

Unidad 4

Programación Orientada a Objetos

Asignatura: Programación Orientada a Objetos

Licenciatura en Ciencias de la Computación

Licenciatura en Sistemas de Información

Tecnicatura en Programación Web

Departamento de Informática

FCEFN – UNSJ

Año 2024

Propósitos que persigue la unidad

- ✓ Que el estudiante distinga aspectos relevantes inherentes a la implementación de lenguajes orientados a objetos en general y a Python en particular.
- ✓ Que el estudiante adquiera aspectos relevantes de la programación orientada a eventos.
- ✓ Que el estudiante desarrolle aplicaciones bajo el paradigma de la programación orientada a eventos, haciendo uso de los conceptos aprendidos de la programación orientada a objetos.

Bibliografía

Kenneth C. Louden - Lenguajes de Programación Principios y Práctica.
Editorial Thomson (2004) – capítulo 6.

Mark Summerfield - Programming in Python 3-Addison Wesley (2009)

Steven F. Lott – Mastering Object-Oriented Python – Second Edition – Pact
Publishing (2019)

Dusty Phillips - Python 3 Object-oriented Programming, 2nd Edition-Packt
Publishing (2015)

Alejandro Rodas de Paz - Tkinter GUI application development cookbook-
Packt Publishing (2018)

Alan D. Moore - Python GUI programming with Tkinter-Packt Publishing
(2018)

Bhaskar Chaudhary - Tkinter GUI application development blueprints-Packt
Publishing (2018)

Modelo de Ejecución o Computadora Virtual

- ✓ Cuando se implementa un lenguaje de programación, las estructuras de datos y algoritmos que se utilizan en tiempo de ejecución de un programa escrito en ese lenguaje, definen un **modelo de ejecución o computadora virtual** definida por la implementación del lenguaje.
- ✓ El lenguaje de máquina de esta computadora virtual es el programa ejecutable que produce el traductor del lenguaje.
- ✓ Las estructuras de datos de esta computadora virtual son las estructuras de datos que en tiempo de ejecución se utilizan para la ejecución del programa.
- ✓ Las operaciones primitivas son aquellas efectivamente ejecutables en tiempo de ejecución.
- ✓ Las estructuras de control de secuencia, control de datos y gestión de almacenamiento son las que se emplean en tiempo de ejecución, independientemente de la representación de software, hardware o microprograma.

Implementación de un lenguaje de programación y Computadora Virtual

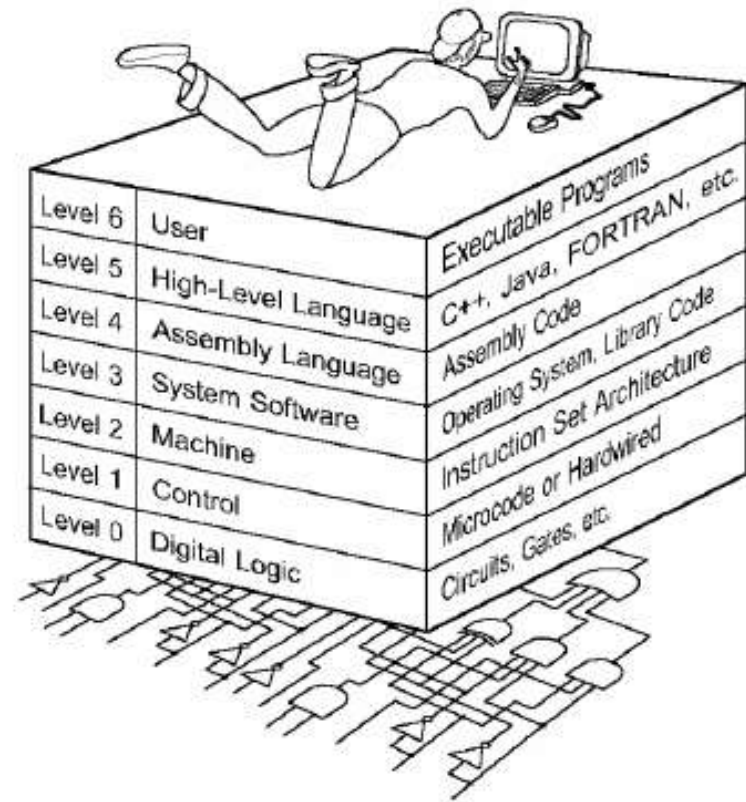
- ✓ Al momento de la implementación de un lenguaje, el implementador construye la computadora virtual a partir de los elementos de hardware y software que provee la computadora subyacente.
- ✓ La implementación de un lenguaje está determinada por múltiples decisiones que debe tomar el implementador en función de los recursos de hardware y software disponibles en la computadora subyacente y los costos de su uso.

Jerarquías de computadoras

- ✓ La computadora virtual que un programador utiliza cuando decide hacer un programa en algún lenguaje de alto nivel está formada, de hecho, por una **jerarquía de computadoras virtuales**.
- ✓ El último nivel es una **computadora de hardware real**. Sin embargo, el programador rara vez tiene trato directo con esta computadora.
- ✓ Esta computadora de hardware se transforma sucesivamente a través de capas de software (o microprogramas) en una computadora virtual que puede ser radicalmente distinta.
- ✓ El segundo nivel de computadora virtual (o tercero si un microprograma forma el segundo nivel) está definido por la compleja colección de rutinas que se conoce como **sistema operativo**.

Jerarquía de Computadoras – Modelo General

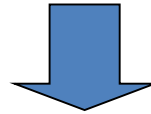
- Cada capa es una máquina virtual que abstrae a las máquinas del nivel inferior.
- Las máquinas, en su nivel, «interpretan» sus instrucciones particulares, utilizando servicios de su capa inferior para implementarlas.
- En última instancia los circuitos terminan haciendo el trabajo.



Computadora Virtual y Objeto

Para la mayoría de las personas, la forma de pensar orientada a objetos es más natural que las técnicas tradicionales. Al fin y al cabo, el mundo está formado por objetos

El paradigma orientado a objetos se basa en la idea de **un objeto que vagamente puede describirse como una colección de localizaciones de memoria, junto con todas las operaciones** que pueden cambiar los valores de dichas localizaciones de memoria.



En el fondo existe una conexión con la computadora, cada objeto se parece un poco a una pequeña **computadora virtual**, éste tiene un **estado** y tiene operaciones (**métodos**) que uno puede ordenarle ejecutar. Esto no parece ser una mala analogía de los objetos del mundo real; todos ellos tienen **características y comportamientos**.

Cuestiones de Diseño e Implementación (1)

Los Lenguajes Orientados a Objetos (LOO) encaran tres problemas de Diseño de Software

- Necesidad de volver a utilizar componentes de Software
- Necesidad de modificar comportamiento (cambios mínimos)
- Necesidad de conservar independencia de diferentes componentes

Cuestiones de Diseño e Implementación (2)

Existen distintas maneras para modificar un componente de software de modo que pueda reutilizarse

- Extensión de los datos y/o de las operaciones
- Restricciones de los datos y/o de las operaciones
- Redefinición de una o más de las operaciones
- Abstracción o la reunión de operaciones similares de dos componentes diferentes en uno nuevo.
- **Polimorfización** o extensión del tipo de datos a los cuales pueden aplicarse las operaciones

Herencia y Polimorfismo

Existen tres tipos básicos de polimorfismo:

- ❖ Polimorfismo paramétrico o **genericidad**, donde los parámetros de tipo pueden quedarse sin especificar (Generics en C# y Template en C++).

Permite generar métodos y funciones que responden de manera similar a argumentos de diferentes tipos (ej. Listas de números, Listas de autos, etc.)

- ❖ La sobrecarga (polimorfismo ad-hoc) donde diferentes funciones o declaraciones de método comparten el mismo nombre, eliminando la ambigüedad mediante el uso de distintos tipos de los parámetros en cada declaración, o modificando la cantidad de parámetros.

- ❖ Polimorfismo de subtipo, donde todas las operaciones de un tipo pueden aplicarse a otro tipo.

En la programación orientada a objetos se refiere a la capacidad de un objeto de clase o tipo A aparecer y ser utilizado como si fuera de otra clase B.

Programación Orientada a Eventos

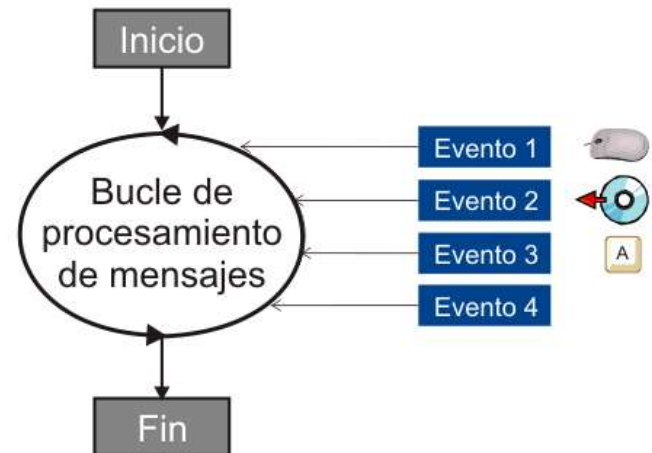
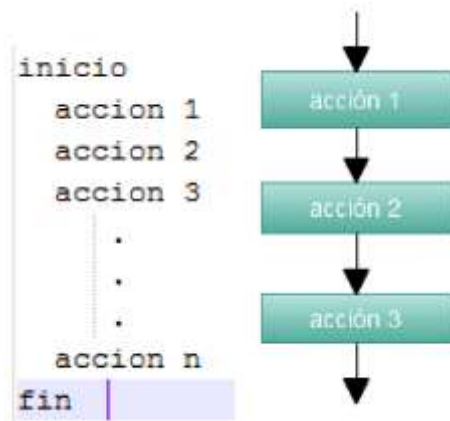
La Programación orientada a eventos es un paradigma. Término común en el vocabulario científico y en expresiones epistemológicas cuando se hacía necesario hablar de modelos o patrones, de programación en el que la estructura y la ejecución de los programas van determinados por los sucesos o acciones que ocurren en el sistema, definidos por el usuario o por el propio sistema.

Con los lenguajes orientados a eventos se pueden realizar en poco tiempo aplicaciones sencillas y muy funcionales, utilizando **interfaces gráficas** en las que se insertan componentes o controles a los que se le programan eventos. Los eventos permiten al usuario realizar una serie de acciones lógicas para un determinado programa.

