

CustomTkinter (autore: Vittorio Albertoni)

Premessa

Scopo di un pacchetto grafico è generalmente quello di programmare software dotato della così detta Graphical User Interface (GUI), cioè di un'interfaccia grafica che ci consenta di interagire con il computer attraverso la visualizzazione di ciò che facciamo, utilizzando un puntatore grafico (mouse) per scegliere cosa fare o, ove possibile, addirittura toccando lo schermo con un dito.

Per arricchire di tutto questo i programmi, o, meglio, gli script Python esistono almeno quattro pacchetti che ci mettono a disposizione altrettanti framework: PyQt, wxPython, PyGTK e Tkinter.

I primi tre si rifanno rispettivamente ai toolkit e alle librerie Qt, wxWidgets e GTK.

Tkinter deriva dalle librerie Tcl/Tk ed è da sempre la libreria standard per Python.

Tra tutte quelle citate, Tkinter è sicuramente la libreria più facile da usare, la più leggera, è molto stabile ma ha il grosso difetto di avere una grafica ferma agli anni 90 del secolo scorso.

Un primo tentativo per ottenere una grafica più moderna è stato fatto con il modulo ttk che fornisce risultati grafici non poi così spartani come quelli della versione normale.

Sull'uso di Tkinter e, volendo, del modulo ttk, ho rilasciato un manuale in formato PDF, intitolato tkinter, allegato all'articolo «Grafica con Python» che ho pubblicato nel maggio 2018 sul mio blog all'indirizzo www.vittal.it.

Il problema della qualità grafica delle interfacce Tkinter pare sia stato definitivamente risolto grazie ad una intuizione dello sviluppatore tedesco Tom Schimansky, che ha incominciato a lavorarci verso la metà del 2021 e da subito la comunità open-source di Python lo ha preso sul serio e, dal 2025, quello che è stato chiamato CustomTkinter, è un progetto maturo ed è diventato il ponte perfetto tra la semplicità del vecchio Python e le esigenze estetiche del design software moderno.

L'intuizione di Tom Schimansky è stata di creare un wrapper sopra Tkinter, prendendone i componenti e ridisegnandoli via codice, racchiudendoli in una nuova libreria.

I fattori di successo della nuova libreria CustomTkinter sono:

- estrema facilità di apprendimento e di uso (per chi già conosce Tkinter non c'è nulla da apprendere, salvo qualche variante nella nomenclatura dei widget),
- estetica all'altezza delle moderne app che vediamo su Windows 11 o su Mac,
- leggerezza e velocità, ben lontane dalla mole di MB delle varie librerie PyQt, ecc.

Con Tkinter, oltre alla possibilità di creare interfacce grafiche per i nostri programmi, abbiamo la possibilità di utilizzare il tool `turtle` per la grafica con la tartaruga e il widget contenitore `canvas` per disegnare linee e figure geometriche.

La vera innovazione apportata da CustomTkinter riguarda tuttavia l'interfaccia grafica per i programmi e questo manualetto è limitato a questo argomento.

Potremmo lavorare con `turtle` anche utilizzando il modulo CustomTkinter ma dovremmo affrontare inutili complicazioni per ottenere un risultato grafico identico a quello che otterremmo utilizzando il modulo Tkinter.

Per quanto riguarda `canvas` possiamo avvalerci del corrispondente widget CustomTkinter utilizzando i comandi previsti da Tkinter.

A chi non basti questo manualetto suggerisco la documentazione ufficiale all'indirizzo <https://customtkinter.tomschimansky.com/documentation/>.

Indice

1	Installazione	3
2	La GUI (Graphical User Interface)	3
3	I widget di CustomTkinter	6
3.1	Widget contenitori	6
3.2	Widget di cui non si può fare a meno per costruire una GUI	7
3.3	Widget CTkTextbox	9
4	Dove non arriva CustomTkinter	10
4.1	Modulo Menu di Tkinter	10
4.2	Modulo filedialog di Tkinter	11
5	La geometria della GUI	11
5.1	Geometria con il metodo pack	12
5.2	Geometria con il metodo grid	13
5.3	Geometria con il metodo place	15
6	Alcuni esempi di integrazione con il linguaggio Python	18
6.1	Potenze e radici	18
6.2	Calcolo del fattoriale di un numero	19
6.3	Un piccolo editor di testo	20
6.4	Master Mind	21
7	Considerazioni conclusive	23

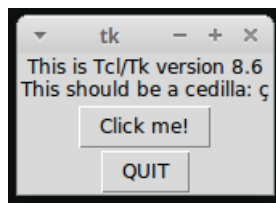
1 Installazione

Prerequisito per il funzionamento del modulo CustomTkinter è che sul computer, ovviamente oltre a Python (ormai parliamo di Python3), sia installato il modulo Tkinter.

Quando installiamo Python su Windows automaticamente installiamo anche Tkinter, avendo l'accortezza di fare questa scelta nel lanciare l'installatore di Python su Windows, che contiene anche la IDLE, che utilizza Tkinter.

La IDLE è un'interfaccia grafica molto utile per scrivere programmi con il linguaggio Python e, nei sistemi operativi Linux e OS X, non si installa automaticamente. Dal momento che essa è molto utile è bene installarla a parte e, in tal modo, si installa automaticamente anche Tkinter. Per farlo in Linux basta ricorrere all'installatore dei programmi in quanto si trova sicuramente nel repository. Per OS X questo è l'indirizzo: <https://www.python.org/download/mac/tcltk/>.

Per verificare che tkinter sia installato e funzioni basta dare il seguente comando a terminale (prompt dei comandi in Windows): `python3 -m tkinter`. Se tutto funziona vediamo comparire questa piccola finestra



Se è tutto posto procediamo con l'installazione del modulo CustomTkinter con

```
pip install customtkinter
```

Per chi lavora su una distro Linux installata dopo il 2024 può essere che si tratti di una distro che ha adottato la PEP 668, allo scopo di eliminare conflitti che possano insorgere tra il gestore di pacchetti del sistema operativo (come apt o dnf) e strumenti di gestione di pacchetti specifici Python (come pip).

In questo caso è inibito l'uso di pip e i moduli python si devono installare con l'installatore del sistema operativo, ma può essere che nel repository non ci sia il modulo customtkinter.

Per rimediare, ferma la pratica, quando possibile, di preferire l'uso dell'installatore del sistema operativo, se pip non è installato lo installiamo con

```
sudo apt (o dnf) install python3-pip
```

ed eludiamo la PEP 668 con il comando a terminale

```
sudo rm -rf /usr/lib/python3.xx/EXTERNALLY-MANAGED
```

e, finalmente potremo dare il comando

```
pip install customtkinter
```

senza problemi.

2 La GUI (Graphical User Interface)

Per generare la GUI dobbiamo innanzi tutto importare CustomTkinter. La più bella istruzione per farlo è

```
from customtkinter import *
```

con la quale diciamo a Python di importare tutti i componenti di CustomTkinter.

Dobbiamo quindi creare l'oggetto genitore della GUI, che viene generalmente chiamato radice, con l'istruzione

```
r = CTk()
```

dove r sta per un qualsiasi identificatore (c'è chi preferisce root o radice) che richiami in qualche modo il fatto che questo oggetto è la radice, il genitore di tutto.

Questo oggetto altro non è che una finestra; è ovviamente un contenitore ed è mettendovi dentro gli widget che costruiamo la GUI.

E' dotato, tra gli altri, di due importanti metodi:

```
title('...')
```

che serve per far apparire la stringa che inseriamo tra apici nella cornice in alto della finestra e diventerà il titolo dell'applicazione;

```
configure(<opzione> = <valore>, ... )
```

con cui possiamo configurare a nostro piacimento la finestra.

Le principali opzioni a disposizione sono:

- . la larghezza della finestra, esprimibile con `width = <numero_pixel>`

- . l'altezza della finestra, esprimibile con `height = <numero_pixel>`

- . il colore di fondo della finestra, esprimibile con `fg_color = <colore>`.

Il contenitore radice è una top-level window ed è solo lì, come vedremo, che possiamo inserire una barra del menu.

Essendo un contenitore esso può contenere direttamente gli widget che servono a dar vita alla GUI.

Tra questi, come vedremo, altri widget contenitori, che possono servire per differenziare la geometria di parti della GUI, la loro colorazione e la loro natura.

Come top-level window è anche quella deputata a mostrare tutta l'applicazione e per ottenere questo, alla fine dello script dell'applicazione stessa, dobbiamo richiamare il metodo `mainloop()`.

Nella cornice alta della finestra compaiono pure i soliti simboli per minimizzarla, ingrandirla e chiuderla.

Esempio:

Il seguente script

```
#!/usr/bin/python3
from customtkinter import *
r = CTk()
r.title('FINESTRA')
r.configure(width = 250, height = 100, bg = 'violet')
r.mainloop()
```

crea questa finestra



che possiamo chiudere cliccando sulla x in alto a destra.

Se confrontiamo questa finestra, pur priva di altri contenuti qualificanti, con una finestra prodotta con Tkinter notiamo che siamo in presenza di qualche cosa di meglio.

Quanto al colore, CustomTkinter eredita la ricchezza di colori di Tkinter e la tabella riprodotta nella pagina seguente indica tutti i nomi della varianti.

