

Premessa

“Raccolta Python #1” è un pdf contenente le versioni testuali (aggiornate alla **versione 3.4.3** di Python) e, ovviamente, del codice sorgente degli esempi di 28 videotutorial su **Python** e **Tkinter** pubblicati gratuitamente da Francesco Milanese sul suo canale Youtube e sul sito www.francescomilanese.com.

L'ebook, che consta di 129 pagine in formato A4, ha quindi un prezzo volutamente basso perché si rivolge a chi ha seguito i videotutorial gratuiti, li ha trovati interessanti e desidera avere un testo in pdf da poter consultare facilmente (mediante le funzioni di ricerca o stampandolo)... o desidera ringraziare in maniera concreta l'autore dei videotutorial, ottenendo comunque una piccola “guida – reference” su Python e Tkinter.

I primi 15 tutorial riguardano le basi di Python, per chi ha già qualche esperienza di programmazione. Non verranno quindi descritti i costrutti, le variabili, ecc... per chi deve imparare da zero, ma solo come tali elementi vengono definiti e trattati in Python. NOTA: questi script sono stati aggiornati alle librerie della versione 3.4.3 di Python, per cui alcuni elementi potrebbero essere diversi da quelli visibili nei videotutorial (realizzati con la versione 2.7 di Python).

I 13 tutorial sulla programmazione Tkinter trattano invece la definizione di interfacce grafiche per le applicazioni in Python. Verrà quindi mostrato come creare finestre per le applicazioni, definire gli elementi di base dell'interfaccia e lavorare con la Canvas e la visualizzazione delle immagini.

Per il sommario completo: <http://www.francescomilanese.com/python-01-sommario.pdf>

SOMMARIO

Prima parte: le basi di Python

1. Introduzione. Cosa ci serve, cosa è richiesto, cosa faremo. Note sull'indentazione	1
• Introduzione	1
• Scrittura degli script: l'IDLE	2
• Importare moduli in uno script	3
• I blocchi: l'importanza dell'indentazione e dei due punti. Esempi	4
2. Tipi di dati: numeri, booleani, none, null, caratteri, stringhe. Lo slicing	6
• Definire le variabili (e i loro tipi)	6
• Arrotondamento e valori in virgola mobile	7
• Auto-assegnazione (e auto-casting)	8
• Numeri complessi	8
• Valori booleani	8
• Il tipo None	9
• Stringhe: caratteri, parole, frasi	9
• Stringhe – Alcune operazioni e funzioni di base	10
• Lo slicing (notazione due punti) sui dati iterabili	12
• Slicing e stringhe	13
3. Tipi di dati: le liste	14
• Liste e slicing	15
• Metodi di base della classe List	16
4. Tipi di dati: gli insiemi (Set)	19
• Operazioni sugli insiemi	19
• Argomenti possibili per SET(); esempio: stringhe	20
5. Tipi di dati: le Tuple	21
• Il metodo “in”	21
• Indici degli elementi delle Tuple	22
• Tuple e slicing	22
• Output multipli	22

6. Tipi di dati: i Dizionari (Dict)	23
• Creare un Dict (dizionario) anche vuoto	23
• Inserire elementi (coppie chiave-valore) in un dizionario	24
• Recuperare il valore associato ad una chiave	24
• Eliminare un elemento (coppia chiave-valore)	24
• Metodi di base disponibili per i dizionari	25
7. I comandi Dir e Type	27
• Dir	27
• Type	28
8. Casting: conversioni tra tipi di dati differenti	29
• Casting per i tipi primitivi di Python	29
9. Controllo del flusso: il costrutto <i>if</i>	31
• Operatori relazionali per le condizioni	31
10. Controllo del flusso: cicli <i>for</i> e <i>while</i> . Istruzioni <i>break</i> , <i>continue</i> , <i>pass</i>	33
• Il costrutto del ciclo FOR. Indentazione del blocco	33
• La funzione range	33
• FOR sui tipi di dati iterabili	35
• ZIP: scorrere contemporaneamente più oggetti iterabili	35
• Il costrutto WHILE	36
• Le istruzioni BREAK, CONTINUE, PASS	37
11. Definire ed utilizzare le funzioni. Parametri di input e valori restituiti in output	39
• Invocare una funzione	39
• Valori restituiti (output) dalle funzioni	40
• Commento di documentazione per una funzione	40
• Esempi pratici	40
12. Gestione delle eccezioni: try, except, finally, raise	43
• Catturare e gestire le eccezioni: TRY-EXCEPT	43
• Esempio di eccezione e sua gestione	44
• Blocchi ELSE e FINALLY per TRY-EXCEPT	45
• RAISE: lanciare volontariamente un'eccezione	46
13. Classi, oggetti, campi e metodi. Metodi privati. Overloading degli operatori	47
• La parola chiave CLASS	47
• Oggetti: istanze di una classe	48

• L'oggetto SELF	48
• Il costruttore	48
• Metodi privati (funzioni private)	49
• Overloading degli operatori	49
• Classi e overloading degli operatori: esempio completo	51
 14. Ereditarietà	 52
• Creare classi figlie in Python	52
• Istanziare classi figlie in Python	53
• Esempio completo	53
 15. Lettura e scrittura con i file	 54
• Metodo FILE(): l'handler (gestore) dei file	54
• Apertura del file: MODE	54
• Chiudere il file: CLOSE	54
• Scrivere su file	55
• Lettura da file	55
• Esempio completo	55

Seconda parte: creare interfacce grafiche con Tkinter

1. Introduzione. Creare una finestra. Dimensioni, coordinate e titolo della finestra	57
• Eseguire gli script salvati su file .py	57
• Tkinter	58
• Creare una finestra. Dimensioni e coordinate	59
• Definire il titolo della finestra	60
 2. Elementi di interfaccia: Label (etichette di testo) e Button (pulsanti)	 62
• Creare una Label (etichetta, campo testuale)	62
• Posizionamento: PACK	63
• Personalizzare la Label	64
• Creare un Button (pulsante)	64
 3. Posizionare gli elementi dell'interfaccia. Layout GRID	 67
• Riempire lo spazio: FILL	68
• L'attributo SIDE	69
• Un layout ordinato: il metodo GRID. Righe e colonne	71
• Allineare e centrare gli oggetti	72
• Attributi Padding, Colspan, Rowspan	72

4. Aggiungere barra ed elementi del menù	74
• Creare la barra e una voce di menù	75
• ADD_CASCADE e ADD_COMMAND	75
• Il metodo CONFIG per aggiungere la barra alla finestra	76
• Creare un altro menù	76
• Ordine di visualizzazione dei menù nella barra	77
• Associare azioni alle voci di menù	77
5. Creare delle finestre di dialogo	80
• Finestra di dialogo MESSAGEBOX e SHOWINFO	82
• Finestre SHOWWARNING e SHOWERROR	82
• Chiudere la finestra: DESTROY, con conferma	82
6. Altre operazioni effettuabili con i menù	85
• Finestra di dialogo del file browser per aprire un file	85
• Per selezionare una cartella da disco...	86
• Creare voci di menù di tipo Checkbutton	86
• Valori uguali per VAR nei Checkbutton	87
7. Elementi di interfaccia: Radiobutton e Spinbox	90
• Creare i Radiobutton	91
• Valori uguali per Radiobutton diversi: VALUE	93
• Raggruppamento dei Radiobutton: VARIABLE	93
• Spinbox: creazione e parametri	94
• Specificare i valori possibili per una Spinbox: VALUES	95
8. Elementi di interfaccia: Listbox e Slider	97
• LISTBOX	98
• Selezione multipla per le Listbox: MULTIPLE e EXTENDED	99
• Indici degli elementi di una Listbox	99
• Lo Slider: selezione di un valore in un range	100
• Parametri degli Slider: LENGTH, WIDTH, SLIDERLENGTH	101
• Intervallo dei valori di uno Slider: FROM_, TO, TICKS	101
9. La Canvas. Disegnare linee e figure. Visualizzare immagini	104
• Creare una Canvas	105
• Aggiungere elementi alla Canvas: linee e figure	106
• Visualizzare immagini nella Canvas	107
• Immagini, eventi e azioni: cambiare immagine a runtime	109
10. Un esempio completo	112

11. Trascinare una figura sulla Canvas	118
12. Creare un file txt contenente gli elementi di una Listbox	122
13. Dati da file a Listbox, da Listbox a file	125

* * *
* * *

Le basi - Tutorial 1

Introduzione

Cosa ci serve, cosa è richiesto, cosa faremo

Note sull'indentazione

INTRODUZIONE

Python è un potente linguaggio di programmazione di alto livello.

È un linguaggio:

- ***interpretato***: gli script vengono scritti "in chiaro", non c'è bisogno di compilazione e linking: l'interprete Python eseguirà "quello che avete scritto" senza passaggi intermedi;
- ***interattivo***: potete scrivere ed eseguire "al volo" degli script con una GUI interattiva propria del linguaggio, detta IDLE (maggiori dettagli in seguito);
- ***orientato agli oggetti***; supporta la definizione di classi con metodi e campi ed ereditarietà;
- ***portabile***; un programma scritto in Python può essere eseguito con tutti i sistemi operativi più diffusi; inoltre, è sufficiente un compilatore C per creare una versione personalizzata di Python, in quanto tale linguaggio è anche...
- ... ***open source***!

[Curiosità: è stato creato da Guido van Rossum, programmatore olandese che gli diede questo nome per via della sua passione per i Monty Python.]

Questa guida, comunque, NON è una guida di programmazione "da zero", non insegna le basi della programmazione ad oggetti (la conoscenza del concetto di classe, ad esempio, è data per scontata), ma vuole essere un'introduzione alla sintassi e alle funzionalità di base di Python per chi ha già qualche esperienza di programmazione, ma non conosce (ancora) tale linguaggio.

La guida, essendo "di base", NON tratterà, inoltre, argomenti quali la definizione di GUI (trattata invece nella seconda parte, relativa a Tkinter), l'interazione con la rete, l'interazione con i database o il multithreading.

Python è facilmente estendibile mediante l'importazione di **librerie esterne** (dette **moduli**), cosa che consente di aggiungere classi e metodi già disponibili per implementare particolari funzioni senza dover inventare nuovamente la ruota; è possibile quindi creare con facilità applicazioni multimediali, che interagiscano con i database, ecc....

La documentazione ufficiale di Python, completamente gratuita, è disponibile a questo link: <http://docs.python.org/3.1/download.html> .

SCRITTURA DEGLI SCRIPT: L'IDLE

È possibile scrivere script in Python in maniera interattiva, mediante la **GUI di Python** (detta **IDLE**, installata automaticamente insieme al linguaggio), oppure con un editor di testo o un IDE (i file sorgente scritti in linguaggio Python sono tutti in chiaro!) salvando il tutto in un file con estensione **.py**; in questo caso, vi basterà scrivere nel prompt dei comandi del sistema (o creare un file batch che esegua tale comando, o altre soluzioni...):

```
python [nomeFilePython].py
```

per eseguire lo script.

Per eseguire Python in maniera interattiva, vi basterà aprire **l'IDLE** (fornito di serie, quando si installa Python sul proprio sistema) e... cominciare a scrivere!

Il **prompt (la riga di comando)** dell'IDLE è identificato mediante il simbolo `>>>`.

A destra di tale simbolo possiamo quindi iniziare a scrivere le nostre istruzioni, premendo di volta in volta Invio per eseguirle o, come vedremo, aprire un blocco (un insieme di istruzioni, ad esempio per definire una funzione, una classe, un ciclo o un altro costrutto).

Scrivete ad esempio:

```
>>> print("Ciao!")
```

e premete Invio. Avete appena fatto conoscenza con la funzione *print*, che stampa a video un valore (passato come argomento tra le parentesi tonde) e che verrà usata molte volte nel corso della guida.

[NOTA: da qui in poi, la presenza del simbolo >>> servirà ad indicare che quelle che seguono sono le istruzioni da digitare nell'idle o nel file di script; l'assenza di tale simbolo indicherà, invece, la definizione di istruzioni all'interno di blocchi oppure l'output restituito dall'IDLE. La distinzione tra i due casi apparirà chiara a seconda del contesto; se, infine, scriverete i vostri script in file .py, anziché nell'IDLE, allora il simbolo >>> non sarà presente – e non andrà scritto, ovviamente!! – in nessuna riga, in quanto si tratta del simbolo del prompt dei comandi dell'IDLE.]

IMPORTARE MODULI IN UNO SCRIPT

Per importare un modulo (un insieme di funzioni, costanti e altri elementi, utili in casi specifici), è sufficiente scrivere:

```
>>> import [nome modulo]
```

e premere Invio; ad esempio, provate a digitare nell'IDLE le seguenti istruzioni:

```
>>> # Commento. Le righe di commento in Python iniziano col simbolo  
'cancellino'. Mettere un simbolo # all'inizio di ogni riga di commento  
(questa è, quindi, un'unica riga)  
  
>>> # Questo script genera un numero casuale compreso tra 1 e 100 e lo
```

```
stampa a video. Fa uso del metodo randrange di random del modulo random
>>> import random
>>> n = random.randrange(1,100)
>>> print(n)
4
```

Il valore stampato a video sarà quindi un intero tra 1 e 100; da notare, sopra, l'assenza del simbolo `>>>`: il numero 4 è quindi un esempio di output restituito dall'IDLE.

Abbiamo appena importato il **modulo "random"**, che mette a disposizione vari metodi per generare numeri casuali (random, appunto); tra questi metodi, abbiamo *randrange*, che genera un intero casuale del valore compreso in un intervallo specificato (nel nostro caso, tra 1 e 100); con *print*, poi, abbiamo stampato a video il valore contenuto nella variabile *n*, alla quale avevamo assegnato il numero casuale generato mediante il metodo *randrange*.

I BLOCCHI: L'IMPORTANZA DELL'INDENTAZIONE E DEI DUE PUNTI

Prima di iniziare ad esaminare tipi di dati e script più interessanti, comunque, una nota importantissima sul concetto di BLOCCO DEFINITO CON **L'INDENTAZIONE**: in Python, la delimitazione dei blocchi di codice per raggruppare delle istruzioni (ad esempio, per definire cicli o funzioni) non avviene mediante l'uso di parentesi graffe o istruzioni di chiusura (es.: *end*, *end if*, ...), ma semplicemente **RIENTRANDO** il testo delle istruzioni, ossia mediante l'indentazione, con una o più (cicli annidati, blocchi annidati) pressioni del tasto **TAB**.

Nella maggior parte dei casi, comunque, al termine della dichiarazione di inizio di un blocco (*if*, *for*, ecc...) o di una classe o di una funzione, sarà necessario aggiungere il **simbolo : (due punti)** alla fine del rigo, cosa che ci aiuterà a riconoscere l'inizio del blocco. I vari casi saranno trattati comunque uno per uno nel corso della guida.

Esempio 1

Definizione di due *for* annidati in Python (notare l'indentazione, all'interno di ogni blocco, nonché l'assenza del simbolo >>> a partire dal primo *for*, in quanto le istruzioni che seguono fanno parte di quel blocco, per cui per Python si tratta in realtà di un unico elemento):

```
>>> for i in range(1,10):
    for j in range(1,10):
        print(i*j)
    # Qui sono ancora dentro il ciclo di i, ma fuori da quello di
    # j (un solo "colpo di TAB")
>>> # Qui sono fuori da entrambi i cicli (niente indentazione ed è tornato
il simbolo >>> del prompt dell'IDLE)
```

Esempio 2

Definizione di una classe contenente un campo e un metodo in Python:

```
>>> class rettangolo:
    lato1 = 0
    def __init__(self, l1, l2):
        self.lato1 = l1
        self.lato2 = l2
>>> # Qui sono fuori al di fuori della classe (niente indentazione)
```

L'uso dell'indentazione è OBBLIGATORIO: l'interprete Python valuterà come errati (errore di sintassi) gli script privi dell'indentazione (ove richiesta).

Iniziamo quindi a conoscere i vari tipi di dati disponibili in maniera "nativa" in Python e i metodi che ci consentono di lavorare con i dati; passeremo, poi, alle istruzioni dedite al controllo del flusso del programma (costrutto *if*, cicli *for* e *while*), alle funzioni, alle classi (con ereditarietà) ed infine alle istruzioni di base per effettuare operazioni di input-output con i file.

* * *

Le basi - Tutorial 2

Tipi di dati: numeri, booleani, none, null, caratteri, stringhe

Lo slicing

DEFINIRE LE VARIABILI (E I LORO TIPI)

In Python, una variabile viene creata semplicemente specificandone il nome ed assegnandole un valore, senza preoccuparsi di doverne definire il tipo.

I tipi di dati primitivi di Python, comuni anche ad altri linguaggi, sono:

- dati numerici (interi e a virgola mobile);
- booleani (True / False);
- None;
- stringhe (i caratteri singoli vengono trattati come stringhe con un solo elemento).

L'insieme dei dati primitivi di Python è poi arricchito dalla presenza di tipi più "complessi":

- liste;
- insiemi;
- tuple;
- dizionari.

Li prenderemo tutti in esame.

Sarà poi possibile, ovviamente, definire nuove classi, ciascuna con proprio campi e metodi.

Per definire una variabile numerica, in Python, sarà sufficiente quindi scrivere:

```
>>> nomeVariabile = valore;
```

Il limite di dimensione di un numero è quello dato dalla memoria del computer.

Esempio:

```
>>> a = 5;
>>> b = a / 2;
>>> print(a, b)
5 2.5
```

Da notare, quindi, che la funzione `print` può stampare in un colpo solo i valori di più variabili, se questi vengono indicati separati da una virgola quando si invoca la funzione.

ARROTONDAMENTI E VALORI IN VIRGOLA MOBILE

Per specificare valori in virgola mobile, dichiarateli con il `.0` finale (la virgola dei valori decimali, in Python, viene resa con il punto `.`), ad esempio così:

```
>>> c = 2.0
>>> print(c)
2.0
```

o effettuate un **casting** esplicito (conversione forzata del tipo di dato; ne ripareremo più avanti), utile nel caso di variabili intere inserite in espressioni, ad esempio così:

```
>>> b = float(a) / 2
>>> print(b)
2.5
```

Il tipo **float** identifica, appunto, i valori in virgola mobile, e con `float(n)` possiamo convertire in float il valore numerico `n`; allo stesso modo, con `int(p)` possiamo convertire in intero (privo di valori decimali) il parametro `p`.

AUTO-ASSEGNAZIONE (E AUTO-CASTING)

Un variabile può anche essere convertita "al volo", assegnandole... se stessa:

```
>>> b = float(a) / 2
>>> print(b)
2.5
>>> b = int(b)
>>> print(b)
2
```

NUMERI COMPLESSI

È possibile anche definire numeri complessi, utilizzando i suffissi *J* o *j*:

```
>>> valore = 5 - 2j
>>> print(valore)
(5-2j)
```

e ricavarne le parti reali e immaginarie mediante i campi *real* e *imag* (parleremo di campi e metodi degli oggetti in maniera più approfondita in seguito):

```
>>> print(valore.real)
5.0
>>> print(valore.imag)
-2.0
```

VALORI BOOLEANI

In **Python**, i valori booleani sono: ***True*** e ***False***.

È possibile creare una variabile booleana semplicemente assegnandole ***True*** o ***False***, SENZA apici, con i quali si definiscono invece delle *Stringhe* ("True" e "False")!

Scriveremo dunque, ad esempio:

```
>>> mioTrue = True
>>> mioFalse = False
```

Scrivere, invece,

```
>>> stringa1 = "True"
>>> stringa2 = "False"
```

comporterà la creazione di due STRINGHE, ovvero due parole, non due valori booleani.

IL TIPO NONE

Il tipo *None* identifica... un valore nullo, senza tipo! In altri linguaggi, la sua controparte è *null*.

Può essere utilizzato, ad esempio, per creare una variabile senza darle un valore preciso (magari perché ancora non è disponibile, ma va calcolato), ad esempio con:

```
>>> nullo = None
>>> print(nullo)
None
```

Notare l'assenza, sia nella definizione che nell'output, di apici o doppi apici: al pari dei booleani *True* e *False*, *None* è un tipo particolare, non una stringa.

STRINGHE: CARATTERI, PAROLE, FRASI

Le Stringhe (parole, insiemi di caratteri, ma anche intere frasi) e i caratteri (singole lettere, viste da Python come stringhe formate da un solo elemento) vanno creati assegnando ad una variabile una o più lettere racchiuse tra apici o doppi apici:

```
>>> stringa1 = 'miaStringa1'
>>> stringa2 = "miaStringa2"
>>> print(stringa1, stringa2)
miaStringa1 miaStringa2
```

In realtà, quindi, non abbiamo un tipo di dato a parte per i caratteri: si tratta di stringhe di un solo elemento.

Le stringhe sono immutabili, nel senso che non è possibile sostituire una parte di una stringa: per modificare una stringa, occorre riassegnarla completamente.

STRINGHE – ALCUNE OPERAZIONI E FUNZIONI DI BASE

È possibile effettuare delle concatenazioni di stringhe mediante il simbolo + :

```
>>> stringa1 = "stringa1"
>>> stringa2 = "stringa2"
>>> stringa3 = stringa1 + "; " + stringa2
>>> print(stringa3)
stringa1; stringa2
```

Oltre a concatenare due stringhe, possiamo anche moltiplicare una stringa per un numero: l'effetto sarà quello di ripetere la stringa per quel dato numero di volte; es.:

```
>>> prova = "prova"
>>> prova = prova * 5
>>> print(prova)
provaprovaprovaprovaprovaprova
```

Con le stringhe è possibile utilizzare particolari caratteri, detti *caratteri di escape*, che implementano operazioni particolari all'interno di una stringa (gli effetti saranno visibili in fase di visualizzazione della stringa):

- `\n` : "line feed" (ritorno a capo)
- `\b` : "backspace"
- `\t` : "tab" (tabulazione)

Ad esempio, digitando nel prompt dell'IDLE le seguenti due istruzioni:

```
>>> stringa = "Ciao come stai?\nBene, grazie!\tTu?"
>>> print(stringa)
```

Python stamperà a video il seguente risultato:

```
Ciao come stai?  
Bene, grazie! Tu?
```

Le stringhe sono degli oggetti non banali, provvisti di un gran numero di metodi; ad esempio:

- il metodo ***find("[carattere]")*** cerca il carattere passato come parametro nella stringa e restituisce l'indice (la posizione) di tale carattere, se presente; in caso di più risultati, restituirà l'indice della prima occorrenza di *[carattere]*; restituirà, infine, *-1* qualora *[carattere]* non dovesse essere trovato:

```
>>> prova = "ciao come stai ?"  
>>> prova.find("c")  
0  
>>> prova.find("i")  
1  
>>> prova.find("x")  
-1
```

- il metodo ***strip()*** rimuove gli spazi all'inizio e alla fine della stringa;
- il metodo ***replace(sottostringa1, sottostringa2)*** sostituisce la porzione di stringa "sottostringa1" con la stringa "sottostringa2", come mostra l'esempio:

```
>>> stringaoriginale = "Ciao a tutti!"  
>>> stringa2 = stringaoriginale.replace("a tutti", "amico mio")  
>>> print(stringa2)  
Ciao amico mio!
```

- il metodo ***split("[stringa di splitting]")*** suddivide la stringa in più parti ad ogni occorrenza di *[stringa di splitting]*; le varie stringhe "generate" verranno inserite in un oggetto di un particolare tipo, la *lista*, che tratteremo nel prossimo capitolo; es.:

```
>>> prova.split(" ")  
['ciao', 'come', 'stai', '?']
```

- il metodo ***join*** unisce gli elementi presenti in una lista in un'unica stringa, separandoli con la stringa che invoca tale metodo; esempio:

```
>>> nuovaStringa = " ".join(['ciao', 'come', 'stai', '?'])
>>> print(nuovaStringa)
ciao come stai ?
>>>
```

Per una trattazione completa sulle funzioni native delle Stringhe, si rimanda alla documentazione ufficiale di Python.

LO SLICING (NOTAZIONE DUE PUNTI) SUI DATI ITERABILI

Una tecnica utilizzabile sulle stringhe e su tutti gli oggetti "iterabili" (ossia, composti da sequenze di oggetti, sulle quali sequenze è possibile effettuare un'iterazione, per "scorrerle"; le stringhe sono, in effetti, sequenze ordinate di caratteri) è quella dello "***slicing***" (lett.: tagliare a fette); in particolare, attraverso la notazione

```
elementoIterabile[indicePartenza : indiceArrivo]
```

È possibile estrapolare una parte di un elemento, in particolare quella racchiusa tra l'indice *indicePartenza* e l'indice *indiceArrivo*; se NON specificheremo uno dei due indici, Python interpreterà la cosa come "dall'inizio" (se non specificheremo il primo indice) o "fino alla fine" (se non specificheremo l'ultimo).

Esempio:

```
>>> stringa = "Ciao come stai ?"
>>> stringa1 = stringa[:2]
>>> print(stringa1)
Ci
>>> stringa2 = stringa[3:5]
>>> print(stringa2)
o
```

```
>>> stringa3 = stringa[5:]
>>> print(stringa3)
come stai ?
>>>
```

SLICING E STRINGHE

In Python, le stringhe sono immutabili, per cui non potremo usare lo slicing per riscrivere porzioni di una stringa, mentre potremo farlo con altri tipi di dati come le liste, come vedremo nel prossimo capitolo.

È possibile recuperare velocemente l'ultimo carattere di una stringa (e, in generale, l'ultimo elemento di un oggetto iterabile) con:

```
[-1]
```

Esempio:

```
>>> stringa = "Ciao come stai ?"
>>> ultimoElemento = stringa[-1]
>>> print(ultimoElemento)
?
>>> print(stringa[-1])
?
>>> stringa[-1]
'?'
>>>
```

* * *

Le basi - Tutorial 3

Tipi di dati: le liste

Una **lista** è un **insieme ordinato** (nel senso che ogni elemento ha un indice, un posto) di **elementi eterogenei**.