Secuencia, interacción, eventos

Existen distintos tipos de programas:

- Secuenciales, también denominados batch, el programa inicia su ejecución, abre un archivo, lee los datos, los procesa, y entrega un resultado, no hay intervención del usuario final.
- Interactivos, en este tipo de programas, el usuario provee datos necesarios para la ejecución del programa, el usuario decide qué parte del programa se ejecuta a través de un menú de opciones, por ejemplo.
- Basados en eventos, el programa se presenta a través de una interface gráfica, y se queda esperando a que el usuario o el sistema, produzca un evento, puede ser un clic del mouse, una pulsación de una tecla, o un tick de reloj, que hace que el programa ejecute una parte del mismo, este tipo de programas pasa la mayor parte del tiempo esperando eventos.



Evento

Los Eventos son las acciones sobre el programa, como por ejemplo:

- Clic sobre un botón
- Doble clic sobre una imagen para expandirla
- Arrastrar un icono
- Pulsar una tecla o una combinación de teclas
- Elegir una opción de un menú
- Escribir en una caja de texto
- Mover el mouse
- Seleccionar un elemento de una caja desplegable
- Seleccionar un día de un calendario
- Tick de un reloj, por ejemplo de un temporizador

Un evento permite al usuario realizar una serie de acciones lógicas, que fueron programadas por el programador en respuesta a dicho evento.

Propiedades

Una propiedad define el tamaño de un objeto en pantalla, el título de una ventana, la etiqueta de un botón, el evento al que responde el objeto, etc.

Los objetos que se ubican en una interface gráfica pueden ser:

- Una ventana
- Un botón de comando
- Una caja de texto
- Una etiqueta
- Una imagen

Las propiedades pueden asignarse al momento de diseñar la interfaz gráfica, y pueden cambiar en tiempo de ejecución.



Interfaz gráfica de Usuario (GUI) Usando TKinter

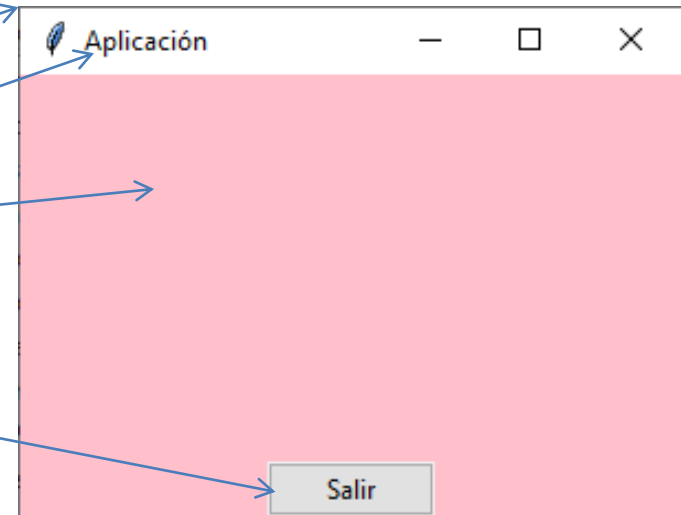
Tkinter

Con Python hay varias 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.

```
from tkinter import *
from tkinter import ttk
class Aplicacion():
    def __init__(self):
        ventana= Tk()
        ventana.geometry('300x200')
        ventana.configure(bg = 'beige')
        ventana.title('Aplicación')
        ttk.Button(ventana, text='Salir',
                    command=ventana.destroy).pack(side=BOTTOM)
        ventana.mainloop()
def testAPP():
    mi_app = Aplicacion()
if __name__ == '__main__':
    testAPP()
```



Las tres componentes en el desarrollo de GUI



Widgets - Tkinter

Un widget en TKinter Python, es un objeto predefinido, que puede ubicarse en el interior de una ventana, tiene un conjunto de propiedades que pueden modificar su aspecto, ubicación, eventos a los que responde, etc.

Widget Class	Descripción
Label	Un widget que se usa para mostrar texto en la pantalla.
Button	Un botón, contiene un texto, que describe la función que realiza cuando es presionado.
Entry	Un widget de entrada, permite ingresar una línea de texto.
Text	El widget text permite el ingreso de múltiples líneas de texto.
Frame	Una región rectangular para agrupar varios widgets relacionados o proveer relleno entre widgets.
ListBox	Una caja con múltiples opciones, se elige una de ellas.
LabelFrame	Una región rectangular que permite agrupar varios widget relacionecionados, que posee un título en la forma de una etiqueta.

Ejemplo: Convertidor Pulgadas a Centímetros

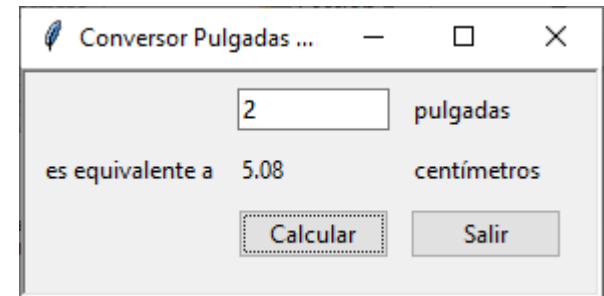
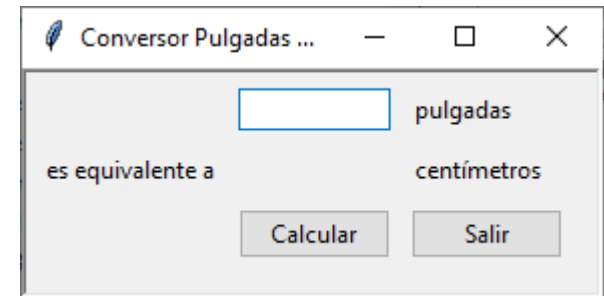
```
from tkinter import *
from tkinter import ttk, messagebox

class Aplicacion():
    __ventana: object
    __pulgadas: object
    __centimetros: object

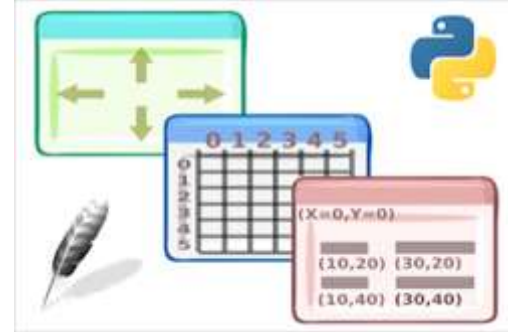
    def __init__(self):
        self.__ventana = Tk()
        self.__ventana.geometry('290x115')
        self.__ventana.title('Convertor Pulgadas a Centímetros')
        mainframe = ttk.Frame(self.__ventana, padding="3 3 12 12")
        mainframe.grid(column=0, row=0, sticky=(N, W, E, S))
        mainframe.columnconfigure(0, weight=1)
        mainframe.rowconfigure(0, weight=1)
        mainframe['borderwidth'] = 2
        mainframe['relief'] = 'sunken'
        self.__pulgadas = StringVar()
        self.__centimetros = StringVar()
        self.pulgadasEntry = ttk.Entry(mainframe, width=7, textvariable=self.__pulgadas)
        self.pulgadasEntry.grid(column=2, row=1, sticky=(W, E))
        ttk.Label(mainframe, textvariable=self.__centimetros).grid(column=2, row=2, sticky=(W, E))
        ttk.Button(mainframe, text="Calcular", command=self.calcular).grid(column=2, row=3, sticky=W)
        ttk.Button(mainframe, text="Salir", command=self.__ventana.destroy).grid(column=3, row=3,
        sticky=W)
        ttk.Label(mainframe, text="pulgadas").grid(column=3, row=1, sticky=W)
        ttk.Label(mainframe, text="es equivalente a").grid(column=1, row=2, sticky=E)
        ttk.Label(mainframe, text="centímetros").grid(column=3, row=2, sticky=W)
        for child in mainframe.winfo_children():
            child.grid_configure(padx=5, pady=5)
        self.pulgadasEntry.focus()
        self.__ventana.mainloop()
```

```
def calcular(self):
    try:
        valor=float(self.pulgadasEntry.get())
        self.__centimetros.set(2.54*valor)
    except ValueError:
        messagebox.showerror(title='Error de tipo',
                              message='Debe ingresar un valor numérico')
        self.__pulgadas.set("")
        self.pulgadasEntry.focus()

def testAPP():
    mi_app = Aplicacion()
    if __name__ == '__main__':
        testAPP()
```



Gestores de geometría



Los gestores de geometría definen cómo se ubican los widgets en la ventana de la aplicación, existen tres gestores: **pack**, **grid** y **place**.

Cada ventana puede tener un gestor distinto, el programador decide cual usar.

A la hora de diseñar las ventanas, se pueden utilizar unos widgets especiales (marcos, paneles, etc.) que actúan como contenedores de otros widgets.

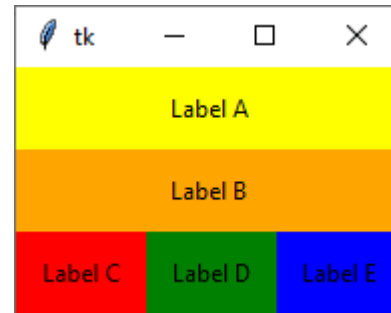
Estos widgets contenedores, se utilizan para agrupar varios controles con el objeto de facilitar la operación a los usuarios. En las ventanas que se utilicen podrá emplearse un gestor con la ventana y otro diferente para organizar los controles dentro de estos widgets.

Geometría Pack (I)

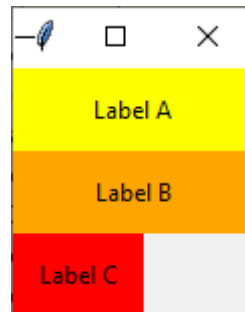
Con este gestor la organización de los widgets se hace teniendo en cuenta los lados de una ventana: arriba (TOP), abajo (BOTTOM), derecha (RIGHT) e izquierda (LEFT).

Con este gestor de geometría, es posible hacer que los controles se ajusten a los cambios de tamaño de la ventana.

