INDICE

Come si e' visto in precedenza, l'interprete funziona come un semplice calcolatore, si digita un'espressione + invio e si ottiene il risultato.
La sintassi che utilizzeremo in principio e' immediata: sono disponibili i principali operatori matematici e logici presenti negli altri linguaggi e si possono utilizzare le parentesi per raggruppare le espressioni.

```
In [2]: 2+2
```

```
Out[2]: 4
```

```
In [3]: (50-5*7)/4
```

```
Out[3]: 3.75
```

INTERI (int, non-mutable)

creare interi

Si inizializzano con numeri, NON SEGUITI DA PUNTO

In [4]:
```
a = 2
b = 1+1 #come risultato di un'operazione
c=2.0 #e' un numero reale
print(type(1))
print(type(b))
print(type(c))#!!
```

```
<class 'int'>
<class 'int'>
<class 'float'>
```

dinamica virtualmente infinita

- Gli interi in Python hanno dinamica infinita!!!
- L'interprete gestisce dietro le quinte il numero di bit necessario a rappresentare l'intero.

In [1]:
```
a=2**1000 # l'operatore ** rappresenta l'elevazione a potenza
#print(a)
```

mi chiedo quante cifre abbia 2 alla millesima potenza ....

In [6]:
```
sa=str(a) #trasformo a in una stringa
print('a ha',len(sa),'cifre') #la funzione print accetta come argomenti, stringhe,
                              #numeri interi e altri tipi predefiniti
```

```
a ha 302 cifre
```

operatori matematici

addizione, sottrazione,moltiplicazione e elevazione a potenza

gli operatori: addizione, sottrazione,moltiplicazione e potenza si comportano come e' prevedibile ...

In [7]:
```
1+2  #addizione
```

Out[7]:
```
3
```

In [8]:
```
6-10 #sottrazione
```

Out[8]:
```
-4
```

In [9]:
```
5*4  #moltiplicazione
```

Out[9]:
```
20
```

In [10]:
```
2**3 #potenza
```

Out[10]:
```
8
```

Modulo

In [11]:
```
print(13%9) #modulo: restituisce il resto della divisione
print(type(13%9))
```

```
4
<class 'int'>
```

divisione

divisione vera

```
In [12]: print(10/4  , type(10/4))#divisione con resto
```

```
2.5 <class 'float'>
```

divisione intera (troncata)

```
In [13]: print(10//4 , type(10//4))#divisione senza resto
```

```
2 <class 'int'>
```

Riassumendo sulla divisione in Python 3:

- L'operatore / mantiene il resto e in generale restituisce un numero reale.
- L'operatore // non mantiene il resto e restituisce il quoziente (eventualmente arrotondato per difetto) come intero

Definizione di un intero con notazione ottale, binaria ed esadecimale

Oltre alla classica notazione in base 10 e' possibile definire un numero intero usando anche:

- notazione binaria ('Ob', zero-bminuscolo)
- notazione ottale ('Oo', zero-ominuscolo)
- notazione esadecimale ('Ox', zero-xminuscolo)

```
In [14]: a=0b10  #notazione binaria
         print(a,type(a))
```

```
2 <class 'int'>
```

```
In [15]: a=0o10 #notazione ottale
         print(a)
```

```
8
```

```
In [16]: a=0xF #notazione esadecimale
         b=0x10
         print(a)
         print(b)
```

```
15
16
```

Conversione di base di un intero

Oltre alla possibilita' di definire un intero usando base binaria, ottale ed esadecimale e' possibile ottenere la rappresentazione in una diversa base usando le funzioni builtin bin, oct ed hex che restituiscono delle stringhe rispettivamente con la rappresentazione binaria, ottale ed esadecimale di un intero

Gli esempi sotto usano anche la funzione int che restituisce un intero a partire da una stringa o piu' in generale da un altro oggetto di tipo compatibile.

