Appunti per un corso di Python

Marcello Galli, Agosto-2014

In rete si trovano moltissime introduzioni al linguaggio Python, e questa e' ancora una ennesima introduzione al linguaggio, prodotto secondario di un corso tenuto all' ENEA di Bologna nel 2014. Vengono qui descritte un po' tutte le proprieta' del linguaggio, corredate da alcuni esempi, presupponendo, nel lettore, solo una conoscenza delle basi della programmazione e di un'infarinatura di un qualche linguaggio di programmazione. Non si entra in modo approfondito in argomenti specializzati, come: *introspezione* , *metaprogrammazione*, certi dettagli sull'ereditarieta', sulle eccezioni, l'integrazione con altri linguaggi, o la descrizione di tutte le possibilita' dei moduli della libreria standard. Python ha una base semplice, ma un contorno di software ed applicazioni veramente vasto.

Contenuto

Introduzione

In Python sono implementate, direttamente nel linguaggio, strutture complesse, di uso comune in programmazione, come: liste, dizionari, tuple, stringhe. Queste sono trattate come tipi di dati e ci sono gia' funzioni per agire su di esse, come funzioni di ordinamento, di ricerca etc. Per questo motivo Python e' particolarmente adatto per applicazioni in cui si devono trattare insiemi di dati eterogenei, con un misto di valori numerici e stringhe; gran parte del lavoro che andrebbe fatto scrivendo apposite software o includendo diverse librerie e' gia' fatto, ed integrato naturalmente nel linguaggio. E' meno adatto ad applicazioni numeriche, specie che facciano uso di algoritmi iterativi, essendo un linguaggio interpretato. Python e' multi-piattaforma e corre, in pratica, su qualunque architettura. Python e' *open source* , gratuito e liberamente disponibile.

Il sito ufficiale di Python e': https://www.python.org/ dove si trova software, documentazione, e tutto il resto. Nel Python Package Index (PyPI): pypi.python.org si trova una quantita' spropositata di software scritto in Python, per fare quasi qualunque cosa.

In Italia c'e' una robusta comunita' di appassionati al linguaggio; il sito della comunita' italiana e' http://www.python.it/ , ove si trovano informazioni e tutorials in italiano. Altro sito italiano e': http://www.python-it.org .

Sono stati pubblicati moltissimi libri su Python, di molti testi inglesi ci sono traduzioni in italiano, ma non sempre sono aggiornate. Siccome Python e' abbastanza cambiato, nel 2008, con la versione 3, conviene sempre cercare testi abbastanza recenti, che coprano la versione 3 del linguaggio.

Fra i tanti libri segnalo:

- **Python in a Nutshell, di Alex Martelli, O'Reilly, 2003.**

  E' un testo di riferimento completo, purtroppo e' datato e non tratta la nuova versione 3 del linguaggio.

- **Learning Python, 5th Edition, di Mark Lutz, O'Reilly, 2013.**

  Un libro completo, ma forse un po' troppo prolisso, copre tutti gli aspetti del Python; quest'ultima versione tratta anche la versione 3 di Python.

- **Python Pocket Reference, 5th Edition, di Mark Lutz, O'Reilly, 2014.**

  Un sommario breve di tutto Python, versione 3 compresa; ne esiste una (pessima) traduzione italiana.

- **Programming Python, 4th Edition, di Mark Lutz, O'Reilly, 2010.**

  Questo libro copre argomenti specifici, come creare applicazioni di rete, produzione interfacce grafiche, applicazioni web, interfacce a database, integrazione con linguaggio C etc.

- **Python Cookbook, di Alex Martelli e David Ascher, O'Reilly, 2002.**

  Questo libro affronta molti problemi specifici, indicando trucchi e metodi particolari, purtroppo anche questo libro e' un po' vecchio e non copre la versione 3 di Python. C'e' anche un *Python Cookbook* di David Beazley e Brian K. Jones, O'Reilly 2013, dello stesso taglio, ma piu' recente.

- **Programmare con Python. Guida completa, di Marco Buttu, LSWR 2014.**

  E' un libro recente, in italiano, ben fatto.

Il Linguaggio Python

Python e' un linguaggio di programmazione sviluppato da Guido Van Rossum negli anni 90. Si propone come un linguaggio open source, moderno, semplice da imparare e comprensibile, ma contemporaneamente potente, inteso a ridurre i tempi di sviluppo del software.

Il linguaggio Python e' attualmente gestito dalla Python Software Foundation, fondata nel 2001, che e' una associazione indipendente, che annovera fra i suoi sponsor Sun, Canonical, O'Reilly, Microsoft, Zope. Guido Van Russum presiede la fondazione e tuttora coordina lo sviluppo del linguaggio.

Van Rossum inizio' a lavorare su Python nel 1989, le prime release sono del 1991, mentre la versione 1 e' del 1994; il nome Python deriva da uno show comico trasmesso dalla BBC: *"Monty Python's Flying Circus"*, di cui Van Rossum era appassionato. Van Rossum ha sempre cercato di mantenere una retrocompatibilita' nel corso dello sviluppo di Python. Questa retrocompatibilita' e' stata abbandonata con la versione 3 del linguaggio, nel 2008. La versione 3 introduce, fra l'altro, un completo supporto alle stringhe in codifica Unicode, introduce i tipi: *byte* e *bytearray* per dati binari, e cambia alcune funzioni di uso molto comune, come il comando print, che diventa una funzione. Data l'enorme quantita' di software scritta con Python 2, e le differenze con Python 3, la transizione a Python 3 e' stata molto lenta, ed ha comportato un vero e proprio fork; e nel 2014 c'e' ancora software importante che non funziona con Python 3. La versione che viene attualmente sviluppata e' la versione 3, mentre la versione 2 e' ferma alla versione 2.7, del 2010, ove sono state riportate alcune funzionalita' della versione 3.

Versioni di Python e date di rilascio	
Python 1.0	Gennaio 1994
Python 1.5	31 Dicembre, 1997
Python 1.6	5 Settembre, 2000
Python 2.0	16 Ottobre, 2000
Python 2.1	17 Aprile, 2001
Python 2.2	21 Dicembre, 2001
Python 2.3	29 Luglio, 2003
Python 2.4	30 Novembre, 2004
Python 2.5	19 Settembre,2006
Python 2.7	3 Luglio, 2010
Python 2.6	1 Ottobre , 2008
Python 2.7	3 Luglio, 2010
Python 3.0	3 Dicembre, 2008
Python 3.1	27 Giugno, 2009
Python 3.2	20 Febbraio, 2011
Python 3.3	29 Settembre, 2012
Python 3.4	16 Marzo, 2014

Il linguaggio ha avuto grande successo, ed ha attratto una vasta schiera di sviluppatori, in particolare e' utilizzato e sponsorizzato dalla "Zope corporation" (vedi http://www.zope.com e http://www.zope.it) che utilizza Python

per il suo prodotto "Zope", del 1998; uno dei primi applicativo web *"open source"* per editoria e commercio elettronico. Per la diffusione di Python e' stato anche importante "Plone", del 2001; un applicativo di grande successo per costruire siti web (un "Content Management System" o CMS). Per Plone vedi: http://www.plone.org , http://www.plone.net o http://www.plone.it .

Python e' utilizzato da grandi imprese attive nel web fra cui YouTube, Yahoo, Dropbox e Google, ove ha lavorato anche van Rossum, fra il 2005 ed il 2012, prima di trasferirsi alla Dropbox; ma oltre alle grandi ci sono anche centinaia di medie e piccole ditte che utilizzano Python per i loro progetti, specialmente per siti ed applicativi web.

Python sta prendendo piede anche in campo scientifico, specialmente per analisi dati e grafica. Prodotti, scritti in Python, specifici per calcoli vettoriali e trattamento di dati scientifici, sono: numpy e scipy. Matplotlib e' un software per la grafica, Mayavi e' un software per la rappresentazione tridimensionale di dati.

Caratteristiche del linguaggio

Interpretato, ma con produzione di "bytecode".

Il linguaggio e' interpretato, ovvero tradotto in linguaggio macchina una istruzione per volta, tipo la shell di Unix, e' quindi un linguaggio che puo' essere usato in modo interattivo, ma inefficiente dal punto di vista del calcolo. Un programma Python puo' anche essere messo in un file, ed eseguito, come un'unica procedura; durante la traduzione viene prodotto un "bytecode", ovvero una versione del programma semi compilata, posta in un file con estensione "pyc". Se il programma viene eseguito una seconda volta, senza modifiche, si utilizza direttamente il byptecode, con un vantaggio in termini di prestazioni.

Sintassi semplice.

La sintassi e' semplice, ci sono pochi construtti e non c'e' bisogno di dichiarare il tipo delle variabili. Non ci sono puntatori, anche se tutte le variabili sono implementate con puntatori. Il Python si occupa di individuare il tipo di ogni variabile quando viene definita, e fa conversioni automatiche quando occorre.

Orientato agli oggetti.

Il Python ha un modo semplice ma efficace di definire e trattare classi, e praticamente tutti i tipi di dati sono oggetti. C'e' anche la possibilita' di definire "moduli": insieme di funzioni Python predefinite, conservate in un file, che possono essere riutilizzate, inserendole in programmi diversi, senza che ci sia confusione fra le variabili del programma e quelle dei moduli stessi.

Strutture complesse gia' implementate.

Il Python comprende una implementazione di strutture complesse, di uso comune in programmazione, come: liste, dizionari, tuple, stringhe. In Python queste sono trattate come tipi di dati e ci sono gia' funzioni per agire su di esse, come funzioni di ordinamento, di ricerca etc.

Facile integrazione con altri linguaggi.

C'e' la possibilita' di usare Python per legare insieme componenti sviluppate con altri linguaggi, tipo Java, C o C++. C'e' anche Jython, un compilatore che traduce un programma Python in un bytecode java.

Funzionalita' gia' pronte.

Python contiene una grande libreria di funzioni, inoltre si trovano in libera distribuzione moduli Python per fare moltissime cose: interfaccia con database, grafica , networking etc.

Caratteristiche peculiari.

Python ha diverse caratteristiche peculiari, proprie dei linguaggi moderni; come un sistema di "garbage collection", per liberare memoria inutilizzata, ed il "dynamic typing", cioe' il fatto che il tipo degli oggetti e' definito durante l'esecuzione del programma e non durante la compilazione. Questo permette di scrivere parti di programma in modo indipendente dal tipo di dato che in esso verra' utilizzato (*polimorfismo*). Inoltre puo' accedere alle strutture interne dei suoi oggetti; questa proprieta e' chiamata *introspezione* o *reflection*. Un programma Python puo' anche modificare se stesso a run-time (*metaprogramming*).

Uso

Python e' disponibile praticamente per tutte le architetture, ed incluso in tutte le distribuzioni di Linux. Si puo' usare in modo interattivo o mettere istruzioni pyhton in file ed eseguirlo. Ci sono anche tool grafici per creare programmi Python, come **idle** , oppure spyder

Esempio di Uso interattivo in Linux (dalla shell):

```
$ Python
>>> 3+2
5
>>> a=16
>>> a
16
>>> Cntrl/D per finire
```

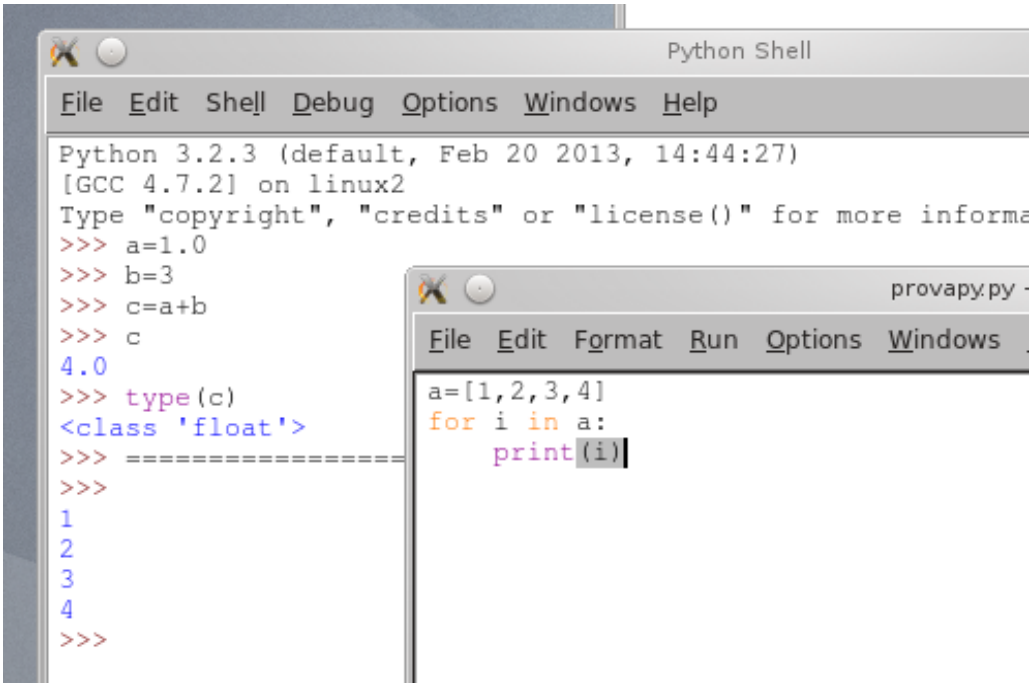
Un file con un programma Python: *file.py* si esegue con:

```
$ Python   file.py
$ Python3  file.py   # per usare una versione specifica di Python
```

In Unix un file eseguibile con, nella prima linea: *#!/usr/bin/python* Puo' essere eseguito direttamente come un comando.

Interfaccia grafica: idle

idle e' un'interfaccia grafica che rende disponibile una shell per eseguire programmi o singoli comandi, ed un editor per scrivere file con programmi Python. Integra anche un debugger.

