Python Essential Summary

case sensitive

commento fino a fine riga
""" commento su piu linee
(una multiline string)
"""

#!/usr/bin/python prima riga per eseguibili Unix
 # -- coding: utf-8 -- default UTF-8

__**doc**__ : docstring ad inizio oggetto, mostrata da help(oggetto)

__**dir(oggetto)**__ : lista attributi oggetto

; separa istruzioni (opzionale)

**** per continuazione nella riga successiva.
Espressioni fra parentesi: anche piu' righe

linee vuote : ignorate ; spazi compattati

indentazione definisce blocchi
: inizio blocco

Variabili: *reference to objects* **id(var)** :address: **type(var)** :tipo

caratteri, numeri, _

primo carattere: alfabetico,

nomi riservati iniziano con _

Oggetti:

- **immutabili** : caratteri, stringhe, numeri, tuple, booleani, frozen sets, bytes
- **mutabili** : liste , dizionari, sets, byte array

__**doc**__ docstring ad inizio oggetto, mostrata da help(oggetto)

Keywords:

```
False      await      else       import     pass
None       break      except     in         raise
True       class      finally    is         return
and        continue   for        lambda     try
as         def        from       nonlocal   while
assert     del        global     not        with
async      elif       if         or         yield
match      case       _
```

Separatori:

separatore	Funzione
( )	racchiudono elementi di una tupla, chiama funzione
[ ]	racchiudono elementi di una lista, indici di sequenze
{ }	racchiudono elementi di un dizionario o set
;	separano istruzioni sulla stessa linea
` `	trasformano una variabile in una stringa (solo Python 2)
" "	racchiudono stringhe (**anche contenenti apici singoli**)
' '	racchiudono stringhe (**anche contenenti apici doppi**)

Tipi: classi sono tipi

- **int** : numero arbitrario di cifre (python3) (32 bit Python2 ed ha tipo long)
  - 0o, 0O: ottali: oct(x): visualizza in ottale
  - 0x, 0X : hex: hex(x): visualizza in esadecimale
  - 0b, OB : binary: bin(x): visualizza in binario
- **float** : doppia precisione, Es.: 3.14 10. .001 1e100 -3.14e-10 0e0
- **str** : stringhe; sono sequenze immutabili di caratteri (da Python 3 in codifica UTF-8)
- **bytes** : valore immutabile fra 0-255 Es.: a=bytes([1,2,3]) : b'x01x02x03x04'. Sono le vecchie stringhe del Python2
- **byte array** : mutabile: a=bytearray([1,2,3])
- **complessi** :Es.: 3.14e-10j
- **bool** : **True**, **False**, 0: Falso, oggetto vuoto: falso
- **None** : assenza di un valore: e' falso.
- **list** : a=[0,1,2,3,4] ; a[0] : eterogenee, mutabili, nestable
- **tuple** : a=(0,1,2,3,4) : immutabili,
- **dict** : D={'key1':3,'key2':6,1:'abc'} ; D['key1']mutabili, keys: hashables
- **set** : mutabili, insiemi di immutabili, in unica copia: a={1,2,3}
- **frozenset** : set immutabili : a=frozenset([1,2,3])
- **range** : range(start,end_escluso,step) : range(1,5,2) => 1,3

Alcuni tipi sono in moduli della libreria standard

- **decimali** e **frazionari**
- **enum** : da Python 3.4, tipo le enum del C
- **OrderedDict** : dizionari ordinati (modulo collection)
- **named tuple** : modulo collection

Operatori

Operatori aritmetici

Operatore	Funzione	Esempi
**	Elevamento a potenza	a**3 ; anche funzione pow(x,y)
*	Moltiplicazione	a*b ; 3*2 => 6
/	Divisione (genera float)	a/b ; 5/2 => 2.5
//	Divisione intera	a//b ; 5//2.0 => 2.0
%	Resto	a%b ; 5%2. => 1.0
+	Addizione	a+b ; 2+5 => 7
-	Sottrazione	a-b ; 2-5 => -3

Operatori bit a bit

Operatore	Funzione	Esempi
<<	shift a sinistra	a<<1 ; 8<<1 fornisce 16
>>	shift a destra	a>>1 ; 8>>1 fornisce 4
&	and sui bit	a&b ; 2&1 fornisce 0
|	or sui bit	a|b ; 2|1 fornisce 3
^	or esclusivo	a^b ; 2^3 fornisce 1
~	inverte i bit	~a ; ~0 fornisce -1

Operatori di assegnazione

Operatore	Funzione	Esempi
=	assegna riferimento	a=3
del(a)	elimina una variable assegnata	a=3 ; del(a) ; a=>NameError
*=	moltipilca ed assegna	a*=3 ; equivale ad a=a*3
/=	divide ed assegna	a/=3 ; equivale ad a=a/3
+=	somma ed assegna	a+=3 ; equivale ad a=a+3
-=	sottrae ed assegna	a-=3 ; equivale ad a=a-3
//=	divisione intera ed assegna	a//=3 ; equivale ad a=a//3
%=	resto ed assegna	a%=3 ; equivale ad a=a%3
**=	potenza e assegna	a**=3;equivale ad a=a**3
&=	or bitwise ed assegna	a&=1 ; equivale ad a=a&1
|=	and bitwise ed assegna	a|=1 ; equivale ad a=a|1
^=	not bitwise ed assegna	a^=3 ; equivale ad a=^a
>>=	bit right shift ed assegna	a>>=3 ; equivale ad a=a>>3
<<=	bit left shift ed assegna	a<<=3 ; equivale ad a=a<<3

