Cos'è Python

Introduzione e un po' di storia

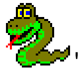
La presente guida intende portare a conoscenza di tutti i programmatori (o gli aspiranti tali) le caratteristiche del linguaggio python, fornire gli elementi di base per la sua comprensione, permettere rapidamente di scrivere semplici programmi e avere una veloce panoramica degli innumerevoli campi in cui si può applicare.
Sarebbe impossibile essere esaustivi in quanto il materiale riguardante python presente in rete è vasto ed in continua crescita.
Comunque cercando di abbinare ad ogni concetto introdotto qualche esempio esplicativo, spero di riuscire a dare alla guida un taglio sufficientemente pratico da permettervi di essere operativi in tempi brevi.

Il mio augurio è che questa manuale sia uno stimolo a scaricare subito dalla rete il linguaggio (presso il sito ufficiale www.python.org) e a cercare informazioni su python nel vasto mondo di internet. Tutto questo cominciando, soprattutto per chi non mastica molto l'inglese, dal sito italiano www.python.it

Un po' di storia

Python è stato creato da Guido Van Rossum, ricercatore di Amsterdam che avendo lavorato ad un progetto di un linguaggio di programmazione con fini didattici di nome ABC, è riuscito a trasferire questa conoscenza in python.

Il nome del linguaggio non ha niente a che spartire con il rettile. Guido Van Rossum stava cercando un nome breve e semplice, mentre seguiva una commedia trasmessa dalla televisione di nome "The monty Python's Flying Circus", ecco l'ispirazione :

"**Python**".

Caratteristiche del linguaggio

Python viene definito un linguaggio di scripting orientato agli oggetti. Infatti esso raccoglie in se la flessibilità e la semplicità dei linguaggi si scripting con la potenza di elaborazione e la ricchezza di funzioni dei più tradizionali linguaggi di programmazione di sistema.

Riporto di seguito le caratteristiche salienti del linguaggio:

Python è free. Questo per utenti linux è normale, ma fa piacere sottolinearlo condividendo profondamente i principi del software Open Source. In ambiente Windows Python potrebbe anche sostituire Visual Basic, liberandosi da tutti i problemi di licenza.
È sufficiente consultare periodicamente il sito www.python.org per rendersi conto come python, pur essendo distribuito gratuitamente, ha un notevole supporto tecnico e ha una comunità in costante crescita.

Python è portabile. Python è stato scritto in ANSI C, quindi la sua portabilità deriva direttamente da quella del C. Questo ha permesso di scrivere presto un interprete python per le principali piattaforme. Esiste un interprete python per Unix, Linux, MS-DOS, MS-Windows (95,98, NT e 2000), Macintosh, Amiga, BeOS, OS/2, VMS, QNX.
Recentemente è stato scritto un interprete anche in java e anche per sistemi Palmari. Se avete un interprete python per il vostro sistema operativo siete a cavallo. Basta prendere un sorgente python ed eseguirlo con fiducia, il risultato è strabiliante.

Python è veloce. Python è un linguaggio interpretato. In questo caso "interpretato" non è sinonimo di lento, infatti python "compila" il proprio codice in un bytecode molto efficiente. Questo permette di raggiungere prestazioni vicine ai linguaggi in codice nativo. Inoltre python implementa molte strutture dati e funzioni come componente intrinseca del linguaggio. Queste strutture sono dette "built-in types and tools" e sono state sviluppate con accurata efficienza.

Python gestisce la memoria automaticamente. Analogamente a cio' che avviene in Java, in python esiste il meccanismo di "garbage collection", il quale permette di liberare il programmatore dall'ansia di allocazione selvaggia della memoria.

Python ha una sintassi chiara. Python presenta una sintassi pulita e sintetica. L'idea migliore è rappresentata dalla indentazione, che non serve più al programmatore per ordinare meglio il codice, ma diventa l'unico strumento per strutturare il codice.
Questo permette un apprendimento più veloce e una maggiore facilità a leggere il codice scritto da altri.

Python è ricco di librerie. Solo la dotazione standard offre numerose librerie alle quali si aggiungono moduli di terze parti che crescono continuamente.
In internet si trova materiale relativo a HTML, PDF, XML, formati grafici, CGI e perfino interi web servers.

Tutte queste caratteristiche stanno convincendo molti grandi attori del mercato informatico ad utilizzare python.
Basta citarne alcuni:

- Red Hat ha implementato in python il proprio tool di installazione.
- Infoseek usa python nei propri prodotti per la ricerca sul web.
- Yahoo! ha sviluppato in python alcuni servizi di internet.
- La NASA usa python per implementare i sistemi di controllo delle proprie missioni.
- Infine ci sono anche io che sviluppo in python una intranet aziendale.

L'installazione

Prima di cominciare la nostra avventura nel linguaggio python, mi sembra giusto indicarvi le modalita' di installazione del software.
Tutto il materiale si trova sul sito ufficiale di python, in particolare nella sezione "Download".
Per l'installazione è sufficiente scegliere la propria piattaforma e seguire le istruzioni presentate nel sito. Nel momento in cui sto scrivendo questo manuale è presente la versione 2.0.
Vediamo in dettaglio l'installazione sulle due principali piattaforme del momento:

- **Piattaforma Linux**:
  Per gli utenti del pinguino potrebbe non essere necessario nemmeno l'installazione. Python, infatti, è gia' compreso nelle installazioni standard di tutte le principali distribuzioni. In ogni caso sul sito trovate sia la versione in formato "RPM", che quella in formato "TAR.GZ".
- **Piattaforma Microsoft Windows**:
  Per gli utenti Windows è necessario scaricare il file eseguibile "EXE". Eseguendo il file, Python si autoinstalla aggiungendo subito un gruppo di icone nel menu "Programmi". In particolare è gia' disponibile un interprete da utilizzare dal DOS e un mini IDE con un discreto debug simbolico.

L'interprete interattivo

Come nella migliore tradizione Unix, python si presenta sottoforma di interprete interattivo a riga di comando. Quindi una volta digitata la parola "python" appare uno specifico prompt, nel caso di python si tratta di 3 caratteri di maggiore **(>>>)**.
Basta digitare un comando e si ottiene subito una risposta. La prima cosa che si fa è vedere se l'interprete riconosce le espressioni numeriche, dopodiche' si introduce qualche variabile:

```
>>> 14 + 2
16
>>> 7 * 2
14
>>> a = 6
>>> b = 2
>>> a / b
3
```

Per uscire dall'interprete e tornare al sistema operativo ospite basta premere **CTRL+D**.
Per eseguire un insieme di comandi python in sequenza senza interattivita' è necessario:

- Creare un file ASCII con un qualsiasi editor.
- Scrivere la sequenza di comandi.
- Salvare il file con estensione "py".
- Passare il file come parametro all'interprete python.

In output si ottiene il risultato:

```
c:\>python prova.py
16
14
3
c:\>
```

In realtà se fate la prova con il codice precedente non vedete niente in output, poiché ogni espressione deve essere preceduta dalla parola "print". Ma in questo momento è importante che capiate il concetto, ci sara' modo in seguito di fare la prova.

L'interprete interattivo è un strumento semplice, anche un po' rudimentale, ma veramente utile. Sapete quante volte, incerto sulla sintassi o sul funzionamento di una istruzione, invece di scartabellare il manuale, ho aperto una sessione interattiva e ho fatto qualche prova. Ho subito trovato la soluzione !.

Primi strumenti di lavoro

In python non è necessario definire le variabili prima di utilizzarle, non è necessario nemmeno assegnare ad esse un tipo. Il tutto avviene implicitamente mediante l'istruzione di assegnamento (=), esattamente come avveniva nel vecchio BASIC.
Per visualizzare in output il valore di una variabile è sufficiente utilizzare il comando "print".
In realtà quando si lavora in interattivo è sufficiente digitare il nome della variabile per vedere in output il suo contenuto. Questa è una forzatura inserita solo quando si lavora interattivamente per permettere un semplice debug. Quando si esegue un programma è necessario utilizzare il comando print.

```
>>> a = 12
>>> b = 3
>>> print a,b,(a - b)
12 3 9
```

Si noti la flessibilità del comando print, il quale concatena il valore di diversi parametri suddivisi dalla virgola e li mostra in output.

Quando si costruiscono piccoli programmi da utilizzare dalla linea di comando è molto utile avere a disposizione delle funzioni che permettono di prelevare input da tastiera dall'utente in modo interattivo.

Per fare questo python mette a disposizione due funzioni:

- **input** per ottenere un numero
- **raw_input** per ottenere una stringa alfanumerica

Ecco due esempi:

```
>>> valore = input('mi dai un valore ?')
mi dai un valore ? 5
>>> print valore*valore
25
>>> raw_input('mi dai una stringa ?')
mi dai una stringa ? html.it
html.it
```

In realtà c'è un modo per inserire una stringa anche con il comando input, basta inserire al prompt la stringa racchiusa tra le virgolette, in questo modo python interpreta il valore come stringa e assegna alla variabile il tipo stringa.

Tipi di dati e operatori

Python possiede tutti i classici tipi di dati, comuni al linguaggio C (dal quale è nato) e a molti altri linguaggi.
Per chiarezza riassumo i tipi di dati nella seguente tabella:

Tipo di dato	Rappresentazione interna	Esempi
Intero	32 bit (tipo long del C)	1200, -56, 0
Intero lungo	oltre 32 bit con crescita in base alle esigenze	999999999L, -3232323L
Reale	32 bit (tipo double del C)	1.23 3.14e-10, 4.0E210
Booleano	intero con 1=VERO e 0=FALSO (come in C)	0, 1
Complesso	coppia di numeri reali	3+4j, 5.0+4.1j, 3j
Stringhe	lista di caratteri	'stefano', "l'acqua"

È interessante notare come non ci sia limite ad un numero intero, purchè sia seguito da un carattere "L" o "l", in base al valore python alloca la memoria necessaria.
Quando si usa una stringa è possibile racchiudere il suo valore indifferentemente con il carattere «**"**» oppure «**'**». Questo permette di superare facilmente il problema dell'utilizzo dei suddetti caratteri nel valore stesso della stringa. Ricordo comunque che è possibile forzare l'inserimento di ognuno dei due apici semplicemente raddopiandoli.

Oltre a questi semplici tipi di dati, python offre anche strutture dati complesse implementate in modo nativo ("**built-in types**" come si dice ufficialmente), questo offre al programmatore strumenti potenti senza cercare librerie da terze parti.
In particolare mi riferisco a liste, dizionari, tuple e files; tutte strutture di cui vi parlero' nei prossimi capitoli.

Numerosi sono anche gli operatori che riassumo brevemente nelle tabelle seguenti:

Operatori sul tipo intero:

Operatore	Descrizione	Esempi
+, -	somma e sottrazione di interi	10+12, 5-1
*, /	moltiplicazione e divisione	10*12, 10/2
%	resto della divisione	10%3=1, 5.3%2.5=0.3
<<, >>	shift bit a bit a sinistra e destra	24<<1=48, 10>>1=5

Operatori booleani:

Operatore	Descrizione	Esempi
or, and	or e and logici	x or y, z and k
not	negazione logica	(not 0)=1
<, <=, >, >=, ==, <>, !=	operatori di confronto	(10==10)=1, ('a'!='a')=0
¦	or bit a bit	x ¦ y
&	and bit a bit	x & y
^	or esclusivo bit a bit	x ^ y

Operatori per le stringhe:

Operatore	Descrizione	
+	concatenamento	('a'+'b')='ab'
*	ripetizione	('a'*3)='aaa'

s[i]	indicizzazione dei caratteri	s='abc' s[0]='a'
s[i:j]	slicing	s='abc' s[1:2]='b'
len(s)	lunghezza	s='abc' len(s)=3
%	formattazione di stringhe	('ciao %s' % 'stefano')='ciao stefano'

Nell'ultima tabella ho introdotto qualche concetto che merita un chiarimento.

