# Desarrollo de interfaces gráficas con TKinter

En este documento veremos cómo construir interfaces gráficas sencillas usando la librería *TKinter* de Python. Veremos cómo instalarla, configurar la apariencia general de la aplicación y responder a las acciones que haga el usuario sobre ella.

## 1. Primeros pasos con TKinter

*TKinter* (normalmente pronunciado "tequinter") es una librería que permite crear interfaces gráficas de usuario (*GUI, Graphical User Interface*) en Python. Normalmente viene incorporada ya en el propio lenguaje, cosa que podemos comprobar ejecutando el comando `python -m tkinter` (Windows) o `python -m TKinter` (Linux). Aparecerá una pequeña ventana de prueba con algunos botones para interactuar con ella.

Para incorporar *TKinter* a nuestros programas, la importaremos. Normalmente se le suele asignar un alias `tk`. También se suele utilizar un submódulo específico llamado `ttk` para algunos controles o *widgets* que han sido rediseñados con una apariencia más moderna.

```
import tkinter as tk
from tkinter import ttk
```

### 1.1. Definiendo la ventana principal

Para crear la ventana principal de la aplicación, instanciamos un objeto de tipo `Tk`. Podemos asignarle un título a la ventana con el método `title`, y también indicar un tamaño y posición iniciales con el método `geometry`:

```
import tkinter as tk
from tkinter import ttk

ventana = tk.Tk()
ventana.title("Ventana de prueba")
# Ventana de 600 x 400
# ubicada en x = 100, y = 200
ventana.geometry("600x400+100+200")

ventana.mainloop()
```

El método `mainloop` es el que muestra la ventana, y permanece en ejecución hasta que se cierre ésta. Todo lo que tengamos que hacer en la configuración de la aplicación (añadir elementos, comportamiento, etc),

deberemos hacerlo antes de iniciar este método.

# 2. Controles y contenedores

Los elementos que podemos disponer en una ventana se llaman *controles*, y pueden colocarse directamente sobre la ventana principal, o agrupados en otros elementos llamados *contenedores*, de modo que la ventana puede tener varios contenedores y/o controles.

## 2.1. Controles más habituales

Veremos algunos de los controles más habituales que podemos emplear en nuestras aplicaciones. Algunos de ellos tienen una versión mejorada o actualizada en el subpaquete `ttk` de TKinter, llamados *widgets temáticos*. Aquí podemos consultar el listado de controles incluidos.

### 2.1.1. Etiquetas

Las etiquetas sirven para mostrar información en ciertas parte de la ventana, como los nombres de los campos de un formulario a rellenar. Se definen con la clase `Label`, y al crearlas debemos indicar el contenedor donde se van a ubicar (por ejemplo, la ventana principal) y el texto que se mostrará.

```
lbl_nombre = tk.Label(ventana, text="Nombre:")
lbl_nombre.pack()
```

> **NOTA**: el método `pack` conecta la etiqueta con la ventana contenedora. Existen otras formas de añadir controles a contenedores, como veremos más adelante.

Alternativamente, podemos usar la versión actualizada de `Label` en el subpaquete `ttk`:

```
lbl_nombre = ttk.Label(ventana, text="Nombre:")
lbl_nombre.pack()
```

### 2.1.2. Botones

Los botones permiten al usuario interactuar con la aplicación pulsándolos. Se definen con la clase `Button`, indicando, como en el caso de las etiquetas, el contenedor y el texto del botón.

```
btn_aceptar = tk.Button(ventana, text="Aceptar")
btn_aceptar.pack()
```

Del mismo modo, podemos emplear la versión reciente de `ttk`:

```
btn_aceptar = ttk.Button(ventana, text="Aceptar")
btn_aceptar.pack()
```

### 2.1.3. Cuadros de texto

La clase `Entry` permite definir cuadros de texto de **una sola línea**. Podemos crear el control simplemente pasándole el elemento contenedor:

```
txt_nombre = tk.Entry(ventana)
txt_nombre.pack()
```

Para obtener en un momento dado el valor contenido en un cuadro de texto, usamos su método `get`:

```
valor_escrito = txt_nombre.get()
```

