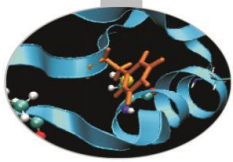
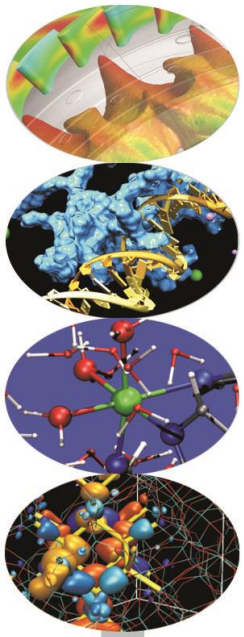
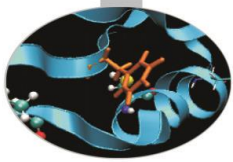


Tipi di Dato



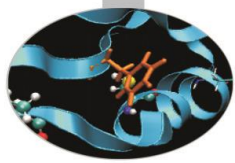
Data Types



Python comparato ad altri linguaggi di programmazione sulla gestione dei tipi di dato:

- Linguaggi staticamente tipati: linguaggi in cui il tipo è noto a tempo di compilazione. Le variabili vengono associate ad un tipo tramite una dichiarazione come in c/c++.
- Linguaggi dinamicamente tipati: linguaggi in cui il tipo è noto a tempo di esecuzione. L'assegnamento permette di individuare il tipo di variabile.
- Linguaggi fortemente tipati: linguaggi nei quali i tipi sono sempre imposti. Un intero non può essere trattato come una stringa senza una conversione esplicita.
- Linguaggi debolmente tipati: linguaggi nei quali i tipi possono essere ignorati, p.e. PHP.

Data Types



Python è un linguaggio **dinamicamente e fortemente tipato.**

Esempio

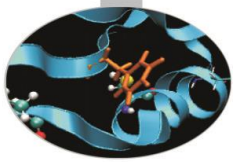
```
>>> a='123'      # tipo string
>>> b=4          # tipo intero
>>> c=b+int(a)   # casting ad intero
>>> c
        127              # ok
>>> d=b+a        # ERRORE!
Traceback (most recent call last):
    File "<pyshell#18>", line 1, in <module>
     d=b+a
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

vs C/C++ char a[]='123'

vs PH $a='2' a è stringa

$a+=1 a è intero

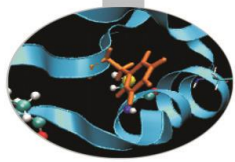
Identificatori



E’ possibile associare ad un oggetto un nome simbolico che non necessita di essere dichiarato.

- Gli *identificatori* sono i nomi dati per identificare qualcosa. Alcune regole:
  - Il primo carattere dell'identificatore deve essere una lettera dell'alfabeto (maiuscola o minuscola) o un carattere _
  - Il resto del nome dell'identificatore può consistere di lettere, caratteri di sottolineatura ('_') o cifre (0-9).
  - I nomi degli identificatori sono case-sensitive.
  - Esempi di identificatori **validi** sono i, __mio_nome, nome_23 e a1b2_c3
  - Esempi di identificatori **non validi** sono 2cose, questo è spaziato, mio-nome "questo è tra virgolette".
  - Esistono parole **riservate**: *and, or, else, if, for, while, def, class* ect etc.

Identificatori



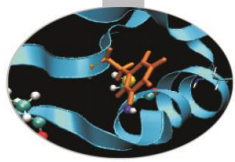
In Python non esistono dichiarazioni esplicite di variabili. Vengono create nel momento dell'assegnamento e distrutte all'uscita dello scope.

Per conoscere il tipo di oggetto associato ad una variabile si utilizza la funzione *built-in type().*

Esempio

```
>>> a=5
>>> type(a)
<type 'int' >
>>>b=3.2
>>>type(b)
<type'float'>
>>>a=7.2
>>>type(a)
<type'float'>
>>>c=a
>>>C=b
>>>c==C
False
```

Tipi di dato

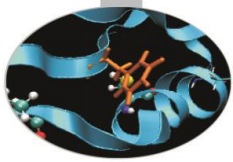


- Python dispone di due tipologie di dato, dati semplici:
  - Int
  - Long
  - Float
  - Complex
  - String

E contenitori.

- Uno dei punti di forza di Python è nei contenitori disponibili, che sono molto efficienti, comodi da usare, e versatili:
  - tuple ()
  - list []
  - dict {}
  - set

Tipo Numerico



Il linguaggio mette a disposizione quattro tipi numerici di dato :

- ***Integer***

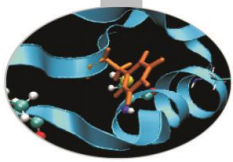
Il tipo *int* è valido per tutti i numeri interi che sono compresi tra [-2147483648 e 2147483647], vedi (*sys.maxint*)

Un intero può essere espresso in base decimale oppure in base esadecimale o ottale antecedendo al numero 0x e 0 rispettivamente:

Esempio