Le stringhe in python sono delle liste di caratteri (più precisamente si tratta del tipo "lista" di cui parlero' nei prossimi capitoli). Per questo è possibile prelevare i caratteri attraverso il loro indice, tenendo presente che si parte sempre da 0.

```
>>> s = 'http://www.html.it'
>>> print s[0]
'h'
>>> print s[4]
':'
>>> print s[17]
't'
```

Ancora più interessante è la possibilità di prelevare sottostringhe utilizzando la tecnica di "**slicing**" (affettare). Tale tecnica permette di tagliare una parte della lista indicando l'indice di partenza e l'indice finale.
È necessario specificare che verra' prelevata la stringa partendo dal carattere indicato dall'indice iniziale incluso, fino all'indice finale escluso.
Inoltre omettendo uno dei due indici è possibile indicare a python di andare fino in fondo alla stringa.
Infine, è possibile indicare nell' indice finale un numero negativo, in questo caso python conterà i caratteri a ritroso.
Vediamo qualche esempio:

```
>>> s = 'http://www.html.it'
>>> print s[7:10]
'www'
>>> print s[7:]
'www.html.it'
>>> print s[:4]
'http'
>>> print s[11:-3]
'html'
```

L'operatore % permette di sostituire in una stringa dei parametri prefissati con il valore contenuto dentro a delle variabili, esattamente come avviene nella istruzione **printf** del C. Facciamo subito qualche esempio complesso per capire il funzionamento:

```
>>> nome = 'stefano'
>>> eta = 29
>>> risultato = "%s ha %d anni" % (nome, eta)
>>> print risultato
'stefano ha 29 anni'
```

Nelle prime due righe ho creato una variabile stringa e una numerica.
Nella stringa "%s ha %d anni" ho inserito due parametri:

- %s : parametro di tipo stringa
- %d : parametro di tipo numerico

Ora voglio sostituire questi due parametri con i valori delle variabili "nome" ed "eta".
Questo si può realizzare con una sola istruzione mediante l'operatore %.
Fate attenzione alle parentesi che racchiudono nome ed eta, esse sono necessarie poiché l'operatore % utilizza una tupla (struttura dati che vedremo in seguito), quindi io ho creato una tupla composta dal nome e l'eta'.

Esistono anche altri tipi di parametri, i principali dei quali sono riassunti in questa tabella (per chi conosce il C rimando al comando printf):

Operatori per le stringhe:

Parametro	Descrizione
%s	stringa
%c	singolo carattere
%d	numero decimale
%u	intero senza segno
%o	numero in notazione ottale
%x	numero in notazione esadecimale
%g	numero reale in notazione normale
%e	numero reale in notazione scientifica

Python offre altre funzioni per gestire le stringhe, esse sono tutte raggruppate nel modulo denominato " **string** ".
Non avendo ancora parlato di moduli, introdurro' queste funzioni in seguito.

Liste

Una lista rappresenta una collezione ordinata di oggetti.
Esattamente come avviene per gli array dei linguaggi di programmazione tradizionali è possibile inserire in sequenza gli oggetti ed accedere ad essi mediante un indice.
Essendo python poco tipizzato, esso lascia la possibilità di inserire oggetti eterogenei nella stessa lista.
Questo può essere un aspetto interessante che può risultare utile quando si sviluppano applicazione RAD (Rapid Application Development), dove non si ha il tempo di creare strutture complesse per contenere dati di natura diversa.

Di seguito elenco una serie di esempi che illustrano le caratteristiche delle liste e gli operatori ad esse associate:

Tutte le liste iniziano con l'elemento numero 0:

```
>>> lista1 = ['a','b','c',1] # si noti la lista eterogenea
>>> lista1[0]
'a'
```

È possibile conoscere la lunghezza di una lista con la funzione len(x):

```
>>> lista1 = ['a','b','c',1]
>>> len(lista1)
4
```

Esiste un elemento particolare che rappresenta la lista vuota ([]):

```
>>> lista2 = []
>>> len(lista2)
0
```

È possibile selezionare un sottoinsieme di elementi adiacenti nella lista mediante i potenti operatori di "**slicing**":

```
>>> lista1 = ['a','b','c',1]
>>> lista1[2:3] # il risultato è una lista con un solo elemento
['c']
```

È possibile concatenare due liste (operatore "**+**"), in questo caso gli elementi della seconda lista vengono accodati a quelli della prima lista:

```
>>> lista1 = ['a','b','c']
>>> lista2 = ['d','è,'f']
>>> lista_tot = lista1 + lista2
>>> lista_tot
['a','b','c','d','è,'f']
```

Esiste un operatore assai originale che permette di concatenare in modo ripetitivo una lista (operatore "*"):

```
>>> lista1 = [1,2]
>>> lista_tot = lista1 * 3
>>> lista_tot
[1,2,1,2,1,2]
```

È possibile aggiungere un elemento in fondo alla lista, in questo caso si utilizza la funzione append() dell'oggetto lista:

```
>>> lista1 = [1,2]
>>> lista1.append(3)
>>> lista1
```

```
[1,2,3]
```

È possibile ordinare una lista in modo diretto o inverso, utilizzando rispettivamente le funzioni sort() e reverse():

```
>>> lista1 = ['c',a','q','d']
>>> lista1.sort()
>>> lista1
['a','c','d','q']
>>> lista1.reverse()
>>> lista1
['q','d','c','a']
```

Potete scoprire come si comporta python in caso di liste eterogenee semplicemente facendo qualche prova con l'interprete interattivo (buon divertimento!).

È possibile recuperare facilmente l'indice in cui è posizionato un elemento dato, basta utilizzare il metodo index:

```
>>> lista1 = ['a','c','d','q']
>>> lista1.index('d')
2
```

La funzione index può rivelarsi veramente utile per effettuare delle ricerche di un elemento nella lista.

Infine per cancellare un elemento:

```
>>> lista1 = ['a','c','d','q']
>>> del lista1[2]
>>> lista1
[a','c','q']
```

Dizionari

Un dizionario rappresenta una collezione "**non ordinata**" di oggetti.
Gli oggetti sono identificati univocamente da una **chiave** (generalmente una stringa) invece che mediante un indice numerico, come avviene nelle liste.
Ogni elemento del dizionario è rappresentato da una coppia (**chiave : valore**), la chiave serve per accedere all'elemento e recuperare il valore.
Esattamente ciò che avviene quando si cerca una parola sul vocabolario, in tal caso il valore che si cerca è una frase che spiega il significato della chiave.
In analogia alle liste, anche per i dizionari python lascia la possibilità di inserire oggetti eterogenei nella stesso dizionario.

Di seguito elenco una serie di esempi che illustrano le caratteristiche dei dizionari e gli operatori ad essi associati. Questa volta voglio utilizzare un esempio più reale, quindi costruisco un dizionario che associa ad ogni nome di persona il voto preso in un ipotetico esame universitario di python.

È possibile conoscere il numero di elementi di un dizionario mediante la funzione len(x):

```
>>> diz1 = {'stefano':23,'elena':19,'enrico':25,'simone':30}
# poteva andare meglio !

>>> len(diz1)
4
```

Esiste un elemento particolare che rappresenta il dizionario vuoto (**{}**):

```
>>> diz2 = {}
>>> len(diz2)
0
```

È possibile controllare l'esistenza di una chiave nel dizionario mediante la funzione has_key(x). Il risultato di tale funzione è booleano (1=vero, 0=falso).

```
>>> diz1.has_key('stefano')
1
>>> diz1.has_key('luca')
0
```

Come si evince dall'esempio, Stefano ha partecipato all'esame, mentre Luca no.

Esistono due metodi che permettono di estrarre dal dizionario la lista delle chiavi e la lista dei valori (rispettivamente keys() e values()):

```
>>> diz1.keys()
['stefano','elena','enrico','simone']
>>> diz1.values()
[23,19,25,30]
```

Il metodo keys può essere molto utile per elencare in ordine alfabetico i partecipanti all'esame:

```
>>> k = diz1.keys()
>>> k.sort()
>>> k
['elena','enrico','simone','stefano']
```

Inserire un nuovo elemento nel dizionario è molto semplice, si effettua un assegnamento:

```
>>> diz1['pierino'] = 18
>>> diz1
```

```
{'stefano':23,'elena':19,'enrico':25,'simone':30,'pierino':18}
```

Infine per cancellare il più secchione della classe:

```
>>> del diz1['simone']
>>> diz1
{'stefano':23,'elena':19,'enrico':25,'pierino':18}
```

Secondo me questa struttura rappresenta il migliore strumento di python. I dizionari sono semplici da utilizzare, efficienti e molto flessibili, soprattutto sono utili per chi gestisce dati in strutture relazionali. Infatti combinando opportunamente liste e dizionari è possibile creare un vero e proprio database relazionale.
Per fare un esempio si potrebbe creare il database dei voti universitari nel seguente modo:

```
>>>> db = []
>>>> db.append({'esame':'guida a python','nome':'stefano','voto':23})
>>>> db.append({'esame':'guida a python','nome':'elena','voto':19})
>>>> db.append({'esame':'progr. ad oggetti','nome':'stefano','voto':28})
>>>> db.append({'esame':'guida a python','nome':'enrico','voto':18})
```

Questa è una tabella di 4 record composti da 3 campi ognuno. I 3 campi sono stati chiamati rispettivamente esame, nome e voto. Ogni record è rappresentato da un elemento della lista "db", la quale rappresenta l'intero database.

Internamente i dizionari sono implementati mediante una **tabella hash**, quindi l'individuazione di un elemento in base alla chiave è estremamente veloce.

Dopo anni di lavoro con diversi linguaggi di programmazione, io ho maturato la convinzione di non poter fare a meno di una struttura dati come i dizionari. Io ho lavorato diversi anni con Microsoft Visual C++, con la libreria ad oggetti MFC (Microsoft Foundation class), e di conseguenza ho spesso utilizzato le mappe. Le mappe di MFC sono identiche ai dizionari di python, ma essendo implementate come libreria esterna al linguaggio sono incredibilmente lente e macchinose.
In questo caso, essendo i dizionari gia' integrati nell'interprete, Python riesce a fornire uno strumento leggero e potentissimo.

Tuple
Una tupla è simile ad una lista con una sottile differenza:

- La lista è un tipo mutabile
- La tupla è un tipo non mutabile

Cerchiamo di chiarire questo concetto di python.
Le variabili di tipo mutabile possono cambiare di stato durante la loro vita, infatti per una lista è possibile aggiungere o togliere elementi in qualsiasi momento.
Per i tipi non mutabili cio' non è possibile, è possibile solamente cambiare in blocco l'intero valore.
Le tuple sono utilizzate quando si deve essere certi che nessuno possa modificare il contenuto dell'elenco, e quindi non si possa aggiungere o togliere elementi.
Per il resto le tuple hanno il medesimo funzionamento delle liste. C'è solo una piccola differenza sintattica : sono racchiuse tra parentesi tonde (e non quadre come le liste). Gli operatori sono gli stessi delle liste (a parte quelli che mutano il valore, che chiaramente non hanno motivo di esistere):

Vediamo solamente qualche piccolo esempio:

```
>>> t1 = (1, 'a')
>>> t2 = (2, 'b')
>>> print t1[0]
1
>>> print t1*2
(1, 'a', 1, 'a')
>>> len(t1)
2
>>> t3 = t1 + t2
>>> print t3[1:3]
('a', 2)
```

Istruzioni di base

In questo capitolo intendo elencare i vari tipi di istruzioni presenti in python. Esse sono suddivise in categorie ben precise.
Particolare attenzione verra' posta per i costrutti che permettono di controllare il flusso di controllo delle istruzioni.
Riporto di seguito un elenco dei tipi di istruzioni presenti nel linguaggio:

I commenti

È sempre buona norma commentare il codice, e quindi anche python offre una sintassi per i commenti.
Il carattere che identifica il commento è : "**#**". esso si può usare all'inizio della riga oppure alla fine di una istruzione.