La clase `Text` se emplea para **cuadros multilínea**, de forma similar a la anterior. Podemos indicar el tamaño inicial con los parámetros `width` y `height`. A la hora de recuperar el texto con `get`, debemos indicar la línea y columna donde empezar, y hasta dónde llegar. También podemos asignarle un contenido inicial con el método `insert`, especificando la posición desde donde insertar, y el texto a añadir.

```
texto = tk.Text(ventana, width=15, height=10)
# Insertamos desde el inicio del cuadro de texto
texto.insert('1.0', 'Hola, buenas')
texto.pack()
...
# Recuperar todo el texto
valor_escrito = texto.get('1.0', 'end')
```

La clase `Entry` también tiene su versión actualizada en el subpaquete `ttk`, pero `Text` no la tiene.

### 2.1.4. Casillas de verificación (*checkboxes*)

Para añadir casillas de verificación en nuestras aplicaciones usaremos la clase `Checkbutton`, indicando como es habitual el contenedor y el texto que acompañará al control:

```
check = tk.Checkbutton(ventana, text="Acepto las condiciones")
check.pack()
```

También podemos dejar el control inicialmente marcado o desmarcado, con los métodos `select` y `deselect`:

```
check = tk.Checkbutton(ventana, text="Acepto las condiciones")
check.select()
check.pack()
```

### 2.1.5. Botones de radio

Los botones de radio permiten formar un grupo de botones donde sólo uno de ellos puede estar seleccionado a la vez. Se definen con la clase `Radiobutton`, indicando el contenedor, el texto asociado al control y el valor ( `value` ) que dicho botón contiene. Puede ser un valor numérico o alfanumérico.

```
rb1 = tk.Radiobutton(ventana, text="Radio 1", value=1)
rb2 = tk.Radiobutton(ventana, text="Radio 2", value=2)
rb1.pack()
rb2.pack()
```

Igual que ocurre con los *checkboxes*, podemos usar los métodos `select` o `deselect` para dejarlos inicialmente seleccionados o no:

```
rb1 = tk.Radiobutton(ventana, text="Radio 1", value=1)
rb1.select()
...
```

### 2.1.6. Listas

Las listas que podemos definir en la mayoría de interfaces gráficas pueden ser de dos tipos: desplegables y fijas. Para definir listas **desplegables** en TKinter usamos la clase `OptionMenu`. Indicamos la ventana contenedora, junto con la variable donde nos guardaremos el valor actualmente seleccionado, y las opciones que contendrá:

```
resultado = tk.StringVar()
opciones = ['Uno', 'Dos', 'Tres']
lista = tk.OptionMenu(ventana, resultado, *opciones)
lista.pack()
```

Más adelante en este documento explicaremos con más detalle cómo usar estas variables para consultar/modificar el valor del elemento.

Alternativamente, también podemos usar la clase `Combobox` del subpaquete `ttk`. En este caso, debemos indicar la ventana contenedora y, en su propiedad `values`, los valores de la lista. La apariencia visual de este segundo control es diferente a la de *OptionMenu*.

```
from tkinter import ttk
...
opciones = ['Uno', 'Dos', 'Tres']
combo = ttk.Combobox(ventana)
combo['values'] = opciones
combo.pack()
```

Para **listas fijas** se emplea la clase `Listbox`. El parámetro `selectmode` permite definir el modo de selección (simple o múltiple, básicamente). Podemos encontrar más información aquí.

```
lista = tk.Listbox(ventana, selectmode=tk.MULTIPLE)
lista.insert(1, "Uno")
lista.insert(2, "Dos")
lista.insert(3, "Tres")
lista.pack()
```

### 2.1.7. Imágenes

A la hora de incluir imágenes en nuestras aplicaciones TKinter, podemos usar las clases `Image` e `ImageTk` de la librería *Pillow* (necesitaremos instalarla previamente con `pip install Pillow`, como se explica en este documento). Las usamos como sigue:

```
# Cargamos la imagen en objeto Image
imagen = ImageTk.PhotoImage(Image.open(r'D:\imagenes\foto.png'))
# Mostramos la imagen en una etiqueta
lbl_imagen = tk.Label(ventana, image=imagen)
lbl_imagen.pack()
```