Le **liste** vengono create assegnando, ad una **variabile**, un **insieme di valori** (anche nessuno: **lista vuota**) separati da **virgole** e racchiusi tra **parentesi quadre**:

```
>>> miaLista = ['a', 6, 'ciao', 5]
>>> print(miaLista)
['a', 6, 'ciao', 5]
>>>
```

Il primo **indice** di una lista ha posto **0**, mentre è possibile accedere all'**ultimo elemento** di una **lista** con **-1**, al **penultimo** con **-2** e così via:

```
>>> miaLista = [1, 2, 3, 4, 5]
>>> miaLista[0]
1
>>> miaLista[-1]
5
>>> miaLista[-2]
4
>>>
```

Qualunque cosa può far parte di una **lista**, anche un'altra **lista**:

```
>>> miaLista = [4, 'ciao']
>>> miaLista2 = [2, miaLista, 'bravo']
>>> print(miaLista2)
[2, [4, 'ciao'], 'bravo']
>>> miaLista2[1]
[4, 'ciao']
>>>
```

LISTE E SLICING

Come avviene con altri **oggetti iterabili**, con le **liste** è possibile utilizzare la tecnica dello **slicing**, ed anzi è possibile anche **modificare** le **liste** con tale tecnica (a differenza delle **stringhe**, le **liste NON sono immutabili**):

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista[:3]
['ciao', 2, 1]
>>> miaLista2 = miaLista[3:5]
>>> print(miaLista2)
['a', True]
>>> miaLista[-3:] = [] # Assegno agli ultimi tre elementi della lista la
lista vuota; in pratica, li cancello!
>>> print(miaLista)
['ciao', 2, 1, 'a']
>>>
```

Tramite lo **slicing**, poi, è possibile **duplicare** facilmente una **lista**:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista2 = miaLista[:]
>>> print(miaLista2)
['ciao', 2, 1, 'a', True, None, 'bye']
>>>
```

METODI DI BASE DELLA CLASSE LIST

Ecco alcuni dei **metodi** disponibili per gli **oggetti di tipo list** (le **liste**, appunto):

- **append**

Consente di aggiungere l'elemento passato come parametro in coda alla lista che lo sta invocando.

Esempio:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista.append("prova")
>>> print(miaLista)
['ciao', 2, 1, 'a', True, None, 'bye', 'prova']
>>>
```

- **insert**

Inserisce un dato elemento in una posizione specificata, spostando gli altri elementi (non sovrascrive elementi esistenti).

Esempio:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista.insert(3, False)
>>> print(miaLista)
['ciao', 2, 1, False, 'a', True, None, 'bye']
>>>
```

- **extend**

Come `append`, ma per le liste: accetta come parametro una lista, ed effettua una "fusione" tra le due, aggiungendo gli elementi della seconda lista in coda alla prima.

Esempio:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista2 = ["Hello", 4, False]
>>> miaLista.extend(miaLista2)
>>> print(miaLista)
['ciao', 2, 1, 'a', True, None, 'bye', 'Hello', 4, False]
>>>
```

- **remove**

Rimuove, dalla lista che lo invoca, l'elemento posizionato nel posto indicato dal parametro passato a tale metodo.

Esempio:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista.remove(2)
>>> print(miaLista)
['ciao', 1, 'a', True, None, 'bye']
>>>
```

- **index**

Restituisce il l'indice (il "posto") dell'elemento passato come parametro nella lista, se presente (posizione iniziale: 0); genera errore se l'elemento non è presente

Esempio:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista.index(2)
1
>>> miaLista.index("hello")

Traceback (most recent call last):
  File "<pyshell#199>", line 1, in <module>
    miaLista.index("hello")
ValueError: 'hello' is not in list
>>>
```

- **sort**

Ordina (con criterio alfabetico e/o numerico crescente; None precede tutti) gli elementi contenuti in una lista; non è possibile utilizzarlo su liste contenenti interi e stringhe (errore “tipi non ordinabili”, in Python 3.4).

Esempio:

```
>>> listaNumeri = [5, 3, 20, 4, 10, 22]
>>> print(listaNumeri)
[5, 3, 20, 4, 10, 22]
>>> listaNumeri.sort()
>>> print(listaNumeri)
[3, 4, 5, 10, 20, 22]
```

- **reverse**

Inverte l'ordine degli elementi contenuti nella lista che lo invoca:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> print(miaLista)
['ciao', 2, 1, 'a', True, None, 'bye']
>>> miaLista.reverse()
>>> print(miaLista)
['bye', None, True, 'a', 1, 2, 'ciao']
```

- **len**

Sta per length (lunghezza), restituisce -sotto forma di numero intero, utilizzabile come variabile - il numero di elementi che compongono una lista; esempio:

```
>>> miaLista = ['ciao', 2, 1, 'a', True, None, 'bye']
>>> len(miaLista)
7
>>> lunghezza = len(miaLista)
>>> print(lunghezza)
7
```

- **count**

Conta il numero di occorrenze dell'elemento passato come parametro nella lista che lo invoca; attenzione: NON è ricorsivo, nel senso che non cercherà l'elemento passato come parametro in eventuali liste contenute all'interno della lista che lo invoca.

Esempio:

```
>>> miaLista = [1, 2, 3, 4]
>>> miaLista.count(4)
1
>>> miaLista2 = [1, 2, miaLista]
>>> miaLista2.count(3)
0
```

Questi sono solo i **metodi** principali dell'oggetto **list**; una trattazione completa, si rimanda alla documentazione ufficiale di Python.

* * *

Le basi - Tutorial 4

Tipi di dati: gli insiemi (set)

Gli **insiemi** ("set") sono gruppi di **elementi indicizzati** che **non** contengono **duplicati** (eventuali duplicati inseriti in fase di creazione verranno ignorati da **Python**, che inserirà solo una istanza dell'elemento).

Vengono creati mediante la **funzione** `set(elemento)`:

```
>>> mioInsieme = set(['ciao', 'come', 'stai'])
>>> mioInsieme
{'stai', 'ciao', 'come'}
>>> mioInsieme2 = set(['ciao', 'bene', 'grazie'])
```

OPERAZIONI SUGLI INSIEMI

Le **operazioni** effettuabili sugli **insiemi** matematici (**unione**, **differenza**, **intersezione**, **xor**) per generare nuovi **insiemi** possono essere effettuate anche sui **set** di **Python**:

- **l'unione** viene implementata mediante il simbolo `|` (in pratica, è un OR):

```
>>> insiemeUnione = mioInsieme | mioInsieme2
>>> insiemeUnione
{'bene', 'stai', 'grazie', 'ciao', 'come'}
```

- **la differenza** viene implementata mediante il simbolo `-`:

```
>>> insiemeDifferenza = mioInsieme - mioInsieme2
>>> insiemeDifferenza
{'come', 'stai'}
```

- **l'intersezione** viene implementata mediante il simbolo **&** (in pratica, è un AND):

```
>>> insiemeIntersezione = mioInsieme & mioInsieme2
>>> insiemeIntersezione
{'ciao'}
```

- infine, abbiamo lo **XOR**, implementato mediante il simbolo **^** :

```
>>> insiemeXOR = mioInsieme ^ mioInsieme2
>>> insiemeXOR
{'come', 'grazie', 'bene', 'stai'}
```

ARGOMENTI POSSIBILI PER SET(); ESEMPIO: STRINGHE

Negli esempi precedenti ho utilizzato delle **sequenze (liste) di elementi** (racchiuse tra le parentesi quadre [] e separate da virgole), ma l'argomento di **set()** può essere un elemento qualsiasi, con esiti differenti; passando, ad esempio, una stringa, essa verrà scomposta nelle lettere che la formano, ma attenzione: **set NON considera i duplicati**, per cui potrebbe verificarsi una situazione del genere:

```
>>> insiemeDaStringa = set("ciao come stai")
>>> insiemeDaStringa
{'a', ' ', 'c', 'e', 'i', 'm', 'o', 's', 't'}
```

* * *

Le basi - Tutorial 5

Tipi di dati: le Tuple

Le **tuple** sono **sequenze non ordinate e immutabili** di oggetti di **vario tipo**.

Vengono create utilizzando le **parentesi tonde ()** e separando gli elementi mediante la **virgola**:

```
>>> miaTupla = (1, 'c', 'parola', 10)
```

Una **tupla** con un solo elemento va creata così:

```
>>> nuovaTupla = ('x',)
```

ossia, con il **singolo elemento seguito da una virgola**, il tutto dentro le parentesi tonde.

Come anticipato, **non possono essere modificate**; scrivere ad esempio:

```
>>> miaTupla[1] = 'ciao'
```

non modificherà l'elemento della **tupla**, ed anzi genererà un'**eccezione** (errore).

IL METODO “IN”

Le **tuple** mettono a disposizione il **metodo in**, che restituisce un **booleano** (True / False) per verificare se un elemento appartiene alla **tupla**:

```
<valore> in <tupla>
```

Esempio:

```
>>> risposta = 'parola' in miaTupla
>>> print(risposta)
True
```

INDICI DEGLI ELEMENTI DELLE TUPLE

È possibile recuperare un elemento della **tupla** con un determinato **indice** con le parentesi quadre:

```
>>> miaTupla[2]
'parola'
```

(il primo elemento ha **indice 0**).

Per recuperare velocemente l'**ultimo elemento** di una **tupla**, sarà sufficiente usare l'**indice -1**:

```
>>> miaTupla[-1]
10
```

TUPLE E SLICING

Con le **tuple** è infine possibile utilizzare la tecnica dello **slicing**, per estrapolare "**porzioni**" di **tupla**; ad esempio:

```
>>> miaTupla[:2]
(1, 'c')
```

OUTPUT MULTIPLI

Da notare che, nel caso di **più valori di output**, gli elementi verranno restituiti come componenti di una **nuova tupla**, infatti potremmo creare nuove **tuple** proprio con questa tecnica:

```
>>> nuovaTupla = miaTupla[:2]
>>> print(nuovaTupla)
(1, 'c')
```

* * *

Le basi - Tutorial 6

Tipi di dati: i Dizionari (Dict)

La struttura dati **dizionario (dict)** è, sostanzialmente, un **array associativo**, ovvero una collezione di **coppie di elementi** (una “*chiave*”, che deve essere univoca, e un “*valore*”, associato alla chiave, che può essere presente anche più volte – come valore – nel Dizionario) **non ordinati**.

CREARE UN DICT (DIZIONARIO), ANCHE VUOTO

Si crea un **dizionario** mediante la **funzione *dict()***:

```
>>> mioDizionario = dict()
```

oppure specificando i valori all'interno di **parentesi graffe**, separando le **coppie chiave-valore** mediante **virgole** e una **chiave** dal proprio **valore** con il simbolo **:** (due punti), in questo modo:

```
>>> nuovoDizionario = {'chiave0' : 0, 'chiave1' : True}
>>> print(nuovoDizionario)
{'chiave1': True, 'chiave0': 0}
```

Il “**dizionario vuoto**” può essere creato, oltre che mediante *dict()*, con le **parentesi graffe senza contenuto**:

```
>>> dizionarioVuoto = {}
>>> print(dizionarioVuoto)
{}
```

INSERIRE ELEMENTI (COPPIE CHIAVE-VALORE) IN UN DIZIONARIO

A questo punto, possiamo inserire delle **chiavi** nel **dizionario** (con le parentesi quadre) ed associare, a tali **chiavi**, dei **valori**; come **chiavi**, possiamo utilizzare solo **oggetti immutabili** (stringhe, interi, tuple; NON è possibile utilizzare le liste), mentre come **valori** possiamo utilizzare **qualsiasi cosa**, anche funzioni o classi; ad esempio:

```
>>> mioDizionario['chiave1'] = 'stringa'
>>> mioDizionario['chiave2'] = 10
>>> mioDizionario['chiave3'] = None
>>> mioDizionario['chiave4'] = False
```

RECUPERARE IL VALORE ASSOCIATO AD UNA CHIAVE

È possibile recuperare il **valore associato ad una chiave** scrivendo semplicemente:

```
print(mioDizionario[nomeChiave])
```

Esempio:

```
>>> print(mioDizionario['chiave4'])
False
```

ELIMINARE UN ELEMENTO (COPPIA CHIAVE-VALORE)

Un elemento di un dizionario va eliminato mediante **del** e la chiave:

```
del mioDizionario[nomeChiave]
```

Esempio:

```
>>> print(mioDizionario)
{'chiave3': None, 'chiave2': 10, 'chiave1': 'stringa', 'chiave4': False}
>>> del mioDizionario['chiave1']
>>> print(mioDizionario)
{'chiave3': None, 'chiave2': 10, 'chiave4': False}
```

METODI DI BASE DISPONIBILI PER I DIZIONARI

Vediamo ora alcuni dei metodi messi a disposizione dai dizionari.

- **values**

Restituisce, in una lista, i valori presenti nel dizionario; esempio (utilizzando il dict "mioDizionario" creato precedentemente):

```
>>> mioDizionario.values()
[None, 10, 'stringa', False]
```

- **keys**

Restituisce, in una lista, le chiavi presenti nel dizionario; esempio (utilizzando il dict "mioDizionario" creato precedentemente):

```
>>> mioDizionario.keys()
['chiave3', 'chiave2', 'chiave1', 'chiave4']
```

- **items**

Restituisce, in una lista di tuple, le coppie chiave-valore presenti nel dizionario (ogni tupla è costituita da una chiave e dal valore ad essa associato); esempio (utilizzando il dict "mioDizionario" creato precedentemente):

```
>>> mioDizionario.items()
[('chiave3', None), ('chiave2', 10), ('chiave1', 'stringa'),
 ('chiave4', False)]
```

- **clear**

Svuota il dizionario, eliminando tutte le coppie chiave-elemento in esso contenuto:

```
>>> print(mioDizionario)
{'chiave3': None, 'chiave2': 10, 'chiave4': False}
>>> mioDizionario.clear()
>>> print(mioDizionario)
{}
>>>
```

- **in**

Restituisce il booleano *True* se una determinata chiave, passata come parametro, è presente nel dizionario che ha invocato tale metodo, *False* altrimenti; es.:

```
>>> mioDizionario = {'chiave3': None, 'chiave2': 10,
```

```
'chiave1': 'stringa', 'chiave4': False}
>>> print(mioDizionario)
{'chiave3': None, 'chiave2': 10, 'chiave1': 'stringa',
'chiave4': False}
>>> 'chiave2' in mioDizionario
True
>>> 3 in mioDizionario
False
>>>
```

* * *

Le basi - Tutorial 7

I comandi dir e type

Due comandi molto utili per conoscere la **tipologia** o gli **attributi** (campi e metodi) degli **oggetti** sono *dir* e *type*.

DIR

La funzione *dir* ci consente di conoscere gli **attributi** di un **oggetto**.

Esempio:

```
>>> miaLista = ['ciao', 3]
>>> dir(miaLista)
['_add_', '__class__', '__contains__', '__delattr__', '__delitem__',
 '__delslice__', '__doc__', '__eq__', '__ge__', '__getattr__',
 '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__setslice__',
 '__str__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
```

L'output è restituito sotto forma di **lista** (insieme di voci racchiuse tra []), contenente tutti i **campi** e i **metodi** di un oggetto di tipo "list".

TYPE

Dato un **oggetto**, per capire di che **"tipo"** è (quale **classe**), possiamo utilizzare invece *type*:

```
>>> type(miaLista)
<class 'list'>
```

utilizzabile anche con alcuni degli **elementi** restituiti da **dir**.

Questi due comandi possono rivelarsi molto utili per **"scoprire"** le **proprietà degli oggetti**, specialmente quando si ha a che fare con **elementi** provenienti da **moduli esterni**.

* * *

Le basi - Tutorial 8

Casting: conversioni tra tipi di dati differenti

Python permette grande flessibilità nell'utilizzo di vari **tipi di dato**, tuttavia non è possibile effettuare operazioni tra tipi diversi senza un **casting** (conversione forzata); ad esempio, scrivere:

```
>>> elemento1 = 'a'
>>> elemento2 = 4
>>> somma = elemento1 + elemento2
```

genererà un **errore**:

```
Traceback (most recent call last):
File "<pyshell#26>", line 1, in <module> somma = elemento1+elemento2
TypeError: Can't convert 'int' object to str implicitly
```

CASTING PER I TIPI PRIMITIVI DI PYTHON

Il **casting** viene effettuato mediante **funzioni specifiche**; per i **tipi primitivi**, abbiamo ad esempio:

- ***tuple(elemento)***;

che effettua una **conversione a tupla**; funziona solo con **elementi "iterabili"**, come lo **stringhe** o le **liste**:

```
>>> elementoStringa = 'stringa'
>>> convAtupla = tuple(elementoStringa)
>>> convAtupla
('s', 't', 'r', 'i', 'n', 'g', 'a')
```

mentre con altri, come gli **interi**, genera **errore**:

```
>>> intero1 = 5
>>> convAtupla = tuple(intero1)
```

```
Traceback (most recent call last):
File "<pyshell#35>", line 1, in <module>
convAtupla = tuple(interol)
TypeError: 'int' object is not iterable
```

- ***list(elemento);***

conversione a lista; funziona solo con **elementi "iterabili"**, per cui, ad esempio, non è possibile effettuare una conversione passando, come parametro, un **intero**:

```
>>> elementoIntero = 10
>>> convAlista = list(elementoIntero)
Traceback (most recent call last):
File "<pyshell#28>", line 1, in <module>
convAlista = list(elementoIntero)
TypeError: 'int' object is not iterable
```

mentre è possibile **convertire delle stringhe in elementi separati** (caratteri) appartenenti ad una **list**:

```
>>> elementoStringa = 'stringa'
>>> convAlista = list(elementoStringa)
>>> convAlista
['s', 't', 'r', 'i', 'n', 'g', 'a']
```

- ***int(elemento);***

conversione a intero; funziona solo con **numeri e stringhe**, altrimenti genera errore:

```
Traceback (most recent call last):
File "<pyshell#32>", line 1, in <module>
convAintero = int(convAlista)
TypeError: int() argument must be a string or a number, not 'list'
```

Per le varie funzioni di **casting** esistono anche molte **altre forme**, che consentono ad esempio **conversioni tra numeri da intero ad esadecimale**, ecc...; per una trattazione completa sull'argomento, quindi, si rimanda alla documentazione ufficiale di Python.

* * *

Le basi - Tutorial 9

Controllo del flusso: il costrutto IF

L'**if** permette di **verificare** una determinata **condizione** ed intraprendere, in base all'**esito della “domanda”** (l'espressione da verificare con l'*if*, appunto), delle **azioni**.

La *sintassi* è semplicissima (attenzione però all'**indentazione**):

```
if condizione1:
    blocco azione/i se si verifica la condizione 1
elif condizione2:
    blocco azione/i per condizione2
else:
    blocco azione/i per tutti gli altri casi
```

Da notare il segno **:** (due punti) posto alla fine dell'**espressione da verificare**.

Potete utilizzare tutti gli **elif** che vi servono, anche nessuno (neanche l'ultimo **else** è obbligatorio).

OPERATORI RELAZIONALI PER LE CONDIZIONI

Per verificare le condizioni, è possibile avvalersi dei seguenti operatori relazionali:

<	Minore di; es.: 3 < 5
<=	Minore uguale a; es.: 3 <= 5
==	Uguale a; es.: 5 == 5
>=	Maggiore uguale di; es.: 5 >= 3
>	Maggiore di; es.: 5 > 3.
!=	Diverso da; es.: 3 != 5.
<>	Diverso da; es.: 3 <> 5.

Esempio:

```
>>> valore = 5
>>> if valore<3:
    print("minore")
elif valore==3:
    print("uguale")
else:
    print("maggiore") # A questo punto premere Invio un paio di
    volte... ed ecco quale sarà l'output stampato a video
    dall'IDLE

maggiore
>>>
```

In **Python** non esiste il **costrutto *switch***, presente in molti altri linguaggi.

* * *

Le basi - Tutorial 10

Controllo del flusso: i cicli FOR e WHILE

Le istruzioni BREAK, CONTINUE, PASS

IL COSTRUTTO DEL CICLO FOR. INDENTAZIONE DEL BLOCCO

Il costrutto **for** ci consente di effettuare dei **CICLI**, ossia di **ripetere un insieme di istruzioni** per un determinato numero di volte.

La sintassi è:

```
for [indice] in [sequenza]:  
    # istruzioni (attenzione all'indentazione!)
```

Attenzione ai **:** (due punti), alla fine della dichiarazione del **for**., necessari per aprire il blocco di istruzioni da eseguire all'interno del ciclo. Le istruzioni appartenenti al ciclo dovranno avere quindi una tabulazione (indentazione) in più rispetto alle altre istruzioni presenti nel codice, come visto ad esempio per il costrutto **if**.

LA FUNZIONE RANGE

Nella sua forma base, **for** viene creato mediante la **funzione range**, disponibile in tre versioni:

- **range(n);**

Esegue il ciclo **n-volte**. Il valore di **[indice]**, ad ogni iterazione, sarà pari al valore di **n**. Il valore iniziale di **n** sarà 0.

Esempio:

```
>>> for n in range(4):  
        print(n)           # Indentato!
```

stamperà a video il seguente risultato:

```
0  
1  
2  
3
```

- **range(inizio, fine);**

Esegue il ciclo *(fine-inizio)-volte*.

Ad ogni **iterazione**, il valore di *n* sarà pari a **(inizio + numeroIterazione)**.

Esempio:

```
>>> for n in range(3, 6):  
        print(n)           # Indentato!
```

stamperà a video il seguente risultato:

```
3  
4  
5
```

- **range(inizio, fine, passo);**

Esegue il ciclo *((fine-inizio) / passo)-volte*.

Il valore di **n** varierà da inizio a fine con incremento pari a passo (appunto...) ad ogni **iterazione**.

Esempio:

```
>>> for n in range(10, 2, -1):  
        print(n)           # Indentato!
```

stamperà a video il seguente risultato:

```
10  
9  
8  
7  
6  
5  
4  
3
```

FOR SUI TIPI DI DATI ITERABILI

È possibile comunque generare un **for** "scorrendo" un qualsiasi **dato iterabile**; in questo caso, la sintassi sarà:

```
for [indice] in [oggetto iterabile]:  
    # istruzioni (attenzione all'indentazione!)
```

Esempio:

```
>>> for lettera in "stringa":  
        print(lettera)
```

stamperà a video il seguente risultato:

```
s  
t  
r  
i  
n  
g  
a  
  
>>>
```

ZIP: SCORRERE CONTEMPORANEAMENTE PIÙ OGGETTI ITERABILI

Il costrutto **for** ci offre, inoltre, l'interessante possibilità di **scorrere contemporaneamente più oggetti iterabili**, depositando i valori correnti di ciascun oggetto ad ogni passo in un indice differente.

Per usufruire di tale opzione, dovremo utilizzare il **costrutto zip**, come mostrato nella regola della sintassi e nell'esempio:

```
for indice1, indice2, ..., indiceN in zip(iterabile1, iterabile2, ...,  
iterabileN):  
    # Corpo istruzioni, attenzione all'indentazione
```

Esempio:

```
>>> lista1 = [1, 2, 3]  
>>> lista2 = ["Ciao", "come", "stai"]  
>>> lista3 = [True, False, None]  
>>> for indice1, indice2, indice3 in zip(lista1, lista2, lista3):  
        print(indice1, indice2, indice3)
```

stamperà a video il seguente risultato:

```
1 Ciao True
2 come False
3 stai None
>>>
```

IL COSTRUTTO WHILE

Il costrutto **while**, invece, effettua le operazioni specificate fintantoché si verifica una **condizione**.

Sintassi:

```
while condizione:
    [operazioni]
```

Attenzione, quindi, al segno : (due punti) posto subito dopo la **condizione da verificare** e all'**indentazione** che identifica il corpo del metodo.

*[NOTA --- Tenere a mente che il **while**, a differenza del **for**, non raggiungerà mai la sua condizione di terminazione se non verrà specificato, in qualche modo, di incrementare un contatore o terminare in base a qualche altro evento! Spetta al programmatore occuparsi di tale aspetto!]*

Esempio:

```
>>> x = 0
>>> while x<10:
    print(x)
    x = x+1      # Istruzione di incremento, per arrivare alla
                 condizione di terminazione ("x uguale 10") indicata nel while
```

stamperà a video il seguente risultato (dopo aver premuto Invio un paio di volte):

```
0
1
2
3
4
5
```

```
6
7
8
9
>>>
```

LE ISTRUZIONI BREAK, CONTINUE, PASS

Esistono delle istruzioni particolari che permettono, comunque, di **"forzare" il flusso** del programma nei **cicli** o di "tenere occupato" il corpo di un blocco **if**, **for** o **while**, ad esempio quando ancora dobbiamo determinare per bene il contenuto di questi blocchi ma, intanto, vogliamo "tenere il posto".

- **break**

L'istruzione **break** ci consente di **terminare bruscamente un ciclo**; ad esempio, si è verificata una condizione (non quella di terminazione del ciclo, ma un'altra; ad esempio, il ciclo era un ciclo di attesa su una porta di comunicazione e, nel frattempo, è arrivato un messaggio dalla rete, ecc...).

Per terminare il ciclo, sarà sufficiente scrivere, come corpo dell'istruzione di verifica:

```
break
```

Esempio:

```
>>> c = 0
>>> for c in range(0,10):
    print(c)
    if (c*2>6):
        break
```

stamperà a video il seguente risultato:

```
0
1
2
3
4
>>>
```

- **continue**

L'istruzione *continue* indica a **Python** di passare direttamente all'**iterazione successiva di un ciclo (while o for)** senza eseguire le altre istruzioni presenti nel corpo del blocco poste dopo *continue*.

