a  
2 b
```

## DIZIONARI

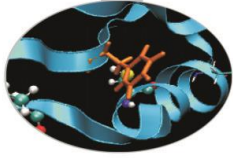
```
>>> a={1:'a',2:'b'}  
>>> for el in a:  
print el
```

```
1  
2
```

## DIZIONARI

```
>>> a={1:'a',2:'b'}  
>>> for el in (1,2,3):  
print a.get(el)
```

```
a  
b  
None
```



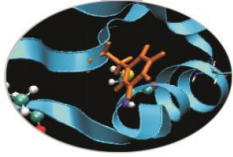
# Iteratori

Per ciclare sugli oggetti di tipo iterable (lista o tupla o stringa) in generale (**containers**) si utilizza il ciclo for.

Il ciclo for per operare all'interno degli elementi del **containers** utilizza un oggetto detto **iteratore**.

L'oggetto **iteratore** ha le caratteristiche necessarie per muoversi nel contenitore: possiede una funzione (metodo *next()*) che restituisce il primo dato disponibile nel contenitore

# Iteratori



Gli iteratori esistono già definiti all'interno di ogni oggetto di tipo contenitore come attributo `__iter__`; a sua volta l'iteratore possiede un metodo `next` che può essere usato per ritornare il valore del primo elemento disponibile

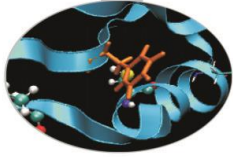
Infatti:

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

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

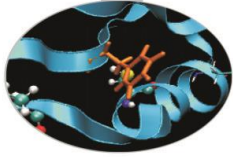
```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',  
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__',  
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',  
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',  
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',  
'__setitem__', '__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append',  
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

# Iteratori



```
>>> my_iter=a.__iter__()  
>>> dir(my_iter)  
['__class__', '__delattr__', '__doc__', '__getattribute__', '__hash__',  
'__init__', '__iter__', '__length_hint__', '__new__', '__reduce__',  
'__reduce_ex__', '__repr__', '__setattr__', '__str__', 'next']  
>>> my_iter.next()  
1  
>>> my_iter.next()  
2  
>>> my_iter.next()  
3
```

usando questi oggetti è possibile implementare un ciclo for più performante soprattutto a livello di memoria quando il dominio su cui si vuole ciclare è molto grande (`xrange`)



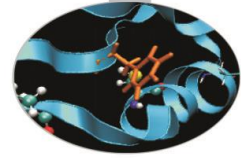
## range e xrange

Come visto negli esempi del corso range restituisce una lista di elementi su cui ciclare.

*xrange* invece sfrutta il concetto di iteratore di container per non creare da principio tutta la lista di indici ma fornendo uno strumento che a *run-time* genera l'indice desiderato.

Il risultato è un ciclo for più performante e decisamente meno costoso in termini di memoria quando le dimensioni del ciclo sono grandi.





# range e xrange

```
from memtracktest import memory
```

```
from time import sleep
```

```
#range
```

```
X=[100,1000,10000,100000,1000000,10000000]
```

```
for x in X:
```

```
    m1=memory()
```

```
    #range
```

```
    a=range(x)
```

```
    m2=memory()
```

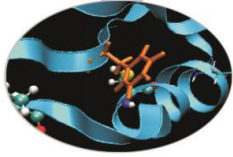
```
    print "Size of the cycle: ",x," Memory used by range: ",m2-m1," bytes"
```

```
    #xrange
```

```
    b= xrange(x)
```

```
    m3=memory()
```

```
    print "Size of the cycle: ",x," Memory used by xrange: ",m3-m2," bytes"
```



# range e xrange

Size of the cycle: 100 Memory used by range: 0.0 bytes

Size of the cycle: 100 Memory used by xrange: 4096.0 bytes

Size of the cycle: 1000 Memory used by range: 8192.0 bytes

Size of the cycle: 1000 Memory used by xrange: 0.0 bytes

Size of the cycle: 10000 Memory used by range: 135168.0 bytes

Size of the cycle: 10000 Memory used by xrange: 0.0 bytes

Size of the cycle: 100000 Memory used by range: 1056768.0 bytes

Size of the cycle: 100000 Memory used by xrange: 0.0 bytes

Size of the cycle: 1000000 Memory used by range: 10887168.0 bytes

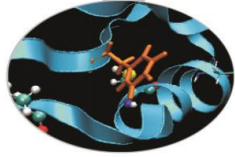
Size of the cycle: 1000000 Memory used by xrange: 0.0 bytes

Size of the cycle: 10000000 Memory used by range: 108879872.0 bytes

Size of the cycle: 10000000 Memory used by xrange: 0.0 bytes

Range aumenta linearmente con la dimensione del ciclo  
(la lista viene effettivamente generata e salvata in memoria)  
*xrange* ha un costo costante.

# Shallow & Deep Copying



Con *shallow copying* si intende una copia per referenza.

Con *deep copying* si intende una copia per valore.

Python di default utilizza una shallow copy nella copia di oggetti mutabili e un deep copy nella copia di oggetti immutabili.

## Esempio:

```
>>>a='Ciao'          # stringa è immutabile
```

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

```
>>>b+=' Mondo'
```

```
>>>a;b
```

```
'Ciao'
```

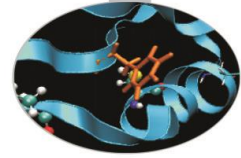
```
'Ciao mondo'
```

```
>>>lista=[1,2]       # lista è mutabile
```

```
>>>b=lista
```

```
>>>b==lista
```

# Shallow & Deep Copying



True

```
>>>b is lista
```

True

```
>>>b.append(3)
```

```
>>>b ; lista
```

```
[1,2,3]
```

```
[1,2,3]
```

```
>>>c=lista[ : ] # slicing opera per valore sulle liste
```

```
>>>c is lista ; c==lista
```

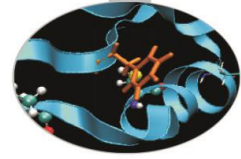
False

True

```
>>>c.append(5)
```

```
>>> c ; l
```

```
[1,2,5]
```



# Shallow & Deep Copying

[1,2]

```
>>> lista=[1,2,[3,4]]
```

```
>>> d=lista[:]
```

```
>>> d[2].append(5)
```

```
>>> d; lista
```

# NOT DEEP! Lo slicing agisce solo ad un livello

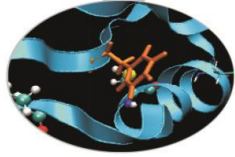
[1,2,[3,4,5]]

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

Python dispone del modulo *copy* per controllare le operazioni di assegnamento tramite *deep* e *shallow copying*.

Il modulo *copy* contiene due funzioni *copy* e *deepcopy* rispettivamente per la *shallow* e la *deep copy*.

# Shallow & Deep Copying



## Esempio:

```
>>>from copy import *  
>>>l=[1,2,[3,4]]  
>>>d=copy.deepcopy(l)  
>>>d.append(5)  
>>>d; l  
[1,2,[3,4,5]]  
[1,2,[3,4]]
```

La funzione *deepcopy* agisce su ogni livello di profondità.