Podemos elegir el tamaño de la imagen usando el método `resize` del objeto `Image`:

```
imagen = ImageTk.PhotoImage(Image.open(r'D:\imagenes\foto.png').resize((200, 200)))
...
```

También podemos cambiar la imagen que se esté mostrando en la etiqueta en cualquier momento, con la propiedad `image` de dicha etiqueta:

```
imagen = ImageTk.PhotoImage(Image.open('imagen2.png').resize((300, 100)))
lbl_imagen.configure(image=imagen)
lbl_imagen.image = imagen
```

### 2.1.8. Menús

La gestión de una barra de menú superior se realiza con la clase `Menu`. Asociaremos el objeto creado a la propiedad de configuración `menu` de nuestra aplicación:

```
ventana = tk.Tk()
menu = tk.Menu(ventana)
ventana.config(menu = menu)
```

Los submenús dentro de la barra principal se definen creando diferentes objetos `Menu`, y asociándolos al principal. Podemos especificar una propiedad `tearoff` para evitar que se puedan desacoplar los menús de la barra.

```
menu_archivo = tk.Menu(menu, tearoff=False)
menu_editar = tk.Menu(menu, tearoff=False)
...
menu.add_cascade(label="Archivo", menu=menu_archivo)
menu.add_cascade(label="Editar", menu=menu_editar)
...
```

Las opciones concretas dentro de cada menú se definen con el método `add_command`, para cada menú concreto. En el parámetro `command` podemos indicar qué función se ejecutará cuando elijamos esa opción

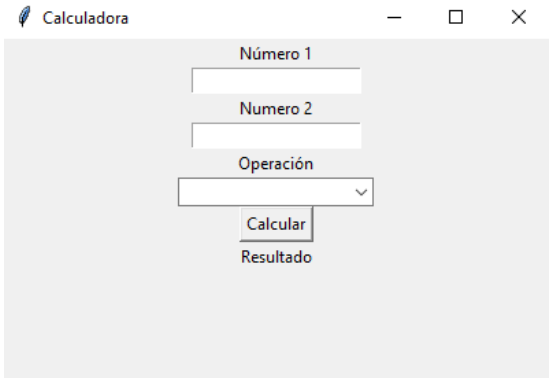
```
def nuevo():
    ...

def abrir():
    ...

menu_archivo.add_command(label="Nuevo", command=nuevo)
menu_archivo.add_command(label="Abrir", command=abrir)
menu_archivo.add_separator()
...
```

> **Ejercicio 1**:

> Crea una aplicación *TKinter* llamada **calculadora1.py** con una apariencia como esta (y sin añadir ninguna funcionalidad de momento). El desplegable debe contener los símbolos de las cuatro operaciones aritméticas básicas: +, -, * y /




> **NOTA**: la solución de este ejercicio se presenta en vídeo más adelante, junto con la del *Ejercicio 3*

## 2.2. El gestor de geometría

La forma en que se disponen los elementos en la ventana o en un contenedor se puede controlar a través del *geometry manager*. De hecho, cuando hemos invocado al método `pack` en los ejemplos anteriores para añadir controles a la ventana, estábamos utilizando una de las formas que ofrece este gestor de disponer los controles. La veremos ahora con más detalle, junto con algunas alternativas más.

### 2.2.1. *pack*

El método `pack` ubica los elementos según la configuración que se establezca como parámetro. Por defecto los ubica con su tamaño por defecto, uno debajo de otro. Pero podemos especificar una serie de parámetros adicionales para modificar esa disposición:

- `ipadx` : define un relleno a los lados del control, para darle más tamaño horizontal
- `ipady` : define un relleno arriba y abajo, para darle más tamaño vertical
- `padx` y `pady` : son similares, pero definen el relleno externo (no interno) del control.
- `fill` : indica que el control debe rellenar el contenedor en la dirección indicada: `tkinter.X` (horizontalmente), `tkinter.Y` (verticalmente) o `tkinter.BOTH` (ambas direcciones). El área de relleno dependerá de los controles que haya alrededor. Por ejemplo, si elegimos un rellenado vertical (*tkinter.Y*), pero hay un control debajo del actual, el rellenado sólo abarcará hasta donde aparezca ese control.
- `expand` : si lo ponemos a *True*, se reserva más espacio para este control respecto al resto. Si lo ponemos en varios controles, el espacio disponible se reparte uniformemente entre los controles involucrados.
- `anchor` : permite anclar el control a algún punto cardinal del contenedor: N (norte), S (sur), E (este), W (oeste), NE (nordeste)... incluso CENTER.
- `side` : indica la alineación del elemento. Por defecto es `tkinter.TOP` , pero podemos elegir otras opciones, como `tkinter.LEFT` , `tkinter.RIGHT` o `tkinter.BOTTOM` .

Veamos algunos ejemplos de uso:

```
boton1 = tk.Button(ventana, text="Botón 1")
# Expande horizontalmente
boton1.pack(expand=tk.X)

boton2 = tk.Button(ventana, text="Botón 2")
# Anclaje en esquina inferior izquierda del
# área (ampliada) disponible
boton2.pack(expand=True, anchor=tk.SW)
```

Aquí se puede consultar más información sobre el método `pack`.

### 2.2.2. *grid*

El método `grid` define una rejilla para colocar los componentes dentro, indicando en cada uno la fila, columna y número de celdas que ocupará. Usaremos los parámetros `row` y `column` para asignar la casilla inicial de cada control, y los parámetros `rowspan` y `columnspan` para expandir el control un determinado número de filas o columnas. También podemos usar los parámetros `padx` y `pady` para obtener una separación de los elementos con respecto a las celdas que los contienen.

```
texto = tk.Entry(ventana)
texto.grid(row=0, column=0)
boton1 = tk.Button(ventana, text="Botón 1")
boton1.grid(row=1, column=0, columnspan=2)
boton2 = tk.Button(ventana, text="Botón 2")
boton2.grid(row=1, column=1)
etiqueta = tk.Label(ventana, text="Etiqueta")
etiqueta.grid(row=2, column=0)
```

### 2.2.3. *place*

El método `place` ubica un elemento directamente en las coordenadas X e Y que le indiquemos. No es un método muy habitual porque, al usar coordenadas absolutas, perdemos la noción de dónde están los demás componentes, y cómo pueden solaparse. Podemos indicar coordenadas absolutas con `x` e `y`, y también coordenadas relativas con `relx` y `rely`, además de un parámetro `anchor` para definir un punto de anclaje, como ocurre con `pack`:

```
# Etiqueta en la esquina superior izquierda
etiqueta = tk.Label(ventana, text="Etiqueta")
etiqueta.place(anchor=tk.NW)
# Botón en mitad de la ventana
boton1 = tk.Button(ventana, text="Botón 1")
boton1.place(relx=0.5, rely=0.5, anchor=tk.CENTER)
```

## 2.3. Propiedades de los controles

Podemos modificar desde el código algunas propiedades de los controles, como por ejemplo tipo de letra, color de texto y de fondo, incluso el texto en sí que se está mostrando en un control.

### 2.3.1. Colores y fuentes

Los colores de los controles se pueden especificar con los parámetros `foreground` (texto) y `background` (fondo), al definir los controles. Por ejemplo:

```
etiqueta = tk.Label(ventana, text='Etiqueta',
    background='blue', foreground='white')
```

Los colores disponibles se pueden consultar aquí.

