Índice

Interfaces Gráficas en Python (TKinter)

Introducción

Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero **Tkinter** es fácil de usar, es multiplataforma y, además, viene incluido con Python en su versión para Windows, para Mac y para la mayoría de las distribuciones GNU/Linux. Se le considera el estándar de facto en la programación GUI con Python.

Tkinter es un binding de la biblioteca Tcl/Tk que está también disponible para otros lenguajes como Perl y Rubí.

A pesar de su larga historia, su uso no está demasiado extendido entre los usuarios de equipos personales porque su integración visual con los sistemas operativos no era buena y proporcionaba pocos **widgets** (controles) para construir los programas gráficos.

Sin embargo, a partir de **TKinter 8.5** la situación dio un giro de ciento ochenta grados en lo que se refiere a integración visual, mejorando en este aspecto notablemente; también en el número de widgets que se incluyen y en la posibilidad de trabajar con estilos y temas, que permiten ahora personalizar totalmente la estética de un programa. Por ello, ahora Tkinter es una alternativa atractiva y tan recomendable como otras.

Este tutorial tiene como objetivo principal introducir al desarrollador que no está familiarizado con la programación GUI en Tkinter. Para ello, seguiremos una serie de ejemplos que muestran, de manera progresiva, el uso de los elementos que son necesarios para construir una aplicación gráfica: ventanas, gestores de geometría, widgets, menús, gestión de eventos, fuentes, estilos y temas.

Consultar la versión de Tkinter

Normalmente, el paquete Tkinter estará disponible en nuestra instalación Python, excepto en algunas distribuciones GNU/Linux. Para comprobar la versión de Tkinter instalada existen varias posibilidades:

1) Iniciar el entorno interactivo de Python e introducir:

```
>>> import tkinter
>>> tkinter.Tcl().eval('info patchlevel')
```

2) Si tenemos el instalador Pip introducir:

```
$ pip3 show tkinter
```

Instalar Tkinter

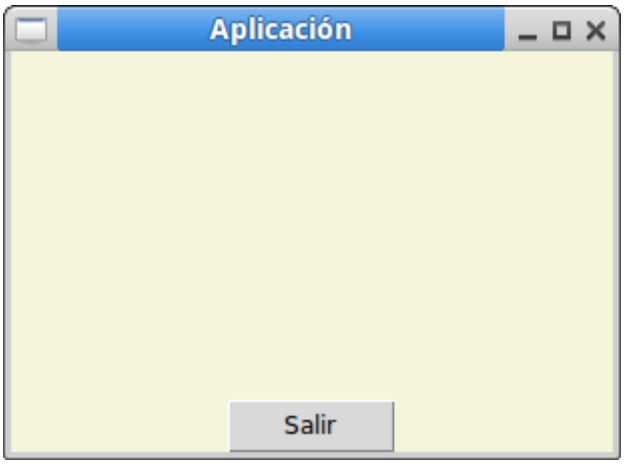
Si en nuestra instalación de Python, en un equipo con GNU/Linux, no se encuentra el paquete Tkinter instalar con:

```
$ sudo apt-get install python3-tk
```

(En el resto de plataformas cuando se instala Python se incluyen también los módulos de Tkinter).

La primera aplicación con Tkinter

El siguiente ejemplo crea una aplicación que incluye una ventana con un botón en la parte inferior. Al presionar el botón la aplicación termina su ejecución. Una ventana es el elemento fundamental de una aplicación GUI. Es el primer objeto que se crea y sobre éste se colocan el resto de objetos llamados widgets (etiquetas, botones, etc.).




```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# Las dos líneas siguientes son necesarias para hacer
# compatible el interfaz Tkinter con los programas basados
# en versiones anteriores a la 8.5, con las más recientes.

from tkinter import *    # Carga módulo tk (widgets estándar)
from tkinter import ttk  # Carga ttk (para widgets nuevos 8.5+)

# Define la ventana principal de la aplicación

raiz = Tk()

# Define las dimensiones de la ventana, que se ubicará en
```

```
# el centro de la pantalla. Si se omite esta línea la
# ventana se adaptará a los widgets que se coloquen en
# ella.

raiz.geometry('300x200') # anchura x altura

# Asigna un color de fondo a la ventana. Si se omite
# esta línea el fondo será gris

raiz.configure(bg = 'beige')

# Asigna un título a la ventana

raiz.title('Aplicación')

# Define un botón en la parte inferior de la ventana
# que cuando sea presionado hará que termine el programa.
# El primer parámetro indica el nombre de la ventana 'raiz'
# donde se ubicará el botón

ttk.Button(raiz, text='Salir', command=quit).pack(side=BOTTOM)

# Después de definir la ventana principal y un widget botón
# la siguiente línea hará que cuando se ejecute el programa
# construya y muestre la ventana, quedando a la espera de
# que alguna persona interactúe con ella.

# Si la persona presiona sobre el botón Cerrar 'X', o bien,
# sobre el botón 'Salir' el programa llegará a su fin.

raiz.mainloop()
```

La primera aplicación, orientada a objetos

A continuación, se muestra la misma aplicación pero orientada a objetos. Aunque este tipo de programación siempre es recomendable con Python no es imprescindible. Sin embargo, si vamos a trabajar con Tkinter es lo más adecuado, sobre todo, porque facilita la gestión de los widgets y de los eventos que se producen en las aplicaciones. Desde luego, todo van a ser ventajas.

Normalmente, cuando se ejecuta una aplicación gráfica ésta se queda a la espera de que una persona interactúe con ella, que presione un botón, escriba algo en una caja de texto, seleccione una opción de un menú, sitúe el ratón en una posición determinada, etc., o bien, se produzca un suceso en el que no haya intervención humana como que termine un proceso, que cambie el valor de una variable, etc. En cualquiera de estos casos, lo habitual

será vincular estos eventos o sucesos con unas acciones a realizar, que pueden ser mejor implementadas con las técnicas propias de la programación orientada a objetos.

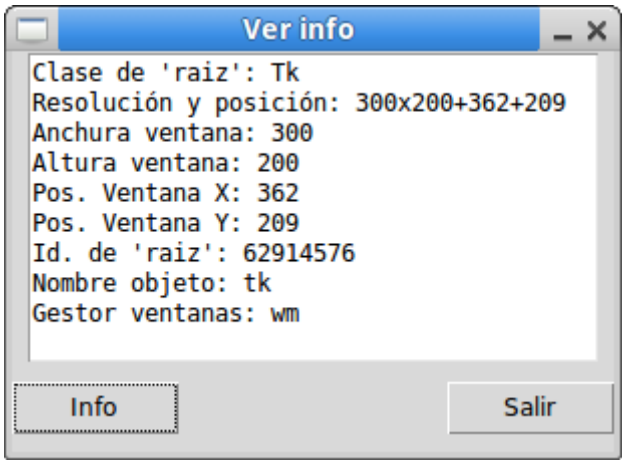
```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk

# Crea una clase Python para definir el interfaz de usuario de
# la aplicación. Cuando se cree un objeto del tipo 'Aplicacion'
# se ejecutará automáticamente el método __init__() qué
# construye y muestra la ventana con todos sus widgets:

class Aplicacion():
    def __init__(self):
        raiz = Tk()
        raiz.geometry('300x200')
        raiz.configure(bg = 'beige')
        raiz.title('Aplicación')
        ttk.Button(raiz, text='Salir',
                   command=raiz.destroy).pack(side=BOTTOM)
        raiz.mainloop()

# Define la función main() que es en realidad la que indica
# el comienzo del programa. Dentro de ella se crea el objeto
# aplicación 'mi_app' basado en la clase 'Aplicación':

def main():
    mi_app = Aplicacion()
    return 0

# Mediante el atributo __name__ tenemos acceso al nombre de un
# un módulo. Python utiliza este atributo cuando se ejecuta
# un programa para conocer si el módulo es ejecutado de forma
# independiente (en ese caso __name__ = '__main__') o es
# importado:

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

Obtener información de una ventana

Para finalizar este capítulo se incluye una aplicación basada en los ejemplos anteriores que sirve para algo, concretamente, para mostrar información relacionada con la ventana.

Para ello, en la ventana de la aplicación se han agregado nuevos widgets: un botón con la etiqueta "Info" y una caja de texto que aparece vacía.

También, se ha incluido un método que será llamado cuando se presione el botón "Info" para obtener la información e insertarla en la caja de texto:




```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk

# La clase 'Aplicacion' ha crecido. En el ejemplo se incluyen
# nuevos widgets en el método constructor __init__(): Uno de
# ellos es el botón 'Info'  que cuando sea presionado llamará
# al método 'verinfo' para mostrar información en el otro
# widget, una caja de texto: un evento ejecuta una acción:

class Aplicacion():
    def __init__(self):

        # En el ejemplo se utiliza el prefijo 'self' para
        # declarar algunas variables asociadas al objeto
        # ('mi_app')  de la clase 'Aplicacion'. Su uso es
        # imprescindible para que se pueda acceder a sus
        # valores desde otros métodos:

        self.raiz = Tk()
        self.raiz.geometry('300x200')
```

```
        # Impide que los bordes puedan desplazarse para
        # ampliar o reducir el tamaño de la ventana 'self.raiz':

        self.raiz.resizable(width=False,height=False)
        self.raiz.title('Ver info')

        # Define el widget Text 'self.tinfo ' en el que se
        # pueden introducir varias líneas de texto:

        self.tinfo = Text(self.raiz, width=40, height=10)

        # Sitúa la caja de texto 'self.tinfo' en la parte
        # superior de la ventana 'self.raiz':

        self.tinfo.pack(side=TOP)

        # Define el widget Button 'self.binfo' que llamará
        # al metodo 'self.verinfo' cuando sea presionado

        self.binfo = ttk.Button(self.raiz, text='Info',
                                command=self.verinfo)

        # Coloca el botón 'self.binfo' debajo y a la izquierda
        # del widget anterior

        self.binfo.pack(side=LEFT)

        # Define el botón 'self.bsalir'. En este caso
        # cuando sea presionado, el método destruirá o
        # terminará la aplicación-ventana 'self.raíz' con
        # 'self.raiz.destroy'

        self.bsalir = ttk.Button(self.raiz, text='Salir',
                                 command=self.raiz.destroy)

        # Coloca el botón 'self.bsalir' a la derecha del
        # objeto anterior.

        self.bsalir.pack(side=RIGHT)

        # El foco de la aplicación se sitúa en el botón
        # 'self.binfo' resaltando su borde. Si se presiona
        # la barra espaciadora el botón que tiene el foco
        # será pulsado. El foco puede cambiar de un widget
        # a otro con la tecla tabulador [tab]
```

```
        self.binfo.focus_set()
        self.raiz.mainloop()

    def verinfo(self):

        # Borra el contenido que tenga en un momento dado
        # la caja de texto

        self.tinfo.delete("1.0", END)

        # Obtiene información de la ventana 'self.raiz':

        info1 = self.raiz.winfo_class()
        info2 = self.raiz.winfo_geometry()
        info3 = str(self.raiz.winfo_width())
        info4 = str(self.raiz.winfo_height())
        info5 = str(self.raiz.winfo_rootx())
        info6 = str(self.raiz.winfo_rooty())
        info7 = str(self.raiz.winfo_id())
        info8 = self.raiz.winfo_name()
        info9 = self.raiz.winfo_manager()

        # Construye una cadena de texto con toda la
        # información obtenida:

        texto_info = "Clase de 'raiz': " + info1 + "\n"
        texto_info += "Resolución y posición: " + info2 + "\n"
        texto_info += "Anchura ventana: " + info3 + "\n"
        texto_info += "Altura ventana: " + info4 + "\n"
        texto_info += "Pos. Ventana X: " + info5 + "\n"
        texto_info += "Pos. Ventana Y: " + info6 + "\n"
        texto_info += "Id. de 'raiz': " + info7 + "\n"
        texto_info += "Nombre objeto: " + info8 + "\n"
        texto_info += "Gestor ventanas: " + info9 + "\n"

        # Inserta la información en la caja de texto:

        self.tinfo.insert("1.0", texto_info)

def main():
    mi_app = Aplicacion()
    return 0

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

En la aplicación se utiliza el método pack() para ubicar los widgets en una posición determinada dentro de la ventana. Dicho método da nombre a uno de los tres gestores de geometría existentes en Tkinter, que son los responsables de esta tarea.