È utile, ad esempio, se durante l'esecuzione del ciclo si verifica una condizione e certe istruzioni del ciclo non devono essere eseguite, mentre altre sì, e comunque è necessario effettuare altre iterazioni); in tal caso, sarà sufficiente scrivere:

```
continue
```

- **pass**

pass non fa assolutamente nulla, ma permette di riempire il **blocco** delle istruzioni di *if*, *for* o *while*, ad esempio perché non sappiamo ancora cosa metterci (se stiamo lavorando in un team e siamo in attesa della definizione di un modulo da utilizzare nel corpo di un *while*, intanto scriviamo il *while* con il *pass*, poi lo sostituiamo) e queste istruzioni non possono essere seguite da una riga vuota o da una riga di commento, per cui possiamo utilizzare *pass* per... “passare oltre”, momentaneamente, in attesa di rimpiazzare l'istruzione; ad esempio:

```
while (condizione):  
    pass
```

* * *

Le basi - Tutorial 11

Definire ed utilizzare le funzioni

Parametri di input e valori restituiti in output

Le **funzioni** sono porzioni di codice dotate di nome, parametri in input e valori restituiti in output, richiamabili da altre parti del programma.

In Python, le funzioni vengono dichiarate specificandone il nome dopo la **parola chiave**

```
def
```

secondo la seguente **sintassi** (attenzione all'indentazione; per il momento vedremo solo la sintassi, in seguito un esempio completo):

```
>>> def nomeFunzione([0, 1 o più parametri in input, separati da  
virgole]):  
    # Corpo funzione ed eventuale commento di documentazione (vd. dopo)  
    [return [1 o più valori da restituire in output, opzionali]]
```

INVOCARE UNA FUNZIONE

Per richiamarle (“invocarle”), sarà sufficiente **scrivere il loro nome** seguito dalle **parentesi tonde** (contenenti 0, 1 o più **parametri**, a seconda dei casi), in un'altra porzione dello script:

```
>>> [1 o più valori in output, opzionali = ] nomeFunzione([eventuali  
parametri])
```

VALORI RESTITUITI (OUTPUT) DALLE FUNZIONI

Le **funzioni** possono restituire 0, 1 o più **valori in output**:

- **0 valori**, o meglio il valore **None**, scrivendo alla fine della funzione *"return"*, oppure *"return None"*, oppure *pass* o proprio niente, chiudendo il blocco della funzione eliminando l'indentazione);

- **1 valore**, restituito con *"return [valore]"*, da assegnare ad una variabile, ad esempio con:

```
>>> a = miaFunzione(); # Assegna ad "a" il valore di output della  
funzione "miaFunzione"
```

- **2 o più valori**, restituiti elencandoli dopo *return* e separandoli con delle virgole (es.: *return 'ciao', 2*), da assegnare a 2 o più variabili separate da delle virgole:

```
>>> var1, var2 = miaFunzioneCon2output();
```

COMMENTO DI DOCUMENTAZIONE PER UNA FUNZIONE

È possibile specificare un **"commento di documentazione"** in una funzione racchiudendo del testo all'interno di una coppia di delimitatori *"""* (tre doppi apici).

Potremo poi accedere a tale **documentazione** mediante il **campo** `__doc__` delle **funzioni**, per cui sarà sufficiente scrivere:

```
[nome funzione].__doc__
```

(i segni **underscore** `_` sono **due** prima di *doc* e due dopo *doc*).

ESEMPIO 1

Nella prima parte di questo esempio definiamo una semplice funzione di somma, creando anche un commento di documentazione.

La funzione prende in input un parametro (*n*) ed effettua la somma di tutti i numeri da 0 a *n* mediante un ciclo *for*, depositando il tutto nella variabile *accumulatore*, che verrà poi restituita (output) tramite *return* al termine della funzione:

```
>>> def somma(n):  
    """  
    Stringa di documentazione. Effettua la somma dei primi (n-1) numeri  
    e la restituisce.  
    """  
    accumulatore = 0  
    for a in range(0, n):  
        accumulatore = accumulatore + a  
    return accumulatore  
  
>>> # Due pressioni del tasto Invio (o una riga vuota, in un file .py) e  
siamo fuori dal blocco della funzione (attenzione, come sempre,  
all'indentazione)
```

Nella seconda parte dello script, invochiamo la funzione (da un altro punto dello script Python, quindi, al di fuori della definizione della funzione), assegnando il valore restituito dalla funzione alla variabile “*primi9numeri*” e passando come parametro (input) alla funzione, durante la sua invocazione, il valore 9:

```
>>> primi9numeri = somma(10)  
>>> print(primi9numeri)  
45  
>>> print(somma.__doc__)      # Stampiamo la documentazione  
Stringa di documentazione. Effettua la somma dei primi (n-1) numeri e la  
restituisce.  
>>>
```

ESEMPIO 2

Vediamo ora un semplice esempio di funzione che non prende parametri in input (nessun argomento tra le parentesi tonde, né in fase di definizione né in fase di invocazione) ma che, quando viene invocata, restituisce sempre tre valori (in particolare, in questo caso, tre stringhe).

Definiamo la funzione:

```
>>> def outputMultipli():  
    return "Ciao", "come", "stai"  
  
>>>
```

Invochiamo ora la funzione, assegnandola a tre variabili (una per ogni valore restituito dalla stessa), separate da virgole a sinistra del simbolo “=” (l'operatore di assegnazione):

```
>>> out1, out2, out3 = outputMultipli()
>>> print(out1)
Ciao
>>> print(out1, out2, out3)
Ciao come stai
>>>
```

L'argomento delle funzioni in Python è molto vasto e le possibilità messe a disposizione da tale strumento non possono essere mostrate in un corso di base come questo; per una trattazione più approfondita, si rimanda alla documentazione ufficiale di Python.

* * *

Le basi - Tutorial 12

Gestione delle eccezioni: try, except, finally, raise

Le **eccezioni** sono **eventi "anomali"** (o che producono risultati tali, come la **divisione per zero**) che possono portare, se non opportunamente previste e gestite, alla **terminazione dell'intero programma** --- cosa che avviene, comunque, solo se nessuno dei blocchi dell'**esecuzione** che racchiudono quello ove si è verificata l'**eccezione** consente di catturare e gestire quest'ultima; in tal caso, l'**eccezione** si propagherà fino al blocco più esterno, che poi equivale all'intero programma, chiudendolo, ossia **terminandone l'esecuzione**.

CATTURARE E GESTIRE LE ECCEZIONI: TRY - EXCEPT

Le parti di **codice "sensibile"** (come le operazioni di I/O sul file system, le operazioni in rete, particolari operazioni matematiche o sulle liste, ecc...) dovrebbero essere sempre inserite in un particolare **blocco**, detto **try**; le **eccezioni** (eventualmente) lanciate dalle istruzioni presenti in tale blocco potranno quindi essere **gestite** da altri particolari blocchi (uno o più), detti blocchi **except**, associati al **try**.

Il meccanismo di **cattura e gestione delle eccezioni in Python** è detto quindi **try-except**, ed è molto simile a quello di altri linguaggi di programmazione (come il **try-catch** di **Java**, ad esempio).

La forma base della **gestione delle eccezioni in Python** è quindi la seguente (attenzione ai due punti e all'indentazione):

```
try:
    # Porzione di codice da tenere d'occhio, contenente istruzioni
    "sensibili"
except [eccezione1]:
    # Operazioni da effettuare se è stata lanciata un'eccezione di tipo
    eccezione1
except [eccezione2]:
    # Operazioni da effettuare se è stata lanciata un'eccezione di tipo
    eccezione2
# ... tutti gli except "specifici" che vogliamo, per gestire vari tipi di
    eccezioni ...
except[eccezioneN]:
    # Operazioni da effettuare se è stata lanciata un'eccezione di tipo
    eccezioneN
except:
    # Operazioni da eseguire se è stata lanciata un'eccezione gestibile
    con i blocchi except "specifici" precedenti
```

ESEMPIO DI ECCEZIONE E SUA GESTIONE

La **divisione per 0** genera un errore:

```
>>> a = 5
>>> b = a/0
Traceback (most recent call last):
  File "<pyshell#213>", line 1, in <module>
    b = a/0
ZeroDivisionError: division by zero
>>>
```

per cui, per **catturarla e gestirla** potremo scrivere qualcosa come:

```
>>> a = 5
>>> try:
    b = a/0
except:
    print("Divisione per zero! Tranquillo, non terminerò il
    programma: eccezione catturata e gestita!")
```

o, ancora meglio (visto che l'**eccezione di divisione per zero** è un'eccezione "nota" ed ha una sua **exception** "dedicata"):

```
>>> a = 5
>>> try:
    b = a/0
except ZeroDivisionError:
    print("Divisione per zero! Tranquillo, non terminerò il
    programma: eccezione catturata e gestita!")
```

Esistono quindi molte **eccezioni** "note" e catturabili in maniera specifica, per essere **riconosciute e gestite** opportunamente; per un elenco completo, si rimanda alla documentazione ufficiale di Python.

BLOCCHI ELSE E FINALLY PER TRY-EXCEPT

Al blocco **try-except** è possibile poi aggiungere i **blocchi else** (eccezione non lanciata: effettua le operazioni presenti nel blocco **else** e vai avanti) e/o **finally** (ossia: esegui COMUNQUE le istruzioni presenti in tale **blocco**, sia che sia stata lanciata un'eccezione gestita da un blocco **except**, sia che non sia stata lanciata alcuna eccezione), per cui il blocco diventerà (attenzione, come sempre, ai due punti e all'indentazione!):

```
try:
    # Porzione di codice da tenere d'occhio, contenente istruzioni
    "sensibili"
except [eccezione1]:
    # Operazioni da effettuare se è stata lanciata un'eccezione di tipo
    eccezione1
except [eccezione2]:
    # Operazioni da effettuare se è stata lanciata un'eccezione di tipo
    eccezione2
# ... tutti gli except "specifici" che vogliamo, per gestire vari tipi di
    eccezioni ...
except[eccezioneN]:
    # Operazioni da effettuare se è stata lanciata un'eccezione di tipo
    eccezioneN
except:
    # Operazioni da eseguire se è stata lanciata un'eccezione gestibile
    con i blocchi except "specifici" precedenti
else:
    # Operazioni da eseguire solo se non sono state lanciate eccezioni
    dalle istruzioni presenti nel blocco try
finally:
    # Operazioni da eseguire COMUNQUE, sia che siano state lanciate (e
    gestite) eccezioni che non.
```

RAISE: LANCIARE VOLONTARIAMENTE UN'ECCEZIONE

In **Python** è possibile, inoltre, lanciare volontariamente un'eccezione, ad esempio per **gestire un evento particolare** o per passare il controllo ad un blocco più esterno (cosa che avviene, in particolare, lanciando un'eccezione senza parametri).

Per lanciare un'eccezione, utilizzare l'istruzione ***raise***:

```
raise [eventuale nome dell'eccezione [, eventuale messaggio da mostrare a video]]
```

Esempi:

```
>>> raise
Traceback (most recent call last):
File "<pyshell#288>", line 1, in <module>
raise
RuntimeError: No active exception to reraise
```

```
>>> raise ZeroDivisionError
Traceback (most recent call last):
File "<pyshell#219>", line 1, in <module>
raise ZeroDivisionError
ZeroDivisionError
```

* * *

Le basi - Tutorial 13

Classi, oggetti, campi e metodi

Metodi privati. Overloading degli operatori

Le **classi** sono le “blueprint” (i progetti, gli schemi) di un oggetto, mentre un **oggetto** è una particolare manifestazione di una classe; ad esempio, la variabile “a” con valore 5 è un oggetto della classe dei numeri interi.

Sono ad esempio classi native di **Python**: gli interi, i float, le stringhe e i booleani.

In questo tutorial vedremo quindi **come definire classi** (tipi) di oggetti personalizzati e come realizzare (“**istanziare**”) degli oggetti.

LA PAROLA CHIAVE CLASS

Le **classi** di **oggetti**, in **Python**, vengono definite utilizzando la **parola chiave** *class*, seguita dal nome della classe e dal simbolo : (due punti).

A partire dalla riga successiva, nel blocco identificato come appartenente alla **classe** mediante **l'indentazione**, andranno poi definiti **campi** e **metodi**.

Esempio:

```
class figureRegolari:
    # corpo della classe
    [pass]          # Opzionale (ecco il perché delle [], da non scrivere)
```

OGGETTI: ISTANZE DI UNA CLASSE

Si crea un oggetto concreto, ovvero una **istanza** di una **classe**, mediante la scrittura:

```
variabileIstanza = nomeClasse([eventuali parametri al costruttore])
```

ad esempio:

```
>>> quadrato1 = figureRegolari()
```

L'OGGETTO SELF

All'interno della **classe**, ci si riferirà all'**oggetto istanziato** ("se stesso"; ad esempio, "*this*" in Java) mediante la **parola chiave** *self*, che poi dovrà anche essere il primo **parametro** di tutti i **metodi** della **classe** (**costruttore incluso**).

IL COSTRUTTORE

Il **metodo principale** di una **classe** è il **costruttore**, che ha sempre firma "interna":

```
__init__(self, [eventuali parametri])
```

(due tratti di underscore *_* prima e dopo la parola chiave *init*).

Il **parametro** *self* va messo, come detto, sempre come primo parametro dei metodi di una classe (costruttore incluso) nella loro definizione, **ma non va passato come parametro** quando tali metodi vengono invocati.

self va utilizzato anche per definire gli **attributi (campi)** della **classe**.

Esempio:

```
>>> class quadrato:
    lato = 0
    def __init__(self, l):
        self.lato = l
    def getPerimetro(self):
        return (4*self.lato)
    def getArea(self):
```

```
        return (self.lato * self.lato)

>>>
>>> quadrato1 = quadrato(4)
>>> print(quadrato1.lato)
4
>>> areaQuadrato1 = quadrato1.getArea()
>>> print(areaQuadrato1)
16
>>>
>>> print(quadrato1.getPerimetro())
16
>>>
```

METODI PRIVATI (FUNZIONI PRIVATE)

È possibile definire, all'interno delle **classi**, dei **metodi privati**, che potranno essere invocati solo da altri metodi della stessa classe, non "dall'esterno".

Per creare un **metodo privato**, è sufficiente far precedere il suo nome da 2 tratti di underscore (NON bisogna porre però due tratti di underscore alla fine del nome, vedremo in seguito perché); ad esempio:

```
def __metodo1(self, [altri parametri opzionali])
```

OVERLOADING DEGLI OPERATORI

Python, come molti altri linguaggi di programmazione, consente l'**overloading degli operatori**, ossia ci consente di “riadattare” alle nostre esigenze, per le nostre classi, alcuni **elementi della sintassi di Python**, come ad esempio gli **operatori relazionali**.

Ad esempio, $3 < 5$ ha un senso nella sintassi di base di **Python** perché i due valori sono degli **interi**, ma possiamo ridefinire nelle nostre **classi** il significato di $<$, in modo da utilizzarlo per confrontare secondo un nostro criterio due **oggetti** (per esempio, *persona1* $<$ *persona2* potrebbe indicare un confronto in base all'età, per cui se *persona1* dovesse avere un campo numerico *età* con valore inferiore a quello di *persona2*, il controllo "*persona1* $<$ *persona2*" sarà *True*; a breve mostreremo proprio il codice di questo esempio).

I **metodi "speciali"**, per i quali è consentito l'**overloading**, vengono indicati con due tratti di underscore **prima e dopo** la loro sigla (ed ecco perché per i metodi privati i due tratti di underscore vanno messi solo prima del nome, non anche dopo); sono **metodi speciali "di base"**, ad esempio, i seguenti:

__len__	"Length", lunghezza di qualcosa (una stringa) o numero di elementi in una sequenza (liste, ...).
__str__	"To string", una versione in stringa (informativa) dell'oggetto (vd. esempio in seguito)
__gt__	"Greater than", cioè maggiore di (>)
__lt__	"Less than", cioè minore di (<)
__eq__	"Equal", cioè uguale a (==)
__le__	"Less equal", cioè minore uguale a (<=)
__ge__	"Greater equal", cioè maggiore uguale a (>=)

Nel caso di **operatori**, come quelli **relazionali**, ci si riferirà all'**istanza** "di sinistra" con **self** e all'**istanza** "di destra" con **other**; a breve, come anticipato, un esempio completo.

Esistono, ovviamente, molti altri **metodi speciali**, ma per una trattazione più approfondita si rimanda alla documentazione ufficiale di Python.

CLASSI E OVERLOADING DEGLI OPERATORI: ESEMPIO

Esempio pratico di creazione di una classe, istanziazione di due oggetti e utilizzo dell'overloading dell'operatore > su tali elementi:

```
>>> class persona:
    nome = ""
    cognome = ""
    eta = 0
    def __init__(self, n, c, e):
        self.nome = n
        self.cognome = c
        self.eta = e
    def __str__(self):
        print("Persona con cognome: " + self.cognome + ", nome: " +
              self.nome + ", età: " + str(self.eta))
    def __lt__(self, other):
        return (self.eta < other.eta)

>>>
>>> miaPersonal = persona("Mario", "Rossi", 50)
>>> miaPersonal.__str__()
Persona con cognome: Rossi, nome: Mario, età: 50
>>> miaPersona2 = persona("Giorgio", "Bianchi", 40)
>>> miaPersonal < miaPersona2
False
>>>
```

* * *

Le basi - Tutorial 14

Ereditarietà

Sfruttando il meccanismo dell'**ereditarietà** (un concetto cardine della **programmazione a oggetti**), è possibile definire classi “*figlie*” che ereditano campi e attributi degli “*antenati*”, in modo da averli già pronti, ma potranno anche sovrascriverli (ovvero **ri-definirli**, ad esempio impostando un valore fisso per un campo o modificare il corpo di un metodo definito in un suo antenato) o presentare **nuovi campi e metodi**, non messi a disposizione dagli antenati.

L'esempio tipico, per comprendere questo concetto, è quello dei **poligoni**: una classe “*poligonoRegolare*” potrebbe presentare un metodo “*calcolaArea*” e un metodo “*calcolaPerimetro*”; la classe figlia “*esagono*” potrà quindi sovrascrivere i due metodi e aggiungere, ad esempio, “*calcolaApotema*”.

CREARE CLASSI FIGLIE IN PYTHON

Implementare l'**ereditarietà delle classi** in **Python** è semplicissimo: nella definizione della **classe** figlia, sarà sufficiente passare come “**parametro**” il **nome della classe genitore**, come segue:

```
class figlia(nomeClassePadre):  
    # Corpo della classe figlia, che eredita campi e metodi della classe  
    padre e degli altri (eventuali) antenati, a catena  
    [pass]
```

ISTANZIARE CLASSI FIGLIE IN PYTHON

Per creare **un'istanza** di una **classe** che **eredita** da un'altra **classe** basterà utilizzare la solita **sintassi**:

```
variabileIstanza = nomeClasse([eventualiParametri])
```

ESEMPIO

Ecco un esempio completo sull'ereditarietà in Python:

```
>>> class rettangolo:
    lato1 = 0
    lato2 = 0
    def __init__(self, l1, l2):
        self.lato1 = l1
        self.lato2 = l2
    def calcolaPerimetro(self):
        return ((self.lato1*2) + (self.lato2*2))
    def calcolaArea(self):
        return (self.lato1 * self.lato2)

>>> class quadrato(rettangolo):
    def __init__(self, l):
        self.lato1 = l
        self.lato2 = l

>>> mioQuadrato = quadrato(5)
>>> print(mioQuadrato.calcolaPerimetro())
20
>>> print(mioQuadrato.calcolaArea())
25
>>>
```

* * *

Le basi - Tutorial 15

Lettura e scrittura con i file

METODO FILE(): L'HANDLER (GESTORE) DEI FILE

Per associare un **file** ad una **variabile** (handler, gestore), e quindi effettuare **operazioni di I/O su file**, **Python** mette a disposizione il **metodo *open***:

```
f = open("[path]", "[mode]");
```

che, assegnato ad una **variabile**, permette appunto di **caricare il file** con percorso **"path"**.

APERTURA DEL FILE: MODE

Le **modalità di apertura del file** vanno specificate mediante il **parametro *mode***, una stringa che può assumere i seguenti valori:

- **"a"** --- *append*, modalità di scrittura in coda al file (non sovrascriverà dati esistenti); se un file col nome specificato nel path non esiste lo crea, altrimenti scrive in coda;
- **"w"** --- *write*, modalità di scrittura; se un file col nome specificato nel path non esiste lo crea, altrimenti lo sovrascrive del tutto;
- **"r"** --- *read*, modalità di lettura; funziona solo se il file col nome specificato come path esiste, altrimenti restituisce errore (da gestire)

CHIUDERE IL FILE: CLOSE

Per rilasciare il **file**, poi, sarà sufficiente invocare il **metodo *close***).

SCRIVERE SU FILE

Una volta aperto un **file** in **scrittura**, è possibile **scrivere** all'interno dello stesso con il **metodo** `write("[stringa]")`;

A breve un esempio completo.

LETTURA DA FILE

Per **leggere un certo numero di bytes** da un **file** aperto in **lettura**, è possibile utilizzare il **metodo**:

- `read([intero]);`

legge [intero] bytes dal file aperto, o tutto il file se sono presenti meno di [intero] bytes nello stesso.

mentre è possibile **leggere tutte le righe del file** (per i file di tipo ASCII: file di testo, non binario) scorrendole con un **ciclo for** e un indice, mediante il **metodo** `readlines`:

```
f = open("prova.txt", "r")    # Qui il file "prova.txt" si trova nella
                              # stessa cartella dello script Python; per percorsi diversi, indicare il
                              # percorso assoluto (dalla root) o, in caso di sottocartelle, relativo
for indice in f.readlines():
    # ad ogni iterazione, "indice" conterrà una riga del file, per cui
    # provate a scrivere ad esempio print indice...
```

Esistono, ovviamente, molti altri **metodi** riguardanti i **file**, ma per una trattazione completa si rimanda alla documentazione ufficiale di Python.

ESEMPIO COMPLETO

Il codice riportato nella pagina seguente crea un file di nome *"Prova.txt"* nel percorso (path) C:\ (attenzione! Cambiate il valore della variabile se avete già un file con tale nome in C:\, altrimenti questo verrà sovrascritto!) ed esegue varie operazioni sul file, aprendolo in varie modalità.

```
>>> variabile = open("prova.txt", "w")
```

```
>>> variabile.write("Ciao, come stai?\nBene grazie, tu?")
>>> variabile.close()
>>> variabile2 = open("prova.txt", "r")
>>> for riga in variabile2.readlines():
    print(riga)
Ciao, come stai?
Bene grazie, tu?

>>> variabile2.close()
>>> variabile = open("prova.txt", "a")
>>> variabile.write("\n\nAndiamo a prendere un caffè?")
>>> variabile.close()
>>> variabile = open("prova.txt", "r")
>>> for riga in variabile.readlines():
    print(riga)

Ciao, come stai?
Bene grazie, tu?
Anch'io, grazie!
Andiamo a prendere un caffè?

>>> variabile.close()
>>>
```

```
*      *      *
*      *      *
```

Tkinter - Tutorial 1

Introduzione. Creazione di una finestra

Dimensioni, coordinate e titolo della finestra

Questo è il primo di una serie di tredici tutorial dedicati alla **creazione di interfacce grafiche in Python** utilizzando le librerie *Tkint*.

ESEGUIRE GLI SCRIPT SALVATI SU FILE .PY

Nel corso di questi tutorial definiremo degli script che, a differenza di quelli trattati nella prima parte di questo pdf (e relativi alle basi di **Python**), non andranno scritti riga per riga nell'IDLE, ma **in file .py a parte**, per cui poi dovremo **eseguire tali file di script**, ad esempio invocandoli dall'IDLE o, da riga di comando, con “*python [nomescript].py*”.

Esistono vari modi e strumenti per definire ed eseguire dei file *.py*; ad esempio, potete procedere in questo modo:

1. avviate l'IDLE di **Python** e scegliete “**New File**” dal menù **File**;
2. nella nuova schermata che apparirà, definite gli script (ad esempio, copiando e incollando il codice di questi tutorial), **SENZA** il simbolo `>>>` (che rappresenta il prompt dei comandi dell'IDLE, per digitare ed eseguire interattivamente i comandi; questo è, invece, un file di script);
3. salvate lo script cliccando su “**Save**” dal menù **File**, specificando in particolare l'estensione *.py* per il file;
4. eseguite lo script scegliendo, sempre nella finestra del file appena salvato, “**Run – Run**

Module” (shortcut **F5**); l'output verrà visualizzato nella schermata dell'IDLE, aperta all'inizio, prima di creare il file dello script (al punto 1).

TKINTER

Non sarà necessario installare nulla dato che le **librerie Tkinter** sono già incluse nella versione di **Python** che abbiamo scaricato (la 3.4, al momento della stesura di questa mini-guida).

Possiamo trovare tutte le informazioni su **Tkinter** dal sito Internet ufficiale, <https://wiki.python.org/moin/TkInter>, organizzato con la struttura *Wiki*. Troviamo, ad esempio, una introduzione a **Tkinter**, e soprattutto, tutte le **reference**, sia all'interno del sito che come **pdf** liberamente scaricabile.

Poiché questi tutorial non possono essere esaustivi di tutte le potenzialità di **Tkinter**, consigliamo di far riferimento a tale documentazione per gli **approfondimenti**.

Sempre sul sito troviamo una raccomandazione fondamentale per quanto riguarda **l'importazione delle librerie**; infatti, fino alla versione 3.0 di Python dovremo scrivere *from tkinter* con la *T* maiuscola, mentre dalla 3.0 in poi dovremo scriverlo con la *t* minuscola.

Iniziamo dunque proprio dal comando **import**.

[NOTA: da qui in poi, per i prossimi capitoli, non lavoreremo più nell'IDLE (con istruzioni interattive), ma definendo dei file .py per ogni tutorial ed eseguendo tali file, ad esempio dall'IDLE (come visto nella sezione precedente) o dal prompt di sistema, con python nomefile.py, o con altre tecniche.]