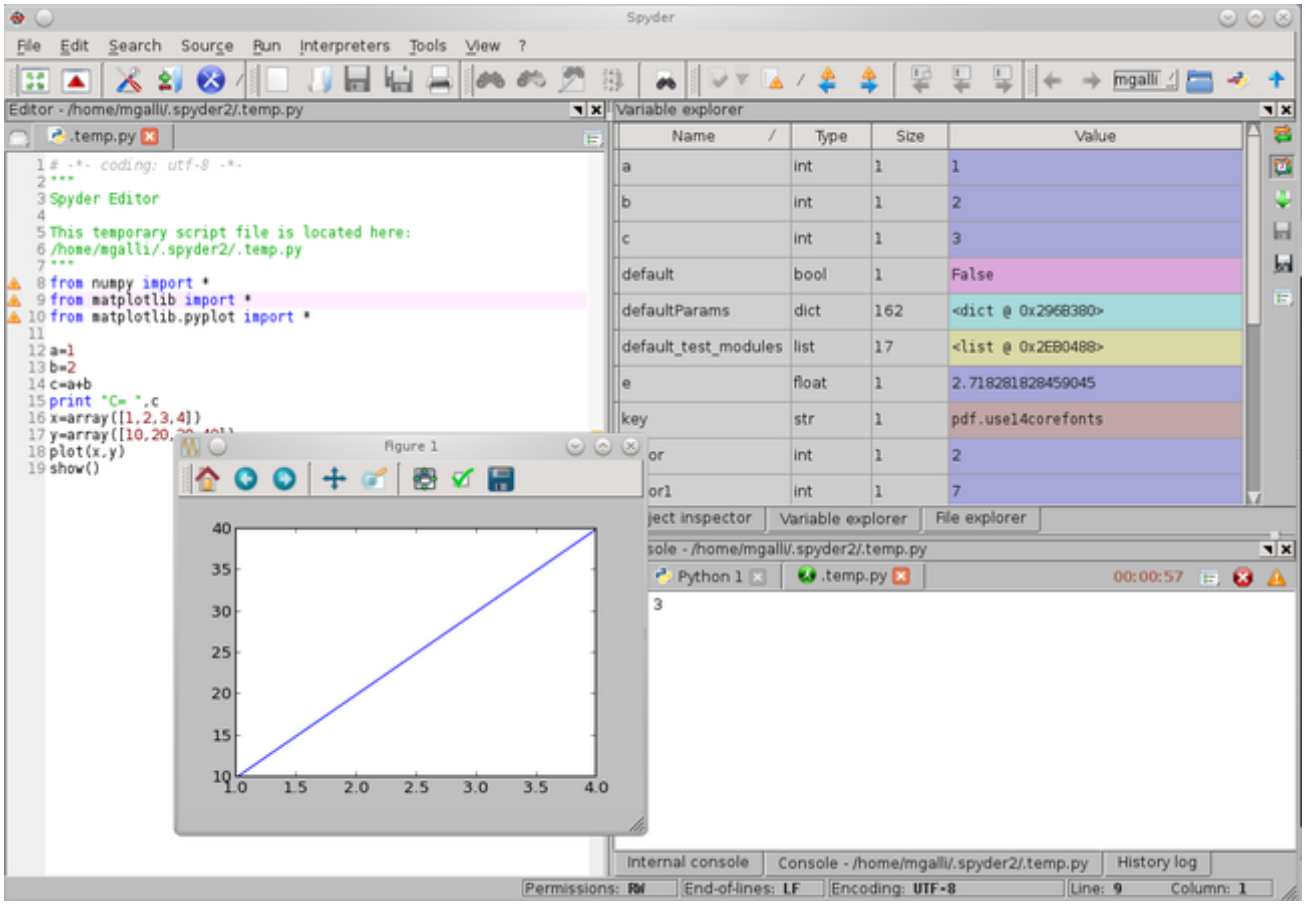


Esistono anche altre interfacce dedicate a Python;

- "*Ipython*" e' una shell Python con diverse estensioni per l'interazione con il sistema operativo;
- "*Ipython notebook*" attiva un web server locale e permette di utilizzare l'interfacia del browser come shell grafica per l'ambiente Python;
- "*spyder*" e' un'interfaccia grafica per analisi dati scientifici, con uso di moduli come munpy, matplotlib o scipy.

  La figura mostra l'uso di Spyder per un semplice grafico.

.



Sintassi

Variabili

Python e' **case sensitive**

I nomi delle variabili iniziano con un carattere alfabetico e contengono caratteri, numeri o underscore: "_" . Nomi che iniziano con "_" hanno significati speciali, ed alcuni sono definiti dall'interprete.

Una linea che termina con il carattere: "\" continua nella successiva; espressioni fra parentesi possono occupare piu' righe e linee vuote vengono ignorate. Piu' istruzioni possono stare sulla stessa riga, se separate da punto e virgola "**;**"

I commenti sono identificati dal carattere: "**#**", ed il commento va dal carattere a fine linea. Nell'ambiente Unix, se si crea un file eseguibile e si mette: **#!/usr/bin/python** all'inizio del file, si segnala al sistema che il file va eseguito come programma python.

Caratteri bianchi all'inizio di una riga sono utilizzati per definire blocchi logici del programma (vedremo poi come).

Riferimenti ed oggetti

Python e' orientato alla programmazione ad oggetti, nel senso che tutti i tipi di variabili e tutte le entita' su cui Python opera sono,o si comportano, come oggetti.

I nomi delle variabili sono in realta' riferimenti (reference) ad oggetti, cioe' sono implementati internamente come indirizzi degli oggetti. Ridefinire una variabile significa assegnare il riferimento ad un nuovo oggetto, e lasciare il vecchio oggetto con un riferimento in meno.

Gli oggetti ereditano tutti l'oggetto base "*object*", che ha , fra i suoi attributi, un tipo ed un "*reference count*": il numero di riferimenti che puntano ad esso. Quando il "*reference count*" diventa zero, un sistema di garbage collection automaticamente elimina l'oggetto. In caso di ereditarieta' complicate puo' essere che un oggetto , nella gerarchia, abbia un riferimento a se stesso. Per questi casi ci sono algoritmi particolari per vedere quando l'oggetto puo' essere eliminato.

Oggetti base in Python (fra cui numeri e caratteri), sono **immutabili**, nel senso che una volta creati non si possono cambiare; le variabili sono riferimenti a questi oggetti, e maneggiare le variabili non tocca gli oggetti cui si riferiscono. Salvo che quando un oggetto non ha piu' riferimenti viene eliminato.

Oggetti **mutabili** sono oggetti composti, costruiti internamente con insiemi di riferimenti ad oggetti immutabili. Questi si possono modificare a run-time.

Keywords

Python ha pochi comandi (keywords), che in Python 3 sono:

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

Tipi

In Python ci sono alcuni tipi base, come interi, float, caratteri etc., ma ogni oggetto definisce implicitamente un tipo di variabili. Alle variabili non e' assegnato un tipo a priori ed il tipo delle variabili non va dichiarato, come in FORTRAN, C o Pascal, ma quando ad una variabile viene assegnato un oggetto viene definito anche il tipo della variabile. Questo viene chiamato "*run-time binding*", "*late binding*" o "*dynamic binding*", e permette al programma di essere scritto in modo indipendente dai tipi di variabili (polimorfismo). Ovviamente non si riescono a fare operazioni che non sono ammesse per il tipo che la variabile rappresenta, ed il Python in questi casi da errori, che possono essere rilevati sono all'esecuzione del programma, visto non c'e' una compilazione e l'interprete ignora i tipi delle variabili e non fa controlli.

Python, se puo' , effettua automaticamente la conversione fra tipi nelle operazioni numeriche.

Python ha anche tipi che sono **sequenze**; le sequenze sono insiemi di elementi ordinati, cui ci si puo' riferire tramite indici interi. Gli indici partono da zero, analogamente a quanto accade in C, non da uno, come in FORTRAN. Indici negativi partono dal fondo della sequenza. Gli indici sono rappresentati come numeri, o variabili, fra parentesi quadre. Ad esempio: se la variabile 'a' rappresenta una sequenza, il suo primo elemento sara': a[0], il suo secondo elemento a[1] e cosi' via.

I tipi di base sono:

- **int** : interi con segno , (immutabili), possono avere un numero illimitato di cifre.
  - se preceduti da 0O oppure 0o sono in notazione ottale
  - se preceduti da 0x oppure 0X sono in notazione esadecimale
  - se preceduti da 0b oppure 0B sono binari

  Esempi:

  7 ; 2147483647 ; 0o177 ; 0b100110111 ; 0xdeadbeef 792281625142643375935439503 36

  oct(x) visualizza il numero x in ottale,

  hex(x) in esadecimale

  bin(x) in binario
- **float** : numeri reali, in doppia precisione, (immutabili),

  l'esponente e' preceduto dalla lettera **E** on **e** e segue la parte frazionaria.

  Esempi:

  3.14 ; 10. ; .001 ; 1e100 ; 3.14e-10 ; 0e0
- **complessi** : sono somma di reale e di immaginario (immutabili),

  la parte immaginaria e' seguito da **j** oppure **J**

  Esempi:

  1+3.14j ; 10.j ; 1.e2+10j ; 0+.001j ; 1e100j ; 3+3.14e-10j
- **decimali e frazionari** :

  sono tipi numerici supportati dalla libreria standard, ove sono create le classi relative. I decimali hanno un numero fisso di cifre decimali, i frazionari sono frazioni, usati per rappresentare senza approssimazione i numeri razionali.
- **bool** : booleani,

  possono essere veri o falsi; assumono uno dei 2 valori: **True**, **False**. Il numero 0 e' considerato falso, altri numeri sono considerati veri, una stringa definita e' vera, un oggetto vuoto e' falso,
- **None** : e' una variabile speciale che indica l'assenza di un valore.

E' usata specialmente per stringhe ed e' il tipo: "*NoneType*"; la variabile None e' considerata falsa.

- **str** : stringhe

  sono sequenze di di caratteri (immutabili). Sono in codifica unicode in Python3, in codifica ascii in Python 2. Le stringhe, intese come sequenze di caratteri, permettono di accedere a singoli caratteri tramite un indice intero. Le stringhe sono rappresentate con caratteri fra apici doppi o semplici.

  Esempi:

  "abcd" ; '123AAQH"

- **bytes** : sequenze di interi, nel range 0-255, (immutabili).

  Esistono solo in Python3, ma non in Python2. Siccome le stringhe in Python3 sono in codifica Unicode non sono piu' uno strumento adatto a trattare piccoli interi, per questo e' stato introdotto questo tipo di dati.

- **byte array** : e' un tipo analogo al tipo byte, ma mutabile

- **list** : sono sequenze di oggetti eterogenei (mutabili).

  Sono rappresentate come elementi separati da virgole, racchiusi fra parentesi quadre. Le liste possono contenere ogni tipo di oggetto, liste comprese.

  Esempi:

  [0,1,2,3,4] ; [1,2,'abc/,12.5E3] ; [0,1,2,['a','b','c'],32.4]

- **tuple** : sono sequenze di oggetti eterogenei immutabili.

  Sono analoghe alle liste, ma sono rappresentate con valori racchiusi fra parentesi tonde. Le tuple possono contenere liste, che sono mutabili, la lista nella tupla puo' mutare, ma non essere tolta dalla tupla, che e' immutabile.

  Esempi:

  (0,1,2,3,4) ; (1,2,'abc/,12.5E3) ; (0,1,2,('a','b','c'),32.4)

- **dict** : dizionari, od array associativi.

  Sono insiemi i cui elementi non hanno come indice un numero, ma sono individuati da un oggetto, detto chiave (key). Gli elementi di un dizionario non hanno un ordine definito, come le sequenze. Anche i dizionari contengono oggetti eterogenei, ma le chiavi devono essere oggetti immutabili; siccome internamente i dizionari sono rappresentati come "hash tables" le chiavi devono essere oggetti "hashable", cioe' adatti ad essere trasformati in indirizzi dagli algoritmi interni di Python.

  I dizionari sono rappresentati come coppie "chiave:valore", con elementi separati da virgole e fra parentesi graffe. Un elemento del dizionario e' individuato dalla chiave, messa fra parentesi quadre dopo la variabile che si riferisce al dizionario

  Esempio:

  D={'key1':3,'key2':6,1:'abc'}

  qui D['key1'] si riferisce al numero 3, D[1] alla stringa 'abc'

- **set** : insiemi.

  Sono gli insiemi della matematica insiemistica. Sono composti di oggetti non sono ordinati che non sono recuperabili con un indice; su di loro si possono fare operazioni come: unione, intersezione, differenza etc. etc.

  Sono oggetti mutabili, ma che contengono oggetti immutabili ed in un unica copia. Sono rappresentati da elementi fra parentesi graffe, separati da virgole.

Esempio:

a={1,2,3,'cc',1,2}

a contiene: {'cc', 1, 2, 3} : non ci sono oggetti multipli

- **frozenset** : sono come i set, ma immutabili.

  Si possono creare da una lista, tupla o dizionario, con la funzione frozenset.

  Esempio:

  a=frozenset([1,2,3])

- **range** : sono sequenze di interi.

  Sono create con la funzione range, che ha come argomenti il primo, l'ultimo valore (escluso) ed il passo della sequenza.

  Esempio:

  range(2) e' la sequenza: 0,1

  range(2,4) e' la sequenza 2,4

  range(1,5,2) e' la sequenza 1,3

Stringhe, liste, tuple, range, sono sequenze. Le sequenze, ma anche i dizionari e gli insiemi sono detti **iterable** , sono cioe' oggetti i cui elementi si possono estrarre uno per volta, scorrendo sulle componenti dell'oggetto. Questo perche' sono oggetti che implementano tutte le funzioni per fare questo. Questo modo di classificare gli oggetti in base alla loro interfaccia e' chiamato "**duck typing**": qui e' il comportamento di un oggetto definisce il suo tipo. E' un concetto che si ritrova nei moderni linguaggi interpretati; una specie di polimorfismo, ma questa e' un'idea ancora piu' generale e l'interfaccia stessa diventa la definizione del tipo di dato.

In python 2 c'era il **tipo long** per interi a molte cifre ed **int** era limitato ad interi a 32 bit. Un numero che finiva con un "**L**" era long anche se era definito con poche cifre. Con python3 gli interi possono sempre avere molte cifre ed il tipo 'long' e' scomparso.

La funzione **type(oggetto)** dice di che tipo e' l'oggetto, la funzione **isistance** e' un test sul tipo:

Esempio:

type([]) produce: <class 'list'>

isinstance([], list) produce: True

Docstring:

All'inizio del file, o di una funzione o classe, e' buona pratica mettere una stringa descrittiva, anche su piu' linee. Questa finisce nella variabile **__doc__** dell'oggetto o della funzione e serve da documentazione, E' mostrata dalla funzione help con l'oggetto come argomento. Questa stringa descrittiva si chiama "*docstring*".