Tkinter: Diseñando ventanas gráficas

Introducción

Para definir el modo en que deben colocarse los widgets (controles) dentro de una ventana se utilizan los gestores de geometría. En Tkinter existen tres gestores de geometría: **pack**, **grid** y **place**.

Si una aplicación tiene varias ventanas, cada una de ellas puede estar construida con cualquiera de estos gestores, indistintamente. Será el desarrollador el que tendrá que elegir el que mejor resuelva el diseño que tenga por delante en cada momento.

También, indicar que para construir las ventanas se pueden utilizar unos widgets especiales (marcos, paneles, etc.) que actúan como contenedores de otros widgets. Estos widgets se utilizan para agrupar varios controles al objeto de facilitar la operación a los usuarios. En las ventanas que se utilicen podrá emplearse un gestor con la ventana y otro diferente para organizar los controles dentro de estos widgets.

A continuación, vamos a conocer las características de los tres gestores geométricos y a desarrollar una misma aplicación, utilizando cada uno de ellos.

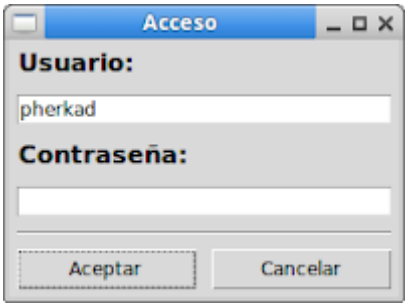
La aplicación consta de una ventana típica de acceso a un sistema que muestra en una caja de entrada la cuenta del usuario actual del equipo y presenta otra caja para introducir su contraseña. En la parte inferior hay dos botones: uno con el texto 'Aceptar' para validar la contraseña (mediante la llamada a un método) y otro con 'Cancelar' para finalizar la aplicación.

El gestor de geometría Pack

Con este gestor la organización de los widgets se hace teniendo en cuenta los lados de una ventana: arriba, abajo, derecha e izquierda.

Si varios controles se ubican (todos) en el lado de arriba o (todos) en el lado izquierdo de una ventana, construiremos una barra vertical o una barra horizontal de controles. Aunque es ideal para diseños simples (barras de herramientas, cuadros de diálogos, etc.) se puede utilizar también con diseños complejos. Además, es posible hacer que los controles se ajusten a los cambios de tamaño de la ventana.

El ejemplo muestra la aplicación comentada con su ventana construida con el gestor pack:



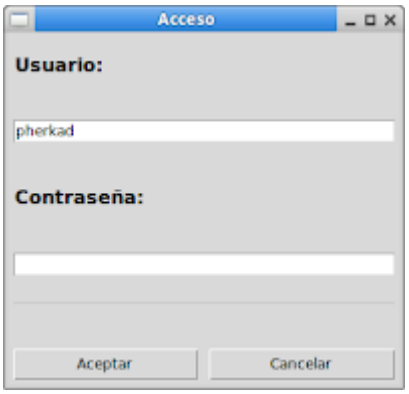

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk, font
import getpass

# Gestor de geometría (pack)

class Aplicacion():
    def __init__(self):
        self.raiz = Tk()
        self.raiz.title("Acceso")

        # Cambia el formato de la fuente actual a negrita para
        # resaltar las dos etiquetas que acompañan a las cajas
        # de entrada. (Para este cambio se ha importado el
        # módulo 'font' al comienzo del programa):

        fuente = font.Font(weight='bold')

        # Define las etiquetas que acompañan a las cajas de
        # entrada y asigna el formato de fuente anterior:

        self.etiq1 = ttk.Label(self.raiz, text="Usuario:",
                               font=fuente)
        self.etiq2 = ttk.Label(self.raiz, text="Contraseña:",
                               font=fuente)

        # Declara dos variables de tipo cadena para contener
        # el usuario y la contraseña:

        self.usuario = StringVar()
        self.clave = StringVar()

        # Realiza una lectura del nombre de usuario que
        # inició sesión en el sistema y lo asigna a la
        # variable 'self.usuario' (Para capturar esta
```

```
        # información se ha importando el módulo getpass
        # al comienzo del programa):

        self.usuario.set(getpass.getuser())

        # Define dos cajas de entrada que aceptarán cadenas
        # de una longitud máxima de 30 caracteres.
        # A la primera de ellas 'self.ctext1' que contendrá
        # el nombre del usuario, se le asigna la variable
        # 'self.usuario' a la opción 'textvariable'. Cualquier
        # valor que tome la variable durante la ejecución del
        # programa quedará reflejada en la caja de entrada.
        # En la segunda caja de entrada, la de la contraseña,
        # se hace lo mismo. Además, se establece la opción
        # 'show' con un "*" (asterisco) para ocultar la
        # escritura de las contraseñas:

        self.ctext1 = ttk.Entry(self.raiz,
                                textvariable=self.usuario,
                                width=30)
        self.ctext2 = ttk.Entry(self.raiz,
                                textvariable=self.clave,
                                width=30, show="*")
        self.separ1 = ttk.Separator(self.raiz, orient=HORIZONTAL)

        # Se definen dos botones con dos métodos: El botón
        # 'Aceptar' llamará al método 'self.aceptar' cuando
        # sea presionado para validar la contraseña; y el botón
        # 'Cancelar' finalizará la aplicación si se llega a
        # presionar:

        self.boton1 = ttk.Button(self.raiz, text="Aceptar",
                                 command=self.aceptar)
        self.boton2 = ttk.Button(self.raiz, text="Cancelar",
                                 command=quit)

        # Se definen las posiciones de los widgets dentro de
        # la ventana. Todos los controles se van colocando
        # hacia el lado de arriba, excepto, los dos últimos,
        # los botones, que se situarán debajo del último 'TOP':
        # el primer botón hacia el lado de la izquierda y el
        # segundo a su derecha.
        # Los valores posibles para la opción 'side' son:
        # TOP (arriba), BOTTOM (abajo), LEFT (izquierda)
        # y RIGHT (derecha). Si se omite, el valor será TOP
        # La opción 'fill' se utiliza para indicar al gestor
```

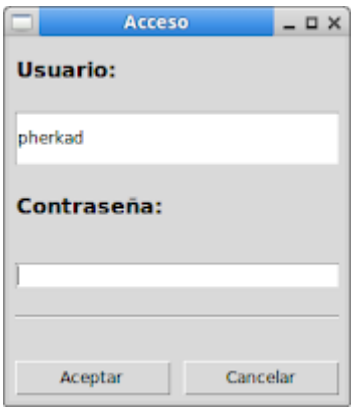
```
        # cómo expandir/reducir el widget si la ventana cambia
        # de tamaño. Tiene tres posibles valores: BOTH
        # (Horizontal y Verticalmente), X (Horizontalmente) e
        # Y (Verticalmente). Funcionará si el valor de la opción
        # 'expand' es True.
        # Por último, las opciones 'padx' y 'pady' se utilizan
        # para añadir espacio extra externo horizontal y/o
        # vertical a los widgets para separarlos entre sí y de
        # los bordes de la ventana. Hay otras equivalentes que
        # añaden espacio extra interno: 'ipàdx' y 'ipady':

        self.etiq1.pack(side=TOP, fill=BOTH, expand=True,
                        padx=5, pady=5)
        self.ctext1.pack(side=TOP, fill=X, expand=True,
                         padx=5, pady=5)
        self.etiq2.pack(side=TOP, fill=BOTH, expand=True,
                        padx=5, pady=5)
        self.ctext2.pack(side=TOP, fill=X, expand=True,
                         padx=5, pady=5)
        self.separ1.pack(side=TOP, fill=BOTH, expand=True,
                         padx=5, pady=5)
        self.boton1.pack(side=LEFT, fill=BOTH, expand=True,
                         padx=5, pady=5)
        self.boton2.pack(side=RIGHT, fill=BOTH, expand=True,
                         padx=5, pady=5)

        # Cuando se inicia el programa se asigna el foco
        # a la caja de entrada de la contraseña para que se
        # pueda empezar a escribir directamente:

        self.ctext2.focus_set()

        self.raiz.mainloop()

    # El método 'aceptar' se emplea para validar la
    # contraseña introducida. Será llamado cuando se
    # presione el botón 'Aceptar'. Si la contraseña
    # coincide con la cadena 'tkinter' se imprimirá
    # el mensaje 'Acceso permitido' y los valores
    # aceptados. En caso contrario, se mostrará el
    # mensaje 'Acceso denegado' y el foco volverá al
    # mismo lugar.

    def aceptar(self):
        if self.clave.get() == 'tkinter':
            print("Acceso permitido")
```

```
            print("Usuario:   ", self.ctext1.get())
            print("Contraseña:", self.ctext2.get())
        else:
            print("Acceso denegado")

            # Se inicializa la variable 'self.clave' para
            # que el widget 'self.ctext2' quede limpio.
            # Por último, se vuelve a asignar el foco
            # a este widget para poder escribir una nueva
            # contraseña.

            self.clave.set("")
            self.ctext2.focus_set()

def main():
    mi_app = Aplicacion()
    return 0

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

Como hemos comentado antes la aplicación permite cambiar la dimensión de la ventana. Si lo hacemos los widgets se adaptarán al nuevo tamaño, teniendo en cuenta la configuración particular de cada uno de ellos. Para comprobar el funcionamiento podemos arrastrar los bordes de la ventana para ampliar o reducir el tamaño y comprobar cómo trabaja el gestor **pack**:



También, para verificar cómo actúa la opción **fill** podemos cambiar el valor actual **X** del widtget **self.ctext1.pack** por Y y arrastrar los bordes de la ventana. Si arrastramos hacia abajo el widget se expandirá verticalmente:



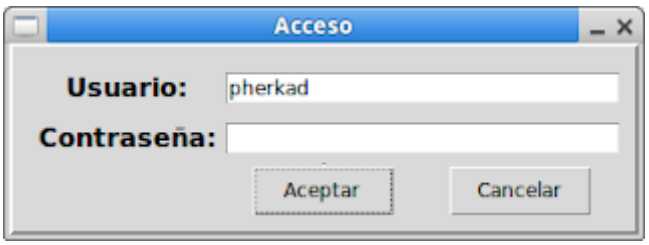
El gestor de geometría Grid

Este gestor geométrico trata una ventana como si fuera una cuadrícula, formada por filas y columnas como un tablero de ajedrez, donde es posible situar mediante una coordenada (fila, columna) los widgets; teniendo en cuenta que, si se requiere, un widget puede ocupar varias columnas y/o varias filas.

Con este gestor es posible construir ventanas complejas y hacer que los controles se ajusten a un nuevo tamaño de las mismas. Se recomienda su uso con diseños en los que los controles deben aparecer alineados en varias columnas o filas, es decir, siguiendo la forma de una tabla.