Poiché sto usando la versione 3.4 di Python, scriverò *from tkinter* con la *t* minuscola seguito da **import** e **asterisco** (l'asterisco sta per “all”, “tutti”, per importare cioè tutti i sottomoduli):

```
from tkinter import *
```

CREARE UNA FINESTRA. DIMENSIONI E COORDINATE

Il comando per **generare una finestra** è un semplicissimo: `nomefinestra = Tk()`

```
finestra = Tk()
```

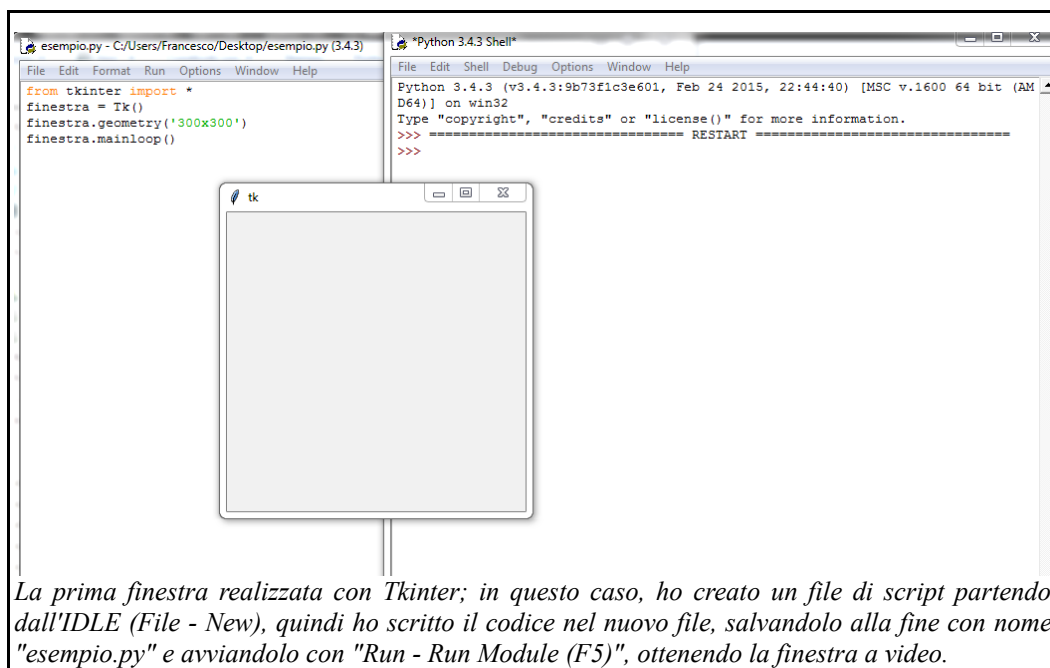
Per definire la **grandezza della finestra** utilizziamo il metodo ***geometry*** relativamente alla finestra che abbiamo chiamato, appunto, *"finestra"*. Basterà passare come argomento una stringa con la larghezza, la x e l'altezza:

```
finestra.geometry('300x300')
```

Infine, scriviamo `finestra.mainloop()` per far partire effettivamente la finestra.

```
finestra.mainloop()
```

Adesso possiamo salvare il file dello script e avviare il programma, invocando il file dall'IDLE o eseguendolo da riga di comando, come detto precedentemente.



La prima finestra realizzata con Tkinter; in questo caso, ho creato un file di script partendo dall'IDLE (File - New), quindi ho scritto il codice nel nuovo file, salvandolo alla fine con nome "esempio.py" e avviandolo con "Run - Run Module (F5)", ottenendo la finestra a video.

Queste quattro, semplici righe di codice ci hanno permesso di creare la prima finestra.

Possiamo modificare la stringa di ***geometry*** per aggiungere la posizione, in pixel, in cui verrà visualizzata la finestra appena creata; lo facciamo aggiungendo +, la coordinata X (orizzontale, in pixel) ove vogliamo posizionare la finestra, + e la coordinata Y (verticale, anche questa in pixel).

La finestra verrà quindi creata in posizione (1000, 300):

```
finestra.geometry('300x300+1000+300')
```

Facciamo un'ulteriore prova creando la finestra più in basso e più a destra rispetto a prima...

```
finestra.geometry('300x300+1100+500')
```

DEFINIRE IL TITOLO DELLA FINESTRA

Riportiamo tutto come prima e aggiungiamo l'ultimo elemento, ovvero il titolo della finestra.

Anche qui il comando è molto semplice: *finestra.title* e il titolo passato come stringa, per cui scriviamo ad esempio

```
finestra.title('La mia prima finestra con Tkinter')
```

Questo primo tutorial probabilmente rende l'idea di quanto sia facile creare interfacce grafiche con **Python e Tkinter**.

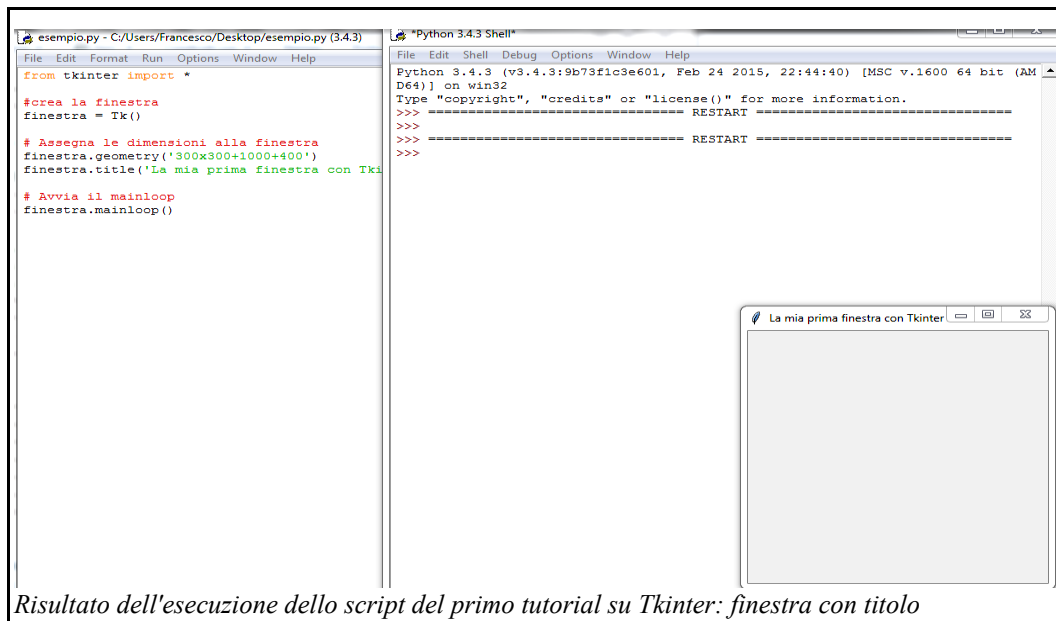
Qui di seguito, il **codice completo** dello script trattato in questo tutorial; il risultato dell'esecuzione è visibile nella pagina seguente.

```
from tkinter import *

#crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

# Avvia il mainloop
finestra.mainloop()
```



* * *

Tkinter - Tutorial 2

Elementi di interfaccia: Label (etichette di testo) e Button (pulsanti)

In questo secondo tutorial del mini-corso su **Tkinter** impareremo ad aggiungere i primi elementi alla nostra finestra.

CREARE UNA LABEL (ETICHETTA, CAMPO TESTUALE)

Riscriviamo lo script definito nel tutorial precedente aggiungendo un'istruzione che ci consente di creare il primo elemento: un testo che creeremo istanziando un widget Label (lett.: “etichetta”; un testo non modificabile dall'utente).

```
from tkinter import *

#crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

# Aggiungiamo un testo
testo = Label(text="Il mio primo testo")

# Avvia il mainloop
finestra.mainloop()
```

Il tutto sta quindi nel definire una variabile (per “gestire”, in seguito, l'etichetta) assegnandole la funzione nativa Label, avente come attributo il testo da visualizzare:

```
# Aggiungiamo un testo
testo = Label(text="Il mio primo testo")
```

Tuttavia, facendo partire il programma... **non verrà visualizzato nulla**.

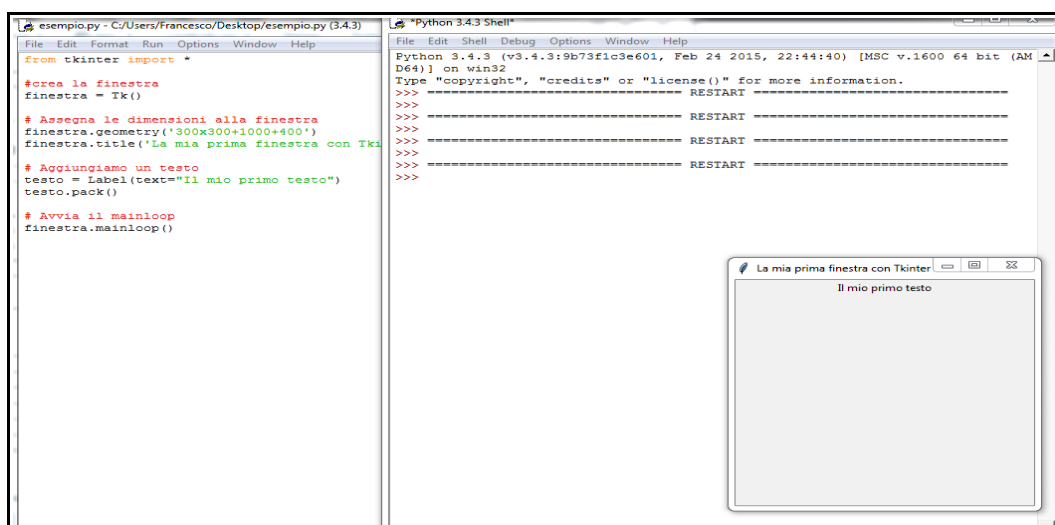
Questo perché **Python** non sa **dove posizionare** esattamente l'etichetta.

[NOTA: in caso di errore “Invalid Character” per i doppi apici, copiati e incollati da questo PDF, provate a sostituire quei caratteri con apici singoli o con doppi apici di un altro font; Python non riconosce alcuni simboli (come i doppi apici o il trattino – del font Courier) e restituisce errore; in quei casi, bisogna sostituirli.]

POSIZIONAMENTO: PACK()

Dedicheremo il prossimo tutorial al posizionamento; per il momento, utilizziamo il metodo **pack()**, che posizionerà il testo **dentro il suo contenitore, in posizione centrale**:

```
# Aggiungiamo un testo
testo = Label(text="Il mio primo testo")
testo.pack()
```



Possiamo risparmiare una linea di codice aggiungendo **pack** direttamente in coda al momento della costruzione, ottenendo lo stesso risultato:

```
# Aggiungiamo un testo
testo = Label(text="Il mio primo testo").pack()
```

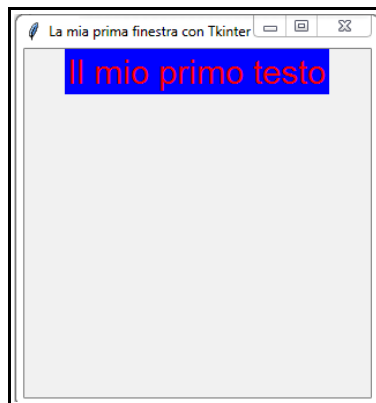
Da notare che **Python** ha identificato da solo la finestra come “contenitore” del testo, infatti ci siamo permessi il lusso di omettere il primo parametro della funzione nativa *Label*, ovvero il contenitore del nostro oggetto, senza ottenere errori:

```
# Aggiungiamo un testo
testo = Label(finestra, text="Il mio primo testo").pack()
```

PERSONALIZZARE LA LABEL

Possiamo specificare altre **caratteristiche** del nostro testo e già dai nomi è molto facile capire cosa andranno a modificare, ad esempio **fg** sta per *foreground* e **bg** per *background*, mentre **font** permette di cambiare tipo e grandezza del carattere (per l'elenco completo degli attributi di *Label* si rimanda alla documentazione ufficiale).

```
# Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica", 22)).pack()
```



CREARE UN BUTTON (PULSANTE)

Passiamo adesso all'inserimento di un altro elemento: il **pulsante** (*Button*).

La **sintassi** è molto simile a quella delle etichette **Label**:

```
bottone = Button(text="Un bottone inutile").pack()
```

Aggiungendo *pack()* alla fine, metteremo in colonna il bottone rispetto al testo...



... e, tuttavia, otterremo un bottone inutile, poiché non gli abbiamo ancora **associato una funzione**.

Per associare una funzione dobbiamo specificare l'attributo *command* = *nome della funzione*, senza parentesi. Questa sarà la funzione invocata da Python quando l'utente farà click sul Button.

```
bottone = Button(text = "Un bottone inutile",  
command=funzioneDelBottone).pack()
```

Definiamo allora la funzione secondo la sintassi di **Python** (attenzione ad indentare bene il codice). Diciamo alla funzione di creare un nuovo testo e di metterlo al di sotto del pulsante.

```
def funzioneDelBottone():  
    testo_due = Label(text="Un secondo testo!", fg="red",  
font=("Helvetica", 22)).pack()
```

Eseguendo lo script così creato (il cui codice completo è riportato alla fine di questo tutorial), al click sul pulsante verrà visualizzato a video il testo: **Python** intercetta quindi il **click dell'utente** e invoca la funzione “*funzioneDelBottone*”, associata al **Button**, eseguendola.



Nel prossimo tutorial vedremo **come allineare bene gli elementi** che abbiamo imparato a creare.

Qui di seguito, il codice dello script definito in questo tutorial (il risultato visivo, dopo il click del mouse sul bottone, è identico allo screenshot pubblicato nella pagina precedente, in basso).

```
from tkinter import *
def funzioneDelBottone():
    testo_due = Label(text="Un secondo testo!", fg="red",
        font=("Helvetica",22)).pack()

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red",
    font=("Helvetica",12)).pack()
bottone = Button(text="Un bottone inutile").pack()

#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter - Tutorial 3

Posizionare gli elementi dell'interfaccia. Layout GRID

In questo tutorial parleremo del **posizionamento degli elementi**, per cui ho modificato il nostro ultimo programma (visto al termine del tutorial precedente) aggiungendo altri due **Label**, “*testo_due*” e “*testo_tre*”, in modo da fare diversi esempi di posizionamento di oggetti all'interno di una finestra.

```
from tkinter import *

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
#testo_zero = Label(text='Questo testo va in alto', fg="red", bg="black",
font=("Helvetica",12)).pack()

testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack()

testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack()

testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",
font=("Helvetica",12)).pack()

#bottone = Button(text="Un bottone inutile").pack()

#Avvia il mainloop
finestra.mainloop()
```



Tutti i label hanno come dimensione del font 12.

Il label “*testo_tre*” ha uno sfondo bianco e “*testo_due*” uno sfondo verde.

Come sempre, **Tkinter** ha posizionato i nostri elementi in colonna, al centro del loro contenitore che, ricordiamo, è la finestra.

RIEMPIRE LO SPAZIO: FILL

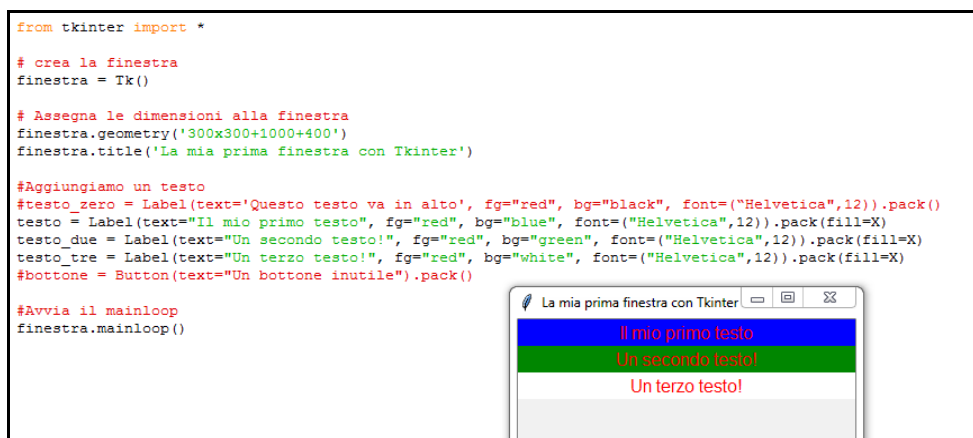
Modifichiamo il metodo *pack* aggiungendo tra parentesi il parametro *fill=X*.

```
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack(fill=X)

testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack(fill=X)

testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",
font=("Helvetica",12)).pack(fill=X)
```

Intuitivamente, il comando *fill* (riempi) dirà al label di **occupare tutto lo spazio orizzontale disponibile...** e in effetti è proprio così.



Possiamo dire ai nostri label di dividersi anche lo spazio verticale, per cui impostiamo il **fill** a **BOTH** (“entrambi”) e l'attributo **expand** a 1.

```
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack(fill=BOTH, expand=1)

testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack(fill=BOTH, expand=1)

testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",
font=("Helvetica",12)).pack(fill=BOTH, expand=1)
```

Anche **ridimensionando la finestra**, i tre label divideranno l'altezza per tre. Lo spazio orizzontale, invece, sarà occupato per intero da ciascun label.



L'ATTRIBUTO SIDE

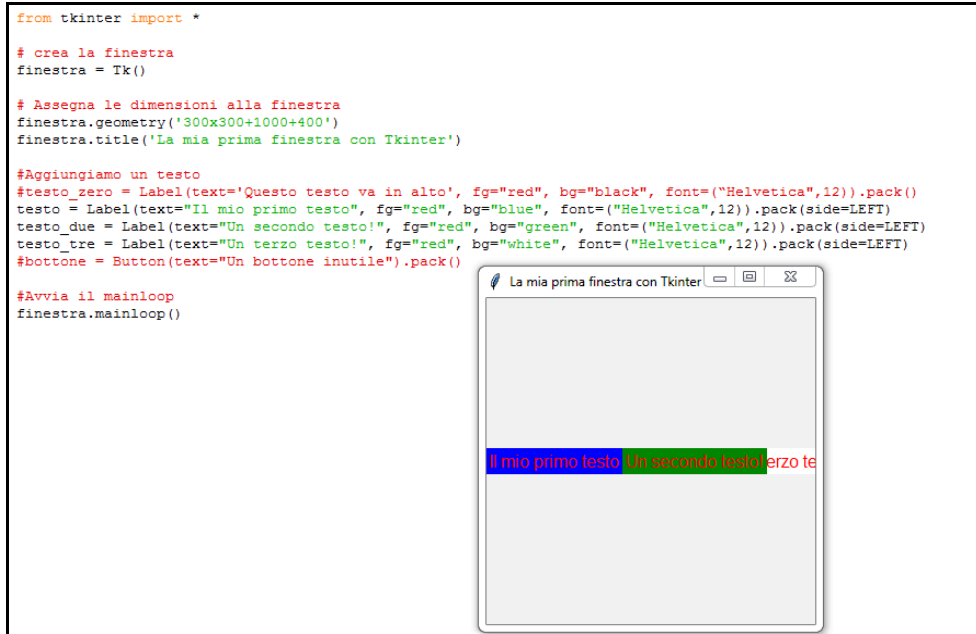
Possiamo decidere di **affiancare i label anziché averli in colonna**.

L'attributo che ci viene in aiuto, in questo caso, è **SIDE** che impostiamo a **left** per tutti:

```
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack(side=LEFT)

testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack(side=LEFT)

testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",
font=("Helvetica",12)).pack(side=LEFT)
```



Impostando il **SIDE** dell'ultimo *label* a **Right**, vincoleremo quest'ultimo al **lato destro** della finestra, come si può vedere allargando quest'ultima.

Ora aggiungerò il seguente testo (oggetto “*testo_zero*”) per mostrare un altro valore che possiamo dare a **SIDE**, ovvero **TOP**. Neanche a dirlo, posizionerà il testo in cima alla finestra.

```
testo_zero = Label(text='Questo testo va in alto', fg="red", bg="black", font=("Helvetica",12)).pack(side=TOP)
```



Di default, il **SIDE** dei livelli è **TOP**, infatti impostandoli tutti a questo valore torneremo alla situazione di partenza.

UN LAYOUT ORDINATO: IL METODO GRID. RIGHE E COLONNE

Passiamo adesso a un altro metodo che può aiutarci a creare un layout ordinato, ovvero **GRID**.

Per presentare **grid** ho aggiunto **altri due testi** con uno sfondo rispettivamente arancione e ciano.

Nella porzione di codice che segue ho già sostituito **pack** con **grid** alla fine di tutti i **label**.

```
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).grid()

testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).grid()

testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",
font=("Helvetica",12)).grid()

testo_quattro = Label(text="Un quarto testo!", fg="red", bg="- orange",
font=("Helvetica",12)).grid()

testo_cinque = Label(text="Un quinto testo!", fg="red", bg="cyan",
font=("Helvetica",12)).grid()
```

GRID permette di assegnare gli oggetti a una griglia indicando il numero di riga, **row**, e colonna, **column**:

```
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).grid(row=0, column=0)
```

Mettiamo l'oggetto **“testo_due”** sulla stessa riga di testo, ma sulla colonna accanto, mentre **“testo_tre”** sarà messo nella colonna successiva.

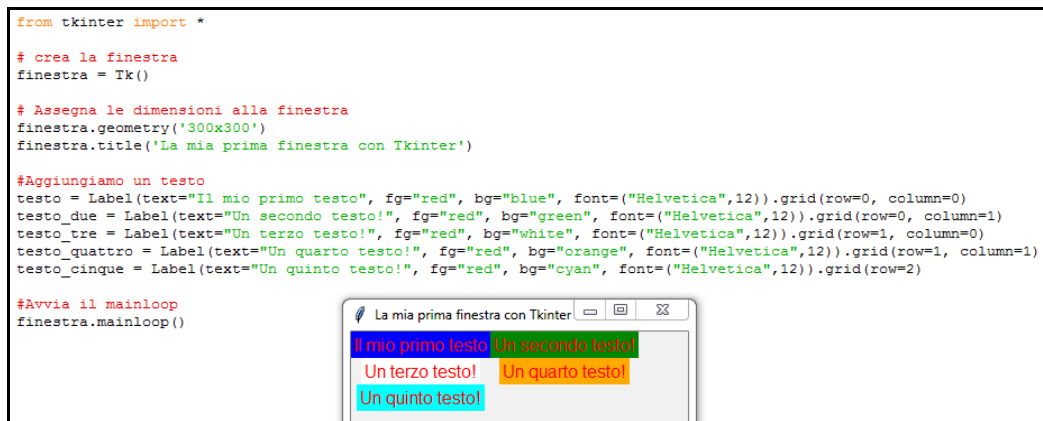
```
testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).grid(row=0, column=1)

testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",
font=("Helvetica",12)).grid(row=1, column=0)

testo_quattro = Label(text="Un quarto testo!", fg="red", bg="orange",
font=("Helvetica",12)).grid(row=1, column=1)

testo_cinque = Label(text="Un quinto testo!", fg="red", bg="cyan",
font=("Helvetica",12)).grid(row=2, column=0)
```

Vediamo come visualizza tutto **Tkinter** (screenshot del risultato visibile nella pagina seguente).



ALLINEARE E CENTRARE GLI OGGETTI

I testi sono posizionati correttamente, ma sono tutti **centrati rispetto alla propria colonna**.

Osserviamo inoltre che **la larghezza della colonna** è data dalla larghezza dell'elemento più lungo.

Per **allineare** i testi a destra o sinistra dobbiamo specificare l'attributo ***sticky*** a cui assegneremo la **W di West** per l'allineamento a sinistra (e, quindi, **E di EAST** per l'allineamento a destra):

```
testo_due = Label(text="Un secondo testo!", fg="red", bg="green",  
font=("Helvetica",12)).grid(row=0, column=1, sticky=W)
```

L'allineamento a sinistra, ovviamente, non è visibile per l'elemento più grande, dato che definisce la larghezza della colonna. Proviamo, quindi, con l'oggetto ***testo_tre***:

```
testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",  
font=("Helvetica",12)).grid(row=1, column=0, sticky=W)
```

Con ***testo_quattro*** proviamo quindi l'allineamento a destra (E, per EAST):

```
testo_quattro = Label(text="Un quarto testo!", fg="red", bg="-orange",  
font=("Helvetica",12)).grid(row=1, column=1, sticky=E)
```



ATTRIBUTI PADDING, COLSPAN, ROWSPAN

Il metodo **GRID** contiene anche attributi come il ***padding*** o ***colspan*** e ***rowspan***, ovvero attributi che consentono di specificare uno spazio vuoto tra l'elemento e la “cella” della griglia che lo contiene, tra le varie colonne e tra le varie righe, rispettivamente.

Come sempre, per approfondire sia ***pack*** che ***grid***, raccomando caldamente la documentazione sul

sito di **Tkinter**.

Qui di seguito, il codice completo dell'esempio trattato in questo tutorial.