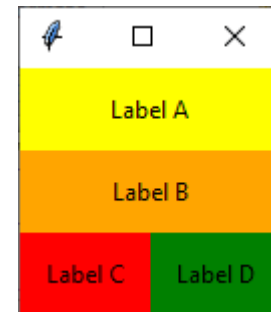
```
import tkinter as tk
class Aplicacion(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title='Pack Geometry'
        label_a = tk.Label(self, text="Label A", bg="yellow")
        label_b = tk.Label(self, text="Label B", bg="orange")
        label_c = tk.Label(self, text="Label C", bg="red")
        label_d = tk.Label(self, text="Label D", bg="green")
        label_e = tk.Label(self, text="Label E", bg="blue")
        opts = { 'ipadx': 10, 'ipady': 10, 'fill': tk.BOTH }
        label_a.pack(side=tk.TOP, **opts)
        label_b.pack(side=tk.TOP, **opts)
        label_c.pack(side=tk.LEFT, **opts)
        label_d.pack(side=tk.LEFT, **opts)
        label_e.pack(side=tk.LEFT, **opts)
if __name__=='__main__':
    app = Aplicacion()
    app.mainloop()
```



```
label_a.pack(side=tk.TOP, **opts)
label_b.pack(side=tk.TOP, **opts)
label_c.pack(side=tk.LEFT, **opts)
```



```
label_a.pack(side=tk.TOP, **opts)
label_b.pack(side=tk.TOP, **opts)
label_c.pack(side=tk.LEFT, **opts)
label_d.pack(side=tk.LEFT, **opts)
```

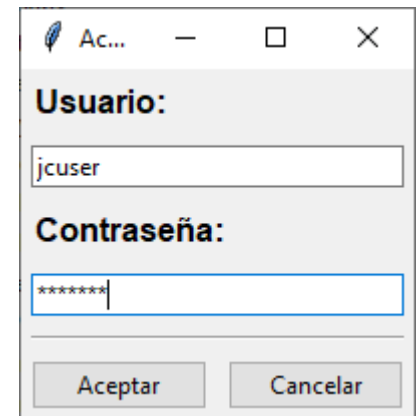


Geometría Pack (II)

```
from tkinter import *
from tkinter import ttk, font
class Aplicacion():
    __ventana: object
    __clave: object
    __usuario: object
    def __init__(self):
        self.__ventana = Tk()
        self.__ventana.title("Acceso")
        fuente = font.Font(weight='bold')
        self.usuarioLbl = ttk.Label(self.__ventana, text="Usuario:",
                                    font=fuente)
        self.contraseniaLbl = ttk.Label(self.__ventana, text="Contraseña:",
                                         font=fuente)
        self.__usuario = StringVar()
        self.__clave = StringVar()
        self.__usuario.set("")
        self.ctext1 = ttk.Entry(self.__ventana,
                                textvariable=self.__usuario,
                                width=30)
        self.ctext2 = ttk.Entry(self.__ventana,
                                textvariable=self.__clave,
                                width=30, show="*")
        self.separ1 = ttk.Separator(self.__ventana, orient=HORIZONTAL)
        self.boton1 = ttk.Button(self.__ventana, text="Aceptar",
                                 command=self.aceptar)
        self.boton2 = ttk.Button(self.__ventana, text="Cancelar",
                                 command=quit)
        self.usuarioLbl.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.contraseniaLbl.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)
        self.ctext2.focus_set()
        self.__ventana.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 testAPP():
    mi_app = Aplicacion()
    return 0
if __name__ == '__main__':
    testAPP()
```



Geometría Pack (III)

Se definen las posiciones de los widgets dentro de la ventana. Todos los controles se van colocando hacia el lado de arriba (TOP), excepto, los dos últimos, los botones, que se situarán de la siguiente forma: el primer botón hacia el lado de la izquierda (LEFT) y el segundo a su derecha (RIGHT).

'side': los valores posibles para la propiedad son: TOP (arriba), BOTTOM (abajo), LEFT (izquierda) y RIGHT (derecha). Si se omite, el valor será TOP.

'fill': la propiedad '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 propiedad 'expand' es True.

'padx', 'pady': las propiedades '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: 'ipadx' y 'ipady':

```
self.usuarioLbl.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.contraseniaLbl.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)
```

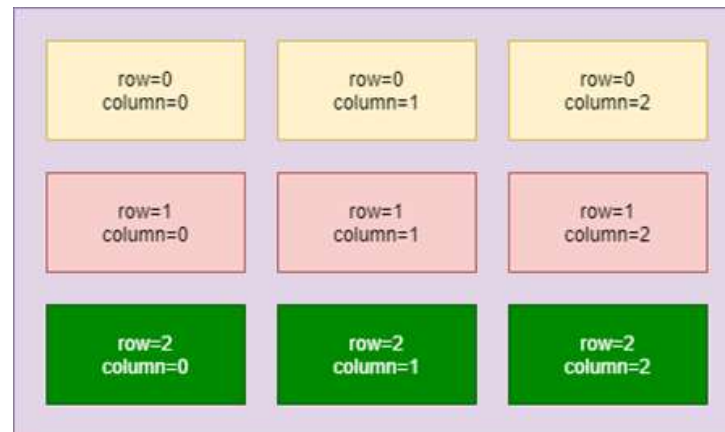
Geometría Grid (I)

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.

```
from tkinter import *

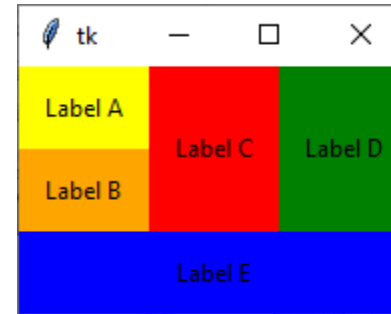
class Ventana(object):
    __ventana: object
    def __init__(self):
        self.__ventana=Tk()
        self.__ventana.title('Geometría de Celdas en Ventana')
        for r in range(0, 5):
            for c in range(0, 5):
                cell = Entry(self.__ventana, width=10)
                cell.grid(padx=5, pady=5, row=r, column=c)
                cell.insert(0, '{}, {}'.format(r, c))
    def ejecutar(self):
        self.__ventana.mainloop()

if __name__ == '__main__':
    ventana=Ventana()
    ventana.ejecutar()
```



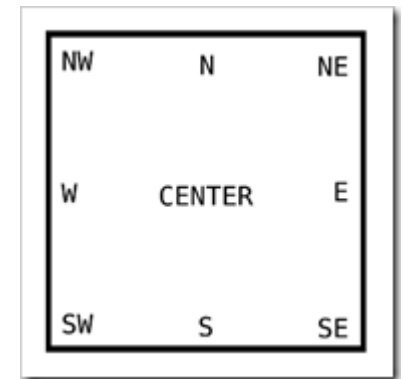
Geometría Grid (II)

```
import tkinter as tk
class Aplicacion(tk.Tk):
    def __init__(self):
        super().__init__()
        label_a = tk.Label(self, text="Label A", bg="yellow")
        label_b = tk.Label(self, text="Label B", bg="orange")
        label_c = tk.Label(self, text="Label C", bg="red")
        label_d = tk.Label(self, text="Label D", bg="green")
        label_e = tk.Label(self, text="Label E", bg="blue")
        opts = { 'ipadx': 10, 'ipady': 10, 'sticky': 'nswe' }
        label_a.grid(row=0, column=0, **opts)
        label_b.grid(row=1, column=0, **opts)
        label_c.grid(row=0, column=1, rowspan=2, **opts)
        label_d.grid(row=0, column=2, rowspan=2, **opts)
        label_e.grid(row=2, column=0, columnspan=3, **opts)
if __name__ == '__main__':
    app = Aplicacion()
    app.mainloop()
```



La propiedad **'sticky'** indica a qué borde se alinea el widget, expresado en las direcciones cardinales: Norte, Sur, Oeste y Este. Estos valores están representados por las constantes de Tkinter `tk.N`, `tk.S`, `tk.W`, y `tk.E`, así como las versiones combinadas `tk.NW`, `tk.NE`, `tk.SW` y `tk.SE`. Por ejemplo, `sticky = tk.N` alinea el widget con el borde superior de la celda (norte), mientras que `sticky = tk.SE` coloca el widget en la esquina inferior derecha esquina de la celda (sureste).

Estas anclas de dirección aplican cuando el widget tiene un tamaño inferior al del contenedor, entonces el widget se une al contenedor según el punto cardinal especificado.

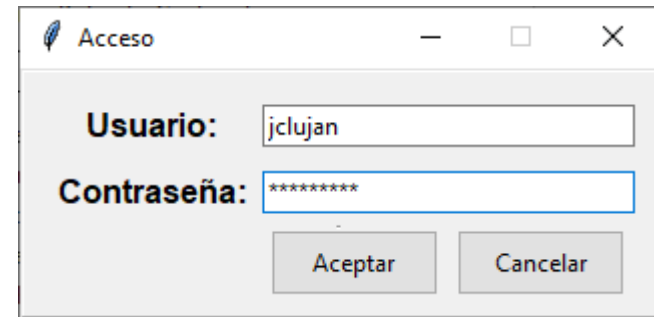


Geometría Grid (III)

```
from tkinter import *
from tkinter import ttk, font
class Aplicacion():
    __ventana=None
    __usuario=None
    __clave=None
    def __init__(self):
        self.__ventana = Tk()
        self.__ventana.title("Acceso")
        self.__ventana.resizable(0,0)
        fuente = font.Font(weight='bold')
        self.marco = ttk.Frame(self.__ventana, borderwidth=2,
                                relief="raised", padding=(10,10))
        self.usuarioLbl = ttk.Label(self.marco, text="Usuario:",
                                     font=fuente, padding=(5,5))
        self.contraseniaLbl = ttk.Label(self.marco, text="Contraseña:",
                                           font=fuente, padding=(5,5))
        self.__usuario = StringVar()
        self.__clave = StringVar()
        self.__usuario.set("")
        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=self.quit)
        self.marco.grid(column=0, row=0)
        self.usuarioLbl.grid(column=0, row=0)
        self.ctext1.grid(column=1, row=0, columnspan=2)
        self.contraseniaLbl.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)
        self.ctext1.focus_set()
        self.__ventana.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 testAPP():
    mi_app = Aplicacion()
    return 0
if __name__ == '__main__':
    testAPP()
```



Geometría Place (I)

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.