Grid con ventana no dimensionable

El siguiente ejemplo pretende ilustrar cómo usar el gestor grid con una ventana no dimensionable. También, utiliza un widget **Frame** con efecto 3D que contendrá al resto de controles:



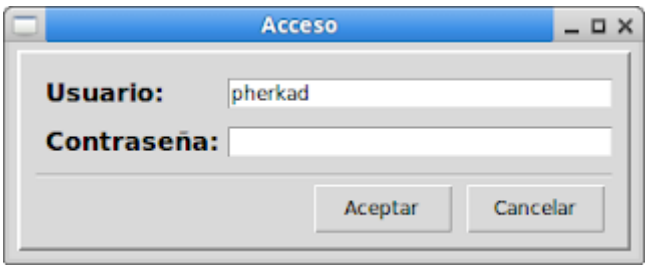

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk, font
import getpass

# Gestor de geometría (grid). Ventana no dimensionable

class Aplicacion():
```

```
    def __init__(self):
        self.raiz = Tk()
        self.raiz.title("Acceso")

        # Establece que no se pueda modificar el tamaño de la
        # ventana. El método resizable(0,0) es la forma abreviada
        # de resizable(width=False,height=False).

        self.raiz.resizable(0,0)
        fuente = font.Font(weight='bold')

        # Define un widget de tipo 'Frame' (marco) que será el
        # contenedor del resto de widgets. El marco se situará
        # en la ventana 'self.raiz' ocupando toda su extensión.
        # El marco se define con un borde de 2 píxeles y la
        # opción 'relief' con el valor 'raised' (elevado) añade
        # un efecto 3D a su borde.
        # La opción 'relief' permite los siguientes valores:
        # FLAT (llano), RAISED (elevado), SUNKEN (hundido),
        # GROOVE (hendidura) y RIDGE (borde elevado).
        # La opción 'padding' añade espacio extra interior para
        # que los widgets no queden pegados al borde del marco.

        self.marco = ttk.Frame(self.raiz, borderwidth=2,
                               relief="raised", padding=(10,10))

        # Define el resto de widgets pero en este caso el primer
        # paràmetro indica que se situarán en el widget del
        # marco anterior 'self.marco'.

        self.etiq1 = ttk.Label(self.marco, text="Usuario:",
                               font=fuente, padding=(5,5))
        self.etiq2 = ttk.Label(self.marco, text="Contraseña:",
                               font=fuente, padding=(5,5))

        # Define variables para las opciones 'textvariable' de
        # cada caja de entrada 'ttk.Entry()'.

        self.usuario = StringVar()
        self.clave = StringVar()
        self.usuario.set(getpass.getuser())
        self.ctext1 = ttk.Entry(self.marco, textvariable=self.usuario,
                                width=30)
        self.ctext2 = ttk.Entry(self.marco, textvariable=self.clave,
                                show="*",
                                width=30)
```

```
        self.separ1 = ttk.Separator(self.marco, orient=HORIZONTAL)
        self.boton1 = ttk.Button(self.marco, text="Aceptar",
                                 padding=(5,5), command=self.aceptar)
        self.boton2 = ttk.Button(self.marco, text="Cancelar",
                                 padding=(5,5), command=quit)

        # Define la ubicación de cada widget en el grid.
        # En este ejemplo en realidad hay dos grid (cuadrículas):
        # Una cuadrícula de 1fx1c que se encuentra en la ventana
        # que ocupará el Frame; y otra en el Frame de 5fx3c para
        # el resto de controles.
        # La primera fila y primera columna serán la número 0.
        # La opción 'column' indica el número de columna y la
        # opción 'row' indica el número de fila donde hay que
        # colocar un widget.
        # La opción 'columnspan' indica al gestor que el
        # widget ocupará en total un número determinado de
        # columnas. Las cajas para entradas 'self.ctext1' y
        # 'self.ctext2' ocuparán dos columnas y la barra
        # de separación 'self.separ1' tres.

        self.marco.grid(column=0, row=0)
        self.etiq1.grid(column=0, row=0)
        self.ctext1.grid(column=1, row=0, columnspan=2)
        self.etiq2.grid(column=0, row=1)
        self.ctext2.grid(column=1, row=1, columnspan=2)
        self.separ1.grid(column=0, row=3, columnspan=3)
        self.boton1.grid(column=1, row=4)
        self.boton2.grid(column=2, row=4)

        # Establece el foco en la caja de entrada de la
        # contraseña.

        self.ctext2.focus_set()
        self.raiz.mainloop()

    def aceptar(self):
        if self.clave.get() == 'tkinter':
            print("Acceso permitido")
            print("Usuario:   ", self.ctext1.get())
            print("Contraseña:", self.ctext2.get())
        else:
            print("Acceso denegado")
            self.clave.set("")
            self.ctext2.focus_set()
```

```
def main():
    mi_app = Aplicacion()
    return 0

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

Grid con ventana redimensionable



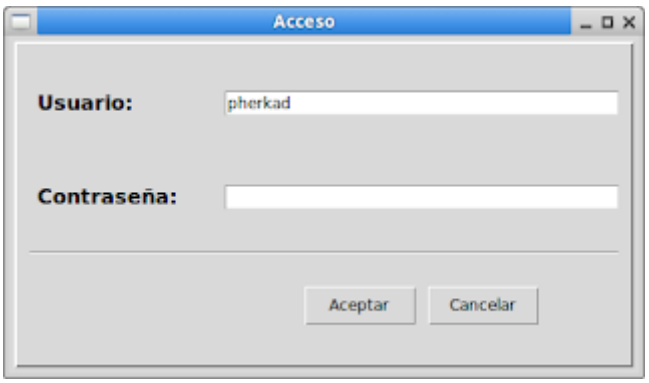
A continuación, la aplicación se implementa con **grid** con la posibilidad de adaptar los widgets al espacio de la ventana, cuando cambie de tamaño:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk, font
import getpass

# Gestor de geometría (grid). Ventana redimensionable

class Aplicacion():
    def __init__(self):
        self.raiz = Tk()
        self.raiz.title("Acceso")
        fuente = font.Font(weight='bold')
        self.marco = ttk.Frame(self.raiz, borderwidth=2,
                               relief="raised", padding=(10,10))
        self.etiq1 = ttk.Label(self.marco, text="Usuario:",
                               font=fuente, padding=(5,5))
        self.etiq2 = ttk.Label(self.marco, text="Contraseña:",
                               font=fuente, padding=(5,5))
        self.usuario = StringVar()
        self.clave = StringVar()
        self.usuario.set(getpass.getuser())
        self.ctext1 = ttk.Entry(self.marco, textvariable=self.usuario,
                                width=30)
```

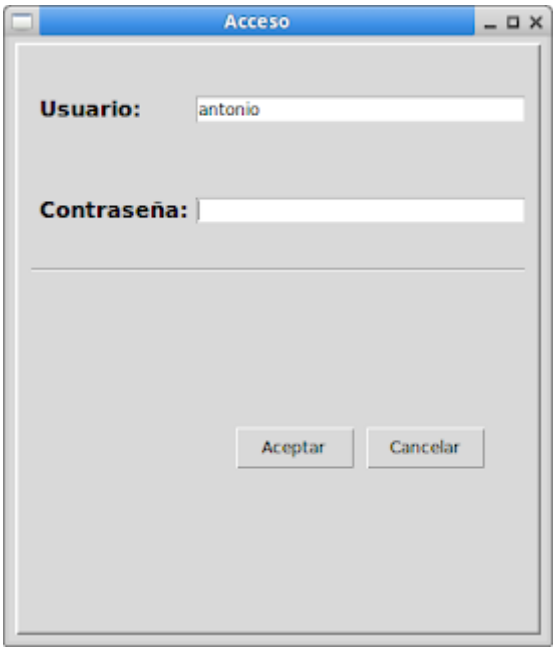
```
        self.ctext2 = ttk.Entry(self.marco, textvariable=self.clave,
                                show="*", width=30)
        self.separ1 = ttk.Separator(self.marco, orient=HORIZONTAL)
        self.boton1 = ttk.Button(self.marco, text="Aceptar",
                                 padding=(5,5), command=self.aceptar)
        self.boton2 = ttk.Button(self.marco, text="Cancelar",
                                 padding=(5,5), command=quit)

        # Para conseguir que la cuadrícula y los widgets se
        # adapten al contenedor, si se amplía o reduce el tamaño
        # de la ventana, es necesario definir la opción 'sticky'.
        # Cuando un widget se ubica en el grid se coloca en el
        # centro de su celda o cuadro. Con 'sticky' se
        # establece el comportamiento 'pegajoso' que tendrá el
        # widget dentro de su celda, cuando se modifique la
        # dimensión de la ventana. Para ello, se utilizan para
        # expresar sus valores los puntos cardinales: N (Norte),
        # S (Sur), (E) Este y (W) Oeste, que incluso se pueden
        # utilizar de forma combinada. El widget se quedará
        # 'pegado' a los lados de su celda en las direcciones
        # que se indiquen. cuando la ventana cambie de tamaño.
        # Pero con definir la opción 'sticky' no es suficiente:
        # hay activar esta propiedad más adelante.

        self.marco.grid(column=0, row=0, padx=5, pady=5,
                        sticky=(N, S, E, W))
        self.etiq1.grid(column=0, row=0,
                        sticky=(N, S, E, W))
        self.ctext1.grid(column=1, row=0, columnspan=2,
                         sticky=(E, W))
        self.etiq2.grid(column=0, row=1,
                        sticky=(N, S, E, W))
        self.ctext2.grid(column=1, row=1, columnspan=2,
                         sticky=(E, W))
        self.separ1.grid(column=0, row=3, columnspan=3, pady=5,
                         sticky=(N, S, E, W))
        self.boton1.grid(column=1, row=4, padx=5,
                         sticky=(E))
        self.boton2.grid(column=2, row=4, padx=5,
                         sticky=(W))

        # A continuación, se activa la propiedad de expandirse
        # o contraerse definida antes con la opción
        # 'sticky' del método grid().
        # La activación se hace por contenedores y por filas
        # y columnas asignando un peso a la opción 'weight'.
```

```
        # Esta opción asigna un peso (relativo) que se utiliza
        # para distribuir el espacio adicional entre columnas
        # y/o filas. Cuando se expanda la ventana, una columna
        # o fila con un peso 2 crecerá dos veces más rápido
        # que una columna (o fila) con peso 1. El valor
        # predeterminado es 0 que significa que la columna o
        # o fila no crecerá nada en absoluto.
        # Lo habitual es asignar pesos a filas o columnas donde
        # hay celdas con widgets.

        self.raiz.columnconfigure(0, weight=1)
        self.raiz.rowconfigure(0, weight=1)
        self.marco.columnconfigure(0, weight=1)
        self.marco.columnconfigure(1, weight=1)
        self.marco.columnconfigure(2, weight=1)
        self.marco.rowconfigure(0, weight=1)
        self.marco.rowconfigure(1, weight=1)
        self.marco.rowconfigure(4, weight=1)

        # Establece el foco en la caja de entrada de la
        # contraseña.

        self.ctext2.focus_set()
        self.raiz.mainloop()

    def aceptar(self):
        if self.clave.get() == 'tkinter':
            print("Acceso permitido")
            print("Usuario:   ", self.ctext1.get())
            print("Contraseña:", self.ctext2.get())
        else:
            print("Acceso denegado")
            self.clave.set("")
            self.ctext2.focus_set()

def main():
    mi_app = Aplicacion()
    return 0

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

Después de ejecutar la aplicación, si ampliamos el tamaño de la ventana podemos comprobar cómo se ajustan los controles al nuevo espacio disponible según las direcciones descritas en cada opción **sticky**:



También, para ver el funcionamiento de los pesos cambiaremos el peso que se asigna a la fila 4, que es donde se encuentran los botones **'Aceptar'** y **'Cancelar'**, con el valor 5 para multiplicar por 5 el espacio a añadir en esta fila cuando se expanda la ventana:



El gestor de geometría Place

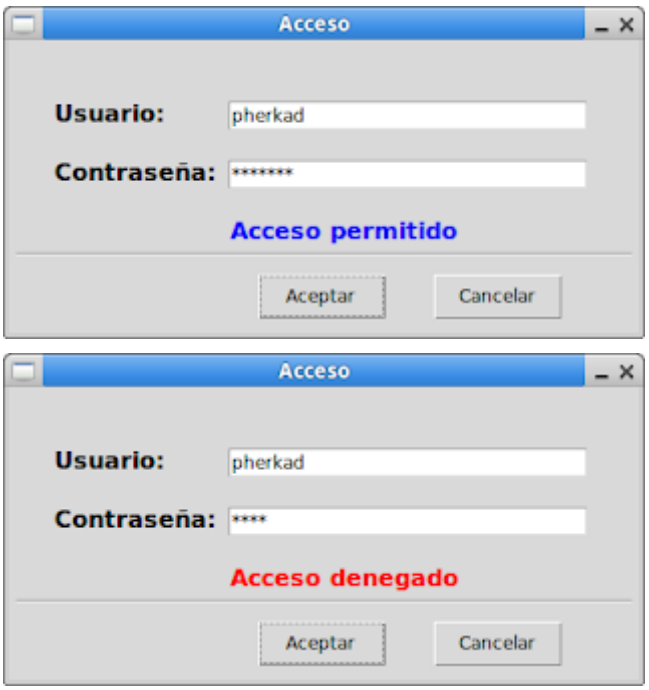
Este gestor es el más fácil de utilizar porque se basa en el posicionamiento absoluto para colocar los widgets, aunque el trabajo de "calcular" la posición de cada widget suele ser bastante laborioso. Sabemos que una ventana tiene una anchura y una altura determinadas (normalmente, medida en píxeles). Pues bien, con este método para colocar un widget simplemente tendremos que elegir la coordenada (x,y) de su ubicación expresada en píxeles.

La posición (x=0, y=0) se encuentra en la esquina superior-izquierda de la ventana.

Con este gestor el tamaño y la posición de un widget no cambiará al modificar las dimensiones de una ventana.

Para finalizar, mostramos la famosa aplicación realizada con el gestor de geometría **place**. En este caso el modo de mostrar el mensaje de la validación se hace utilizando una etiqueta

que cambia de color dependiendo si la contraseña es correcta o no. También, utiliza un método adicional para "limpiar" el mensaje de error cuando se hace clic con el ratón en la caja de entrada de la contraseña. El evento del widget se asocia con el método utilizando el método **bind()**.




```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk, font
import getpass

# Gestor de geometría (place)

class Aplicacion():
    def __init__(self):
        self.raiz = Tk()

        # Define la dimensión de la ventana

        self.raiz.geometry("430x200")

        # Establece que no se pueda cambiar el tamaño de la
        # ventana

        self.raiz.resizable(0,0)
        self.raiz.title("Acceso")
        self.fuente = font.Font(weight='bold')
```

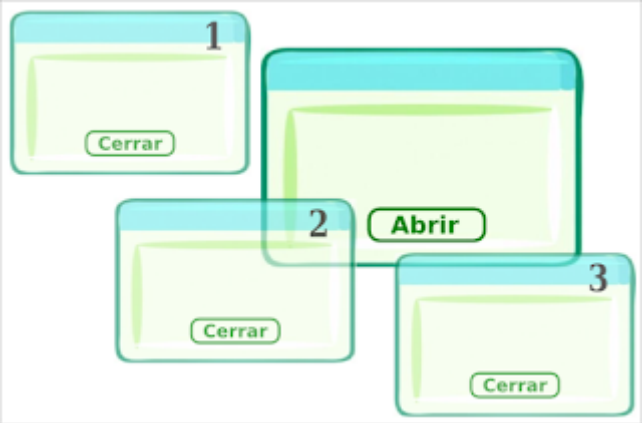
```
        self.etiq1 = ttk.Label(self.raiz, text="Usuario:",
                               font=self.fuente)
        self.etiq2 = ttk.Label(self.raiz, text="Contraseña:",
                               font=self.fuente)

        # Declara una variable de cadena que se asigna a
        # la opción 'textvariable' de un widget 'Label' para
        # mostrar mensajes en la ventana. Se asigna el color
        # azul a la opción 'foreground' para el mensaje.

        self.mensa = StringVar()
        self.etiq3 = ttk.Label(self.raiz, textvariable=self.mensa,
                    font=self.fuente, foreground='blue')

        self.usuario = StringVar()
        self.clave = StringVar()
        self.usuario.set(getpass.getuser())
        self.ctext1 = ttk.Entry(self.raiz,
                               textvariable=self.usuario, width=30)
        self.ctext2 = ttk.Entry(self.raiz,
                               textvariable=self.clave,
                               width=30,
                               show="*")
        self.separ1 = ttk.Separator(self.raiz, orient=HORIZONTAL)
        self.boton1 = ttk.Button(self.raiz, text="Aceptar",
                                 padding=(5,5), command=self.aceptar)
        self.boton2 = ttk.Button(self.raiz, text="Cancelar",
                                 padding=(5,5), command=quit)

        # Se definen las ubicaciones de los widgets en la
        # ventana asignando los valores de las opciones 'x' e 'y'
        # en píxeles.

        self.etiq1.place(x=30, y=40)
        self.etiq2.place(x=30, y=80)
        self.etiq3.place(x=150, y=120)
        self.ctext1.place(x=150, y=42)
        self.ctext2.place(x=150, y=82)
        self.separ1.place(x=5, y=145, bordermode=OUTSIDE,
                          height=10, width=420)
        self.boton1.place(x=170, y=160)
        self.boton2.place(x=290, y=160)
        self.ctext2.focus_set()

        # El método 'bind()' asocia el evento de 'hacer clic
        # con el botón izquierdo del ratón en la caja de entrada
```

```
        # de la contraseña' expresado con '<button-1>' con el
        # método 'self.borrar_mensa' que borra el mensaje y la
        # contraseña y devuelve el foco al mismo control.
        # Otros ejemplos de acciones que se pueden capturar:
        # <double-button-1>, <buttonrelease-1>, <enter>, <leave>,
        # <focusin>, <focusout>, <return>, <shift-up>, <key-f10>,
        # <key-space>, <key-print>, <keypress-h>, etc.

        self.ctext2.bind('<button-1>', self.borrar_mensa)
        self.raiz.mainloop()

    # Declara método para validar la contraseña y mostrar
    # un mensaje en la propia ventana, utilizando la etiqueta
    # 'self.mensa'. Cuando la contraseña es correcta se
    # asigna el color azul a la etiqueta 'self.etiq3' y
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk, font
import getpass

# Gestor de geometría (place)

class Aplicacion():
    def __init__(self):
        self.raiz = Tk()

        # Define la dimensión de la ventana

        self.raiz.geometry("430x200")

        # Establece que no se pueda cambiar el tamaño de la
        # ventana

        self.raiz.resizable(0,0)
        self.raiz.title("Acceso")
        self.fuente = font.Font(weight='bold')
        self.etiq1 = ttk.Label(self.raiz, text="Usuario:",
                               font=self.fuente)
        self.etiq2 = ttk.Label(self.raiz, text="Contraseña:",
                               font=self.fuente)

        # Declara una variable de cadena que se asigna a
        # la opción 'textvariable' de un widget 'Label' para
        # mostrar mensajes en la ventana. Se asigna el color
```

```
        # azul a la opción 'foreground' para el mensaje.

        self.mensa = StringVar()
        self.etiq3 = ttk.Label(self.raiz, textvariable=self.mensa,
                     font=self.fuente, foreground='blue')

        self.usuario = StringVar()
        self.clave = StringVar()
        self.usuario.set(getpass.getuser())
        self.ctext1 = ttk.Entry(self.raiz,
                                textvariable=self.usuario, width=30)
        self.ctext2 = ttk.Entry(self.raiz,
                                textvariable=self.clave,
                                width=30,
                                show="*")
        self.separ1 = ttk.Separator(self.raiz, orient=HORIZONTAL)
        self.boton1 = ttk.Button(self.raiz, text="Aceptar",
                                 padding=(5,5), command=self.aceptar)
        self.boton2 = ttk.Button(self.raiz, text="Cancelar",
                                 padding=(5,5), command=quit)

        # Se definen las ubicaciones de los widgets en la
        # ventana asignando los valores de las opciones 'x' e 'y'
        # en píxeles.

        self.etiq1.place(x=30, y=40)
        self.etiq2.place(x=30, y=80)
        self.etiq3.place(x=150, y=120)
        self.ctext1.place(x=150, y=42)
        self.ctext2.place(x=150, y=82)
        self.separ1.place(x=5, y=145, bordermode=OUTSIDE,
                          height=10, width=420)
        self.boton1.place(x=170, y=160)
        self.boton2.place(x=290, y=160)
        self.ctext2.focus_set()

        # El método 'bind()' asocia el evento de 'hacer clic
        # con el botón izquierdo del ratón en la caja de entrada
        # de la contraseña' expresado con '<button-1>' con el
        # método 'self.borrar_mensa' que borra el mensaje y la
        # contraseña y devuelve el foco al mismo control.
        # Otros ejemplos de acciones que se pueden capturar:
        # <double-button-1>, <buttonrelease-1>, <enter>, <leave>,
        # <focusin>, <focusout>, <return>, <shift-up>, <key-f10>,
        # <key-space>, <key-print>, <keypress-h>, etc.
```

```
        self.ctext2.bind('<button-1>', self.borrar_mensa)
        self.raiz.mainloop()

    # Declara método para validar la contraseña y mostrar
    # un mensaje en la propia ventana, utilizando la etiqueta
    # 'self.mensa'. Cuando la contraseña es correcta se
    # asigna el color azul a la etiqueta 'self.etiq3' y
    # cuando es incorrecto el color rojo. Para ello. se emplea
    # el método 'configure()' que permite cambiar los valores
    # de las opciones de los widgets.

    def aceptar(self):
        if self.clave.get() == 'tkinter':
            self.etiq3.configure(foreground='blue')
            self.mensa.set("Acceso permitido")
        else:
            self.etiq3.configure(foreground='red')
            self.mensa.set("Acceso denegado")

    # Declara un método para borrar el mensaje anterior y
    # la caja de entrada de la contraseña

    def borrar_mensa(self, evento):
        self.clave.set("")
        self.mensa.set("")

def main():
    mi_app = Aplicacion()
    return 0

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

Tkinter: Tipos de ventanas



Ventanas de aplicación y de diálogos

En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para diseñar las ventanas de una aplicación. A continuación, vamos a explorar los distintos tipos de ventanas que podemos crear y los usos que tienen.

En Tkinter existen dos tipos de ventanas: las ventanas de aplicación, que suelen ser las que inician y finalizan las aplicaciones gráficas; y desde las que se accede a las ventanas de diálogo, que en conjunto constituyen la interfaz de usuario.

Tanto unas como otras, desde el punto de vista del diseño, se construyen exactamente igual con los gestores de geometría ya conocidos: pack, grid y place.

El siguiente ejemplo muestra una ventana de aplicación con un botón 'Abrir' que cada vez que se presione abre una ventana hija (de diálogo) diferente y en una posición diferente. Observaremos que las ventanas hijas que se vayan generando se sitúan siempre en un plano superior con respecto a las creadas con anterioridad. Además, si abrimos varias ventanas podremos interactuar sin problemas con todas ellas, cambiar sus posiciones, cerrarlas, etc.

En este caso, tanto para crear la ventana de aplicación como las hijas se utiliza el gestor de geometría pack pero notaremos algunas diferencias en el código:

La ventana de aplicación se define con **Tk()**:

```
self.raiz = Tk()
```

Las ventanas de diálogo (hijas) se definen con **Toplevel()**:

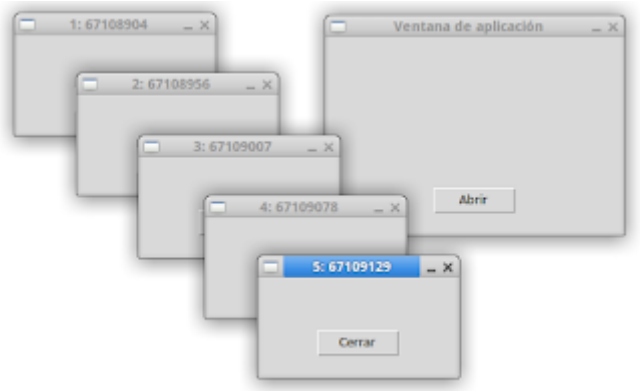
```
self.dialogo = Toplevel()
```

El método **mainloop()** (bucle principal) hace que se atienda a los eventos de la ventana de aplicación:

```
self.raiz.mainloop()
```

El método **wait_window()** hace que se atienda a los eventos locales de la ventana de diálogo mientras espera a ser cerrada:

```
self.raiz.wait_window(self.conectar)
```




```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk

class Aplicacion():
    ''' Clase Aplicacion '''

    # Declara una variable de clase para contar ventanas

    ventana = 0

    # Declara una variable de clase para usar en el
    # cálculo de la posición de una ventana

    posx_y = 0

    def __init__(self):
        ''' Construye ventana de aplicación '''

        # Declara ventana de aplicación

        self.raiz = Tk()
```

```
        # Define dimensión de la ventana 300x200
        # que se situará en la coordenada x=500,y=50

        self.raiz.geometry('300x200+500+50')

        self.raiz.resizable(0,0)
        self.raiz.title("Ventana de aplicación")

        # Define botón 'Abrir' que se utilizará para
        # abrir las ventanas de diálogo. El botón
        # está vinculado con el método 'self.abrir'

        boton = ttk.Button(self.raiz, text='Abrir',
                           command=self.abrir)
        boton.pack(side=BOTTOM, padx=20, pady=20)
        self.raiz.mainloop()

    def abrir(self):
        ''' Construye una ventana de diálogo '''

        # Define una nueva ventana de diálogo

        self.dialogo = Toplevel()

        # Incrementa en 1 el contador de ventanas

        Aplicacion.ventana+=1

        # Recalcula posición de la ventana

        Aplicacion.posx_y += 50
        tamypos = '200x100+'+str(Aplicacion.posx_y)+ \
                  '+'+ str(Aplicacion.posx_y)
        self.dialogo.geometry(tamypos)
        self.dialogo.resizable(0,0)

        # Obtiene identicador de la nueva ventana

        ident = self.dialogo.winfo_id()

        # Construye mensaje de la barra de título

        titulo = str(Aplicacion.ventana)+": "+str(ident)
        self.dialogo.title(titulo)
```

```
        # Define el botón 'Cerrar' que cuando sea
        # presionado cerrará (destruirá) la ventana
        # 'self.dialogo' llamando al método
        # 'self.dialogo.destroy'

        boton = ttk.Button(self.dialogo, text='Cerrar',
                           command=self.dialogo.destroy)
        boton.pack(side=BOTTOM, padx=20, pady=20)

        # Cuando la ejecución del programa llega a este
        # punto se utiliza el método wait_window() para
        # esperar que la ventana 'self.dialogo' sea
        # destruida.
        # Mientras tanto se atiende a los eventos locales
        # que se produzcan, por lo que otras partes de la
        # aplicación seguirán funcionando con normalidad.
        # Si hay código después de esta línea se ejecutará
        # cuando la ventana 'self.dialogo' sea cerrada.

        self.raiz.wait_window(self.dialogo)

def main():
    mi_app = Aplicacion()
    return(0)

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

Ventanas modales y no modales

Las ventanas hijas del ejemplo anterior son del tipo **no modales** porque mientras existen es posible interactuar libremente con ellas, sin ningún límite, excepto que si cerramos la ventana principal se cerrarán todas las ventanas hijas abiertas.

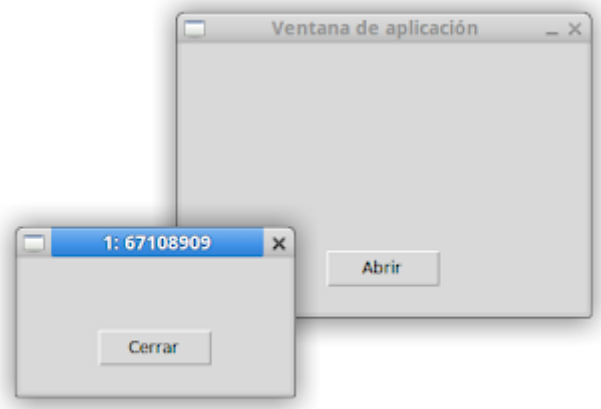
Un ejemplo evidente que usa ventanas no modales está en las aplicaciones ofimáticas más conocidas, que permiten trabajar con varios documentos al mismo tiempo, cada uno de ellos abierto en su propia ventana, permitiendo al usuario cambiar sin restricciones de una ventana a otra.

El caso contrario, es el de las **ventanas modales**. Cuando una ventana modal está abierta no será posible interactuar con otras ventanas de la aplicación hasta que ésta sea cerrada.

Un ejemplo típico es el de algunas ventanas de diálogo que se utilizan para establecer las preferencias de las aplicaciones, que obligan a ser cerradas antes de permitirse la apertura de otras.

Para demostrarlo, utilizamos el siguiente ejemplo en el que sólo es posible mantener abierta sólo una ventana hija, aunque si la cerramos podremos abrir otra.

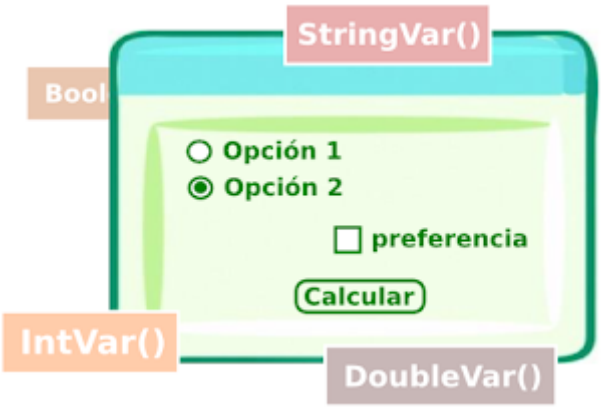
El método **grab_set()** se utiliza para crear la ventana modal y el método **transiet()** se emplea para convertir la ventana de diálogo en **ventana transitoria**, haciendo que se oculte cuando la ventana de aplicación sea minimizada.




```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk

class Aplicacion():
    ventana = 0
    posx_y = 0

    def __init__(self):
        self.raiz = Tk()
        self.raiz.geometry('300x200+500+50')
        self.raiz.resizable(0,0)
        self.raiz.title("Ventana de aplicación")
        boton = ttk.Button(self.raiz, text='Abrir',
                           command=self.abrir)
        boton.pack(side=BOTTOM, padx=20, pady=20)
        self.raiz.mainloop()

    def abrir(self):
        ''' Construye una ventana de diálogo '''

        self.dialogo = Toplevel()
        Aplicacion.ventana+=1
        Aplicacion.posx_y += 50
```

```
        tamypos = '200x100+'+str(Aplicacion.posx_y)+ \
                  '+'+ str(Aplicacion.posx_y)
        self.dialogo.geometry(tamypos)
        self.dialogo.resizable(0,0)
        ident = self.dialogo.winfo_id()
        titulo = str(Aplicacion.ventana)+": "+str(ident)
        self.dialogo.title(titulo)
        boton = ttk.Button(self.dialogo, text='Cerrar',
                           command=self.dialogo.destroy)
        boton.pack(side=BOTTOM, padx=20, pady=20)

        # Convierte la ventana 'self.dialogo' en
        # transitoria con respecto a su ventana maestra
        # 'self.raiz'.
        # Una ventana transitoria siempre se dibuja sobre
        # su maestra y se ocultará cuando la maestra sea
        # minimizada. Si el argumento 'master' es
        # omitido el valor, por defecto, será la ventana
        # madre.

        self.dialogo.transient(master=self.raiz)

        # El método grab_set() asegura que no haya eventos
        # de ratón o teclado que se envíen a otra ventana
        # diferente a 'self.dialogo'. Se utiliza para
        # crear una ventana de tipo modal que será
        # necesario cerrar para poder trabajar con otra
        # diferente. Con ello, también se impide que la
        # misma ventana se abra varias veces.

        self.dialogo.grab_set()
        self.raiz.wait_window(self.dialogo)

def main():
    mi_app = Aplicacion()
    return(0)

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

Variables de control en Tkinter



Variables de control

Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valores y facilitar su disponibilidad en otras partes del programa. Pueden ser de tipo numérico, de cadena y booleano.

Cuando una variable de control cambia de valor el widget que la utiliza lo refleja automáticamente, y viceversa.

Las variables de control también se emplean para conectar varios widgets del mismo tipo, por ejemplo, varios controles del tipo **Radiobutton**. En este caso tomarán un valor de varios posibles.

Declarar variables de control

Las variables de control se declaran de forma diferente en función al tipo de dato que almacenan:

```
entero = IntVar()  # Declara variable de tipo entera
flotante = DoubleVar()  # Declara variable de tipo flotante
cadena = StringVar()  # Declara variable de tipo cadena
booleano = BooleanVar()  # Declara variable de tipo booleana
```

También, en el momento de declarar una variable es posible asignar un valor inicial:

```
blog = StringVar(value="Python para impacientes")
```

Método set()

El método **set()** asigna un valor a una variable de control. Se utiliza para modificar el valor o estado de un widget:

```
nombre = StringVar()
id_art = IntVar()
nombre.set("Python para impacientes")
id_art.set(1)
blog = ttk.Entry(ventana, textvariable=nombre, width=25)
arti = ttk.Label(ventana, textvariable=id_art)
```

Método get()

El método **get()** obtiene el valor que tenga, en un momento dado, una variable de control. Se utiliza cuando es necesario leer el valor de un control:

```
print('Blog:', nombre.get())
print('Id artículo:', id_art.get())
```

Método trace()

El método **trace()** se emplea para "detectar" cuando una variable es leída, cambia de valor o es borrada:

```
widget.trace(tipo, función)
```

El primer argumento establece el tipo de suceso a comprobar: '**r**' lectura de variable, '**w**' escritura de variable y '**u**' borrado de variable. El segundo argumento indica la función que será llamada cuando se produzca el suceso.

En el siguiente ejemplo se define una variable de control de tipo cadena y con el método **trace()** se asocian su lectura y cambio de valor a dos funciones que son llamadas cuando ocurran estos sucesos. Concretamente, cuando se utilice el método **set()** se llamará a la función '**cambia()**' y cuando se use **get()** a la función '**lee()**'.