Nomi simbolici dei colori in Tkinter

alice blue – AliceBlue – antique white – AntiqueWhite – AntiqueWhite1 ... AntiqueWhite4 – aqua – aquamarine – aquamarine1 ... aquamarine4 – azure – azure1 ... azure4
beige – bisque – bisque1 ... bisque4 – black – blanched almond – BlanchedAlmond – blue – blue violet – blue1 – blue2 – blue3 – blue4 – BlueViolet – brown – brown1 ... brown4 – burlywood – burlywood1 ... burlywood4
cadet blue – CadetBlue – CadetBlue1 – CadetBlue2 – CadetBlue3 – CadetBlue4 – chartreuse – chartreuse1 ... chartreuse4 – chocolate – chocolate1 ... chocolate4 – coral – coral1 ... coral4 – cornflower blue – CornflowerBlue – cornsilk – cornsilk1 ... cornsilk4 – crimson – cyan – cyan1 ... cyan4
dark blue – dark cyan – dark goldenrod – dark gray – dark green – dark grey – dark khaki – dark magenta – dark olive green – dark orange – dark orchid – dark red – dark salmon – dark sea green – dark slate blue – dark slate gray – dark slate grey – dark turquoise – dark violet – DarkBlue – DarkCyan – DarkGoldenrod – DarkGoldenrod1 ... DarkGoldenrod4 – DarkGray – DarkGreen – DarkGrey – DarkKhaki – DarkMagenta – DarkOliveGreen – DarkOliveGreen1 ... DarkOliveGreen4 – DarkOrange – DarkOrange1 ... DarkOrange4 – DarkOrchid – DarkOrchid1 ... DarkOrchid4 – DarkRed – DarkSalmon – DarkSeaGreen – DarkSeaGreen1 ... DarkSeaGreen4 – DarkSlateBlue – DarkSlateGray – DarkSlateGray1 ... DarkSlateGray4 – DarkSlateGrey – DarkTurquoise – DarkViolet – deep pink – deep sky blue – DeepPink – DeepPink1 ... DeepPink4 – DeepSkyBlue – DeepSkyBlue1 ... DeepSkyBlue4 – dim gray – dim grey – DimGray – DimGrey – dodger blue – DodgerBlue – DodgerBlue1 ... DodgerBlue4
firebrick – firebrick1 ... firebrick4 – floral white – FloralWhite – forest green – ForestGreen – fuchsia
gainsboro – ghost white – GhostWhite – gold – gold1 ... gold4 – goldenrod – goldenrod1 ... goldenrod4 – gray – gray0 – gray1 ... gray100 – green – green yellow – green1 ... green4 – GreenYellow – grey – grey0 ... grey100
honeydew – honeydew1 ... honeydew4 – hot pink – HotPink – HotPink1 ... HotPink4
indian red – IndianRed – IndianRed1 ... IndianRed4 – indigo – ivory – ivory1 ... ivory4
khaki – khaki1 ... khaki4
lavender – lavender blush – LavenderBlush – LavenderBlush1 ... LavenderBlush4 – lawn green – LawnGreen – lemon chiffon – LemonChiffon – LemonChiffon1 ... LemonChiffon4 – light blue – light coral – light cyan – light goldenrod – light goldenrod yellow – light gray – light green – light grey – light pink – light salmon – light sea green – light sky blue – light slate blue – light slate gray – light slate grey – light steel blue – light yellow – LightBlue – LightBlue1 ... LightBlue4 – LightCoral – LightCyan – LightCyan1 ... LightCyan4 – LightGoldenrod – LightGoldenrod1 ... LightGoldenrod4 – LightGoldenrodYellow – LightGray – LightGreen – LightGrey – LightPink – LightPink1 ... LightPink4 – LightSalmon – LightSalmon1 ... LightSalmon4 – LightSeaGreen – LightSkyBlue – LightSkyBlue1 ... LightSkyBlue4 – LightSlateBlue – LightSlateGray – LightSlateGrey – LightSteelBlue – LightSteelBlue1 ... LightSteelBlue4 – LightYellow – LightYellow1 ... LightYellow4 – lime – lime green – LimeGreen – linen
magenta – magenta1 ... magenta4 – maroon – maroon1 ... maroon4 – medium aquamarine – medium blue – medium orchid – medium purple – medium sea green – medium slate blue – medium spring green – medium turquoise – medium violet red – MediumAquamarine – MediumBlue – MediumOrchid – MediumOrchid1 ... MediumOrchid4 – MediumPurple – MediumPurple1 ... MediumPurple4 – MediumSeaGreen – MediumSlateBlue – MediumSpringGreen – MediumTurquoise – MediumVioletRed – midnight blue – MidnightBlue – mint cream – MintCream – misty rose – MistyRose – MistyRose1 ... MistyRose4 – moccasin
navajo white – NavajoWhite – NavajoWhite1 ... NavajoWhite4 – navy – navy blue – NavyBlue
old lace – OldLace – olive – olive drab – OliveDrab – OliveDrab1 ... OliveDrab4 – orange – orange red – orange1 ... orange4 – OrangeRed – OrangeRed1 ... OrangeRed4 – orchid – orchid1 ... orchid4
pale goldenrod – pale green – pale turquoise – pale violet red – PaleGoldenrod – PaleGreen – PaleGreen1 ... PaleGreen4 – PaleTurquoise – PaleTurquoise1 ... PaleTurquoise4 – PaleVioletRed – PaleVioletRed1 ... PaleVioletRed4 – papaya whip – PapayaWhip – peach puff – PeachPuff – PeachPuff1 ... PeachPuff4 – peru – pink – pink1 ... pink4 – plum – plum1 ... plum4 – powder blue – PowderBlue – purple – purple1 ... purple4
red – red1 ... red4 – rosy brown – RosyBrown – RosyBrown1 ... RosyBrown4 – royal blue – RoyalBlue – RoyalBlue1 ... RoyalBlue4
saddle brown – SaddleBrown – salmon – salmon1 – salmon2 – salmon3 – salmon4 – sandy brown – SandyBrown – sea green – SeaGreen – SeaGreen1 – SeaGreen2 – SeaGreen3 – SeaGreen4 – seashell – seashell1 ... seashell4 – sienna – sienna1 ... sienna4 – silver – sky blue – SkyBlue – SkyBlue1 ... SkyBlue4 – slate blue – slate gray – slate grey – SlateBlue – SlateBlue1 ... SlateBlue4 – SlateGray – SlateGray1 ... SlateGray4 – SlateGrey – snow – snow1 ... snow4 – spring green – SpringGreen – SpringGreen1 ... SpringGreen4 – steel blue – SteelBlue – SteelBlue1 ... SteelBlue4
tan – tan1 ... tan4 – teal – thistle – thistle1 ... thistle4 – tomato – tomato1 ... tomato4 – turquoise – turquoise1 ... turquoise4
violet – violet red – VioletRed – VioletRed1 ... VioletRed4
wheat – wheat1 ... wheat4 – white – white smoke – WhiteSmoke
yellow – yellow green – yellow1 ... yellow4 – YellowGreen

3 I widget di CustomTkinter

Secondo la definizione che ne dà Wikipedia un widget è un componente grafico dell'interfaccia utente di un programma, che ha lo scopo di facilitare all'utente l'interazione con il programma stesso. Il termine deriva dalla contrazione dei termini "window" e "gadget" ed ha una valenza un tantino dispregiativa, dal momento che un gadget è solitamente una cosa di poco conto e poco seria: probabilmente ciò deriva dal fatto che il termine è stato coniato nell'epoca in cui le finestre con i pulsanti e l'uso del mouse facevano la loro prima apparizione in un mondo in cui l'informatica si riteneva seria solo se praticata su terminali a riga di comando.

In sostanza i widget sono i mattoncini con i quali possiamo costruire una GUI (Graphical User Interface).

Si tratta, anche nel linguaggio informatico, di veri e propri oggetti, dotati di metodi per svolgere determinati compiti.

Ne propongo qui una rassegna, limitandomi a quelli di utilità più ricorrente.

3.1 Widget contenitori

Sono dei recipienti destinati a contenere forme disegnate o altri widget.

Canvas

Il canvas (termine inglese che in italiano significa canovaccio o tela) è un'area rettangolare in cui possiamo mettere di tutto (anche altri widget) ma che è particolarmente adatta per disegnarci sopra, come avviene, appunto, per la tela del pittore.

L'oggetto informatico canvas è infatti dotato di parecchi metodi per disegnare.

La sua costruzione avviene con l'istruzione

```
<identificatore> = CtkCanvas(<genitore>, <opzione> = <valore>, ...)
```

dove

<identificatore> è il nome che diamo al nostro canvas e servirà per richiamarne i metodi;

<genitore> identifica il contenitore in cui viene inserito il canvas;

<opzione> e <valore> servono per definire determinati attributi del nostro canvas.

Le opzioni possibili sono molte, ma ci basta conoscere queste tre fondamentali:

- . la larghezza del canvas, esprimibile con `width = <numero_pixel>`,

- . l'altezza del canvas, esprimibile con `height = <numero_pixel>`,

- . il colore di fondo del canvas, esprimibile con `bg = <colore>`,

dove <colore> è una stringa contenente il nome di un colore desunto dalla tabella riportata nella pagina precedente.

