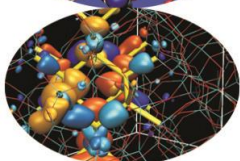
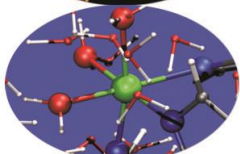
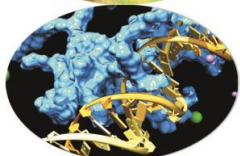
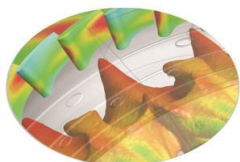
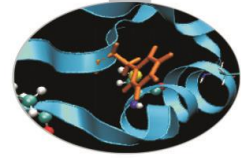


# Costrutti





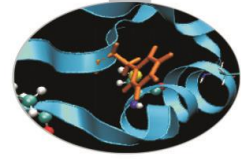
# Control Flow

Con *control flow* (o *strutture di controllo*) si intende l'insieme dei costrutti che servono a gestire il flusso di esecuzione di un programma, cioè a specificare quando e come eseguire gli statements del codice.

Nei più diffusi linguaggi di programmazione i costrutti variano ma possono essere classificati macroscopicamente in:

- *Costrutti condizionali*, l'esecuzione dipende dal raggiungimento di certe condizioni.
- *Costrutti iterativi*, gli statements vengono eseguiti zero o più volte.
- *Costrutti non locali*, gestione delle eccezioni.
- *Costrutti fondamentali*, funzioni, metodi.

Python in quanto linguaggio *imperativo* dispone dei più comuni costrutti di controllo. L'indentazione del codice fornisce l'accorpamento dei comandi in *blocchi* di statements, al posto delle parentesi usate nei più tradizionali linguaggi di programmazione, p.e. C/C++.

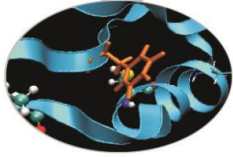


# If Statement

Il costrutto condizionale *if* ha la seguente sintassi:

```
if condition1:  
    block 1  
elif condition2:  
    block 2  
    ...  
elif conditionN:  
    block N  
else:  
    block statements
```

*If* è un costrutto condizionale: se la condizione è soddisfatta viene eseguito il blocco di istruzioni corrispondente altrimenti si analizzano le condizioni di verità dei blocchi *elif*.



# If Statement

Il blocco *if* può avere più blocchi *elif*.

*elif* è la versione abbreviata di *else if*. Il costrutto *elif* è opzionale.

*else* corrisponde al caso di default, come *elif* è opzionale.

In un *if statement* al più viene eseguito un unico blocco (*suite*) di istruzioni.

## Esempio

```
>>> x=2000
```

```
>>> if x<10:
```

```
    print 'Piccolo'
```

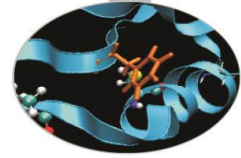
```
elif x<100:
```

```
    print 'Medio'
```

```
else:
```

```
    print 'Grande'
```

```
'Grande'
```



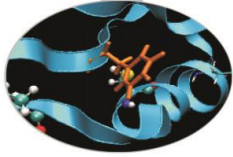
# If Statement

L'espressione condizionale che deve essere verificata utilizza i comuni operatori logici che il linguaggio mette a disposizione:

- Comparazione:  $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $==$ ,  $!=$
- Identità: *is*, *is not*
- Membership: *in*, *not in*
- Logical: *not*, *and*, *or*

## Esempio

```
if answer=='copy':  
    copyfile='file.tmp'  
elif answer=='run' or answer=='execute':  
    run=True  
elif answer=='quit':  
    quit=True  
else:  
    print "No Valid Answer "
```



# Switch Case in Python

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

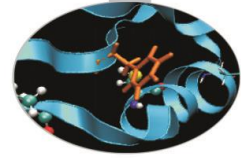
Python non dispone del costrutto condizionale switch, la gestione di espressioni condizionali con  $N$  sottocasi viene gestita con il costrutto *if-elif-else*.