Unrolling ed assegnazioni multiple

Operazione	Commento
a,b = b,a	scambio dei valori di a e b
a,b,c=1,2,3	assegna i numeri, in sequenza alle 3 variabili
a=1,2,3	a diventa una tupla di 3 oggetti
c,d,e=a	funziona se a e' una tupla o lista di 3 oggetti
a,b=1	provoca errore
a=b=3	sia a che b sono riferimenti all'oggetto '3'

Solo Python 3, assegnazioni miste: a variabili e liste:

```
a,*b = 1,2,3,4        # in b finisce la lista [2,3,4]
a,*b, c = 1,2,3,4     # in b finisce la lista [2,3]  , 4 va in c
```

Operatori logici

Operatore	Funzione	Esempi
>	maggiore	a > b
<	minore	a < b
<=	minore od eguale	a <= b
>=	maggiore od eguale	a >=b
==	eguale	a == b
!=	diverso (<> solo python2)	b != b
2 < 5 < 6 ; 2<=3 >1 ; gli operatori si combinano		

Operatore	Funzione	Esempi
in	vero se x compare in s	x in s
not in	vero se x non compare in s	x not in s
is	vero se x ed y sono lo stesso oggetto	x is y
is not	vero se non sono lo stesso oggetto	x is not y

Operatore	Significato	Esempio
or	or logico	x or y
and	and logico	x and y
not	negazione	not x

```
x or y      Vale x se x e' True,  altrimenti valuta y e restituisce y
x and y     Vale x se x e' False, altrimenti valuta y e restituisce y

x or  y or  z     restituisce il primo vero  (o l'ultimo)
x and y and z     restituisce il primo falso (o l'ultimo)
```

Da Python 2.5 sono possibili espressioni condizionali, tipo l'peratore ternario del C

```
a = espressione_1 if condizione else espressione_2
a=  32+5/2  if b>0   else  322.9
```

Conversioni : esistono conversioni automatiche;

Operatore	Funzione	Esempi
abs(x)	Valore assoluto	abs(-3) produce: 3
divmod(x,y)	divisione e resto	divmod(5,2) produce: (2, 1)
int(x)	muta in intero	int('3'), int(3.2) producono : 3
float(x)	muta in float	a='3' ; float(a) produce: 3.0
complex(r,i)	crea numero complesso	complex(3,2) produce: (3+2j)
c.conjugate()	complesso coniugato	(3+2j).conjugate() da: (3-2j)
round(x,n)	arrotonda, a n decimali	round(3.12345,2) da: 3.12
bin(x)	rappresentazione binaria	bin(2) da: '0b10'
oct(x)	rappresentazione ottale	oct(8) da: '0o10'
hex(x)	rappresentazione esadecimale	hex(16) da: '0x10'
str(x)	muta in stringa	str(2.345) da: 2.345
repr(x)	rappresenta oggetto come stringa	in python 2 anche: `oggetto`

```
import math
math.trunc(x) : tronca ad interi
math.floor(x) : approssima all'intero piu' piccolo
math.ceil(x)  : approssima all'intero piu' grande
```

Operazioni per sequenze

Operatore	Funzione	Esempi
in	vero se x compare in s	x in s
not in	vero se x non compare in s	x not in s
s + t	concatena sequenze	[1,2]+[4,5] da:[1,2,4,5]
s * n	ripete la sequenza	[1,2]*3 da: [1,2,1,2,1,2]
s[i]	elemento numero i	
s[i:j]	elementi da i a j, j escluso	
s[i:j:k]	da i a j con passo k, k negativo va all'indietro	
len(s)	numero elementi nella sequenza	
min(s)	minimo elemento nella sequenza	
max(s)	massimo elemento nella sequenza |	
s.count(x)	conta quante volte x e' nella sequenza	
s.index(x,i,j)	posizione di x nella sequenza, cercando fra posizioni i e j	
map(f,s)	applica la funzione f alla sequenza producendo una lista	

Slicing (sottosequenze):

```
a='0123456'

a[0]        vale '0'
a[1]        vale '1'
a[-1]       vale '6'
a[-2]       vale '5'

a[:] a[0:]  sono tutta la stringa
a[:0]       e' la stringa vuota

a[:3]      vale '012'
a[3:]      vale '3456'

a[0:2]     vale: '01'
a[1:3]     vale: '12'
a[:-1]     vale: '012345'
a[-3:-1]   vale: '45'
a[-1:-3]   vale '' (la stringa vuota)

a[::-1]    vale  '6543210'  (ribalta la stringa)

a[0:5:2]   vale '024'    (da 0 a 5, passo 2 )

a[1::2] vale: '135' , si va dall'elemento 1 alla fine della stringa
                      prendendo un elemento si ed uno no.

a[1::3] vale:  '14'    si prende un elemento ogni 3.

a[-1:-5:-2] vale '64' : passo negativo va all'indietro.
a[-1:-3:1]   vale '' :la stringa vuota:  il passo e' +1 e' in avanti.
                      e' come: a[-1:-3]
```