```
>>> a=300          #decimale
>>> b=0x12c        #esadecimale
>>> c=0454         #ottale
>>> a_oct=oct(a)   #ottale
>>> a_hex=hex(a)   #esadecimale
```

Tipo Numerico



- ***Long Integer***

Il tipo **long** è analogo al tipo intero con l'unica eccezione che il valore massimo e minimo che può assumere è limitato solo dalla memoria a disposizione.

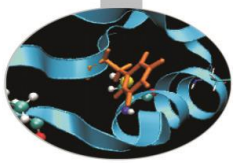
Esempio

```
>>> a = 1254546699L     # il suffisso L indica un long type
>>> b = 484564848766
>>> b
>>> 484564848766L
 >>>2**1024
```

```
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639474124377767893424865485276302219601246094119453082952085005768838150682342462881473913110540827237163350510684586298239947245938479716304835356329624224137216L
```

Possono contenere interi di qualsiasi dimensione!!

Tipo Numerico

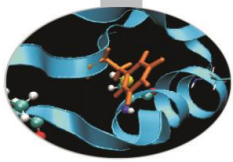


ESEMPIO

```
>>>a=2147483647
>>>type(a)
<type 'int'>
>>>a+=1
>>>type(a)
<type 'long'>
```

Gli int sono automaticamente trasformati in long quando necessario.

Tipo Numerico



- ***Floating Point Number***

Il tipo **float** rappresenta numeri reali in doppia precisione

esempio:

```
>>> a = 12.456
>>> c = 12232e-2
>>> b = .2
>>> 6.12244e-5
```

Attenzione nell'utilizzo di interi e float!!

Cosa Succede se eseguiamo le seguenti operazioni??

100/3

100//3

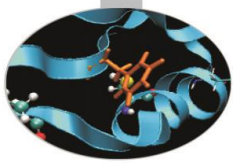
100.0/3

100.0//3

100%3

divmod(100,3)

Tipo Numerico



In Python digitando:

>>> 0.2

0.2000000000000001 ????

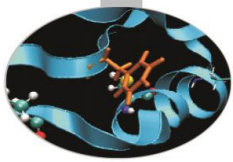
La rappresentazione dei numeri a calcolatore è binaria. Alcuni numeri decimali non possono essere rappresentati correttamente e questo provoca piccoli errori di arrotondamento.

Rappresentazione floating point : $\pm M \cdot B^E$

32-Bit

64-Bit

Tipo Numerico



Per esempio il numero frazionario:

$$0.125 = \frac{1_{10}}{8_{10}} = \frac{1}{10} + \frac{2}{100} + \frac{5}{1000}$$

Può essere rappresentato correttamente dal numero binario frazionario:

$$0.001 = \frac{0}{2} + \frac{0}{4} + \frac{1}{8}$$

Per alcuni numeri frazionari questo non avviene:

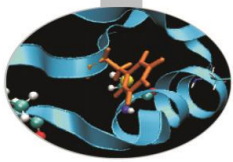
$0.2_{10} = \frac{1_{10}}{5_{10}}$ corrisponde al numero binario frazionario: 0.001001100110011…

In Python digitando:

>>> 0.2

0.2000000000000001 **OK!!!**

Tipo Numerico



Il prompt di Python di default utilizza la funzione built-in repr() per il display di numeri floating point, che per default approssima alla 17-esima cifra significativa.

Quando si necessita maggiore accuratezza (p.e. applicazioni finanziarie) è possibile ricorrere all'utilizzo del modulo Decimal: la precisione è specificata dall'utente e i numeri frazionari binari sono rappresentati esattamente a scapito di un più lento processamento dei dati.

- ***Complex Number***

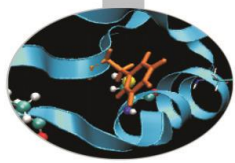
Un numero **complex** rappresenta un tipo numerico complesso in doppia precisione. Si accede alla parte reale e immaginaria di un numero complesso attraverso le funzioni *'real'* e *'imag'.*

esempio