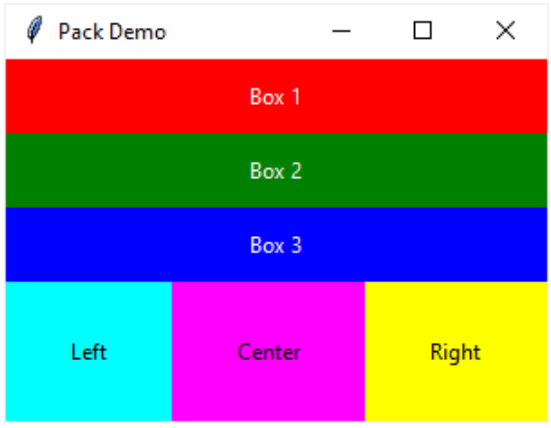
Del mismo modo, podemos cambiar el tipo de letra con el parámetro `font`:

```
etiqueta = tk.Label(ventana, text='Etiqueta',
    background='blue', foreground='white', font=('Arial', 12))
```

Respecto a la ventana principal, también podemos cambiar su color de fondo con la propiedad `background`, de este modo:

```
ventana['background'] = 'white'
```

> **Ejercicio 2**:
>
> Construye una aplicación llamada **ejercicio_gestor_geometria.py** donde, usando etiquetas, el gestor de geometría y distintos colores, obtengas una apariencia como ésta:




> **AYUDA:** vídeo con la solución del ejercicio

### 2.3.2. Estados

Algunos controles, como los botones, admiten estar en distintos estados. Por ejemplo, podemos desactivar un control con un estado `tk.DISABLED`, y activarlo cuando suceda algo.

```
boton1 = tk.Button(ventana, text="Botón 1", state=tk.DISABLED)
boton1.pack()
...
if boton1['state'] == tk.DISABLED:
    boton1['state'] = tk.NORMAL
```

### 2.3.3. Tamaños

Podemos modificar el tamaño (anchura y/o altura) de los controles cuando los definimos, a través de los parámetros `width` y `height`:

```
etiqueta = tk.Label(ventana, text="Etiqueta", width=20)
```

### 2.3.4. Valores en controles

Podemos acceder a los valores de los controles, bien para consultarlos, bien para modificarlos. Ya hemos visto en ejemplos anteriores que, por ejemplo, el método `get` de los cuadros de texto permite acceder al contenido que tienen. Sin embargo, también podemos asociar variables a los controles y, a través de esas variables, recuperar o modificar el contenido de esos controles.

Por ejemplo, para **comprobar en un momento determinado si un *checkbox* o *radio button* está marcado o no**, debemos asociarle una variable de tipo `tk.IntVar` al parámetro `variable`, y consultar con `get` el valor de dicha variable. Será 1 si está marcada y 0 si no lo está.

Así podría quedar en el caso de un *checkbox*:

```
resultado_check = tk.IntVar()
check = tk.Checkbutton(ventana, text="Acepto las condiciones",
    variable=resultado_check).pack()
...
if resultado_check.get() == 1:
    # Checkbox marcado
```

Así quedaría en el caso de un grupo de *radio buttons*:

```
valor_radio = tk.IntVar()
rb1 = tk.Radiobutton(ventana, text="Radio 1", value=1, variable=valor_radio)
rb2 = tk.Radiobutton(ventana, text="Radio 2", value=2, variable=valor_radio)
rb1.select()
rb1.pack()
rb2.pack()
...
if(valor_radio.get() == 1):
    # Radio button 1 seleccionado
```

Notar que el parámetro `value` admite valores numéricos o alfanuméricos. En el caso de que pongamos alfanuméricos, la variable ya no podría ser de tipo *IntVar*, sino que debería ser de tipo *StringVar*.

En el caso de **cuadros de texto (*Entry*) o etiquetas**, asociaremos una variable de tipo `tk.StringVar`, mediante la cual podemos asignar o consultar el texto. Lo asignamos al parámetro `textvariable`:

```
texto_etiqueta = tk.StringVar()
etiqueta = tk.Label(ventana, text="Texto inicial", textvariable=texto_etiqueta)
etiqueta.pack()

texto_entry = tk.StringVar()
texto = tk.Entry(ventana, textvariable=texto_entry)
texto.pack()
...
texto_etiqueta.set("Otro texto")
texto_entry.set("Hola")
```

> **NOTA**: el método `set` también se puede usar para cambiar el valor de otros controles, como botones de radio, o listas, y dejar seleccionado uno de los *items*.