Con

```
c = CtkCanvas(r, width = 150, height = 100, bg='red')
```

costruiamo un canvas chiamato `c`, nella finestra radice `r`, con le dimensioni indicate e colore rosso.

Come avviene per tutti i widget, dopo che il canvas è stato costruito, ogni opzione è modificabile con il metodo `configure()` di cui ogni widget è dotato.

Ogni pixel del canvas è identificato da una coppia di coordinate, la prima indicante la posizione in orizzontale (convenzionalmente chiamata `x`) e la seconda indicante la posizione in verticale (convenzionalmente chiamata `y`). L'origine del sistema di coordinate sta nell'angolo superiore di sinistra del canvas, che ha coordinate `x = 0` e `y = 0`.

I metodi che abbiamo a disposizione per disegnare nel canvas sono quelli stessi del modulo Tkinter e per essi rimando al mio manualetto citato in premessa.

Frame

Il frame (termine inglese che in italiano significa telaio) è uno spazio rettangolare in cui possiamo collocare altri widget e vedremo come.

La costruzione dell'oggetto frame che vogliamo concretamente usare avviene con l'istruzione

```
<identificatore> = CTkFrame(<genitore>, <opzione> = <valore>, ...)
```

dove

<identificatore> è il nome che diamo al nostro frame;

<genitore> identifica il contenitore in cui viene inserito il frame;

<opzione> e <valore> servono per definire determinati attributi del nostro frame.

Le opzioni possibili sono molte. Per l'economia di questo manualletto ci basti conoscere queste tre fondamentali:

- . la larghezza del frame, esprimibile con `width = <numero_pixel>`

- . l'altezza del frame, esprimibile con `height = <numero_pixel>`

- . il colore di fondo del frame, esprimibile con `fg_color = <colore>`

- . lo spessore del bordo, esprimibile con `border_width = <numero_pixel>`

- . il colore del bordo, esprimibile con `border_color = <colore>`

Per come indicare il colore rimando a quanto detto per il canvas. Il colore di default, nel caso non lo si indichi, è un grigio chiaro (lightgrey).

Con

```
f = CTkFrame(r, width = 150, height = 100, fg_color = 'yellow', border_width = 5, border_color = 'green')
```

costruiamo un frame chiamato `f`, nella finestra radice `r`, con le caratteristiche indicate.

Il tutto modificabile ricorrendo al metodo `configure()`.

Widget collocati per errore in un frame o per i quali si è cambiata idea si possono eliminare con il metodo `destroy()` di ciascun widget (scrivendo `<identificatore_widget>.destroy()`)

3.2 Widget di cui non si può fare a meno per costruire una GUI

Qualsiasi interfaccia utente deve necessariamente ed almeno consentire all'utente di comunicare con il computer, al computer di comunicare con l'utente ed all'utente di far partire o di arrestare una o più attività del computer.

Per poter fare queste cose CustomTkinter ci mette a disposizione tre widget: uno per fornire dati al computer (che viene chiamato `CTkEntry`), uno per scrivere dati che ci restituisce il computer o per rendere visibili istruzioni per l'utente (cose per le quali torna comodo quello chiamato `CTkLabel`) ed uno per dare il via o per stoppare determinate azioni del computer (cose per le quali va benissimo quello chiamato `CTkButton`).

CTkEntry

Il widget `CTkEntry` consiste in una finestrella che serve per immettere una singola linea di testo.

Si costruisce con l'istruzione

```
<identificatore> = CTkEntry(<genitore>, <opzione> = <valore>, ...)
```

dove <genitore> è l'identificatore del contenitore in cui vogliamo inserire il widget.

Tra le tante opzioni disponibili forse la più utile è quella che ci consente di stabilire la larghezza della finestrella in cui inserire il testo, per default di 140 pixel. Per modificare questa dimensione è disponibile l'opzione `width = <numero_pixel>`.

Con le opzioni `fg_color = <colore>` e `text_color = <colore>` possiamo stabilire, rispettivamente, il colore dello sfondo della finestrella e del carattere scritto. Con l'opzione `justify =` possiamo allineare il contenuto della finestrella, utilizzando i parametri `LEFT`, `CENTER` e `RIGHT`.

Per la rassegna completa delle opzioni rimando alla documentazione ufficiale.

Il principale metodo del widget `Entry` è ovviamente quello di leggere e ritornare ciò che è stato scritto nella finestrella; si tratta del metodo `get()`, che ritorna una stringa contenente ciò che è stato scritto. Attenzione che il metodo ritorna una stringa anche se è stato immesso un numero, per cui, se si vuole un numero occorre fare il casting.

Per ripulire il contenuto della finestra abbiamo il metodo `delete(0, END)`; se al posto di 0 mettiamo l'indice del primo carattere da cancellare e al posto di END l'ultimo, possiamo cancellare solo parte del contenuto.

Altro utile metodo è `focus()`, con il quale possiamo stabilire se la nostra finestra debba essere già pronta per ricevere per prima i dati di input. Nel qual caso al suo interno vedremo un cursore lampeggiante.

Per richiamare una funzione una volta confermato l'inserimento si usa il metodo `bind('<Return>', <nome_funzione>)` (la funzione deve avere event tra i parametri attesi).

Esempi:

Con l'istruzione

```
inp = CTkEntry(f, width = 100, fg_color = 'Lightblue')
```

costruiamo nel frame `f` una finestra di input che chiamiamo `inp`, di ampiezza 100 pixel, con sfondo azzurro.

Con l'istruzione

```
inp.focus()
```

la indichiamo come già predisposta a ricevere dati.

Se nella finestra scriviamo una stringa di testo, ad esempio un nome, possiamo inserire la stringa stessa in una variabile stringa, per esempio chiamata `nome`, con l'istruzione

```
nome = inp.get()
```

Se nella finestra scriviamo un numero che deve poi essere utilizzato come tale per fare calcoli, per inserire questo numero in una variabile numerica di tipo intero dobbiamo usare l'istruzione

```
n = int(inp.get())
```

e, per inserirlo in una variabile numerica a virgola mobile, l'istruzione

```
n = float(inp.get())
```

CTkLabel

Il widget `CTkLabel` consiste in una finestra per esporre testo o numeri. E' utile sia per scrivere nella più ampia finestra della GUI istruzioni di compilazione e avvertenze varie, sia per scrivere le risposte del calcolatore alle elaborazioni che abbiamo chiesto.

Si costruisce con l'istruzione

```
<identificatore> = CTkLabel(<genitore>, <opzione> = <valore>, ...)
```

dove `<genitore>` è l'identificatore del contenitore in cui vogliamo inserire il widget.

Tra le tante opzioni disponibili quella sicuramente più importante è `text = ' .... '` che espone la stringa di testo racchiusa tra apici.

Altrettanto importanti sono il metodo `text_color = <colore>` per dare un colore al testo e quello per la scelta del carattere con cui scriverlo: `font = (<famiglia>, <dimensione>, <tipo>)`.

Per quanto riguarda il colore di fondo, quello di default è il solito grigio chiaro (`lightgrey`). Per modificarlo, soprattutto per renderlo magari uguale a quello non di default scelto per il contenitore, abbiamo il metodo `fg_color = <colore>`.

Attenzione meritano pure l'opzione `width = <valore>` e `height = <valore>` con le quali possiamo fissare un'ampiezza e un'altezza, in pixel, della label. Questo merita attenzione in quanto, una volta fissata l'ampiezza, ciò che scriviamo per l'opzione `text` non comparirà interamente se ha un numero di caratteri o di righe superiore. Se utilizziamo queste opzioni abbiamo il vantaggio di poter posizionare il testo all'interno della label utilizzando l'opzione `anchor = <parametro>`, dove `parametro` può essere `'w'` per un allineamento a sinistra, e `'e'` per un allineamento a destra. Per default il testo è al centro, `parametro 'center'`.

Se la label contempla più righe possiamo usare i parametri `'ne'` per un allineamento in alto a destra, `'sw'` per un allineamento in basso a sinistra, ecc. Se non utilizziamo queste opzioni la label ha un'ampiezza e un'altezza elastiche che si adattano al contenuto; in questo caso, per evitare che l'ampiezza vada oltre una dimensione tollerabile, si può usare l'opzione `wraplength = <numero_pixel>`, in modo che quanto esposto vada a capo su una nuova riga una volta raggiunta l'ampiezza indicata.

Anche dopo la costruzione della label possiamo modificare o inserire nuove opzioni con il metodo

```
<identificatore>.configure(<opzione> = <valore>).
```

E' questo il metodo che utilizziamo, con l'opzione `text = <valore>` per scrivere i risultati di elaborazioni non ancora visibili all'apertura della GUI. Bene sapere che, in questa sede, `<valore>`, anziché essere una stringa tra apici, può essere il nome di una variabile, sia di tipo stringa sia di tipo numerico, non scritto tra apici. La variabile di tipo numerico non ha bisogno di casting se è l'unica componente della scritta; occorre il casting `string(<variabile_numerica>)` solo se è necessario un concatenamento con una stringa.