Rappresentazione da decimale a binaria, ottale, esadecimale:

```
In [8]: print('rappresentazione di 35:')
        print('base 2:', bin(35))
        print('base 8:',oct(35))
        print('base 16:',hex(35))
```

```
rappresentazione di 35:
base 2: 0b100011
base 8: 0o43
base 16: 0x23
```

da binario:

```
In [9]: print('rappresentazione di 0b100011:')
```

```
print('base 10:',int('0b100011',base=2))
print('base 8:',oct(0b100011))
print('base 16:',hex(0b100011))
```

```
rappresentazione di 0b100011:
base 10: 35
base 8: 0o43
base 16: 0x23
```

da esadecimale:

In [10]:
```
print('rappresentazione di 0x23:')
print('base 10:',int('0x23',base=16))
print('base 8:',oct(0x23))
print('base 2:',bin(0x23))
```

```
rappresentazione di 0x23:
base 10: 35
base 8: 0o43
base 2: 0b100011
```

Operatori bitwise

esistono anche operatori che agiscono sui singoli bit.
Per comprenderne il significato e' opportuno lavorare con la rappresentazione binaria:

definisco due interi i e j

In [2]:
```
i = 0b11010011
j = 0b11101100
print('i=',i,bin(i))
print('j=',j,bin(j))
```

```
i= 211 0b11010011
j= 236 0b11101100
```

and

l'operatore and bit a bit si indica col carattere &

In [21]:
```
#and
iandj = bin(i&j)

print('Operazioni bit a bit sui numeri',i,' e ',j,':')
print(bin(i),' &')
print(bin(j),' =')
print('-'*13)
print(iandj)
print(i&j)
```

```
Operazioni bit a bit sui numeri 211  e  236 :
0b11010011  &
0b11101100  =
-------------
0b11000000
192
```

or

In [22]:
```
#or
iorj  = bin(i|j)

print(bin(i),' |')
print(bin(j),' =')
print('-'*13)
print(iorj)
print(i|j)
```

```
0b11010011  |
```

```
0b11101100  =
-------------
0b11111111
255
```

or esclusivo

In [23]:
```
#xor
ixorj  = bin(i^j)

print(bin(i),' ^')
print(bin(j),' =')
print('-'*13)
print(ixorj)
```

```
0b11010011  ^
0b11101100  =
-------------
0b111111
```

inversione dei bit, complemento a 1

In [24]:
```
#inversione dei bit (complemento a 1)
tildei=~i

print('~',bin(i),' =')
print('-'*15)
print('',bin(tildei), '(rappresentazione con segno di un numero binario in complemento a due)')
```

```
~ 0b11010011  =
---------------
 -0b11010100 (rappresentazione con segno di un numero binario in complemento a due)
```

Per verificare l'effetto dell'operazione di complemento a 1 uso il modulo non standard bitstring

In [25]:
```
import bitstring

print('~',bin(i),' =')
print('-'*15)
print('  ',bitstring.BitArray(int=tildei,length = len(bin(tildei))-2).bin,
            '(stringa di bit, bit di segno compreso)')
```

```
~ 0b11010011  =
---------------
   100101100 (stringa di bit, bit di segno compreso)
```

shift a destra e a sinistra

In [26]:
```
#shift a dx
idx = i>>3
#shift a sx
isx = i<<2

print()
print()
print(bin(i),' >> 3 =')
print('-'*18)
print(bin(idx))
print()

print()
print(bin(i),' << 2 =')
print('-'*18)
print(bin(isx))
print()
```

```
0b11010011  >> 3 =
```

```
------------------
0b11010


0b11010011  << 2 =
------------------
0b1101001100
```

NUMERI REALI in virgola mobile (float, non-mutable)

- Il tipo `float` rappresenta numeri reali in virgola mobile.
- Tutorial floating point.

Istanziare un numero reale

In [27]:
```
a=2.1
print(type(a))
```

```
<class 'float'>
```

Si puo' utilizzare anche la notazione esponenziale (dove e sta per 10**):

In [28]:
```
a=2.34e6
b=2.34*10**6
c=5e-8
print(a)
print(b)
print(a==b)
```

```
2340000.0
2340000.0
True
```

- Per chi ha esperienza in C/c++ o in altri linguaggi compilati: non c'e' distinzione tra singola e doppia precisione.
- Il massimo valore rappresentabile dipende dalla piattaforma utilizzata.