Operatori

In Python ci sono tutti gli operatori cui linguaggi come il C o Java ci hanno abituati, come in C, indici fra parentesi quadre si usano per accedere a singoli elementi di sequenze. Si possono pero' usare gli indici anche per estrarre sezioni delle sequenze (slicing). Le parentesi quadre sono a tutti gli effetti operatori, e, quando si creano classi,se ne puo' fare l'override. Alcuni operatori usati per i numeri sono ridefiniti per le sequenze, in modo da fare, su queste, operazioni particolari. Le stringhe hanno un operatore di formattazione, che si comporta in modo analogo alla funzione printf del C.

Operatori aritmetici

In Python 3 la divisione fra 2 interi fornisce un numero float. In Python 2 invece forniva un intero, troncando il risultato; in Python 3, per avere un risultato troncato, si usa l'operatore di divisione intera: '//' , sia su interi che float.

Operazioni fra interi danno interi, se uno dei 2 operandi e' un float viene prodotto un float; Python in questi casi fa conversioni automatiche.

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

Operatori bit a bit

Questi operatori agiscono sui numeri seguendo una logica binaria e cambiano i singoli bit delle variabili. Questi operatori sono definiti per interi con segno o booleani. Per come sono rappresentati i numeri negativi (complemento a 2) l'operatore '~' cambia segno agli interi e sottrae 1. Il complemento a 2 ha infatti i numeri negativi con il primo bit a sinistra che vale 1, ed ottiene numeri negativi invertendo i bit e sommando 1.

Operatore	Funzione	Esempi
<<	shift a sinistra	a<<1 ; 8<<1 fornisce 16
>>	shift a destra	a>>1 ; 8>>1 fornisce 4
&	or sui bit	a&b ; 2&1 fornisce 0
|	and 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

In Python ci sono diversi operatori di assegnazione; ogni operazione numerica puo' essere combinata con una assegnazione, per modificare una variabile e riassegnarla a se stessa.

Operatore	Funzione	Esempi

=	assegna riferimento	a=3
*=	moltiplica 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

L'assegnazione si comporta in modo diverso per oggetti mutabili e non mutabili; *a=3* crea un oggetto immutabile "3" ed una reference "a", che punta ad esso; se riassegno la reference: *a=3 ; b=a ; a=4* , Python, dopo aver creato una nuova reference "b" a "3", crea l'oggetto "4" e riassegna la reference "a" all'oggetto "4". Il valore di "b" continua a valere 3. Se invece ho un oggetto mutabile, come una lista, nell'istruzione: *a=[3] ; b=a ; a=[4]* , la riassegnazione: *b=a* crea una reference "b", che punta anche lei a *[3]*. La lista e' unica, se la modifico tramite il riferimento *a* (*a=[4]*) vedo modificato anche *b*.

In Python si possono fare assegnazioni multiple, ma il numero di oggetti a sinistra e destra deve essere lo stesso:

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'

In python 3 si possono fare assegnazioni multiple, di parti di un oggetto mettendo il resto in una lista:

```
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

Restituiscono uno degli argomenti.

Operatore	Significato	Esempio
or	or logico	x or y
and	and logico	x and y

not	negazione	not x

L'operatore "*not*" restituisce True o False a seconda di x .

Gli operatori "*not*" ed "*or*" **restituiscono uno degli argomenti** non, semplicemente, False o True.

```
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)
```

Operatori logici per i confronti.

Restituiscono : True o False

Operatore	Funzione	Esempi
>	maggiore	a > b
<	minore	a < b
<=	minore od eguale	a <= b
>=	maggiore od eguale	a >=b
==	eguale	a == b
!=	diverso	b != b

Si puo' controllare se una variabile e' in un intervallo con la sintassi compatta del tipo: 1 < a < 3

In Python 2 c'era anche l'operatore "<>" con lo stesso significato di "!="

Operatori logici per l'appartenenza

L'operatore "*in*" restituisce True o False a seconda che un oggetto faccia parte di una sequenza, dizionario o set, l'operatore "*is*" controlla se due riferimenti puntano allo stesso oggetto.

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

Separatori

Questi separatori li abbiamo gia' visti nella descrizione della sintassi di Python. Alcuni si comportano come veri e propri operatori, ad esempio '[]' per gli indici delle sequenze oppure '()' per le chiamate a funzioni.

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)

Conversioni fra tipi

Queste funzioni effettuano conversioni fra diversi tipi di variabili, o effettuano operazioni particolari, non sono proprio operatori, ma le elenco qui per completezza.

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`

Molte altre funzioni che operano sui numeri sono nel pacchetto "*math*", dedicato alle funzioni matematiche, ad esempio:

```
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

Sono sequenze, e quindi con elementi identificati da indice intero fra parentesi quadre, le liste, le tuple, i byte, i bytearray. Alle sequenze si applicano gli operatori della tabella seguente. Alcuni operatori, come +, assumono significato particolare se applicati a sequenze (si dice che per le classi delle sequenze si fa l'overloading dell'operatore).

Le operazioni che ottengono sottoinsiemi di sequenze sono dette di 'slicing', in italiano e' si trova l'orribile traduzione 'affettamento' ; ma 'sottosequenza' o 'sezione della sequenza' sarebbe una traduzione piu' decente. Indici negativi partono dal fondo della sequenza. Vedremo meglio le sequenze quando si tratteranno le stringhe e le liste.

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	
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	

Operazioni per sequenze mutabili

Le liste ed i bytearray, come sequenze mutabili hanno anche tutte le operazioni per fare modifiche alla sequenza

Operazione	Effetto
s[i] = x	modifica elemento i della sequenza
s[i:j] = t	modifica elementi da i a j (j escluso)
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

Operatore di formattazione per le stringhe: %

Python3 ha anche la funzione 'format', che lavora in modo analogo, ma ha sintassi un po' diversa.

La sintassi dell'operatore e':

```
'stringa di formattazione' % (tupla di valori)
```

La stringa di formattazione contiene dei "placeholder", simboli che vengono sostituiti dai valori della tupla, nell'ordine. Questi "placeholder" iniziano con il carattere: % ed indicano il formato con cui il valore viene rappresentato

Nella tabella alcuni esempi di questi "placeholders"

	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	singolo 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

Esempi:

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

In Python 3 c'e' anche la funzione "format", e si voleva eliminare l'operatore "%" . Qui gli identificatori per la formattazione sono fra graffe, il primo numero e' il numero dell'argomento della funzione, il secondo specifica il formato. Ecco alcuni esempi di uso della funzione format:

```
'{0:.2f}'.format(1.3333))                    produce: '1.33'
format(1.3333, '.2f')                        produce: '1.33'
'{page}: {book}'.format(page=2, book='PR5E') produce: '2: PR5E'
'{0:10} = {1:10}'.format('spam', 123.4567)   produce: 'spam       =   123.4567'
```

Precedenza degli operatori

La precedenza degli operatori e' nella lista seguente: da quelli con precedenza maggiore a quelli con precedenza minore:

```
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)
```

Funzioni per help

Nell'uso interattivo possono essere utili le funzioni dir ed help. La funzione dir: **dir(oggetto)** mostra gli operatori ed i membri dell'oggetto. La funzione help: **help(oggetto.funzione)** mostra cosa fa una funzione, in pratica stampa il suo docstring. La funzione **vars(oggetto)** mostra il dizionario delle variabili definite in un oggetto.

Istruzioni

Istruzione print

L'istruzione *print* non esiste in Python 3. In Python 2 *print* era un'istruzione, con Python3 diventa la funzione *print()*.

La funzione print accetta un numero variabile di argomenti di tipo diverso e li stampa, usando, per la rappresentazione testuale degli oggetti, la funzione str() degli oggetti stessi.

Print ha anche argomenti opzionali con valori di default, che specificano il file su cui si stampa, il separatore fra gli oggetti stampati, ed il carattere da mettere alla fine della stampa. Di default usa uno spazio per separare gli argomenti, 'n' (il carattere di fine linea) a fine stampa e come file si usa l'output standard del sistema. La sintassi completa per stampare 3 oggetti a,b,c sarebbe quindi:

print(a,b,c,sep=' ',end='n',file=sys.stdout)

La sintassi degli argomenti per le chiamate a funzioni verra' ad ogni modo spiegata in modo dettagliato piu' avanti.

Assegnazione

Abbiamo gia' visto l'istruzione di assegnazione, che crea un oggetto e gli assegna una variabile, che e' un riferimento all'oggetto.

Esempi:

```
a  = 3
b  = 6.0
a *= b
c=(a+b) * 3
a=b=3
```

Blocchi logici

Python segue i principi della programmazione strutturata, quindi istruzioni cicliche ed istruzioni condizionali eseguono blocchi logici ben individuati ed un'istruzione "*goto*" di salto incondizionato non esiste.

I blocchi logico in Python sono identificati con indentazione (rientro): tutte le istruzioni del blocco hanno davanti lo stesso numero di spazi, ed il blocco finisce quando l'indentazione del blocco cessa.

I blocchi dentro altri blocchi (nested o annidati) hanno ulteriore indentazione rispetto al blocco che li contiene. Si consiglia di usare spazi bianchi e non tabulazione, visto che gli spazi rappresentati dal tasto di tabulazione possono essere diversi a seconda dei computer e dei programmi usati.

Esecuzione condizionale

Il costrutto: "*if.. then .. else*" che ritroviamo in tutti i linguaggi: se la prima condizione e' vera (quella che segue l'if) allora e' eseguito il primo blocco, altrimenti si prosegue alla condizione successiva; se nessuna e' vera viene eseguita la condizione che segue l'istruzione "*else*". Le clausole "*elif*" ed "*else*" sono opzionali

I blocchi sono delimitati dall'indentazione ed iniziano con due punti ":"

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

Nell'esempio successivo abbiamo un caso di clausole if annidate (nested if). In Python bisogna sempre fare molta attenzione ai rientri. Altri linguaggi usano le parentesi, che sono una scelta piu' comune e meno soggetta ad errori.

```
if a==b:
   c=d
   if e==f:
      k=0
   else:
      k=1
elif a>b:
   c=f
   e=d
else:
   c=0
   e=0
```

Se c'e' una sola istruzione la si puo' mettere nella stessa linea della condizione:

```
if a==b: k=0
```

Istruzione with

L'istruzione with e' inserita con Python 3, ma esiste anche in Python 2.6; e' un modo di definire un oggetto locale ad un blocco e di gestire eccezioni relative all'oggetto. La sintassi e':

```
with espressione  as variabile:
   ...
   ...
```

L'espressione produce un oggetto, che deve implementare il "context manager protocol"; ovvero ha funzioni: "*__enter__(self)*" ed "*__exit__(self,type,value,traceback)*" ,che sono chiamate alla creazione e distruzione dell'oggetto. Servono per gestire un blocco che e' il contesto in cui l'oggetto e' valido ed eventualmente errori (eccezioni) nel blocco che segue l'istruzione "*with*".

La variabile e' un riferimento all'oggetto; il riferimento e' valido nel blocco che segue, dopo viene eliminato, assieme all'oggetto. Un esempio classico e' la chiusura automatica di un file:

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

L'open restituisce un riferimento ad un oggetto file: "*f*" creato dalla funzione: "*open*" . Questo, all'apertura del file, esegue una sua funzione "*__enter__*" . A fine blocco *f* non e' piu' definito ed il garbage collector di python interviene. L'oggetto file, al momento della sua eliminazione, chiude il file, con *__exit__*.

Se in apertura del file, ci sono errori "*__exit__*" li gestisce ed in questo caso "*f*" non e' assegnato ed il blocco non e' eseguito. "*__exit__*" e' chiamata anche per errori nel blocco che segue l'istruzione "*with*".

Oltre che per files questo sistema e' usato anche per gestire threads, locks etc. Dal python 3.1 ci sono anche i context manager multipli, con una istruzione with tipo:

```
with A() as a, B() as b:
   ...statements...
```

Che e' poi equivalente a:

```
with A() as a:
   with B() as b:
       ...statements...
```

Istruzioni cicliche

L'istruzione ciclica di Python e' il ciclo "*while"*, che esegue un blocco finche' la condizione e' vera. La condizione e' testata all'inizio del blocco. Alla fine del blocco, **in ogni caso**, viene eseguito il blocco alla istruzione "**else"**. Anche le istruzioni "*while"* possono essere annidate:

```
while x:
   i+=1
   x-=1
else:
   y=i
```

questa incrementa i e cala x di uno, fino a che x e' zero (falso) quando questo accade il ciclo termina e viene eseguita l'istruzione all'else

Entro un ciclo ci sono istruzioni particolari che alterano la sequenza:

```
break       interrompe il ciclo
continue    passa al giro successivo
pass        non fa nulla , ad esempio:

       while 1:pass  # e' un loop infinito
```

Se, nel blocco del while, viene eseguita una istruzione break si esce dal ciclo **senza eseguire il blocco all'else**

Iterabili ed iteratori

Anche qui abbiamo istruzioni cicliche, ma con una logica diversa.

Liste, dizionari, tuple, sets hanno la caratteristica di essere "*iterabili*" (iterable). Un insieme di oggetti e' iterabile se, su di esso, si puo' definire un oggetto che, in sequenza, assume un diverso valore fra quelli dell'insieme. Questo oggetto che "itera" sulla sequenza e' detto "iteratore".

Un iteratore funziona anche su oggetti complessi ed eterogenei, a differenza di un indice intero, o di un indirizzo, che puo' essere usato in un ciclo solo per descrivere una sequenza di oggetti tutti delle stesse dimensioni.

L'istruzione che crea un iteratore e cicla su tutti gli elementi di un insieme e' l'istruzione "*for*".

```
for i in [1,2,3,4]:
   k+=i
else:
   print('fine')
```

Il riferimento "*i*" assume, in sequenza, i valori della lista, e per ogni valore che "*i*" assume si esegue il blocco che segue l'istruzione "*if*". A fine blocco viene eseguito il blocco della istruzione "*else*", a meno che un'istruzione break non interrompa il ciclo.