Esempio:

Se abbiamo il risultato 8 di un'operazione matematica nella variabile numerica chiamata `r` da mostrare in una label chiamata `l1` possiamo scrivere

```
l1.configure(text = r)
```

Se però vogliamo mostrare il risultato con la scritta «il risultato è: 8» dobbiamo scrivere

```
l1.configure(text = 'il risultato è: ' + string(r))
```

CTkButton

Il widget `CTkButton` è un pulsante sul quale si clicca con il mouse per far fare qualche cosa al computer.

Si costruisce con l'istruzione

```
<identificatore> = CTkButton(<genitore>, <opzione> = <valore>, ...)
```

dove `<genitore>` è l'identificatore del contenitore in cui vogliamo inserire il widget.

Le opzioni sono molte ma le più importanti sono `text = ' ... '` con cui si indica tra apici la stringa di testo esplicativa da scrivere sul pulsante (il pulsante ha dimensione elastica e si adatta alla lunghezza della scritta e si sviluppa anche in altezza se la scritta va a capo, ciò che è possibile inserendo `\n` nella stringa stessa) e `command = <identificatore_funzione>`, con cui si indica la funzione che deve essere eseguita al click sul pulsante.

Con `width = <numero_pixel>` e `height = <numero_pixel>` è comunque possibile specificare la larghezza e l'altezza del pulsante, se il pulsante deve ospitare un'immagine anziché una semplice scritta.

Utili potrebbero risultare anche `fg_color = <colore>` e `text_color = <colore>` per dare un colore diverso dal solito grigio chiaro e nero di default, rispettivamente allo sfondo del pulsante ed alla scritta.

Eventuali allineamenti della scritta sul pulsante diversi da quello di default (centro) si possono gestire con l'opzione `anchor` nello stesso modo visto per la `CTkLabel`.

3.3 Widget CTkTextbox

Abbiamo visto che il widget `CTkEntry` può accettare una sola riga di testo e ciò, per la funzione di acquisizione di un input alla quale è destinato, è più che sufficiente.

Per trattare più grandi quantità di testo, come potrebbe essere necessario in un'applicazione di editing di testo, abbiamo il widget `CTkTextbox`.

Si costruisce con l'istruzione

```
<identificatore> = CTkTextbox(<genitore>, <opzione> = <valore>, ...)
```

dove `<genitore>` è l'identificatore del contenitore in cui vogliamo inserire il widget.

Le opzioni più importanti riguardano

- * l'ampiezza della finestra, che possiamo stabilire con `width = <numero_pixel>` (per default è di 200),

- * l'altezza della finestra, che possiamo stabilire con `height = <numero_pixel>` (per default è di 200),

- * il font, che possiamo scegliere con `font = (<famiglia>, <dimensione>, <tipo>)`

- * il colore del carattere, che scegliamo con `text_color = <colore>` (per default è nero),

* il trattamento della riga eccedente la larghezza della finestra. Per andare a capo si preme il tasto Invio della tastiera. Per default, quando si raggiunge il limite destro della finestra si innescia a capo automatico sul carattere senza riguardo all'interezza della parola; il che corrisponde all'opzione `wrap = 'char'`. Per evitare l'interruzione della parola si deve usare l'opzione `wrap = 'word'`, in modo che a capo automatico si inneschi dopo l'ultima parola completa battuta prima di raggiungere il limite destro della finestra. L'opzione `wrap = 'none'` esclude che si vada a capo automaticamente e l'eccedenza scritta nella riga rimane nascosta e diventa visibile scorrendo la riga stessa.

4 Dove non arriva CustomTkinter

Vi sono un paio di cose molto importanti per un programma dotato di interfaccia grafica che gli sviluppatori di CustomTkinter non hanno ritenuto di assoggettare alla loro opera di abbellimento.

Si tratta del modulo per creare un menu a discesa e del modulo per creare finestre di dialogo per il trattamento di file.

4.1 Modulo Menu di Tkinter

CustomTkinter non ha modo di trattare il classico menu a discesa tipico delle applicazioni dotate di GUI e, per dotare la nostra GUI costruita con CustomTkinter dobbiamo avvalerci del widget Menu che ci fornisce Tkinter.

Per poterlo fare dobbiamo importare, insieme al modulo `customtkinter` anche il modulo Menu di tkinter con l'istruzione in testa al nostro script

```
from tkinter import Menu
```

Il widget Menu ha la particolarità di non poter essere inserito in un altro widget e può essere pertanto inserito solo nel contenitore radice, che è il genitore di tutto il progetto.

Per costruire un menu abbiamo innanzi tutto bisogno di una barra in cui collocarlo; la barra si costruisce con l'istruzione

```
<identificatore_barra> = Menu(<identificatore_radice>).
```

Poi dobbiamo inserire la barra nel contenitore radice con l'istruzione

```
<identificatore_radice>.configure(menu = <identificatore_barra>).
```

Finalmente diamo avvio alla costruzione del menu nella barra con l'istruzione

```
<identificatore_menu> = Menu(<identificatore_barra>).
```

Ora aggiungiamo le voci del menu, ciascuna con l'istruzione

```
<identificatore_menu>.add_command(label = '...', command = <funzione_da_eseguire>)
```

dove tra gli apici inseriamo una stringa per il nome da dare alla voce di menu e come funzione da eseguire richiamiamo una funzione altrove definita.

Possiamo separare un comando dall'altro con `<identificatore_menu>.add_separator()`.

Chiudiamo con l'istruzione

```
<identificatore_barra>.add_cascade(label = '...', menu = <identificatore_menu>)
```

che dà la pennellata finale.

Esempio:

Con queste istruzioni costruiamo un menu `m` nella barra `b` del contenitore radice `r` con una sola voce Chiudi che richiama la funzione predefinita `quit` per chiudere la finestra.

```
b = Menu(r)
r.config(menu = b)
m = Menu(b)
m.add_command(label = 'Chiudi', command = quit)
b.add_cascade(label = 'File', menu = m)
```

4.2 Modulo filedialog di Tkinter

Nel menu di certe applicazioni è necessario inserire voci che ci aiutino a caricare o a salvare il lavoro da svolgere o svolto: si pensi, per esempio, ad un editor di testo o a un'agenda calendario.

Per fare questo Tkinter ha un modulo aggiuntivo, che si chiama `filedialog` e che, per poter essere utilizzato, occorre caricare esplicitamente con l'istruzione

```
import tkinter.filedialog as fd
```

in modo da rendere disponibile il modulo `filedialog` e dargli un nome (in questo caso ho scelto `fd` ma potremmo utilizzare qualsiasi nome). Ora `fd` è un oggetto e possiamo richiamare i suoi metodi con la classica sintassi della programmazione a oggetti (`fd.<nome_metodo>`).

I due metodi che ci fornisce questo modulo sono

```
askopenfilename(<opzione> = <valore>, ... )
```

```
asksaveasfilename(<opzione> = <valore>, ... ).
```

Il richiamo di questi metodi apre la classica finestra di dialogo con cui ci viene chiesto, rispettivamente, dove e con che nome trovare il file da aprire o dove e con quale nome salvare il file da salvare. La chiusura della finestra dopo l'inserimento dei dati richiesti, ritorna il path del file.

Le opzioni più importanti riguardano:

- . l'inserimento di un titolo per la finestra di dialogo, che facciamo con `title = '...'`,

- . l'inserimento del tipo e dell'estensione del file, che facciamo con `filetypes =` seguito da una lista di tuple di due elementi, il primo ad indicare il tipo del file, il secondo ad indicarne l'estensione. La sintassi completa è

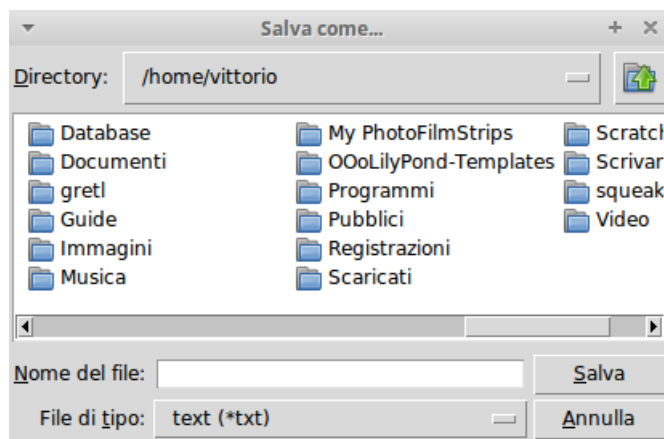
```
filetypes = [(<tipo_file>, <estensione>), (<tipo_file>, <estensione>), ...].
```

Esempio:

Con l'istruzione

```
p = fd.asksaveasfilename(title = 'Salva come...', filetypes = [('text', '*.txt'), ('python', '*.py')])
```

apriamo questa finestra



Se nella finestrella NOME DEL FILE inseriamo, per esempio, `script_prova` e, aprendo la finestra FILE DI TIPO, cliccando sul rettangolino sulla destra, scegliamo `python (*.py)`, dopo aver chiuso la finestra cliccando su SALVA nella nostra variabile `p` avremo il path `/home/vittorio/script_prova.py`.

5 La geometria della GUI

Ora viene la parte più delicata e laboriosa della creazione dell'interfaccia grafica perché occorre decidere quali widget ci servono, dove dobbiamo collocarli, sia in senso funzionale sia in senso estetico, come dimensionare e armonizzare il tutto, in modo che la nostra GUI risulti facile da utilizzare e bella da vedere.