In [29]:
```
#queste informazioni sono accessibili grazie al modulo sys
#ed alla struttura float_info in particolare
import sys
print(sys.float_info)
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308,
min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

se volete avere informazioni aggiuntive ...

In [9]:
```
#help(sys.float_info)
```

Numeri speciali

Infinity

In [37]:
```
a = float('inf')
type(a)
```

Out[37]:
```
builtins.float
```

In [38]:
```
a = 1.2e600
a
```

Out[38]:
```
inf
```

In [39]:
```
a+1
```

Out[39]: inf

In [40]:
```
1/a
```

Out[40]: 0.0

In [41]:
```
a/a
```

Out[41]: nan

NaN

In [43]:
```
b = float('nan')
type(b)
```

Out[43]: builtins.float

In [44]:
```
b = 1./b
b
```

Out[44]: nan

In [45]:
```
b+1
```

Out[45]: nan

riconoscere nan e inf

il modulo math mette a disposizione delle funzioni per riconoscere nan e inf

In [46]:
```
import math
a=1e700
math.isinf(a)
```

Out[46]: True

In [47]:
```
math.isnan(a/a)
```

Out[47]: True

operatori matematici

- I numeri float supportano ovviamente i principali operatori matematici (+,-,/,**)
- Nel prossimo esempio oltre a provare vari operatori, useremo la funzione (builtin) eval che restituisce il valore dell'espressione Python passata come stringa alla funzione stessa.

In [58]:
```
a = 2.0
b = 7.3
#definisco le operazioni come stringhe
c = 'a+b' #addizione
d = 'a-b' #sottrazione
e = 'a*b' #moltiplicazione
f = 'a/b' #divisione
g = 'a//b'#divisione intera
h = 'a**b'#elevazione a potenza
i = 'pow(a,b)'#elevazione a potenza con funzione pow
l = '-a'  #negazione
m = 'abs(-a)'  #valore assoluto con funzione abs
n = 'b%a' #resto
#uso la funzione eval per 'stampare' i risultati delle operazioni definite come stringhe:
```

```
for esp in(c,d,e,f,g,h,i,l,m,n):
    print(esp , '=' ,eval(esp))
```

```
a+b = 9.3
a-b = -5.3
a*b = 14.6
a/b = 0.273972602739726
a//b = 0.0
a**b = 157.58648490814926
pow(a,b) = 157.58648490814926
-a = -2.0
abs(-a) = 2.0
b%a = 1.2999999999999998
```

Ecco una tabella riassuntiva dei vari operatori:

Operation	Result
`x + y`	sum of *x* and *y*
`x - y`	difference of *x* and *y*
`x * y`	product of *x* and *y*
`x / y`	quotient of *x* and *y*
`x // y`	floored quotient of *x* and *y*
`x % y`	remainder of `x / y`
`-x`	*x* negated
`+x`	*x* unchanged
`abs(x)`	absolute value or magnitude of *x*
`int(x)`	*x* converted to integer
`float(x)`	*x* converted to floating point
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.
`c.conjugate()`	conjugate of the complex number *c*
`divmod(x, y)`	the pair `(x // y, x % y)`
`pow(x, y)`	*x* to the power *y*
`x ** y`	*x* to the power *y*

- Il modulo math mette a disposizione le funzioni matematiche per i numeri float definite dal linguaggio C in *math.h*.
- Al solito per sapere quali sono le funzioni e le grandezze definite nel modulo math avete diverse opzioni:
  - help(math)
  - dir(math)
  - in IDLE, se, dopo aver importato math, digitate `math.` e poi premete il tasto TABULAZIONE, compare un menù a tendina con elencate tutte le funzioni disponibili nel modulo

In [32]:
```
import math
a='math.pi'
b='math.e'
c='math.sin(math.pi/2.)'
d='math.log(math.e)'
e='math.log10(10.**3.45)'

for esp in(a,b,c,d,e):
    print(esp , '=' ,eval(esp))
```

```
math.pi = 3.141592653589793
math.e = 2.718281828459045
```

```
math.sin(math.pi/2.) = 1.0
math.log(math.e) = 1.0
math.log10(10.**3.45) = 3.45
```

Float e rappresentazione binaria

E' necessario tenere conto che i numeri float in Pytyon (e più in generale in un calcolatore) sono rappresentati in base 2 (binary64).

Ad esempio il numero decimale 0.125 puo' essere rappresentato dalla seguente somma di frazioni: *[Math Processing Error]* dove i denominatori sono potenze progressive del 10.

Lo stesso numero puo' essere rappresentato anche in termini di somma di frazioni il cui denominatore sia una potenza del due: *[Math Processing Error]* Di conseguenza la rappresentazione binaria del numero decimale 0.125 e': *[Math Processing Error]*

Sfortunatamente non tutte le frazioni decimali possono essere rappresentate esattamente in termini di frazioni binarie e di conseguenza quando vogliamo rappresentare un numero decimale con un float il numero binario floating point che viene memorizzato nella macchina e' una approssimazione del numero decimale di partenza:

In [33]:
```
a=0.1
print(a)
```