```
def cambia(*args):
    print("Ha cambiado su valor")

def lee(*args):
    print("Ha sido leido su valor")
```

```
variable = StringVar()
variable.trace("w", cambia)
variable.trace("r", lee)
variable.set("Hola")
print(variable.get())
print(variable.get())
```

Estrategias para validar y calcular datos

Cuando se construye una ventana con varios widgets se pueden seguir distintas estrategias para validar los datos que se introducen durante la ejecución de un programa:

- Una opción posible podría ser validar la información y realizar los cálculos después de que sea introducida, por ejemplo, después de presionar un botón.
- Otra posibilidad podría ser hacer uso del método **trace()** y de la opción **'command'**, para validar y calcular la información justo en el momento que un widget y su variable asociada cambien de valor.

A continuación, se muestra un mismo caso práctico utilizando ambas técnicas.

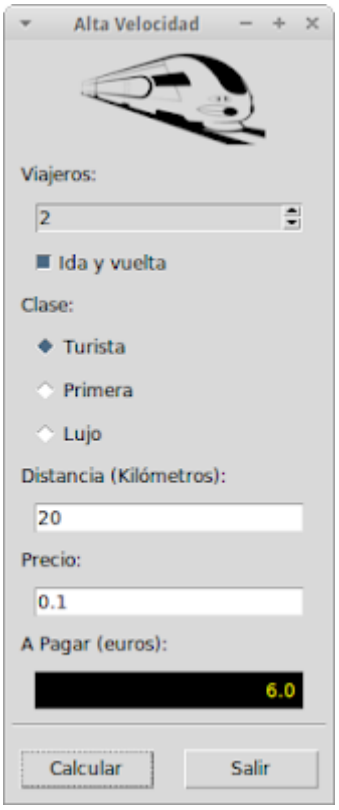
El programa de este ejemplo calcula el coste de un viaje en tren teniendo en cuenta el número de viajeros; el tipo de billete (de sólo ida o de ida y vuelta); la clase en la cual se viaja (que puede ser clase turista, primera o lujo); la distancia en kilómetros y el precio por kilómetro (por defecto es 0,10 céntimos de euro).

El cálculo del importe a pagar se realiza multiplicando número de viajeros por km y precio, con los siguientes incrementos:

- Si el viaje es de ida y vuelta se multiplica el total por 1,5
- Si la clase es primera se multiplica el total por 1,2 y si es de lujo se multiplica por 2

Validación y cálculo posterior

En la primera solución la validación de los datos y el cálculo del importe a pagar se realiza después de presionar el botón "Calcular".




```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
from tkinter import ttk

# Calcula coste de viaje con validación y cálculo posterior

class Aplicacion():
    def __init__(self):
        self.raiz = Tk()
        self.raiz.title("Alta Velocidad")

        # Declara variables de control

        self.num_via = IntVar(value=1)
        self.ida_vue = BooleanVar()
        self.clase   = StringVar(value='t')
        self.km = IntVar(value=1)
        self.precio = DoubleVar(value=0.10)
        self.total = DoubleVar(value=0.0)

        # Carga imagen para asociar a widget Label()

        tren = PhotoImage(file='tren-128x64.png')
```

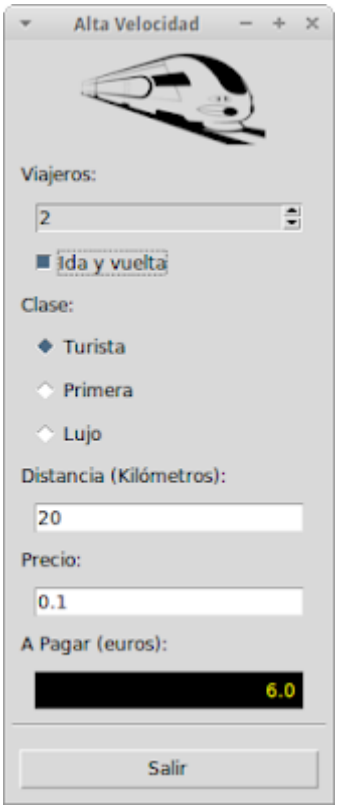
```
        # Declara widgets de la ventana
        # Se incluye el widget de tipo Button 'Calcular' que utiliza
        # la opción 'command' para validar datos y calcular el
        # importe a pagar cuando sea presionado

        self.imagen1 = ttk.Label(self.raiz, image=tren,
                                 anchor="center")
        self.etiq1 = ttk.Label(self.raiz, text="Viajeros:")
        self.viaje = Spinbox(self.raiz, from_=1, to=20, wrap=True,
                             textvariable=self.num_via,
                             state='readonly')
        self.idavue = ttk.Checkbutton(self.raiz, text='Ida y vuelta',
                                      variable=self.ida_vue,
                                      onvalue=True, offvalue=False)
        self.etiq2 = ttk.Label(self.raiz, text="Clase:")
        self.clase1 = ttk.Radiobutton(self.raiz, text='Turista',
                                      variable=self.clase, value='t')
        self.clase2 = ttk.Radiobutton(self.raiz, text='Primera',
                                      variable=self.clase, value='p')
        self.clase3 = ttk.Radiobutton(self.raiz, text='Lujo',
                                       variable=self.clase, value='l')
        self.etiq3 = ttk.Label(self.raiz,
                               text="Distancia Kilómetros):")
        self.dist = ttk.Entry(self.raiz, textvariable=self.km,
                              width=10)
        self.etiq4 = ttk.Label(self.raiz, text="Precio:")
        self.coste = ttk.Entry(self.raiz, textvariable=self.precio,
                               width=10)
        self.etiq5 = ttk.Label(self.raiz, text="A Pagar (euros):")
        self.etiq6 = ttk.Label(self.raiz, textvariable=self.total,
                               foreground="yellow", background="black",
                               borderwidth=5, anchor="e")
        self.separ1 = ttk.Separator(self.raiz, orient=HORIZONTAL)

        self.boton1 = ttk.Button(self.raiz, text="Calcular",
                                 command=self.calcular)
        self.boton2 = ttk.Button(self.raiz, text="Salir",
                                 command=quit)

        self.imagen1.pack(side=TOP, fill=BOTH, expand=True,
                          padx=10, pady=5)
        self.etiq1.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.viaje.pack(side=TOP, fill=X, expand=True,
                        padx=20, pady=5)
```

```
        self.idavue.pack(side=TOP, fill=X, expand=True,
                         padx=20, pady=5)
        self.etiq2.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.clase1.pack(side=TOP, fill=BOTH, expand=True,
                         padx=20, pady=5)
        self.clase2.pack(side=TOP, fill=BOTH, expand=True,
                         padx=20, pady=5)
        self.clase3.pack(side=TOP, fill=BOTH, expand=True,
                         padx=20, pady=5)
        self.etiq3.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.dist.pack(side=TOP, fill=X, expand=True,
                       padx=20, pady=5)
        self.etiq4.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.coste.pack(side=TOP, fill=X, expand=True,
                        padx=20, pady=5)
        self.etiq5.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.etiq6.pack(side=TOP, fill=BOTH, expand=True,
                        padx=20, pady=5)
        self.separ1.pack(side=TOP, fill=BOTH, expand=True,
                         padx=5, pady=5)
        self.boton1.pack(side=LEFT, fill=BOTH, expand=True,
                         padx=10, pady=10)
        self.boton2.pack(side=RIGHT, fill=BOTH, expand=True,
                         padx=10, pady=10)
        self.raiz.mainloop()

    def calcular(self):

        # Función para validar datos y calcular importe a pagar

        error_dato = False
        total = 0
        try:
            km = int(self.km.get())
            precio = float(self.precio.get())
        except:
            error_dato = True
        if not error_dato:
            total =  self.num_via.get() * km * precio
            if self.ida_vue.get():
                total = total * 1.5
            if self.clase.get() == 'p':
```

```
                total = total * 1.2
            elif self.clase.get() == '1':
                total = total * 2
            self.total.set(total)
        else:
            self.total.set("¡ERROR!")

def main():
    mi_app = Aplicacion()
    return 0

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

Validación y cálculo inmediato

En la segunda solución no se utiliza el botón **'Calcular'**. La validación y el cálculo se realizan al cambiar el valor de cualquier widget, mostrando el resultado inmediatamente. Para los widget de entrada de datos (**Entry()**) se definen dos trazas para detectar cualquier cambio en los datos y en el resto de widgets se utiliza la opción **'command'**.




```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from tkinter import *
```