Per fare questo, al servizio della nostra creatività e della nostra progettualità (cose che dobbiamo avere noi), CustomTkinter ci offre tre strumenti per collocare i widget nei contenitori, strumenti che corrispondono ad altrettanti metodi di cui è dotato ciascun oggetto widget:

- . il metodo `.pack()`,
- . il metodo `.grid()`,
- . il metodo `.place()`.

Una volta creato il widget, per poterlo vedere nel suo contenitore dobbiamo richiamare uno di questi suoi metodi con la sintassi

```
<identificatore_widget>.<metodo>.
```

Bene chiarire subito che è raccomandabile, per più widget nello stesso contenitore, utilizzare lo stesso metodo; in caso contrario si potrebbero creare conflitti che rendono ingovernabile il rendering di tutta la GUI.

Se per una zona della GUI ci torna comoda la geometria generata dal metodo `pack` e per un'altra zona ci torna comoda la geometria generata dal metodo `grid`, creiamo due frame separati, uno per ciascuna geometria.

5.1 Geometria con il metodo `pack`

Con il metodo `pack` i widget vengono inseriti impacchettati uno via l'altro, nell'impostazione di default dall'alto in basso: il primo inserito sta in alto, il secondo sta sotto di lui, ecc. Possiamo modificare questa impostazione con l'opzione `side` = che accetta i valori precostituiti `LEFT`, `RIGHT`, `BOTTOM` e `TOP` (quello di default).

Per comprendere l'effetto di queste opzioni occorre considerare che, nella geometria `pack`, il contenitore è elastico e, quando vi si inserisce il primo widget, esso si restringe attorno ad esso. Ciò non toglie che la cavità in cui inserire gli altri widget, anche se non si vede più, rimanga a disposizione. Nell'impostazione di default la cavità disponibile si sviluppa sotto il widget inserito, in quanto questo è nella posizione `TOP`, per cui il successivo inserimento si collocherà sotto il primo widget. Se avessimo inserito il primo widget con l'opzione `side` = `BOTTOM`, la cavità libera si svilupperebbe verso l'alto, sopra il primo widget inserito, e il successivo si collocherebbe sopra. In entrambi i casi la cavità occupata si sviluppa a sinistra e a destra sotto o sopra del widget nel contenitore e lascia libera tutta la fascia orizzontale sottostante o sovrastante in cui poter lavorare scegliendo sinistra o destra.

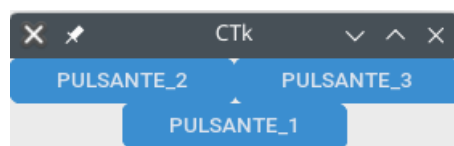
Purtroppo quando scegliamo l'opzione `side` = `LEFT` o `side` = `RIGHT`, la cavità libera rimane, rispettivamente, a destra o a sinistra del widget e, se dopo aver inserito, per esempio, un widget con l'opzione `side` = `LEFT`, ne inseriamo uno con l'opzione `side` = `TOP` ce lo ritroveremo sì al top ma nella parte destra della finestra e mai più, con la geometria `pack` nello stesso contenitore, potremo inserirlo al top e al centro.

Con questa geometria siamo pertanto costretti a ricorrere spesso a sdoppiamenti dei contenitori.

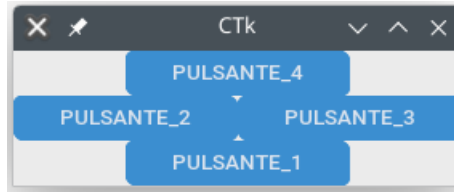
Esempio:

Il seguente script

```
from customtkinter import *
r = CTk()
p1 = CTkButton(r, text = 'PULSANTE_1')
p1.pack(side = BOTTOM)
p2 = CTkButton(r, text = 'PULSANTE_2')
p2.pack(side = LEFT)
p3 = CTkButton(r, text = 'PULSANTE_3')
p3.pack(side = LEFT)
r.mainloop()
crea questa finestra
```



Se volessimo realizzare una finestra così concepita



dovremmo ricorrere a questo script

```
from customtkinter import *
r = CTk()
p1 = CTkButton(r, text = 'PULSANTE_1')
p1.pack(side = BOTTOM)
p4 = CTkButton(r, text = 'PULSANTE_4')
p4.pack(side = TOP)
p2 = CTkButton(r, text = 'PULSANTE_2')
p2.pack(side = LEFT)
p3 = CTkButton(r, text = 'PULSANTE_3')
p3.pack(side = LEFT)
r.mainloop()
```

nel quale vediamo come, per tenere al centro il quarto pulsante, dobbiamo impacchettarlo nella cavità prima che questa sia affettata dall'opzione `side = LEFT` degli altri pulsanti.

Altra alternativa per ottenere lo stesso risultato sarebbe quella di creare un altro contenitore, ad esempio così:

```
from customtkinter import *
r = CTk()
zona_alta = CTkFrame(r)
zona_alta.pack()
p4 = CTkButton(zona_alta, text = 'PULSANTE_4')
p4.pack()
p1 = CTkButton(r, text = 'PULSANTE_1')
p1.pack(side = BOTTOM)
p2 = CTkButton(r, text = 'PULSANTE_2')
p2.pack(side = LEFT)
p3 = CTkButton(r, text = 'PULSANTE_3')
p3.pack(side = LEFT)
r.mainloop()
```

Come si vede, la geometria con il metodo `pack` non è così semplice da realizzare, soprattutto se pensiamo a GUI che abbiano una certa ricchezza di alternative e l'aspirazione di essere belle da vedere.

A quest'ultimo proposito si sappia che al fitto impacchettamento dei widget, uno a ridosso dell'altro, si può rimediare con le opzioni `padx = <numero_pixel>` e `pady = <numero_pixel>` del metodo `pack()` con le quali otteniamo che il widget venga inserito con attorno un margine, rispettivamente in orizzontale e in verticale, della dimensione in pixel indicata, margine che eredita il colore di sfondo del contenitore. In questo modo possiamo almeno distribuire le cose con un certo senso estetico.

5.2 Geometria con il metodo grid

Anche nella geometria grid il contenitore è elastico e si restringe attorno ai widget man mano li inseriamo. Il grande vantaggio del metodo sta nel fatto che la cavità nella quale inseriamo i widget è idealmente divisibile in righe e colonne, a mo' di griglia (grid, appunto), numerate da 0 in su con l'origine in alto a sinistra.

Possiamo scegliere in quale cella della griglia collocare un widget con le opzioni `column = <numero_intero>` e `row = <numero_intero>` che identificano le coordinate della cella in termini di colonna e riga.

Se vogliamo che la cella si estenda su più colonne usiamo l'opzione `columnspan = <numero_colonne>` e se vogliamo che si estenda su più righe usiamo l'opzione `rowspan = <numero_righe>`.

Esempio:

Per realizzare, con il metodo `grid`, una finestra simile alla seconda contemplata nell'esempio contenuto nel paragrafo precedente lo script è il seguente:

```
from customtkinter import *
r = CTk()
p4 = CTkButton(r, text = 'PULSANTE_4')
p4.grid(row=0, column=0, columnspan=2)
p2 = CTkButton(r, text = 'PULSANTE_2')
p2.grid(row=1, column=0)
p3 = CTkButton(r, text = 'PULSANTE_3')
p3.grid(row=1, column=1)
p1 = CTkButton(r, text = 'PULSANTE_1')
p1.grid(row=2, column=0, columnspan=2)
r.mainloop()
```

Anche con il metodo `grid` abbiamo a disposizione le opzioni `padx = <numero_pixel>` e `pady = <numero_pixel>` per costituire attorno ai widget che inseriamo delle cornici distanziatrici.

Infine, dal momento che le celle di ogni riga e di ogni colonna si dimensionano prendendo la dimensione della più alta o della più larga, può accadere spesso che un widget inserito in una cella non la occupi tutta. Per default, in questi casi, il widget si centra orizzontalmente e verticalmente nella cella e ciò può creare disallineamenti non graditi. Tutto si può regolare con l'opzione `sticky` = che accetta i valori `'w'` e `'e'` per allineare orizzontalmente il contenuto della cella, rispettivamente, a sinistra e a destra, lasciandolo centrato in senso verticale, `'n'` e `'s'` per allineare verticalmente il contenuto della cella, rispettivamente, in alto e in basso, lasciandolo centrato in senso verticale, con le possibili combinazioni `'nw'`, `'ne'`, `'sw'` e `'se'` si allineano i contenuti negli angoli, rispettivamente, in alto a sinistra, in alto a destra, in basso a sinistra e in basso a destra.