```
0.1
```

anche se Python mostra come output 0.1 cio' che e' memorizzato non e' 1/10!
A partire da Python 2.7 e 3.1 l'interprete mostra la più breve rappresentazione decimale che e' rappresentata dal numero binario memorizzato nelle versioni precedenti veniva utilizzata una rappresentazione con 17 cifre decimali (vedi figura sotto che illustra cosa succedeva con python 2.6)

```
Python 2.6.2 (r262:71605, Apr 14 2009, 22:40:02) [MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.

    **********************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    **********************************************************************

IDLE 2.6.2
>>> 0.1
0.10000000000000001
>>>
```

In [66]:
```
a=0.1
#in realta' cio' che e' memorizzato e' diverso
#da 1/10, per verificarlo basta forzare
#la rappresentazione di un sufficiente
#numero di cifre decimali
print("%.25f"%a)
#print("{0:.25f}".format(a))
```

```
0.1000000000000000055511151
```

Quindi e' necessario ricordarsi che, sebbene normalmente Python visualizzi 0.1 in realta' cio' che e' memorizzato e' il più vicino numero rappresentabile come frazione binaria.
Questo si vede bene nel seguente esempio nel quale si istanziano 3 numeri che hanno la stessa rappresentazione binaria.

In [35]:
```
a=0.1
b=0.10000000000000001
c=0.1000000000000000055511151231257827021181583404541015625

print('-'*10)
print(a)
print(b)
print(c)

print('-'*10)
print(a==b)
print(a==c)
print(b==c)
```

```
----------
```

```
0.1
0.1
0.1
----------
True
True
True
```

E' da notare quanto detto riguarda la 'natura' della rappresentazione binaria floating point e che NON SI TRATTA DI UN BACO DI Python.
Cio' puo' essere sperimentato in tutti i linguaggi che supportano l'aritmetica floating point (sebbene in alcuni di questi sia relativamente più difficile evidenziare le differenze di rappresentazione binaria e decimale).

In relazione a quanto detto e' possibile capire perche' sia molto 'pericoloso' basarsi su test di uguaglianza tra float:

In [36]:
```
0.1+0.1+0.1==0.3
```

Out[36]: False

il modulo decimal

Il modulo decimal "provides support for decimal floating point arithmetic."
In particolare il modulo decimal definisce un tipo Decimal con le seguenti caratteristiche:

- il numero è memorizzato non in codifica binaria ma come una sequenza di cifre decimali
- ogni numero e' rappresentato da una numero fissato di cifre
- si tratta comunque di un numero floating point cioe', dato il numero di cifre, la virgola puo stare ovunque

creare un decimal

da float

se creo un decimal da un float ottengo una copia del float in questione

In [73]:
```
import decimal
a=decimal.Decimal(0.1)
print(a)
```

```
0.1000000000000000055511151231257827021181583404541015625
```

In [67]:
```
a+a+a
```

Out[67]: 0.30000000000000004

da stringa

Se invece passo una stringa

In [35]:
```
b=decimal.Decimal('0.1')
b
```

Out[35]: Decimal('0.1')

da toupla

In [40]:
```
#segno(0=+),cifre,esponente
c=decimal.Decimal((0, (0, 1), -1))
c
```

Out[40]: Decimal('0.1')

somma di float vs somma di decimal

somma di float

In [41]:
```
0.1 + 0.1 + 0.1 - 0.3
```

Out[41]: 5.551115123125783e-17

somma di decimal

In [42]:
```
a=decimal.Decimal('0.1')
b=decimal.Decimal('0.3')
a+a+a-b
```

Out[42]: Decimal('0.0')

Precisione

ricordiamoci comunque che si tratta di numeri decimali con precisione finita:

In [51]:
```
#modifico la precisione
decimal.getcontext().prec=4
A=decimal.Decimal(1)/decimal.Decimal(3)
print(A)
```

```
0.3333
```

In [52]:
```
#modifico la precisione
decimal.getcontext().prec=10
A=decimal.Decimal(1)/decimal.Decimal(3)
print(A)
```