```
from tkinter import *

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
#testo_zero = Label(text='Questo testo va in alto', fg="red", bg="black",
font=("Helvetica",12)).pack()
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).grid(row=0, column=0)
testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).grid(row=0, column=1, sticky=W)
testo_tre = Label(text="Un terzo testo!", fg="red", bg="white",
font=("Helvetica",12)).grid(row=1, column=0, sticky=W)
testo_quattro = Label(text="Un quarto testo!", fg="red", bg="orange",
font=("Helvetica",12)).grid(row=1, column=1, sticky=E)
testo_cinque = Label(text="Un quinto testo!", fg="red", bg="cyan",
font=("Helvetica",12)).grid(row=2, sticky=W)
#bottone = Button(text="Un bottone inutile").pack()

#Avvia il mainloop
finestra.mainloop()

* * *
```

Tkinter - Tutorial 4

Aggiungere barra ed elementi del menù

In questo quarto tutorial del mini-corso su **Tkinter** vedremo **come aggiungere una barra dei menù**. Partiamo da un **progetto semplificato** rispetto a quello con cui ci eravamo lasciati, ovvero una semplice finestra con due testi incolonnati; il codice del progetto di partenza per questo tutorial è quindi il seguente:

```
from tkinter import *

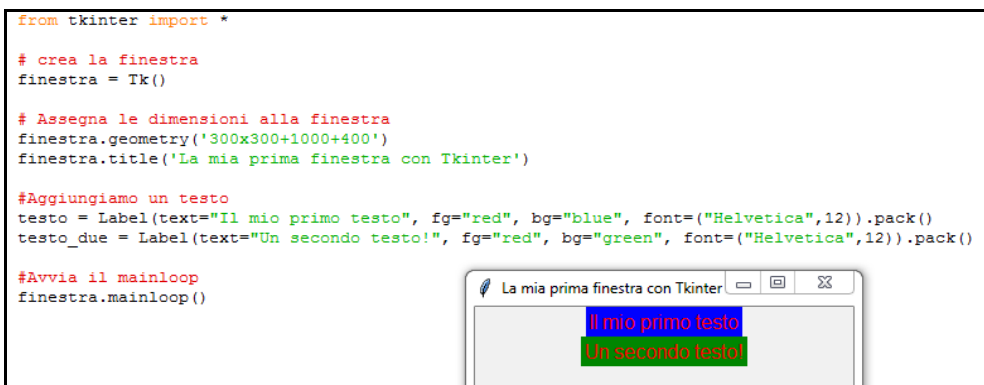
# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack()
testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack()

#Avvia il mainloop
finestra.mainloop()
```

Il risultato è visibile all'inizio della prossima pagina.



CREARE LA BARRA E UNA VOCE DI MENU

Per prima cosa dobbiamo creare la barra dei menù generale, ovvero un **oggetto Menu** che ha come *parent* (oggetto genitore o “contenitore”) la finestra dell'applicazione:

```
# Creare un menu
barra_menu = Menu(finestra)
```

Creiamo quindi **un nuovo oggetto menù** che, questa volta, avrà **come parent la barra dei menù**. Come si può capire dal nome, questo menu sarà **il classico menù File**:

```
menu_file = Menu(barra_menu)
```

ADD_CASCADE E ADD_COMMAND

In che rapporto sta il **menu file con la barra dei menù**?

Attraverso *add_cascade* diciamo a Tkinter che vogliamo che la barra dei menu abbia **un menu a discesa** la cui etichetta sarà **File** e che farà riferimento a *menu_file*:

```
barra_menu.add_cascade(label = "File", menu = menu_file)
```

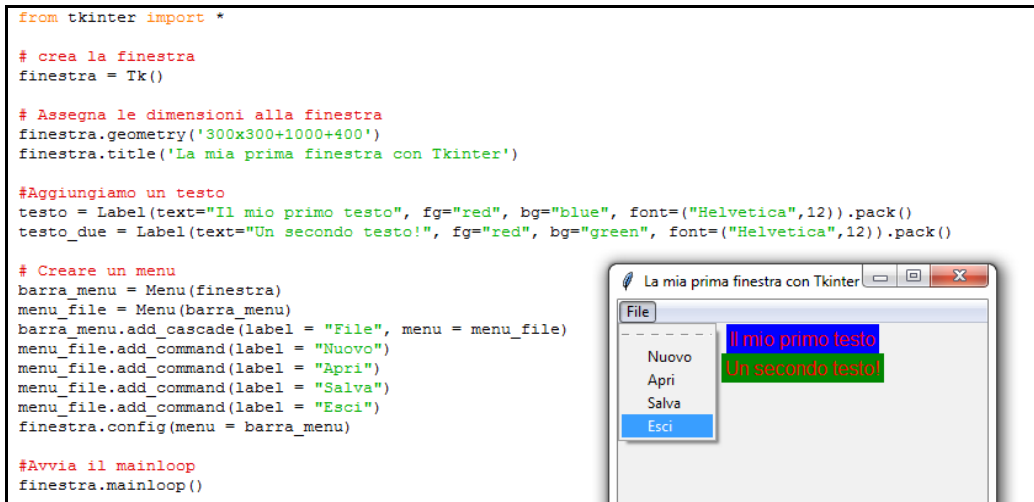
Ora non resta che **aggiungere le voci di menu a menu_file**, e questo è reso possibile dall'istruzione *add_command*. Per ora specifichiamo soltanto i nomi dei vari elementi (le voci di menù), senza attribuire loro alcuna proprietà:

```
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label = "Esci")
```

IL METODO CONFIG PER AGGIUNGERE LA BARRA ALLA FINESTRA

Infine, usiamo l'istruzione `finestra.config(menu = barra_menu)` per dire che vogliamo questa barra dei menu come menu della finestra. Avremo così il nostro menù, per ora inutilizzabile.

```
finestra.config(menu = barra_menu)
```



Possiamo rimuovere, impostandolo a zero, il *tearoff* del menu file.

Il *tearoff* è quella linea tratteggiata che serve a staccare il menu a cascata trasformandolo in finestra.

```
menu_file = Menu(barra_menu, tearoff=1)
```

Impostiamolo di nuovo a zero per liberarcene definitivamente:

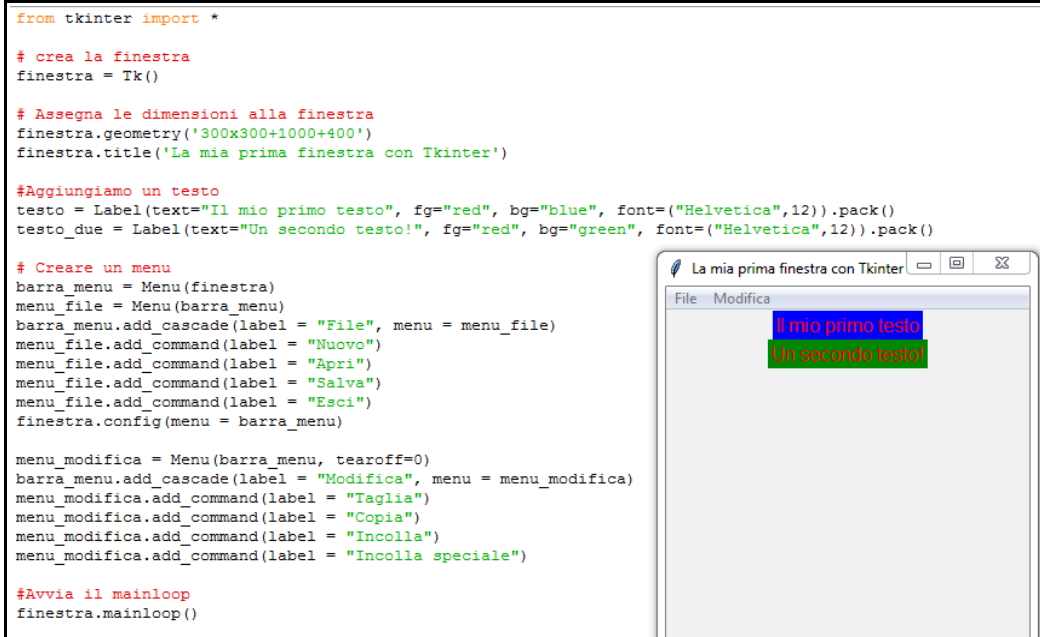
```
menu_file = Menu(barra_menu, tearoff=0)
```

CREARE UN ALTRO MENU

Con un semplice copia/incolla possiamo creare un nuovo menù, chiamato stavolta *Modifica*, che contiene le **classiche istruzioni del menu modifica**, quindi *taglia*, *copia*, eccetera.

```
menu_modifica = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "Modifica", menu = menu_modifica)
menu_modifica.add_command(label = "Taglia")
menu_modifica.add_command(label = "Copia")
menu_modifica.add_command(label = "Incolla")
menu_modifica.add_command(label = "Incolla speciale")
```

Il menù *Modifica* verrà quindi visualizzato accanto al menù *File*.



ORDINE DI VISUALIZZAZIONE DEI MENU NELLA BARRA

Notiamo che l'ordine di creazione influenza l'ordine di visualizzazione dei menù, infatti spostando il menù *Modifica* sopra il menù *File*, ce lo ritroveremo in prima posizione anche nella barra del nostro programma.

ASSOCIARE AZIONI ALLE VOCI DI MENU

Infine, vediamo come associare un'azione a una voce di menù.

Il comando è praticamente identico a quello dei pulsanti.

Aggiungiamo un nuovo oggetto, di tipo *Entry*, che corrisponde alla classica casella di testo.

Adesso dovremo scegliere una variabile da associare a quello che scriveremo nella casella.

Chiamiamo questa variabile di tipo stringa "*contenuto*":

```
contenuto = StringVar()
casella_testo = Entry(finestra, textvariable = contenuto).pack()
```

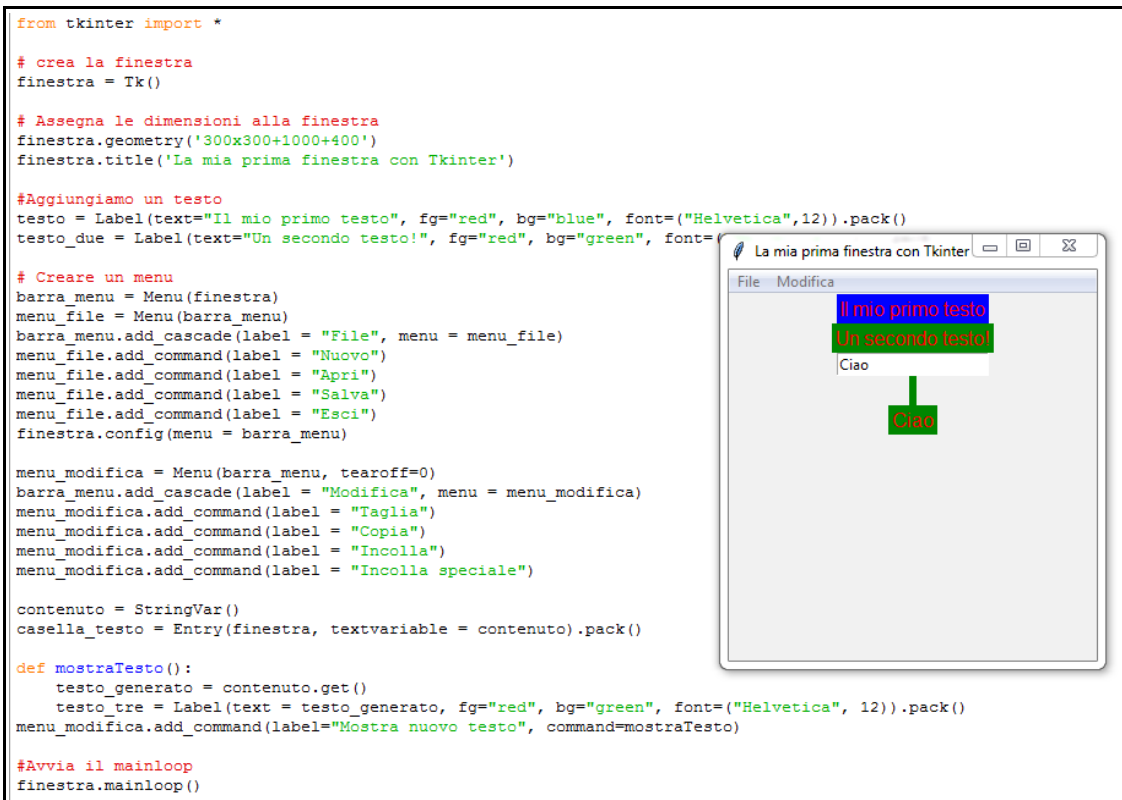
Quello che farà la nostra voce di menù sarà prendere il contenuto della casella di testo e visualizzarlo su un nuovo *label*.

Come per i pulsanti (*Buttons*), dobbiamo passare a **command** il **nome di una funzione senza parentesi**:

```
def mostraTesto():  
    testo_generato = contenuto.get()  
    testo_tre = Label(text = testo_generato, fg="red", bg="green",  
        font=("Helvetica", 12)).pack()  
    menu_modifica.add_command(label="Mostra nuovo testo", command=mostraTesto)
```

Attraverso il **metodo GET** prendiamo il valore di *contenuto* e lo mostriamo attraverso un nuovo label, *testo_tre*.

Per comodità diamo a *testo_tre* gli stessi attributi grafici di *testo_due* e vediamo se funziona. Cliccando su quella voce di menù, otterremo il testo generato.



Anche per questo tutorial è tutto; nella pagina seguente, **il codice completo dello script** prodotto in questa lezione (il risultato visivo è quello dell'immagine riportata qui sopra).

```
from tkinter import *

def mostraTesto():

    testo_generato = contenuto.get()

    testo_tre = Label(text = testo_generato, fg="red", bg="green",
    font=("Helvetica", 12)).pack()

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack()

testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack()

contenuto = StringVar()
casella_testo = Entry(finestra, textvariable = contenuto).pack()

# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label = "Esci")
finestra.config(menu = barra_menu)
menu_modifica = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "Modifica", menu = menu_modifica)
menu_modifica.add_command(label = "Taglia")
menu_modifica.add_command(label = "Copia")
menu_modifica.add_command(label = "Incolla")
menu_modifica.add_command(label = "Mostra nuovo testo",
command=mostraTesto)

#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter - Tutorial 5

Creare delle finestre di dialogo

In questo tutorial vedremo **come costruire le finestre di dialogo più comuni** che possiamo chiamare da un menù in **Tkinter**.

Nel tutorial precedente abbiamo creato una voce di menù che, se cliccata, creava un nuovo *Label*; adesso la elimineremo, **trasformando il menù Modifica in un più semplice menù About**, che però visualizzerà a video una **finestra di dialogo** (l'argomento di questo tutorial).

Riscriviamo quindi lo script con questo contenuto:

```
from tkinter import *

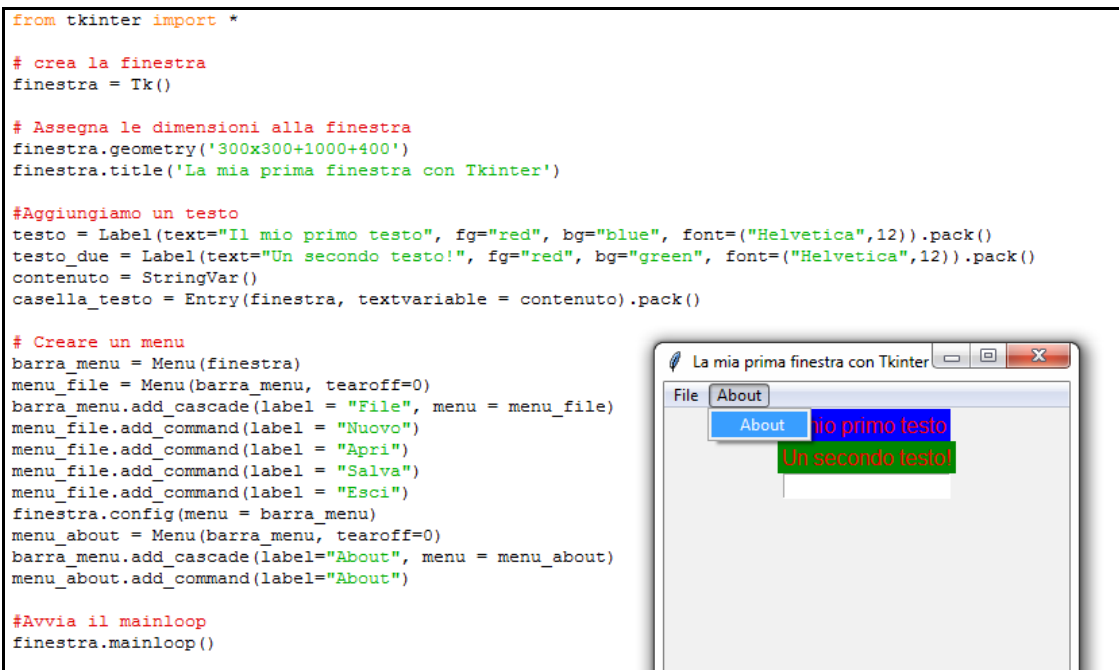
# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack()
testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack()
contenuto = StringVar()
casella_testo = Entry(finestra, textvariable = contenuto).pack()
```

```
# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label = "Esci")
finestra.config(menu = barra_menu)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label="About", menu = menu_about)
menu_about.add_command(label="About")

#Avvia il mainloop
finestra.mainloop()
```



Il nuovo menù, ovviamente, non fa nulla, perché non abbiamo associato nessuna funzione.

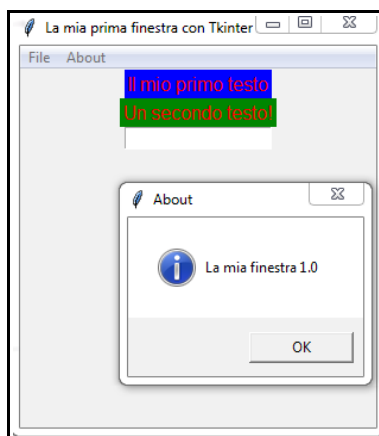
Aggiungiamo, quindi, un **command** alla voce *About* e creiamo la **funzione chiama_about**.

```
def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")
menu_about.add_command(label = "About", command = chiama_about)
```

FINESTRA DI DIALOGO MESSAGEBOX E SHOWINFO

La funzione *chiamata* `about` sarà semplicissima: per visualizzare una finestra di dialogo con un testo informativo, infatti, basterà scrivere `messagebox.showinfo` e passare come **parametri** il titolo della finestra e il messaggio da visualizzare. Tutto qui.

Vediamo il risultato: avremo, a video, la nostra finestra con **l'icona info**.



FINESTRE SHOWWARNING, SHOWERROR

Cambiando `showinfo` in `showwarning` o `showerror`, cambierà l'icona corrispondente (ovvero: informazione, avviso, errore). Rimando alla documentazione ufficiale di **Tkinter** per l'elenco completo dei metodi di `messagebox`.

CHIUDERE LA FINESTRA: DESTROY, CON CONFERMA

Vediamo adesso come **associare una funzione alla voce "Esci" del menù**.

Definiamo quindi la funzione uscita impostando, innanzitutto, il comando `finestra.destroy` per chiudere la finestra:

```
def uscita():  
    finestra.destroy()  
menu_file.add_command(label="Esci", command=uscita)
```

Il comando funziona, ma vorremmo, prima di chiudere, almeno **una richiesta di conferma**.

Ci viene incontro il comando *messagebox.askyesno* a cui dovremo dare, come sempre, un *title* e un *message*:

```
def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero
    uscire?")
    if risposta:
        finestra.destroy()
```

```
from tkinter import *

# crea la finestra
finestra = Tk()

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

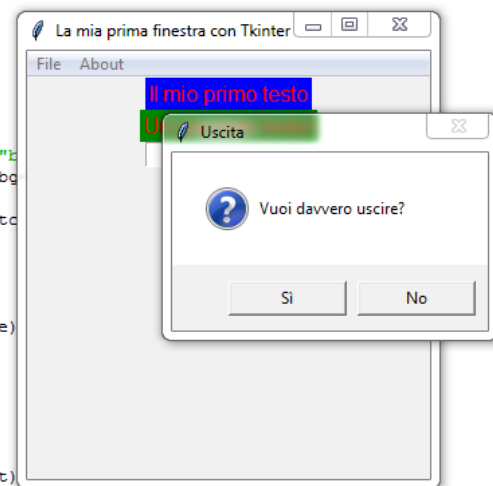
def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red", bg="blue")
testo_due = Label(text="Un secondo testo!", fg="red", bg="blue")
contenuto = StringVar()
casella_testo = Entry(finestra, textvariable = contenuto)

# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
finestra.config(menu = barra_menu)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label="About", menu = menu_about)
menu_about.add_command(label = "About", command = chiama_about)

#Avvia il mainloop
finestra.mainloop()
```



Stiamo quindi associando una variabile “*risposta*” al nostro *messagebox*.

Il valore di “*risposta*” sarà un **booleano**: *True* se l'utente cliccherà su Ok, altrimenti *False*. In questo modo, potremo controllare la risposta con un *if*.

Possiamo scrivere “*if risposta==1:*”, o uguale a *True* o semplicemente “*if risposta:*” per rispondere al click su Yes.

In caso di click su Yes, chiudiamo la finestra. Non è neanche necessario specificare il caso alternativo, infatti cliccando su No il box si chiude senza fare nulla, riportandoci alla finestra dell'applicazione; cliccando invece su Yes, terminerà il programma.

Possiamo sostituire *askyesno* con *askokcancel*, cambierà soltanto il nome dei pulsanti visualizzati.

Qui di seguito, **il codice completo dello script** prodotto in questo tutorial:

```
from tkinter import *
def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")
def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()
# crea la finestra
finestra = Tk()
# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')
#Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack()
testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack()
contenuto = StringVar()
casella_testo = Entry(finestra, textvariable = contenuto).pack()
# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label = "About", command=chiama_about)
finestra.config(menu = barra_menu)
#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter - Tutorial 6

Altre operazioni effettuabili con i menù

In questo sesto tutorial del corso su **Tkinter** parleremo di altre operazioni che possiamo svolgere attraverso i **menù**.

Nella lezione precedente abbiamo visto come creare una barra dei menù e svolgere semplici azioni come far apparire il *messagebox* con le informazioni sul programma o chiudere la finestra.

FINESTRA DI DIALOGO DEL FILE BROWSER PER APRIRE UN FILE

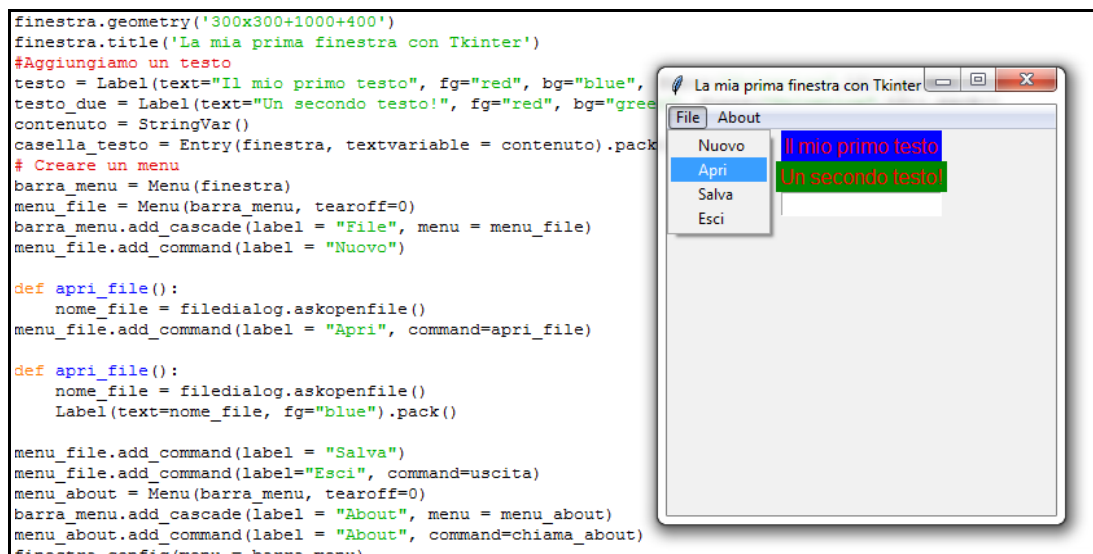
Adesso vediamo come chiamare la finestra di dialogo per **aprire un file**. Aggiungiamo il classico *command* alla voce del menù e definiamo la **funzione *apri_file***:

```
def apri_file():  
    nome_file = filedialog.askopenfile()  
    menu_file.add_command(label = "Apri", command=apri_file)
```

La finestra di dialogo viene quindi creata attraverso il metodo *filedialog.askopenfile*.

Per capire cosa effettivamente ci restituisca la finestra, creiamo un'etichetta “*label_blu*” al volo e mettiamola in coda a ciò che è già mostrato a video:

```
def apri_file():  
    nome_file = filedialog.askopenfile()  
    Label(text=nome_file, fg="blue").pack()
```



Si aprirà quindi la classica finestra di apertura del file, ed ecco cosa dovremo gestire per aprire correttamente il file: abbiamo il **percorso**, la **modalità *r***, ovvero lettura, e informazioni sull'**encoding**.

Restituendoci il percorso, la funzione ci dà la possibilità di aprire quel file caricandolo in una variabile *handler* o di eseguire altre operazioni a piacere.

PER SELEZIONARE UNA CARTELLA DA DISCO...

Possiamo anche chiamare la finestra di dialogo che ci permette di selezionare non un file ma **una cartella**, e il suo nome sarà ovviamente *askdirectory*:

```
def apri_file():
    nome_file = filedialog.askdirectory()
    Label(text=nome_file, fg="blue").pack()
```

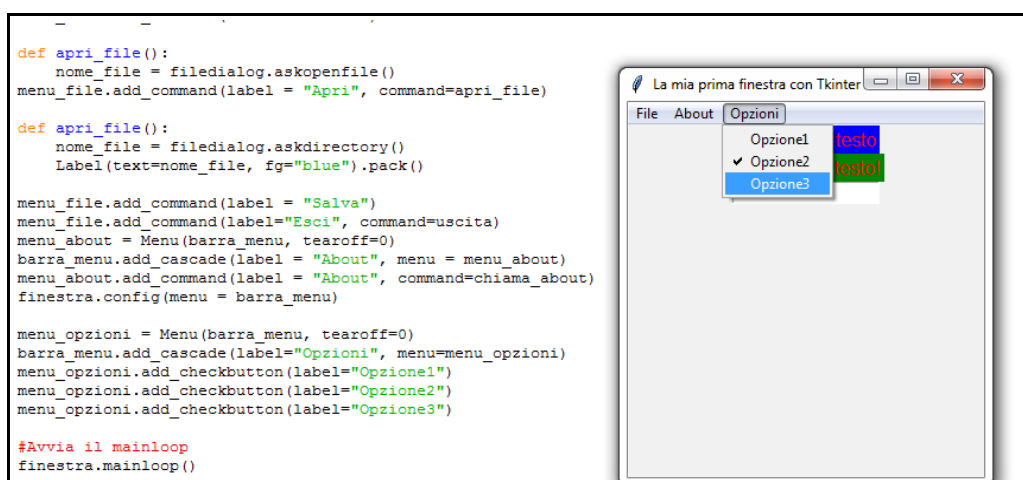
CREARE VOCI DI MENU DI TIPO CHECKBUTTON

Un'altra cosa interessante che possiamo fare è **creare dei menu con i *checkboxbutton*** (le classiche caselle di spunta).

Creo quindi un nuovo menù chiamato *Opzioni* ma, anziché inserire ***add_command***, scrivo ***add_checkbutton***:

```
menu_opzioni = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label="Opzioni", menu=menu_opzioni)
menu_opzioni.add_checkbutton(label="Opzione1")
menu_opzioni.add_checkbutton(label="Opzione2")
menu_opzioni.add_checkbutton(label="Opzione3")
```

Una volta fatto click su una voce di menù, vedremo **il segno di spunta** che apparirà accanto ad indicare che l'opzione è attiva. Cliccando di nuovo, il segno di spunta sparirà.



Potremo poi **assegnare una funzione** che, effettivamente, faccia qualcosa.

VALORI UGUALI PER VAR NEI CHECKBUTTON

Assegnare un identico valore ***var*** a più voci di menu, ci permette di assegnare loro lo stesso comportamento:

```
menu_opzioni.add_checkbutton(label="Opzione1", var=1)
menu_opzioni.add_checkbutton(label="Opzione2", var=1)
menu_opzioni.add_checkbutton(label="Opzione3", var=2)
```

Cliccando su *Opzione1*, infatti, faremo apparire il segno di spunta anche accanto a *Opzione2*, mentre *Opzione3* continuerà a comportarsi in modo autonomo, avendo un valore *var* diverso.

Anche per quanto riguarda i menù, il consiglio è quello di esplorare la documentazione per trovare ciò che più può rispondere alle vostre necessità. Le possibilità offerte da **Tkinter** sono le più disparate; ad esempio, potrete chiamare una finestra predefinita **per selezionare un colore** tramite *colourchooser.askcolor()*.

Qui di seguito, **il codice completo dello script** prodotto in questo tutorial:

```
from tkinter import *

def apri_file():
    nome_file = colorchooser.askcolor()
    Label(text=nome_file, fg="blue").pack()

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

#Aggiungiamo un testo
testo = Label(text="Il mio primo testo", fg="red", bg="blue",
font=("Helvetica",12)).pack()
testo_due = Label(text="Un secondo testo!", fg="red", bg="green",
font=("Helvetica",12)).pack()
contenuto = StringVar()
casella_testo = Entry(finestra, textvariable = contenuto).pack()

# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
```

```
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri", command=apri_file)
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label = "About", command=chiama_about)
menu_opzioni = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label="Opzioni", menu=menu_opzioni)
menu_opzioni.add_checkbutton(label="Opzione1")
menu_opzioni.add_checkbutton(label="Opzione2")
menu_opzioni.add_checkbutton(label="Opzione3")
finestra.config(menu = barra_menu)

#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter - Tutorial 7

Elementi di interfaccia: Radiobutton e Spinbox

In questo tutorial parleremo di due elementi tipici delle interfacce grafiche: ***Radiobutton*** (caselle di scelta esclusiva) e ***Spinbox*** (casella di scelta per numeri in un range prefissato).

