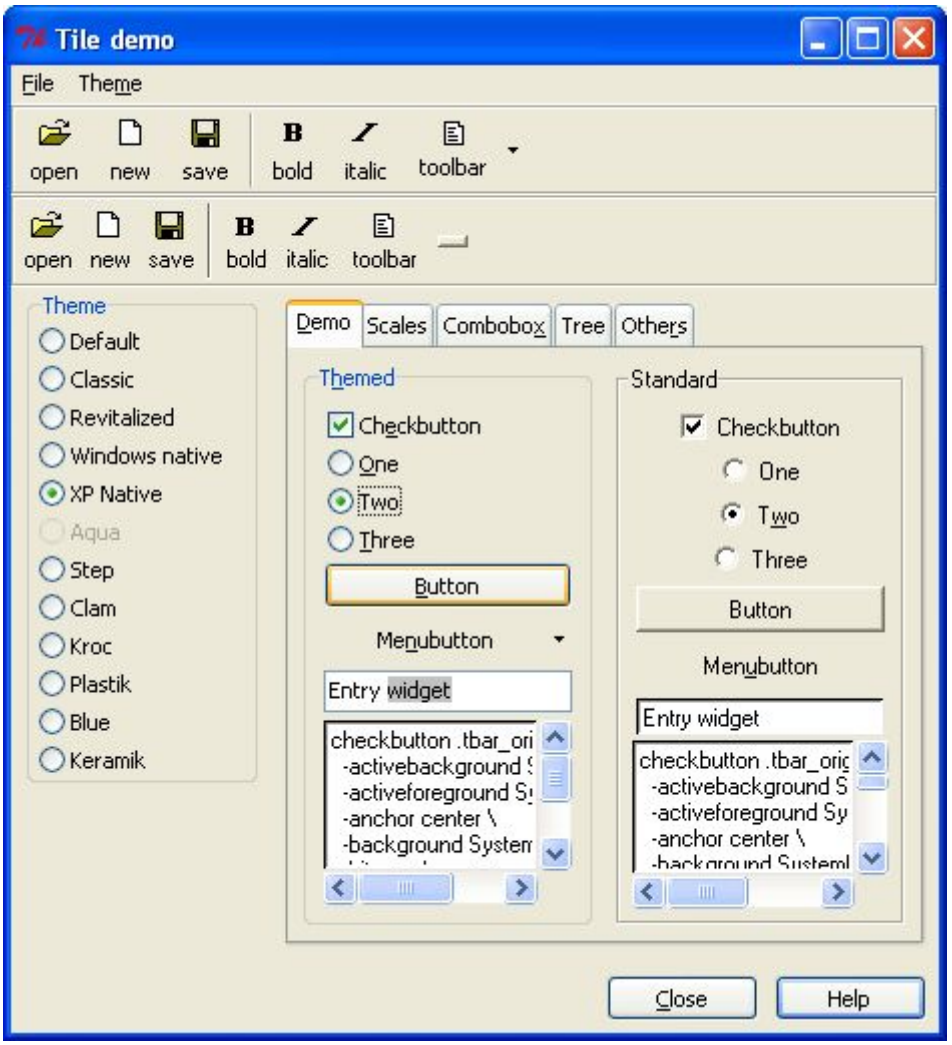
Desarrollo de aplicaciones de escritorio

Introducción

Python es siempre una buena elección para desarrollar aplicaciones de escritorio. Ofrece una gran cantidad de librerías para diseñar y desplegar interfaces de usuario rápidamente. Entre ellas se encuentran Tcl/Tk, GTK+, Qt y wxWidgets. Estas son librerías gráficas de código abierto y multiplataforma desarrolladas en C y C++, pero como Python permite escribir extensiones en dichos lenguajes y luego importarlas como módulos convencionales, utilizarlas es realmente muy fácil.

En este curso estaremos trabajando con Tcl/Tk. Es una librería mucho más pequeña en comparación a los otros gigantes, pero suficiente para tener una aplicación de escritorio funcional en pocos minutos y perfectamente preparada para aplicaciones comerciales. Además, accedemos a ella a través del módulo estándar `tkinter`, de modo que no requiere instalaciones adicionales.