```
0.3333333333
```

in ogni caso:

In [53]:
```
A+A+A
```

Out[53]: Decimal('0.9999999999')

In [54]:
```
A+A+A == decimal.Decimal('1.0')
```

Out[54]: False

per approfondimenti su decimal vedi qui

Il modulo fractions

anche i numeri razionali hanno il loro modulo standard

In [55]:
```
import fractions
F=fractions.Fraction(1,3)
print(F)
```

```
1/3
```

e quindi:

In [57]:
```
b=F+F+F
print(b)
```

```
1
```

complex (non-mutable)

Tra i tipi standard di Python ci sono anche i numeri complessi.

Creare un numero complesso

Un numero complesso con parte reale *r* e parte immaginaria *i* puo' essere creato con la notazione

r+ij

la parte immaginaria e' scritta con il `suffisso j o J`

oppure con la funzione

complex(r,i)

In [38]:
```
#inizializzazione di un numero complesso
a = 1.0  + 1.5j # suffisso minuscolo
b = 2.0  + 2.5J # suffisso maiuscolo
c = 3.5J + 3.0  # prima la parte immaginaria
d = complex(4.0,4.5) # funzione 'costruttore' con argomenti passati per posizione
e = complex(imag=5.5,real=5.0) # funzione 'costruttore' con argomenti passati per nome

print(type(c))
print(a,b,c,d,e)
```

```
<class 'complex'>
(1+1.5j) (2+2.5j) (3+3.5j) (4+4.5j) (5+5.5j)
```

Operatori

In [39]:
```
a = 1.0  + 1.5j
b = 2.0  + 2.5J
#provare a 'giocare' con i vari operatori matematici
e1 = 'a+b'
e2 = 'a-b'
e3 = 'a*b'
e4 = 'a/b'
e5 = 'a**b'
#valuto le espressioni ei con i che va da 1 a 5
#mediante la funzione eval
#per capire questo costrutto tenere presente che:
# - ei e' il nome di una stringa
# - eval(ei) e' una stringa che rappresenta una espressione
# - eval(eval(ei)) e' il risultato dell'espressione descritta dalla stringa eval(ei)
for i in range(1,6):
    esp = 'e%d'%i
    print(eval(esp),'=',eval(eval(esp)))
```

```
a+b = (3+4j)
a-b = (-1-1j)
a*b = (-1.75+5.5j)
a/b = (0.5609756097560976+0.0487804878048781j)
a**b = (-0.2662834597647353-0.08158796335770331j)
```

membri

- *real*: (attributo) restituisce la parte reale
- *imag*: (attributo) restituisce la parte immaginaria
- *conjugate()*: (funzione) restituisce il complesso coniugato

In [40]:
```
a = 1.0  + 1.5j
#senza parentesi
a.real
```

Out[40]: 1.0

In [41]:
```
a.imag
```

Out[41]: 1.5

In [42]:
```
a.conjugate()
```

Out[42]: (1-1.5j)

abs ed il modulo cmath

abs

per determinare il valore assoluto di un numero complesso usare la funzione builtin abs

In [43]:
```
a = 1. - 1.j
abs(a)
```

Out[43]: 1.4142135623730951

cmath.phase

il modulo cmath mette a disposizione le funzioni del modulo math per i numeri complessi. Una tra queste e' la funzione phase che restituisce la fase in radianti

In [1]:
```
import cmath
a = 1. - 1.j
cmath.phase(a)*180./cmath.pi
```

Out[1]: -45.0

creare complessi con cmath.rect(argomento,fase)

In [2]:
```
import math
import cmath
#nota: cmath.sqrt restituisce un numero complesso
#      cmath.rect si aspetta come argomenti due numeri reali
a = cmath.rect(math.sqrt(2),-45*cmath.pi/180.)
a
```

Out[2]: (1.0000000000000002-1j)

rappresentazione polare con cmath.polar

restituisce modulo e fase in radianti

In [3]:
```
cmath.polar(a)
```

Out[3]: (1.4142135623730951, -0.7853981633974482)

altre funzioni di cmath

In [4]:
```
cmath.sin(a)
```

Out[4]: (1.2984575814159776-0.6349639147847359j)

In [5]:
```
(e**(a*1j)-e**(a*-1j))/(2j)
```

Out[5]: (1.2984575814159776-0.6349639147847359j)

$$\text{sen } y = \frac{e^{iy} - e^{-iy}}{2i} \qquad \cos y = \frac{e^{iy} + e^{-iy}}{2}$$

In [6]:
```
cmath.cos(a)
```

Out[6]: (0.8337300251311488+0.9888977057628652j)

In [7]:
```
(e**(a*1j)+e**(a*-1j))/(2)
```

Out[7]: (0.8337300251311488+0.9888977057628652j)

math o cmath?

le funzioni definite in cmath restituiscono comunque un numero complesso (anche con argomento reale e positivo):

In [48]:
```
import cmath
cmath.sqrt(3.)
```

Out[48]: (1.7320508075688772+0j)

D'altra parte se uso math e faccio la radice di un numero negativo ...

In [50]:
```
import math
#math.sqrt(-3)
```

il modulo numpy.lib.scimath (disponibile per chi ha installato il modulo numpy) definisce delle funzioni 'universali' che trattano in modo unificato numeri reali e complessi

In [54]:
```
import numpy.lib.scimath
numpy.lib.scimath.sqrt(3)
```

Out[54]: 1.7320508075688772

In [55]:
```
numpy.lib.scimath.sqrt(-3)
```

Out[55]: 1.7320508075688772j

In [74]:
```
print(dir(numpy.lib.scimath))
```