Un ciclo for su un dizionario restituisce le chiavi, nell'esempio che segue viene stampato 'a', poi 'b', infine 'c'

```
D={'a':0,'b':1,'c':3}
for i in D:
   print(i)
```

Per iterare su una coppia (chiave, valore) di un dizionario bisogna utilizzare una tupla:

```
for (key, value) in D.items():
   print(key, '=>', value)
```

Nel caso seguente iteriamo su una tupla ed usiamo una tupla come iteratore:

```
T = [(1, 2), (3, 4), (5, 6)]
for (a, b) in T:
     print(a, b)
```

Per avere un ciclo su numeri interi, come in FORTRAN o C, si utilizza la funzione "*range*", oppure la funzione "*enumerate*":

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


for i,a in enumerate(['a','b','c']):
     k[i]=a
```

"*enumerate*" restituisce una tupla (numero crescente, elemento dell'enumerable) nello specifico si assegnano valori sia ad "*i*" che ad "*a*" ; ai diversi giri abbiamo per la tupla con "*i*" ed "*a*":

(1,'a') ; (2,'b') ; (3,'c')

List comprehensions

Sono cicli che assegnano una serie di valori ad una lista, opzionalmente i valori possono essere filtrati da una condizione "*if*"

[x for x in range(5) if x % 2 == 0]

produce una lista con i numeri pari fino a 5: [0, 2, 4]

Si possono avere cicli annidati:

[x + y for x in range(3) for y in [10, 20, 30]]

Qui x va da 0 a 3, e, per ogni x, y va da 10 a 30 ed abbiamo la lista;

[10, 20, 30, 11, 21, 31, 12, 22, 32]

Questo sistema puo' anche essere usato per creare dizionari, usando la funzione zip, che unisce a 2 a 2, in tuple, gli elementi di due sequenze:

Esempi:

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

produce:   {'b': 2, 'c': 3, 'a': 1}

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

produce:   {1: 1, 2: 4, 3: 9, 4: 16}

D = {c: c * 4 for c in 'SPAM'}

produce: {'S': 'SSSS', 'P': 'PPPP', 'A': 'AAAA', 'M': 'MMMM'}
```

Si possono anche inserire operazioni in una comprehension:

D = {c.lower(): c + '!' for c in ['SPAM', 'EGGS', 'HAM']}

produce: {'eggs': 'EGGS!', 'spam': 'SPAM!', 'ham': 'HAM!'}

Funzioni per iterabili

Ci sono diverse funzioni, che fanno parte del linguaggio Python (funzioni builtin), utili per lavorare con iterabili.

La funzione **list** crea un lista, a partire da una serie di valori, o da un iterabile.

La funzione **dict** produce un dizionario, a partire da una sequenza di tuple di 2 elementi: (chiave,valore).

La funzione **range** genera una sequenza di interi, il primo argomento e' il primo valore, il secondo l'ultimo (escluso), il terzo e' il passo.

La funzione **map** applica una funzione ad un iterabile

La funzione **zip** unisce a 2 a 2 due 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))
```

La funzione **filter** applica ad un iterabile una funzione logica elemento per elemento e tiene solo gli elementi per cui la funzione e' True:

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

"*lambda*" e' un modo di definire una funzione in un unica istruzione, come vedremo dopo.

La funzione **iter** crea un iteratore per un iterable:

```
R=[1,2,3]
I=iter(R)
```

```
next(I)
```

"*next*" alle chiamate successive produce: 1 ,2 , 3 se lo si chiama ancora, dopo che e' arrivato a 3, da errore

Negli esempi della tabella che segue 's' indica un iterabile.

Funzione	Effetto
all(s)	True se tutti gli elementi sono veri
any(s)	True se qualche elemento e' vero
list(1,2,3)	produce la lista [1,2,3]
list(range(3))	produce la lista [0,1,2]
list(range(2,10,2))	produce: [2, 4, 6, 8]
list(map(abs,[-1,0,2]))	produce: [1, 0, 2]
tuple(map(abs,[-1,0,2]))	produce: (1, 0, 2)

Funzioni exec ed eval

La funzione "*exec*" permette di eseguire una stringa come istruzione Python:

```
exec('e=3')      # e , che non era definito, diventa 3
```

La funzione "*eval*" valuta un'espessione Python, quelle che possono stare a destra di una assegnazione, e ne ritorna il risultato:

```
eval('abs(-3)+2')  # produce il valore 5
```

Le Stringhe

In Python le stringhe sono sequenze di caratteri, in codifica Unicode UTF-8, che , una volta definite, non possono essere modificate. Tutto quello che abbiamo visto per le sequenze si applica anche alle stringhe; in questa parte riassumiamo e completiamo le informazioni sulle stringhe.

Nel codice Python le stringhe sono rappresentate come una sequenza di caratteri fra apici. Possono essere utilizzati indifferentemente apici singoli, oppure il doppio apice. Se una stringa e' delimitata da apici singoli puo' contenere doppi apici e viceversa, ad esempio sono stringhe valide:

```
'12345"6789'
"12345'6789"
```

Una stringa puo' essere vuota; una stringa vuota e' definita da:

```
a=''
```

Una stringa puo' essere composta da piu' linee, se delimitata da tre apici o tre doppi apici, esempio:

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

"""
altra stringa multilinea
anche questa linea fa parte della stringa """
```

Stringhe una dietro l'altra sono concatenate, anche se sono separate da spazi:

```
a='abcd' 'efg'
a== "abcdefg"
```

Entro le stringhe hanno significato particolare alcune sequenze precedute da backslash : "\" Alcune di queste sono un residuo delle codifiche che venivano utilizzate il controllo del carrello per le stampanti a modulo continuo, altre sono usate per inserire valori in diverse codifiche.

La decodifica di questi caratteri speciali non avviene nelle stringhe "raw", che sono indicate dalla lettera *r* prima della stringa, ad esempio:

a=r"12n4g"

Alcuni di queste sequenze speciali sono:

**** : 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))

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

\ooo : carattere in notazione ottale (ooo sono le cifre ottali)

\0 : carattere nullo

Per specificare la codifica unicode si possono usare le notazioni:

\uxxxx : ove xxxx sono cifre esadecimali

\Uxxxxxxxx : ove xxxxxxxx sono cifre esadecimali

\N{name} : nome unicode del carattere

In Python 2 i caratteri fra apici rappresentavano stringhe in codifica ASCII, e per usare codifica Unicode si doveva usare la lettera *u* od *U* avanti alla stringa: a=u"asdl" . Questo non e' piu' necessario in Python 3 perche' di default la codifica e' UTF-8.

Una notazione simile a quella usata per le stringhe puo' essere usata per definire sequenze di byte. Per farlo, davanti alla stringa si mette il carattere "*b*" ; ad esempio per creare sequenza di bytes ed assegnarle il riferimento "*a*"

```
a=b'abcd'
```

Ci sono diversi caratteri che indicano che si va a capo, quello che si usa in Unix (e Linux) e': \n , il DOS non usa gli stessi caratteri di Unix per indicare la fine di una linea. \r \f \t \v sono comandi che servivano a controllare il carrello nelle stampanti a modulo continuo.

Sottostringhe

Ai singoli caratteri di una stringa od a sottostringhe si puo' accedere facilmente, utilizzando un indice, fra parentesi quadre. Il primo carattere di una stringa ha l'indice 0. Indici negativi indicano che si inizia a contare da fine linea: -1 e' l'ultimo elemento. Sottostringhe sono individuate da coppie di indici separate da :, il primo valore della coppia e' l'indice del primo carattere che si estrae, vengono estratti caratteri fino al secondo indice (escluso), quindi ad esempio 0:3 indica caratteri nelle posizioni: 0,1,2, il carattere in posizione 3 non viene estratto. Se manca il primo indice si inizia dal primo carattere, se manca il secondo si arriva a fine stringa. Un terzo valore, opzionale, indica il passo con cui si estraggono i caratteri; ad esempio un passo 2 estrae un carattere si ed uno no. Se il terzo valore manca si intende sia 1 e vengono estratti tutti gli elementi fra gli indici indicati. Passi negativi vanno all'indietro, nella sequenza dei caratteri, questo modo di estrarre elementi (slicing), vale per tutte le sequenze, non solo per le stringhe.

Esempi:

```
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]
```

Una stringa puo' facilmente essere separata in caratteri, con una assegnazione del tipo:

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

f1,f2 ed  f3 assumono i valori dei caratteri 'a', 'b', 'c'
```

Operazioni su stringhe

L'operatore "**+**" concatena le stringhe; l'operatore "*****" ripete una stringa un certo numero di volte:

Esempi:

```
a='0123456'
b='abcdefg'

a+b  e' la stringa: '0123456abcdefg'
b*2  e' la stringa: 'abcdefgabcdefg'
```

L'operatore "**in**" da risultato: *True* se una sottostringa e' compresa in una stringa:

Esempi:

```
'0' in a    vale: True
'01' in a   vale: True
'09' in a   vale: False
```

In una istruzione for su una stringa la variabile assume i valori dei caratteri della stringa. Ad esempio, l'espressione nell'esempio seguenteproduce, nei cicli del loop, un valore di "i" che assume, in ordine, tutti i valori dei caratteri della stringa

```
for i in a:
   i
```

Funzioni per le stringhe

Ci sono molte funzioni per trattare le stringhe; se: a= '0123456' ; b='abcedfg' :

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

min(a) e' '0' il carattere piu' piccolo (nella sequenza dei caratteri ASCII)

max(a) e' '6'

max(b) vale: 'g'

Alcune funzioni sono attributi dell'oggetto stringa, per cui la sintassi e' diversa:

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

b.capitalize() e' la stringa: 'Abcdefg' (primo carattere maiuscola)

b.upper() e' la stringa: 'ABCDEFG' (muta i caratteri in maiuscolo)

b.lower() muta i caratteri in minuscolo

b.replace('a','X') e' la stringa: 'Xbcdefg' , cambia il carattere a in X

b.replace('a','X',3) effettua la sostituzione per le prime 3 sottostringhe che trova (di default fa la sostituzione per tutte)

split : la funzione split, dato in argomento un separatore, crea una lista con le parti della stringa. L'argomento di default di split e' uno spazio bianco, bianchi ripetuti sono compattati; la funzione e' un attributo dell'oggetto stringa.

Esempi:

```
 b='1,2,3' ; s=b.split(',')  # produce ['1', '2', '3']

'   a   b '.split()  produce: ['a', 'b']
```

join : la funzione unisce piu' stringhe in una sola, separandole con una stringa data.

```
'-',join(s)      #  produce:  '1-2-3'
```

Esempi di Conversioni di numeri in stringhe e viceversa (qui a='123456'):

```
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'
```

Ci sono molte altre funzioni per le stringhe: funzioni che riconoscono se una stringa e' minuscola o maiuscola od un numero, per trovare sottostringhe, per contare quante volte una sottostringa e' contenuta in una stringa, per trasformare tabulazioni in spazi bianchi, per trovare in che punto di una stringa si trova una sottostringa, per vedere se una sottostringa termina od inizia con certi caratteri, per creare una stringa che ne contenga un'altra piu' piccola al centro etc. Alcune di queste funzioni sono nella tabella che segue:

```
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
```

Abbiamo gia' visto l'operatore "**%**" , che permette di trasformare numeri e stringhe in un'unica stringa "formattando" i dati in modo simile a come viene effettuato dalle funzioni di stampa del linguaggio C, ad esempio:

```
a=3
b=7.9E3
c=' xxx %s yyy %i zzz %f '% ('000',a,b)

c==' xxx 000 yyy 3 zzz 7900.000000 '
```

L'istruzione fa si che nella stringa: ' xxx %s yyy %i zzz %f ' vengono sostituiti i valori della tupla: ('000',a,b), intesi rispettivamente come una stringa, un intero, un decimale. %s %i %f sono qui indicatori di formato, rispettivamente per stringhe, interi, valori float.

Esistono parecchi specificatori di formato, ad esempio: %5.3f e' un numero float in un campo di 5 spazi, con 3 cifre per i decimali.

Come nel linguaggio C, sono possibili sostituzioni piu' complesse, in modo da creare stringhe che siano adatte a stampe di tabelle etc.

Esempi:

```
x=3
'%2d %3d %4d' % (x, x*x, x*x*x)
produce: 3   9   27
%2d indica decimale con spazio per 2 cifre, %3d, con 3 cifre.
```

Si puo' anche usare un dizionario per le variabili da formattare:

```
template = '%(motto)s, %(pork)s and %(food)s'
template % dict(motto='spam', pork='ham', food='eggs')

produce: 'spam, ham and eggs'
```

In python3 c'e' anche la funzione "**format**" per fare questo, e Rossun diceva che l'operatore "%" sarebbe stato eliminato prima o poi. L'uso della funzione format e':

> '{0} {1}, {2:.0f} you'.format(1, 'spam', 4.0)
>
> format ha come argomenti le variabili da formattare, ed e' un attributo delle stringhe. La stringa di formattazione ha i place-holders per gli argomenti rappresentati con numeri, fra parentesi graffe, seguiti da ":" e la specifica di formato.

Si puo' anche usare la sintassi seguente (keyword arguments) per specificare i place-holders

```
'{motto}, {0} and {food}'.format(42, motto=3.14, food=[1, 2])

Produce: '3.14, 42 and [1, 2]'
```

Liste

Le liste sono sequenze ordinate di oggetti enterogenei, accessibili tramite un indice numerico, che inizia da 0 , sono mutabili. Uno dei punti di forza del Python e' facilita' con cui si possono utilizzare le liste, e come si possono costruire strutture di dati complesse combinando liste e dizionari. Rispetto ai vettori, che troviamo in altri linguaggi, le liste hanno la caratteristica di contenere oggetti eterogenei, non tutti della stessa lunghezza; si possono fare anche liste di liste, o di dizionari o liste di funzioni. Le liste sono implementate internamente con referenze ad oggetti.

Una lista puo' crescere in modo dinamico, cioe' non si deve decidere all'inizio quanto sara' grande.

Operazioni sulle liste

Le liste, come sequenze, hanno tutte le caratteristiche e gran parte degli operatori che abbiamo gia' visto per le stringhe; agli elementi di una lista si accede con un indice intero, racchiuso fra parentesi quadre, e per gli indici e le slice (sezioni delle liste) vale quanto gia' visto per le stringhe. Il caso di liste di sequenze, si accede agli elementi "interni" con 2 indici, ognuno fra parentesi quadre, esempio: A[2][6]; il secondo indice e' relativo alla posizione dell'elemento della sequenze piu' "interna". A[2] e' tutta la sequenza interna.

L'operatore '+' concatena 2 liste, l'operatore '*' ripete un certo numero di volte una lista, abbiamo gli operatori "**in**" e "**not in**" , che danno vero o falso a seconda che un oggetto faccia parte o no della lista. A differenza delle stringhe, le liste sono oggetti mutabili, quindi ci sono anche operatori e funzioni per modificare liste od elementi delle liste.

Le liste si indicano con una serie di valori, separati da virgole e racchiusi fra parentesi quadre, ad esempio, per definire una lista A , composta dai numeri da 0 a 9:

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

Per copiare una lista si deve usare una sintassi del tipo: **B=A[:]** , ovvero si devono copiare gli elementi; B=A non copia la lista, crea un nuovo riferimento B alla lista A. Se modifico A mi trovo le modifiche in B. Questo contrariamente a quello che succede con elementi non mutabili: a=3 ; b=a ; a=4 : crea un oggetto "4" ed un riferimento che punta a "4", l'oggetto "3" ed il suo riferimento "b" non sono alterati

Esempi:

```
A=[0,1,2,['a','b','c'],'fgk',3.4E10]] : definizione lista complessa ove
                                        un elemento e' a sua volta una lista
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[3:5]   :  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 )

del A[3]     : elimina l'elemento numero 3 (il quarto)

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

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

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

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.
```

```
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',]
```

Gli elementi di una lista si possono assegnare a variabili, estraendo gli elementi, il numero di variabili deve essere eguale al numero di elementi della lista, in Python3 e' possibile non avere numero di variabili eguali agli elementi, ma assegnare parte degli elementi ad una nuova lista

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

b,c,*d=a

      Qui: b==1 ; c==2 ; d==[3,4,5]

b,c,*d,e=a

       Qui: b==1 ; c==2 ; d==[3,4] ; e==5
```

Si possono unire a 2 a 2 gli elementi di una lista con la funzione zip:

```
a=[1,2,3]
b=['a','b','c']
c=zip(a,b)
```

In python 2 "c" contiene: [(1, 'a'), (2, 'b'), (3, 'c')]

In python 3 "c" e' uno zip object, con dentro la sequenza: (1, 'a') (2, 'b') (3, 'c') si deve fare: cc=list(c) per avere la lista

Si puo' applicare una funzione ad una lista con la funzione map:

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

In python2 map produce una lista.