```
>>>r=12+5j
>>>r.imag
5.0
>>>type(r.real)
<type 'float'>
```

'j' indica la parte immaginaria

Operazioni su dati numerici



In Python le operazioni sui dati numerici sono gestite dai seguenti operatori:

- Operatori Unari: -,+,~
- Operatori Binari: -,+,*,/,%,**
- Operatori Logici (solo su int e long int): and, or, xor
- Operatori bit a bit (solo su int e long int): &, | ,^

Esistono inoltre funzioni *built-in* per lavorare con dati numerici, tra cui:

-abs(number)

-pow(x, y[, z])

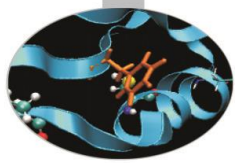
-round(number[, ndigits])

-coerce

Nell'eseguire operazioni tra variabili numeriche di diverso tipo viene seguita la seguente regola di conversione implicita:

Int→Long→Float→Complex

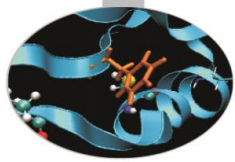
Operazioni su Dati Numerici



Esempio:

```
>>> k=5
>>> s=5+j
>>> type(s+k)          #conversione a numero complesso
<type 'complex'>
>>> 4 and 2           # confronto logico
2
>>> 4 & 2
0                     # confronto bit a bit tra i numeri binari 100 e 10
>>> 4 | 2
6                     # confronto bit a bit tra i numeri binari 100 e 10
```

Operazioni su Dati Numerici



Il modulo *math* fornisce alcune delle più comuni funzioni matematiche.

Il modulo *math* non lavora su dati numerici complessi, per i quali esiste lo specifico modulo *cmath*.

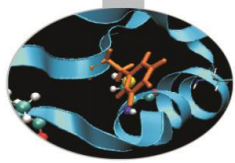
Le funzioni disponibili sono:

- Funzioni trigonometriche*: cos, sin, tan, asin, acos, atan, sinh, cosh, tanh.*
- Funzioni elevamento potenza e logaritmi: *pow*, *exp*, *log*, *log10*, *sqrt*
- Rappresentazione – trasformazione angoli *ceil* , *floor*, *fabs*, *degrees*, *radians.*

Nel modulo *math* sono inoltre definite le costanti numeriche *pi* ed *e.*

Le stesse funzioni sono disponibili per i numeri complessi nel modulo *cmath.*

Operazioni su Dati Numerici



Per utilizzare il modulo math è sufficiente importarlo con la seguente sintassi:

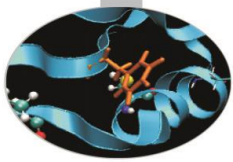
```
import math
```

Oppure

```
from math import *    (import all function)
from math import sin, cos   (import  specific function sinx, cosx)
```

```
>>>dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Bool Type



E’ possibile utilizzare i valori interi per rappresentare valori booleani con la convenzione che 0 corrisponda a *FALSE* e tutti i valori interi positivi corrispondano a *TRUE.*

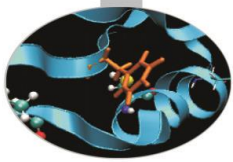
E’ tuttavia buona norma di programmazione utilizzare il tipo **bool** per rappresentare valori booleani. Una variabile di tipo *bool* può assumere valori *TRUE* e *FALSE* che possono essere interscambiati con i valori **1** e **0**

rispettivamente.

ESEMPIO

```
>>> a=1
>>> type(a)
<type 'int'>
>>> if(a):
        print 'True'
True
>>> a=False
>>> type(a)
<type 'bool'>
```

Stringhe



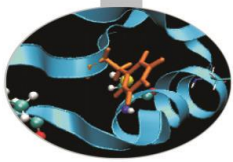
- Una stringa letterale è una sequenza di caratteri racchiusa tra doppi o singoli apici.

```
>>> a='ciao'
>>> b="Mondo"
>>> a+b
'ciaoMondo'
```

- Le sequenze di apici tripli “”” o ''' possono essere utilizzate per stringhe che spaziano su più righe, o che contengono apici singoli o doppi (o tripli dell'altro tipo):

```
>>> a""" Sono una stringa di più righe e
contengo apici 'singoli', "doppi" e
'''tripli'''"""
>>> print a
 Sono una stringa di più righe e
contengo apici 'singoli', "doppi" e
'''tripli'''
```