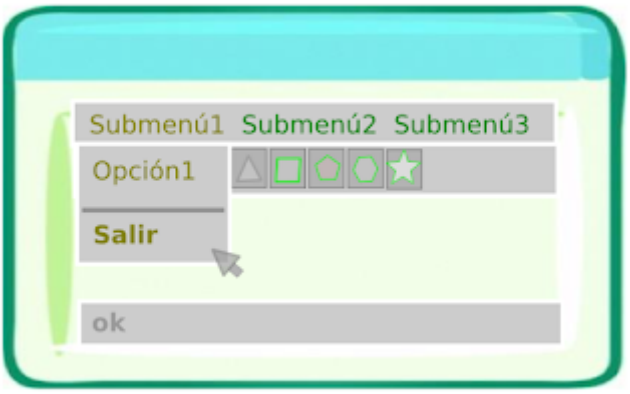
```
from tkinter import ttk

# Calcula coste de viaje con validación y cálculo inmediato

class Aplicacion():
    def __init__(self):
        self.raiz = Tk()
        self.raiz.title("Alta Velocidad")

        # Declara variables de control

        self.num_via = IntVar(value=1)
        self.ida_vue = BooleanVar()
        self.clase   = StringVar(value='t')
        self.km = IntVar(value=1)
        self.precio = DoubleVar(value=0.10)
        self.total = DoubleVar(value=0.0)

        # Define trazas con variables de control de los widgets Entry()
        # para detectar cambios en los datos. Si se producen cambios
        # se llama a la función 'self.calcular' para validación y para
        # calcular importe a pagar

        self.km.trace('w', self.calcular)
        self.precio.trace('w', self.calcular)

        # Llama a función para validar y calcular

        self.calcular()

        # Carga imagen para asociar a widget Label()

        tren = PhotoImage(file='tren-128x64.png')

        # Declara widgets de la ventana
        # En los widgets de tipo Spinbox, Checkbutton y Radiobutton
        # se utiliza la opción 'command' para llamar a la función
        # 'self.calcular' para validar datos y calcular importe a
        # pagar de forma inmediata

        self.imagen1 = ttk.Label(self.raiz, image=tren,
                                 anchor="center")
        self.etiq1 = ttk.Label(self.raiz, text="Viajeros:")
        self.viaje = Spinbox(self.raiz, from_=1, to=20, wrap=True,
                             textvariable=self.num_via,
                             state='readonly',
```

```
                        command=self.calcular)
        self.idavue = ttk.Checkbutton(self.raiz, text='Ida y vuelta',
                                      variable=self.ida_vue,
                                      onvalue=True, offvalue=False,
                                      command=self.calcular)
        self.etiq2 = ttk.Label(self.raiz, text="Clase:")
        self.clase1 = ttk.Radiobutton(self.raiz, text='Turista',
                                      variable=self.clase, value='t',
                                      command=self.calcular)
        self.clase2 = ttk.Radiobutton(self.raiz, text='Primera',
                                      variable=self.clase, value='p',
                                      command=self.calcular)
        self.clase3 = ttk.Radiobutton(self.raiz, text='Lujo',
                                      variable=self.clase, value='l',
                                      command=self.calcular)
        self.etiq3 = ttk.Label(self.raiz,
                               text="Distancia (Kilómetros):")
        self.dist = ttk.Entry(self.raiz, textvariable=self.km,
                              width=10)
        self.etiq4 = ttk.Label(self.raiz, text="Precio:")
        self.coste = ttk.Entry(self.raiz, textvariable=self.precio,
                               width=10)
        self.etiq5 = ttk.Label(self.raiz, text="A Pagar (euros):")
        self.etiq6 = ttk.Label(self.raiz, textvariable=self.total,
                               foreground="yellow", background="black",
                               borderwidth=5, anchor="e")
        self.separ1 = ttk.Separator(self.raiz, orient=HORIZONTAL)

        self.boton1 = ttk.Button(self.raiz, text="Salir",
                                 command=quit)

        self.imagen1.pack(side=TOP, fill=BOTH, expand=True,
                          padx=10, pady=5)
        self.etiq1.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.viaje.pack(side=TOP, fill=X, expand=True,
                        padx=20, pady=5)
        self.idavue.pack(side=TOP, fill=X, expand=True,
                         padx=20, pady=5)
        self.etiq2.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.clase1.pack(side=TOP, fill=BOTH, expand=True,
                         padx=20, pady=5)
        self.clase2.pack(side=TOP, fill=BOTH, expand=True,
                         padx=20, pady=5)
        self.clase3.pack(side=TOP, fill=BOTH, expand=True,
```

```
                        padx=20, pady=5)
        self.etiq3.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.dist.pack(side=TOP, fill=X, expand=True,
                       padx=20, pady=5)
        self.etiq4.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.coste.pack(side=TOP, fill=X, expand=True,
                        padx=20, pady=5)
        self.etiq5.pack(side=TOP, fill=BOTH, expand=True,
                        padx=10, pady=5)
        self.etiq6.pack(side=TOP, fill=BOTH, expand=True,
                        padx=20, pady=5)
        self.separ1.pack(side=TOP, fill=BOTH, expand=True,
                         padx=5, pady=5)
        self.boton1.pack(side=RIGHT, fill=BOTH, expand=True,
                         padx=10, pady=10)
        self.raiz.mainloop()

    def calcular(self, *args):

        # Función para validar datos y calcular importe a pagar

        error_dato = False
        total = 0
        try:
            km = int(self.km.get())
            precio = float(self.precio.get())
        except:
            error_dato = True
        if not error_dato:
            total =  self.num_via.get() * km * precio
            if self.ida_vue.get():
                total = total * 1.5
            if self.clase.get() == 'p':
                total = total * 1.2
            elif self.clase.get() == 'l':
                total = total * 2
            self.total.set(total)
        else:
            self.total.set("¡ERROR!")

def main():
    mi_app = Aplicacion()
    return 0
```

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

Menús, barras de herramientas y de estado en Tkinter



Introducción

Tkinter cuenta con widgets específicos para incluir **menús de opciones** de distintos tipos en una aplicación. Además, siguiendo unas pautas en la construcción de ventanas es posible agregar otros elementos como **barras de herramientas y de estado**.

Todos estos componentes se utilizan con frecuencia en el desarrollo de interfaces porque facilitan mucho la interacción de los usuarios con las aplicaciones gráficas.

Menús de opciones

Los menús pueden construirse agrupando en una barra de menú varios submenús desplegables, mediante menús basados en botones, o bien, utilizando los típicos menús contextuales que cambian sus opciones disponibles dependiendo del lugar donde se activan en la ventana de la aplicación.

Entre los tipos de opciones que se pueden incluir en un menú se encuentran aquellas que su estado representan un valor lógico de activada o desactivada (**add_checkbutton**); las que permiten elegir una opción de entre varias existentes (**add_radiobutton**) y las que ejecutan directamente un método o una función (**add_command**).

Las opciones de un menú pueden incluir iconos y asociarse a atajos o combinaciones de teclas que surten el mismo efecto que si éstas son seleccionadas con un clic de ratón. También, en un momento dado, pueden deshabilitarse para impedir que puedan ser seleccionadas.

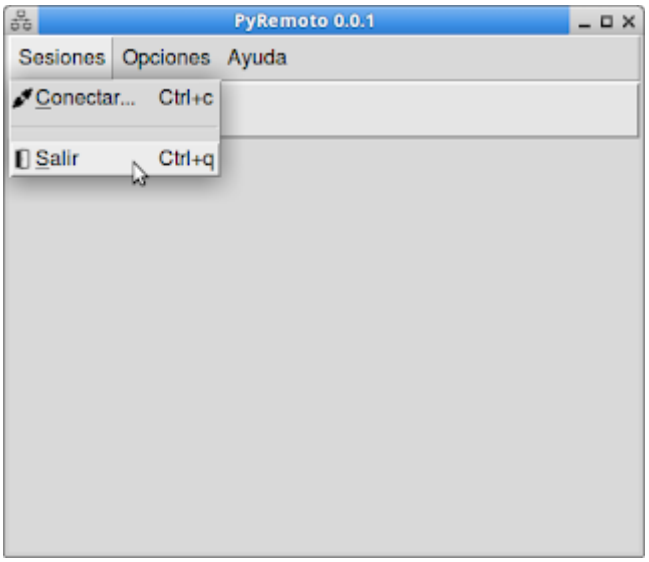
Barras de herramientas

Una aplicación Tkinter además de menús de opciones puede incorporar barras de herramientas construidas a base de botones apilados de manera horizontal o vertical. Estas barras de herramientas suelen situarse, muchas veces, justo debajo de la barra de menú de la aplicación y ejecutan los procesos más utilizados en el sistema.

Barra de estado

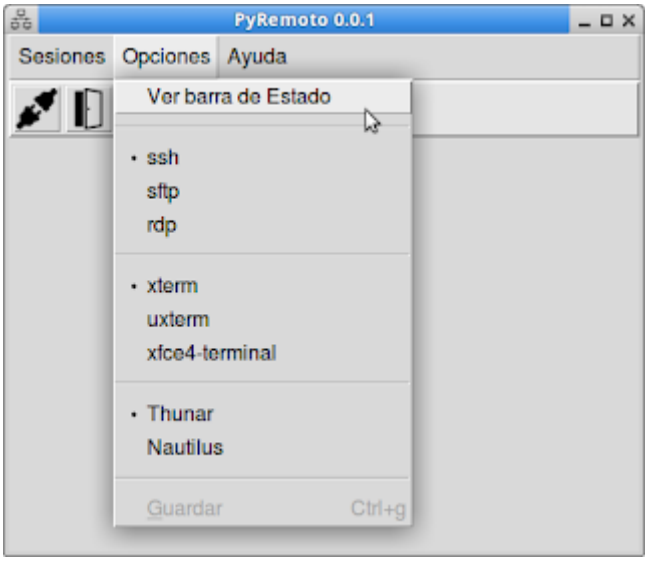
Las barras de estado se utilizan con asiduidad en los programas gráficos para mostrar información que resulte útil a los usuarios. Normalmente, ocupan la última línea de la ventana de aplicación y en algunos casos puede ocultarse para dejar más espacio disponible a dicha ventana.

PyRemoto, un ejemplo de implementación

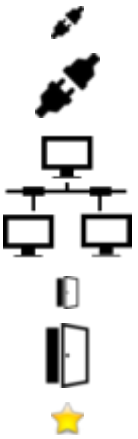


PyRemoto es un ejemplo que pretende mostrar cómo incluir menús y barras en una aplicación basada en Tkinter.

La ventana principal de esta aplicación cuenta con una barra de título, una barra de menú con tres submenús desplegables con distintos tipos de opciones, un menú contextual que se activa haciendo clic con el botón secundario del ratón en cualquier lugar de la ventana, una barra de herramientas con dos botones con imágenes y una barra de estado que es posible ocultar o mostrar mediante la opción "**Ver barra de estado**" del submenú "**Opciones**".

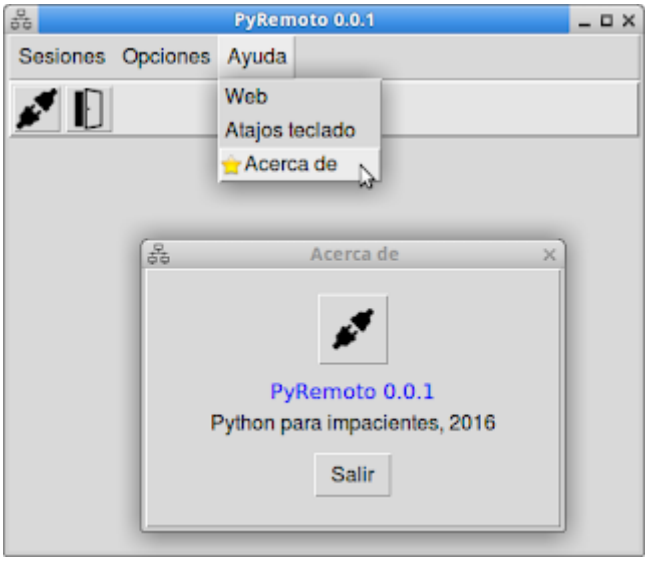


En el submenú "**Opciones**" se encuentra la opción "**Guardar**" que se habilitará cuando se elija alguna opción de entre las disponibles en este submenú; y se deshabilitará cuando se utilice la propia opción "**Guardar**", -prevista- para salvar las preferencias seleccionadas por el usuario.



La aplicación de ejemplo utiliza imágenes de diferentes tamaños que se localizan en una carpeta llamada "**imagen**". Estas imágenes son utilizadas en la barra de título, los submenús, la barra de herramientas y en la ventana de diálogo "**Acerca de**" del submenú

"**Ayuda**". Han sido obtenidas del sitio flaticon.com que ofrece colecciones de imágenes organizadas por distintas temáticas a programadores y a diseñadores gráficos.



También, en el programa se incluyen varias combinaciones de teclas asociadas a distintas opciones del menú, declaradas con el método **bind()**.

¿Cómo funciona?
La aplicación la inicia la función **main()** que comprueba que todas las imágenes de la aplicación existen en la carpeta "**imagen**". Si todas las imágenes están disponibles se creará el objeto aplicación mediante la clase **PyRemoto()**. Esta clase cuenta con un método **__init__()** que construye la ventana de la aplicación, los menús, la barra de herramientas y una barra de estado.

La aplicación está incompleta pero enseña cómo organizar los métodos que son llamados desde las distintas opciones de los menús y barras.