Per fare un po' di ordine, **ripuliamo il codice** rispetto alle precedenti lezioni, lasciando soltanto la finestra con i menu *File* e *About*; ecco quindi lo script di partenza per questo tutorial:

```
from tkinter import *

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

# Creare un menu
barra_menu = Menu(finestra)
```

```
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label = "About", command=chiama_about)
finestra.config(menu = barra_menu)

#Avvia il mainloop
finestra.mainloop()
```

CREARE I RADIOBUTTON

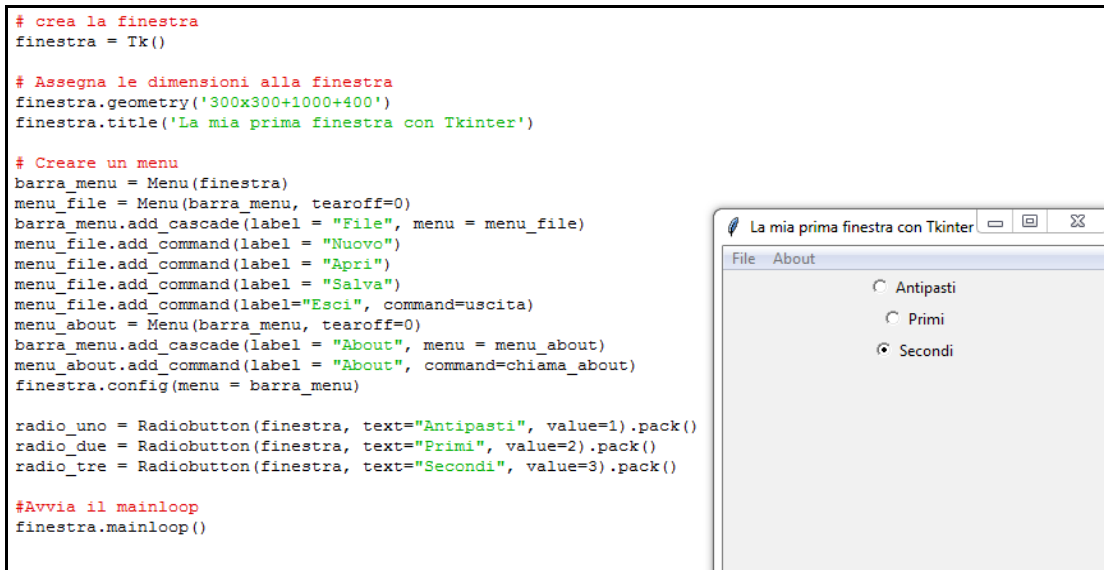
I **Radiobutton** sono le caselle di scelta, dalla forma tipicamente circolare (mentre i **Checkbutton** hanno, in genere, forma quadrata) che servono **per definire delle scelte esclusive**, ovvero di scegliere una o più voci tra un dato insieme **escludendo le altre opzioni**.

Di base, quando si fa un click su un **Radiobutton** vengono disattivati automaticamente tutti gli altri **Radiobutton** presenti nell'applicazione, a meno di non definire dei **“gruppi logici”** per i **Radiobutton** (e, ovviamente, vedremo come fare, nel corso di questo tutorial, con l'utilizzo di **variable**).

È possibile, comunque, definire **altre regole o combinazioni** per impostare l'attivazione e la disattivazione dei **Radiobutton** presenti in un contenitore... beh, andiamo con calma, iniziamo con la creazione degli elementi grafici, poi passeremo a definirne le regole!

Creiamo il primo insieme di **Radiobutton** immaginando di dover far scegliere a un cliente tra gli antipasti, i primi e i secondi.

```
radio_uno = Radiobutton(finestra, text="Antipasti", value=1).pack()
radio_due = Radiobutton(finestra, text="Primi", value=2).pack()
radio_tre = Radiobutton(finestra, text="Secondi", value=3).pack()
```



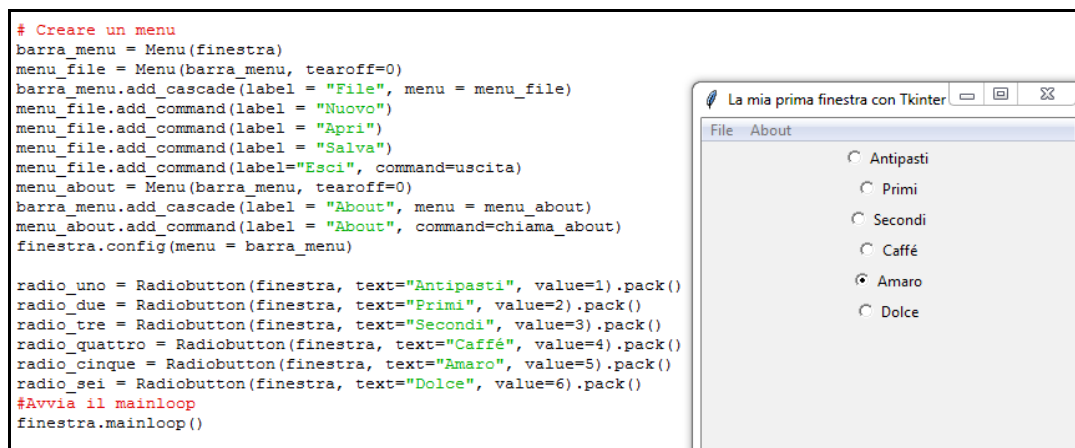
Passiamo tre parametri ad un oggetto *Radiobutton*, quando lo creiamo: il nome del **parent**, (in questo caso, *finestra*), **l'etichetta** da dare al bottone e, infine, **values**, che definiremo in seguito.

Come visibile eseguendo il programma, **possiamo scegliere solo una delle tre opzioni**, che subito escluderà le altre due; questo avviene perché i numeri assegnati a **value** sono tutti diversi.

Aggiungiamo altri tre *Radiobutton* a cui assegneremo i nomi di prodotti da fine pasto, come il caffè, l'amaro e il dolce. Anche stavolta metteremo dei valori diversi per **values**.

```
radio_quattro = Radiobutton(finestra, text="Caffé", value=4).pack()
radio_cinque = Radiobutton(finestra, text="Amaro", value=5).pack()
radio_sei = Radiobutton(finestra, text="Dolce", value=6).pack()
```

Ovviamente, non è cambiato nulla: la scelta potrà essere **solo tra uno degli elementi proposti**.



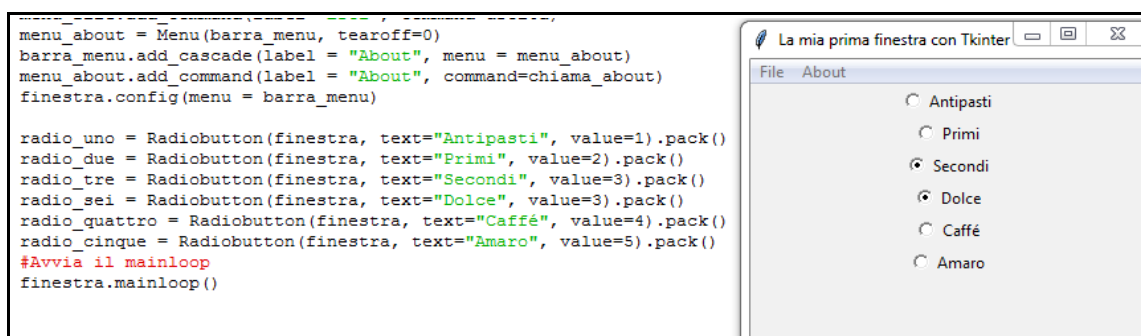
VALORI UGUALI PER RADIOBUTTON DIVERSI: VALUE

Supponiamo però di avere dei secondi molto costosi e di voler regalare il dolce a chiunque scelga il secondo, e di permettere la scelta del dolce solo a chiunque abbia scelto il secondo.

Vincoliamo, cioè, due elementi (in questo caso, dolce e secondo); per far ciò, basterà **assegnare lo stesso numero di value**:

```
radio_tre = Radiobutton(finestra, text="Secondi", value=3).pack()
radio_sei = Radiobutton(finestra, text="Dolce", value=3).pack()
```

Ecco che la scelta del secondo comporterà anche la selezione del dolce e viceversa, mentre **un'altra scelta li deselezionerà entrambi**.



RAGGRUPPAMENTO DEI RADIOBUTTON: VARIABLE

Per evitare il fallimento, decidiamo però di permettere ai clienti di scegliere un pasto principale (tra antipasti, primi e secondi) e un prodotto da fine pasto (tra caffè, amaro e dolce).

Dobbiamo fare in modo, cioè, che queste terne di **Radiobutton** appartengano a **due gruppi “logici” separati**, per cui **le scelte saranno esclusive solo all'interno di ciascun gruppo**, senza influenzare i *Radiobutton* di altri gruppi.

Questa operazione è svolta dal parametro **variable**, in questo modo:

```
radio_uno = Radiobutton(finestra, text="Antipasti", value=1,
variable=1).pack()

radio_due = Radiobutton(finestra, text="Primi", value=2,
variable=1).pack()

radio_tre = Radiobutton(finestra, text="Secondi", value=3,
variable=1).pack()

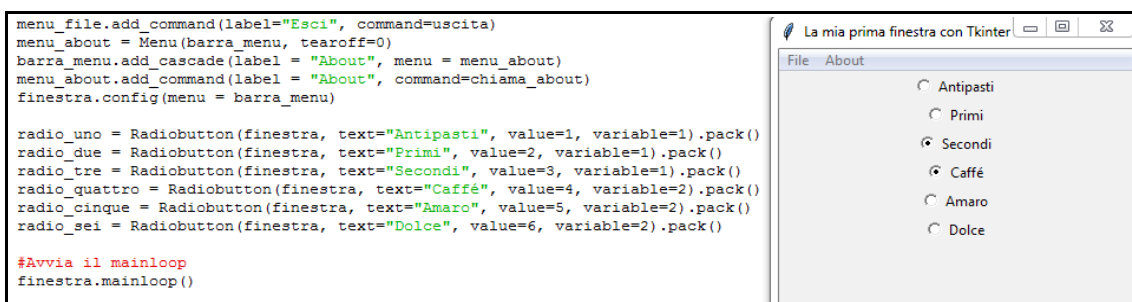
radio_quattro = Radiobutton(finestra, text="Caff ", value=4,
```

```
variable=2).pack()

radio_cinque = Radiobutton(finestra, text="Amaro", value=5,
variable=2).pack()

radio_sei = Radiobutton(finestra, text="Dolce", value=6,
variable=2).pack()
```

Tutti i **Radiobutton** con identico valore di **variable** saranno trattati come membri dello stesso gruppo, quindi **la selezione esclusiva potrà avvenire solo tra essi**, e non influenzerà gli altri. Potremo così selezionare un antipasto e un caffè, un antipasto e un amaro o un antipasto e un dolce.



Possiamo assegnare a **variable** anche **valori diversi da quelli numerici**, ad esempio una stringa, mantenendo però l'identità con il resto dei membri del gruppo.

```
radio_quattro = Radiobutton(finestra, text="Caff ", value=4,
variable="due").pack()

radio_cinque = Radiobutton(finestra, text="Amaro", value=5,
variable="due").pack()

radio_sei = Radiobutton(finestra, text="Dolce", value=6,
variable="due").pack()
```

SPINBOX: CREAZIONE E PARAMETRI

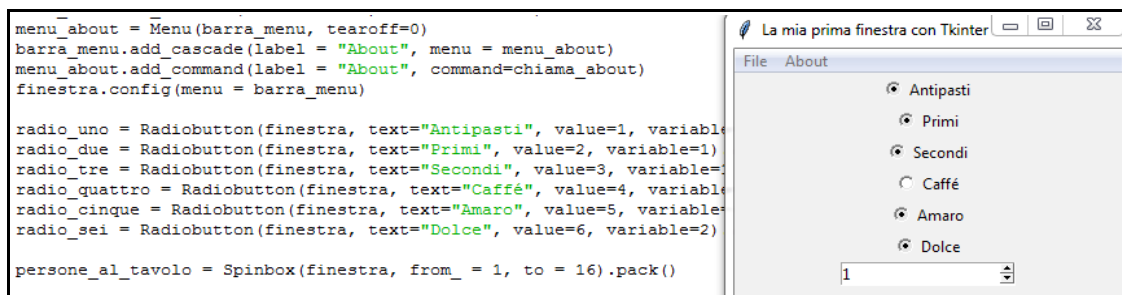
Aggiungiamo infine uno **Spinbox** (una casella che ci consente di definire un valore numerico) per sapere quante persone si siederanno al tavolo.

I parametri da passare a **Spinbox** sono: il nome del *parent* (in questo caso, la variabile *finestra*) e un range di *valori*.

Avremo quindi:

```
persone_al_tavolo = Spinbox(finestra, from_ = 1, to = 16).pack()
```

Attenzione: il valore di partenza dev'essere preceduto da *from_*, mentre per il valore finale useremo *to*. Con lo **Spinbox** non è quindi possibile andare oltre il limite inferiore o superiore.



SPECIFICARE I VALORI POSSIBILI PER UNA SPINBOX: VALUES

Infine, potremo decidere di accettare solo un numero definito di persone per tavolo perch  abbiamo solo tavoli di una certa dimensione o non possiamo mettere una sedia a capotavola. Attraverso **values** possiamo **restringere il numero delle opzioni disponibili**:

```
persone_al_tavolo = Spinbox(finestra, values = (1,2,4,8,12)).pack()
```

Qui di seguito, **il codice completo dello script** prodotto in questo tutorial:

```
from tkinter import *

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

radio_uno = Radiobutton(finestra, text="Antipasti", value=1,
variable=1).pack()

radio_due = Radiobutton(finestra, text="Primi", value=2,
variable=1).pack()
```

```
radio_tre = Radiobutton(finestra, text="Secondi", value=3,
variable=1).pack()

radio_quattro = Radiobutton(finestra, text="Caff ", value=4,
variable=2).pack()

radio_cinque = Radiobutton(finestra, text="Amaro", value=5,
variable=2).pack()

radio_sei = Radiobutton(finestra, text="Dolce", value=6,
variable=2).pack()

persone_al_tavolo1 = Spinbox(finestra, from_ = 1, to = 16).pack()
persone_al_tavolo2 = Spinbox(finestra, values=(1,2,4,8,12)).pack()


# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label = "About", command=chiama_about)
finestra.config(menu = barra_menu)


#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter - Tutorial 8

Elementi di interfaccia: Listbox e Slider

In questo ottavo tutorial sulle basi di **Tkinter** per la definizione di interfacce grafiche in Python parleremo di **Listbox** e **Slider**, gli ultimi due widget di base da esaminare prima di passare al **Canvas**.

Partiamo dal progetto elaborato nelle ultime lezioni, "ripulito" da tutti gli elementi interni alla finestra:

```
from tkinter import *

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')

# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
```

```
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label = "About", command=chiama_about)
finestra.config(menu = barra_menu)

#Avvia il mainloop
finestra.mainloop()
```

LISTBOX

Una **Listbox** è, come suggerisce il nome, una **lista di elementi**, dalla quale potremo scegliere **uno o anche più elementi** (selezione multipla) contemporaneamente, a seconda di come imposteremo il componente via codice.

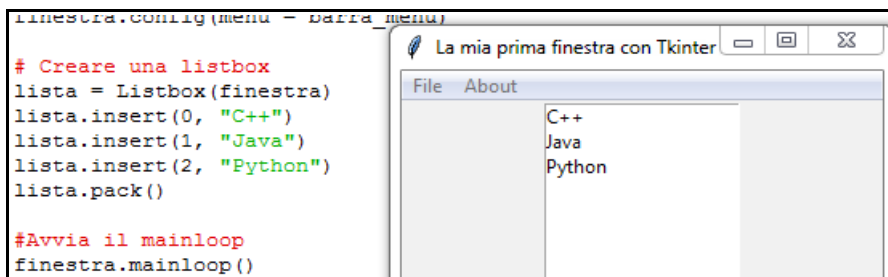
Le **Listbox** sono quindi perfette per rappresentare **liste** o, in generale, **raccolte di elementi**, che l'utente potrà selezionare visivamente all'interno di una casella (“*box*”).

Per creare un **Listbox** dovremo aggiungere il classico **widget** con l'iniziale maiuscola. La novità sta nel fatto che non dovremo scrivere **.pack()** in coda al costruttore.

Tra il **.pack()** che inseriremo a parte e il costruttore, infatti, dovremo **inserire gli elementi della lista**:

```
# Creare una listbox
lista = Listbox(finestra)
lista.insert(0, "C++")
lista.insert(1, "Java")
lista.insert(2, "Python")
lista.pack()
```

La sintassi è, quindi, **nomeLista.insert()** e, come argomenti della funzione **insert** dentro la parentesi, l'indice e il nome dell'elemento. Il risultato è visibile nell'immagine seguente (prossima pagina).



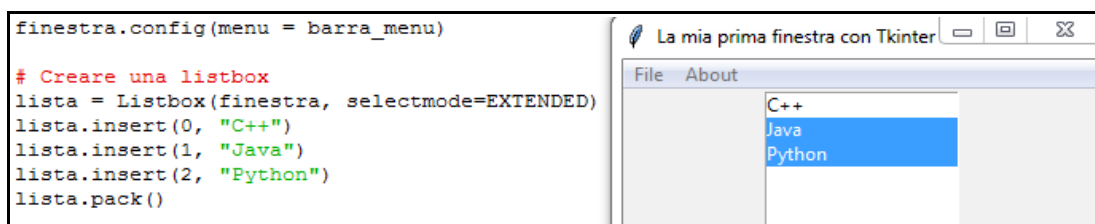
SELEZIONE MULTIPLA PER LE LISTBOX: MULTIPLE E EXTENDED

Di default, è possibile selezionare un solo elemento da una **ListBox**; per permettere la selezione di più elementi, scriviamo “*selectmode = MULTIPLE*”:

```
lista = Listbox(finestra, selectmode=MULTIPLE)
```

Possiamo comunque usare il valore di default, ovvero **EXTENDED**, ed effettuare selezioni multiple premendo CTRL o SHIFT mentre facciamo click col mouse.

```
lista = Listbox(finestra, selectmode=EXTENDED)
```



INDICI DEGLI ELEMENTI DI UNA LISTBOX

Se non vogliamo tenere a mente l'indice o se vogliamo usare un ciclo *for* per inserire i nomi in automatico, possiamo usare **END**, in maiuscolo, al posto dell'indice numerico:

```
lista.insert(2, "Python")
lista.insert(END, "Ruby")
lista.pack()
```

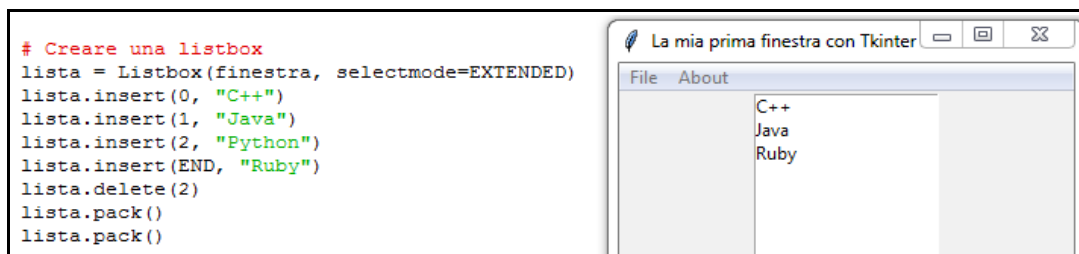
Va notato, tuttavia, che l'indice non si comporta effettivamente come tale. Innanzitutto la lista non sarà ordinata per indice, ma per ordine di inserzione, infatti l'elemento “C++” resta sempre in cima:

```
lista.insert(5, "C++")
lista.insert(1, "Java")
```

inoltre, quando utilizziamo il metodo ***delete***, seguito dall'indice, questo non elimina l'indice che abbiamo assegnato, ma attribuisce l'indice 0 al primo elemento, l'indice 1 al secondo, e così via, infatti scrivendo:

```
lista.insert(0, "C++")
lista.insert(1, "Java")
lista.insert(2, "Python")
lista.insert(END, "Ruby")
lista.delete(2)
lista.pack()
```

a sparire non sarà l'elemento “Java”, a cui abbiamo dato indice 2, ma “Python”, il terzo elemento della lista.; inoltre, chiedendo di eliminare l'indice 0, anziché dare errore, eliminerà “C++”.



LO SLIDER: SELEZIONE DI UN VALORE IN UN RANGE

Passiamo adesso al secondo e ultimo elemento: lo **Slider**, indicato in **Tkinter** con il widget ***Scale***.

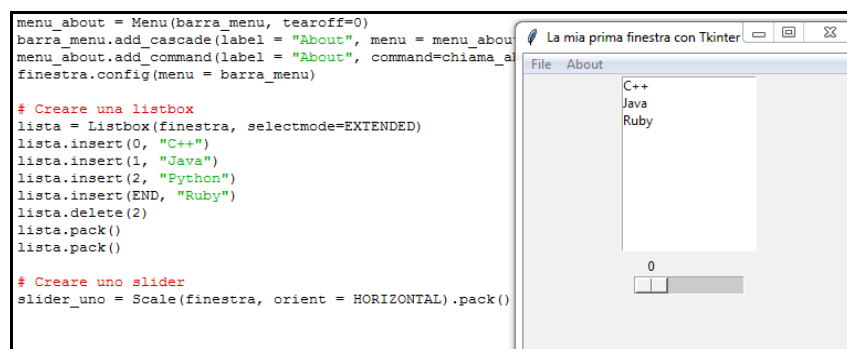
Creando uno **Slider** vuoto, **Tkinter** genererà uno **Slider** verticale numerato da 1 a 100:

```
# Creare uno slider
slider_uno = Scale(finestra).pack()
```

Attraverso i primi attributi definiamo l'aspetto dello Slider.

Per iniziare, cambiamo l'orientamento da verticale a orizzontale scrivendo **“*orient* = *HORIZONTAL*”**:

```
slider_uno = Scale(finestra, orient = HORIZONTAL).pack()
```



PARAMETRI DEGLI SLIDER: LENGTH, WIDTH, SLIDERLENGTH

Il parametro **length** ci permette di impostare la lunghezza in pixel dello Slider, che occuperà in questo caso tutta la finestra, restando comunque nell'intervallo (di valori) 1-100.

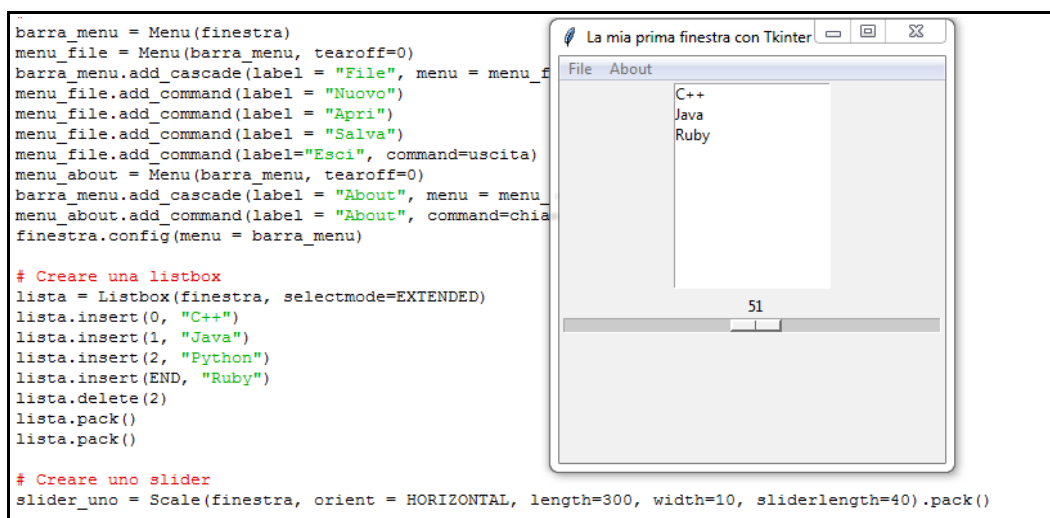
```
slider_uno = Scale(finestra, orient = HORIZONTAL, length=300).pack()
```

Con **width** cambiamo lo spessore dello Slider, rendendolo più alto.