In python3 un oggetto map da trasformare in una lista con la funzione "list". list(c) contiene: [1, 2, 3]

Nei loop sulle liste e' utile la funzione "*range*", ad esempio list(range(4)) restituisce una lista contenente [0, 1, 2, 3]. Per scorrere una lista senza sapere a priori quanto e' lunga possiamo usare:

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

List comprehension

La list comprehension e' una espressione ciclica fra parentesi quadre che genera una lista:

```
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]
```

Tuple

Le tuple sono sequenze simili alle liste, ma, a differenza delle liste, sono sequenze immutabili e non possono essere cambiate nel corso del programma. Contengono oggetti eterogenei, identificati da un indice numerico. Le tuple sono immutabili, ma i loro elementi, se sono mutabili, possono essere cambiati, per cui se un elemento di una tupla e' una lista questa puo' essere alterata e svuotata, ma non puo'essere tolta dalla tupla. Tuple che contengono solo oggetti immutabili possono essere usate come chiavi in un dizionario.

Le tuple sono rappresentate come insiemi di valori separati da virgole e racchiusi fra parentesi tonde; per il resto abbiamo operatori e notazioni analoghe a quelle delle liste, dei dizionari e delle stringhe:

Esempi:

```
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'operatore di somma: '+' concatena tuple in una nuova tupla, l'operatore: '*' ripete una tupla un certo numero di volte:

Esempio:

```
T3=(2,3)
T4=(5,5)
T5=T3+T4
T5        : e' la tupla:  (2, 3, 5, 5)

T6=T3*2
T6        : e' la tupla: (2, 3, 2, 3)
```

Tuple possono essere mutate in liste e viceversa:

```
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.
```

C'e' un modulo aggiuntivo di Python (modulo "*collections*", introdotto nella versione 2.6) che introduce le "Named tuple": tuple indicizzabili per nome, tipo dizionari, oltre che per numero.

Sets e Frozensets

I sets sono stati introdotti con Python 2.4, e sono insiemi non ordinati, mutabili, di oggetti immutabili (numeri, caratteri, tuple , NON liste e dizionari). Gli oggetti sono unici nel set (non ce ne sono 2 uguali). I set sono gli insiemi della matematica insiemistica e su di essi si possono fare operazioni di unione, intersezione, differenze etc.

In Python 3, ma non in Python 2, i set possono essere rappresentati da valori fra parentesi graffe, separati da virgole.

I frozensets sono analoghi ai sets ,ma immutabili.

Esempi:

```
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'])
```

La funzione add modifica un set aggiungendogli elementi, ma non si possono aggiungere liste al set, solo solo oggetti hashable (immutabili).

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

Esistono operatori: > < >= <= per individuare subset o superset; danno True o false

Esempio.:

```
a < b : vero se b contiene tutti gli elementi di a
```

Dizionari

I dizionari sono vettori associativi, quello che in altri linguaggi di programmazione viene anche indicato con "hash" o "mappings". Somigliano a liste, ma gli indici (keys o chiavi) sono stringhe, od altri oggetti Python immutabili, ad esempio tuple.

Internamente sono implementati con l'uso di 'hash tables', per cui le chiavi devono essere oggetti "hashables", cioe' Python deve poterli trasformare internamente in indirizzi; tutti gli oggetti immutabili hanno queste caratteristiche, ed anche oggetti istanze di classi create dall'utente soddisfano questo criterio.

Si chiamano dizionari perche' l'uso di stringhe per ritrovare elementi di un insieme e' analogo a quello che si fa quando si cerca una parola in un dizionario. Si usano per trattare insiemi di elementi contraddistinti da un nome, piuttosto che da un indice numerico, come per le liste.

I dizionari sono oggetti mutabili ed un dizionario puo' contenere oggetti eterogenei: numeri, stringhe, liste, dizionari. Si possono costruire strutture complesse con dizionari e liste annidate.

Gli elementi di un dizionario non hanno un ordine definito, e per individuare gli elementi si utilizza, al solito, una coppia di parentesi quadre, che contengono la chiave dell'elemento.

I dizionari non implementano gli operatori "=" e "*" come le liste. Per aggiungere un elemento basta definirne la chiave ed il valore.

Un dizionario e' rappresentato da coppie di chiavi e valori, separati da virgole, e racchiusi fra parentesi graffe.

Esempio:

```
D={'chiave':3,'altrachiave':6,1:'abcd'}

D['chiave']       : vale 3
D['altrachiave']  : vale 6
D[1]              : vale 'abcd'

D.get('chiave')   vale 3, ma se la chiave 3 non si trova da il valore None
                  invece di dare errore, come: D['chiave']
```

Come per le liste abbiamo che:

```
D={}             : crea un dizionario vuoto
D['a']=799.0     : aggiunge elemento, se la chiave esiste gia' lo modifica
D[232]='s'

D['b']=996       : crea elemento o lo modifica se c'e'
D['b']=99        : cambia il valore dell'elemento con chiave 'b'

del D['b']       : elimina l'elemento con chiave 'b'

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

D.pop('c')       : estrae dal dizionario il valore con chiave 'c' e lo elimina.

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

D.clear()         : svuota il dizionario

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

Se abbiamo dizionari di dizionari possiamo reperire gli elementi del dizionario interno con la sintassi seguente:

```
D3={'cibo':{'carne':1,'frutta':2}} : dizionario che contiene un dizionario

D3['cibo']['frutta']              : e' l'elemento di valore 2
```

La funzione "**in**" e' riferita alle chiavi, ad esempio

```
D4={'a':0,'b':1,'c':3}

'a' in D4      :  e' True
0 in D4        :  e' False, 0 non e' una chiave di ricerca.
```

Un ciclo for sul dizionario restituisce le chiavi, ma si possono estrarre anche i valori o le coppie (chiave,valore) in una tupla:

```
for i in D4:
   i

  restituisce 'a', al secondo ciclo 'b', poi 'c'

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

sorted(D4) : solo in python 3, produce una lista ordinata di chiavi
```

In python 2 keys(), values() ed items() producevano liste, in Python3 oggetti su cui iterare e se se ne vuole fare una lista occorre convertirli con la funzione list: L=list(D4.keys())

La sintassi delle liste permette di implementare in modo compatto l'analogo dell'istruzione switch (o case) che hanno certi linguaggi di programmazione:

```
{'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
                       diverse a seconda del valore di s.
```

La funzione "dict" crea un dizionario a partire da liste di tuple, le chiavi si possono specificare in argomento o fornire in argomento una sequenza di tuple di 2 elementi: (chiave,valore):

```
D=dict(a:10,b='abc') : crea {'a':10,'b':'abc'}

D=dict( [('a',10),('b','abc')] )

D=dict( (('a',10),('b','abc'))

D=dict(  {'a': 10, 'b': 'abc'} )
```

La funzione "zip" permette di creare un dizionario a partire da 2 sequenze:

```
L1=['a','b','c'] ; L2=[0,1,2]
LX=zip(L1,L2)
LY=dict(zip(L1,L2))
```

In Python3 "zip" produce, a partire da due sequenze, un iterabile (in Python 2 una lista), con tuple di 2 elementi: ('a', 0), ('b', 1), ('c', 2)

La funzione "dict" trasforma l'iterabile in un dizionario:

{'a': 0, 'c': 2, 'b': 1}

In Python 3 (e Python 2.7) esiste una "dictionary comprehensions" ,analoga alla "list comprehension", che si puo' usare per definire gli elementi di un dizionario:

```
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
```

Funzioni

In Python le funzioni sono oggetti di base ("first-class objects") : l'istruzione **def** le crea ed assegna loro un nome, che e' un riferimento. Al solito si usano i due punti ed il rientro (indentation) per delimitare il corpo della funzione. Gli argomenti, fra parentesi tonde, seguono il nome della funzione; alla chiamata la funzione viene eseguita, producendo un oggetto, che e' il risultato della funzione e che viene restituito con un'istruzione "*return*" . In caso l'istruzione "*return*" non compaia nella funzione, viene restituito l'oggetto vuoto: "None".

La sintassi per la definizione di una funzione e' del tipo:

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

Per chiamare la funzione si utilizza una sintassi del tipo:

```
nomefunzione(r,s,t)
g=nomefunzione(r,s,t)   : qui il risultato della funzione e' assegnato a 'g'
```

Il nome della funzione e' semplicemente un riferimento alla funzione, si distingue dalla chiamata alla funzione, ove devono apparire le parentesi tonde dopo il nome. Il nome puo' stare in una lista, essere passato in argomento a funzioni, e le funzioni possono anche essere chiavi di dizionari; solo le parentesi tonde dopo il nome indicano che la funzione va eseguita.

Per vedere se un nome e' un riferimento ad una funzione si puo' usare la funzione "*callable(nome)*", che restituisce True se l'oggetto cui ci si riferisce ha l'attributo "*__call__*", ovvero se e' una funzione.

Qui sotto vediamo, come esempio, una funzione che crea funzioni (function factory):

```
def maker(N):                 # funzione "factory"
        def action(X):        # qui definisce una funzione
            return X ** N
        return action

fa=maker(2)         # creo una funzione fa che fa il quadrato
fb=maker(3)         # creo una funzione fa che fb il cubo

a(10)         # produce 10*10    =>  100
b(10)         # produce 10*10*10 => 1000
```

Argomenti

Gli argomenti sono passati per assegnazione: in Python le variabili sono riferimenti ad oggetti; nel passaggio degli argomenti, alla variabile nella funzione viene e' assegnato lo stesso oggetto della variabile corrispondente nella chiamata; cioe' viene fatta una copia dei riferimenti agli oggetti.

I tipi delle variabili vengono definiti solo all'assegnazione dei riferimenti, per cui una funzione, a priori, non sa quali sono i tipi degli argomenti ed eventuali inconsistenze producono errori solo quando la funzione viene eseguita. In questo modo Python implementa naturalmente il polimorfismo, cioe' una stessa funzione puo' essere usata per dati di tipo diverso. Ad esempio la funzione:

```
def somma(a,b):
    return a+b
```

se chiamata con: somma(3,2) produrra' 5, se chiamata con stringhe come argomenti: somma('aa','bb') restituira' la stringa 'aabb'. Se chiamata come: somma('a',1) produrra' un errore.

Riassegnare le variabili in argomento entro la funzione non ha effetti sul chiamante, ma gli oggetti mutabili possono essere cambiati nelle funzioni, operando sui loro riferimenti.

Le funzioni possono avere valori di default per gli argomenti. Ad esempio una funzione definita con:

def func(a='uno'):

puo' essere chiamata semplicemente con:

func()

ed il suo argomento a sara' il default: la stringa 'uno' ; oppure con

func('due')

ed il suo argomento a sara' la stringa 'due'

Una funzione puo' anche essere chiamata dando valori ai parametri per nome (keyword arguments), con una sintassi tipo:

func(a='sei')

in questo caso alla variabile "a" entro la funzione, viene assegnata la stringa "sei".

Una funzione puo' essere definita in modo che i suoi argomenti siano visti, entro la funzione, come una tupla o come un dizionario; le definizioni della funzione avranno in questi casi rispettivamente la sintassi:

def func(*nome):

def func(**nome):

Nel caso del dizionario gli argomenti sono passati per nome ed i nomi diventano le chiavi del dizionario.