Istruzioni di assegnamento

In python è possibile assegnare un valore ad una variabile mediante l'operatore "**=**".
Ricordiamo che le variabili in python NON devono essere dichiarate, ma esse nascono durante la prima istruzione di assegnamento (esattamente come avveniva nel vecchio Basic). Nel momento del primo assegnamento viene anche deciso il tipo della variabile.
Una singolare possibilità offerta da python è rappresentata da l'assegnamento multiplo nel quale si possono inizializzare più variabili direttamente sulla stessa riga di codice. Per capire quest'ultimo concetto basta osservare l'esempio sottostante:

```
>>> a = 'viva python'
>>> b = 2
>>> c, d = 'su', 'html.it' # assegnamento multiplo
>>> print a,b,c,d
viva python 2 su html.it
```

Ricordiamo che le regole da seguire nella scelta dei nomi delle variabili è simile a quella dei più comuni linguaggi di programmazione, in particolare:

- Ogni variabile deve iniziare con una lettera oppure con il carattere underscore "_", dopodiche' possono seguire lettere e numeri o il solito underscore.
- Python è un linguaggio case sensitive, quindi distingue le variabili composte da caratteri minuscoli da quelle scritte con caratteri maiuscoli.
- Esistono delle parole riservate che non possono essere utilizzate per i nomi delle variabili. Esse sono le seguenti:
    - and, assert, break, class, continue, def, del, elif, else, except,
    - exec, finally, for, from, global, if, import, in, is, lambda,
    - not, or, pass, print, raise, return, try, while

Istruzione di condizionamento

Il famoso costrutto "**if else**" che permette di eseguire gruppi di istruzioni diverse in base ad una prefissata condizione è molto semplice in python:

```
if <test1>:
        <gruppo di istruzioni 1>
elif test2:                                      # opzionale
        <gruppo di istruzioni 2>
else:                                            # opzionale
        <gruppo di istruzioni 3>
```

Ricordiamo che i gruppi di istruzione devono essere indentati nello stesso modo in quanto è solo l'indentazione che permette di strutturare l'istruzione if else.
A differenza del C, o del Pascal, non esistono parole che delimitano l'inizio e la fine del blocco (le parentesi

graffe del C o begin e end del pascal).
Da notare che i gruppi "elif" e "else" sono opzionali. Vediamo un piccolo esempio:

```
>>> a = 5
>>> if a == 5:
...     print 'ok'
... elif a > 5:
...     print 'troppo'
... else:
...     print 'poco'
ok
```

Istruzioni di ciclo

Per effettuare dei cicli è possibile utilizzare due costrutti:

- Il ciclo **for** che permette di iterare su una lista di elementi.
- Il ciclo **while** che permette di fare un ciclo di tipo generale ponendo la condizione di uscita come prima istruzione del ciclo.

Vediamo la sintassi del costrutto for:

```
for <contatore del ciclo> in <lista>:
        <gruppo di istruzioni 1>
```

Il contatore del ciclo è una variabile alla quale viene assegnato, ad ogni passo del ciclo, un valore della lista.
In questo modo all'interno del gruppo di istruzioni 1 è possibile operare sul singolo elemento della lista.
Vediamo un esempio:

```
>>> lista1 = ['a','b','c']
>>> for i in lista1:
...     print i+','
a,
b,
c,
```

Il costrutto for sembra molto limitato perché, a differenza di altri linguaggi di programmazione, permette di operare solamente sulle liste.
In realtà questa piccola limitazione viene facilmente superata utilizzando la funzione "**range**".
La funzione range permette di costruire una lista di numeri partendo da 0 fino ad un valore scelto, in questo modo è sufficiente creare la lista da utilizzare nel costrutto for con questa istruzione per ottenere il medesimo risultato di altri linguaggi di programmazione. Vediamo un esempio che spiega bene questo trucco:

```
>>> lista2 = range(4)
>>> for i in lista2:
...     print 'elemento' + str(i)
elemento 0
elemento 1
elemento 2
elemento 3
```

Notare che la lista ottenuta con range è una lista di numeri, e non di stringhe, questo è il motivo dell'utilizzo della funzione "str" per convertire il numero in stringa da stampare a video.
Se non si vuole cominciare da 0 basta indicare alla funzione range anche il valore iniziale.

```
>>> range(2,5)
[2,3,4]
```

Vediamo ora la sintassi del costrutto while:

```
while <test>:
        <gruppo di istruzioni 1>
```

Questo costrutto è il più generale ciclo che esiste in python, infatti in questo caso non siamo legati ad alcuna variabile da assegnare in modo ciclico.
Siamo liberi di utilizzare una condizione qualsiasi. In questo caso si continua ad iterare nel ciclo fino a quando la condizione rimane vera, appena essa diventa falsa si esce dal ciclo e si eseguono le istruzioni successive.
Vediamo un esempio:

```
>>> a=0
>>> b=10
>>> while a<b:
...     print a,
...     a = a + 1
0 1 2 3 4 5 6 7 8 9
```

Istruzioni di interruzione cicli

Quando si esegue un ciclo qualsiasi è sempre possibile forzare l'uscita dal ciclo in ogni momento, questo è possibile mediante degli appositi comandi, che elenco di seguito:

- **break** - Questo comando permette di saltare fuori da un ciclo ignorando le restanti istruzioni da eseguire.
- **continue** - Questo comando permette di saltare alla prima istruzione della prossima iterazione del ciclo.
- **else** - Sia il ciclo for che il ciclo while hanno un costrutto aggiuntivo opzionale che permette di eseguire un blocco di istruzioni al verificarsi di una uscita forzata dal ciclo.

In conseguenza di quello appena detto la sintassi dei due costrutti di ciclo si estendono nel seguente modo:

```
for <contatore-del-ciclo> in <lista>:
        <gruppo di istruzioni 1>
else:
        <gruppo di istruzioni 2>       # in seguito a un break

while <test>:
        <gruppo di istruzioni 1>
else:
        <gruppo di istruzioni 2>       # in seguito a un break
```

```
>>> a = 0;
>>> b = 10;
>>> while a<b:
        print a,
        a = a + 1
        if a == 5:
                break;
else:
        print 'a'
0 1 2 3 4
```

Nell'esempio sopra il ciclo si è interrotto quando la variabile a è diventata 5.

Lavorare con i files

È interessante notare come la gestione dei files in python è stata implementata internamente al linguaggio (built-in), questo permette una notevole velocita' e semplicita' nella loro gestione.
Penso che sia noto a tutti il concetto di file. In particolare, nel caso di un linguaggio di programmazione, è importante poter gestire i file per salvare delle informazioni sul disco e renderle di conseguenza persistenti.
Python permette di svolgere le comuni operazioni sui files: aprire e chiudere un file, leggere e scrivere in sequenza byte sul medesimo file ecc... . Tutto questo sia per files di testo che files binari.
Di seguito viene proposta una tabella con le principali funzioni built-in per la gestione dei files:

Operazione	Descrizione
output = open('pippo.txt','w')	apertura di un file in scrittura
input = open('dati','r')	apertura di un file in lettura
s = input.read()	lettura dell'intero contenuto del file
s = input.read(N)	lettura di N bytes
s = input.readline()	lettura di una riga (per files di testo)
s = input.readlines()	restituisce l'intero file come lista di righe (per files di testo)
output.write(s)	scrivo un intero file
output.writelines(L)	scrive la lista L in righe nel file
output.close(L)	chiusura del file

Vediamo un esempio.

```
>>> miofile = open('pippo.txt','w')
>>> miofile.write('ciao html.it\n')
>>> miofile.close()

>>> miofile = open('pippo.txt','r')
>>> miofile.readline()
'ciao html.it\012'
>>> miofile.readline()
''
```

Le prime tre istruzioni permettono di creare un nuovo file denominato "pippo.txt" contenente una stringa.
Successivamente viene aperto lo stesso file e viene letto riga per riga. L'ultima riga restituisce una stringa vuota, questa condizione permette di riconoscere la fine del file.
Se dovessero esserci delle righe vuote nel file, esse non vengono lette da python come righe vuote, ma righe contenenti un carattere di "A CAPO".

Combinando le funzionalita' dei files con le funzioni che python offre per gestire le stringhe si possono creare semplici programmi estremamente utili.
Fino ad ora abbiamo fatto piccoli esempi senza alcuna utilita' pratica, adesso per mostrare come operare sui files di testo facciamo un esempio concreto.

Quando mi è stato commissionato questo manuale sono state fissate delle regole di stesura del documento, alcune regole erano le seguenti:

- La E maiuscola va accentata e non apostrofata, in particolare è necessario usare il carattere **È** (codifica iso8859 : È).
- Le parole Online e Offline vanno scritte come unica parola senza trattini o spazi.

Siccome io sono pigro e distratto e sono abituato ad apostrofare tutto (in pratica non uso caratteri accentati poiché lavoro con sistemi operativi diversi), allora ho pensato di scrivere tutto e poi dare in pasto il mio lavoro ad un programma che corregga accuratamente i miei errori.
Si sa, la pigrizia è la benzina dei programmatori !

Comunque ho scritto il seguente programma con blocco notes di windows:

```
import sys
import string

f_input = open(sys.argv[1],'r')
f_output = open('new'+sys.argv[1],'w')

while 1:
        riga = f_input.readline()
        if riga=='':
                break;
        riga2 = string.replace(riga,"E'",'&Egrave;')
        riga2 = string.replace(riga2,"On-line",'Online')
        riga2 = string.replace(riga2,"On line",'Online')
        riga2 = string.replace(riga2,"Off-line",'Offline')
        riga2 = string.replace(riga2,"Off line",'Offline')
        f_output.write(riga2)

f_input.close()
f_output.close()
```

Le prime due righe permettono di caricare delle librerie standard di python. Tali librerie si chiamano MODULI, e ne parleremo in modo approfondito in seguito. Successivamente vengono aperti due files:

- **f_input** è un file aperto in lettura. Il nome di tale file viene prelevato dal parametro che viene scritto dopo il nome del parametro. Infatti analogamente al linguaggio C, gli argomenti che vengono passati sulla riga di comando quando si richiama un programma vengono consegnati al programmatore in una lista predefinita di nome "sys.argv". Ricordo che argv[0] contiene il nome del programma stesso, di conseguenza argv[1] è il primo parametro.
- **f_output** è il file di output dove intendo creare la versione corretta del mio documento. In questo caso come secondo parametro passo la lettera "w" poiché intendo creare il file in scrittura.