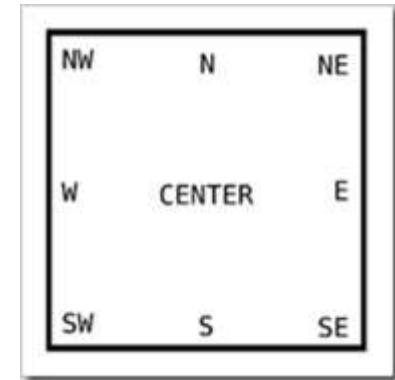
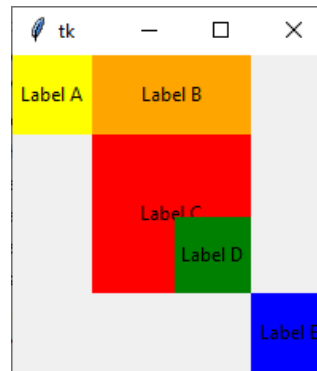
Se sabe que una ventana tiene una anchura y una altura determinadas (normalmente, medida en píxeles).

Este método para colocar un widgets simplemente se tendrá 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.

```
import tkinter as tk
class App(tk.Tk):
    def __init__(self):
        super().__init__()
        label_a = tk.Label(self, text="Label A", bg="yellow")
        label_b = tk.Label(self, text="Label B", bg="orange")
        label_c = tk.Label(self, text="Label C", bg="red")
        label_d = tk.Label(self, text="Label D", bg="green")
        label_e = tk.Label(self, text="Label E", bg="blue")
        label_a.place(relwidth=0.25, relheight=0.25)
        label_b.place(x=100, anchor=tk.N,
            width=100, height=50)
        label_c.place(relx=0.5, rely=0.5, anchor=tk.CENTER,
            relwidth=0.5, relheight=0.5)
        label_d.place(in_=label_c, anchor=tk.N + tk.W,
            x=2, y=2, relx=0.5, rely=0.5,
            relwidth=0.5, relheight=0.5)
        label_e.place(x=200, y=200, anchor=tk.S + tk.E,
            relwidth=0.25, relheight=0.25)
if __name__ == "__main__":
    app = App()
    app.mainloop()
```



La primera etiqueta se coloca con las opciones `relwidth` y `relheight` configuradas en 0.25, lo que significa que su ancho y alto son el 25% de su contenedor. Por defecto, los widgets se colocan en la posición `x = 0` y `y = 0`, alineado al noroeste, es decir, la esquina superior izquierda de la pantalla.

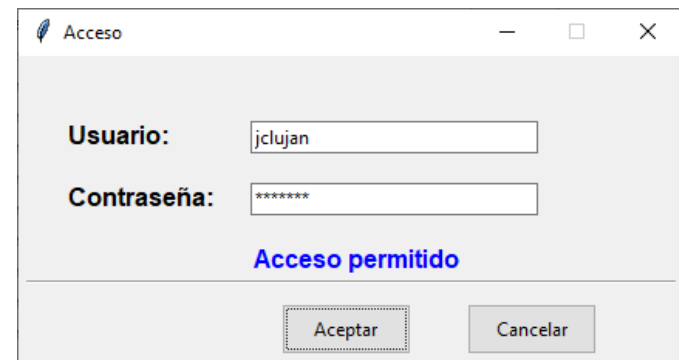
Geometría Place (II)

```
from tkinter import *
from tkinter import ttk, font
class Aplicacion():
    __ventana: object
    __clave: object
    __usuario: object
    def __init__(self):
        self.__ventana = Tk()
        self.__ventana.geometry("430x200")
        self.__ventana.resizable(0,0) # la ventana no se puede redimensionar
        self.__ventana.title("Acceso")
        self.fuente = font.Font(weight='bold')
        self.etiq1 = ttk.Label(self.__ventana, text="Usuario:",
                               font=self.fuente)
        self.etiq2 = ttk.Label(self.__ventana, text="Contraseña:",
                               font=self.fuente)
        self.__mensaje = StringVar()
        self.etiq3 = ttk.Label(self.__ventana, textvariable=self.__mensaje,
                               font=self.fuente, foreground='blue')
        self.__usuario = StringVar()
        self.__clave = StringVar()
        self.__usuario.set("")
        self.ctext1 = ttk.Entry(self.__ventana,
                                textvariable=self.__usuario, width=30)
        self.ctext2 = ttk.Entry(self.__ventana,
                                textvariable=self.__clave, width=30,
                                show="*")
        self.separ1 = ttk.Separator(self.__ventana, orient=HORIZONTAL)
        self.boton1 = ttk.Button(self.__ventana, text="Aceptar",
                                 padding=(5,5), command=self.aceptar)
        self.boton2 = ttk.Button(self.__ventana, text="Cancelar",
                                 padding=(5,5), command=quit)
        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.ctext1.focus_set()
self.ctext2.bind('<Button-1>', self.borrar_mensaje)
self.__ventana.mainloop()
def aceptar(self):
    if self.__clave.get() == 'tkinter':
        self.etiq3.configure(foreground='blue')
        self.__mensaje.set("Acceso permitido")
    else:
        self.etiq3.configure(foreground='red')
        self.__mensaje.set("Acceso denegado")
def borrar_mensaje(self, evento):
    self.__clave.set("")
    self.__mensaje.set("")
def testAPP():
    mi_app = Aplicacion()
    return 0
if __name__ == '__main__':
    testAPP()
```

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_mensaje**' 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.



Ventanas de aplicación y de diálogo (I)

Los gestores de geometría vistos, se utilizan para diseñar las ventanas de una aplicación. 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.

El siguiente ejemplo muestra una ventana de aplicación con un botón '**Abrir**' que cada vez que se presione abre una ventana hija (ventana de diálogo) diferente y en una posición diferente.

Las ventanas hijas que se vayan generando se sitúan siempre en un primer plano con respecto a las creadas con anterioridad.

Si se abren varias ventanas se podrá 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**, con algunas diferencias en el código:

La ventana de aplicación se define con **Tk()**: **self.__ventana = 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.__ventana.mainloop()**

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

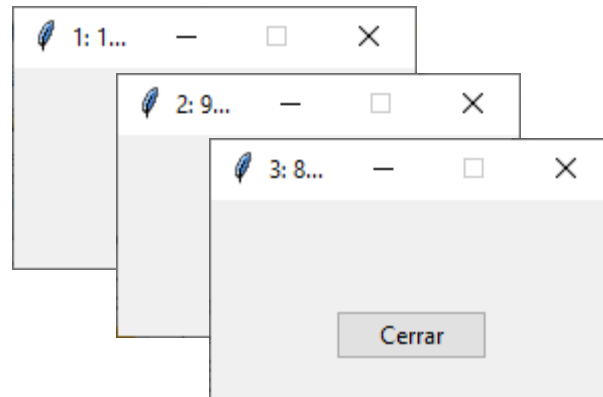
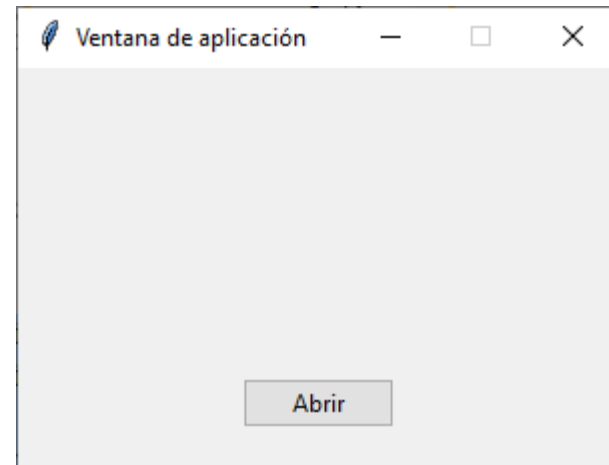
self.__ventana.wait_window(self.__dialogo)

Ventanas de aplicación y de diálogo (II)

```
from tkinter import *
from tkinter import ttk
class 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
    __ventanaPrincipal=None
    __dialogo=None
    def __init__(self):
        self.__ventanaPrincipal = Tk()
        self.__ventanaPrincipal.geometry('300x200+500+50')
        self.__ventanaPrincipal.resizable(0,0)
        self.__ventanaPrincipal.title("Ventana de aplicación")
        boton = ttk.Button(self.__ventanaPrincipal, text='Abrir',
                           command=self.abrir)
        boton.pack(side=BOTTOM, padx=20, pady=20)
        self.__ventanaPrincipal.mainloop()
    def abrir(self):
        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)
        self.__ventanaPrincipal.wait_window(self.__dialogo)
```

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

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



Ventanas modales y no modales (I)

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 se cierra 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.

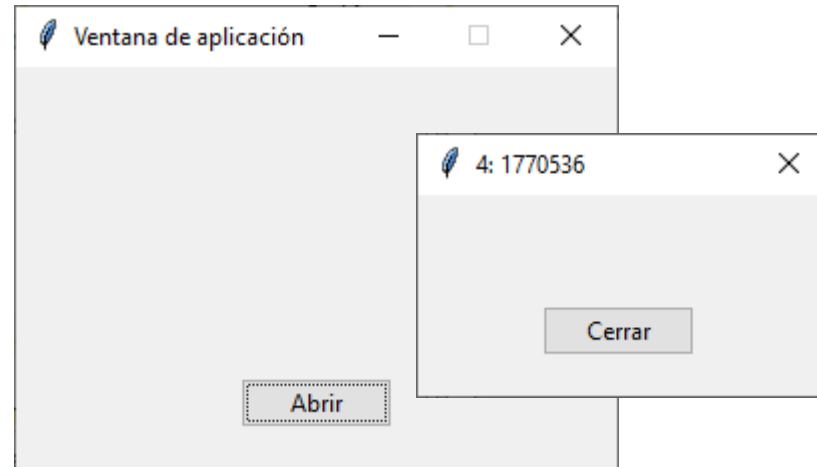
El caso contrario, es el de las **ventanas modales**. Cuando una ventana modal está abierta no será posible interactuar con otras ventanas de la aplicación hasta que ésta sea cerrada. Un ejemplo típico es el de algunas ventanas de diálogo que se utilizan para establecer las preferencias de las aplicaciones, que obligan a ser cerradas antes de permitirse la apertura de otras.