Esempio:

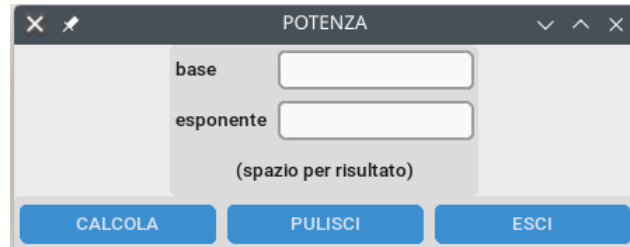
Questo script

```
from customtkinter import *
r = CTk()
r.title('POTENZA')
zona_alta = CTkFrame(r)
zona_alta.pack()
l1 = CTkLabel(zona_alta, text = 'base')
l1.grid(column = 0, row = 0, sticky = 'w', padx = 4)
e1 = CTkEntry(zona_alta)
e1.grid(column = 1, row = 0, padx = 4, pady = 4)
l2 = CTkLabel(zona_alta, text = 'esponente')
l2.grid(column = 0, row = 1, sticky = 'w', padx = 4)
e2 = CTkEntry(zona_alta)
e2.grid(column = 1, row = 1, padx = 4, pady = 4)
l3 = CTkLabel(zona_alta, text = '(spazio per risultato)')
l3.grid(row = 2, columnspan = 2, pady = 4)
zona_bassa = CTkFrame(r)
zona_bassa.pack()
b1 = CTkButton(zona_bassa, text = 'CALCOLA')
b1.grid(column = 0, row = 0, padx = 4, pady = 6)
```

```

b2 = CtkButton(zona_bassa, text = 'PULISCI')
b2.grid(column = 1, row = 0, padx = 4)
b3 = CtkButton(zona_bassa, text = 'ESCI')
b3.grid(column = 2, row = 0, padx = 4)
r.mainloop()
crea questa finestra

```



Si tratta di una GUI destinata ad un'applicazione che calcola la potenza ennesima di un numero.

Richiamo l'attenzione su come la GUI sia stata costruita con due frame, uno per la zona alta e uno per la zona bassa: ho ritenuto utile agire così in quanto la zona alta è utilizzata e stabilizzata su due colonne e, per poter collocare armonicamente i tre pulsanti mi serviva una zona di tre colonne.

Si nota inoltre come la collocazione dei due frame sia stata fatta con il metodo pack e, all'interno dei frame, si sia agito con il metodo grid: a seconda dei casi si usa ciò che si ritiene più funzionale. Purché si evitino conflitti. Se, per esempio, la zona alta della GUI fosse stata disegnata nel contenitore r con il metodo grid invece di creare il frame per la zona alta, non avremmo più potuto impacchettare il frame della zona bassa con il metodo pack in quanto il contenitore r sarebbe già stato strutturato con il metodo grid.

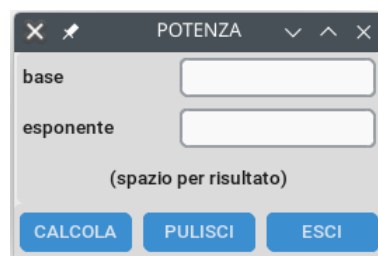
I widget sono stati collocati senza indicarne le dimensioni, per cui ciò che vediamo sopra nel rendering ha acquisito le dimensioni di default per i widget CtkEntry e CtkButton, che è di 140 pixel e la nostra GUI non è molto bella da vedere e merita una aggiustatina.

Cominciamo dalla dimensione dei pulsanti, che può essere ridotta a 80 pixel, aggiungendo tra le opzioni dei tre pulsanti l'opzione `width = 80`.

A questo punto la larghezza della GUI diventa la somma dei pixel dei tre pulsanti (240 pixel) più gli spazi di 4 pixel aggiunti con le opzioni `padx`: in tutto 244 pixel.

Nella zona superiore, manteniamo la larghezza di default di 140 pixel per le CtkEntry e pareggiamo la larghezza di 244 pixel della zona inferiore assegnando alle due CtkLabel la larghezza di 104 pixel aggiungendo tra le opzioni delle prime due label l'opzione `width = 104` e l'opzione `anchor = 'w'` per allineare a sinistra la scritta.

Con questi aggiustamenti otteniamo questa GUI



5.3 Geometria con il metodo place

La prima grande differenza tra questo metodo e i due che abbiamo visto prima sta nel fatto che il contenitore non si restringe attorno ai widget man mano che li inseriamo ma rimane tutto sempre disteso secondo la dimensione di default o quella che gli abbiamo assegnato.

Ogni widget si inserisce determinando un punto del contenitore al quale ancorarlo e determinando come ancorarlo.

Il punto si determina indicandone le coordinate in pixel con le opzioni `x = <pixel>` e `y = <pixel>`, ricordando che l'origine, dove `x` è uguale a 0 e `y` è uguale a 0 sta nell'angolo in alto a sinistra del contenitore. Oppure si determina con le opzioni `relx = <valore>` e `rely = <valore>`, dove il valore è un numero decimale compreso tra 0 e 1 che indica una frazione, rispettivamente, dell'ampiezza del contenitore e dell'altezza del contenitore: il valore 0.5 indica il punto centrale dell'asse, 0.25 indica un punto che sta al primo quarto dell'asse, ecc.

L'ancoraggio al punto si determina con l'opzione `anchor = <valore>`, dove il valore può essere `'n'`, `'e'`, `'s'`, `'w'`, `'ne'`, `'nw'`, `'se'` e `'sw'` e indica il lato o l'angolo di ancoraggio stesso. Il valore di default è `'nw'` e sta ad indicare che il widget è ancorato al punto nel suo angolo in alto a sinistra. Se indichiamo `'n'` il widget viene ancorato al punto con il centro del suo lato superiore, se indichiamo `'e'` il widget viene ancorato al punto con il centro del suo lato destro, ecc.

Nella geometria con il metodo `place` non abbiamo a disposizione le opzioni `padx` e `pady` e i distanziamenti tra i widget devono essere creati individuando adeguatamente i punti di ancoraggio.

Secondo alcuni questo è il metodo più semplice dei tre che abbiamo esaminato. Personalmente non condivido questo parere in quanto la difficoltà di individuare esattamente le coordinate dei punti cui ancorare i widget per realizzare una GUI ben fatta mi pare notevole.

Un aiuto potrebbe derivare dalla possibilità di sostituire ai riferimenti in pixel riferimenti in unità di misura più alla nostra portata, come centimetri, millimetri o pollici: per farlo, mentre i pixel si indicano semplicemente con un numero, occorre inserire tra apici il numero immediatamente seguito dalla lettera `c` per indicare centimetri, dalla lettera `m` per indicare millimetri o dalla lettera `i` per indicare pollici: per esempio, il valore di 22 millimetri si indica con `'22m'`. In questo modo diventa forse più agevole lavorare con le coordinate dei punti.

Rimane il grosso problema dato dal fatto che, per collocazioni dei widget con giusti distanziamenti, occorrerebbe sempre conoscere esattamente gli spazi occupati dai widget che via via si inseriscono nella GUI. Questo problema può essere superato dall'esistenza di due metodi, comuni a qualsiasi widget, che ne ritornano le dimensioni in pixel

`<identificatore_widget>.winfo_width()` ritorna la larghezza,

`<identificatore_widget>.winfo_height()` ritorna l'altezza.

Attraverso questi due metodi possiamo delegare allo script il compito di misurare gli spazi occupati dai widget via via inseriti in modo da sapere dove sia possibile inserire gli altri. C'è però un problema: per poter misurare l'ingombro di un oggetto occorre prima crearlo e, in uno script `CustomTkinter`, la creazione degli oggetti avviene solo con l'istruzione di `mainloop` finale. Il problema si supera inserendo nello script, appena dopo il costruttore del widget e appena prima del metodo per misurarne la larghezza o l'altezza il metodo

`<identificatore_contenitore>.update()`.

I creatori di `CustomTkinter` e, prima ancora quelli di `Tkinter`, le hanno pensate proprio tutte. Esempi:

Per realizzare, con il metodo `place`, una finestra simile alla seconda contemplata nell'esempio contenuto nel paragrafo dedicato al metodo `pack` lo script è il seguente:

```
from customtkinter import *
r = CTk()
r.configure(width = 300, height = 80)
b4 = CTkButton(r, text = 'PULSANTE_4')
b4.place(relx = 0.5, anchor = 'n')
b2 = CTkButton(r, text = 'PULSANTE_2')
b2.place(relx = 0.25, anchor = 'n', rely = 0.25)
b3 = CTkButton(r, text = 'PULSANTE_3')
b3.place(relx = 0.75, anchor = 'n', rely = 0.25)
b1 = CTkButton(r, text = 'PULSANTE_1')
b1.place(relx = 0.5, anchor = 'n', rely = 0.5)
r.mainloop()
```

Come si vede il tutto è basato su posizionamenti indicati in termini relativi.

Per realizzare, con il metodo `place`, una finestra simile a quella realizzata prima con il metodo `grid` potremmo utilizzare lo script seguente:

```
from customtkinter import *
r = CTk()
r.title('POTENZA')
r.configure(width = 260, height = 110)
l1 = CTkLabel(r, text = 'base')
l1.place(x = 15, y = 2)
e1 = CTkEntry(r, width = 140)
e1.place(x = 115, y = 2)
l2 = CTkLabel(r, text = 'esponente')
l2.place(x = 15, y = 26)
e2 = CTkEntry(r, width = 140)
e2.place(x = 115, y = 26)
l3 = CTkLabel(r, text = '(spazio per risultato)')
l3.place(x = 15, y = 50)
b1 = CTkButton(r, text = 'CALCOLA', width = 80)
b1.place(x = 2, y = 74)
b2 = CTkButton(r, text = 'PULISCI', width = 80)
b2.place(x = 90, y = 74)
b3 = CTkButton(r, text = 'ESCI', width = 80)
b3.place(x = 178, y = 74)
r.mainloop()
```

In questo caso ho posizionato i widget utilizzando esclusivamente riferimenti assoluti. L'unico modo per riuscire ad azzeccare questi riferimenti è di comporre lo script verificando via via quello che succede lanciando nell'IDLE la parte già scritta e procedere alle rettifiche e alle determinazioni del caso.