Ho creato un ciclo un po' strano, infatti la condizione del while è sempre vera. Potrebbe sembrare che non si esca mai da tale ciclo, invece l'uscita avviene attraverso l'istruzione **break** quando si raggiunge la fine del file di input. Ad ogni passo del ciclo leggo una intera riga del file di input, su questa riga applico 5 sostituzioni.
La funzione **replace** del modulo **string** sostituisce interi pezzi di stringa con altre stringhe. Ad esempio nella prima replace sostituisco tutte le E apostrofate con il corretto codice speciale html.

In questo modo correggo tutte le mie dimenticanze e spero che abbia funzionato tutto !.

Funzioni

Importanza delle funzioni

Le funzioni sono raggruppamenti di istruzioni che prendono in ingresso un insieme di valori detti parametri e restituiscono un risultato come elaborazione dei parametri stessi.
Le funzioni sono uno strumento importante, poiché:

- permettono il riutilizzo del codice, infatti funzioni usate molte volte in un programma possono essere inglobate in un unica funzione.
- permettono di strutturare il codice del programma in blocchi omogenei dal punto di vista logico al fine di migliorare il lavoro del programmatore.

Come scrivere funzioni

La sintassi per la definizione di una funzione è la seguente:

```
def nome_funzione(<lista parametri divisi da virgola>):
        <blocco di istruzioni>
        return <risultato>     # opzionale
```

Una volta definita una funzione in questo modo è possibile richiamarla semplicemente invocando il suo nome, seguito dalla lista di valori che si intende passare come parametri.
Facciamo un esempio creando una funzione che esegue il quadrato di un numero.

```
def quadrato(valore):
        ris = valore * valore
        return ris

a = 5
print quadrato(a)
```

Eseguendo questo programma il risultato sara' 25. Le prime tre righe definiscono la funzione quadrato. Da questo momento in poi il nome "quadrato" diventa parte del namespace del modulo corrente (l'insieme dei nomi significativi all'interno del modulo).
Richiamando la funzione, il valore passato come parametro viene trasferito alla funzione attraverso il nome del parametro ("valore"), viene eseguita la funzione e viene restituito il risultato al modulo chiamante.
Tutto questo permette di incapsulare nel nome "quadrato" una operazione più complessa.

Variabili locali, variabili globali e passaggio di parametri

Introducendo le funzioni si devono distinguere le "variabili locali alle funzioni" (quindi utilizzabili solo da esse) e le "variabili globali", ossia appartenenti al namespace del modulo (quindi utilizzabili al di fuori della funzione).
Quando si richiama una funzione vengono trasferiti i valori delle variabili ai parametri delle funzioni. Questo permette alla funzione di venire a conoscenza di informazioni proveniente dal blocco chiamante.

In genere nei linguaggi di programmazione esistono due modalita' di passaggio:

- **Passaggio per valore**: Viene trasferito alla funzione solo una copia della variabile, quindi la funzione non può alterare il valore di quella variabile.
- **Passaggio per riferimento**: In questo caso viene trasferita la variabile vera e propria e quindi la funzione può alterare il valore della variabile.

In python i parametri passati alle funzioni sono sempre trasferiti per valore. Questo a differenza di altri linguaggi, come il C e il Pascal, dove si possono passare i parametri anche per riferimento.
Infatti quando si utilizza una variabile, python cerca prima il nome di quella variabile nel namespace locale. Se la ricerca non da esito positivo, si prosegue con il namespace globale e solo successivamente si va a cercare il nome tra le funzioni builtin (cioe' quelle predefinite in python stesso).
Questo meccanismo permette di utilizzare il valore delle variabili globali, ma di non poterle mai modificare, in

quanto un nuovo assegnamento alla variabile provoca la creazione dello stesso nome in un namespace nuovo.
Chiariamo questo curioso concetto con un esempio:

```
def funzione1():
    x = 20      # variabile locale con lo stesso nome
    print x     # verra' visualizzato 20 e non 10
    print y     # verra' visualizzato 19

x = 10          # variabili globale
y = 19
funzione1()
```

Il discorso cambia radicalmente per le strutture dati mutabili, le quali possono essere modificate all'interno delle funzioni.
Esse infatti non sono delle vere e proprie variabili e quindi passibili di modifiche.

Parametri facoltativi

È possibile introdurre dei parametri che hanno la caratteristica di essere facoltativi, infatti essi assumono un valore prestabilito se non vengono indicati.
Vediamo un esempio per spiegare la semplice sintassi da utilizzare:

```
def funzione2(a,b = 30):
    print a, b

x = 10
y = 19
funzione2(x)
funzione2(x,y)
```

La funzione denominata "funzione2" ha due parametri: a e b. Essa viene invocata due volte:

- Il risultato della prima chiamata sara' 10, 30, infatti il parametro **b**, non essendo indicato, assume il valore 30.
- Il risultato della seconda chiamata sara' 10, 19.

Procedure

Nel linguaggio pascal si distinguono le funzioni dalle procedure. Le procedure sono delle funzioni che non restituiscono alcun risultato.
In python si possono implementare facilmente delle procedure, è sufficiente non introdurre l'istruzione **return** alla fine della funzione. In questo modo la funzione non restituisce alcun valore al blocco chiamante.

Moduli

Perche' esistono i moduli e cosa sono

Quando si inizia a diventare programmatori esperti, i programmi crescono a vista d'occhio e diventa estremamente difficile gestire l'intero programma in un unico file di script.
Per questo è opportuno suddividere il programma in diversi files di script. Questo metodologia permette di creare delle raccolte di funzioni che possono essere utilizzate in progetti diversi.
I moduli quindi sono dei files di script (files con estensione "PY") che possono essere richiamati da altri programmi python per riutilizzare le funzioni contenute in essi.
I moduli possono essere creati facilmente dal programmatore, ma ne esistono moltissimi gia' precostruiti e contenuti dentro alla installazione di python. Essi in genere raccolgono funzioni utili per risolvere svariate problematiche, per esempio:

- moduli per la gestione delle stringhe,
- moduli per le chiamate alle funzioni del sistema operativo,
- moduli per la gestione di internet,
- moduli per la posta elettronica ecc...

Inoltre, una possibilità da non sottovalutare, è rappresentata dall'utilizzo di moduli scritti da terzi fornitori.
Navigando in internet è possibile trovare moduli di python che risolvono ogni sorta di problema; molti di essi, in pieno stile open source, sono di utilizzo gratuito e coperti da licenza GPL.

Come utilizzare i moduli

Per scrivere un modulo, basta semplicemente creare un file con estensione PY e riempirlo con tutte le funzioni necessarie al modulo stesso.
Per utilizzare il modulo appena creato è sufficiente importarlo con il comando "**import**". Ad esempio, supponendo di aver creato il modulo "libreria.py", basta scrivere in cima al programma:

```
import libreria
```

Dopo aver importato il modulo possiamo semplicemente richiamare le funzioni contenute in esso utilizzando la **dot notation**.
Digitando il nome del modulo, un punto e il nome della funzione riusciamo ad entrare nel modulo e richiamare la funzione desiderata.
Ad esempio, supponendo che "libreria.py" contenga le funzioni apri(), chiudi() e sposta(oggetto), posso richiamare le funzioni nel seguente modo:

```
import libreria

libreria.apri()
ogg=5
libreria.sposta(ogg)
libreria.chiudi()
```

Se non si desidera utilizzare il nome della libreria tutte le volte che si richiama una funzione, è possibile importare anche i nomi delle funzioni direttamente.
Ad esempio, per importare la funzione sposta(oggetto):

```
import libreria
from libreria import sposta

libreria.apri()
ogg=5
sposta(ogg)
libreria.chiudi()
```

In questo caso la funzione sposta viene a far parte dell'insieme dei nomi definiti nel programma (namespace), e non necessita' più del prefisso del nome del modulo (libreria).

Se si desidera importare tutti i nomi delle funzioni di "libreria.py" all'interno del programma basta utilizzare il carattere asterisco:

```
from libreria import *

apri()
ogg=5
sposta(ogg)
chiudi()
```

Bisogna considerare un altro vantaggio offerto dai moduli: quando si interpreta un programma python, esso crea per ogni modulo una versione "semicompilata" in un linguaggio intermedio più veloce (bytecode).
Dopo una esecuzione, è possibile notare la presenza di tanti files con estensione **PYC** (Python compiled).
Ogni file ha lo stesso nome del sorgente PY, ma con estensione PYC.
In questo modo python non deve interpretare il codice tutte le volte, ma solamente quando viene effettuata una modifica. L'interprete confronta ogni volta la data e l'ora del file PY con il file PYC, se sono diverse interpreta il codice, altrimenti esegue direttamente il bytecode.
Questo meccanismo, peraltro usato ormai da molti linguaggi, permette di aumentare notevolmente la velocita' del codice.

Moduli predefiniti

Python è dotato di una raccolta di moduli standard molto vasta. Per utilizzare tali moduli è necessario solamente importarli come abbiamo visto in precedenza.
Vediamo alcuni moduli standard di notevole importanza:

- **string**: modulo che raccoglie funzioni per gestire le stringhe, ad esempio:
  - funzioni per la conversione da stringa a numero
  - funzioni per la conversione da minuscolo a maiuscolo
  - funzioni per la sostituzioni di pezzi di stringhe
- **sys**: modulo che raccoglie funzioni per reperire informazioni di sistema, ad esempio reperire gli argomenti passati sulla linea di comando.
- **os**: modulo per eseguire operazioni del sistema operativo sottostante, ad esempio copiare, rinominare o cancellare un file.

Nella documentazione inclusa con python si trova l'elenco completo di tutti i moduli predefiniti.

I Namespaces

Per capire meglio il concetto della importazione dei nomi delle funzione bisogna parlare del concetto di namespace (termine gia' introdotto in precedenza).
Ogni modulo ha un insieme di nomi (nomi di variabili e nomi di funzioni) che rappresentano i nomi utilizzabili all'interno di esso. Questo insieme viene detto namespace (spazio dei nomi).
Per trasportare i nomi da un namespace ad un altro si può usare il comando import che abbiamo visto.
Mediante la funzione **dir()** è possibile vedere l'elenco dei nomi presenti nel namespace corrente.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
```

Questi nomi, mostrati sopra, sono nomi definiti dall'interprete prima di cominciare. Se io importo il modulo libreria ottengo il seguente risultato:

```
>>> from libreria import *
>>> dir()
['__builtins__', '__doc__', '__name__', 'apri', 'chiudi', 'sposta']
```

Come si intuisce, adesso tutte le funzioni contenute in libreria.py fanno parte del namespace.

Il concetto di namespace si applica anche alle funzioni, quando si entra in una funzione viene creato un

nuovo namespace, ecco perché non sono più visibili le variabili precedenti, ma solo le variabili locali alla funzione stessa.

Gestione delle eccezioni

Il momento peggiore per un programmatore è rappresentato dal blocco del programma a causa di un errore. Esistono due tipi di errori:

- **Errori sintattici**: in questo caso è errato il nome di un comando o di una funzione.
- **Errori di runtime**: in questo caso i nomi sono corretti, ma c'è un errore dovuto al valore assunto da determinate variabili. Esempi di errori di runtime sono i seguenti:
  - Divisione per zero
  - lettura di file inesistenti

Quando si ha un errore, il programma si ferma e viene evidenziato un messaggio di errore. Questo evento provoca una brusca interruzione del flusso di controllo del programma, quindi non è più possibile mantenere in vita il programma. Questo risulta vero per gli errori sintattici, per i quali l'interprete python non sa cosa deve eseguire, ma non è completamente vero per gli errori runtime. Nel seguente esempio si ha un errore di runtime, in questo caso si dice che è avvenuta una ECCEZIONE, ossia un evento accidentale (in realtà in questo esempio un po' forzato ...).