Existen otros tipos de datos que podemos asociar a controles, aunque no son tan habituales: `tk.DoubleVar` y `tk.BooleanVar`.

## 2.4. Contenedores

Los contenedores permiten organizar los controles por grupos dentro de la aplicación, y posicionarse o bien en la ventana principal o dentro de otro contenedor. Fundamentalmente existen de dos tipos: `Frame` (panel normal donde ubicar componentes) y `LabelFrame` (panel etiquetado, con un marco que rodea al contenedor y un título para el mismo).

Por ejemplo, podemos crear un `Frame` o un `LabelFrame` asociado a la ventana principal, que contenga elementos dentro:

```
ventana = tk.Tk()

marco1 = tk.Frame(ventana)
marco1.grid(row=0, column=0)

marco2 = tk.LabelFrame(ventana, text="Marco 2")
marco2.grid(row=0, column=1)

etiqueta = tk.Label(marco1, text="Etiqueta")
etiqueta.pack()
...
```

# 3. Eventos

Los eventos son sucesos que ocurren en una aplicación y que permiten que dicha aplicación responda de algún modo. Por ejemplo, cuando el usuario pulsa un botón, o escribe un texto en un cuadro de texto, o simplemente pasa el ratón sobre una etiqueta, son eventos a los que la aplicación puede responder. Veremos a continuación cómo definir esas respuestas.

## 3.1. Ejemplo introductorio

Imaginemos que queremos que la aplicación "haga algo" cuando el usuario pulsa un botón *Aceptar*. Lo que debemos hacer es definir un parámetro `command` al definir el botón, con el nombre de la función que se ejecutará al pulsarlo:

```
btn_aceptar = tk.Button(ventana, text="Aceptar",
    command=aceptar)
btn_aceptar.pack();
```

> **NOTA**: observa cómo pasamos el *nombre* de la función, sin comillas, y sin paréntesis.

En otra parte del código podemos definir nuestra función con el código que se ejecutará al pulsar el botón.

```
def aceptar():
    # Código a ejecutar
```

## 3.2. Gestión general de eventos en *TKinter*

No todos los *widgets* de *TKinter* tienen un parámetro *command* asociado, ni todos los eventos que se pueden producir sobre un *widget* se pueden recoger con ese parámetro *command*. Por ejemplo, ¿cómo podría cambiar el color de un botón al pasar el ratón por encima?

La forma en que *TKinter* (y también otras librerías gráficas) gestionan los eventos es a través de unas funciones especiales llamadas *event handlers* o "gestores de eventos". Estos gestores reciben como parámetro un objeto con el evento que se ha producido, para poder acceder a datos del mismo. Por ejemplo, si hemos pulsado una tecla, el objeto del evento recogerá qué tecla se ha pulsado.

Para asociar un gestor de eventos con un evento en concreto, usaremos el método `bind` del objeto sobre el que queremos definir el evento. Indicaremos primero el tipo de evento que se produce, y luego el gestor o función que se encargará de gestionarlo. Por ejemplo, para responder a un evento de teclado en la aplicación podemos hacer algo así:

```
import tkinter as tk

def evento_teclado(evento):
    print(evento.char)

ventana = tk.Tk()
ventana.bind('<Key>', evento_teclado)
ventana.mainloop()
```

El código anterior saca por consola cada tecla que se pulse en la aplicación.

Ejemplos de eventos habituales que podemos poner como primer parámetro del método `bind`:

- `<Button-1>`: hacer clic en el botón izquierdo del ratón
- `<Enter>` / `<Leave>`: el ratón entra / sale del *widget* o control vinculado
- `<Motion>`: el usuario mueve el ratón dentro del control vinculado
- `<FocusIn>` / `<FocusOut>`: un foco de teclado entra/sale en el *widget*
- `<Key>`: eventos de teclado (pulsar teclas)
- `<Return>`: el usuario pulsa Intro sobre el control vinculado

## 3.3. Algunas acciones especiales

Existen algunas acciones especiales en una interfaz gráfica, que pueden llevarse a cabo con ciertos comandos ya predefinidos. Vamos a echar un vistazo a las más habituales.