Python ha un'alternativa più semplice ed espressiva per il costrutto *switch-case* presente in altri linguaggi di programmazione.

In molti linguaggi come in C/C++ il costrutto *switch -case* può gestire solo dei valori interi. La soluzione implementata in Python permette di gestire il costrutto *case* in modo generale.

Il funzionamento del costrutto *switch-case* può essere emulato utilizzando opportunamente gli array associativi – dictionary.

```
{key1: f1, key2: f2,...keyN: fN}
```



# For Statement

Il costrutto *for* è un costrutto iterativo che viene utilizzato per ripetere una suite di istruzioni un numero definito di volte.

La sintassi:

```
for <variable> in <sequence>:  
    block
```

Python può eseguire dei loop su qualsiasi tipo di sequenza di dati: tuple, liste, stringhe.

## Esempio

```
>>> s = 'Ciao'
```

```
>>> for i in s:
```

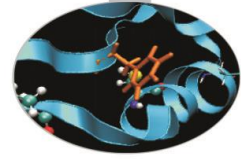
```
    print s
```

```
C
```

```
i
```

```
a
```

```
o
```



# For Statement

Per iterare su una sequenza di interi, Python dispone della funzione built-in `range([start,] stop[, step])` che restituisce una lista di interi .

Per ottimizzare l'efficienza del codice nei loop è preferibile utilizzare la funzione `xrange([start,] stop[, step])`.

Il vantaggio di `xrange` risiede nell'utilizzo della memoria: `xrange` richiede sempre la stessa quantità di memoria indipendentemente dalla dimensione del range di valori da rappresentare.

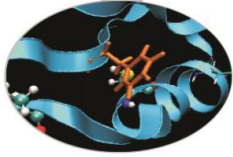
## Esempio

```
x=1000000000
t1=time.clock()
for el in range(x):
    pass
t2=time.clock()
print "Time ", t2-t1, "s"
Time 6.66 s
```

**VS**

```
x=1000000000
t1=time.clock()
for el in xrange(x):
    pass
t2=time.clock()
print "Time ",t2-t1, "s"
Time 1.9 s
```





# While Statement

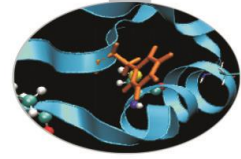
Il costrutto *while* permette di eseguire un blocco di istruzioni un numero indefinito di volte finché una particolare espressione risulta *True*.

La sintassi generale:

```
while (condition):  
    block
```

L'espressione *condition* viene valutata ad ogni ciclo e il ciclo viene ripetuto finché *condition==True*

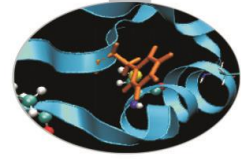
In Python non esiste il costrutto *do-while*. Il ciclo *while* viene eseguito almeno una volta se la condizione di partenza risulta *True*.



# While Statement

## Esempio

```
import random
number=random.randint(0,10)
guess=False
while(not guess):
    s=input('Inserisci un intero: ')
    if s==number:
        print 'Complimenti! Numero segreto indovinato'
        guess=True
    elif s<number:
        print 'Numero deve essere più grande'
        guess=False
    elif s>number:
        print 'Numero deve essere più piccolo'
        guess=True
```



# Continue - Break

L'esecuzione di un loop può essere interrotta qualora siano verificate condizioni eccezionali.

L'istruzione *break* interrompe l'esecuzione del loop. Per i cicli annidati, *break* interrompe esclusivamente l'esecuzione del loop più interno.

L'istruzione *continue* salta all'iterazione successiva di un ciclo. Nel caso di cicli annidati, *continue* agisce sul ciclo più interno.