```
>>> a, b = 5, 0
>>> print a / b
Traceback (most recent call last):
  File "", line 1, in ?
    print a / b
ZeroDivisionError: integer division or modulo by zero
```

È possibile controllare questi eventi accidentali senza terminare il programma, per fare questo si utilizza un costrutto che permette di rilevare le eccezioni e passare il controllo di flusso ad un altro blocco di istruzioni. Tale costrutto è il seguente:

```
try:
        <gruppo di istruzioni sotto test>
except:
        <gruppo di istruzioni da eseguire in caso di errore>
else:                           # opzionale
        <gruppo di istruzioni2>
finally:                        # opzionale, al posto di except
        <gruppo di istruzioni3>
```

Tutte le istruzioni incluse nel blocco try sono tenute sotto controllo durante l'esecuzione del programma. Se tutto va bene il blocco except viene ignorato. In caso contrario, se avviene una eccezione, si salta al blocco di istruzione che seguono la parola except.
Se incapsuliamo ogni operazione a rischio di eccezioni in un blocco try siamo in grado di reagire ad ogni errore runtime senza interrompere il programma.
Tornando all'esempio procedente si può operare in questo modo:

```
>>> a, b = 5, 0
>>> try:
...     print a / b
... except:
...     print 'non hai studiato matematica !'

non hai studiato matematica !
```

È possibile anche gestire tipologie diverse di eccezioni contemporaneamente, ed eseguire blocchi di istruzioni diverse a seconda del tipo di errore.
Per permettere questo è necessario fare seguire alla parola except il nome della classe di errore. Tale nome è rilevabile ad esempio dal messaggio python in caso di eccezione.
Riportando il solito esempio, per gestire in modo più puntuale l'eccezione avremo potuto scrivere nel seguente modo:

```
>>> a, b = 5, 0
>>> try:
...  print a / b
... except ZeroDivisionError:
...   print 'non hai studiato matematica, ma con python sei forte !'

non hai studiato matematica, ma con python sei forte !
```

Nel caso si verificasse una eccezione non legata alla divisione per zero si verificherebbe ugualmente un blocco del programma, anche in presenza del blocco try.

Se si desidera eseguire un particolare gruppo di istruzioni solo nel caso in cui non avvenga nessuna eccezione, basta inserire la parola chiave else, seguito dal blocco di istruzioni desiderate.
Se si desidera far eseguire ugualmente il blocco except anche in caso di nessuna eccezione, è sufficiente sostituire la parola chiave except con la parola chiave finally.

Una nota finale per chi volesse approfondire l'aspetto di gestione delle eccezioni di python. È possibile creare delle classi di eccezioni personalizzate, le quali possono essere generate in un punto qualsiasi utilizzando il comando raise. Ritengo superfluo, ai fini di questo manuale, introdurre ulteriori spiegazioni per questa possibilità di python estremamente avanzata e articolata.

Classi e cenni di programmazione ad oggetti

Nei capitoli introduttivi abbiamo detto che Python è un linguaggio orientato agli oggetti. In realtà esso permette sia la programmazione tradizionale (procedurale) che il nuovo paradigma ad oggetti. Quindi python si inquadra nei linguaggi **ibridi**, come il C++.
In questo capitolo cercheremo di introdurre teoricamente i concetti della programmazione orientata agli oggetti, accompagnando ogni concetto con qualche piccolo esempio. Solo in seguito trasferiremo i concetti teorici in python, illustrando la sintassi necessaria all'utilizzo degli oggetti in python stesso.

Cenni di programmazione orientata agli oggetti

La programmazione tradizionale si è sempre basata sull'utilizzo di strutture dati (come le liste, le tuple ecc...) e su funzioni e procedure (tutti concetti gia' visti in precedenza).
Questo metodo di sviluppo del software viene detto funzionale (o procedurale), con esso si organizza un intero programma in moduli che raccolgono gruppi di funzioni. Ogni funzione accede ad uno o più gruppi di dati.

I metodi di sviluppo funzionale hanno pero' notevoli debolezze:

- A causa dei stretti legami tra le funzioni e i dati, si arriva ad un punto in cui ogni modifica software provoca degli effetti collaterali su altri moduli con enormi difficolta di debug della applicazione.
- Difficolta' di riutilizzo del software. Ogni volta che si vuole riciclare una funzione bisogna apportare delle modifiche strutturali per adeguarla alla nuova applicazione.

La programmazione orientata agli oggetti è un modo alternativo di scomposizione di un progetto software: in essa l'unità elementare di scomposizione non è più l'operazione (la procedura) ma **l'oggetto**, inteso come modello di un'entità reale (un oggetto del mondo reale).
Questo approccio porta ad un modo nuovo di concepire un programma: il software è ora costituito da un insieme di entità (gli oggetti) interagenti, ciascuna provvista di una struttura dati e dell'insieme di operazioni che l'oggetto è in grado di effettuare su quella struttura. Poiché ciascun oggetto incapsula i propri dati e ne difende l'accesso diretto da parte del mondo esterno, si è certi che cambiamenti del mondo esterno non influenzeranno l'oggetto o il suo comportamento. D'altra parte per utilizzare un oggetto basta conoscere che dati esso immagazzina e che operazioni esso fornisce per operare su questi dati, senza curarsi dell'effettiva realizzazione interna dell'oggetto stesso.

Questo nuovo modo di sviluppare programmi software è più vicino alla nostra realtà quotidiana. Pensiamo ad un oggetto del mondo reale, ad esempio una automobile.
Una automobile ha un insieme di caratteristiche: il colore , la cilindrata ecc... Inoltre essa dispone di operazioni da svolgere esclusivamente con essa, ad esempio:

- accensione.
- cambio della marcia.
- parcheggio.

È giusto correllare le sue caratteristiche e le suo operazioni in una sola entita' (un solo oggetto).

Inoltre la programmazione ad oggetti favorisce la programmazione di gruppo, poiché gli oggetti non possono dipendere dall'implementazione di altri oggetti, ma solo dalle loro operazioni, perciò un programmatore può sviluppare un oggetto senza preoccuparsi della struttura degli altri elementi che compongono il sistema.

Vediamo ora i principali concetti su cui si basa questo modo di programmare:

Classe

Una classe definisce un tipo di oggetto. Se intendo creare un nuovo oggetto devo indicare al programma le sue caratteristiche.
Per fare questo devo definire una classe di appartenza.
In particolare la classe definisce:

- Le variabili contenute nell'oggetto, che si chiamano **dati membri**

- Le funzioni contenute nell'oggetto, che si chiamano **metodi**. Queste funzioni permettono di svolgere operazioni solo sui dati membri dell'oggetto stesso. Ogni metodo agisce esclusivamente sui dati membri della classe di appartenza. Questo vincolo rappresenta la grande forza della programmazione ad oggetti, la quale costringe il programmatore ad organizzare il software per componenti riciclabili ben distinti.

Ad esempio, se volessi costruire un programma che conserva un archivio di persone, potrei creare la classe delle "**persone**".
Tale classe potrebbe avere i seguenti dati membri:

- nome della persona
- cognome della persona
- indirizzo
- telefono
- stato civile

Inoltre si potrebbero definire i seguenti metodi:

- cambia l'indirizzo della persona
- cambia il telefono della persona
- cambia lo stato civile della persona

Questi metodi agiscono esclusivamente sui dati membri della classe.

Oggetti

Una volta definita una classe, siamo in grado di creare tanti oggetti che riflettono le caratteristiche della classe stessa.
Infatti gli oggetti non sono altro che "**istanze**" della classe. Tornando al nostro esempio delle persone, possiamo creare i seguenti oggetti:

- Stefano Riccio
- Rossi Mario
- Elena Bianchi

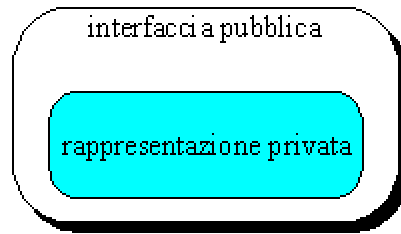
Queste sono tre istanze della classe persona. Ogni oggetto ha una copia dei dati membri dove conserva tutte le proprie informazioni.

Incapsulamento

La definizione di classe permette di realizzare l'incapsulamento. Esso consiste nella protezione dei dati membri dagli accessi non desiderati.
Questo avviene perché dall'esterno di un oggetto si può accedere ad un dato membro solo mediante un metodo e non direttamente usando il nome del dato membro stesso. Ad esempio, se volessi modificare l'indirizzo di una persona non posso semplicemente assegnare il nuovo indirizzo al nome della variabile (dato membro), come si farebbe nella programmazione tradizionale.
Bensi' sono obbligato ad utilizzare l'apposito metodo.

In realtà l'incapsulamento non è obbligatorio. Si può decidere, singolarmente su ogni dato membro, se renderlo protetto oppure no.

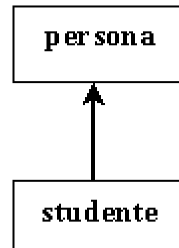
Generalmente gli oggetti hanno un insieme di metodi e dati che sono resi di dominio pubblico, mentre altri sono inaccessibili dall'esterno. Questo principio è conosciuto con il nome di "information hiding" (mascheramento dell'informazione) e fa si che un oggetto sia diviso in due parti ben distinte: una **interfaccia pubblica** e una **rappresentazione privata**.



Il mascheramento dell'informazione permette di rimuovere dal campo di visibilità esterno all'oggetto alcuni metodi o dati che sono stati incapsulati nell'oggetto stesso. L'incapsulamento e il mascheramento dell'informazione lavorano insieme per isolare una parte del programma o del sistema dalle altre parti, permettendo così al codice di essere modificato ed esteso senza il rischio di introdurre indesiderati effetti collaterali.
Questo metodo risulta molto utile per costruire librerie software da rilasciare a terze parti. Infatti, chi utilizza la libreria vede solamente l'interfaccia pubblica e ignora tutto il resto.

Ereditarieta'

Un altro concetto importantissimo della programmazione ad oggetti e l'ereditarieta'.
Con essa è possibile definire una classe sfruttando le caratteristiche di un'altra classe (che diventa la classe madre).
Ad esempio, se volessimo creare la classe degli studenti, dovremmo costruire una classe simile a quella delle persone, con l'aggiunta di qualche informazione in più.



Per evitare di scrivere tutto nuovamente, posso indicare al programma di **ereditare** tutte le caratteristiche dalla classe "persona" e aggiungere alcuni dati membri e alcuni metodi.
In questo modo posso affermare che gli studenti sono delle persone (frase filosofica !), quindi ereditano da essi le loro caratteristiche. Inoltre posso aggiungere ad esempio i seguenti dati membri specifici degli studenti:

- scuola di appartenza
- classe di appartenza

Inoltre si potrebbero definire i seguenti metodi specifici della classe studenti:

- cambia scuola
- promosso

L'ereditarieta' permette di utilizzare anche i metodi delle classi progenitrici, cosi per cambiare il nome di uno studente posso semplicemente utilizzare il metodo apposito della classe persona.

Un metodo, in una classe discendente, può nascondere la visibilità di un metodo di una classe antenata, semplicemente definendo il metodo con lo stesso nome. Lo stesso nome può così essere utilizzato in modo appropriato per oggetti che sono istanze di classi diverse. Ereditando dalle classi antenate i metodi che operano correttamente anche nelle classi discendenti, ed eliminando quei metodi che devono agire in modo differente, il programmatore estende realmente una classe antenata senza doverla completamente ricreare.

Ereditarieta' singola e multipla

La forma più comune di creazione di una classe è la **specializzazione**, mediante la quale viene creata una nuova classe poiché quella esistente è troppo generica. In questo caso è possibile utilizzare il meccanismo dell'ereditarieta' singola. Questo è il caso dell'esempio proposto delle persone e degli studenti.
Un altro tipico metodo di creazione è la **combinazione** con la quale la nuova classe è creata combinando gli oggetti di altre classi. In questo caso, se esiste, si utilizza il meccanismo dell'ereditarieta' multipla.
Per porre in relazione gli oggetti, sono quindi utilizzati due tipi di ereditarietà: quella singola e quella multipla.
Attraverso l'ereditarieta' singola, una sottoclasse può ereditare i dati membri ed i metodi da un'unica classe, mentre con l'ereditarieta' multipla una sottoclasse può ereditare da più di una classe.
L'ereditarietà singola procura i mezzi per estendere o per perfezionare le classi, mentre l'ereditarieta' multipla fornisce in più i mezzi per combinare o unire classi diverse.

Le opinioni riguardo la necessita' o meno del meccanismo dell'ereditarieta' multipla sono piuttosto controverse.
I contrari sostengono che si tratta di un meccanismo non strettamente necessario, piuttosto complesso e difficile da utilizzare correttamente. Quelli a favore dicono esattamente l'opposto ed infatti sostengono che l'ereditarieta' multipla è una caratteristica fondamentale di un linguaggio object oriented.
L'ereditarieta'à multipla incrementa sicuramente le capacita' espressive di un linguaggio, ma comporta un costo elevato in termini di complessita' della sintassi e di overhead di compilazione e di esecuzione.

Python permette l'utilizzo della ereditarieta' multipla, ma non intendo discuterne ulteriormente a causa della sua complessita' e del suo scarso utilizzo pratico.

Polimorfismo e overloading degli operatori

I linguaggi object-oriented permettono la creazione di gerarchie di oggetti (mediante l'ereditarieta') cui sono associati metodi che hanno lo stesso nome, per operazioni che sono concettualmente simili ma che sono implementate in modo diverso per ogni classe della gerarchia.
Di conseguenza, la stessa funzione richiamata su oggetti diversi, può causare effetti completamente differenti. Questa funzionalita' viene chiamata polimorfismo.

Un caso molto interessante di polimorfismo è rappresentato dall'overloading degli operatori. Con questa tecnica è possibile definire degli operatori matematici sugli oggetti che permettono di compiere operazioni sulle istanze in base al tipo di oggetto con il quale lavorano.
Approfondiremo questo aspetto in seguito.

Classi in python
La sintassi per le classi in python

Ecco di seguito la sintassi da utilizzare per definire una classe in python:

```
class <nome classe> [(<classe madre>,...)]:
        <elenco dati membro da inizializzare>
        <elenco metodi>
```

I dati membro si esprimono come le normali variabili di python. Se devo inizializzare dei dati membro faccio un semplice assegnamento, altrimenti posso definirli al momento dell'utilizzo, esattamente come avviene per le variabili normali.

Per i metodi basta utilizzare la medesima sintassi delle funzioni con alcuni accorgimenti. Ogni metodo deve avere come primo parametro l'oggetto stesso, infatti ogni volta che viene richiamato il metodo, python sistema nel primo parametro il riferimento all'oggetto stesso. Questo permette di accedere ai dati membri appartenenti allo stesso oggetto, normalmente si usa chiamare questo parametro "**self**".
Vediamo l'esempio delle persone scritto praticamente in codice python:

```
class persona:
        nome = ''
        cognome = ''
        indirizzo = ''
        telefono = ''
        stato_civile = ''

        def cambia_indirizzo(self,s):
                self.indirizzo = s
        def cambia_telefono(self,s):
                self.telefono = s
        def cambia_stato_civile(self,s):
                self.stato_civile = s
        def stampa(self):
                print self.nome,self.cognome,self.indirizzo,
                self.stato_civile

class studente (persona):      # ecco l'ereditarieta' !
        scuola = ''
        classe = 0

        def cambia_scuola(self,s):
                self.scuola = s
        def promosso(self):
                if self.classe == 5:
                        print 'scuola finita'
                else:
                        self.classe = self.classe + 1
```

Se ora voglio istanziare un oggetto, è sufficiente richiamare il nome della classe come se fosse una funzione.
Se voglio accedere ai dati membri e ai metodi basta utilizzare il punto.

```
p1 = persona()        # le parentesi sono obbligatorie
p1.nome = 'stefano'   # assegnazione diretta ai dati membri
p1.cognome = 'riccio'
p1.cambia_indirizzo('via html n. 10') # richiamo un metodo
p1.stampa()           # risultato : stefano riccio via html n. 10
```

p1 è un oggetto della classe persona che contiene quei valori.
Notate come abbiamo assegnato ai dati membri i valori direttamente oppure attraverso gli appositi metodi.
Ho utilizzato il metodo "cambia_indirizzo" passando un solo parametro (il metodo ne chiede due !), questo è

stato possibile perché python associa al parametro "self" l'oggetto p1. Utilizzando self posso accedere direttamente ai dati membro all'interno della classe.
Se ora voglio istanziare un oggetto di tipo studente, posso utilizzare i stessi dati membri e metodi della classe progenitrice:

```
s1 = studente()
s1.nome = 'mario'
s1.cognome = 'rossi'
s1.cambia_indirizzo('via html n. 10')
s1.cambia_scuola('liceo')
s1.stampa()     # risultato : mario rossi via html n. 10
```

Il risultato del metodo "stampa" non permette di visualizzare la scuola e la classe di appartenenza.
Questo avviene poiché viene richiamato il metodo della classe "persona" (classe madre), la quale non può conoscere il valore dei dati membri della classe "studente" (classe figlia).
Per risolvere questo problema ci viene in aiuto il polimorfismo, infatti è necessario **ridefinire** il metodo stampa costruendone una versione specifica per gli studenti.

Riscrivo la classe studente in questo modo:

```
class studente (persona):
        scuola = ''
        classe = 0

        def cambia_scuola(self,s):
                self.scuola = s
        def promosso(self):
                if self.classe == 5:
                        print 'scuola finita'
                else:
                        self.classe = self.classe + 1
        def stampa(self):
                print self.nome,self.cognome,self.indirizzo,
                self.stato_civile
                print 'scuola: '+self.scuola+' classe '
                +str(self.classe)
```

Il metodo stampa è polimorfico, sembra una brutta parola, ma questo significa che assume una forma diversa in base al tipo di oggetto sul quale viene applicato.

Inizializzazione di una classe

L'operazione di istanziazione di una classe provoca la creazione di un oggetto.
Spesso è necessario inizializzare i dati membri dell'oggetto nel momento della istanziazione di esso. Nel nostro esempio è utile inizializzare subito la persona con il suo nome e cognome (infatti il nome e cognome sono assegnati alla nascita e non cambiano più).
Per fare questo la teoria della programmazione orientata agli oggetti prevede l'uso di una funzione **costruttore**. In python esiste qualcosa di simile: è la funzione denominata "**__init__()**".

Quando all'interno di una classe viene dichiarata una funzione con questo nome, allora essa viene invocata automaticamente alla creazione dell'oggetto. Vediamo come modificare la classe "persona" per inizializzare automaticamente il nome e il cognome:

```
class persona:
        indirizzo = ''
        telefono = ''
        stato_civile = ''

        def __init__(self,n,c):
           self.nome = n
           self.cognome = c
```

```
        def cambia_indirizzo(self,s):
           self.indirizzo = s
        def cambia_telefono(self,s):
           self.telefono = s
        def cambia_stato_civile(self,s):
           self.stato_civile = s
        def stampa(self):
           print self.nome,self.cognome,self.indirizzo,
           self.stato_civile
```

Con la chiamata:

```
p1 = persona('stefano','riccio')
p1.stampa()     # risultato : stefano riccio
```

ottengo l'inizializzazione dei due dati membri. Da notare che ho eliminato l'assegnazione della stringa vuota alle due variabili (nome, cognome) nella classe. Esse infatti sono superflue poiché le variabili vengono create all'interno della funzione __init__.

L'incapsulamento in python

In precedenza abbiamo parlato di incapsulamento e subito dopo non abbiamo applicato questo principio. Infatti la prima operazione fatta sull'oggetto p1 è stata quella di inizializzare i dati membri "nome e cognome" direttamente.
Python assume che tutti i dati membri siano pubblici e quindi modificabili dall'esterno dell'oggetto. Per indicare a python di tenere protetti i dati membri, e quindi realizzare veramente l'incapsulamento, si deve anteporre al nome della variabile i caratteri "__" (2 underscore).
Cosi facendo nessuno può modificare il dato senza utilizzare l'apposita funzione. Ad esempio, nella nostra classe "persona" è utile proteggere tutti i dati membri:

```
class persona:
        __indirizzo = ''
        __telefono = ''
        __stato_civile = ''

        def __init__(self,n,c):
           self.__nome = n
           self.__cognome = c
        def cambia_indirizzo(self,s):
           self.__indirizzo = s
        def cambia_telefono(self,s):
           self.__telefono = s
        def cambia_stato_civile(self,s):
           self.__stato_civile = s
        def stampa(self):
           print self.__nome,self.__cognome,self.__indirizzo,
           self.__stato_civile
```

Se tento di assegnare direttamente "__nome" e "__cognome" il comando viene ignorato:

```
p1.__nome = 'prova'
p1.stampa()
            # il risultato rimane: stefano riccio
            via html n. 10
```

Overloading degli operatori in python

Esiste un altro concetto della programmazione orientata agli oggetti molto interessante. Esso consiste nella possibilità di effettuare il polimorfismo anche degli operatori, oltre che dei metodi.
Supponiamo di creare la classe dei numeri complessi nel seguente modo:

```
class num_comp:
        parte_reale = 0
        parte_immaginaria = 0

        def somma(self,num):
          # num è un oggetto di classe num_comp
          self.parte_reale = self.parte_reale + num.parte_reale
          self.parte_immaginaria = self.parte_immaginaria +
          num.parte_immaginaria

        def stampa(self):
          print str(self.parte_reale) + '+'
          +str(self.parte_immaginaria) + 'i'
```

Un numero complesso è composto da due valori: la parte reale e la parte immaginaria, e un metodo che ne fa la somma.
Per utilizzare questa classe devo scrivere il seguente programma:

```
n1 = num_comp()
n1.parte_reale = 1
n1.parte_immaginaria = 1

n2 = num_comp()
n2.parte_reale = 2
n2.parte_immaginaria = 3

n1.somma(n2)
n1.stampa()    # risultato 3+4i
```

La sintassi, purtroppo, risulta piuttosto articolata. Che bello sarebbe poter scrivere (n1 **+** n2) !, proprio come si fa in matematica !.
Per poter scrivere questo è necessario rendere l'operatore "**+**" del python polimorfico, in modo che si accorga di lavorare con i numeri complessi e richiami la giusta funzione.

Per fare questo si deve dichiarare la classe nel seguente modo:

```
class num_comp:
        parte_reale = 0
        parte_immaginaria = 0

        def __add__(self,num):
          ris = num_comp()
          ris.parte_reale = self.parte_reale + num.parte_reale
          ris.parte_immaginaria = self.parte_immaginaria
          + num.parte_immaginaria
          return ris

        def stampa(self):
          print str(self.parte_reale) + '+'
          +str(self.parte_immaginaria) + 'i'
```

Ho dichiarato una funzione speciale, denominata "**__add__**", in grado di effettuare l'overloading (la sostituzione) dell'operatore ADD (+).
Essa crea un nuovo oggetto e somma le singole variabili. Vediamo la nuova sintassi di chiamata della funzione:

```
n1 = num_comp()
n1.parte_reale = 1
n1.parte_immaginaria = 1
```

```
n2 = num_comp()
n2.parte_reale = 2
n2.parte_immaginaria = 3

r = n1 + n2     # come in matematica !
r.stampa()      # risultato 3+4i
```

Attraverso l'overloading degli operatori si possono creare classi molto sofisticate utilizzabili con una sintassi veramente molto elegante.

Di seguito vi presento una tabella con alcune funzioni speciali con le quali è possibile effettuare l'overloading:

Metodo speciale	Descrizione	Esempi
__init__	Costruttore di oggetto	p1 = persona()
__add__	Operatore di somma	n1 + n2
__or__	Operatore OR	x ¦ y
__len__	Lunghezza	len(x)
__cmp__	Confronto	x == y

Classi e moduli

I sorgenti python sono raggruppati in moduli. Anche i moduli, come le classi, sono uno strumento per organizzare bene un programma e ottenere una componentistica da riutilizzare in futuro.
Si pone il problema di come organizzare le classi nei moduli. È possibile inserire più classi nello stesso modulo, ma io consiglio di utilizzare un modulo per ogni classe.
In questo modo i files sul disco equivalgono alle classi implementate e si ottiene un miglior ordine del programma. Si troveranno bene con questo sistema coloro che provengono dal linguaggio Delphi !.

Conclusioni

Una considerazione finale prima di concludere l'argomento della programmazione ad oggetti.
È difficile esaurire l'argomento della programmazione ad oggetti, poiché esso è molto vasto e articolato.
Bisogna tenere in considerazione che ancora oggi è difficile programmare bene ad oggetti per diversi motivi:

- Ogni linguaggio di programmazione implementa in modo diverso e solo in parte le linee teoriche della programmazione ad oggetti (anche python non fa eccezione).
- La maggior parte dei linguaggi sono ibridi e quindi non obbligano il programmatore ad usare correttamente gli oggetti.
- Non è semplice modellare un problema della vita reale ad oggetti per trasformalo in un programma.

Python è un linguaggio ibrido, e quindi lascia libero il programmatore di scegliere se utilizzare o meno gli oggetti. Questo potrebbe essere un punto a sfavore di python rispetto ad altri linguaggio object orientd puri, come java per esempio.
Il mio consiglio è quello di iniziare con qualche piccolo esempio orientato ad oggetti e cercare di lavorare in quella direzione, risulta sicuramente un software più pulito ed elegante.

GUI

Arrivati a questo punto qualcuno potrebbe chiedersi se e' possibile, con python, costruire quelle belle applicazioni, in ambiente window, con l'interfaccia grafica, il menu, i bottoni e tutto il resto.
Quello che abbiamo imparato e' utile per costruire degli script da eseguire sulla linea di comando, ma con python si puo' osare di piu'.

Sono stati implementati diversi moduli che permettono di aggiungere ad un programma python una bella interfaccia grafica a finestre. In particolare intendo introdurre qualche concetto in merito a 3 librerie:

- **Tkinter**: Un modulo che permette di interfacciare python a **X-Windows**.
  Infatti Tkinter utilizza il linguaggio **Tk**, utilizzato da molti anni in ambiente unix per sviluppare software in ambiente X-Windows.
  Questa libreria deve essere utilizzata da coloro che intendono sviluppare interfacce grafiche multipiattaforma, quindi programmi che devono girare sotto unix, linux, windows e anche sotto Mac.
  Questa libreria ha il vantaggio di essere gia' compresa nella installazione standard, quindi non sono necessarie installazioni aggiuntive per iniziare a lavorare con l'interfaccia grafica.

- **Win32Extension**: Una famiglia di moduli per costruire interfacce grafiche in ambiente **Microsoft Windows**.
  Win32Extension e' una vasta raccolta di moduli, scritti da Mark Hammond, che permettono di utilizzare le API Win 32 di windows e le MFC (Microsoft Foundation Class).
  Questa libreria deve essere utilizzata da coloro che sono interessati a lavorare esclusivamente in ambiente windows, infatti in tal caso si possono sfruttare tutte le potenzialita' di windows.
  Io ho trovato problematica la ricerca di documentazione in rete, ma se qualcuno e' veramente interessato puo' acquistare il libro dell'autore della libreria, che e' il riferimento assoluto (O'Reilly book Python Programming on Win32).

- **WxPython**: Una libreria che permette di creare delle interfacce grafiche portabili su diverse piattaforme. WxPython e' basata sulla libreria WxWindows, la quale implementa l'intera struttura in C++. Attualmente WxWindows (e di conseguenza WxPython) supporta MS Windows (16-bit, Windows 95 and Windows NT), Unix con GTK+, Unix con Motif, e Mac.
  Questa libreria e' simile a Tkinter, ma molto piu' moderna e potente. L'intera libreria e tutte le informazioni presso il sito ufficiale (www.wxpython.org).

Di seguito vi propongo esempi nelle tre diverse librerie:

Un piccolo esempio con Tkinter

Per utilizzare Tkinter e' necessario compiere le seguenti operazioni:

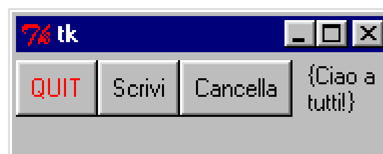
- Importare il modulo chiamato (in modo originale) "Tkinter".
- Inizializzare l'ambiente grafico richiamando la funzione "Tk()".
- Ogni finestra grafica deve essere creata partendo da un oggetto base chiamato "Frame". Il frame e' un contenitore di altri oggetti grafici (che in Tk vengono chiamati "**Widgets**").
- Dopo aver creato il frame, dobbiamo creare tutti i widgets contenuti in esso, nel nostro esempio tre bottoni e una casella di testo.
- Una volta definito l'aspetto grafico della applicazione, si deve richiamare la funzione **mainloop()** incaricata di far partire il programma e di rimanere in attesa di input dall'utente. L'input dell'utente proviene dalla azione che egli effettua con il mouse sui widgets.

Vediamo il listato completo del programma:

```
from Tkinter import *         # importo il modulo

# costruisco una mia classe che gestisce la finestra
class Application(Frame):

    # metodo che scrive un messaggio a video
    def scrivi_messaggio(self):
```

```
                self.mess["text"] = "Ciao a tutti!",

    # metodo che pulisce il messaggio a video
    def cancella_messaggio(self):
                self.mess["text"] = "",

    # metodo costruttore che crea gli oggetti grafici
    def __init__(self, master=None):
                f = Frame(master)
                f.pack()

        # crea il bottone di uscita (di colore rosso)
        self.esci = Button(f)
        self.esci["text"] = "QUIT"
        self.esci["fg"]   = "red"
        self.esci["command"] =  f.quit
        self.esci.pack({"side": "left"})

        # crea il bottone che permette di scrivere il messaggio
        self.butt_mess = Button(f)
        self.butt_mess["text"] = "Scrivi",
        self.butt_mess["command"] = self.scrivi_messaggio
        self.butt_mess.pack({"side": "left"})

        # crea il bottone che permette di pulire il messaggio
        self.butt_canc_mess = Button(f)
        self.butt_canc_mess["text"] = "Cancella",
        self.butt_canc_mess["command"] = self.cancella_messaggio
        self.butt_canc_mess.pack({"side": "left"})

        # crea l'oggetto grafico che contiene il messaggio
        self.mess = Message(f)
        self.mess["text"] = "",
        self.mess.pack({"side": "left"})


# corpo principale del programma
finestra = Tk()
app = Application(finestra)
finestra.mainloop()
```

Ecco come si presenta graficamente la applicazione:



Ho cercato di commentare il codice in modo da far capire i vari passaggi.
Si puo' notare che ho programmato ad oggetti. Infatti ho creato una classe, denominata "Applicazione", che contiene i dati e le operazioni necessarie alla costruzione della interfaccia grafica.
I dati sono stati creati dinamicamente al momento della loro assegnazione e sono i seguenti:

- L'oggetto "esci" che e' il bottone di uscita.
- L'oggetto "butt_mess" che e' il bottone che scrive il messaggio.
- L'oggetto "butt_canc_mess" che e' il bottone che cancella il messaggio.
- L'oggetto "mess" che e' la zona grafica che contiene il messaggio.

Per tutti questi oggetti ho fatto 3 operazioni:

- ho creato una istanza,
- ho inizializzato alcuni dati membro che ne controllano l'aspetto grafico e
- ho chiamato la funziona **pack** che li visualizza a video.

Per quanto concerne i bottoni, il dato membro "command" richiede il nome della funzione che risponde all'evento della pressione del tasto. Infatti quando viene premuto il tasto "butt_mess" python invoca la funzione denominata "scrivi_messaggio".

Il corpo principale del programma, in fondo al codice, non fa altro che eseguire le tre operazioni principali che ho elencato precedentemente.

Chiaramente in questa sede e' impossibile esaminare tutto il modulo, il quale e' ricco di oggetti grafici di ogni tipo. È possibile scaricare dal sito di python un discreto manuale in inglese, chiamato Tkinter Life Preserver, con la spiegazione di tutti gli oggetti.

Un piccolo esempio con Win32Extension

Per utilizzare le librerie di Mark Hammond si deve conoscere il mondo windows, e in particolare le Api Win32. Quindi, anche in questo caso non posso approfondire l'argomento, anche perche' sarebbe piu' vasto di quello che si possa immaginare.

Vi offro comunque un esempio con qualche piccola spiegazione.
Il processo di creazione di una finestra in windows e' piuttosto articolato, e' necessario svolgere i seguenti passi:

- Si deve creare una classe di appartenenza della finestra. Tale classe definisce le funzionalita' della finestra, ad esempio quale funzione e' responsabile della sua gestione. In windows infatti, per ogni finestra grafica, esiste una funzione che gestisce completamente tale finestra.
- Si deve registrare in windows la classe appena creata, in modo che esso ne venga a conoscenza. Questa operazione in python si svolge richiamando la funzione "win32ui.RegisterWndClass"
- Si deve creare l'oggetto finestra con tutte le sue caratteristiche grafiche. Questa operazione in python si svolge richiamando la funzione "CreateWindowEx"
- Si deve richiamare una funzione, denominata "ShowWindow", incaricata di mostrare a video il risultato della finestra.