Operazioni per sequenze mutabili

Operazione	Effetto
s[i] = x	modifica elemento i della sequenza
s[i:j] = [t]	modifica elementi da i a j (j escluso),t e' una lista
del s[i:j:k]	elimina elementi, gli indici dei restanti elementi cambiano
s.append(x)	aggiunge elemento in fondo, come: s[len(s):len(s)] = t
s.clear()	come del s[:] , svuota la sequenza
ss=s.copy()	fa una copia della sequenza s
s.insert(i, x)	come s[i:i]=[x], gli elementi seguenti cambiano di indice
s.pop(i)	estrae l'elemento i e lo elimina, pop() estrae l'ultimo
s.remove(x)	elimina il primo elemento di valore x che trova
s.reverse()	ribalta la sequenza
s.sort()	ordina la sequenza, modificandola

Stringhe:

```
a='12345"6789'        # apici doppi fra semplicie e viceversa
a="12345'6789"

a='abcd' 'efg'    -->   a== "abcdefg" # compatta stringhe

a=''

''' questa stringa
continua in questa riga '''


a=r"12\n4g"  --> raw string non interpreta: "\"


\\    :    e' il carattere "\" (Backslash)
\'    :    apice singolo '
\"    :    doppio apice "
\a    :    e' un suono di allarme       (ASCII Bell (BEL)  )
\b    :    indica indietro un carattere (ASCII Backspace (BS) )
\f    :    indica che si va a capo      (ASCII Formfeed (FF) )
\n    :    indica che si va a capo      (ASCII Linefeed (LF))
\r    :    indica che si va a capo      (ASCII Carriage Return (CR))
\t    :    tabulazione orizzontale      (ASCII Horizontal Tab (TAB))
\v    :    tabulazione verticale        (ASCII Vertical Tab (VT))

\0   :    carattere nullo

\xhh    : carattere in notazione esadecimale ( hh sono le cifre esadecimali)
\ooo    : carattere in notazione ottale ( ooo sono le cifre ottali)

Per inserire unicode:

\uxxxx      : ove  xxxx sono cifre  esadecimali
\Uxxxxxxxx  : ove  xxxxxxxx sono cifre  esadecimali
\N{name}    : nome unicode del carattere

a=r"12\n4g"    #  raw string non interpreta //

a=b'abcd'      # stringa di bytes, non UTF-8
```

Operatore di formattazione per le stringhe: %

	Rappresentazione valore
%s	come stringa (usa funzione str())
%r	come stringa (usa funzione repr()
%10s	stringa, occupa minimo 10 spazi
%-10s	10 spazi, ma allineato a sinistra
%c	singlo carattere
%5.1f	float in 5 spazi, una cifra decimale
%5d	decimale, in 5 spazi
%i	intero
%08i	intero, in 8 spazi, ma spazi iniziali con degli zeri

```
a='%s  %s %.2f' % (42,'stringa',1/3.0)   produce:  '42  stringa 0.33'
a='%10d %-10s'%(12.3,'AAA')              produce:  '        12 AAA       '
```

Formattazione tramite format():

```
"3 valori {1} {0} {nom}".format("primo",'secondo',nom=123)
'3 valori secondo primo 123'

" 2 numeri {1:10d} {0:10.2f}".format(123.456,77)
' 2 numeri         77     123.46'

"{0!s}".format(123.4567)
'123.4567'

"{0!r:>10s}".format(123.4567)
'  123.4567'
```

	Significato
!s	muta in stringa (usa funzione __str__()) !r usa __repr__()
:> :< :^	allineano a desta , a sinistra, al centro
:%M/%d/%Y	per oggetti datetime
:d :f :e	decimali, float , con esponente, come per %()
:g	float o esponente a seconda della grandezza

f-string: stringhe con arbitrarie espressioni Python fra {} (da Python 3.6)

```
name = 'gigi'
num=123
a=F'Hello {name}, Your number is {num*2}'
'Hello gigi, Your number is 246'

num=124.3
f"{num!s:>10s}"
'     124.3'
```

Template string:

```
from string import Template
student = 'Anna'
template = Template('Hello $name!')
a=template.substitute(name=student)

'Hello Anna!'
```

Precedenza degli operatori

```
creazione dizionari: { } ; creazione liste [] ;  creazione tuple( ) ,
funzioni , slicing, indici
riferimenti ad attributi di oggetti
esponente: **
operatori "unary" : +x, -x , ~x
operatori numerici:  / // % * - +
operatori sui bit: << >>  & ^ |
operatori logici: == != <> < <= > >=
Operatori di appartenenza: is   in
operatori logici: not and or
generazione funzioni lambda (che vedremo dopo)
```

Statements

print(*objects,sep=' ',end='n',file=None,flush=False)

If statement:

```
if a==b:
   c=d
   e=f
elif a>b:
   c=f
   e=d
else:
   c=0
   e=0

if a==b: k=0     # su una riga se una sola istruzione
```

With statement:

```
with open("x.txt") as f:
    data = f.read()
    ....
```

While statement:

```
while x:
   i+=1
   x-=1
else:            # eseguito in ogni caso (salvo break)
   y=i
```

Alterano i loop:

```
break      # interrompe il ciclo
continue   # passa al giro successivo
pass       # non fa nulla;  while 1:pass  # e' un loop infinito
```