### 3.3.1. Cerrar la aplicación

Además de poder cerrar la aplicación con el botón de cierre (esquina superior derecha, normalmente, en la mayoría de sistemas operativos), nos puede interesar cerrarla desde algún evento en un botón, por ejemplo. Para ello, en el código de ese evento debemos llamar al método `quit` o `destroy` de la ventana principal. Por ejemplo:

```
btn_cerrar = tk.Button(ventana, text="Cerrar", command=ventana.quit)
```

### 3.3.2. Diálogos predefinidos

TKinter incorpora algunos *popups* o cuadros de diálogo predefinidos para, por ejemplo, mostrar mensajes en la aplicación, o dejar que el usuario elija un archivo que abrir/guardar.

A la hora de **mostrar mensajes**, usaremos el objeto `messagebox` de TKinter, que ofrece diferentes métodos para mostrar distintos tipos de mensajes:

- `showinfo()` : para mensajes informativos
- `showwarning()` : para advertencias
- `showerror()` : para errores más graves
- `askquestion()` : para preguntar algo al usuario y recoger su respuesta

Aquí vemos algunos ejemplos:

```
import tkinter as tk
from tkinter import messagebox

# Pasamos el título del diálogo y el texto a mostrar
messagebox.showwarning("Cuidado", "Hay campos vacíos en el formulario")
# Recogemos respuesta
respuesta = messagebox.askquestion("Confirmación", "¿Estás seguro de continuar?")
if respuesta == "yes":
    # Confirmado
```

Para gestionar **archivos que abrir/guardar** usaremos el módulo `filedialog` de TKinter. Con el método `askopenfilename` podemos abrir un diálogo para elegir un fichero para apertura. Especificamos los parámetros `initialdir` con la carpeta inicial donde buscar, `title` con el título de la ventana de diálogo y `filetypes` con una tupla de valores con los tipos de archivo compatibles:

```
import tkinter as tk
from tkinter import filedialog

nombre_fichero = filedialog.askopenfilename(initialdir='.', title='Seleccionar imager
    filetypes=(("Imágenes JPG", "*.jpg"), ("Imágenes PNG", "*.PNG")))
imagen = ImageTk.PhotoImage(Image.open(nombre_fichero))
```

Para guardar un archivo, usamos el método `asksaveasfilename`, indicándole el título del diálogo y la extensión por defecto para guardar (si la hay):

```
import tkinter as tk
from tkinter import filedialog

nombre_fichero = filedialog.asksaveasfilename(title='Seleccionar fichero destino',
    defaultextension='.txt')
```

**Ejercicio 3:**

Crea una aplicación **calculadora2.py** que sea una versión ampliada del ejercicio *calculadora1* propuesto anteriormente. En este caso, cuando pulsemos el botón de *Calcular* deberemos recoger los valores escritos en los dos cuadros de texto y, dependiendo del elemento seleccionado en el desplegable, actualizar el valor de la etiqueta *Resultado* con el resultado de la operación. Si alguno de los dos cuadros de texto, o la operación del desplegable, están vacíos, mostraremos un mensaje de error que diga "Los campos no pueden estar vacíos".

**AYUDA:** vídeo con la solución del ejercicio

# 4. Agrupando la aplicación en una clase

Es también habitual que el código de la aplicación se agrupe en una clase que herede de `Frame`, y desde el programa principal simplemente se cree una ventana que use ese *frame*. Podría quedar algo así:

```
class Aplicacion(tk.Frame):

    def __init__(self, ventana):
        super().__init__(ventana) #Llamamos al constructor de Frame
        self.master = ventana
        self.pack()
        self.crear_elementos()

    def crear_elementos():
        # Código para definir los elementos de la aplicación

    # Aquí podrían ir otros métodos, como eventos de la aplicación

# Programa principal
ventana = tk.Tk()
app = Aplicacion(ventana)
app.mainloop()
```

**Ejercicio 4:**

Crea una aplicación **calculadora3.py** que cree una clase `Calculadora` de tipo `Frame` con el código de gestión de la calculadora del ejercicio *calculadora2* anterior.