```
['__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__name__', '__package__',
'_fix_int_lt_zero', '_fix_real_abs_gt_1', '_fix_real_lt_zero', '_ln2', '_tocomplex', 'any', 'arccos',
'arcsin', 'arctanh', 'asarray', 'isreal', 'log', 'log10', 'log2', 'logn', 'nt', 'nx', 'power', 'sqrt']
```

bool

In [51]:
```
e=False
type(e)
```

Out[51]: builtins.bool

- Il tipo bool e' rappresentato da due costanti `False` e `True`
- In contesto di calcoli (per esempio se utilizzate con operatori aritmetici) si comportano come gli interi 0 (False) e 1 (True).

In [52]:
```
True + True
```

Out[52]: 2

In [53]:
```
10*False
```

Out[53]: 0

operatori logici (parametri bool, restituiscono bool)

- gli operatori logici accettano bool come operandi e restituiscono un bool come risultato:

Operation	Result	Notes
`x or y`	if *x* is false, then *y*, else *x*	(1)
`x and y`	if *x* is false, then *x*, else *y*	(2)
`not x`	if *x* is false, then `True`, else `False`	(3)

Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is `False`.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is `True`.
3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

In [54]:
```
a=True
b=False

a or b
```

Out[54]: True

In [55]:
```
a and b
```

Out[55]: False

In [56]:
```
not a
```

Out[56]: False

In [57]:
```
#l'operatore xor non esiste
#in Pyton ma puo' essere facilmente
#implementato con l'operadore disuguaglianza !=
#che vedremo anche nel seguito
a != b #equivalente a (a xor b)
```

Out[57]: True

concatenamento

Gli operatori logici possono essere concatenati ma attenzione alle precedenze:

In [58]:
```
a = True
b = False
c = False
```

In [59]:
```
a or b and c
# and viene calcolato prima di or
# a or (b and c)
```

Out[59]: True

In [60]:
```
#se vogliamo che sia eseguito
(a or b) and c
```

Out[60]: False

operatori di comparazione (accettano parametri di vari tipi, restituiscono bool)

- gli operatori di comparazione accettano come argomento oggetti di vario tipo e restituiscono un bool

In [61]:
```
2 < 2
```

Out[61]: False

In [62]:
```
2 <= 2
```

Out[62]: True

In [63]:
```
2*3 == 6
```

Out[63]: True

In [64]:
```
2. * 3.2 == 6.4
```

Out[64]: True

In [65]:
```
0.1 * 3. == 0.3 # questo dovrebbe ricordarvi qualcosa !!!
```

Out[65]: False

concatenamento

anche gli operatori di comparazione possono essere concatenati:

dati

In [76]:
```
a = 1
b = 2
c = 3
d = 1
```

e' possibile scrivere:

In [67]:
```
a < b < c
```

Out[67]: True

che e' equivalente a:

In [68]:
```
a < b and b < c
```

Out[68]: True

con la differenza che nella prima forma b viene 'valutato' una volta sola

Un altro esempio di concatenamento:

In [78]:
```
a < b < c > d
```

Out[78]: True

Quindi anche la precedente e' una espressione valida e non implica alcuna comparazione tra
a e d
o tra
b e d

Conversioni tra tipi numerici

In [70]:
```
#conversione int -> float
float(1)
```

Out[70]: 1.0

In [71]:
```
#conversione float -> int
int(1.2)
```

Out[71]: 1

In [72]:
```
#conversione int - > complex
```