No estaremos utilizando ninguna herramienta para diseñar nuestras interfaces. Es importante aprender a desarrollar aplicaciones gráficas “manualmente” (esto es, escribiendo el código directamente). Además, los pocos programas que cumplen esa función están descontinuados o lejos de proveer una solución confiable. Por otro lado, en el caso de librerías más grandes como GTK+ o Qt, hacer uso de dichas herramientas (como Glade o QtDesigner) requiere de un conocimiento previo de su funcionamiento para no terminar convirtiéndose en un impedimento para el desarrollo de la aplicación.



En la imagen observamos una aplicación de muestra escrita en Python usando Tcl/Tk. Me interesa destacar los distintos *controles* que incluye (botones, cajas de texto, etiquetas, menúes, etc.), a los cuales también se conoce como *widgets*. ¡Incluso la ventana es un *widget*!

El módulo `tkinter` está estructurado siguiendo el paradigma de orientación a objetos. Si bien no hemos visto nada respecto de ello, basta con saber que el concepto central de este paradigma es el de *clase*, que se parece a una función pero la distinguimos por la convención de nombramiento que lleva (`CamelCase` en lugar de `lower_case`). Por ejemplo:

```
a = Clase()
```

En la terminología de este paradigma, en el código anterior `Clase` es propiamente el nombre de una clase y `a` es una *instancia* de `Clase`. Puesto que cada *widget* de Tcl/Tk es una clase, tendrán nombres que sigan esa convención, similar a las excepciones (`ValueError`, `TypeError`, etc.).

Estructura de una aplicación

Lo primordial es importar los módulos `tkinter` y `ttk`. Este último introduce algunos *widgets* nuevos a la colección de Tcl/Tk y optimiza otros ya existentes en el primero.

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

Por convención se acostumbra a importar `tkinter` con el nombre de `tk` (ese es efectivamente el propósito de la palabra reservada `as` en este contexto).

Luego creamos la ventana principal y le asignamos un título de la siguiente forma.

```
ventana_principal = tk.Tk()
ventana_principal.title("Mi primera aplicación")
ventana_principal.mainloop()
```

La función `mainloop()` ejecuta el bucle principal de la aplicación, aquel encargado de dibujar todos los controles y manejar los eventos como veremos más adelante. Nótese que esta función bloquea la ejecución del código hasta que la ventana principal se cierre.

Para ser un poco más precisos, vamos a configurar el tamaño de la ventana, indicando vía la función `config()` o `configure()` que debe tener un ancho de 300px y un alto de 200px.

```
ventana_principal.title("Mi primera aplicación")
ventana_principal.config(width=300, height=200)
```

Ahora bien, una vez configurada la ventana, dentro de ella vamos a colocar algunos controles vía la función `place()`, que toma cuatro argumentos: la posición (`x`, `y`, tomando el origen de coordenadas en el extremo superior izquierdo) y el tamaño (`width`, `height`). Por ejemplo, el siguiente código crea un botón con el texto “Hola mundo” y lo ubica en la posición 50, 10.

```
boton = ttk.Button(text="Hola mundo")
boton.place(x=50, y=10)
```

En este caso, `ttk.Button` acepta el parámetro `text` que indica el texto que mostrará el botón. Podemos usar la función `help()` en la consola interactiva para conocer los argumentos de cada control.

Cuando se omiten los valores `width` y `height` en una llamada a `place()` son inferidos de acuerdo a la longitud del texto.

Para asociar la presión del botón con una función de Python, usamos el parámetro `command`.

```
def boton_presionado():
   print("Presionado")

boton = ttk.Button(text="Hola mundo", command=boton_presionado)
boton.place(x=50, y=10)
```

Los argumentos pasados al crear un *widget* pueden ser luego adulterados vía la función `config()`. Por ejemplo, para cambiar el texto del botón en algún otro lugar del programa:

```
boton.config(text="Nuevo texto")
```

Hemos visto cómo crear una ventana, un botón y cómo asociarlo a una función de Python cuando es presionado. A continuación haremos un pequeño recorrido por el resto de los *widgets* principales que componen una interfaz de usuario: etiquetas, cajas de texto, listas, y más.