Para demostrarlo, se utilizará 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

Ventanas modales y no modales (II)

```
from tkinter import *
from tkinter import ttk
class Aplicacion():
    ## variables de clase
    ventana = 0
    posx_y = 0
    __ventanaPrincipal: object
    __dialogo: object
    def __init__(self):
        self.__ventanaPrincipal = Tk()
        self.__ventanaPrincipal.geometry('300x200+500+50')
        self.__ventanaPrincipal.resizable(0,0)
        self.__ventanaPrincipal.title("Ventana de aplicación")
        boton = ttk.Button(self.__ventanaPrincipal, text='Abrir',
                           command=self.abrir)
        boton.pack(side=BOTTOM, padx=20, pady=20)
        self.__ventanaPrincipal.mainloop()
    def abrir(self):
        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)
        self.__dialogo.transient(master=self.__ventanaPrincipal)
        self.__dialogo.grab_set()
        self.__ventanaPrincipal.wait_window(self.__dialogo)
    def testAPP():
        mi_app = Aplicacion()
        return(0)
if __name__ == '__main__':
    testAPP()
```



Convierte la ventana 'self.__dialogo' en transitoria con respecto a su ventana maestra 'self.__ventanaPrincipal'.

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.

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.

Variables de control (I)

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.

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

`entero = IntVar()` # Declara variable de tipo entera

`unFlotante = DoubleVar()` # Declara variable de tipo flotante

`cadena = StringVar()` # Declara variable de tipo cadena

`booleano = BooleanVar()` # Declara variable de tipo booleana

También se le puede dar un valor a una variable:

`tituloAPP=StringVar(value='Recarga de Sube')`

Variables de control (II)

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('Luis Artime')
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('Usuario:', 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.

```
def cambia(*args):
    print("Ha cambiado su valor")
def lee(*args):
    print("Ha sido leído 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 (I)

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 validar la información y realizar los cálculos después de que sea introducida, por ejemplo, después de presionar un botón.
- Otra posibilidad podría ser haciendo 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.

Ejemplo de la validación posterior a la introducción de los datos, ya se vio en el programa de conversión de pulgadas a centímetros.

```
ttk.Button(mainframe, text="Calcular", command=self.calcular).grid(column=2, row=3, sticky=W)
```

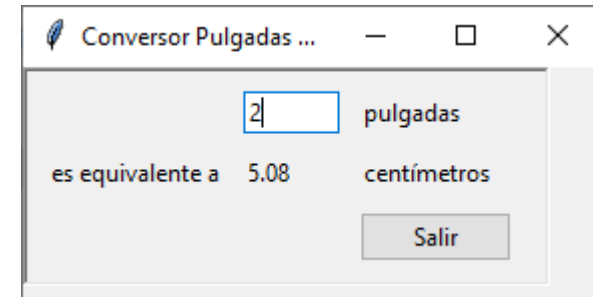
```
def calcular(self):  
    try:  
        valor=float(self.pulgadasEntry.get())  
        self.__centimetros.set(2.54*valor)  
    except ValueError:  
        messagebox.showerror(title='Error de tipo',  
                              message='Debe ingresar un valor numérico')  
        self.__pulgadas.set("")  
        self.pulgadasEntry.focus()
```

Estrategias para validar y calcular datos (II)

```
from tkinter import *
from tkinter import ttk, messagebox
class Aplicacion():
    __ventana: object
    __pulgadas: object
    __centimetros: object
    def __init__(self):
        self.__ventana = Tk()
        self.__ventana.geometry('290x115')
        self.__ventana.title('Conversor Pulgadas a Centímetros')
        mainframe = ttk.Frame(self.__ventana, padding='5 5 12 5')
        mainframe.grid(column=0, row=0, sticky=(N, W, E, S))
        mainframe.columnconfigure(0, weight=1)
        mainframe.rowconfigure(0, weight=1)
        mainframe['borderwidth'] = 2
        mainframe['relief'] = 'sunken'
        self.__pulgadas = StringVar()
        self.__centimetros = StringVar()
        self.__pulgadas.trace('w', self.calcular)
        self.pulgadasEntry = ttk.Entry(mainframe, width=7, textvariable=self.__pulgadas)
        self.pulgadasEntry.grid(column=2, row=1, sticky=(W, E))
        ttk.Label(mainframe, textvariable=self.__centimetros).grid(column=2, row=2, sticky=(W, E))
        ttk.Button(mainframe, text='Salir', command=self.__ventana.destroy).grid(column=3, row=3, sticky=W)
        ttk.Label(mainframe, text='pulgadas').grid(column=3, row=1, sticky=W)
        ttk.Label(mainframe, text='es equivalente a').grid(column=1, row=2, sticky=E)
        ttk.Label(mainframe, text='centímetros').grid(column=3, row=2, sticky=W)
        for child in mainframe.winfo_children():
            child.grid_configure(padx=5, pady=5)
        self.pulgadasEntry.focus()
        self.__ventana.mainloop()
```

```
def calcular(self, *args):
    if self.pulgadasEntry.get()!="":
        try:
            valor=float(self.pulgadasEntry.get())
            self.__centimetros.set(2.54*valor)
        except ValueError:
            messagebox.showerror(title='Error de tipo',
                                message='Debe ingresar un valor numérico')
            self.__pulgadas.set("")
            self.pulgadasEntry.focus()
    else:
        self.__centimetros.set("")

def testAPP():
    mi_app = Aplicacion()
    if __name__ == '__main__':
        testAPP()
```



Se define una traza con la variable de entrada, 'self.__pulgadas', self.__pulgadas.trace('w', self.calcular), para detectar cambios en los valores ingresados. Si se producen cambios se llama a la función 'self.calcular' para validación y para calcular el equivalente a centímetros de las pulgadas ingresadas.

Menú de opciones (I)



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.

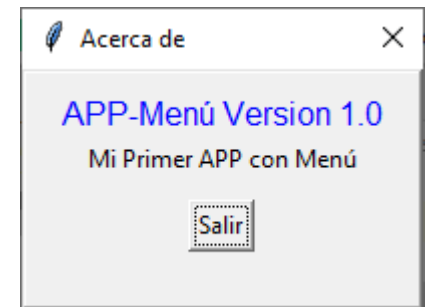
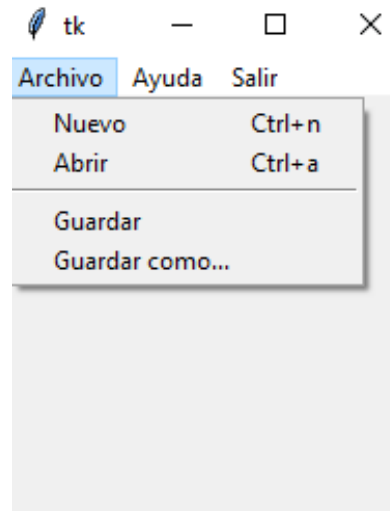
Entre los tipos de opciones que se pueden incluir en un menú se encuentran aquellas que su estado representan un valor lógico de activada o desactivada (**add_checkbutton**); las que permiten elegir una opción de entre varias existentes (**add_radiobutton**) y las que ejecutan directamente un método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

Menú de opciones (II)

```
import tkinter as tk
from tkinter import *
from tkinter import ttk, font
class App(tk.Tk):
    __version__='Version 1.0'
    def __init__(self):
        super().__init__()
        self.fuente = font.Font(weight='normal')
        barraMenu=Menu(self)
        menuArchivo=Menu(barraMenu, tearoff=0)
        menuAyuda=Menu(barraMenu, tearoff=0)
        menuSalir=Menu(barraMenu, tearoff=0)
        menuArchivo.add_command(label="Nuevo",
                                command=self.nuevo, accelerator="Ctrl+n")
        menuArchivo.add_command(label="Abrir",
                                command=self.abrir, accelerator='Ctrl+a')
        menuArchivo.add_separator()
        menuArchivo.add_command(label="Guardar")
        menuArchivo.add_command(label="Guardar como...")
        menuAyuda.add_command(label='Acerca de...', command=self.acercaDe)
        menuSalir.add_command(label='Salir Ctrl+q', command=self.destroy)
        barraMenu.add_cascade(label="Archivo", menu=menuArchivo)
        barraMenu.add_cascade(label="Ayuda", menu=menuAyuda)
        barraMenu.add_cascade(label="Salir", menu=menuSalir)
        self.config(menu=barraMenu)
        self.bind("<Control-n>",
                  lambda event: self.nuevo())
        self.bind("<Control-a>",
                  lambda event: self.abrir())
        self.bind("<Control-q>",
                  lambda event: self.destroy())
    def nuevo(self):
        print('Nuevo')
    def abrir(self):
        print('Abrir')
```

```
def acercaDe(self, *args):
    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)
    etiq2 = Label(marco1, text="APP-Menú "+self.__version__,
                  foreground='blue', font=self.fuente)
    etiq2.pack(side=TOP, padx=10)
    etiq3 = Label(marco1,
                  text="Mi Primer APP con Menú")
    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)
    self.wait_window(acerca)
if __name__ == "__main__":
    app = App()
    app.mainloop()
```



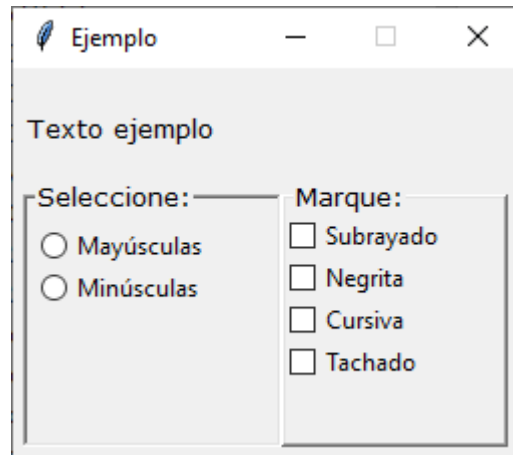
LabelFrame, RadioButton y CheckButton (I)

El widget LabelFrame es una variante del widget Frame.

Dibuja un rectángulo que agrupa a sus widgets hijos, permitiendo poner un título al grupo de widgets.

Los widgets agrupados deben estar relacionados.

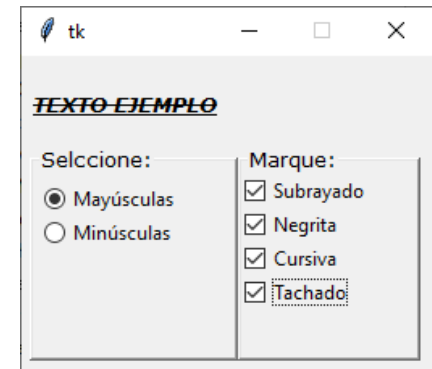
Por ejemplo, agrupar un grupo de widgets RadioButton, de los cuales se elige uno, o agrupar un grupo de CheckButton, que configuran varias opciones.



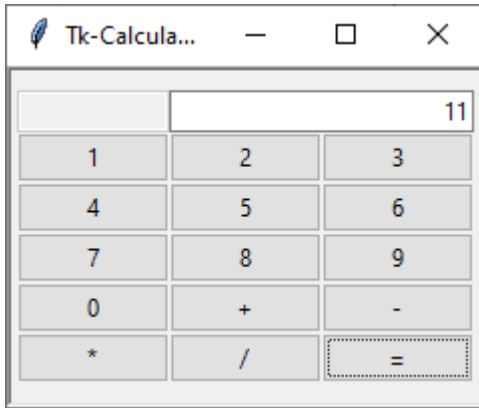
LabelFrame, RadioButton y CheckButton (II)