For statement:

```
for i in [1,2,3,4]:
   k+=i
else:
   print('fine')   # eseguito sempre se non c'e' break


D={'a':0,'b':1,'c':3}
for i in D:            # loop su chiavi
   print(i)

for (key, value) in D.items():  # loop su tupla
   print(key, '=>', value)

T = [(1, 2), (3, 4), (5, 6)]  # loop su tupla
for (a, b) in T:
     print(a, b)

for i in range(3):    # produce la sequenza: 0,1,2
     print(i)

for i,a in enumerate(['a','b','c']): # tupla: numero,valore
     k[i]=a                         #  (1,'a') (2,'b') (3,'c')
```

List comprehensions:

```
[x for x in range(5) if x % 2 == 0]
[x + y for x in range(3) for y in [10, 20, 30]] #  [10, 20, 30, 11, 21,

D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}

D = {x: x ** 2 for x in [1, 2, 3, 4]}       # {1: 1, 2: 4, ....

D = {c: c * 4 for c in 'SPAM'}              # {'S': 'SSSS', 'P': 'PPPP',..
```

Funzioni specifiche

Conversioni:

```
int(a)     e' il numero 123456, analogamente str(450) e' la stringa '450'
float(a)   muta una stringa in un numero in virgola mobile.
str(1.6E3) e' la stringa '1600.0'

ord('A')   da 65, codice ascii del carattere
chr(65)    fornisce 'A'

bin(2)     produce: '0b10'
hex(16)    produce: '0x10'
oct(8)     produce: '0o10'
```

Funzioni su iterabili:

```
Z = zip((1, 2, 3), (10, 20, 30))

list(Z) vale:  [(1, 10), (2, 20), (3, 30)]
dict(Z) vale:     {1: 10, 2: 20, 3: 30}
tuple(Z) vale:  ((1, 10), (2, 20), (3, 30))

list( filter((lambda x: x>0 ),[1,2,3,-1]) )  produce [1, 2, 3]

all(s) :  True se tutti gli elementi sono veri

any(s) : True se qualche elemento e' vero

sum(s) : somma, per dizionari somma gli indici

min(s) : minimo, per dizionari minimo degli indici

max(s) : massimo per dizionari massinmo degli indici

sorted(s) : lista ordinata (per dizionari lista di keys ordinate)
sorted(s,key='len',reverse=True)

map(funzione,s)       : applica funzione, elementi iterabile come ergomenti
map(funzione,s1,s2)   : 2 argomenti, dai 2 iterabili
```

Operatori e funzioni per le stringhe

```
f1,f2,f3='abc'  # unrolling

a+b             # concatena stringhe

b*2             # ripete stringhe

'0'  in "01234"  # True

for i in 'abcd' :
    print(i)

'substring' in 'string'     : True  se e' substring, o False
'substring' not in 'string' : False se e' substring, o True

len('0123456')  : e' '7' : il numero di caratteri della stringa

min('0123456')  : e' '0': il carattere piu' piccolo (nella sequenza dei caratteri ASCII)
max('0123456')  : e' '6': il carattere piu' grande nella seq. ASCII
max('abcd')**   : e' 'd'

'abcd'.index('c')   e': 2 : l'indice del carattere 'c' nella stringa

'abcd'.capitalize(): e' la stringa: 'Abc' (primo carattere maiuscola)
'abcd'.upper()     : e' la stringa: 'ABCD' ( muta  i caratteri in maiuscolo)
'abcd'.lower()     : muta i caratteri in minuscolo
'abc def'.title()  : ogni parola capitalized

'abcd'.replace('a','X')   : e' la stringa: 'Xbcd' , cambia il carattere a in X
'abcd'.replace('a','X',3) : effettua la sostituzione per le prime 3 sottostringhe
                                che trova (di default fa la sostituzione per tutte)

split :  crea una lista con le parti della stringa:

         '1,2,3'.split(',')  # produce ['1', '2', '3']
         ' a   b '.split()     #produce: ['a', 'b']

join :   unisce piu' stringhe ; attributo della stringa separatore:

         '-'.join(['1','2','3'])      #  produce:  '1-2-3'


    stringa.rstrip()     : elimina \n alla fine

stringa.isalpha()    : True se contiene solo caratteri alfabetici
    stringa.isnumeric()  : True se e' un numero
    stringa.islower()    : True se caratteri minuscoli
    stringa.isupper()   : True se caratteri maiuscoli

    stringa.expandtabs(4)          : mette 4 spazi al posto dei tab

    stringa.ljust(width,fillchar)  : allinea la stringa a sinistra
    stringa.rjust(width,fillchar)  : allinea la stringa  a destra

    stringa.startswith(stringa2) : True se inizia con la stringa2
    stringa.endsswith(stringa2)  : True se finisce con la stringa2
```