```
slider_uno = Scale(finestra, orient = HORIZONTAL, length=300,  
width=30).pack()
```

Con **sliderlength** cambiamo la lunghezza del pezzo trascinabile, e qui Tkinter riconosce che anziché uno **scale** si tratta di uno Slider:

```
slider_uno = Scale(finestra, orient = HORIZONTAL, length=300, width=10,  
sliderlength=40).pack()
```



INTERVALLO DEI VALORI DI UNO SLIDER: FROM_, TO, TICKS

Cambiamo adesso l'intervallo dei valori in cui si muove lo **Slider** utilizzando due comandi già visti con lo **Spinbox**: **from_** e **to**:

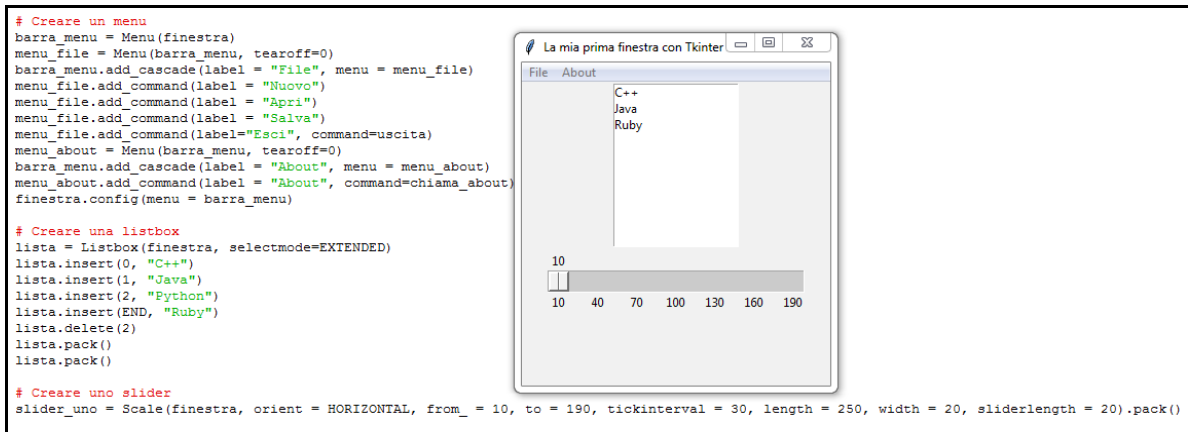
```
slider_uno = Scale(finestra, orient = HORIZONTAL, from_ = 10, to=200,  
length=250, width=20, sliderlength=20).pack()
```

Lo **Slider** così definito potrà quindi “muoversi” tra 10 e 190.

Definiamo, infine, dei ***ticks***, ovvero delle tacche sulla scala graduata attraverso ***tickinterval***:

```
slider_uno = Scale(finestra, orient = HORIZONTAL, from_ = 10, to = 190,  
tickinterval = 30, length = 250, width = 20, sliderlength = 20).pack()
```

Impostiamo il valore a 30 e, per far quadrare i conti, abbassiamo il massimo a 190.



Va detto, comunque, che i ***ticks*** servono solo come riferimento visivo e non funzionano da "calamite" per lo ***Slider***, che dunque potrà sempre selezionare i valori tra un ***tick*** e un altro.

Qui di seguito, il codice completo dello script prodotto in questo tutorial:

```
from tkinter import *

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('300x300+1000+400')
finestra.title('La mia prima finestra con Tkinter')
```

```
lista = Listbox(finestra)
lista.insert(0, "C++")
lista.insert(1, "Java")
lista.insert(2, "Python")
lista.insert(END, "Ruby")
lista.delete(2)
lista.pack()

slider_uno = Scale(finestra, orient = HORIZONTAL, from_ = 10, to = 190,
tickinterval = 30, length = 250, width = 20, sliderlength = 20).pack()

# Creare un menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label = "About", command=chiama_about)
finestra.config(menu = barra_menu)

#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter – Tutorial 9

La Canvas. Disegnare linee e figure. Visualizzare immagini

In questo nono tutorial su **Tkinter** vedremo come disegnare sulla **Canvas**, ovvero su una **superficie** all'interno della finestra dell'applicazione.

La **Canvas** è quindi un oggetto che fa da "tela" (e in effetti è questo il suo significato letterale) per tutte le nostre creazioni grafiche.

Per far spazio alla **Canvas** nella finestra applicativa del nostro esempio, ho allargato la finestra a 600x600 pixel. Ecco il codice di partenza dello script che andremo a definire in questo tutorial:

```
from tkinter import *

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

def uscita():
    risposta = messagebox.askyesno(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

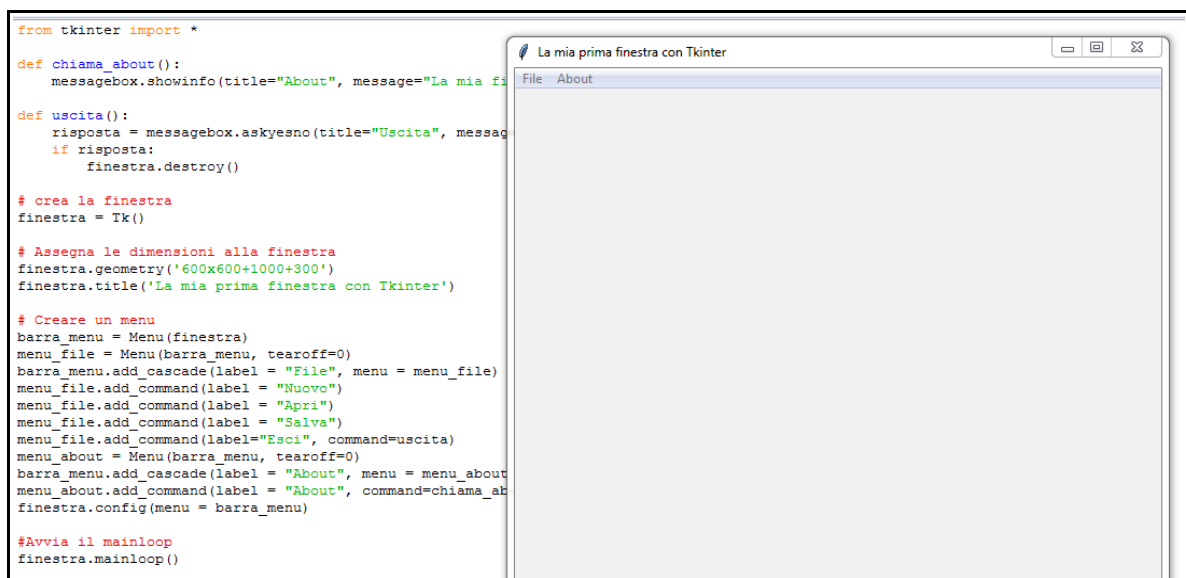
# crea la finestra
finestra = Tk()

# Assegna le dimensioni alla finestra
finestra.geometry('600x600+1000+300')
finestra.title('La mia prima finestra con Tkinter')

# Creare un menu
```

```
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label = "Nuovo")
menu_file.add_command(label = "Apri")
menu_file.add_command(label = "Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff=0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label = "About", command=chiama_about)
finestra.config(menu = barra_menu)

#Avvia il mainloop
finestra.mainloop()
```



CREARE UNA CANVAS

L'oggetto **Canvas** si crea come tutti gli altri oggetti e, come per l'oggetto **Listbox**, è meglio separare il *pack()* dalla creazione vera e propria.

Il nome del widget è **Canvas**, con la C maiuscola, mentre *width* e *height* sono la lunghezza e la larghezza in pixel:

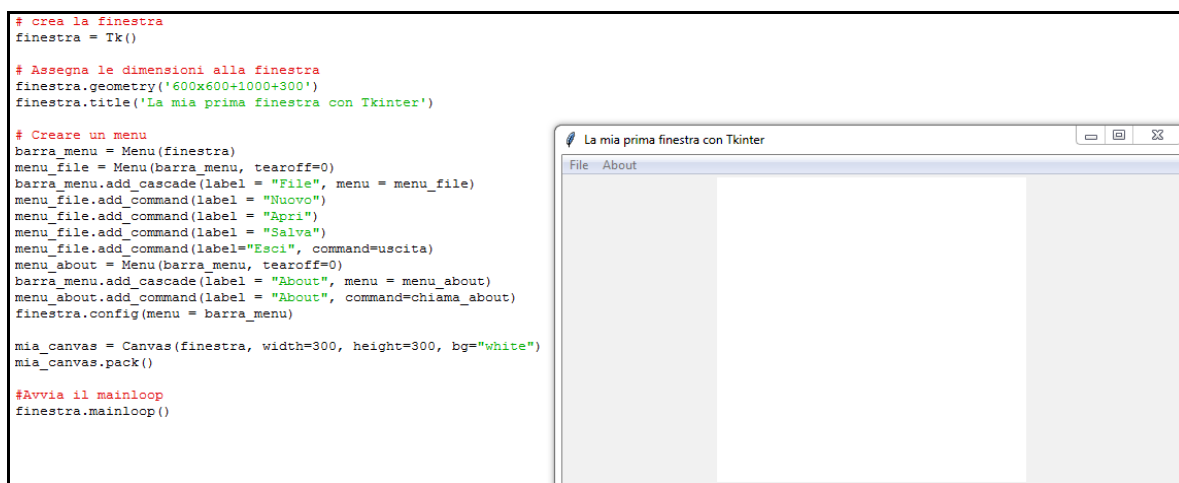
```
# Crea la canvas
mia_canvas = Canvas(finestra, width=300, height=300)
mia_canvas.pack()
```

Facendo partire il programma con queste modifiche... non vedremo nulla.

In realtà la **Canvas** esiste già, ma il suo **colore di sfondo di default** è quello del resto della finestra.

Cambiamo quindi lo sfondo e impostiamolo, ad esempio, a bianco, mediante l'attributo **bg** (*background*):

```
mia_canvas = Canvas(finestra, width=300, height=300, bg="white")
```



AGGIUNGERE ELEMENTI ALLA CANVAS: LINEE E FIGURE

Aggiungiamo il primo elemento. Anche in questo caso la documentazione è molto vasta, poiché **Tkinter** ci permette di creare una grande varietà di figure.

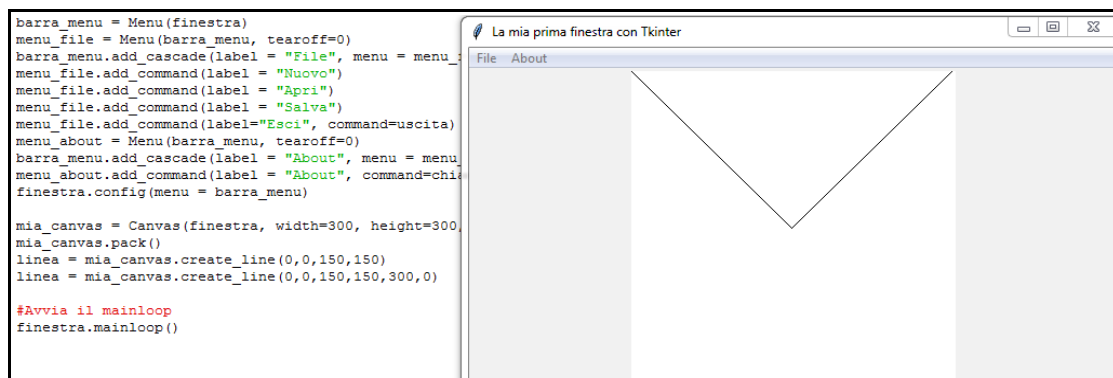
Iniziamo dalla **linea** passando come attributi del metodo **create_line** le coordinate nella forma x1, y1, x2, y2, e così via per tutti i punti che vogliamo. Tracciamo una linea da 0,0 va a 150,150, cioè al centro dello schermo:

```
linea = mia_canvas.create_line(0,0,150,150)
```

Notiamo che, a differenza del piano cartesiano, **l'origine si trova in alto a sinistra** anziché in basso a sinistra, per cui il punto 0,0 è in alto a sinistra e 150,150 è al centro dello schermo; in pratica, maggiore sarà il valore di y, più in basso si troverà il punto.

Per avere la conferma di quanto appena detto aggiungiamo un terzo punto a X 300 e y 0, che porterà la nostra linea nell'angolo in alto a destra:

```
linea = mia_canvas.create_line(0,0,150,150,300,0)
```



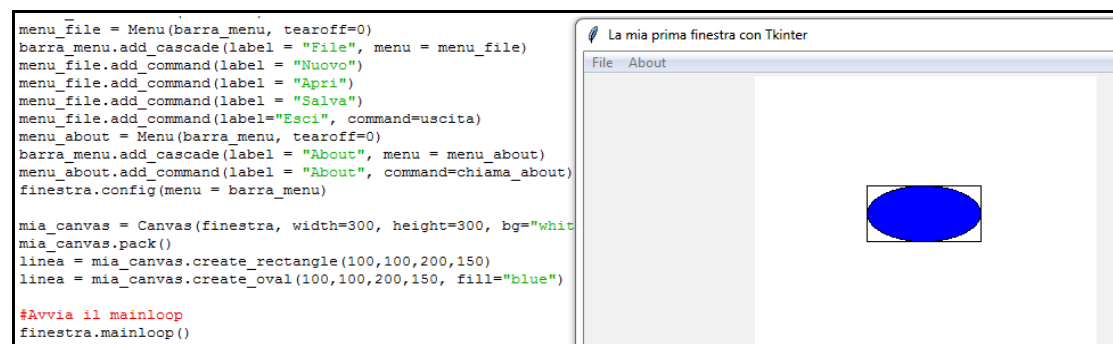
Disegnare un **rettangolo** è altrettanto facile: a `create_line` sostituiamo `create_rectangle` e inseriamo le coordinate del vertice in alto a sinistra e di quello in basso a destra:

```
linea = mia_canvas.create_rectangle(100,100,200,150)
```

Il primo vertice è, infatti, a 100,100 e il secondo è a 200,150. Inserendo una distanza uguale per lunghezza e altezza, anziché un rettangolo avremo un **quadrato**.

Con la stessa tecnica possiamo costruire **cerchi e ovali** a partire dal comando `create_oval`:

```
linea = mia_canvas.create_oval(100,100,200,150, fill="blue")
```



Nella documentazione trovate altri attributi come ad esempio *fill*, per riempire il rettangolo con un colore particolare.

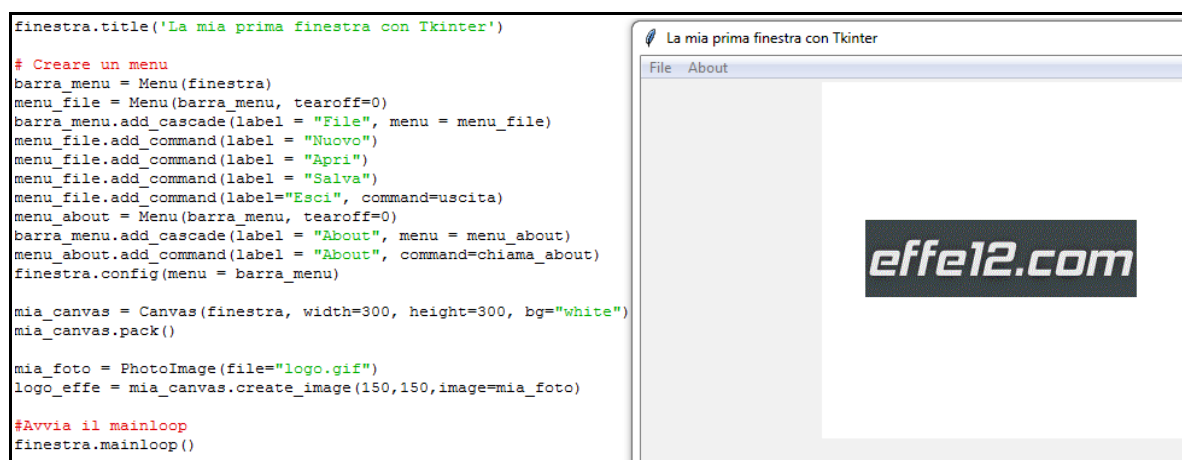
VISUALIZZARE IMMAGINI NELLA CANVAS

Per visualizzare un'immagine nella Canvas, dobbiamo prima caricare il file immagine (associandolo ad una variabile), quindi invocare il metodo `create_image` di un oggetto Canvas. Tale metodo prende tre parametri: i primi due si riferiscono alle coordinate X Y che avrà il centro dell'immagine, mentre il terzo (*image*) è la variabile alla quale abbiamo associato l'immagine.

[NOTA: i nomi dei file immagine utilizzati negli esempi (come “logo.gif” o “newlogo.gif”) sono, ovviamente, puramente indicativi, per cui potrete fare dei test con qualsiasi immagine vogliate, a patto di utilizzare immagini con estensione GIF o PPM/PGM e di specificare correttamente il percorso del file immagine, come indicato più avanti nel tutorial.]

Avremo quindi:

```
mia_foto = PhotoImage(file="logo.gif")
logo_effe = mia_canvas.create_image(150,150,image=mia_foto)
```



L'oggetto immagine lo definiamo come ***PhotoImage(file = percorso)***, che nel mio caso ho inserito **nella stessa cartella del file dello script**. Inserendo il file immagine in un'altra cartella, si dovrà specificare un **path assoluto** (dalla root del sistema) o **relativo** (se il file immagine si troverà in una sottocartella, a partire dalla posizione nella quale si trova il file dello script).

Purtroppo, il comando ***PhotoImage*** può leggere solo **GIF** o **PPM/PGM**.

Facendo partire il programma otterremo un **messaggio di errore**: troppo presto per creare l'immagine. Dovremo spostare ***Photoimage*** sotto la creazione della finestra, con ***Tk()***, ovvero dopo la creazione della finestra. Fatto questo, potremo avviare lo script, osservando l'immagine scelta al centro della **Canvas**.

IMMAGINI, EVENTI E AZIONI: CAMBIARE IMMAGINE A RUNTIME

Esaminiamo la prima interazione.

Creiamo una nuova immagine per il nostro logo; poi, aggiungiamo un bottone che, se cliccato, chiamerà la funzione **cambia_logo**, per cambiare appunto l'immagine visualizzata all'interno della Canvas. Dovremo quindi **cambiare l'attributo image** di *logo_effé* da *mia_foto* a *mia_foto2*.

Il comando è **“*canvas.itemconfig(oggetto, attributo)*”**; i due parametri saranno, in questo caso, *logo_effé* e *image=mia_foto2*:

```
mia_foto2 = PhotoImage(file="newlogo.gif")
def cambia_logo():
    mia_canvas.itemconfig(logo_effé, image=mia_foto2)
bottone = Button(finestra, text="Cambia il logo",
command=cambia_logo).pack()
```

Al click sul pulsante, il logo cambierà, proprio come previsto:



Nella pagina seguente, **il codice completo dello script** prodotto in questo tutorial.

```
from tkinter import *

def cambia_logo():
    mia_canvas.itemconfig(logo_effe, image=mia_foto2)

def chiama_about():
    messagebox.showinfo(title="About", message="La mia finestra 1.0")

def uscita():
    risposta = messagebox.askokcancel(title="Uscita", message="Vuoi davvero uscire?")
    if risposta:
        finestra.destroy()

#Crea la finestra
finestra = Tk()
finestra.geometry('600x600+1000+300')
finestra.title('La mia prima finestra con Tkinter')
mia_foto = PhotoImage(file="logo.gif")
mia_foto2 = PhotoImage(file="newlogo.gif")
bottone = Button(finestra, text="Cambia il logo",
command=cambia_logo).pack()

#Crea la canvas
mia_canvas = Canvas(finestra, width=300, height=300, bg="white")
logo_effe = mia_canvas.create_image(150,150, image=mia_foto)
mia_canvas.pack()

#Crea il menu
barra_menu = Menu(finestra)
menu_file = Menu(barra_menu, tearoff = 0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label="Nuovo")
menu_file.add_command(label="Apri")
menu_file.add_command(label="Salva")
menu_file.add_command(label="Esci", command=uscita)
menu_about = Menu(barra_menu, tearoff = 0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label="About", command=chiama_about)
```

```
finestra.config(menu=barra_menu)
```

```
#Avvia il mainloop
```

```
finestra.mainloop()
```

* * *

Tkinter – Tutorial 10

Un esempio completo

In questo decimo tutorial su **Tkinter** vedremo come mettere insieme le cose viste finora.

Il codice, riportato dopo la seguente descrizione bene, è comunque ben commentato, per descrivere passo dopo passo le varie operazioni.

Per prima cosa ho creato una **Toolbar** sulla sinistra e una **Canvas** a destra.

Scegliendo tra “*quadrato*” e “*cerchio*” e cliccando su due punti nella **Canvas**, il programma disegnerà la figura. All'interno del codice troverete il commento che spiega in dettaglio la funzione. Si tratta, in sostanza, di **prendere le coordinate del mouse** dai due click, salvarle in un array (un vettore di elementi) e passare i suoi valori al metodo che disegna il rettangolo o l'ovale.

Attraverso il pulsante che regola il **colore di sfondo** potrete chiamare la finestra che vi permetterà di scegliere, appunto, il colore di sfondo (**bg**) della Canvas. Per fare questa operazione è bastato scrivere tre righe di codice.

Cambialogo è un **Checkbutton** che permette di scegliere tra la versione grigia e quella rossa di un logo; ovviamente, come per il tutorial precedente, i nomi dei file immagine utilizzati in questo tutorial sono puramente indicativi, per cui potrete fare dei test con qualsiasi immagine vogliate, a patto di utilizzare immagini con estensione GIF o PPM/PGM e di specificare correttamente il percorso del file immagine.

Infine, lo **Slider** permette di cambiare al volo le dimensioni della **Canvas** (qui abbiamo addirittura una sola riga di codice).

“**Cancella tutto**” ripristina la situazione di partenza.