Stringhe



Per accedere al singolo carattere si può ricorrere all'operatore [] oppure ad una sottostringa con l'operatore [begin:end] (*slicing*)

Esempio

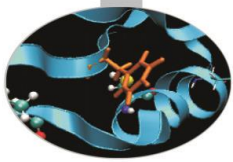
```
>>> a = "Hello world"
>>> a[1]
'e'
>>> a[1:3]
'el'
```

Non è possibile modificare il singolo carattere, ma è possibile assegnare alla stringa un nuovo valore

Esempio

```
>>> a='Primo valore'
>>> a = "Change value"  #Ok riassegnamento
>>> a[2] = '3'     #Errore
```

Manipolazione – Formattazione



Gli operatori + e * possono essere utilizzati per la manipolazione di stringhe. La precedenza tra gli operatori viene mantenuta.

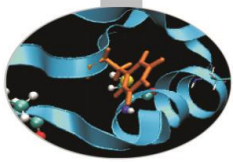
Esempio

```
>>> a = 'Hello'
>>> a+a+a              # Concatenazione
'HelloHelloHello'
>>> a = 'He'+'l'*2+'o World'      # Concatenazione e ripetizione
>>> a
'Hello World'
```

Una stringa è una sequenza di caratteri: lettere, numeri, simboli e sequenze di escape (o di controllo).

L'escaping permette di effettuare il quoting di un singolo carattere. Attraverso l'escaping è possibile aggiungere apici o altri caratteri all'interno di una stringa.

Manipolazione - Formattazione



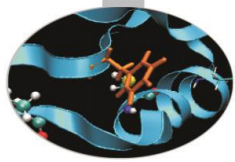
Esempio

```
>>> a = 'What's your name?'    #Errore
SyntaxError: invalid syntax
>>> a = "What's your name?"    # Ok con doppio apice
>>> a = 'What\'s your name?'   # Ok con escape sentence
```

Altri caratteri frequentemente utilizzati nella gestione delle stringhe sono:

- \t Tab — *'Ciao\tciao!'* → *Ciao ciao!*
- \n New Line — *'Ciao\nciao!'* → *Ciao*
  ciao!
- \\ Backslash — *'c:\\Programmi\\pp'* → *c:\Programmi\pp*
- \" Doppio apice — *'Repeat: \"Hello\"'* → *Repeat: "Hello"*
- \' Apice — *"Repeat:\'Hello\'"* → *Repeat: 'Hello'*
- \b Backspace — *"Hello \b World"* → *HelloWorld*

Manipolazione - Formattazione



Una stringa letterale preceduta da **r** o **R** viene detta *Raw String:* un carattere preceduto dal backslash viene incluso nella stringa senza cambiamenti.

Esempio

```
>>> a = r'Hello \t World'        #Raw string
>>> a
'Hello \t World'
```

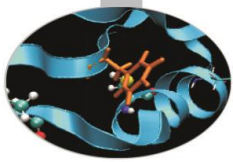
Anche in python è possibile formattare l'output. Il carattere % ha un significato speciale se usato con le stringhe.

Esempio

```
>>> "Oggi è %s %d %s" % ("Venerdì'',20,"Febbraio")
>>>print _
Oggi è Venerdì 20 Febbraio
```

Per la formattazione si possono utilizzare anche: %f, %c, %x, %o, %u, %%, %e.

Metodi Built-in



Le stringhe sono, come ogni entità in Python, oggetti e dispongono di una serie di funzionalità accessibili tramite dei metodi *built-in*.

- ***Manipolazione:*** concatenazioni, split, eliminazione caratteri e unioni.

  -split([sep [,maxsplit]])

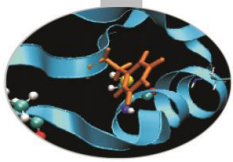
  -replace (old, new[, count])

  -strip([chars])

Esempio

```
>>> s='Ciao Mondo'
>>> s.split('o',1)
['Cia', ' Mondo‘]
>>> s.replace('o','i',1)
'Ciai Mondo‘
s.strip('C')
'iao Mondo '
```

Metodi Built-in



- ***Formattazione:*** allineamento, maiuscole, minuscole

  -center(width[, fillchar]) e ljust(width[, fillchar]) e rjust(width[,fillchar])

  -upper() e lower() e swapcase()

Esempio

```
>>> s = 'Hello'
>>> s.center(10,'.')
'..Hello...'
>>> s.upper()
'HELLO'
```