Liste:

```
A=[0,1,2,['a','b','c'],'fgk',3.4E10]] : lista nested: lista come elemento

   A[3]==['a','b','c']
   A[3][1]=='b'

A=[]         : crea una lista vuota ed un riferimento 'A' alla lista.
A=['asdfg']  : lista di un solo elemento (una stringa)
               Qui A[0][0] e' il carattere 'a'

A=list((1,2,3)) : crea lista da una sequenza, qui la tupla: (1,2,3)

A[0]        : primo elemento
A[-1]       : ultimo elemento (numeri negativi iniziano a contare dalla fine)
A[-2]       : penultimo elemento
A[:] A[0:]  : sono tutta la lista
A[:0]       :    e' la lista vuota
A[:3]       : elementi dal primo al terzo (quarto, con indice 3, escluso)
A[3:]       : elementi dal quarto in poi
A[1:5]      : dal secondo al quinto elemento
A[1:5:2]    : dal secondo al quinto con passo 2
A[-1:-5:-2] : dall'ultimo al quint'ultimo, con passo 2

A[0]=1.3   : sostituisce il primo elemento con un valore dato (qui 1.3)
A[0:3]='x' : sostituisce i primi 3 valori della lista con un unico elemento
             (qui x), la lista ora ha 2 elementi in meno.

del A[3]     : elimina l'elemento numero 3 (il quarto)
del A[3:5]   : elimina gli elementi dal 3 (compreso) al 5 (escluso)
del A[3:5:2] : come sopra, ma con passo 2: uno si, uno no
A[0:3]=[]    : elimina i primi 3 elementi (da 0 a 2 )

A[0:3]=['a','b','c'] : sostituisce i primi 3 elementi della lista
                       con elementi di una seconda lista

len(A)         : lunghezza lista A
max(A), min(A) : massimo e minimo valore, se gli elementi non
                 sono confrontabili numericamente si ha un errore.

A.sort()    : ordina gli elementi della lista
A.reverse() : mette gli elementi in ordine inverso

A.append('v')     : aggiunge un elemento 'v' in fondo alla lista
A.extend([2,3,4]) : estende con altra sequenza, aggiungendola in fondo

A.insert(3,'v') : inserisce l'elemento v nella posizione 3, gli altri
                  elementi vengono spostati.

A.pop()  : restituisce l'ultimo elemento e lo elimina dalla lista.
A.pop(i) : restituisce l'elemento al posto i e lo elimina dalla lista.

A.count('v') : da il numero di volte che l'elemento 'v' e' nella lista

A.index('v')    : restituisce l'indice del primo elementi 'v' che trova
```

Unione di liste:

```
B=[1,2,3]+[4,5,6]  : unione di liste: B==[1,2,3,4,5,6]
B=['ab']*3         : ripetizione di  liste: B=['ab','ab','ab',]
```

Unrolling liste:

```
a=[1,2,3,4,5]
g1,g2,g3,g4,g5=a

b,c,*d=a      ==>  b==1 ; c==2 ; d==[3,4,5]
b,c,*d,e=a    ==>  b==1 ; c==2 ; d==[3,4] ; e==5
```

Zip:

```
a=[1,2,3]
b=['a','b','c']
c=zip(a,b)      # zip fa iterabile in Python 3

list(c)         # lista di tuple di 2 elementi
```

Map:

```
c=map(abs,[-1,-2,-3]) => [1,2,3]
```

Range:

```
a=range(2,6,2)  => range(2,6,2) #  => 2,4
a=range(3)      => range(0, 3)  #  range e' un iterabile

for i in a :   # per a=range(0,3) i => 0,1,2
   ....
   ....

for i in range(len(A)) :
   ....
   ....

i=iter(a)  # creo iteratore

next(i) => 0
next(i) => 1
next(i) => 2
next(i) => StopIteration exception
```

List comprehension:

```
b=[x*x for x in range(3)]              :  genera [0,1,2]
b=[x for x in range(6) if x%2 ]        :  genera [1, 3, 5]
b=[y for x in range(2) for y in [5,6]] : genera [5, 6, 5, 6]


linea="1 2 3"
b= [float(a) for a in linea.split() ]  : anche usando funzioni
```

Tuple:

```
T=(1,3.5,'c')   : definisce una tupla di 3 elementi
TT=()           : e' una tupla vuota
T[0]=33         : da errore, la tupla non e' mutabile
T[1]            : il secondo elemento ( nel nostro 3.5)
len(T)          : numero elementi
T.index(3.5)    : indice dell'elemento 3.5 ( e' al posto 1 )

L=list(T6)  :  muta la tupla in una lista: [2, 3, 2, 3]
TT=tuple(L) : fa il contrario, muta una tupla in una lista.
```

Set, Frozenset:

```
a=set()             : definisce set  vuoto

b={1,2,3}           : definisce un set in Python 3
k= {'h','a','m'}    : definisce un set in Python 3

c=set('fgh')        : crea il set: {'h','g','f'}
c=set('fghhh')      : sempre {'h','g','f'} elementi non sono ripetuti.

b=set(['a','b','c']) :  set possono essere costruiti da liste

d = b | c  :        set unione : set(['a', 'c', 'b', 'f'])
e = d & b  :        set intersezione set(['a', 'c', 'b'])
d - b      :        set differenza set(['f'])

b.add('abc')      :   aggiunge i caratteri come elementi
k.add((10,20,30)) :   aggiunge una tupla
```

Operatori per set:

```
> < >= <= (per sottoinsiemi)

| & -  (unione intersezione insieme differenza)
```