**for** <variable> **in** >iterable:

statements1

**for** <variable2> **in** <iterable>:

**if** (condition2):

**continue**

**elif** (condition):

**break**

statements2

**while**(condition):

statements1

**if**(condition):

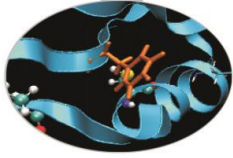
**continue**

**elif**(condition2):

**break**

statements2

# Continue - Break



**Esempio- break:** calcolo dei numeri primi fino ad n

```
n=raw_input('Inserisci un numero intero ')
```

```
n=int(n)
```

```
for k in range(3,n+1):
```

```
    flag=1
```

```
    print 'k', k
```

```
    for i in range(2,k):
```

```
        print 'i',i
```

```
        if k%i == 0:
```

```
            flag=1
```

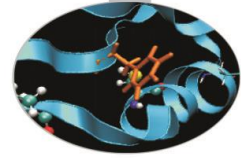
```
            break
```

```
    flag=0
```

```
if flag==0:
```

```
    print 'Numero primo:', k
```

# Continue - Break



## Output

Inserisci un numero intero 6

k 3

i 2

Numero primo: 3

k 4

i 2

k 6

i 2

k 5

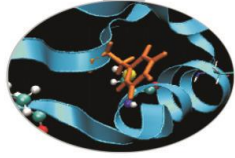
i 2

i 3

i 4

Numero primo: 5

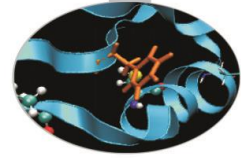
# Continue - Break



**Esempio:** calcolo della distanza minima tra un punto  $p_1$  e un insieme di punti nel primo quadrante

```
from math import *
from pylab import*

lista=[(2,3),(4,5),(1,7),(-1,6),(-2,-4),(2,-8),(1,1)]
p1=input('Inserisci la coordinata lungo x: ')
p2=input('Inserisci la coordinata lungo y: ')
point=(p1,p2)
c_point=()
dist_min=0
cur_dist=-1
dist=10000000
```



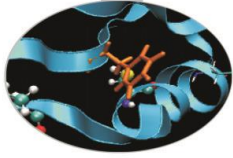
# Continue - Break

```
x=[]  
y=[]  
for k in range(len(lista)):  
    x.append(lista[k][0])  
    y.append(lista[k][1])  
    if lista[k][0]<0 or lista[k][1]< 0:  
        continue  
    cur_dist=sqrt((lista[k][0]-point[0])**2+(lista[k][1]-point[1])**2)  
    if cur_dist==0:  
        print 'Punti coincidenti'  
        break  
    if cur_dist<dist:  
        dist=cur_dist  
        c_point=lista[k]
```

**Jump**

**Break**

# Continue - Break



```
print 'Distanza minima : ', dist
print 'Punto di minimo: ', c_point

plot(x,y,'o',[point[0]],[point[1]],'ro')
```

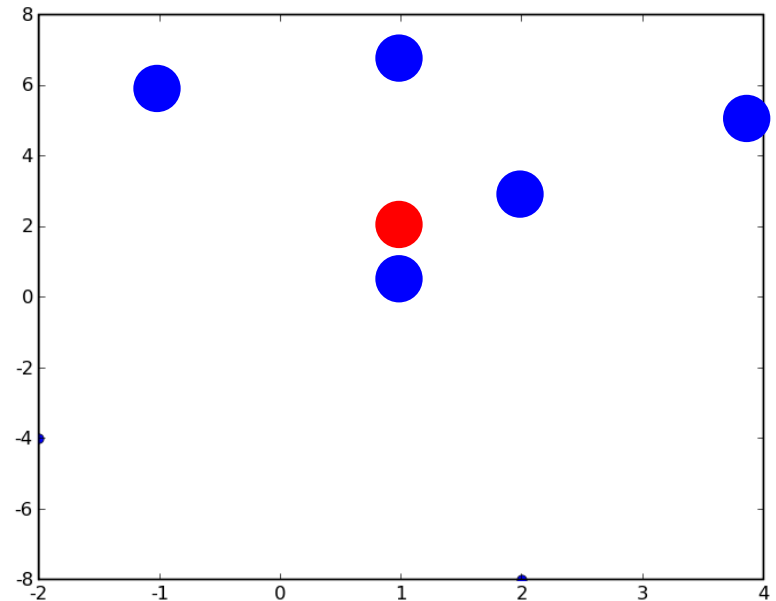
## #OUTPUT

Inserisci la coordinata lungo x: 1

Inserisci la coordinata lungo y: 2

Distanza minima : 1.0

Punto di minimo: (1, 1)







# Pass

- Pass è un'istruzione generica che 'non fa niente'. Può essere inserita in qualsiasi parte del codice.
- E' utile in situazioni in cui è necessario uno statement per ragioni sintattiche.
- Esempio il corpo di una funzione non può essere omissso.