Ecco il codice:

```
#importa la libreria
from tkinter import *

#inizializza delle variabili da assegnare agli elementi della schermata
gestisci_disegno = 0
array_disegno = []
tipo_figura = "cerchio"
colore_sfondo = "#FFFFFF"

#Questa funzione prende la posizione del mouse al primo click
(gestisci_disegno=0) e lo salva in un array

#Al secondo click aggiunge la posizione del mouse all'array e infine passa
i 4 valori alla funzione che

#disegna il cerchio o il quadrato. Poi imposta gestisci_disegno a 0 per
permettere nuovi disegni
def callback(event):
    global gestisci_disegno, array_disegno, tipo_figura
    if gestisci_disegno == 0:
        array_disegno = [event.x, event.y]
        mia_canvas.create_line(event.x, event.y, event.x+1, event.y+1)
```

```
        gestisci_disegno+= 1
    elif gestisci_disegno == 1 and tipo_figura == "quadrato":
        array_disegno += [event.x, event.y]
        mia_canvas.create_rectangle(array_disegno[0], array_disegno[1],
array_disegno[2], array_disegno[3])
        gestisci_disegno = 0
    elif gestisci_disegno == 1 and tipo_figura == "cerchio":
        array_disegno += [event.x, event.y]
        mia_canvas.create_oval(array_disegno[0], array_disegno[1],
array_disegno[2], array_disegno[3])
        gestisci_disegno = 0
```

#La seguente funzione imposta i parametri ai valori iniziali, reimposta il colore a bianco e lo slider a 250.

```
def cancella():
    global array_disegno, gestisci_disegno, logo_effe
    gestisci_disegno = 0
    array_disegno = []
    mia_canvas.delete("all")
    mia_canvas.configure(bg="#FFFFFF")
    slider.set(250)
    logo_effe = mia_canvas.create_image(250,470, image=logo)
```

#Questa funzione dice al programma che abbiamo selezionato il radiobutton cerchio

```
def disegna_cerchio():
    global tipo_figura
    tipo_figura = "cerchio"
```

#Questa funzione dice al programma che abbiamo selezionato il radiobutton quadrato

```
def disegna_quadrato():
    global tipo_figura
    tipo_figura = "quadrato"
```

#Questa funzione chiama la funzione askcolor che restituisce una tripletta RGB e una stringa esadecimale col colore selezionato. Usiamo la stringa per impostare il colore di sfondo

```
def colore():
    (triple, hexstr) = colorchooser.askcolor()
    mia_canvas.configure(bg=hexstr)
```

```
#Questa funzione prende il valore del checkbutton e lo usa per cambiare
l'immagine di logo_effe
def logo_switch():
    if logo_switcher.get():
        mia_canvas.itemconfig(logo_effe, image=logo2)
    else:
        mia_canvas.itemconfig(logo_effe, image=logo)

#Questa funzione riceve il valore dello slider e lo usa per cambiare la
larghezza della canvas
def slider_func(valore):
    mia_canvas.configure(width=int(valore)*2)

#Questa funzione genera un messagebox con l'about del programma
def chiama_about():
    messagebox.showinfo(title="About", message="Il mio primo programma
completo con Tkinter!")

#Questa funzione chiede la conferma per uscire dal programma
def uscita():
    risposta = messagebox.askokcancel(title="Uscita", message="Vuoi
davvero uscire?")
    if risposta:
        finestra.destroy()

#Crea la finestra, la imposta a 900x600 e la visualizza nella posizione
1000,300
finestra = Tk()
finestra.geometry('900x600+1000+300')
finestra.title('La mia prima finestra funzionante con Tkinter')

#Carica le due immagini del logo e la variabile che gestisce il checkbox
#NON DIMENTICARE DI METTERE LOGO.GIF E LOGO2.GIF NELLA STESSA CARTELLA DEL
PROGRAMMA, OPPURE DI CAMBIARE IL PATH PER FILE
logo = PhotoImage(file="logo.gif")
logo2 = PhotoImage(file="logo2.gif")
logo_switcher = BooleanVar()

#Crea il frame della sidebar e tutti gli elementi contenuti: il pulsante
cancella, i due radiobutton, il pulsante per scegliere il colore,
#il checkbox per cambiare il logo e lo slider
sidebar = Frame(finestra)
```

```
b_cancella = Button(sidebar, text="Cancella tutto",
command=cancella).pack(pady=30)

radio_quadrato = Radiobutton(sidebar, text = "Quadrato",
command=disegna_quadrato, value=1).pack()

radio_cerchio = Radiobutton(sidebar, text = "Cerchio",
command=disegna_cerchio, value=2).pack()

b_colore = Button(sidebar, text="Colore di sfondo", command=colore).
pack(pady=30)

cambia_logo = Checkbutton(sidebar, text="Cambia il logo", variable =
logo_switcher, command=logo_switch)

cambia_logo.pack(pady=30)

slider = Scale(sidebar, orient = HORIZONTAL, length=200, from_=250,
to=300, command = slider_funct)

slider.pack()

sidebar.pack(side=LEFT, fill=BOTH, expand=1)

#Crea il frame principale e la canvas, poi aggiunge l'immagine col logo
content = Frame(finestra, bg="#333")
mia_canvas = Canvas(content, width=500, height=500, bg=colore_sfondo)
mia_canvas.pack(pady=50)
logo_effe = mia_canvas.create_image(250,470, image=logo)
content.pack(side=LEFT, fill=BOTH, expand=1)

#Associa il click del pulsante sulla canvas alla funzione callback
mia_canvas.bind("<Button-1>", callback)

#Crea il menu, il menu file con la voce uscita e il menu about con la voce
about
barra_menu = Menu(finestra)

menu_file = Menu(barra_menu, tearoff = 0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label="Esci", command=uscita)

menu_about = Menu(barra_menu, tearoff = 0)
barra_menu.add_cascade(label = "About", menu = menu_about)
menu_about.add_command(label="About", command=chiama_about)

#Assegna barra_menu come menu per la finestra
finestra.config(menu=barra_menu)
```

```
#Avvia il mainloop  
finestra.mainloop()
```

* * *

Tkinter – Tutorial 11

Trascinare una figura sulla Canvas

In questo tutorial vedremo **come trascinare una figura sulla Canvas**; ovvero: cliccando in un punto qualunque della **Canvas**, selezioneremo il nostro oggetto, poi lo muoveremo **tenendo premuto il mouse**.

Vediamo come fare.

Per prima cosa, creiamo una variabile booleana "**cliccato**" e la impostiamo a *False*.

Usiamo quindi **bind** per associare il click del bottone alla funzione **callback**.

Callback controllerà lo stato di **cliccato** e, se *False*, lo imposterà a *True*, e viceversa; inoltre, colorerà il quadrato per farci capire, visivamente, se è stato selezionato o no.

Cliccando sulla Canvas, infatti, il quadrato diventa rosso, e cliccando una seconda volta ritorna blu; inoltre, se il quadrato è selezionato, la funzione memorizza le coordinate x e y del click nelle variabili globali `pre_x` e `pre_y`. Questo ci servirà per il trascinamento.

Il secondo **binding** che facciamo è tra **b1-Motion** e la funzione **callback2**, che possiamo rinominare, ad esempio, in “*spostamento*”.

B1-Motion serve a tener traccia delle coordinate del mouse mentre lo muoviamo tenendo premuto il tasto sinistro... praticamente, quando stiamo trascinando l'oggetto!

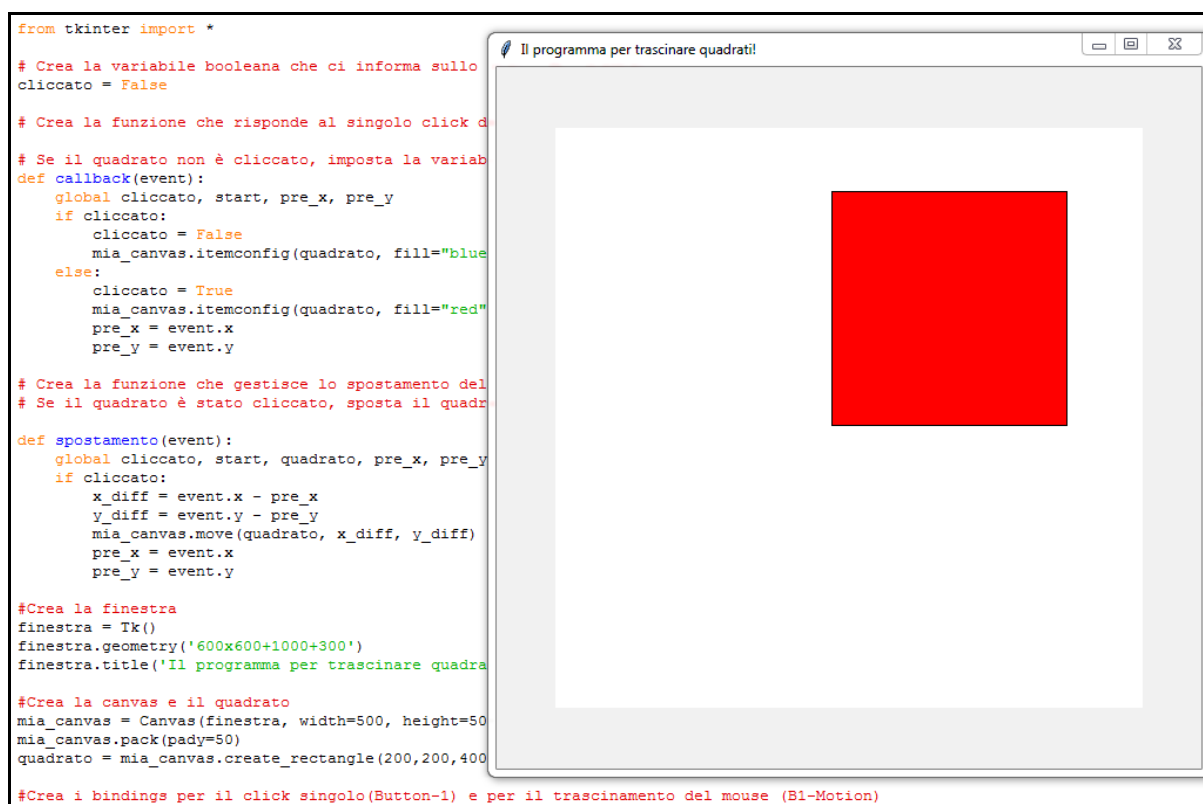
Ecco cosa fa la funzione *spostamento*: se il quadrato è selezionato, calcola la differenza tra la

posizione attuale del mouse e quella del punto in cui avevamo cliccato, e sposta il quadrato tramite il **metodo *move*** della **Canvas**, il quale ci permette di definire l'oggetto che vogliamo spostare e l'**offset di spostamento** sull'asse delle x e delle y.

Poiché, però, non vogliamo spostare il quadrato in ogni istante di un valore grande quanto la distanza tra la posizione del mouse e il punto di click (perché arriveremo in poco tempo a numeri enormi), dobbiamo impostare *pre_x* e *pre_y* all'ultima posizione del mouse; così, in ogni istante, *x_diff* calcolerà la differenza tra la posizione attuale del mouse e quella al termine del ciclo precedente, ottenendo numeri molto piccoli che sono l'ideale per lo spostamento del quadrato.

Proviamo a tener traccia di questi spostamenti scrivendo ***print(x_diff, y_diff)***. Nella console appariranno numeri molto piccoli, che sono proprio i valori di spostamento della figura.

Infine, possiamo **impostare i valori dell'offset** di x o di y a 0, in modo da **vincolare il movimento ad un solo asse**.



Qui di seguito, **il codice completo dello script** prodotto in questo tutorial:

```
from tkinter import *

# Crea la variabile booleana che ci informa sullo stato del quadrato
cliccato = False

# Crea la funzione che risponde al singolo click del mouse

# Se il quadrato non è cliccato, imposta la variabile cliccato a True, e
# viceversa. Inoltre, salva le coordinate del punto del click in pre_x e
# pre_y
def callback(event):
    global cliccato, start, pre_x, pre_y
    if cliccato:
        cliccato = False
        mia_canvas.itemconfig(quadrato, fill="blue")
    else:
        cliccato = True
        mia_canvas.itemconfig(quadrato, fill="red")
        pre_x = event.x
        pre_y = event.y

# Crea la funzione che gestisce lo spostamento del quadrato.
# Se il quadrato è stato cliccato, sposta il quadrato di un offset uguale
# a x_diff, y_diff. Questi due valori sono la differenza tra la posizione
# attuale del mouse e quella immediatamente precedente. Se la funzione è
# stata appena chiamata, la differenza è tra la posizione del mouse e il
# punto di click

def spostamento(event):
    global cliccato, start, quadrato, pre_x, pre_y
    if cliccato:
        x_diff = event.x - pre_x
        y_diff = event.y - pre_y
        mia_canvas.move(quadrato, x_diff, y_diff)
        pre_x = event.x
        pre_y = event.y

#Crea la finestra
finestra = Tk()
```

```
finestra.geometry('600x600+1000+300')
finestra.title('Il programma per trascinare quadrati!')

#Crea la canvas e il quadrato
mia_canvas = Canvas(finestra, width=500, height=500, bg="white")
mia_canvas.pack(pady=50)
quadrato = mia_canvas.create_rectangle(200,200,400,400, fill="blue")

#Crea i bindings per il click singolo(Button-1) e per il trascinamento del
mouse (B1-Motion)
mia_canvas.bind("<Button-1>", callback)
mia_canvas.bind("<B1-Motion>", spostamento)

#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter – Tutorial 12

Creare un file txt contenente gli elementi di una Listbox

In questo tutorial vedremo come creare un file txt contenente gli elementi di una listbox.

Il codice dell'esempio è riportato per intero dopo una breve spiegazione delle sue componenti; in ogni caso, il codice è ampiamente commentato, con le varie operazioni descritte passo per passo.

Iniziamo importando **Tkinter** e creando la solita finestra, questa volta di dimensioni più contenute.

Creiamo l'oggetto **Listbox** e posizioniamolo, in una griglia immaginaria, alla riga 0, colonna 0; praticamente, in alto a sinistra.

Aggiungiamo ora la **casella di testo** che ci servirà a inserire gli elementi nella **Listbox** assegnando la variabile **contenuto** al testo che vi scriveremo dentro. Posizioniamola alla riga uno, ovvero sotto la **Listbox**.

Andiamo poi a dichiarare **contenuto** come **StringVar**.

Facciamo partire il **mainloop** per vedere cosa abbiamo fatto finora: al momento, abbiamo una lista vuota e una **entry**.

Creiamo, dunque, il **pulsante** che ci permetterà di aggiungere il testo scritto nella casella alla lista.

La funzione associata sarà **aggiungi_item**. Posizioniamo il pulsante accanto alla casella.

Il metodo *insert*, con indice *end*, ci permetterà di aggiungere in fondo alla lista il valore di contenuto, ovvero il testo della casella, ricavato con *contenuto.get*.

Andiamo infine ad aggiungere il pulsante che ci permette di **esportare il contenuto della listbox** in un file *txt*. Posizioniamo il pulsante sotto tutto il resto.

La funzione *crea_file* prende l'elenco di tutti gli item contenuti nella *Listbox* attraverso il metodo *get(0,END)* e li salva in lista.

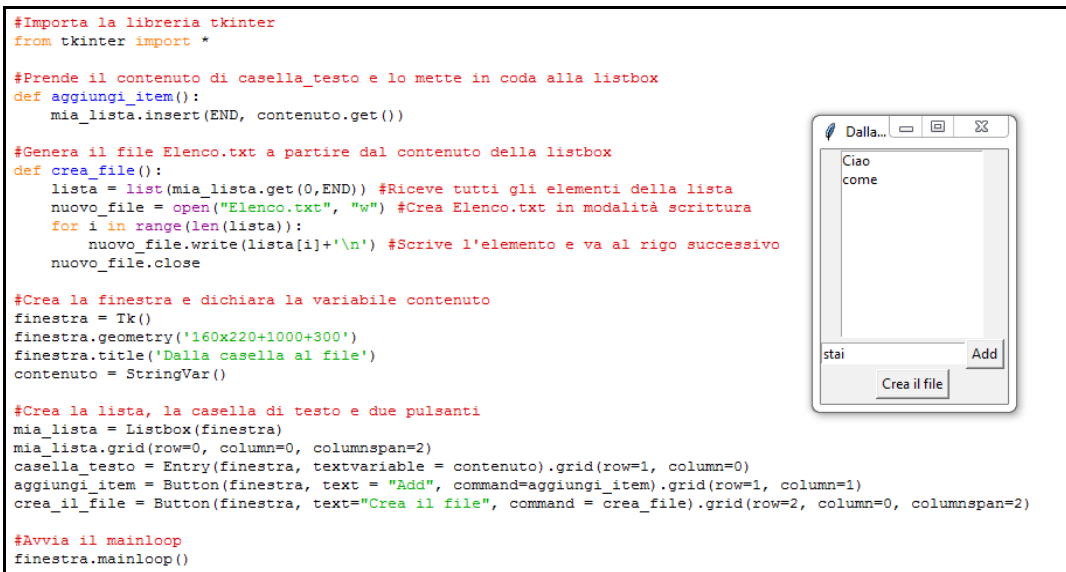
Attraverso *open* verrà generato il file che, non avendo percorso, sarà salvato **nella stessa cartella del file .py dello script**. La *w* indica il fatto che il file è aperto in **modalità scrittura**.

Il ciclo *for* scorrerà tutti gli **elementi della lista** e li **scriverà su un file**. Dopo ogni elemento, il simbolo *\n* indicherà il **ritorno a capo**.

Infine, **chiudiamo il file**.

Proviamo il tutto, ottenendo in effetti un nuovo file “*Elenco.txt*”.

Possiamo sovrascrivere il file semplicemente aggiungendo elementi alla **Listbox** e cliccando nuovamente il pulsante “*Crea il file*”.



Qui di seguito, **il codice completo dello script** prodotto in questo tutorial:

```
#Importa la libreria tkinter
from tkinter import *

#Prende il contenuto di casella_testo e lo mette in coda alla listbox
def aggiungi_item():
    mia_lista.insert(END, contenuto.get())

#Genera il file Elenco.txt a partire dal contenuto della listbox
def crea_file():
    lista = list(mia_lista.get(0,END)) #Riceve tutti gli elementi della
    lista
    nuovo_file = open("Elenco.txt", "w") #Crea Elenco.txt in modalità
    scrittura
    for i in range(len(lista)):
        nuovo_file.write(lista[i]+'\\n') #Scrive l'elemento e va al
        rigo successivo
    nuovo_file.close

#Crea la finestra e dichiara la variabile contenuto
finestra = Tk()
finestra.geometry('160x220+1000+300')
finestra.title('Dalla casella al file')
contenuto = StringVar()

#Crea la lista, la casella di testo e due pulsanti
mia_lista = Listbox(finestra)
mia_lista.grid(row=0, column=0, columnspan=2)
casella_testo = Entry(finestra, textvariable = contenuto).grid(row=1,
column=0)
aggiungi_item = Button(finestra, text = "Add",
command=aggiungi_item).grid(row=1, column=1)
crea_il_file = Button(finestra, text="Crea il file", command =
crea_file).grid(row=2, column=0, columnspan=2)

#Avvia il mainloop
finestra.mainloop()
```

* * *

Tkinter – Tutorial 13

Dati da file a Listbox, da Listbox a file

In questo tutorial vedremo come **leggere i dati** da un file, trasferirli a una **Listbox**, passare gli elementi a una seconda **Listbox** e salvare il tutto su **un nuovo file**.

Anche in questo **tutorial**, come nel precedente, il codice dell'esempio è riportato per intero dopo una breve spiegazione delle sue componenti; in ogni caso, il codice è ampiamente commentato, con le varie operazioni descritte passo per passo.

In questo tutorial, la scelta dei **percorsi per la lettura e la scrittura** dei file è affidata alle **finestre di dialogo**, eliminando così il problema degli errori di battitura.

Bene, cominciamo da una struttura di base, come nei tutorial precedenti: dopo aver importato la libreria **Tkinter** abbiamo creato una finestra 280x160 e un menù la cui unica voce funzionante, finora è quella che ci chiede la conferma al momento dell'uscita.

Definiamo, allora, le restanti funzioni associate alle voci di menù **apri** e **salva**, senza dare ancora istruzioni specifiche.

Aggiungiamo, poi, gli **elementi della finestra**: la **Listbox** di sinistra, i due pulsanti per spostare gli elementi da una **Listbox** all'altra, e la lista di destra.

Per il **layout** usiamo il sistema della **griglia**. Nella prima colonna mettiamo la prima **Listbox**, con estensione di due righe, nella seconda i due pulsanti, una riga per ciascuno, e nella terza la seconda **Listbox**, sempre con estensione di due righe. Associamo al primo pulsante la funzione **aggiungi_item** e al secondo la funzione **rimuovi_item**.

Anche questa volta, definiamo delle **funzioni “segnaposto”** per non farci restituire errore dal programma, e vediamo il risultato.

La disposizione è corretta, ma né i pulsanti né le voci di menù **Apri** e **Salva** fanno ancora qualcosa, quindi passiamo a **scrivere il codice delle funzioni**.

Il comando ***filedialog.askopenfile()*** ci permette di chiamare la finestra di dialogo **Apri**.

Aggiungiamo ***.name*** per assegnare alla variabile soltanto il nome del file aperto e non le altre informazioni che ***askopenfile*** ci restituisce. Vediamo, infatti, che la console ci restituirà, e salverà nella variabile ***filename***, il percorso col nome esatto del file.

Adesso chiediamo al programma di leggere il contenuto del file, riga per riga. Poiché ***readlines*** restituisce una lista di stringhe che terminano con ***"\n"***, aggiungiamo un ciclo ***loop*** che, attraverso il comando ***rstrip("\n")***, elimina il tag ***newline*** dalla coda di ogni stringa.

Approfittiamo del **ciclo for** per passare individualmente **ogni elemento del file alla Listbox**.

Il file viene correttamente caricato e ogni elemento viene assegnato alla lista di sinistra.

Per assicurarci della necessità del comando ***rstrip***, proviamo a stampare a console il risultato "grezzo" della lettura con ***readlines***.

Ogni stringa termina con l'elemento ***\n***, che indica "a capo".

Passiamo adesso alla **funzione “aggiungi_item”**. Scriviamo innanzitutto un **if** che ci garantisca che il passaggio di elementi avvenga con una lista contenente almeno un elemento.

La funzione ***aggiungi_item*** prende l'id dell'elemento selezionato nella lista di sinistra, cerca il valore associato a quell'id e lo trascrive nella lista di destra. L'id della selezione viene restituito da ***prima_lista.curselection***.

Il **metodo *.get*** prende l'id ricevuto da ***curselection*** e restituisce la stringa ad esso associata. Infine, questo valore viene **aggiunto in coda alla seconda Listbox**.

Il pulsante **aggiungi**, grazie all'**if**, non restituisce errore se premuto con una lista vuota, ma funziona egregiamente una volta ottenuta la selezione degli elementi.

La **funzione *rimuovi_item*** è un semplice comando **delete** che riceve come argomento l'id della selezione attuale, ancora una volta ottenuto con **curselection**.

Per finire, la **funzione *salva_file***. Analogamente alla funzione di apertura, riceviamo il **percorso del file** attraverso un **metodo di filedialog**.

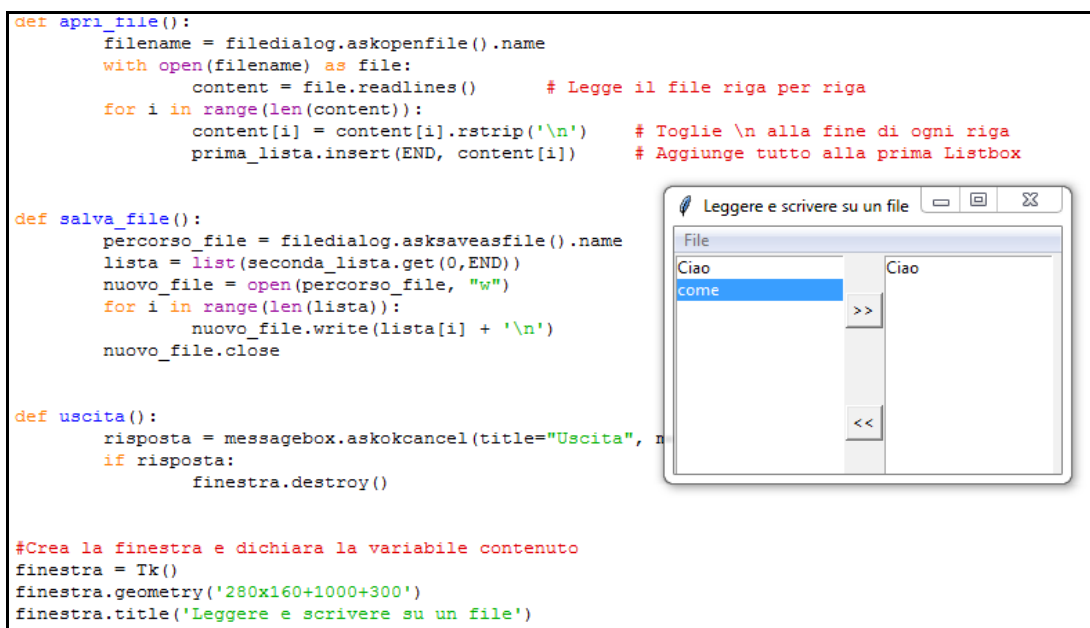
Utilizziamo una lista per ricevere tutti i valori della seconda listBox attraverso il **metodo *get***.

Creiamo poi un **nuovo file** in modalità scrittura, quindi **"w"**, passando come percorso quello ottenuto da **asksaveasfile().name**.

Attraverso un ciclo **for** prendo ogni elemento della lista e lo scrivo sul file, riga per riga. Per avere un solo elemento per riga aggiungo alla fine di ogni stringa il **tag newline '\n'**.

Infine, **chiudo il file**.

In definitiva: ricevo l'elenco da *primoelenco.txt*, trasferisco gli elementi (*items*) sulla seconda lista, scelgo il percorso di destinazione e salvo il nuovo file.



Nella pagina seguente, **il codice completo dello script** prodotto in questo (ultimo) tutorial.

```
#Importa la libreria tkinter
from tkinter import *

def aggiungi_item():
    if prima_lista.size() > 0:    # Il pulsante funziona solo se c'e'
almeno un elemento nella lista a sx
        seconda_lista.insert(END,
prima_lista.get(prima_lista.curselection()))

def rimuovi_item():
    seconda_lista.delete(seconda_lista.curselection())

def apri_file():
    filename = filedialog.askopenfile().name
    with open(filename) as file:
        content = file.readlines()    # Legge il file riga per riga
        for i in range(len(content)):
            content[i] = content[i].rstrip('\n')    # Toglie \n alla fine
di ogni riga
            prima_lista.insert(END, content[i]) # Aggiunge tutto alla
prima Listbox

def salva_file():
    percorso_file = filedialog.asksaveasfile().name
    lista = list(seconda_lista.get(0,END))
    nuovo_file = open(percorso_file, "w")
    for i in range(len(lista)):
        nuovo_file.write(lista[i] + '\n')
    nuovo_file.close

def uscita():
    risposta = messagebox.askokcancel(title="Uscita", message="Vuoi
davvero uscire?")
    if risposta:
        finestra.destroy()
```

```
#Crea la finestra e dichiara la variabile contenuto
finestra = Tk()
finestra.geometry('280x160+1000+300')
finestra.title('Leggere e scrivere su un file')

#Crea le liste e i pulsanti per il passaggio degli item
prima_lista = Listbox(finestra)
prima_lista.grid(row=0, column=0, rowspan=2)
aggiungi_item = Button(finestra, text = ">>", command =
aggiungi_item).grid(row=0, column=1)
rimuovi_item = Button(finestra, text = "<<", command =
rimuovi_item).grid(row=1, column=1)
seconda_lista = Listbox(finestra)
seconda_lista.grid(row=0, column=2, rowspan=2)

#Crea il menu
barra_menu = Menu(finestra)

menu_file = Menu(barra_menu, tearoff = 0)
barra_menu.add_cascade(label = "File", menu = menu_file)
menu_file.add_command(label="Apri", command=apri_file)
menu_file.add_command(label="Salva", command=salva_file)
menu_file.add_command(label="Esci", command=uscita)

#Assegna barra_menu come menu per la finestra
finestra.config(menu=barra_menu)

#Avvia il mainloop
finestra.mainloop
```

```
*      *      *
*      *      *
```