```
complex(1)
```

Out[72]: (1+0j)

In [73]:
```
#conversione int -> bool
bool(0)
```

Out[73]: False

In [74]:
```
#conversione int -> bool
bool(1) == (bool(1) == bool(33))
```

Out[74]: True

Ci sono anche conversioni non permesse, ad esempio se provo a convertire un complex in un float Python mi segnala un `TypeError`:

In [75]:
```
#float(6.2+5.1j)
```

PILLOLE DI SINTASSI: la definizione di funzioni e l'indentazione

Vediamo come fare a definire una funzione senza approfondire.
La sintassi e' molto semplice:

def (): return #NB: la fine della funzione e' determinata dal livello di indentazione del codice

Come si vede il corpo della funzione è `indentato` ad un livello diverso rispetto alla sua definizione.

Più in generale l'indentazione definisce il `livello` del codice.

Definisco due funzioni che restituiscono il quadrato ed il cubo di un numero:

In [76]:
```
def quadrato(val):
    print('Nella funzione quadrato('+str(val)+')')
    return val**2
#sono tornato al livello di indentazione del 'def'
#il blocco di codice che definisce la funzione e' terminato

#definisco una nuova funzione
def cubo(val):
    print('Nella funzione cubo('+str(val)+')')
    return val**3
```

In [77]:
```
print(quadrato(10))
```

```
Nella funzione quadrato(10)
100
```

In [78]:
```
print(cubo(10.))
```

```
Nella funzione cubo(10.0)
1000.0
```

Se passo una stringa alla funzione ...

In [79]:
```
#quadrato('pippo')
```

Posso definire funzioni che accettano più parametri

In [80]:
```
def retta(x,a,b):
    """
Questo e' un esempio di "docstring"
ordinata della retta con pendenza a, intercetta b per ascissa pari a x.

Parameters:
===========
    x: (number) ascissa
    a: (number) pendenza
    b: (number) intercetta
```

```
    Returns:
    ========
        a*x + b

    Examples:
    =========
        vretta = [retta(x,10.,20.) for x in range(10)]
    """
        return a*x+b
```

In [81]:
```
print(retta(2.54,10.,20.))
```

```
45.4
```

Questo lo vedremo nella prossima lezione:

In [82]:
```
print([retta(x,10.,20.) for x in range(10)])
```

```
[20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0, 110.0]
```

PILLOLE DI SINTASSI: l'istruzione if

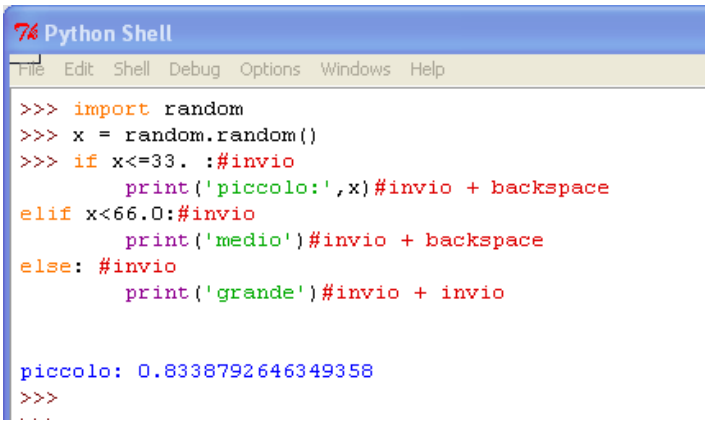
La sintassi della proposizione if e' molto semplice:
if : ... elif : ... elif : ... else: .... #NOTA: la fine dell'ultima istruzione e' determinata # dal livello di indentazione del codice
vediamo un esempio:

In [83]:
```
#generazione di un numero casuale
import random
x=100.*random.random()

if x <= 33.0:
    print('piccino: ',x)
elif x<=66.:
    print('medio: ',x)
else:
    print('grande :',x)
```

```
medio:  61.06033903515961
```

Nell'uso interattivo con IDLE si deve fare attenzione all'indentazione:



Altri link utili

- Ordine di valutazione delle espressioni
- Precedenza operatori
- Comparazione in Python
- Tutorial docs.python.org
- Tipi di base in Python