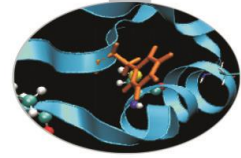
```
>>> def f(a,b):
```

```
    pass
```

```
>>>a=3
```

```
>>>b=4
```

```
>>>f(a,b)
```



# For While & Else Clause

Ai cicli condizionali *for* e *while* può essere associato un *else* statement, con la seguente sintassi:

<pre>for &lt;variable&gt; in &lt;sequence&gt;:     block1 else:     block2</pre>	<pre>while(condition1):     block1 else:     block2</pre>
----------------------------------------------------------------------------------------------	-----------------------------------------------------------------------

Il blocco *else* viene eseguito al termine dell'esecuzione dei rispettivi cicli a patto che questi non siano terminati con l'istruzione *break*.

## Esempio

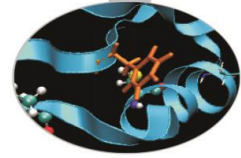
```
n2=1000
```

```
step=5
```

```
print "Number n1 is in range ( %d , %d) with step %d?"%(0,n2,step)
```

```
number=int(raw_input("Insert number n1: "))
```

# For While & Else Clause



```
for i in range(0,n2,step):  
    if i==n1  
        print 'n1 IS in range'  
        break  
else:  
    print 'n1 IS NOT in range'
```

## **#OUTPUT:**

Number n1 is in range ( 0 , 1000) with step 5

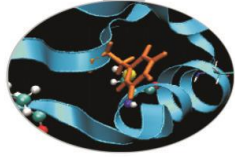
Insert n1: 77

n1 IS NOT in range

Insert n1: 55

n1 IS in range

# Gestione delle Eccezioni



La gestione delle eccezioni è uno strumento che permette di gestire errori che tipicamente si verificano durante l'esecuzione di un programma.

Questo strumento permette di aumentare la robustezza di un programma e prevenire crash che comprometterebbero la normale esecuzione del codice.

Ci sono almeno due tipologie di errore: errori sintattici ed eccezioni, generati quando la sintassi è corretta ma viene prodotto un errore a *run-time*.

## Esempio:

```
>>> for i in range(10) print 'Ciao'
```

```
SyntaxError: invalid syntax
```

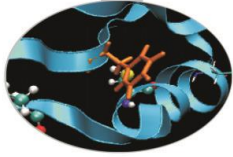
```
>>> a = 10/0
```

```
Traceback (most recent call last):
```

```
File "<pyshell#11>", line 1, in <module>
```

```
  a=10/0
```

```
ZeroDivisionError: integer division or modulo by zero
```



# Try - Except

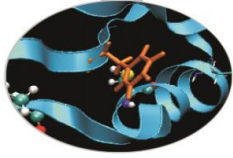
Il costrutto *try-except* permette di rilevare le eccezioni e passare il controllo di flusso ad un blocco preposto: *exception handler*, con la seguente sintassi:

```
try :  
    block1  
except:  
    block2
```

In questo modo viene eseguito il blocco *try* e se viene generato un errore l'esecuzione prosegue nel blocco *except*. Qualsiasi errore viene gestito nel blocco *except* a meno di specificare più blocchi *except* ognuno dedicato alla gestione di una particolare tipologia di errori.