Etiqueta (`Label`)

Una etiqueta es un control cuya única funcionalidad es contener un texto o una imagen. Al igual que los botones, la propiedad que controla el texto es `text`.

```
label = ttk.Label(text="Hola mundo")
```

Para cargar una imagen, creamos una instancia de la clase `tk.PhotoImage` y luego la pasamos como el parámetro `image`.

```
imagen = tk.PhotoImage(file="imagen.gif")
label = ttk.Label(image=imagen)
```

Cajas de texto (`Entry` y `Text`)

Tk provee dos *widgets* para ingresar texto: `ttk.Entry` y `tk.Text`. El primero es generalmente útil para textos pequeños en una misma línea, soportando únicamente texto plano. El segundo, además de permitir texto de múltiples líneas, también puede albergar imágenes y texto con formato, aunque no la cubriremos en este curso.

```
entry = ttk.Entry()
```

Para obtener el texto que el usuario ha ingresado en el control utilizamos la función `get()`. Ejemplo:

```
entry = ttk.Entry()
entry.place(x=10, y=10)

def imprimir_texto():
    print(entry.get())
```

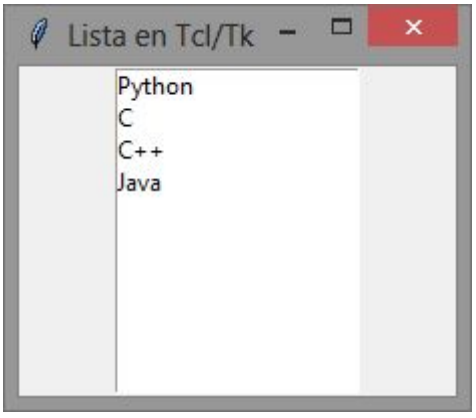
```
boton = ttk.Button(text="Imprimir texto", command=imprimir_texto)
boton.place(x=10, y=50)
```

Vía las funciones `insert()` y `delete()` podemos añadir y remover texto del control en una determinada posición:

```
# Inserta el texto al comienzo del control.
entry.insert(0, "Hola mundo")

# Borra los primeros 10 caracteres.
entry.delete(0, 10)

# Borra todo el texto.
entry.delete(0, tk.END)
```

Lista (`Listbox`)

La lista permite mostrar un conjunto de textos, como se muestra en la siguiente imagen.



Para implementar una lista creamos una instancia de la clase `tk.Listbox` e insertamos elementos vía `insert()`.

```
lista = tk.Listbox()
lista.insert(0, "Python", "C", "C++", "Java")
lista.place(x=10, y=10)
```

El primer argumento de `insert()` es la posición en la que se debe insertar un elemento, empezando desde el 0, o bien la constante `tk.END` que indica el final de la lista.

```
lista.insert(tk.END, "Último elemento")
```

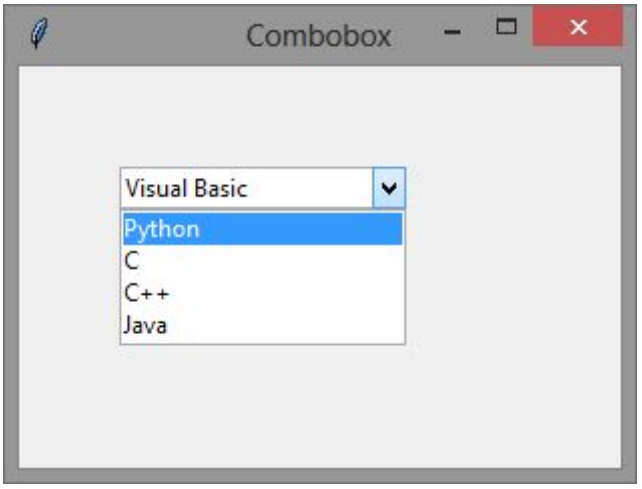
La función `get()` retorna el elemento en una posición determinada, mientras que `curselection()` retorna la posición del elemento seleccionado. Por ende, el siguiente código imprime en pantalla, al presionar el botón, el elemento seleccionado.

```
lista = tk.Listbox()
lista.insert(0, "Python", "C", "C++", "Java")
lista.place(x=10, y=10, height=100)

def imprimir_seleccion():
    print(lista.get(lista.curselection()))

boton = ttk.Button(text="Imprimir seleccion", command=imprimir_seleccion)
boton.place(x=10, y=130)
```

Para leer más sobre otros métodos de las listas véase este enlace.