Questi modo di passare gli argomenti possono essere combinati, in questo caso, nelle chiamate e nella funzione, vanno prima gli argomenti posizionali, poi quelli che finiscono in una tupla, ed infine, con passaggio per nome, quelli che finiscono nel dizionario:

```
def func(a,b,c):  # esempio di funzione
    print(a)
    print(b)
    print(c)


func(1,2,3)           # chiamata con argomenti passati per posizione
func( b=2,a=1,c=3)    # argomenti passati per nome
func(1,c=3,b=2)       # argomenti passati per posizione, quello che resta per nome

def func(*a):
      print(a)
      '''
      Qui in a finisce una tupla di argomenti,
      la chiamata puo' avere numero variabile di argomenti
      func(1,2,3) stampa la tupla: (1,2,3)
      '''
```

```
def func(**d):
      print(d)
      '''
      Qui gli argomenti finiscono in un dizionario
      gli argomenti sono passati per nome
      ed i nomi delle variabili sono le chiavi
      func(a=1,b=2,c=3) stampa:   {'a': 1, 'c': 3, 'b': 2}
      '''


def func(a,*b,**d):
      print(a)
      print(b)
      print(d)
      '''
      Vanno prima gli argomenti posizionali,
      poi quelli per la tupla, infine quelli per il dizionario.
      Chiamata come :  func(1, 2,3,4, s=10,q=20 )
      stampa:  a=1  ;  b=[2,3,4] ; d={s:10,q:20}
      '''
```

In Python3 abbiamo anche funzioni con argomenti passati per nome *dopo* quelli che finiscono in una lista:

```
def func(a,*b,c):

Qui l'ultimo argomento che puo' essere data solo per nome"
con chiamata:   func(1,2, 3, c=40) ,
ed avremo a==1 ; b==[2,3] ; c==40
```

Anche le chiamate possono contenere liste o dizionari:

```
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.
```

Valori restituiti

La funzione restituisce un valore specificato nell'istruzione return. Se non viene eseguita l'istruzione return il valore restituito dalla funzione e' l'oggetto speciale di nome: "*None*" che e' per definizione un oggetto vuoto. Una funzione puo' restituire una tupla, con la sintassi tipo:

return a,b,c

Analoga a "*return*" e' la funzione "*yeld*", che ritorna un valore al chiamante; ma la volta successiva che la funzione viene chiamata l'esecuzione parte da dopo l'istruzione yeld. In questo modo si possono implementare iteratori. La sintassi e':

yeld a

Campo di validita' della funzione (scope della funzione)

Una funzione e' valida dal punto del programma in cui si incontra in poi; quando la funzione viene incontrata viene "eseguita", nel senso che il suo nome (che in realta' e' un riferimento) diviene valido ed ad esso sono associate le operazioni contenute nel corpo della funzione. In questo modo e' possibile definire una funzione in modo diverso a seconda del flusso del programma: ad esempio:

```
if a>b:
   def func(a,b):
      return a-b
else:
   def func(a,b):
      return b-a
```

Queste istruzioni definiscono la funzione "func" come la differenza fra il piu' grande dei due valori in a e b. A seconda dei casi la funzione e' definita in modo diverso.

Una funzione puo' essere definita entro una funzione, ed allora e' vista solo li'.

Campo di validita' delle variabili (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.

Funzioni lambda

Sono funzioni di una sola istruzione, senza nome, con sintassi:

lambda argomento1,argomento1,argomento3: espressione

Esempio:

```
f= lambda x,y : x+y

f(2,3)    produce 5
```

Le Lambda sono usate in contesti particolari, ove e' comodo mettere una piccola funzione in una sola riga, ad esempio per costruire una lista od un dizionario di funzioni:

```
L=[ (lambda  x: x+x ) , ( lambda x: x*x) ]

for f in L :
  print f(3)      # Produce : 6 , 9
```

Esempio: 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
```

Docstring

All'inizio di una funzione puo' esserci una stringa multilinea che descrive la funzione Questa viene conservata nella variabile __doc__

```
def square(x):
   " " "
   Questa funzione eleva
   un numero al quadrato
   " " "
   return x*x
```

Attributi delle funzioni

Le funzioni, come oggetti, hanno attributi, cui ci si puo' riferire con

nomefunzione.attributo

Alcuni attributi delle funzioni sono:

__doc__ : stringa descrittiva

__name__ : nome della funzione

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

Decorators

Sono funzioni che prendono in argomento una funzione, ci fanno modifiche, aggiunte, o fanno cose accessorie, poi restituiscono la funzione modificata:

```
def decname(funcname,a,b,c):
   ... operazioni varie con la funzione funcname
   return funcname
```

C'e' una sintassi abbreviata quando si vuole applicare un decoratore ad una funzione:

```
@nomedecoratore
def f()
    .....
```

Questo e' equivalente a definire la funzione f e poi applicargli il decoratore con:

f=nomedecoratore(f)

I files

In python3 i files sono tipi, l'apertura di un file crea un oggetto 'file' e gli assegna un riferimento. Ci sono diverse funzioni per trattare i files, soprattutto rivolte a files di testo. Per i files di testo la codifica e decodifica UTF-8 dei caratteri e' effettuata dal Python in modo trasparente all'utente. Di default i files sono intesi come files di testo.

Input/Output da terminale

Ad un programma Python, anche nell'esecuzione interattiva, sono associati uno "standard" output ed uno "standard" input, da cui il programma legge e scrive di default.

Per scrivere sullo standard output si usa la funzione "*print*". In Python 2 "*print*" invece di una funzione era un comando. Questo ha creato innumerevoli problemi, dovendo, in tutti i vecchi programmi, inserire le parentesi in un sacco di posti, modificando comandi tipo: "print a" in "print(a)".

La funzione print ha un numero arbitrario argomenti e li stampa , trasformati in stringhe con la funzione str, separati da uno spazio. Ogni comando print stampa una sola linea, a meno che le stringhe stampate non abbiano dentro il carattere: "\n" , che viene interpretato come un fine linea. Assieme a "*print*" viene in genere usato l'operatore di formattazione, per scrivere stringhe entro cui si inseriscono numeri e caratteri definiti a run-time.

La lettura da terminale si puo' fare, in Python 3, con la funzione "*input*", che legge una linea e la mette in una stringa. La funzione input puo' avere come argomento un "*prompt*" che viene stampato prima della lettura. In Python 2 la funzione analoga e' "*raw_input*", mentre "*input*" esegue la funzione eval su una stringa che legge.

Esempi:

```
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: "=> "
```

La funzione "*print*" di Python3 ha diversi argomenti opzionali e la sua sintassi completa e':

```
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
```

Esempio, che stampa un valore per riga e dopo l'ultimo scrive: "====="

```
print("a","b","c",sep="\n",end="====\n")

a
b
c====
```

Uso di files

Per accedere ad un file si usa la funzione "*open*", che crea un oggetto file e ritorna un riferimento ad esso. L'oggetto file ha funzioni per accedere al contenuto del file. Ci sono anche funzioni per leggere il file tutto in una volta e metterne il contenuto in una stringa, e funzioni per farne una lista di stringhe con le singole linee.

L'output e' befferizzato, cioe' non scritto subito sul file, ma posto un un'area di memoria apposita (buffer) e scaricato sul file, tutto insieme, in un secondo momento, per ottimizzare i tempi di calcolo. La funzione "*flush*" scarica il buffer.

Il file ha un puntatore che ricorda dove si e' arrivati nella lettura, che, a diversi comandi di lettura, si sposta in avanti nel file. La funzione seek sposta il puntatore, permettendo di saltare parti del file o di rileggere contenuti gia' letti.

Il file viene chiuso con la funzione "*close*", che elimina il riferimento al file.

La sintassi completa della funzione open e':

```
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 con dati binari,
                ove NON si interpretano i caratteri
                come codifiche UTF-8

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

Gli altri parametri sono raramete usati:

        buffering : da indicazioni sull'uso e dimensione del buffer
        newline   : indicazioni sul carattere di fine linea
        errors    : su come gestire errori di codifica
        closefd   : gestisce chiusura del file

        encoding: il tipo di codifica per i caratteri: Python
                  prevede parecchie codifiche oltre UTF_8: ascii, latin-1,
                  cyrillic,  greek, UTF_16, UTF_32. In genere
                  assume la codifica di default del computer che si
                  sta usando.
```

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

f,f1 etc. sono riferimenti ai files, che sono oggetti.
filetest, filetest2 etc. sono i nomi dei files.

stringa1=f1.readline()   :   legge una linea dal file f1 e la mette in stringa1
stringa2=f1.read()       :   mette in stringa2 tutto il file f1
```

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

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.

data = open('data.bin', 'rb').read() : lettura file binario,
                                       messo in 'data' come insieme di bytes
                                       ad esempio: b'bbbbcccc\ndddd\n'
```

Ci sono sistemi per iterare sulle linee del file, in modo da elaborarne una riga per volta; ed i files possono essere usati in list comprehension per creare liste:

Esempi:

```
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 vengono a costituire una
lista.
```

Eccezioni

Sistemi software ove parti del programma reagiscono ad eventi o si scambiano messaggi sono molto comuni, fra questi abbiamo:

- le interfacce grafiche (agiscono in funzione di messaggi da tastiera e mouse)
- gli applicativi di rete (dalla rete arrivano richieste di connessione)
- i desktop utilizzano messaggi per far parlare le varie componenti (D-bus, usato in Linux da kde4 e gnome)
- il sistema operativo utilizza messaggi per far comunicare diverse parti o comunicare con programmi utente.

Un particolare utilizzo di questa tecnica puo' essere considerato il sistema che gestisce, in molti linguaggi, il verificarsi di errori: una routine, in cui si verifica un errore, lancia un'"eccezione", che e' un messaggio che puo' essere gestito dal programma stesso o puo' causare l'arresto del programma. Questo sistema puo' anche essere usato come sistema generale di comunicazione fra le diverse parti del programma.

Eccezioni in Python

In Python implementano la gestione delle eccezioni le seguenti istruzioni:

```
raise : per generare un'eccezione

assert: genera un'eccezione, di tipo "AssertionError", in base a condizioni

try : intercetta eccezioni verificatesi in una parte del codice
      ed esegue azioni in base a queste

except: definisce un blocco da eseguire
        in caso si verifichi una data eccezione
```

Le eccezioni devono essere classi da in Python 3, ed ereditare la classe Exception; in versioni di Python precedenti alla 2.6 potevano anche essere semplici stringhe.

Sintassi dell'istruzione "*raise*":

```
raise nomeistanza
raise nomeclasse
raise
raise nomeclasse from altraclasse
```

"*raise*" lancia un'eccezione, che puo' essere:

- l'istanza di una classe;
- una classe, ed in questo caso raise la istanzia automaticamente,
- un'eccezione precedente, che viene rilanciata. Rilanciare un'eccezione puo' servire in strutture *try-except* annidate, per passare l'eccezione ad un blocco superiore nella gerarchia.

L'istruzione "*try*" identifica il blocco entro cui verificare se sono state lanciate eccezioni; a questo blocco seguono uno o piu' blocchi individuati da istruzioni "*except*" , che sono eseguiti se si verificano eccezioni di un certo tipo. La sintassi e' la seguente:

```
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
  ...
```

"*try*" identifica il blocco in cui puo' essere generata un'eccezione; i blocchi che seguono le istruzioni "*except*" vengono eseguiti se l'eccezione corrisponde al nome della classe che segue except (qui: ecc1,ecc2,ecc3). Il nome che conclude la linea con "*except*" (qui: var) e' un riferimento che punta all'eccezione, che puo' essere usato nel blocco except.

Il blocco **else** e' eseguito se **non** e' sollevata un'eccezione.

Il blocco **finally** viene eseguito **in ogni caso**, sia se ci sono eccezioni sia se l'esecuzione del blocco try non ne genera. In blocco "*finally*", e' opzionale e serve ad assicurarsi che certe operazioni siano eseguite in ogni caso, anche se si verificano errori

Esempio:

```
class Ecc(Exception):     # definizione di una eccezione
   nome="eccezione Ecc"

class Ecc1(Exception):    # definizione di un'altra eccezione
   nome="eccezione Ecc1"

def eccezsub(a):
   if a<2 : raise Ecc          # lancia eccezioni se a<=2
   if a==2 : raise Ecc1
   print "OK, valore grande:",a

for i in [1,2,3] :
    print "provo il numero:",i
    try:
```

```
      eccezsub(i)                    # blocco try, da testare
      print "OK niente eccezione"
    except Ecc as X :                # cattura eccezione : Ecc
      print "valoretroppo piccolo:",X.nome
    except Ecc1 as X :               # cattura eccezione : Ecc1
      print "valore ma quasi OK:",X.nome
    else:
      print " niente eccezione, valore OK"
    finally:
      print "valore verificato:",i
print "fine delle prove"
```

Se viene lanciata un'eccezione che non corrisponde ad un except questa non viene catturata dal try, ma dal Python, che interrompe il programma. Il blocco finally viene eseguito prima dell'interruzione.

In Python 3 raise puo' avere anche una sintassi del tipo:

```
except eccezione as E
    ....
    raise nuova_eccezione from E
```

L'eccezione: "E" finisce nell'attributo "__*cause*__" della nuova eccezione lanciata. In questo modo si puo' avere una gerarchia di eccezioni, che sono elencate nel messaggio di errore.

In python 2, in quest'ultimo caso, l'istruzione "*except*" ha la forma:

```
except nomeclasse,  nome  # c'e'una virgola invece che "as".
```

Classi di eccezioni generate da Python

In caso di errori durante l'esecuzione del programma, Python genera eccezioni, alcune di queste sono elencate nella tabella seguente:

```
ZeroDivisionError : divisione per zero;   Es.: 3/0
OverflowError     : valore troppo grande  Es.: 10.0**1000
IndexError        : indice sbagliato      Es.: a=[1,2,3] ;  a[8]
IOError           : errore di input/output
ImportError       : non si trova il file da importare
KeyError          : chiavi di dizionario inesistente
                    Es.: d={'a':1} ; d['b']
TypeError         : operazione non permessa su certi dati
                    (Es.: elevazione a potenza con stringhe)
```

Esempio di come si possano utilizzare le istruzioni try ed except in modo da evitare che un certo errore provochi l'interruzione del programma:

```
a=2.0
for i in [1,0] :
    print "divido per:",i
    try:
      b=a/i
    except ZeroDivisionError :
```

```
      print "sto dividendo per zero, metto -1"
      b=-1
    finally:
      print "valore di b:",b
print "fine delle prove"
```

Istruzione assert

L'istruzione "*assert*" verifica una condizione e, se non e' vera, lancia l'eccezione "*AssertionError*". Viene usata nella fase di test dei programmi. Se Python viene chiamato con l'opzione di ottimizzazione '-O' , la variabile interna __debug__ e' False e l'istruzione assert non ha effetto.

Esempio:

```
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)
```

Programmazione ad oggetti

La programmazione ad oggetti nasce per superare difficolta' intrinseche in linguaggi come il C, che si incontrano quando si utilizzano questi linguaggi per scrivere programmi molto grandi (centinaia di migliaia di istruzioni). I concetti alla base della programmazione ad oggetti risalgono agli anni 60-70, ma la programmazione ad oggetti ha iniziato a prendere piede solo negli anni 90, col C++, ed e' stata molto usata per scrivere interfacce grafiche. E' poi diventata di moda, e tutti i linguaggi moderni come Java, Python, Ruby, sono ad oggetti; la programmazione ad oggetti e' stata inserita perfino in vecchi linguaggi come il FORTRAN od il COBOL.

La programmazione ad oggetti utilizza strutture di dati chiamate "*classi*"; le **classi** sono insiemi di dati eterogenei, e contengono anche le funzioni (dette anche "*metodi*") che operano sui dati della classe. Dati e funzioni della classe sono anche detti "*membri*" della classe. I dati sono detti "*attributi*" della classe.

Bisogna distinguere la definizione di classe dalla sua istanza. La definizione della classe ne descrive la struttura, ma di per se non riserva spazio di memoria per la classe e non crea nulla, viene trattata a tutti gli effetti come un nuovo tipo di dato.

L'**istanza** (oggetto) e' una realizzazione della classe, individuata da un nome. L'istanza ha il suo spazio in memoria, i suoi dati, le sue funzioni etc.

La classe viene definita una volta e poi se ne possono fare tante istanze, ognuna col suo nome ed i suoi dati. Ci sono dati propri delle singole istanze e dati "della classe", comuni a tutte le istanze.

Una classe ha dei **membri privati**, che sono visibili solo dall'interno della classe, e **membri pubblici**, che sono visibili dall'esterno. Si interagisce con la classe utilizzando i suoi membri pubblici, di preferenza chiamando le sue funzioni pubbliche.

L'insieme delle funzioni e dati con cui si interagisce con la classe sono chiamati "**interfaccia**" della classe. La classe viene utilizzata come una scatola nera, di cui e' noto l'uso, ma non il contenuto.

Quindi una volta definita l'interfaccia di una classe, si sa come usarla nel resto del programma e se ne puo' ignorare la struttura interna. In grossi progetti si inizia a strutturare il software definendo le interfacce delle classi. Una volta fatto questo, parti diverse del progetto possono essere assegnate a gruppi di sviluppatori diversi.

Una classe puo' essere costruita aggiungendo parti ad una classe pre-esistente. In questo caso si parla di "**ereditarieta'**"; la nuova classe ("**classe derivata**") "eredita" membri dalla classe pre-esistente, detta "**classe base**" o classe padre. La classe derivata puo' anche sostituire membri della classe base con membri definiti al suo interno, in questo caso si dice che fa l'"ovverride" di membri della classe base.

In certi linguaggi (C++) si parla di "*classi astratte*" per indicare classi che servono solo a specificare delle interfacce, che saranno poi implementate in classi derivate. In questo modo le classi finiscono per definire solo dei comportamenti, e non dicono nulla di come questi comportamenti saranno implementati. Questo approccio permette di applicare allo sviluppo del software un modello di top-down; ove prima si disegna lo schema generale del software e poi si definiscono i dettagli implementativi.

Una classe puo' essere costruita in modo da potere essere utilizzata con diversi tipi di dati (interi, float, caratteri od altro). In questo caso, in cui si utilizza la stessa interfaccia per dati diversi, si parla di "*polimorfismo*", o di "*programmazione generica*". Questo permette di ridurre i tempi di sviluppo, non bisogna scrivere classi diverse a seconda del tipo di dati che devono trattare.

Il polimorfismo puo' essere implementato in diversi modi: con i "template" del C++ si scrivono classi per tipi di dati generici, ed al momento della compilazione il compilatore crea la classe relativa al tipo di dato per cui e' richiesta. Un altro approccio e' quello in cui non si sa quale tipo di dato si utilizzera' fino all'esecuzione del programma. E solo in esecuzione viene definito il tipo di dato che verra' utilizzato. Questo di chiama "late binding" o "run time binding". In C++ questo viene implementato utilizzando puntatori ad una classe base, che ha diverse classi derivate a seconda dei tipi. Quando e' il momento viene utilizzata la classe derivata del tipo giusto. In Python i tipi delle variabili sono definiti solo all'assegnazione della variabili, e le funzioni non sono che procedure generiche da applicare alle variabili e non dipendono in principio dai tipi delle variabili. Il Python implementa in modo naturale il polimorfismo.

In certi linguaggi e' possibile definire come operano gli operatori tipo somma, moltiplicazione etc., sulle istanze di una classe. Gli operatori algebrici diventano speciali funzioni della classe, e si puo' dar senso ad operazioni algebriche fra oggetti complessi, non solo fra numeri. Questo si chiama override degli operatori.

Programmazione ad oggetti in Python

Creazioni di classi ed istanze

L'implementazione delle classi in Python e' semplice. Si riduce ad un modo di isolare ed individuare nomi di oggetti. Definire una classe significa dare ai nomi di variabili e funzioni una struttura gerarchica, che riflette la gerarchia di classi base e classi derivate. In Python le classi sono semplicemente contenitori di nomi.

I programatori Python hanno l'abitudine di usare per i nomi delle classi nomi "capitalized", ovvero identificativi composti da parole con la prima lettera maiuscola. Questa convenzione non e' obbligatoria, ma e' sempre raccomandata.

Una classe e' definita con l'istruzione class:

```
class C3:
  """ blocco che definisce la classe"""
  a=[1,2,3]
  b="abc'
  ....
```

Questa istruzione definisce la classe di nome C3; variabili, e funzioni della classe sono definite dopo l'istruzione class, il blocco della definizione della classe, al solito, e' identificato da un rientro. Le variabili definite entro una classe sono locali alla classe (private), sono accessibili alla classe ed alle sue derivate. Per accedere ad esse fuori della classe bisogna che siano identificate in modo esplicito come membri della classe.

```
class C1(object):
  kk=3
  ...
  ...


class C3(C1,C2):
      """ classe C3 che eredita dalle
          classi C1 e C2 ""
      a=66
      .........
      .........
```

Questa sintassi indica che la classe C3 eredita i membri delle classi C1 e C2. I membri di C1 e C2 sono riconosciuti come membri di C3. La classe C1 eredita da object, la classe 'padre' di tutte le classi in Python. In Python 3 tutte le classi, anche se non lo si specifica, ereditano da object. In Python 2 occorre specificarlo, oppure si creano di classi di vecchio tipo, che mancano di alcune caratteristiche ed hanno diverso modo di cercare i nomi degli attributi nella gerarchia definita dall'ereditarieta'.

Per creare istanze della classe C3 si scrivono istruzioni del tipo:

```
oggetto1=C3()
oggetto2=C3()
```

ora oggetto 1 ed oggetto2 si riferiscono a 2 diverse istanze della classe C3; sono in pratica riferimenti a due diverse copie di C3. Occorre fare attenzione a non dimenticare le parentesi tonde: oggetto1=C3 non crea un istanza, ma un nuovo riferimento per la classe C3.

Per distruggere un'istanza si assegna al nome dell'istanza la stringa vuota:

```
a=NomeClasse()
b=NomeClasse()

b=""   # L'istanza 'b' non ha piu' riferimenti e viene eliminata.
```

Attributi

Le variabili di una classe sono chiamate *attributi*, le sue funzioni *metodi*. Insieme, attributi e metodi sono i *membri* della classe. Nel programma Python, una volta istanziata la classe, gli attributi dell'istanza si indicano con: nomeistanza.nomeattributo

Esempio:

```
class NomeClasse(object):
    kk=3

a=NomeClasse()
b=NomeClasse()

      a.kk vale 3
      b.kk vale 3
```

Esistono attributi della classe (validi per tutte le istanze) ed attributi della singola istanza. Nell'esempio sopra kk e' un attributo della classe, tutte le istanze lo hanno eguale quando sono create, ma poi si puo' cambiare. Se lo si cambia riferendosi all'istanza si comporta come appartenente all'istanza e viene ridefinito come una variabile dell'istanza, se invece si cambia riferendosi alla classe cambia per tutte le istanze che non lo hanno ridefinito.

Ad esempio, se ridefinizione di kk per la sola istanza 'a':

a.kk=10

b.kk vale ancora 3, NomeClasse.kk vale 3, ma se ridefinisco kk per la classe:

NomeClasse.kk=100

a.kk resta 10, dato che e' stato cambiato nell'istanza, ma ora b.kk e' cambiato, e se creo una nuova istanza questa avra' il nuovo valore di kk.

Un attributo della classe puo' anche essere definito da 'fuori' della classe:

nomeclasse.jj=77

a.xyz=63

Ora sia b.jj che a.jj valgono 77, se invece definisco un attributo solo per un'istanza quello vale solo per l'istanza ove lo ho definito:

a.xyz=63

L'istanza 'a' ha un nuovo attributo xyz, ma b.xyz , NomeClasse.xyz non esistono

Gli attributi della classe sono conservati internamente in un dizionario della classe, chiamato: __dict__ Il dizionario puo' essere stampato con:

print(NomeClasse.__dict__)

Anche l'istanza ha il dizionario __dict__ ove sono solo gli attributi propri dell'istanza e non della classe, per vederlo:

print(a.__dict__)

Gli attributi che iniziano con un doppio underscore: "__" sono locali alla classe e non sono visti da fuori. In realta' sono solo nascosti ed hanno nome: "_nomeclasse__nomeattr" , per cui per un'istanza della classe si trovano come: "nomeistanza._nomeclasse__nomeattr" , per la classe come: "nomeclasse._nomeclasse__nomeattr"

Anche gli attributi che iniziano con un singolo underscore: "_" sono, per abitudine dei programmatori, attributi locali, ma questi non sono neanche nascosti.

Docstring

Come per le funzioni, anche nelle classi e' buona norma inserire una descrizione della classe all'inizio, in una stringa che viene conservata nella variabile "*__doc__*" della classe che puo' essere stampata con la funzione "*print*" ed e' mostrata anche dalla funzione "*help*". Sia print che help possono essere chiamate con la classe come argomento o con un'istanza come argomento:

```
class NomeClasse(object):
   ''' classe di prova
   per fare prove '''
   kk=3

a=NomeClasse()

help(NomeClasse)          # ma anche help(a)
print(NomeClasse.__doc__) # ma anche print(a.__doc__)
```

Metodi

Le funzioni definite entro le classi, chiamate metodi della classe, devono avere come primo argomento la parola *self* che identifica l'istanza su cui opera la funzione. Le classi in Python non sono che sequenze di operazioni effettuate su variabili individuate da nomi ed occorre distinguere su che istanza si sta operando. Entro una funzione "self" si usa per indicare che una variabile od una funzione appartiene all'istanza della classe e non alla classe in genere. In Python3 esistono istruzioni speciali, come il decoratore @classmethod, che permettono eccezioni a questa regola, ma in genere abbiamo:

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

Entro le funzioni, le variabili di classe o di istanza, e non locali alla funzione, vanno precedute dal nome della classe, con il punto, oppure da self, che punta all'istanza. Altrimenti sono solo interne alla funzione e non sono viste da fuori.

La funzione viene chiamata sull'istanza, con:

```
d=ProvaUno()
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
```

Notare come nella chiamata self non ci sia; siccome la chiamata e' qualificata con il nome dell'istanza davanti, Python sa gia' quale e' l'istanza, e nella chiamata l'istanza e' sottintesa. L'interprete Python la mette lui automaticamente.

Si puo' anche chiamare una funzione sulla classe, ma in questo caso si deve dare l'istanza come primo argomento:

```
ProvaUno.somma(d,100)  # restituisce (1010, 1003)
```

In Python3, se la funzione non utilizza variabili dell'istanza, questa si puo' non mettere in argomento.

Altro esempio:

```
class C3:
   " stringa di documentazione della classe C3"
   a=6
   def printa(self):
     print( C3.a )
   def f(self):
     print( "f di C3, istanza:",self)
   def somma(self,c,d):
      self.b=7
      return c+d+C3.a+self.b


I1=C3()
I2=C3()

I1.f()

"""
Qui chiamo la funzione f sull'istanza I1,
l'istanza e' passata alla funzione in modo
automatico, nell'argomento self.
f stampera' la stringa:
"f di C3, istanza: <__main__.C3 object at 0x1b84ad0>"
"""

I1.somma(1,2)  # ottengo il valore: 16

a e' una variabile della classe e non dell'istanza.

C3.a=100       # cambio una variabile della classe

a questo punto:

I2.somma(1,2)  # Ora ottengo 110
I1.somma(1,2)  # anche qui, 110
fornisce 110, come anche I1.somma(1,2)


class C4:
   " stringa di documentazione della classe C4"
   a=6
```

```
   def printa(self):
     print( C4.a )
   def f(self):
     print (" f di C4, istanza: ",self)
   def somma(self,c,d):
      self.b=7
      return c+d+C4.a+self.b
   def somma2(self,c,d):
      self.b=7
      return c+d+self.a+self.b

I4=C4()

I4.somma(1,2) ed I4.somma2(1,2) mi danno il valore 16.
Nella funzione somma2 a contiene infatti il numero 6, sia come
valore della classe che come valore dell'istanza.
Se cambio solo come variabile di istanza:
I4.a=1000
ho che I4.somma(1,2) mi da sempre 16, mentre I4,somma2(1,2) mi da 1010.
```

Inizializzazione delle istanze

La funzione __init__ , se presente, viene chiamata automaticamente quando si crea un'istanza: Gli argomenti della __init__ sono gli argomenti dell'istruzione che crea l'istanza:

```
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
```

Prima di __init__ viene chiamata __new__ ; funzione usata per cose particolari, come generare classi diverse in funzione di certi parametri (class factory) e vari trucchetti. Questo non avviene per le classi di vecchio tipo del Python 2.

La funzione __del__ viene chiamata prima che l'istanza venga distrutta, anche se la distruzione avviene ad opera del sistema automatico di garbage collection di Python.

Ereditarieta', dettagli

Abbiamo visto che nell'ereditarieta' la classe figlia ha , oltre ai suoi, anche gli attributi della classe padre:

```
class A(object):
   k=3

class B(A):      # eredita A
```

```
   j=5
```

B.k vale 3, la classe B ha k B.j vale 5, ma A.j non esiste

Per vedere le relazioni fra classi si possono usare le funzioni:

```
issubclass(derivata,parent) : che da true se e' una sottoclasse
isinstance(istanza,classe)  : che da true se e' una istanza
```

In Python esiste l'ereditarieta' multipla ed una classe puo' ereditarne diverse:

```
class AA(object):
   aa=222

class BB(object):
   bb=444


class CC(AA,BB):
   pass
```

Qui la classe CC non contiene nulla di suo (ha solo l'istruzione "*pass*"), ma ha sia l'attributo aa, che l'attributo bb.

Quando la struttura dell'ereditarieta' e' complicata puo' essere che un attributo con lo stesso nome compaia in diverse parti della gerarchia (il modo di cercare nella gerarchia e' chiamato: *MRO: Method Resolution Order*). In Python 3 la ricerca degli attributi avviene salendo di un livello e cercando, da sinistra a destra, in tutte le classi del livello superiore, poi salendo ancora di un livello e cosi' via. Le vecchie classi del Python 2 invece risalivano tutta la gerarchia relativa al primo "parent" da sinistra, poi tutta la gerarchia relativa al secondo e cosi' via.

Uso attributi in una classe derivata

Contrariamente a quanto ci si aspetterebbe, in una classe derivata, od entro funzioni di una classe derivata, non si possono usare, senza qualificarli, attributi della classe padre, non si puo' quindi mettere:

```
class A(object):
   k=3

class B(A):
   j=5
   jj=j+k       # Questo, con k nella classe parent, non funziona

B.jj=B.j+B.k    # Questo e' corretto
```

In questo caso Python cerca k nello scope globale e non lo trova, o, se trova un reference di nome k, usa quello. Questo perche' in Python gli statements dentro la classe sono valutati *prima* che la classe venga effettivamente creata e si definiscano le regole di ricerca di attributi nello spazio dei nomi. In Python la gerarchia delle classi e' infatti solo un criterio di ricerca nello spazio dei nomi.

Invece tutto va bene se jj=j+k viene definito dopo la definizione della classe, ad esempio, se fuori della classe si ha:

    B.jj=B.j+B.k

Qui infatti k e' riconosciuto come membro di B, in quanto ereditato da A (Vedi: http://stackoverflow.com/questions/9760595/accessing-parent-class-attribute-from-sub-class-body ed anche:

http://bugs.python.org/issue11339

Problemi analoghi si incontrano se, entro una classe, si usano metodi della classe senza qualificarli con il nome della classe o con self. Infatti nell'eseguire gli statements dentro la classe Python semplicemente gli mette davanti un qualificatore, e se, nell'uso, il qualificatore manca, il programma da errore. Per cui la classe seguente da errore, quando si chiama la funzione *addtwice*:

```
class Bag:
def __init__(self):
    self.data = []
def add(self, x):
    self.data.append(x)
def addtwice(self, x):
    add(x)
    add(x)
```

Funziona se si mette (vedi "The Python Tutorial" , di Van Rossum):

```
class Bag:
def __init__(self):
    self.data = []
def add(self, x):
    self.data.append(x)
def addtwice(self, x):
    self.add(x)
    self.add(x)
```

Funzioni di classi ereditate

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:

```
class AA(object):
   def print3(self):
      print("3 in AA")
class BB(AA):
   def print33(self):
      AA.print3(self)    # chiamo print3 della classe AA, dandogli l'istanza
      print("33 in BB")

b=BB()
b.print33()      # stampa : "3 in AA"  e, nella riga seguente: "33 in BB"
```

Override attributi

La classe figlia puo' ridefinire attributi della classe padre:

```
class A(object):
   k=3

class B(A):      # eredita A
   j=5
```

```
class C(B):      # eredita B
   k=33          # ridefinisce K, delle classe A
```

Qui C eredita B che eredita A, ma C ridefinisce k, per cui A.k e B.k restano 3, ma C.k vale 33

Inizializzazione ed ereditarieta'

L'inizializzazione di una istanza e' effettuata dalla funzione __init__. Occorre pero' tener presente che una classe non chiama in modo automatico la __init__ della classe padre. Questa operazione, se necessaria, deve essere fatta in modo esplicito nella classe __init__ della figlia, con istruzione del tipo:

```
super.__init__(self,..)
```

Overloading operatori e funzioni speciali

Gli operatori algebrici possono essere ridefiniti per una classe, facendoli corrispondere a funzioni speciali della classe; In questo modo e' possibile definire operazioni fra oggetti complessi con la sintassi delle normali operazioni algebriche.

Questo viene fatto definendo le funzioni speciali; queste sono le funzioni vengono chiamate da Python quando incontra operazioni fra istanze della classe.

Ad esempio, una classe che descrive vettori puo' definire una somma ed un prodotto vettoriale con:

```
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
```

Ci sono di queste funzioni per i confronti, gli operatori logici etc.:

```
__add__(self,other) , __sub__(self,other) ,__div__(self,other)
__lt__ (self,other), __le__(self,other)
__eq__(self,other) , __ne__(self,other)
__gt__(self,other) ,  __ge__(self,other)
```

Ci sono tutti gli operatori, perfino un operatore "*__call__*" : usato nel caso la classe sia chiamata come fosse una funzione, con una coppia di parentesi ed argomenti.

Sono importanti alcune funzioni chiamate quando si cerca, o non si trova, un attributo. Queste permettono di definire l'attributo 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.

       Quando una classe simula una lista sono utili anche, per
       operazioni sugli elementi della sequenza che la classe simula:
       __setitem__ , __delitem__ __len__ __contains__ __index__

       Ci sono anche  operatori per creare iteratori sulla sequenza:
       __iter__ ,__next__
```

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__  : sono il nome della classe ed un
                       puntatore alla classe stessa

__bases__ : tupla di classi base (classi da cui si eredita)
            La classe base di tutte e': object.

__dict__  : dizionario di attributi della classe
```

Decoratori di classi

In Python > 2.6 ci sono i decoratori anche per le classi. sono funzioni che prendono in argomento una classe, ci fanno modifiche, aggiunte, o fanno cose accessorie, poi restituiscono la classe modificata.

Esempio:

```
def decname(classname,a,b,c):
   ... operazioni varie con classname
   return classname
```

C'e' una sintassi abbreviata quando si vuole applicare il decoratore ad una classe:

```
@decname
class classname(object):
```

Alcuni decoratori predefiniti sono:

- **@staticmethod :**

  e' un decoratore che rende una funzione statica, ovvero che viene chiamata senza riferirsi ad un'istanza, ma e' una funzione della classe, unica per tutta la classe. (vale per python > 2.2)

- **@classmethod :**

  e' come @staticmethod, ma in automatico ha come primo argomento il nome della classe.(python > 2.2)

- **@abstractmethod :**

  in Python3, crea funzione astratta (virtuale) in una classe padre. Questa funzione deve essere ridefinita nelle classi figlie. Ed una classe con metodi astratti non puo' essere istanziata direttamente, ma solo ereditata.

Metaclassi

Internamente Python crea le classi (le definizioni delle classi) istanziando la classe: "*type*", che, e' una "*metaclasse*", cioe' una classe le cui istanze sono delle classi.

E' possibile estendere la classe "*type*" e creare una propria metaclasse, ove si ridefiniscono le funzioni __init__ e la __new__ in modo da modificare il comportamento di base dela classi. Per utilizzare la propria metaclasse invece della metaclesse *type* si usa una sintassi del tipo:

class nomeclassse(metaclass=nomemetaclasse):

Moduli

Programmi python possono essere organizzati in files, contenuti in una gerarchia di directory.

Un file costituisce un "modulo", un insieme di istruzioni e dati auto-consistente. Il nome del file e' il nome del modulo, senza il suffisso, che per files con istruzioni Python e' "*.py*" . Ma i moduli possono essere in un formato compresso (con zip), ed in questo caso hanno suffisso "*.egg*" (da Python 2.6), oppure possono essere files compilati, scritti con un altro linguaggio, ed in questo caso hanno suffisso "*.so*" .

I nomi dei files contenenti i moduli sono soggetti alle stesse regole dei nomi di variabili, infatti Python usa il nome del file, senza suffisso, come riferimento al modulo. Per cui nomi con spazi o caratteri speciali non sono accettati.

I file dei moduli possono iniziare con una stringa che descrive il modulo e viene conservata nell'attributo __doc__ del modulo stesso. Un dizionario con tutti gli oggetti, le variabili e le funzioni del modulo e' messo da Python nell'attributo __dict__ del modulo.

Diversi moduli sono organizzati in "**packages**", che occupano una directory. Il nome del package e' il nome della directory. Per essere considerata un package una directory deve contenere un file di nome __init__.py , che puo' anche essere vuoto, ma in genere contiene istruzioni che che inizializzano il package.

Un package puo' contenere subpackages, in sottodirectory.

I moduli hanno due funzioni principali:

- costituiscono software riutilizzabile
- identificano ed isolano un insieme di nomi di oggetti: un modulo definisce infatti uno spazio dei nomi (namespace), in cui python cerca i nomi di variabili, oggetti, funzioni.

  Per riferirsi ai nomi del modulo occorre usare il nome del modulo come prefisso. Ad esempio il dizionario __dict__ del modulo sys sara' accessibile con la sintassi: sys.__dict__ .

Python cerca i files con i moduli nella directory corrente, poi nelle directory specificate nella variabile di ambiente: di nome *PYTHONPATH* , nelle directory delle librerie standard, oppure in directory indicate in un file con estensione *.pth* , nella directory principale del python (variabile di ambiente PYTHNHOME), infine nelle directory specifica del computer per pacchetti ausiliari (site-packages o dist-packages).

Per vedere i percorsi utilizzati per cercare i moduli si deve esaminare la variabile *path* del modulo di sistema *sys*, che contiene la lista delle directory ove si cercano i moduli. Ad esempio, in Linux Debian 7, per Python3, senza aver assegnato PYTHONPATH, si ha:

```
['', '/usr/lib/python3.2',
     '/usr/lib/python3.2/plat-linux2',
     '/usr/lib/python3.2/lib-dynload',
     '/usr/local/lib/python3.2/dist-packages',
     '/usr/lib/python3/dist-packages'
]
```

Per utilizzare un modulo in un programma bisogna "importarlo". In modo che Python possa eseguire il codice del modulo, costruire le classi in esso contenute ed organizzare i nomi degli oggetti. La byte-compilation di un modulo viene effettuata quando il modulo viene caricato.

Il comando per importare un modulo in un file *A.py* e':

```
import A
```

a questo punto oggetti definiti nel file *A.py* possono essere utilizzati riferendosi ad essi con un nome tipo: A.nomeoggetto

L'istruzione *import* importa moduli una volta sola nel programma, ulteriori istruzioni *import* per lo stesso modulo non vengono eseguite.

Per riferirsi agli attributi del modulo usando un prefisso a scelta, invece del nome del modulo si usa la sintassi:

import A as newname

Qui ci si riferisce ad un attributo con: newname.attributo , invece che A.attributo , che non vale piu'.

L'istruzione *from* permette di importare solo alcuni oggetti da un modulo, ed i nomi vengono inseriti nel namespace corrente, per cui non occorre piu' il nome del modulo come prefisso. La sintassi e':

from nomefile import nome,altronome

Per usare un nome diverso per un oggetto importato:

from nomemodulo import nomeoggetto as altronome, nomeoggetto2 as altronome2

Per importare tutti i nomi del modulo nel namespace corrente:

from nomefile import *

Se il file con il modulo viene modificato occorre reimportare il file e rieseguire l'istruzione *from*. In Python 2, per ricaricare un modulo c'e' lo statement *reload*. in Python 3 c'e' una funzione, che fa parte del modulo *imp*:

import imp

imp.reload(nomemodulo)

Se i moduli sono in una gerarchia di packages (e di directory) si importano con istruzioni tipo:

import nomepackage1.nomepackade2.modulo

Se si vuole importare un package si usa import con il nome della directory del package e Python esegue il file __init__.py che trova in questa directory. Spesso questo file importa i singoli file del package.:

import nomedir

Se si importa un sub-package con un'istruzione tipo:

import nomedir.nomesubdir.nomesubdir

Vengono eseguiti nell'ordine, i files __init__.py che Python trova nelle diverse directory.

Siccome le directory in cui si cercano i moduli sono nella lista path del modulo sys, si possono aggiungere directory a run time con istruzioni tipo:

```
import sys
sys.path.append('/dir/subdir/')
```

Come esempio poniamo di avere un modulo costituito da un file : Esempio_modulo.py , contenente:

```
"""
Esempio di modulo, che contiene alcuni
attributi e classi
"""
a=1
class A(object):
  AinA=32
class B(object):
  def __init__(self,u,v):
     self.x=u
     self.y=v
```

```
  def printargs(self):
     print( "args:",self.x,",",self.y )
```

Si possono importare gli oggetti del modulo ed eseguire le funzioni con

```
import Esempio_modulo as Es

print( Es.__doc__ )      # stampa la docstring del modulo
print( Es.a       )      # stampa attributo 'a' del modulo
print( Es.A.AinA  )      # stampa attributi della classe 'A'

ist=Es.B(1,2)            # istanzio la classe  'B'
ist.printargs()          # esegui funzione della classe
```

Per importare i nomi nel namespace corrente:

```
from Esempio_modulo import *
A.AinA                        # 'A' e' senza prefisso
```

Libreria standard

Python viene distribuito assieme ad un'ampia collezione di moduli, che costituiscono la libreria standard.

Il modulo "**sys**" contiene funzioni per interfaccia con i comandi che fanno partire il Python, fra queste:

```
import sys

  sys.argv                       : argomenti del programma principale
  sys.exit()                     : esce dal programma
  sys.modules                    : moduli caricati
  sys.path                       : search path dei moduli
  sys.ps1 sys.ps2                : prompt del python
  sys.stdin,sys.stderr,sys.stdout : input/output di default
```

Il modulo "**os**" ha funzioni di interfaccia con il sistema operativo, ha funzioni per gestire i files etc. etc. puo' fare quasi tutto quello che si fa da una shell di Unix:

```
import os

   os.system('pwd')          : esegue comando di shell
   os.environ                : variabili di ambiente
   os.putev(nome,valore)     : aggiunge variabile di ambiente
   os.uname()                : nel sistema in Unix
```

Il modulo "**re**" serve per le espressioni regolari, ad esempio:

```
import re
pobj = re.compile('hello[ \t]*(.*)') : crea espressione regolare
mobj = pobj.match('hello world!')    : fa il match, da True o False
mobj.group(1)                        : sottostringhe espressione regolare
```

Per le funzioni trigonometriche e varie funzioni matematiche c'e' il modulo "**math**" , per date ed ora i moduli "**time**" e "**datetime**" "**pickle**" e "**json**" sono moduli per la serializzazione (trasformare strutture complesse in stringhe), "**tkinter**" serve per l'interfaccia al linguaggio "*tk*" per fare interfacce grafiche, Ci sono poi moduli per accesso a database, per leggere e scrivere files in formato csv, per costruire applicazioni di rete ed altri.

Moduli ausiliari si trovano in rete, in http://pypi.python.org/ c'e' un vasto indice di moduli. Ci sono programmi appositi (easy_install, pip), per recuperare ed installare packages da questo archivio.

Fra questi moduli ausiliari possono essere utili *numpy* che contiene classi per trattare vettori e matrici; matplotlib per fare grafici, scipy per analisi dati scientifici.