```
from tkinter import ttk, font
import tkinter as tk
class LabelConEstilo(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title('Ejemplo')
        self.resizable(0,0)
        self.geometry('255x195')
        self.config(padx=5, pady=5)
        fuente=font.Font(font='Verdana 10',weight='normal')
        self.valorMM=tk.IntVar()
        self.__texto=tk.StringVar()
        self.__texto.set('Texto ejemplo')
        # variables booleanas que controlan el estilo del texto
        self.subrayado=tk.BooleanVar()
        self.negrita=tk.BooleanVar()
        self.cursiva=tk.BooleanVar()
        self.tachado=tk.BooleanVar()
        # Elementos de la ventana
        self.textoLbl=ttk.Label(self, textvariable=self.__texto, font=fuente)
        opts = {'ipadx': 15, 'ipady': 15, 'sticky': 'nsw'}
        self.textoLbl.grid(row=0, column=0, **opts, columnspan=2)
        labelFrameSeleccione=tk.LabelFrame(self, text='Seleccione:', font=fuente,borderwidth=2,
            relief='raised', padx=5, pady=5)
        labelFrameSeleccione.grid(row=1, column=0, **opts)
        labelFrameMarque=tk.LabelFrame(self, text='Marque:', font=fuente,borderwidth=2,
            relief="raised")
        labelFrameMarque.grid(row=1, column=1, **opts)
        ttk.Radiobutton(labelFrameSeleccione, text='Mayúsculas', value=0, variable=self.valorMM,
            command=self.cambiaValorMM).grid(row=2, column=0, columnspan=1, sticky='w')
        ttk.Radiobutton(labelFrameSeleccione, text='Minúsculas', value=1, variable=self.valorMM,
            command=self.cambiaValorMM).grid(row=3, column=0, columnspan=1, sticky='w')
        ttk.Checkbutton(labelFrameMarque, text='Subrayado', variable=self.subrayado,
            command=self.cambiarEstilo).grid(row=2, column=1, columnspan=1, sticky='w')
        ttk.Checkbutton(labelFrameMarque, text='Negrita', variable=self.negrita,
            command=self.cambiarEstilo).grid(row=3, column=1, columnspan=1, sticky='w')
        ttk.Checkbutton(labelFrameMarque, text='Cursiva', variable=self.cursiva,
            command=self.cambiarEstilo).grid(row=4, column=1, columnspan=1, sticky='w')
        ttk.Checkbutton(labelFrameMarque, text='Tachado', variable=self.tachado,
            command=self.cambiarEstilo).grid(row=5, column=1, columnspan=1, sticky='w')
        self.valorMM.set(-1)
```

```
def cambiaValorMM(self):
    if self.valorMM.get()==0:
        self.__texto.set(self.__texto.get().upper())
    else:
        if self.valorMM.get()==1:
            self.__texto.set(self.__texto.get().lower())
def cambiarEstilo(self):
    subrayado=' underline ' if self.subrayado.get()==True else ''
    negrita=' bold ' if self.negrita.get()==True else ''
    cursiva=' italic ' if self.cursiva.get()==True else ''
    tachado=' overstrike ' if self.tachado.get()==True else ''
    fuente='Verdana 10'+subrayado+negrita+cursiva+tachado
    self.textoLbl.configure(font=fuente)
def testAPP():
    app=LabelConEstilo()
    app.mainloop()
if __name__=='__main__':
    testAPP()
```



Ejemplo-Calculadora



Para poder implementar la calculadora, es necesario indagar sobre funciones parciales, provistas en el módulo estándar de Python, **functools**.

Partial

Ésta es probablemente una de las funciones más útiles en toda la librería estándar **functools**

partial() recibe una función A con sus respectivos argumentos y

retorna una nueva función B que, al ser llamada, equivale a llamar a la función A con los argumentos provistos.

Será utilizada a la hora de proveer el comando a ejecutar por cada botón del panel de la calculadora.

```
ttk.Button(mainframe, text='0', command=partial(self.ponerNUMERO, '0')).grid(column=1, row=6, sticky=W)
ttk.Button(mainframe, text='+', command=partial(self.ponerOPERADOR, '+')).grid(column=2, row=6, sticky=W)
```

En el siguiente ejemplo, puede verse el uso de funciones parciales, y la invocación a las mismas:

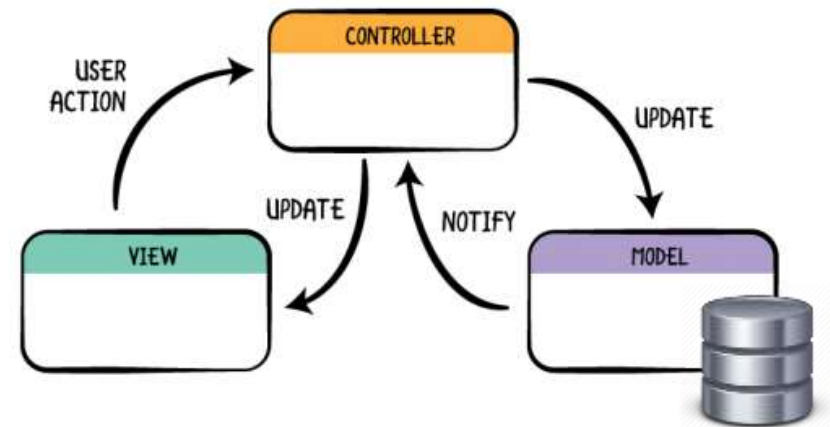
```
from functools import partial
def ponerNumero(num):
    print(num)
def testAPP():
    ponerNumero(8)
    funcionParcial=partial(ponerNumero,5)
    funcionParcial()
    otraFuncionParcial=partial(ponerNumero,7)
    otraFuncionParcial()
if __name__ == '__main__':
    testAPP()
```

Arquitectura MVC (Model-View-Controller)

Algunos de los aspectos centrales de la arquitectura MVC son los siguientes:

- El *Modelo* es donde residen los datos. Cosas como la persistencia, los objetos de modelo, los analizadores, los manejadores, residen allí.
- La *Vista* es la cara visible de la aplicación. Sus clases a menudo son reutilizables ya que no contienen ninguna lógica específica del dominio del problema. Por ejemplo, un Frame es una Vista que incluye a otros widgets, es reutilizable y extensible.
- El *Modelo* y la *Vista* nunca interactúan directamente.
- El *Controlador* media entre la *Vista* y el *Modelo*.
- Cada vez que la *Vista* necesita acceder a datos de back-end, solicita al *Controlador* su intervención con el *Modelo* para obtener los datos requeridos.

La gran ventaja que posee esta técnica de programación es que permite modificar cada uno de ellos sin necesidad de modificar los demás, de modo de desarrollar aplicaciones modulares y escalables que se puedan actualizar fácilmente y añadir o eliminar nuevos módulos o funcionalidades de forma paquetizada, ya que cada “paquete” utiliza el mismo sistema con sus vistas, modelos y controladores.



Ejemplo de utilización del MVC

La empresa de desarrollo de software «IL PHONO», para la que usted es su desarrollador junior, le solicita construir una aplicación, para manejar los contactos, solicitada por un cliente de Australia, para ello le provee el siguiente diseño que se observa en la figura.

La vista deberá proveer un lista de los contactos existentes, mostrando Apellido y nombre de los contactos, un LabelFrame con los datos del contacto, apellido, nombre, email y teléfono.

La aplicación deberá contar con botones:

The mockup shows a window titled 'Lista de Contactos'. On the left is a list of contacts: Rueda, Melisa; López, Carlos; Pérez, Maira; Altamirano, Sandra; Artime, Luis; Burruchaga, Jorge; Ortiz, Emilia; and Latorre, Andrea. On the right is a 'Contacto' form with input fields for 'Apellido', 'Nombre', 'Email', and 'Teléfono'. Below these fields are 'Borrar' and 'Guardar' buttons. At the bottom right is an 'Agregar Contacto' button.

Botón	Acción
	Borra el contacto seleccionado, cuyos datos se muestran el LabelFrame Contacto.
	Guarda y actualiza los datos del contacto, cuyos datos se muestran en el LabelFrame Contacto.
	Abre una nueva ventana para ingresar los datos de un contacto nuevo.

La selección de un contacto de la lista se hace con doble click del mouse.

La aplicación debe permitir almacenar y recuperar los datos de contactos de un archivo JSON, denominado «contactos.json»

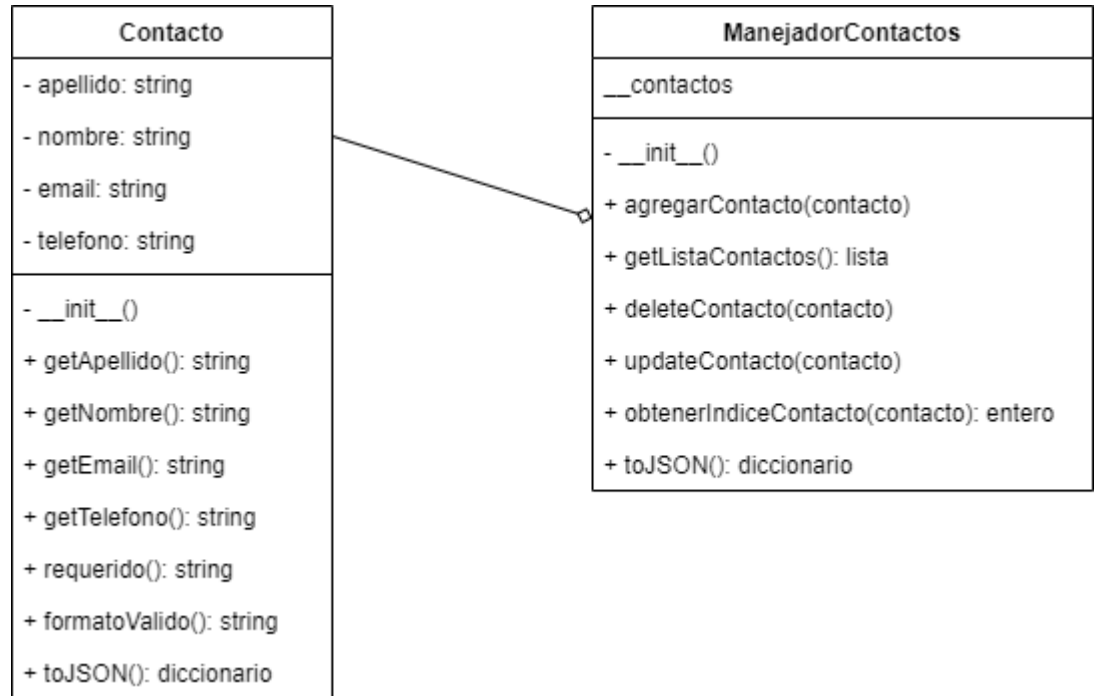
El Modelo(I)