Dizionari:

```
D={}              : crea un dizionario vuoto

D={'chiave':3,'altrachiave':6,1:'abcd'}
D['chiave']       : vale 3
D['altrachiave']  : vale 6
D[1]              : vale 'abcd'

len(D)            : dimensione del dizionario: vale 3 se D ha 3 elementi

D['a']=799.0      : aggiunge elemento, se la chiave esiste gia' lo modifica
D.get('chiave')   : vale 3, ma se la chiave 3 non si trova da il valore None
                  : invece di dare errore, come: D['chiave']
D.get('chiave',defult) : qui il default si specifica direttamente

del D['b']        : elimina l'elemento con chiave 'b'
D.clear()         : svuota il dizionario

D.pop(key)        : data la chiave, estrae l'elemento e lo elimina dal dizionario

D.update ( { 1: 10, 2: 20, 3:30 } ) :  unisce dizionari

La funzione "in" e' riferita alle chiavi,
Un ciclo for sul dizionario restituisce le chiavi

D4.keys()   : restituisce un iterable, che contiene le chiavi: ['a','c','b']
D4.values() : restituisce un iterable, che contiene  i valori: [0,3,1]
D4.items()  : restituisce un iterable, che contiene le tuple: chiave, valore:

D=dict('a':10,'b'='abc')         : dict da keyword args.: crea {'a':10,'b':'abc'}
D=dict( [('a',10),('b','abc')] ) : dict da lista di tuple
D=dict( (('a',10),('b','abc')) ) : dict da tupla di tuple
D=dict(  {'a': 10, 'b': 'abc'} ) : dict da dict

C=D.copy()    : copia le reference dentro D, per copie di oggetti
                esiste D.copy.deepcopy() (modulo copy)
```

Dict comprehension (Python 3 e 2.7):

```
D={ x :x*x for x in [1,2,3] }   : genera: {1: 1, 2: 4, 3: 9}

D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3])}
                             genera: {'b': 2, 'c': 3, 'a': 1}
D = {k: v for (k, v) in zip(['a', 'b', 'c'], [1, 2, 3]) if v%2 == 0 }
                             genera: {'b': 2}  : ho tenuto solo numeri pari
```

Analogo dell'istruzione switch:

```
{'a':0,'b':1,'c':3}['b']   Questo vale 1
f={1:f1,2:f2,3:f3}[s]  questo vale f1 se s==0, f2 se s==1, f3 se s==2.
f()                    se f1,f2,f3 sono funzioni, posso eseguire cose
```

Funzioni

Il nome e' un riferimento alla funzione, () la esegue. In mancanza di return da "None", pass e return by reference

Una funzione vede l'**enclosing scope**

Attributi di funzioni:

```
__doc__   : stringa descrittiva
__name__  : nome della funzione
__code__  : info su argomenti etc.

dir(f)    : mostra dizionario degli attributi della funzione
```

Definizione e chiamata:

```
def nomefunzione(a,b,c):
    '''  docstring:
    descrizione funzione
    '''
    d=a
    e=b+c
    return d+e

nomefunzione(r,s,t)     # pass by reference (copia riferimenti)
g=nomefunzione(r,s,t)

g=nomefunzione(r=2, t=4, s=3)  # argomenti per nome (nome nella funzione)

def func(a='uno'):      # argument keywords

def func(*a):           # mette in tuple

def func(**d):          # mette in dict (passare by keyword o tuple)

def func(a,*b,**d):     # prima posizionali, poi per tupla,
                        # poi keyword arguments per dizionario
                        # Es.: func(1, 2,3,4, s=10,q=20 )

def func(a,*b,c):       # Solo Python 3 :
                        # posizionali poi per tupla,
                        # poi keywords arguments

Unrolling nella chiamata:

func(*a)  : spacchetta l'iterabile a in modo implicito

func(**d) : spacchetta il dizionario in: key1=val1, key2=val2 ..
            Se il dizionario e': {'key1':1,'key2':2,'key3':3}
            la chiamata equivale a:  func(key1=1,key2=2,key3=3)
            Ove key1,key2,key3 sono le variabili nella funzione.
```

Return

```
return riferimento_qualunque

return a,b,c   #  (una tupla)

yeld a         # per iteratori, al next call parte da qui
```

Lambda:

```
f= lambda x,y : x+y   #  f(2,3)    produce 5

L=[ (lambda  x: x+x ) , ( lambda x: x*x) ]
for f in L :
    print f(3)      # Produce : 6 , 9
```

Esempio lambda: dizionario di funzioni:

```
op={'somma':lambda *a: sum(a) , 'massimo':lambda *a: max(a)}
op['somma'](10,20,30)    #  produce: 60
op['massimo'](10,20,30)  #  produce: 30
```

Files

Stampe:

```
 print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout])

    sep    : stringa di separazione fra le variabili stampate
    end    : carattere di fine linea a fine stampa
    file   : riferimento al file su cui si stampa

print(a,b)                  : scrive 2 variabili separate da uno spazio

print("%s xxxx %s" % (a,b)) : scrive a e b separati da xxxx

a=input("=> ")              : legge una linea e la mette nella stringa "a"
                              prima di leggere stampa: "=> "
```

Open:

```
open(nome_file, mode='r', buffering=-1, encoding=None,
      errors=None, newline=None, closefd=True)

mode e' il modo do accesso:
          'r' : per lettura di testo
          'w' : per scrivere testo
          'a' : aggiunge testo a fine file
          '+' : sia lettura che scrittura
          'b' : file binario ove NON si interpretano i caratteri come UTF-8

          I caratteri possono essere combinati, ad esempio
          'wb' ed 'rb' per scrivere o leggere dati binari.
```