Lista desplegable (Combobox)

La lista desplegable es una combinación entre una lista y una caja de texto (`Entry`). Permite seleccionar un elemento entre varios o bien ingresar manualmente un texto.



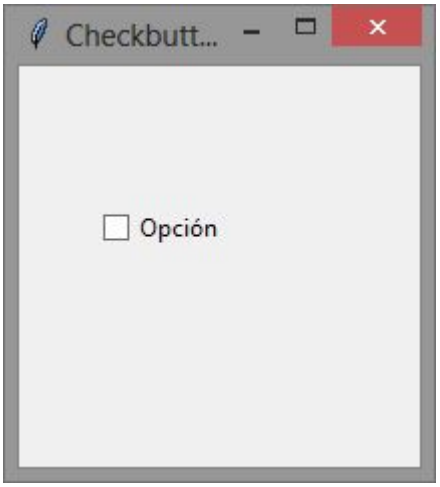
Se implementa creando una instancia de `ttk.Combobox` e indicando los elementos en una lista a través del parámetro `values`.

```
lista_desplegable = ttk.Combobox(
   values=[
        "Visual Basic",
        "Python",
        "C",
        "C++",
        "Java"
   ]
)
lista_desplegable.place(x=10, y=10)
```

Para leer más sobre las listas desplegables véase este enlace.

Casilla de verificación (*Checkbox* / `Checkbutton`)

La casilla de verificación es un control que permite al usuario dirimir entre dos opciones opuestas: sí/no, activado/desactivado, encendido/apagado, etc.



Se implementa a través de la clase `ttk.Checkbutton`.

```
checkbutton = ttk.Checkbutton(text="Opción")
checkbutton.place(x=10, y=10)
```

Para establecer u obtener el estado de una casilla de verificación es necesario enlazarla con una suerte de variable que provee Tk a través de la clase `tk.BooleanVar`.

```
checkbutton_estado = tk.BooleanVar()
checkbutton = ttk.Checkbutton(text="Opción", variable=checkbutton_estado)
```

Hecho esto, obtenemos el valor de la casilla (`True`/`False`) vía `checkbutton_estado.get()` y lo establecemos vía `checkbutton_estado.set()`.

Para una explicación completa sobre este control véase este enlace.

Barra de menús

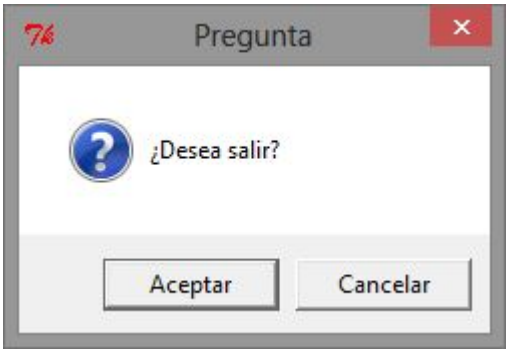
Para incluir un menú en nuestra ventana, primero tenemos que crear una barra de menús a través de la clase `tk.Menu`, la cual es enlazada a la ventana principal vía `config(menu=...)`. Luego, individualmente creamos cada uno de los menús propiamente dichos, también vía la clase `tk.Menu`, los cuales son enlazados a la barra con la función `add_cascade()`. Cada una de las opciones de un menú es añadida vía `add_command()`.

Por ejemplo, el siguiente código crea una barra de menús con los menús “Archivo” y “Ayuda”, dentro de los cuales se encuentran las opciones “Nuevo” y “Acerca de”, respectivamente, que están asociadas con una función de Python que es invocada cuando son presionados.

```
import tkinter as tk
from tkinter import ttk

def nuevo():
    print("Nuevo archivo.")

def acerca_de():
    print("Acerca de:")

ventana_principal = tk.Tk()
ventana_principal.title("Mi primera aplicación")
barra_de_menu = tk.Menu()

menu_archivo = tk.Menu(barra_de_menu, tearoff=0)
menu_archivo.add_command(label="Nuevo", command=nuevo)

menu_ayuda = tk.Menu(barra_de_menu, tearoff=0)
menu_ayuda.add_command(label="Acerca de...", command=acerca_de)

barra_de_menu.add_cascade(label="Archivo", menu=menu_archivo)
barra_de_menu.add_cascade(label="Ayuda", menu=menu_ayuda)

ventana_principal.config(width=300, height=200, menu=barra_de_menu)
ventana_principal.mainloop()
```

Cuadros de diálogo

Los cuadros de diálogo presentan un mensaje al usuario y opcionalmente le solicitan tomar una decisión: sí/no, aceptar/cancelar, reintentar/abortar, etc.