```
import re
class Contacto(object):
    emailRegex = re.compile(r"([a-z0-9_\-\.])+@[a-z0-9_\-\.]+\.[a-z]{2,15}")
    telefonoRegex = re.compile(r"([0-9]{3})[0-9]{7}")
    __apellido: object
    __nombre: object
    __email: object
    __telefono: object
    def __init__(self, apellido, nombre, email, telefono):
        self.__apellido=self.requerido(apellido, 'Apellido es un valor requerido')
        self.__nombre = self.__nombre=self.requerido(nombre, 'Nombre es un valor requerido')
        self.__email = self.formatoValido(email, Contacto.emailRegex, 'Email no tiene formato correcto')
        self.__telefono = self.formatoValido(telefono, Contacto.telefonoRegex, 'Teléfono no tiene formato correcto')
    def getApellido(self):
        return self.__apellido
    def getNombre(self):
        return self.__nombre
    def getEmail(self):
        return self.__email
    def getTelefono(self):
        return self.__telefono
    def requerido(self, valor, mensaje):
        if not valor:
            raise ValueError(mensaje)
        return valor
    def formatoValido(self, valor, regex, mensaje):
        if not valor or not regex.match(valor):
            raise ValueError(mensaje)
        return valor
    def toJSON(self):
        d = dict(
            __class__=self.__class__.__name__,
            __atributos__=dict(
                apellido=self.__apellido,
                nombre=self.__nombre,
                email=self.__email,
                telefono=self.__telefono
            )
        )
        return d
```

Contacto
- apellido: string
- nombre: string
- email: string
- telefono: string
- __init__()
+ getApellido(): string
+ getNombre(): string
+ getEmail(): string
+ getTelefono(): string
+ requerido(): string
+ formatoValido(): string
+ toJSON(): diccionario

El Modelo (II)

```
from claseContacto import Contacto
class ManejadorContactos:
    indice=0
    __contactos: list
    def __init__(self):
        self.__contactos=[]
    def agregarContacto(self, contacto):
        contacto.rowid=ManejadorContactos.indice
        ManejadorContactos.indice+=1
        self.__contactos.append(contacto)
    def getListasContactos(self):
        return self.__contactos
    def deleteContacto(self, contacto):
        indice=self.obtenerIndiceContacto(contacto)
        self.__contactos.pop(indice)
    def updateContacto(self, contacto):
        indice=self.obtenerIndiceContacto(contacto)
        self.__contactos[indice]=contacto
    def obtenerIndiceContacto(self, contacto):
        bandera = False
        i=0
        while not bandera and i < len(self.__contactos):
            if self.__contactos[i].rowid == contacto.rowid:
                bandera=True
            else:
                i+=1
        return i
    def toJSON(self):
        d = dict(
            __class__=self.__class__.__name__,
            contactos=[contacto.toJSON() for contacto in self.__contactos]
        )
        return d
```



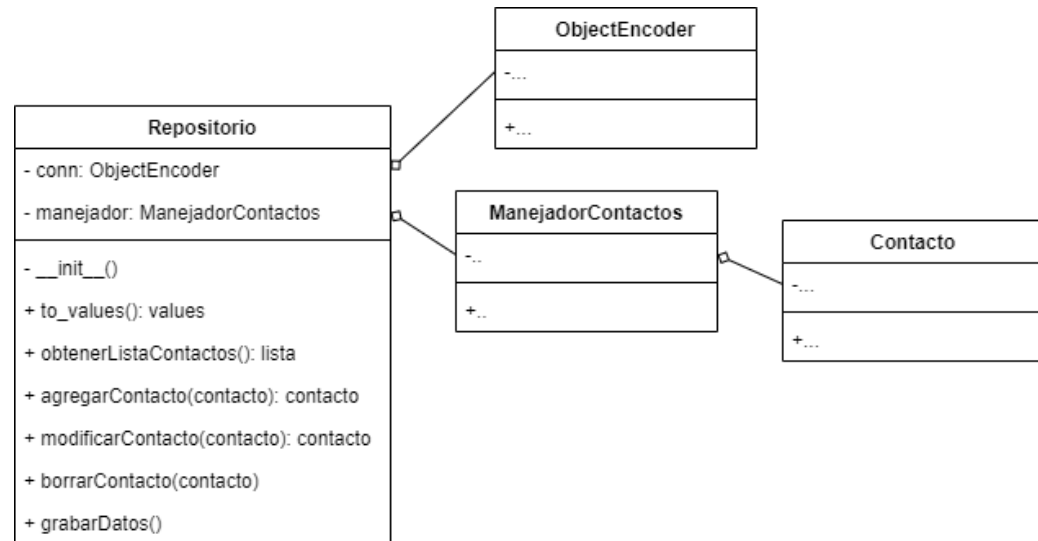
El Modelo (III)

```
import json
from pathlib import Path
from claseManejadorContactos import ManejadorContactos
from claseContacto import Contacto
class ObjectEncoder(object):
    __pathArchivo: object
    def __init__(self, pathArchivo):
        self.__pathArchivo=pathArchivo
    def decodificarDiccionario(self, d):
        if '__class__' not in d:
            return d
        else:
            class_name=d['__class__']
            class_=eval(class_name)
            if class_name=='ManejadorContactos':
                contactos=d['contactos']
                manejador=class_()
                for i in range(len(contactos)):
                    dContacto=contactos[i]
                    class_name=dContacto.pop('__class__')
                    class_=eval(class_name)
                    atributos=dContacto['__atributos__']
                    unContacto=class_(**atributos)
                    manejador.agregarContacto(unContacto)
            return manejador
    def guardarJSONArchivo(self, diccionario):
        with Path(self.__pathArchivo).open("w", encoding="UTF-8") as destino:
            json.dump(diccionario, destino, indent=4)
            destino.close()
    def leerJSONArchivo(self):
        with Path(self.__pathArchivo).open(encoding="UTF-8") as fuente:
            diccionario=json.load(fuente)
            fuente.close()
        return diccionario
```

ObjectEncoder
- pathArchivo
- __init__()
+ decodificarDiccionario(diccionario)
+ guardarJSONArchivo(diccionario)
+ leerJSONArchivo(): diccionario

El Modelo (IV)

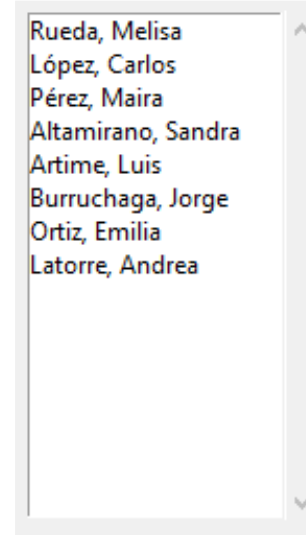
```
from claseContacto import Contacto
from claseObjectEncoder import ObjectEncoder
from claseManejadorContactos import ManejadorContactos
class RespositorioContactos(object):
    __conn: object
    __manejador: object
    def __init__(self, conn):
        self.__conn = conn
        diccionario=self.__conn.leerJSONArchivo()
        self.__manejador=self.__conn.decodificarDiccionario(diccionario)
    def obtenerListaContactos(self):
        return self.__manejador.getListaContactos()
    def agregarContacto(self, contacto):
        self.__manejador.agregarContacto(contacto)
        return contacto
    def modificarContacto(self, contacto):
        self.__manejador.updateContacto(contacto)
        return contacto
    def borrarContacto(self, contacto):
        self.__manejador.deleteContacto(contacto)
    def grabarDatos(self):
        self.__conn.guardarJSONArchivo(self.__manejador.toJSON())
```



La Vista (I)

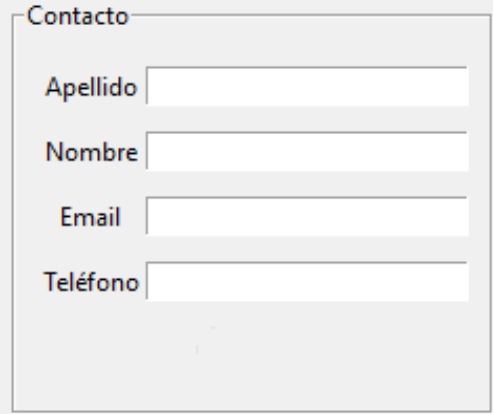
```
import tkinter as tk
from tkinter import messagebox
from claseContacto import Contacto