Quest'altro è un esempio di script nel quale ho delegato allo script stesso la determinazione degli ingombri.

```
from customtkinter import *
r = CTk()
r.title('POTENZA')
r.configure(width = 260, height = 130)
l1 = CTkLabel(r, text = 'base')
l1.place(x = 15, y = 2)
e1 = CTkEntry(r, width = 140)
e1.place(x = 115, y = 2)
r.update()
ingombro_verticale = 2 + e1.winfo_height()
l2 = CTkLabel(r, text = 'esponente')
l2.place(x = 15, y = 4 + ingombro_verticale)
e2 = CTkEntry(r, width = 140)
e2.place(x = 115, y = 4 + ingombro_verticale)
r.update()
ingombro_verticale = ingombro_verticale + e2.winfo_height()
l3 = CTkLabel(r, text = '(spazio per risultato)')
l3.place(x = 15, y = 7 + ingombro_verticale)
r.update()
ingombro_verticale = ingombro_verticale + l3.winfo_height()
b1 = CTkButton(r, text = 'CALCOLA', width = 80)
b1.place(x = 2, y = 10 + ingombro_verticale)
r.update()
```

```

ingombro_orizzontale = 2 + b1.winfo_width()
b2 = CtkButton(r, text = 'PULISCI', width = 80)
b2.place(x = 4 + ingombro_orizzontale, y = 10 + ingombro_verticale)
r.update()
ingombro_orizzontale = ingombro_orizzontale + b2.winfo_width()
b3 = CtkButton(r, text = 'ESCI', width = 80)
b3.place(x = 8 + ingombro_orizzontale, y = 10 + ingombro_verticale)
r.mainloop()

```

I tre esempi presentati utilizzano ciascuno un diverso sistema di indicazione del piazzamento. Ciò non toglie che, nella realtà, si possa agire utilizzando l'un sistema o l'altro alternativamente nello stesso script, a seconda della convenienza.

6 Alcuni esempi di integrazione con il linguaggio Python

6.1 Potenze e radici

Il seguente script serve per calcolare potenze e radici utilizzando la GUI che avevamo costruito a suo tempo con il metodo grid.

```

#!/usr/bin/python3
from customtkinter import *
def calcola():
    base = float(e1.get())
    esponente = e2.get()
    if '/' in esponente:
        def cerca():
            i = 0
            while i < len(esponente):
                if esponente[i] == '/':
                    return i
                i = i + 1
        indice = cerca()
        r = esponente[indice+1:]
        esponente = 1/int(r)
        risultato = base ** esponente
        l3.configure(text = risultato)
    else:
        esponente = float(e2.get())
        risultato = base ** esponente
        l3.configure(text = risultato)
def pulisci():
    e1.delete(0, END)
    e2.delete(0, END)
    l3.configure(text = '')
    e1.focus()
r = Ctk()
r.title('POTENZA')
zona_alta = CtkFrame(r)
zona_alta.pack()
l1 = CtkLabel(zona_alta, text = 'base', width = 104, anchor = 'w')
l1.grid(column = 0, row = 0, sticky = 'w', padx = 4)
e1 = CtkEntry(zona_alta)
e1.grid(column = 1, row = 0, padx = 4, pady = 4)
l2 = CtkLabel(zona_alta, text = 'esponente', width = 104, anchor = 'w')
l2.grid(column = 0, row = 1, sticky = 'w', padx = 4)
e2 = CtkEntry(zona_alta)
e2.grid(column = 1, row = 1, padx = 4, pady = 4)
l3 = CtkLabel(zona_alta, text = '(spazio per risultato)')
l3.grid(row = 2, columnspan = 2, pady = 4)
zona_bassa = CtkFrame(r)
zona_bassa.pack()
b1 = CtkButton(zona_bassa, text = 'CALCOLA', width = 80, command = calcola)
b1.grid(column = 0, row = 0, padx = 4, pady = 6)
b2 = CtkButton(zona_bassa, text = 'PULISCI', width = 80, command = pulisci)
b2.grid(column = 1, row = 0, padx = 4)
b3 = CtkButton(zona_bassa, text = 'ESCI', width = 80, command = quit)
b3.grid(column = 2, row = 0, padx = 4)
r.mainloop()

```

In nero il codice customtkinter per il disegno della GUI e in rosso il codice Python.

La funzione calcola() è costruita in modo che si possa accettare l'inserimento di un esponente frazionario, così da poter calcolare, oltre che potenze, anche radici (ricordo che $\sqrt[n]{x} = x^{\frac{1}{n}}$).

6.2 Calcolo del fattoriale di un numero

Questo script contempla un'applicazione che calcola il fattoriale di un numero.

```
#!/usr/bin/python3
import math
def calcola():
    numero = zonaInput.get()
    n = int(numero)
    risultato.configure(text = math.factorial(n))
def pulisci():
    zonaInput.delete(0, END)
    risultato.configure(text = "")
from customtkinter import *
r = CTK()
r.title('FATTORIALE')
r.configure(fg_color = 'lightgreen')
descrizioneInput = CTKLabel(r, text = "Inserisci il numero di cui vuoi il fattoriale", fg_color = 'lightgreen')
descrizioneInput.pack(padx = 5)
zonaInput = CTKEntry(r, justify = CENTER)
zonaInput.focus()
zonaInput.pack(pady = 5)
zonaPulsanti = CTKFrame(r, fg_color = 'lightgreen')
zonaPulsanti.pack()
pulsanteCalcola = CTKButton(zonaPulsanti, text = 'CALCOLA', width = 80, fg_color = 'green', command = calcola)
pulsanteCalcola.grid(column = 0, row = 0, padx = 5)
pulsantePulisci = CTKButton(zonaPulsanti, text = 'PULISCI', width = 80, fg_color = 'green', command = pulisci)
pulsantePulisci.grid(column = 1, row = 0, padx = 5)
pulsanteEsci = CTKButton(zonaPulsanti, text = 'ESCI', width = 80, fg_color = 'green', command = quit)
pulsanteEsci.grid(column = 2, row = 0, padx = 5)
zonaOutput = CTKFrame(r, fg_color = 'lightgreen')
zonaOutput.pack()
descrizioneOutput = CTKLabel(zonaOutput, text = "Fattoriale del numero inserito:", fg_color = 'lightgreen')
descrizioneOutput.pack(pady = 5)
risultato = CTKLabel(zonaOutput, wraplength = 600, justify = LEFT, fg_color = 'lightgreen', text="")
risultato.pack(pady = 5)
r.mainloop()
```

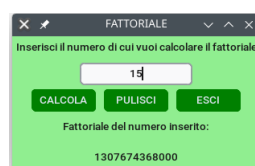
Anche qui abbiamo in nero il codice dedicato alla costruzione della GUI e in rosso il codice Python.

Per calcolare il fattoriale di un numero non è che ci sia un gran bisogno di una GUI, ma ho voluto produrre questo esempio soprattutto per dimostrare l'elasticità dei widget Tkinter.

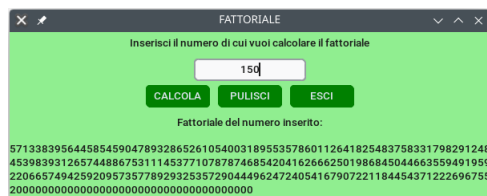
Nell'esempio precedente abbiamo trattato con numeri a virgola mobile (float) che Python scrive per default in notazione scientifica, mai occupando più di 22 caratteri. Le dimensioni della nostra GUI erano pertanto più che sufficienti per evidenziare il risultato del calcolo.

In questo esempio, invece, trattiamo numeri interi e Python, quando scrive numeri interi, per default li scrive tali e quali, in tutta la loro estensione; estensione che nel linguaggio Python, che non conosce stack overflow, può assumere dimensioni ragguardevoli.

Così, se calcoliamo il fattoriale di 15, la nostra GUI ce lo mostra nella normale finestra



Se calcoliamo il fattoriale di 150, che è un numero ben maggiore, la finestra diventa



Questa finestra allarga la label destinata a contenere il risultato fino al limite di 600 pixel indicato nell'opzione `wraplength` e la allunga fino a contenere l'enorme numero andando accapo.

6.3 Un piccolo editor di testo

Un'applicazione per la quale torna utile una GUI è l'editor di testo.