```
# importo tutti i moduli necessari
import win32con
import win32ui
from pywin.mfc import window, dialog, thread, afxres

# classe che crea la finestra (eredita da window.Wnd)
class HelloWindow(window.Wnd):
    def __init__(self):
# creo un oggetto finestra
        window.Wnd.__init__(self, win32ui.CreateWnd())

    classe = win32ui.RegisterWndClass(0, 0, win32con.COLOR_WINDOW + 1)
    zona = (100, 100, 400, 300)
# dico a windows di creare visivamente la finestra
con tutte le sue caratteristiche

        self._obj_.CreateWindowEx(win32con.WS_EX_CLIENTEDGE, \
            classe,'Ciao a tutti!', win32con.WS_OVERLAPPEDWINDOW, \
            zona, None, 0, None)

# classe che definisce la applicazione (eredita da thread.WinApp)
class HelloApp(thread.WinApp):

# metodo che inizializza l'istanza grafica della applicazione
    def InitInstance(self):
        self.frame = HelloWindow()
```

```
        self.frame.ShowWindow(win32con.SW_SHOWNORMAL)
# indica a windows che si tratta della finestra
principale della applicazione

        self.SetMainFrame(self.frame)

# corpo principale del programma, istanzia la applicazione
app = HelloApp()
```

Osservando il codice si possono fare le seguenti osservazioni:

- È necessario importare un insieme di moduli che definiscono tutte le classi di windows. Ad esempio il modulo "win32con" contiene tutte le costanti utilizzate nelle Api Win32.
- Il corpo principale del programma richiama semplicemente una classe, denominata "HelloApp", che gestisce la funzione responsabile della finestra grafica.
- La classe "HelloApp" richiama automaticamente il metodo "InitInstance". Tale richiamo e' contenuto nel costruttore della classe "thread.WinApp". Infatti, utilizzando le MFC si deve programmare solamente effettuando l'overloading dei metodi che si devono personalizzare. In questo esempio la classe "HelloApp" ridefinisce il metodo "InitInstance" al fine di indicare a windows di caricare la nostra nuova finestra. Quindi il programmatore perde la classica sequenzialita' delle operazioni e deve conoscere molto bene cio' che avviene nelle classi madri da cui si deriva.
- nel metodo "InitInstance" vengono svolte tutte le operazioni, compresa la creazione di un oggetto della classe "HelloWindow", incaricata di definire le caratteristiche grafiche della finestra.

Questa applicazione e' veramente minimale, in quanto si limita a mostrare una finestra vuota con il titolo "Ciao a tutti!"

Un esempio con WxPython

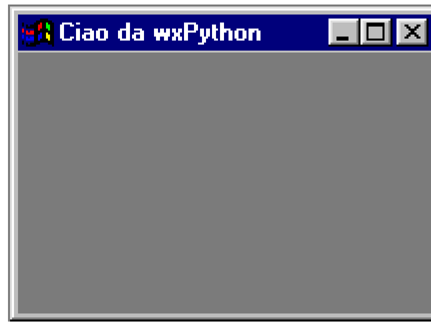
Per utilizzare WxPython e' necessario compiere le seguenti operazioni:

- Importare il modulo chiamato (in modo originale) "wxPython.wx".
- Creare una classe derivata da "WxApp", la quale rappresenta l'intera applicazione. La classe "WxApp" gestisce il sistema dei messaggi e il loop degli eventi.
- Nel metodo "OnInit" di WxApp e' necessario inizializzare l'applicazione. In particolare creare il frame e mostrare la finestra. Il frame rappresenta la finestra principale.

Vediamo il listato completo del programma:

```
from wxPython.wx import *

class MyApp(wxApp):    # derivo la classe applicazione
    def OnInit(self):
        frame = wxFrame(NULL, -1, "Ciao da wxPython") #creo il frame
        frame.Show(true)       #mostro il frame
        return true

app = MyApp(0)  # istanzio l'applicazione
app.MainLoop()  # faccio partire il loop che gestisce gli eventi
```

Ecco come si presenta graficamente la applicazione:



Anche in questo caso ho cercato di commentare il codice in modo da far capire i vari passaggi.
Si puo' notare che tutta la libreria e' organizzata ad oggetti. Infatti ho derivato la classe applicazione e ne ho personalizzato il comportamento.
Il corpo principale del programma, in fondo al codice, non fa altro che istanziare la classe applicazione e far partire il loop che aspetta input dall'utente e gestisce gli eventi.
Anche in questo caso ho fatto un esempio semplicissimo, il passo successivo sarebbe quello di creare dei controlli specifici all'interno del frame. Ma rimando questo ai manuali di WxPython.

Python e i database

Ogni linguaggio di programmazione, degno di tale nome, deve avere degli strumenti per accedere ai maggiori sistemi di gestione di base dati relazionali. Anche python dispone di strumenti atti a risolvere questa problematica.
A tale scopo è stata sviluppata una raccolta di moduli, denominata "**DB-API**".
Tale libreria ha lo scopo di creare un interfaccia unica di accesso ai database, indipendentemente dal tipo di sistema utilizzato. Per fare questo sono stati sviluppati diversi strati:

- Uno strato unico di accesso ai dati, composto da un insieme di funzioni standard.
- Diversi drivers specifici per ogni tipo di database. A tal proposito, esistono drivers per mySQL, Informix, DB2 e anche per ODBC.

Per accedere, ad esempio, a qualsiasi fonte dati ODBC, è necessario caricare l'apposito modulo (chiamato "odbc"). Nel caso di odbc, l'operazione è estremamente semplice. Infatti, una volta installate le win32 extensions si ha gia' il modulo odbc disponibile.
Vi illustro di seguito un esempio con il quale utilizzo il linguaggio SQL per fare delle interrogazioni ad un database.

Facciamo le seguenti ipotesi di lavoro, riferite chiaramente all'ambiente Microsoft Windows:

- Ho creato un database in Microsoft Access denominato "db_articoli.mdb".
- Ho creato un DSN (Data Source Name) con ODBC Manager, denominato "articoli" (vedi il disegno). Esso associa il nome logico "articoli" al file fisico "db_articoli.mdb".
- Il database contiene una tabella, denominata "elenco_articoli", con due campi: codice e descrizione.

Voglio costruire un programma che mostra tutti i record della tabella "elenco_articoli". Ecco il semplice listato che risolve questo problema:

```
import odbc
try:
        s = odbc.odbc('articoli')      # mi collego al DSN
        cur = s.cursor()
        cur.execute('select * from elenco_articoli')
        rec = cur.fetchall()
```

```
        print 'Codice    -- Descrizione       \n'
        for i in rec:
                print i[0] + ' -- ' + i[1] + '\n'
except:
        print 'errore'
```

Il risultato del programma potrebbe essere il seguente:

```
Codice    -- Descrizione

233412 -- matita

567543 -- quaderno

533232 -- gomma
```

Analizzando il codice si possono fare le seguenti osservazioni:

- la variabile "s" rappresenta l'oggetto database. La variabile è stata inizializzata utilizzando il modulo odbc.
- la variabile "cur" rappresenta un cursore sul database. Quindi attraverso questo cursore è possibile navigare nella struttura del database.
- attraverso il metodo "execute" è possibile eseguire una richiesta SQL.
- Il metodo "fetchall()" restituisce una lista contenente tutti i record risultanti dalla query SQL. Ogni elemento della lista "rec" è a sua volta una lista con i valori dei campi. In questo caso i campi sono due:
  - Codice, nella posizione 0;
  - Descrizione, nella posizione 1;
- Con un semplice ciclo posso stampare tutti i record della tabella.

Tutte le funzioni utilizzate fanno parte dello strato standard definito dalla libreria DB-API. Questo significa che il codice non cambia se si utilizza un altro Database al posto di Microsoft Access. L'unico cambiamento riguarda l'utilizzo del modulo apposito.

Python e cgi
Programmazione CGI

Python ha avuto un discreto successo nello sviluppo di applicazioni web. In particolare viene utilizzato come linguaggio di script richiamato da un server web attraverso la metodologia CGI (common gateway interface).

Il meccanismo CGI funziona nel seguente modo:
È necessario avere installato sul proprio computer i seguenti programmi:

- Un server web, come Apache o Internet Information Server. Con questo tipo di programma è possibile costruire un sito internet sul vostro computer. Il server web, infatti, risponde alle chiamate in protocollo HTTP e fornisce come risposta dei files HTML contenuti nel vostro Hard Disk.
- L'interprete python.

Potreste decidere di richiamare un programma python attraverso il web accedendo al sito internet creato dal server web.
Per fare questo è necessario installare un modulo (CGI) che permetta di invocare l'interprete python quando l'url presentata al server web ne richieda l'esecuzione.
I moduli CGI si trovano in rete, ne esistono per tutti i principali web server. Ad esempio per Apachee esiste il **mod_python**, sia per linux che per Microsoft Windows.
Una volta installato mod_python, diventa semplice richiamare un programma python.

Ad esempio: supponiamo che sulla nostra macchina sia stato installato un web server che risponde alle richieste http con il dominio "http://www.mio.it".
Per invocare il programma è sufficiente digitare sul web la seguente url:
"http://www.mio.it/mio_programma.py". In questo caso è stato richiesto il programma python "mio_programma.py" contenuto nella root directory del server web.

Vediamo ora un semplice programma CGI in python:

```
def main():
  print 'Content-type: text/html'
  print
  print '<HTML><HEAD><TITLE> Ciao, mondo!</TITLE><BODY>"
  print 'Ciao, mondo!'
  print '</BODY></HTML>'

if (__name__ == '__main__'):
  main()
```

Come si può osservare dal codice, il programma non fa altro che scrivere in output un file html. Infatti il CGI cattura tutto l'output del programma python e lo manda come risposta sul web. L'utente che ha richiesto l'esecuzione del programma otterra' in risposta il seguente documento html:

```
<HTML><HEAD><TITLE> Ciao, mondo!</TITLE><BODY>
Ciao, mondo!
</BODY></HTML>
```

Questo meccanismo apre le porte alla creazione di siti web dinamici. Ad esempio è possibile creare dinamicamente delle pagine web che contengono dati contenuti dentro ad un database. Per fare cio' è sufficiente interfacciarsi ad un DBMS (come abbiamo visto in precedenza) e costruirsi la pagina web da stampare in output con la funzione "print". Per aiutarci a costruire in modo più rapido delle pagine web esiste una libreria ad oggetti che permette di utilizzare delle classi predefinite. Chi è interessato può trovare informazioni presso HtmlGen,

Programmazione Web lato server

I programmi CGI hanno un difetto: non hanno un grande performance. Infatti, tutte le volte che viene richiesta l'esecuzione di un programma CGI, viene creato un nuovo processo per eseguire una istanza dell'interprete python. Per risolvere questo problema diversi costruttori hanno creato dei piccoli trucchi. Ad esempio: in ambito Microsoft si possono creare dei programmi CGI sottoforma di DLL, in modo che possano

essere caricati in memoria una sola volta e condivisi da tutte le richieste.

Tuttavia, per chi ha esigenze di prestazioni molto elevate, esiste un unica vera soluzione: avere il server web nello stesso programma CGI. Meglio ancora: avere il web server scritto nel linguaggio utilizzato, nel nostro caso python.
In questa situazione diventa tutto più semplice; infatti ogni chiamata CGI risulta essere equivalente ad una chiamata di una funzione all'interno dello stesso programma, quindi all'interno dello stesso processo. Solo in questo modo si possono raggiungere le massime prestazioni.
Questo approccio è il medesimo usato da Java, quando esso è utilizzato come linguaggio di sviluppo lato server (le famose Servlet).

È stato sviluppato in python un intero web server, chiamato "**Zope**". Esso permette la pubblicazione di pagine html statiche, ma permette anche il richiamo diretto di programmi python. Zope dispone di una libreria di oggetti gia' pronti per lo sviluppo di applicazioni sul web. Inoltre esso dispone di una estensione dell'html, denominata "**DTML**", molto simile al meccanismo ASP di Microsoft (Active Server Page), la quale permette anche ad un programmatore inesperto di essere in grado di costruire dei siti web dinamici senza grossi sforzi.
Chi fosse interessato alla piattaforma Zope puo andare sul sito ufficiale www.zope.org. In questo sito si può scaricare il software per la piattaforma linux e windows, consultare una ricchissima guida ed eseguire il tutorial.