Por último, el ejemplo incluye la ventana de diálogo "**Acerca de**", que es de tipo modal, para mostrar el modo de abrir este tipo de ventanas desde una opción del menú.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# name:        pyremoto.py (Python 3.x).
# description: Acceso Remoto a equipos por ssh, sftp y rdp
# purpose:     Construcción de menús, barras de herramientas
#              y de estado
# author:      python para impacientes
#
#------------------------------------------------------------

'''PyRemoto: Acceso Remoto a equipos por ssh, sftp y rdp '''
```

```
__author__ = 'python para impacientes'
__title__= 'PyRemoto'
__date__ = ''
__version__ = '0.0.1'
__license__ = 'GNU GPLv3'

import os, sys, webbrowser, platform
from tkinter import *
from tkinter import ttk, font, messagebox

# PYREMOTO: ACCESO REMOTO A EQUIPOS POR SSH, SFTP y RDP

class PyRemoto():
    ''' Clase PyRemoto '''

    # DECLARAR MÉTODO CONSTRUCTOR DE LA APLICACIÓN

    def __init__(self, img_carpeta, iconos):
        ''' Definir ventana de la aplicación, menú, submenús,
            menú contextual, barra de herramientas, barra de
            estado y atajos del teclado '''

        # INICIALIZAR VARIABLES

        self.img_carpeta = img_carpeta
        self.iconos = iconos

        # DEFINIR VENTANA DE LA APLICACIÓN:

        self.raiz = Tk()

        # ESTABLECER PROPIEDADES DE LA VENTANA DE APLICACIÓN:

        self.raiz.title("PyRemoto " + __version__)  # Título
        self.icono1= PhotoImage(file=self.iconos[0])  # Icono app
        self.raiz.iconphoto(self.raiz, self.icono1)  # Asigna icono app
        self.raiz.option_add("*Font", "Helvetica 12")  # Fuente
predeterminada
        self.raiz.option_add('*tearOff', False)  # Deshabilita submenús
flotantes
        self.raiz.attributes('-fullscreen', True)  # Maximiza ventana
completa
        self.raiz.minsize(400,300)  # Establece tamaño minimo ventana

        # ESTABLECER ESTILO FUENTE PARA ALGUNOS WIDGETS:
```

```
        self.fuente = font.Font(weight='normal') # normal, bold, etc...

        # DECLARAR VARIABLES PARA OPCIONES PREDETERMINADAS:
        # (Estos valores se podrían leer de un archivo de
        # configuración)

        self.CFG_TIPOCONEX = IntVar()
        self.CFG_TIPOCONEX.set(1) # shh
        self.CFG_TIPOEMUT = IntVar()
        self.CFG_TIPOEMUT.set(1) # xterm
        self.CFG_TIPOEXP = IntVar()
        self.CFG_TIPOEXP.set(1) # thunar

        # DECLARAR VARIABLE PARA MOSTRAR BARRA DE ESTADO:

        self.estado = IntVar()
        self.estado.set(1)  # Mostrar Barra de Estado

        # DEFINIR BARRA DE MENÚ DE LA APLICACION:

        barramenu = Menu(self.raiz)
        self.raiz['menu'] = barramenu

        # DEFINIR SUBMENÚS 'Sesiones', 'Opciones' y 'Ayuda':

        menu1 = Menu(barramenu)
        self.menu2 = Menu(barramenu)
        menu3 = Menu(barramenu)
        barramenu.add_cascade(menu=menu1, label='Sesiones')
        barramenu.add_cascade(menu=self.menu2, label='Opciones')
        barramenu.add_cascade(menu=menu3, label='Ayuda')

        # DEFINIR SUBMENÚ 'Sesiones':

        icono2 = PhotoImage(file=self.iconos[1])
        icono3 = PhotoImage(file=self.iconos[2])

        menu1.add_command(label='Conectar...',
                          command=self.f_conectar,
                          underline=0, accelerator="Ctrl+c",
                          image=icono2, compound=LEFT)
        menu1.add_separator()  # Agrega un separador
        menu1.add_command(label='Salir', command=self.f_salir,
                          underline=0, accelerator="Ctrl+q",
                          image=icono3, compound=LEFT)
```

```
        # DEFINIR SUBMENÚ 'Opciones':

        self.menu2.add_checkbutton(label='Barra de Estado',
                           variable=self.estado, onvalue=1,
                           offvalue=0,
                           command=self.f_verestado)
        self.menu2.add_separator()
        self.menu2.add_radiobutton(label='ssh',
                           variable=self.CFG_TIPOCONEX,
                           command=self.f_cambiaropc,
                           value=1)
        self.menu2.add_radiobutton(label='sftp',
                           variable=self.CFG_TIPOCONEX,
                           command=self.f_cambiaropc,
                           value=2)
        self.menu2.add_radiobutton(label='rdp',
                           variable=self.CFG_TIPOCONEX,
                           command=self.f_cambiaropc,
                           value=3)
        self.menu2.add_separator()
        self.menu2.add_radiobutton(label='xterm',
                           variable=self.CFG_TIPOEMUT,
                           command=self.f_cambiaropc,
                           value=1)
        self.menu2.add_radiobutton(label='uxterm',
                           variable=self.CFG_TIPOEMUT,
                           command=self.f_cambiaropc,
                           value=2)
        self.menu2.add_radiobutton(label='xfce4-terminal',
                           variable=self.CFG_TIPOEMUT,
                           command=self.f_cambiaropc,
                           value=3)
        self.menu2.add_separator()
        self.menu2.add_radiobutton(label='Thunar',
                           variable=self.CFG_TIPOEXP,
                           command=self.f_cambiaropc,
                           value=1)
        self.menu2.add_radiobutton(label='Nautilus',
                           variable=self.CFG_TIPOEXP,
                           command=self.f_cambiaropc,
                           value=2)
        self.menu2.add_separator()
        self.menu2.add_command(label='Guardar',
                           command=self.f_opcionguardar,
                           state="disabled", underline=0,
                           accelerator="Ctrl+g")
```

```
        # DEFINIR SUBMENÚ 'Ayuda':

        menu3.add_command(label='Web', command=self.f_web)
        menu3.add_command(label='Atajos teclado',
                          command=self.f_atajos)
        icono4 = PhotoImage(file=self.iconos[3])
        menu3.add_command(label="Acerca de",
                          command=self.f_acerca,
                          image=icono4, compound=LEFT)

        # DEFINIR BARRA DE HERRAMIENTAS:

        self.icono5 = PhotoImage(file=self.iconos[4])
        icono6 = PhotoImage(file=self.iconos[5])

        barraherr = Frame(self.raiz, relief=RAISED,
                          bd=2, bg="#E5E5E5")
        bot1 = Button(barraherr, image=self.icono5,
                      command=self.f_conectar)
        bot1.pack(side=LEFT, padx=1, pady=1)
        bot2 = Button(barraherr, image=icono6,
                      command=self.f_salir)
        bot2.pack(side=LEFT, padx=1, pady=1)
        barraherr.pack(side=TOP, fill=X)

        # DEFINIR BARRA DE ESTADO:
        # Muestra información del equipo

        info1 = platform.system()
        info2 = platform.node()
        info3 = platform.machine()

        # Otro modo de obtener la información:
        # (No disponible en algunas versiones de Windows)

        #info1 = os.uname().sysname
        #info2 = os.uname().nodename
        #info3 = os.uname().machine

        mensaje = " " + info1+ ": "+info2+" - "+info3
        self.barraest = Label(self.raiz, text=mensaje,
                              bd=1, relief=SUNKEN, anchor=W)
        self.barraest.pack(side=BOTTOM, fill=X)

        # DEFINIR MENU CONTEXTUAL
```

```
        self.menucontext = Menu(self.raiz, tearoff=0)
        self.menucontext.add_command(label="Conectar",
                                     command=self.f_conectar)
        self.menucontext.add_command(label="Salir",
                                     command=self.f_salir)

        # DECLARAR TECLAS DE ACCESO RAPIDO:

        self.raiz.bind("<Control-c>",
                       lambda event: self.f_conectar())
        self.raiz.bind("<Control-g>",
                       lambda event: self.f_guardar())
        self.raiz.bind("<Control-q>",
                       lambda event: self.f_salir())
        self.raiz.bind("<Button-3>",
                       self.f_mostrarmenucontext)
        self.raiz.mainloop()

    # DECLARAR OTROS MÉTODOS DE LA APLICACIÓN:

    def f_opcionguardar(self):
        ''' Si opción 'Guardar' está habilitada llama a
            método para guardar opciones de configuración
            de la aplicación '''

        if self.menu2.entrycget(13,"state")=="normal":
            self.menu2.entryconfig(13, state="disabled")
            self.f_guardarconfig()

    def f_guardarconfig(self):
        ''' Guardar opciones de configuración de la aplicación '''
        print("Configuración guardada")

    def f_conectar(self):
        ''' Definir ventana de diálogo para conectar con equipos '''
        print("Conectando")

    def f_cambiaropc(self):
        ''' Habilitar opción 'Guardar' al elegir alguna opción
            de tipo de conexión, emulador de terminar o
            explorador de archivos '''
        self.menu2.entryconfig("Guardar", state="normal")

    def f_verestado(self):
        ''' Ocultar o Mostrar barra de estado '''
```

```
        if self.estado.get() == 0:
            self.barraest.pack_forget()
        else:
            self.barraest.pack(side=BOTTOM, fill=X)

    def f_mostrarmenucontext(self, e):
        ''' Mostrar menú contextual '''
        self.menucontext.post(e.x_root, e.y_root)

    def f_web(self):
        ''' Abrir página web en navegador Internet '''

        pag1 = 'http://python-para-impacientes.blogspot.com/'
        webbrowser.open_new_tab(pag1)

    def f_atajos(self):
        ''' Definir ventana de diálogo con lista de
            combinaciones de teclas de la aplicación '''
        pass

    def f_acerca(self):
        ''' Definir ventana de diálogo 'Acerca de' '''

        acerca = Toplevel()
        acerca.geometry("320x200")
        acerca.resizable(width=False, height=False)
        acerca.title("Acerca de")
        marco1 = ttk.Frame(acerca, padding=(10, 10, 10, 10),
                           relief=RAISED)
        marco1.pack(side=TOP, fill=BOTH, expand=True)
        etiq1 = Label(marco1, image=self.icono5,
                      relief='raised')
        etiq1.pack(side=TOP, padx=10, pady=10,
                   ipadx=10, ipady=10)
        etiq2 = Label(marco1, text="PyRemoto "+__version__,
                      foreground='blue', font=self.fuente)
        etiq2.pack(side=TOP, padx=10)
        etiq3 = Label(marco1,
                      text="Python para impacientes")
        etiq3.pack(side=TOP, padx=10)
        boton1 = Button(marco1, text="Salir",
                        command=acerca.destroy)
        boton1.pack(side=TOP, padx=10, pady=10)
        boton1.focus_set()
        acerca.transient(self.raiz)
```

```
        self.raiz.wait_window(acerca)

    def f_salir(self):
        ''' Salir de la aplicación '''
        self.raiz.destroy()

# FUNCIONES DE LA APLICACIÓN

def f_verificar_iconos(iconos):
    ''' Verifica existencia de iconos

    iconos -- Lista de iconos '''

    for icono in iconos:
        if not os.path.exists(icono):
            print('Icono no encontrado:', icono)
            return(1)
    return(0)

def main():
    ''' Iniciar aplicación '''

    # INICIALIZAR VARIABLES CON RUTAS

    app_carpeta = os.getcwd()
    img_carpeta = app_carpeta + os.sep + "imagen" + os.sep

    # DECLARAR Y VERIFICAR ICONOS DE LA APLICACIÓN:

    iconos = (img_carpeta + "pyremoto64x64.png",
              img_carpeta + "conec16x16.png",
              img_carpeta + "salir16x16.png",
              img_carpeta + "star16x16.png",
              img_carpeta + "conec32x32.png",
              img_carpeta + "salir32x32.png")
    error1 = f_verificar_iconos(iconos)

    if not error1:
        mi_app = PyRemoto(img_carpeta, iconos)
    return(0)

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