class ContactList(tk.Frame):
    def __init__(self, master, **kwargs):
        super().__init__(master)
        self.lb = tk.Listbox(self, **kwargs)
        scroll = tk.Scrollbar(self, command=self.lb.yview)
        self.lb.config(yscrollcommand=scroll.set)
        scroll.pack(side=tk.RIGHT, fill=tk.Y)
        self.lb.pack(side=tk.LEFT, fill=tk.BOTH, expand=1)
    def insertar(self, contacto, index=tk.END):
        text = "{}, {}".format(contacto.getApellido(), contacto.getNombre())
        self.lb.insert(index, text)
    def borrar(self, index):
        self.lb.delete(index, index)
    def modificar(self, contact, index):
        self.borrar(index)
        self.insertar(contact, index)
    def bind_doble_click(self, callback):
        handler = lambda _: callback(self.lb.curselection()[0])
        self.lb.bind("<Double-Button-1>", handler)
```



La Vista (II)

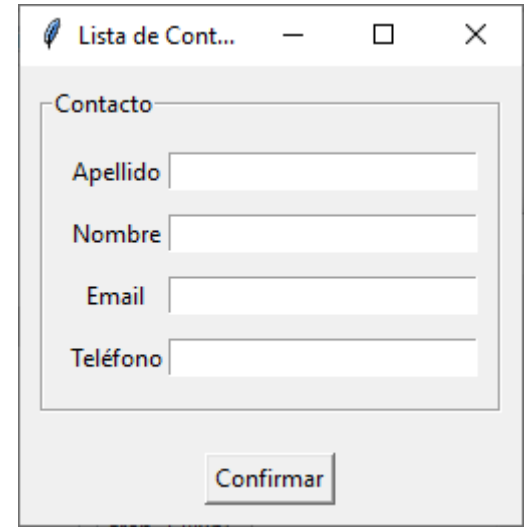
```
class ContactForm(tk.LabelFrame):
    fields = ("Apellido", "Nombre", "Email", "Teléfono")
    def __init__(self, master, **kwargs):
        super().__init__(master, text="Contacto", padx=10, pady=10, **kwargs)
        self.frame = tk.Frame(self)
        self.entries = list(map(self.crearCampo, enumerate(self.fields)))
        self.frame.pack()
    def crearCampo(self, field):
        position, text = field
        label = tk.Label(self.frame, text=text)
        entry = tk.Entry(self.frame, width=25)
        label.grid(row=position, column=0, pady=5)
        entry.grid(row=position, column=1, pady=5)
        return entry
    def mostrarEstadoContactoEnFormulario(self, contacto):
        # a partir de un contacto, obtiene el estado
        # y establece en los valores en el formulario de entrada
        values = (contacto.getApellido(), contacto.getNombre(),
                  contacto.getEmail(), contacto.getTelefono())
        for entry, value in zip(self.entries, values):
            entry.delete(0, tk.END)
            entry.insert(0, value)
    def crearContactoDesdeFormulario(self):
        #obtiene los valores de los campos del formulario
        #para crear un nuevo contacto
        values = [e.get() for e in self.entries]
        contacto=None
        try:
            contacto = Contacto(*values)
        except ValueError as e:
            messagebox.showerror("Error de Validación", str(e), parent=self)
        return contacto
    def limpiar(self):
        for entry in self.entries:
            entry.delete(0, tk.END)
```



The image shows a graphical user interface window titled "Contacto". Inside the window, there are four text input fields arranged vertically. Each field is preceded by a label: "Apellido", "Nombre", "Email", and "Teléfono". The fields are empty, and the window has a standard light gray border and title bar.

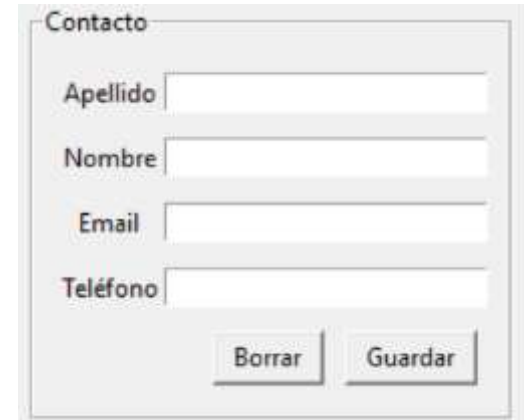
La Vista (III)

```
class NewContact(tk.Toplevel):
    def __init__(self, parent):
        super().__init__(parent)
        self.contacto = None
        self.form = ContactForm(self)
        self.btn_add = tk.Button(self, text="Confirmar", command=self.confirmar)
        self.form.pack(padx=10, pady=10)
        self.btn_add.pack(pady=10)
    def confirmar(self):
        self.contacto = self.form.crearContactoDesdeFormulario()
        if self.contacto:
            self.destroy()
    def show(self):
        self.grab_set()
        self.wait_window()
        return self.contacto
```



The screenshot shows a Tkinter window titled "Lista de Cont...". Inside, there is a frame titled "Contacto" containing four text input fields labeled "Apellido", "Nombre", "Email", and "Teléfono". Below the frame is a single button labeled "Confirmar".

```
class UpdateContactForm(ContactForm):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)
        self.btn_save = tk.Button(self, text="Guardar")
        self.btn_delete = tk.Button(self, text="Borrar")
        self.btn_save.pack(side=tk.RIGHT, ipadx=5, padx=5, pady=5)
        self.btn_delete.pack(side=tk.RIGHT, ipadx=5, padx=5, pady=5)
    def bind_save(self, callback):
        self.btn_save.config(command=callback)
    def bind_delete(self, callback):
        self.btn_delete.config(command=callback)
```



The screenshot shows a Tkinter window titled "Contacto". It contains four text input fields labeled "Apellido", "Nombre", "Email", and "Teléfono". At the bottom right of the window are two buttons: "Borrar" and "Guardar".

La Vista (IV)

```
class ContactsView(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Lista de Contactos")
        self.list = ContactList(self, height=15)
        self.form = UpdateContactForm(self)
        self.btn_new = tk.Button(self, text="Agregar Contacto")
        self.list.pack(side=tk.LEFT, padx=10, pady=10)
        self.form.pack(padx=10, pady=10)
        self.btn_new.pack(side=tk.BOTTOM, pady=5)

    def setControlador(self, ctrl):
        #vincula la vista con el controlador
        self.btn_new.config(command=ctrl.crearContacto)
        self.list.bind_doble_click(ctrl.seleccionarContacto)
        self.form.bind_save(ctrl.modificarContacto)
        self.form.bind_delete(ctrl.borrarContacto)

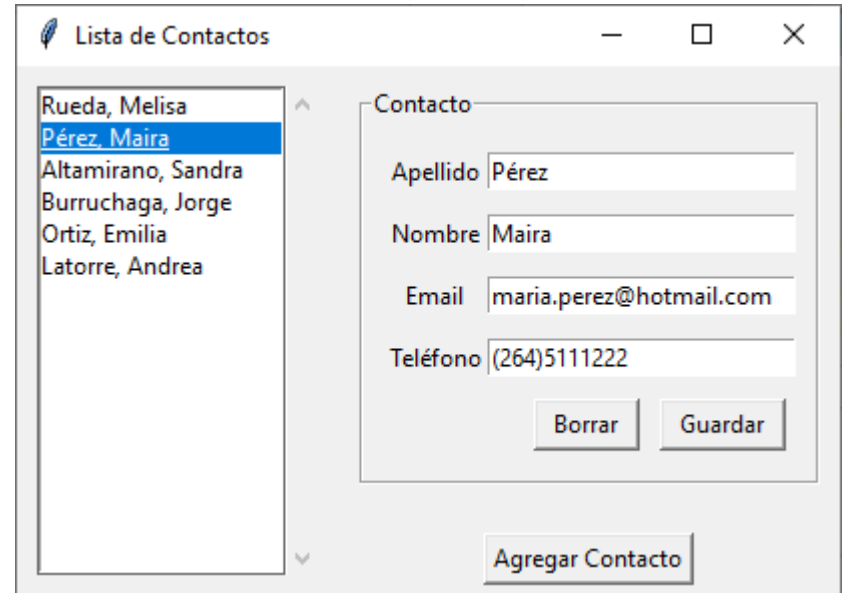
    def agregarContacto(self, contacto):
        self.list.insertar(contacto)

    def modificarContacto(self, contacto, index):
        self.list.modificar(contacto, index)

    def borrarContacto(self, index):
        self.form.limpiar()
        self.list.borrar(index)

    #obtiene los valores del formulario y crea un nuevo contacto
    def obtenerDetalles(self):
        return self.form.crearContactoDesdeFormulario()

    #Ver estado de Contacto en formulario de contactos
    def verContactoEnForm(self, contacto):
        self.form.mostrarEstadoContactoEnFormulario(contacto)
```



El Controlador (I)

```
from vistaContactos import ContactsView, NewContact
from claseManejadorContactos import ManejadorContactos
class ControladorContactos(object):
    def __init__(self, repo, vista):
        self.repo = repo
        self.vista = vista
        self.seleccion = -1
        self.contactos = list(repo.obtenerListaContactos())
    # comandos que se ejecutan a través de la vista
    def crearContacto(self):
        nuevoContacto = NewContact(self.vista).show()
        if nuevoContacto:
            contacto = self.repo.agregarContacto(nuevoContacto)
            self.contactos.append(contacto)
            self.vista.agregarContacto(contacto)
    def seleccionarContacto(self, index):
        self.seleccion = index
        contacto = self.contactos[index]
        self.vista.verContactoEnForm(contacto)
    def modificarContacto(self):
        if self.seleccion == -1:
            return
        rowid = self.contactos[self.seleccion].rowid
        detallesContacto = self.vista.obtenerDetalles()
        detallesContacto.rowid = rowid
        contacto = self.repo.modificarContacto(detallesContacto)
        self.contactos[self.seleccion] = contacto
        self.vista.modificarContacto(contacto, self.seleccion)
        self.seleccion = -1
    def borrarContacto(self):
        if self.seleccion == -1:
            return
        contacto = self.contactos[self.seleccion]
        self.repo.borrarContacto(contacto)
        self.contactos.pop(self.seleccion)
        self.vista.borrarContacto(self.seleccion)
        self.seleccion = -1
```

```
def start(self):
    for c in self.contactos:
        self.vista.agregarContacto(c)
    self.vista.mainloop()
def salirGrabarDatos(self):
    self.repo.grabarDatos()
```

PROGRAMA PRINCIPAL

```
from claseRepositorioContactosJSON import
RespositorioContactos
from vistaContactos import ContactsView
from claseControladorContactos import ControladorContactos
from claseObjectEncoder import ObjectEncoder
def main():
    conn=ObjectEncoder('contactos.json')
    repo=RespositorioContactos(conn)
    vista=ContactsView()
    ctrl=ControladorContactos(repo, vista)
    vista.setControlador(ctrl)
    ctrl.start()
    ctrl.salirGrabarDatos()
if __name__ == "__main__":
    main()
```