Possiamo costruirne uno molto semplice con questo script

```
#!/usr/bin/python3
from customtkinter import *
from tkinter import Menu
import tkinter.filedialog as fd

def apri():
    path = fd.askopenfilename(title = 'Scegli un file')
    if len(path) > 0:
        t.delete('1.0', 'end')
        with open(path, 'r') as f:
            t.insert('1.0', f.read())
        r.title(path)

def salva_come():
    path = fd.asksaveasfilename(title = 'Dove Salvare')
    if len(path) > 0:
        with open(path, 'w') as f:
            f.write(t.get('1.0', 'end'))
        r.title(path)

r = CTk()
r.title('Senza Titolo')
r.configure(width = 600, height = 400)
t = CTkTextbox(r, width = 600, height = 400, fg_color = 'mint cream', wrap = 'word')
t.pack(padx = 5, pady = 5)
mb = Menu(r)
r.configure(menu=mb)
fm = Menu(mb)
fm.add_command(label = 'Apri...', command = apri)
fm.add_command(label = 'Salva come...', command = salva_come)
fm.add_separator()
fm.add_command(label = 'Esci', command = quit)
mb.add_cascade(label = 'File', menu = fm)
r.mainloop()
```

Al solito abbiamo in rosso la parte di codice Python e in nero quella dedicata alla GUI.

E questo è il nostro editor



6.4 Master Mind

Dove la GUI assume un'importanza fondamentale è nei giochi. Per questa esigenza il mondo Python ci offre ben altro ma per alcuni giochi da tavolino dove non vi sia l'esigenza di movimento può tornare utile anche CustomTkinter.

Questo script contempla un'edizione semplificata, costruita utilizzando rigorosamente solo ciò che abbiamo visto in questo manualetto, del gioco Master Mind, con metodi grid e pack che si combinano, con frame e canvas che si alternano.

```
#!/usr/bin/python3
from random import *
def inizializza():
    areaTentativi.delete('pallina')
    areaRisposte.delete('quadrato')
    combinazione.delete('pallina')
    messaggio.configure(text = '')
    global listaGioco
    global listaTentativo
    global tentativi
    tentativi = 0
    global x1
    x1 = 28
    global x2
    x2 = 28
    global y1
    y1 = 10
    global y2
    y2 = 12
    listaBase = ['R', 'V', 'G', 'B']
    listaGioco = [1, 2, 3, 4]
    for i in range(4):
        p = int(random() * 4)
        listaGioco[i] = listaBase[p]
def tentativo(event):
    nomiColori = {'R': 'RED', 'V': 'GREEN', 'G': 'YELLOW', 'B': 'BLUE'}
    stringaColori = 'RVGB'
    stringaTentativo = stringaInput.get()
    verifica = ''
    if len(stringaTentativo) != 4:
        messaggio.configure(text = 'DEVI INSERIRE ALMENO 4 CARATTERI')
        verifica = 'NO'
    for i in range(4):
        if stringaTentativo[i] not in stringaColori:
            verifica = 'NO'
            break
    if verifica == 'NO':
        messaggio.configure(text = 'HAI INSERITO MALE QUALCOSA')
    else:
        messaggio.configure(text = '')
        global listaTentativo
        listaTentativo = [1,2,3,4]
        global x1
        global y1
        global tentativi
        for i in range(4):
            listaTentativo[i] = stringaTentativo[i]
        for i in range(4):
            areaTentativi.create_oval(x1, y1, x1+15, y1+15, fill = nomiColori[listaTentativo[i]])
            x1 += 20
            areaTentativi.addtag_all('pallina')
        stringaInput.delete(0,4)
        x1 = 28
        y1 += 20
        tentativi += 1
        xx = 28
        yy = 5
    if listaTentativo == listaGioco:
        messaggio.configure(text = 'MOLTO BENE!\nHAI VINTO CON %d TENTATIVI\nLA COMBINAZIONE ERA:' %(tentativi))
        for i in range(4):
            combinazione.create_oval(xx, yy, xx+15, yy+15, fill = nomiColori[listaGioco[i]])
            xx += 20
            combinazione.addtag_all('pallina')
    elif tentativi == 9:
        messaggio.configure(text = 'HAI PERSO!\nLA COMBINAZIONE ERA:')
        for i in range(4):
```

```

        combinazione.create_oval(xx, yy, xx+15, yy+15, fill = nomiColori[listaGioco[i]])
        xx += 20
        combinazione.addtag_all('pallina')
    elif verifica != 'NO':
        risposta()
def risposta():
    risposta = []
    for i in range(4):
        if listaTentativo[i] == listaGioco[i]:
            risposta.append('BLACK')
        elif listaTentativo[i] != listaGioco[i] and listaTentativo[i] in listaGioco:
            risposta.append('WHITE')
    risposta.sort()
    global x2
    global y2
    for i in range(len(risposta)):
        areaRisposte.create_rectangle(x2, y2, x2+11, y2+11, fill = risposta[i])
        x2 += 20
        areaRisposte.addtag_all('quadrantino')
    x2 = 28
    y2 += 20
from customtkinter import *
pianoDiGioco = CTK()
pianoDiGioco.title('MASTER MIND')
pianoDiGioco.configure(fg_color = 'gray85')
zonaApertura = CTKFrame(pianoDiGioco)
zonaApertura.grid()
descrizione = CTKLabel(zonaApertura)
descrizione.configure(text = 'Colori in gioco: ', text_color = 'blue')
descrizione.pack()
strisciaColori = CTKCanvas(zonaApertura, width = 150, height = 20)
strisciaColori.pack()
strisciaColori.create_text(30, 10, text = 'R', fill = 'red')
strisciaColori.create_text(60, 10, text = 'V', fill = 'green')
strisciaColori.create_text(90, 10, text = 'G', fill = 'yellow')
strisciaColori.create_text(120, 10, text = 'B', fill = 'blue')
regole = CTKLabel(zonaApertura, text = '9 tentativi a disposizione\nquadrantino nero: colore giusto posto giusto\nquadrantino bianco: colore giusto posto sbagliato', text_color = 'blue')
regole.pack(pady = 5)
zonaGioco = CTKFrame(pianoDiGioco)
zonaGioco.grid()
areaTentativi = CTKCanvas(zonaGioco)
areaTentativi.configure(width = 130, height = 200)
areaTentativi.grid(row = 0, column = 0, sticky = N)
areaRisposte = CTKCanvas(zonaGioco)
areaRisposte.configure(width = 130, height = 200)
areaRisposte.grid(row = 0, column = 1, sticky = N)
zonaInput = CTKFrame(pianoDiGioco)
zonaInput.grid()
descrizione_1 = CTKLabel(zonaInput)
descrizione_1.configure(text = 'Scegli quattro colori indicando\nla loro iniziale maiuscola.\nPoi premi INVIO', text_color = 'BLUE')
descrizione_1.grid(sticky = N)
stringaInput = CTKEntry(zonaInput)
stringaInput.configure(justify = CENTER)
stringaInput.focus()
stringaInput.bind('<Return>', tentativo)
stringaInput.grid(sticky = N)
zonaMessaggi = CTKFrame(pianoDiGioco)
zonaMessaggi.grid()
messaggio = CTKLabel(zonaMessaggi)
messaggio.configure(justify = CENTER, text_color = 'RED')
messaggio.grid(sticky = N)
combinazione = CTKCanvas(zonaMessaggi)
combinazione.configure(width = 130, height = 25)
combinazione.grid(sticky = N)
zonaSceltaFinale = CTKFrame(pianoDiGioco)
zonaSceltaFinale.grid()
pulsanteAltroGioco = CTKButton(zonaSceltaFinale)
pulsanteAltroGioco.configure(text = 'ALTRO GIOCO', fg_color = 'GREEN', width = 13, command = inizializza)
pulsanteAltroGioco.grid(row = 0, column = 0, sticky = N, padx = 4, pady = 4)
pulsanteChiudi = CTKButton(zonaSceltaFinale)
pulsanteChiudi.configure(text = 'CHIUDI', fg_color = 'GREEN', width = 8, command = quit)
pulsanteChiudi.grid(row = 0, column = 1, sticky = N, padx = 4, pady = 4)
inizializza()
pianoDiGioco.mainloop()

```

7 Considerazioni conclusive

Rispetto alle GUI create con Tkinter quelle create con CustomTkinter sono esteticamente migliori.

Per quanto mi riguarda, comunque, vanno benissimo anche quelle di Tkinter.

Nel manuale non ho citato tutte le possibilità «creative» di CustomTkinter, come quella che consente di arrotondare a piacimento gli angoli dei widget CTkEntry, CTkLabel e CTkButton attraverso l'opzione `corner_radius = <numero_pixel>`.

Per la completa conoscenza delle opzioni che caratterizzano questi principali widget, che consentono anche la creazione di bordi, rimando alla documentazione ufficiale, così come per la conoscenza dei molti altri widget che non ho citato in questo manuale, ridotto all'essenziale.

Non mancano comunque alcuni difetti nel lavoro di wrapping fatto da Tom Schimansky.

Per esempio non si capisce perché l'opzione di Tkinter `bg` per indicare il colore di background, sia diventata `fg_color`, quando `fg` indica generalmente il foreground.

E dal momento che ciò avviene soltanto per i widget radicalmente riscritti, ci troviamo a dover usare `fg_color` per il contenitore `frame` e `bg` per il contenitore `canvas`.

Fastidioso difetto anche quello che il colore di background di default dei widget (che ho scoperto sia Gray85) non è lo stesso di quello della finestra principale (che è Gray92).

C'è, insomma, una serie di complicazioni in più rispetto a ciò che avviene utilizzando il Tkinter tradizionale.

Ma a tutto ci si abitua, importante è il risultato.