Esempi e funzioni per i files:

```
f= open('filetest','w') : apre un file di nome filetest per scriverci.
                          e crea un oggetto f, di tipo "file"
f1=open('filetest2','r') : apre un file di nome filetest per leggerlo
f2=open('filetest3','r+'): per leggere e scrivere
f3=open('filetest4','a') : per aggiungere in fondo al file
data = open('data.bin', 'rb').read() : lettura file binario,messo in 'data'

f,f1 etc. sono riferimenti ai files, che sono oggetti.

stringa2=f1.read()       : mette in stringa2 tutto il file f1
stringa2=f1.read(10)     : mette in stringa2 10 byte del file f1

stringa1=f1.readline()   : legge una linea dal file f1 e la mette in stringa1
stringhe=f1.readlines()  : legge tutto il file e ne fa una lista
                           ogni elemento della lista e' una linea del file
f.writelines(stringhe)   : scrive, di seguito, le stringhe della
                           lista. Per essere scritte su diverse linee le
                           stringhe devono finire con: "\n"
f.write('stringa')       : scrive sul file una stringa
f.write('stringa\n')     : scrive sul file una linea
                           (\n e' il carattere di fine linea)
f.flush()                : svuota il buffer, scrivendolo tutto sul file

f.seek(5)                : si posiziona al sesto byte del file
                           ( i bytes si contano a partire da 0)
f.tell()                 : dice a che byte e' posizionato il file

f.truncate(m)            : tronca il file dopo m bytes
f.close()                : chiude il file. L'oggetto 'f' viene distrutto
f.name                   : contiene il nome del file
f.mode                   : stringa che specifica il modo di accesso: 'r','w' etc.
```

Files ed iteratori:

```
for line in open('data.txt'): print(line)

lines = [line.rstrip() for line in open('script2.py')]

lines= [ line.split() for line in open('script2.py')]

In quest'ultimo esempio si fa una lista di liste ove si
separano le parole di ogni linea, che diventano una lista.
```

Eccezioni

Sono classi, in Python 3 devono ereditare da BaseException

Sintassi:

```
try:
  ...
  ...        # blocco entro cui puo' verificarsi l'eccezione
  ...        # Es.:  if a<0: raise ecc1
  ...
except ecc1 as var :
  ...
  ...        # blocco eseguito per l'eccezione: ecc1
  ...
except (ecc2,ecc3) as var :
  ...
  ...    # blocco eseguito per le eccezioni: ecc2 od ecc3
  ...
except
  ...
  ...    # blocco per tutte le altre eccezioni
  ...
else:
  ...
  ...    # blocco eseguito se NON ci sono eccezioni
  ...
finally:
  ...
  ...    # blocco eseguito in ogni caso
  ...
```

Assert:

```
assert 5 > 2                : questa NON lancia un'eccezione
assert 2 > 5 , "messaggio"  : la condizione e' falsa,
                              viene lanciata un'eccezione
                              che contiene il messaggio (opzionale)
```

Classi

Alla definizione la classe viene eseguita. Le classi (le definizioni) sono istanze della classe type.

Gli attributi sono pubblici Se iniziano con "__" vengono nascosti, mettendoci davanti il nome della classe, esempio: "__a" => "nomeclasse.__a"

Le funzioni sono sempre chiamate sull'istanza

Sintassi:

```
class C1:
  """ blocco che definisce la classe"""
  a=[1,2,3]
  b="abc"               # attributi di classe
  c=33

  def func(self,k)
     print(self.c+k)    # self.c : di istanza
     print(C1.a)        # C1.a   :di classe
     ll=88              # locale alla funzione, NON di classe
  ....

 class C2(object):      # Python 2: si deve ereditareo Object
   kk=3                 # o sono classi di vecchio tipo
   ...
   ...

 class C3(C1,C2):
    """ classe C3 che eredita dalle
           classi C1 e C2 ""
    .........
    .........

 oggetto1=C3()      # istances
 oggetto2=C3()

 oggetto1.func(3)   # chiamata a funzione di istanza
```

Attributi:

```
class NomeClasse(object):
    kk=3
    def somma(self,a)
    return (a+self.kk,a+NomeClasse.kk)

a=NomeClasse()    # a.kk vale 3
b=NomeClasse()    # b.kk vale 3

La funzione viene chiamata sull'istanza, con::

d=NomeClasse()
d.somma(10)   # e ritorna la tupla: (13,13)

d.kk=10
d.somma(3)    # ritorna (13,6) , kk e' cambiata solo per l'istanza
```

```
hasattr(oggetto,attributo)# true se l'oggetto ha l'attributo
locals()                  # attributi namespace corrente
```

Init:

```
class ProvaDue(object):
    "docstring di prova "
    kk=3
    def __init__(self,a,b):
       self.ka=a
       self.kb=b
    def printargs(self):
        print "args:",self.ka,",",self.kb

instan=ProvaDue(10,20)        # istanza , con argomenti
instan.printargs()            # stampa le variabili di istanza
```

Ereditarieta':

```
In Python3 tutte le classi ereditano implicitamente dalla classe Object
Membri di classi parent vanno usati qualificati
col nome della classe, entro la definizione della classe

class C3(C1):
    k=C1.c+1   # si deve dire che c e' in C1 ,
               # Python esegue class, e solo dopo crea i reference
               # per i parent,mentre esegue class non conosce gli ancestors,
               # e cerca b nell'env global secondo il criterio LEGB
               # (local, enclosing, global, builtin)
    def func(self,a):
       kk=self.k          # anche qui, deve essere self.k, oppure C3.k

C3.k=C3.b+1    # DOPO la costruzione della classe (fuori della classe)
               # b e' riconosciuto come membro della classe C3,
               # in quanto ereditato
```