- ***Ricerca***: ricerca e interrogazione sulla stringa

  -find(sub [,start [,end]]) index(sub [,start [,end]])

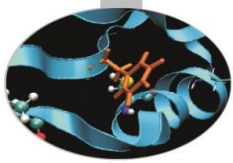
  -rindex(sub [,start [,end]]) e rfind(sub [,start [,end]])

  -count(sub[, start[, end]])

  -isupper() e islower()

  -startswith(prefix[, start[, end]]) e endswith(prefix[, start[, end]])

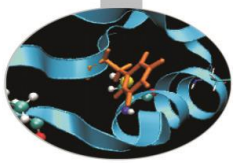
Metodi Built-in



Esempio

```
>>> s = 'Hello World'
>>> s.count('o ',0,5)
1
>>> s.rfind('k')
-1
>>> s.rindex('k')
Traceback (most recent call last):
File "<pyshell#62>", line 1, in <module>
s.rindex('k')
ValueError: substring not found
```

NOTA: Oltre ai metodi *built-in* esiste il modulo *string* per la manipolazione di stringhe. Alcune funzioni sono però già implementate come metodi nella classe *str*.



Operatori di confronto tra stringhe

E' possibile utilizzare gli operatori logici di confronto anche tra tipi di dato stringa.

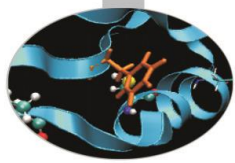
- **Operatori: <, <=, !=, >, >=, ==**

  agiscono seguendo l'ordinamento lessicografico e sono case-sensitive

Esempio

```
>>> a,b = 'Ciao','Mondo'
>>> a<b
True
>>> a = 'CIAO'; b = 'ciao'
>>> a<b
True
```

Confronti tra Stringhe



- **Operatori logici : and, or, not**

  Una stringa vuota è valutata *False* mentre qualsiasi stringa non vuota è valutata *True*.

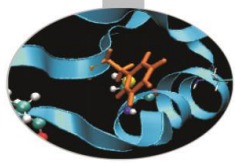
Le funzioni *min*, *max* e *cmp* possono essere utilizzate anche sulle stringhe.

Le stringhe sono inoltre sono dotate dell'operatore *in* e *not in*.

Esempio

```
>>> a = 'HelloWorld'
>>> min(a), max(a)
('H','o')
>>>'k' in a
False
```

Conversioni tra Tipi



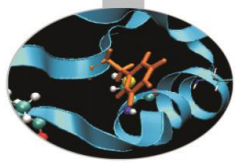
La conversione tra tipi nativi viene eseguita attraverso specifiche funzioni.

- **Conversione da tipo stringa → a tipo numerico :**
  - int(), float(), complex(), long()
  - ord()
  - eval()
- **Conversione da tipo numerico → a stringa:**
  - chr()
  - str()

Esempio

```
>>> a=97.6
>>> print ord(chr(int(a)))
97
>>> b=5+3j
>>> _+b
(102+3j)
```

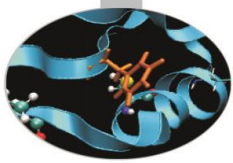
Esempio



Esempio

```
from math import *
g=9.81                  #m/s
v0=15                   #km/h
theta=60 #degree
x=0.5                   #m
y0=1                    #m
print """
Dati iniziali
v0=%f km/h
theta=%d gradi
y0=%f m
x=%f m
"""%(v0,theta, y0,x)
```

Esempio

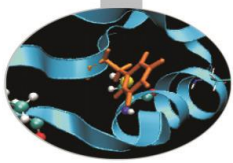


```
#conversione dati
v0=v0/3.6
thetha=degrees(thetha)
y=x*tan(theta) -1/(2*v0**2)*g*x**2/(cos(thetha)**2) + y0
print "Posizione finale =  ", str(y), 'm'
```

OUTPUT:

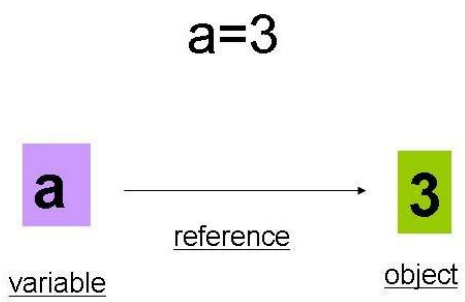
```
Dati
v0=15.000000
theta=60
x=0.500000
y0=1.000000
Posizione finale = 1.16002019469
```

Dynamic typing

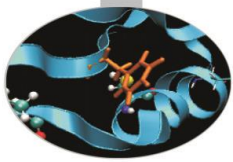


In Python i tipi sono individuati a runtime, senza necessità di dichiarazione all'interno del codice.