## Esempio:

```
>>>s=input('Inserisci un intero: ')  
>>>a=s/10 #Ma se s=0 o s=stringa?
```



# Try - Except

#Soluzione

```
t=False
```

```
while(not t):
```

```
    try:
```

```
        s=input('Inserisci un numero: ')
```

```
        a=10.0/s
```

```
        print a
```

```
        t=True
```

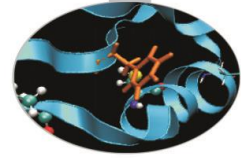
```
    except NameError:
```

```
        print 'Inserisci un numero'
```

```
        t=False
```

```
    except ZeroDivisionError:
```

```
        print 'Divisione per zero!'
```



# Try - Except

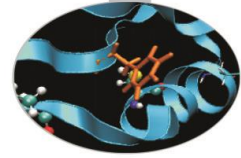
Un *except* clause può gestire più tipologie di eccezioni, che vengono specificate attraverso una tupla di valori:

```
except (exception1, exception2,...,exceptionN)
```

Il costrutto try-except può avere un else-clause opzionale.

```
try:  
    block1  
except e1:  
    block2  
...  
except eN:  
    blockN  
else:  
    blockN+1
```

L' *else-clause* viene eseguito quando il costrutto *try* non genera eccezioni.



# Try - Except

Il blocco *except* ammette anche la sintassi:

*except ExceptionName, e :*

Le informazioni aggiuntive riguardo l'oggetto che hanno causato l'eccezione vengono salvate in e.

## Esempio:

try:

```
s=input("Inserisci un numero: ")
```

```
a=10.0/s
```

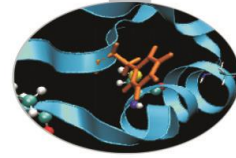
except NameError, e:

```
print "Errore ", e
```

Inserisci un intero: stringa

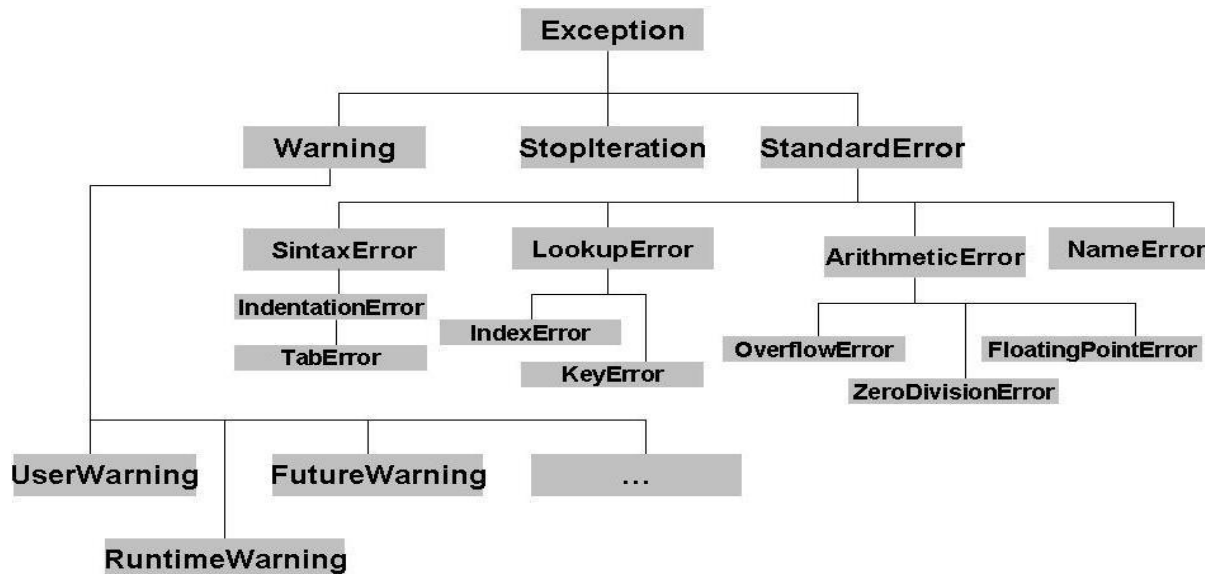
no name 'stringa' is not defined





# Try - Except

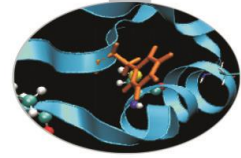
L'ordine con il quale appaiono gli *handlers* delle eccezioni, all'interno di un costrutto try-except, è rilevante: dalla più specifica alla meno specifica.