Funzioni per l'ereditarieta':

type(oggetto)

isistance(class,oggetto)

issubclass(class,oggetto)

a **is** b ** : test sul tipo (o classe)

In una classe, se si devono chiamare le funzioni di una classe ereditata o di un'altra classe, si deve fornire l'argomento "*self*", e chiamare la funzione sulla classe:

```
super.__init__(self,..) # __init__ del parent va chiamata in modo espliciti

super(nomeclasse,self).metodo_del_super(args)

super(args) # in una funzione il super di stesso nome
```

Override operatori:

```
class Vector(object):
  def __init__(self,a,b):
    self.a=a
    self.b=b

  def __add__(self,other):
    return (self.a+other.a,self.b+other.b)

  def __mul__(self,other):
    return self.a*other.a+self.b*other.b


 x=Vector(1,2)
 y=Vector(10,20)

 print x+y           # fornisce la tupla (11,22)
 print x*y           # fornisce il numero 50
```

Attributi a run-time:

```
__getattr__(self,nome)      : viene chiamata quando non si trova
                              un attributo, di dato nome.
                              Questa funzione ritorna l'attributo,
                              definendolo.

__getattribute__(self,nome) : viene chiamata quando ci si riferisce
                              ad un attributo che esiste,
                              ma NON se e' definita __getattr__ .
                              Permette di modificare un attributo a run-time

__getitem__ (self,index)    : viene chiamata quando si incontra,
                              per la classe, l'indice fra quadre X[i]  ,
                              ove X e' l'istanza di una classe.
                              Questo fa apparire la classe come una lista.

hasattr(func,attributo)

locals()                 : attributi namespace corrente
```

Altri membri speciali:

```
__new__    : viene chiamato prima di __init__, per usi particolari
__del__    : chiamato prima della distruzione della classe
__str__    : viene chiamato per convertire l'oggetto in una
             stringa per le stampe dell'oggetto.
__repr__   : viene chiamato per una rappresentazione testuale
             dell'oggetto, ad esempio nell'uso interattivo
__call__   : usato caso mai la classe sia chiamata come fosse
             una funzione
__name__  __class__  : nome della classe ed puntatore alla classe stessa
__bases__ : tupla di classi base (classi da cui si eredita)
__dict__  : dizionario di attributi della classe
```

Moduli

Un modulo e' un file Python.

La sessione interattiva o il main e' il modulo di nome __main__

Nomi dei moduli: come nomi variabili: lettere, cifre, '_'

Import esegue le istruzioni nel modulo

```
modulo.__dict__  : dizionario nomi modulo, es.: sys.__dict__

import A               #  A.nomeoggetto
import A as newname    #  newname.attributo (A.attributo NON funziona)

from nomefile import nome,altronome  # nel namespace corrente

import imp
imp.reload(nomemodulo)  # in Python 2 reload era funzione builtin
                        # per rifare un from occorre un reload prima
```

path, senza definire PYTHONPATH , in Debian 11, bullseye

```
 import sys
 sys.path

 ['', '/usr/lib/python39.zip',
     '/usr/lib/python3.9',
     '/usr/lib/python3.9/lib-dynload',
     '/usr/local/lib/python3.9/dist-packages',
     '/usr/lib/python3/dist-packages',
     '/usr/lib/python3.9/dist-packages']

import sys
sys.path.append('/dir/subdir/')  # aggiunge path a run-time
```

Packages

Packages: directory con moduli e file __init__.py

Hanno attributo *__path__*

Il nome delle funzioni ripete la gerarchia delle dirs:

Import:

import nomepackage # esegue __init__.py import nomepackage1.nomepackade2.modulo # per sub-packages

Uso:

nome.nome.nome.func(.)

Namespaces

- builtin: moduli linkati statici nell'interprete
- global: di un modulo, fuori funzioni e classi
- local: di una funzione o classe

    locals() : dizionario val locali

    vars(oggetto) : dizionario variabili oggetto (anche attributo __dict__)

Ricerca nomi: local, enclosing, global,builtin

Una func vede l'enclosing;

In una func **global** rende una variabile globale

In una funzione **nonlocal** rende la variabile dell'enclosing

Scope delle variabili

Una variabile definita in una funzione non e' vista da fuori della funzione. Puo' avere stesso nome di una variabile esterna senza confusione.

Una variabile definita nel blocco in cui la funzione e' chiamata e' vista entro la funzione, ma non puo' essere modificata entro la funzione, a meno che non sia definita "global" entro la funzione.

Se, entro una funzione, una variabile e' definita come **global** sara' **vista anche nel blocco in cui la funzione e' chiamata**. Ma in ogni caso una variabile e' locale al file in cui si trova.

La dichiarazione di global e' del tipo:

global a,b,c

Questa regola di scope e' chiamata LEGB: Local, Enclosing, Global, Build-in ed e' il modo di cercare i nomi di Python

In Python3 una variabile dichiarata "**nonlocal**" in una funzione:

nonlocal a,b

e' definita nell'ambito del blocco superiore, ma deve gia' esistere nel blocco superiore. Nelle funzioni questo permette di mantenere valori nelle diverse chiamate.