- Creazione di un variabile: la variabile viene creata quando le si assegna un valore
- Tipo di una variabile: il tipo è associato all'oggetto non alla variabile
- Uso di una variabile: quando una variabile appare in un espressione viene rimpiazzata dall'oggetto associato.



Reference Counter



Ciascun oggetto possiede un *type designator* e un *reference counter.*

```
>>>a=3
>>>a='ciao'
>>>a=1.5
```

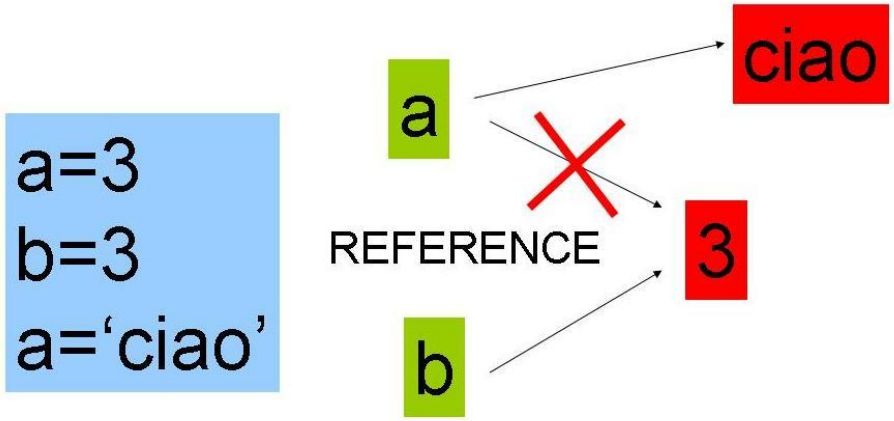
La variabile non ha un tipo associato.
Cambia solo la *reference*. Il tipo 'vive' con l'oggetto non con la variabile → ***type designator***

Ciascun oggetto ha associato un *reference counter*, un contatore al numero di referenze associate all'oggetto.

```
>>>a=3          Reference counter di 3 è pari a 1
>>>b=3          Reference counter di 3 è uguale a 2
>>>a=7          Reference counter di 3 è pari a 1, quello di 7 è pari
   a           1
```

Attraverso il reference counter viene implementato il *garbage collector* di Python. Importando il modulo *sys*, la funzione *sys.getrefcount(object),* ritorna il numero di reference di un oggetto.

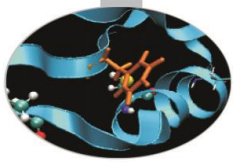
Reference Counter



In Python come vedremo esistono due tipologie di oggetti *mutable* e *immutable*. Gli oggetti *immutable* (interi,stringhe, float, etc etc) NON permettono modifiche *in-place*. Gli oggetti *mutable* permettono modifiche *in-place.*

```
>>> L=[1,2,3]
>>> L2=L
>>> L[0]=200
>>>L2
[200,2,3]
```

Dynamic Typing



Il dynamic typing è presente ovunque in Python: in assegnamenti, in argomenti passati alle funzioni, variabili nei cicli, import dei moduli etc.

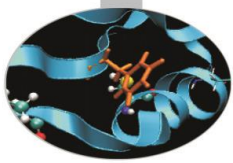
Il dynamic typing garantisce una grande riutilizzabilità del codice ed è alla base del polimorfismo in Python.

Polimorfismo significa avere diverse forme. Il comportamento di una funzione o di un'operazione dipende esclusivamente dal tipo di oggetto su cui sono applicate.

Esempio:

```
>>>a=3
>>>b=3*a                b=9
>>>a='ciao'
>>>b=3*a                b=ciaociaociao
```

Polimorfismo



Il polimorfismo garantisce caratteristiche di concisione e di flessibilità del linguaggio.

Essendo Python un linguaggio interpretato e con typing dinamico, il polimorfismo è intrinseco nel linguaggio.

Esempio:

```
def somma(a,b):
    return a+b
print somma(3,4)
print somma('ciao','  mondo')
print somma([1,2],[3,3])
```

OUTPUT

```
7
Ciao mondo
[1,2,3,3]
```

Il polimorfismo garantisce un'unica interfaccia per operare su tipi di dati differenti.