Tk provee el módulo `tkinter.messagebox` con un conjunto de funciones para crear cuadros de diálogo de forma rápida que retornan el valor seleccionado por el usuario.

```
from tkinter import messagebox
```

Las funciones principales son:

```
# Siempre retornan la cadena "ok".
messagebox.showinfo(
      title="Información", message="Hola mundo")
messagebox.showwarning(
      title="Advertencia", message="Hola mundo")
messagebox.showerror(
      title="¡Error!", message="Hola mundo")

# Retornan True o False.
messagebox.askokcancel(
      title="Pregunta", message="¿Desea salir?")
messagebox.askyesno(
      title="Pregunta", message="¿Desea salir?")
messagebox.askretrycancel(
      title="Operación fallida", message="¿Qué desea hacer?")
```

Otros elementos

- Vista de árbol (Treeview), también utilizado para crear listas con columnas
- Barra de progreso
- Panel de pestañas

Organizando los controles

En este curso hicimos uso únicamente de una forma de posicionar los *widgets* dentro de una ventana o de un control padre: indicando su posición X e Y vía la función `place()`. Existen otros dos métodos para organizar una interfaz en Tcl/Tk: posicionamiento relativo, vía la función `pack()`, y el modelo de grilla, vía `grid()`.

Si bien la elección de un modelo u otro puede resultar trivial en interfaces pequeñas, a medida que una aplicación de escritorio crece se vuelve fundamental que se encuentre bien organizada. Por lo general para aplicaciones medianas y grandes querrás optar por la organización en grilla.

Si te interesa profundizar en el tema, te recomiendo este artículo que explica de forma didáctica los tres modelos de organización, con ejemplos e imágenes.

Distribuir aplicaciones como ejecutables

Habíamos dicho que Python es un lenguaje interpretado, de modo que no se compila a un código que es ejecutado directamente por el procesador, sino que requiere, justamente, de un intérprete. Ahora bien, hemos desarrollado nuestra aplicación de escritorio y queremos distribuirla. ¿Debemos pretender que todos los usuarios que la utilicen tengan Python instalado? Eso sería una experiencia poco amigable.

Para solucionar este problema se hace uso de alguna herramienta que convierte nuestro programa de Python en un archivo ejecutable (.exe en Windows), de modo que el intérprete de Python no tenga que estar necesariamente instalado. Entre ellas se encuentran py2exe, PyInstaller y cx_Freeze. En este curso estaremos empleando la segunda, ya que es sencilla de utilizar.

Podemos instalar PyInstaller vía pip:

```
python -m pip install pyinstaller
```

Su utilización es muy sencilla. Abrimos una terminal en la carpeta en donde se encuentre el *script* que queramos distribuir y ejecutamos:

```
pyinstaller miscript.py
```

Luego de unos cuantos mensajes en la consola, se habrán creado dos carpetas y un archivo: `build`, `dist` y `miscript.spec`, respectivamente. Dentro de `dist` encontrarás el archivo ejecutable junto con sus dependencias listo para ser distribuido y utilizado sin necesidad de tener un intérprete de Python.

Si queremos aún más portabilidad haciendo que PyInstaller genere únicamente un archivo ejecutable sin dependencias, podemos usar la opción `--onefile`:

```
pyinstaller --onefile miscript.py
```

De forma similar, `--noconsole` indica que la consola no debe mostrarse (ideal para aplicaciones de escritorio):

```
pyinstaller --noconsole miscript.py
```

¡Eso es todo! Encontrarás más ejemplos y la documentación completa en la página oficial de PyInstaller (pyinstaller.org).