**NOTA:** Non è necessario importare esplicitamente il modulo exception.

Oltre alle eccezioni *built-in* è possibile per l'utente creare nuove eccezioni.

Il meccanismo di generazioni di eccezioni *user-defined* è legato alla programmazione ad oggetti: qualsiasi nuova eccezione deve ereditare dalla classe base *Exception*.



# Raise Exception

E' possibile forzare la generazione di un'eccezione attraverso lo statement *raise*.

Lo statement *raise* ha la seguente sintassi:

```
raise ExceptionName,Argument
```

Il primo argomento indica il nome dell'eccezione *built-in* o *user-defined*.

**Esempio** - derived Exception:

try:

```
a,b,c=input('a,b,c: ')
```

```
if a==0:
```

```
    raise Exception, 'Impossibile, a=0'
```

```
if b*b-4*a*c<0:
```

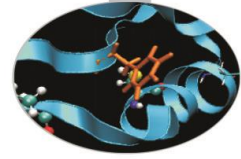
```
    raise Exception, 'b^2-4ac <0, negativo, non ci sono radici reali'
```

```
x1= (-b+math.sqrt(b*b-4*a*c))/(2*a)
```

```
x2= (-b-math.sqrt(b*b-4*a*c))/(2*a)
```

```
print "Le radici sono", x1, x2
```

# Raise Exception



```
except Exception,e:
```

```
    print e
```

```
#OUTPUT:
```

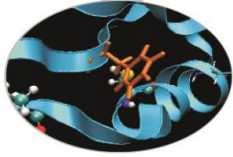
```
a,b,c: 2,1,0
```

```
Le radici sono 0.0 -0.5
```

```
a,b,c: 0,2,1
```

```
Impossibile, a=0
```

# Raise Exception



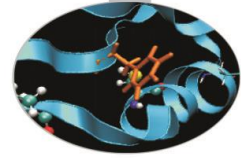
**Esempio:** raise-else clause

#Verificare se un elemento esiste in una matrice

```
found=0
```

```
for x in range(len(mat)):  
    for y in range(len(mat)):  
        k+=1  
        if mat[x][y]==s:  
            found=1  
            break  
    if found: break  
if found==1:  
    print 'Trovato alla '+str(k)+'-esima iterazione'  
else:  
    print 'Non Trovato'
```

# Raise Exception



**#Oppure**

try:

```
for x in range(len(mat)):
```

```
    for y in range(len(mat)):
```

```
        k+=1
```

```
        if mat[x][y]==s:
```

```
            msg='Trovato alla' +str(k)+ '-esima iterazione'
```

```
            raise Exception, msg
```

```
except Exception,msg:
```

```
    print msg
```

```
else:
```

```
    print 'Non trovato'
```



# Finally Statement

Il costrutto *try-except* può essere integrato con lo statement opzionale *finally*.

Il costrutto *finally* serve a racchiudere un blocco di istruzioni che devono essere eseguite in ogni caso, sia che venga o non venga generata un'eccezione.

Se in un blocco *try* viene generata un' eccezione non gestita da un *exception handler*, l'eccezione viene rilanciata dopo la *finally* clause.

## Esempio:

```
a=raw_input('Inserisci un numero: ')
```

```
b=raw_input('Inserisci un secondo numero: ')
```

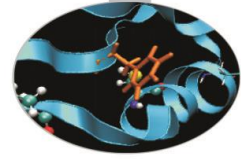
```
try:
```

```
    d=int(a)/int(b)
```

```
except ZeroDivisionError,e:
```

```
    print 'Divisione per zero:', e
```

# Finally Statement



else:

```
print 'Divisione: ',d
```

finally:

```
print 'Esecuzione di finally'
```

## #OUTPUT:

Inserisci un numero: 5

Inserisci un secondo numero: 0

Divisione per zero: integer division or modulo by zero

Esecuzione di finally

Nella pratica è utile l'utilizzo dello statement *finally* per controllare il rilascio di risorse esterne p.e. chiudere un file, una connessione di rete, chiudere una connessione con un database, ect. anche se il codice è andato in *crash*.

## NOTA:

Il costrutto try-except-finally è supportato a partire dalle versioni 2.5.x