Fundamentos de POO para interfaces gráficas

Dentro de la programación existe una especialidad enfocada al desarrollo de programas utilizando interfaces gráficas de usuario, también conocidas como Graphic User Interfaces o GUI.

En ella se dan cita algunos conceptos importantes de la programación, en especial de la programación orientada a objetos.

Aquí trataremos esos conocimientos además de dar los primeros pasos en tres bibliotecas gráficas para Python: Tkinter, WxPython y PySide, también conocida como Qt for Python.

Con ellas crearemos un primer programa para aprender a aplicar los conceptos básicos.

Bucles infinitos

Anteriormente ya creamos algunos scripts o guiones de código que se ejecutan de arriba abajo y finalizan con la última instrucción.

Ahora bien, si os lo paráis a pensar los programas raramente funcionan de esa forma, no se cierran solos sino que normalmente se mantienen abiertos hasta que nosotros los cerramos.

Para poder mantenerse en marcha, el código de los programas funciona dentro de lo que se conoce como "bucles infinitos" para ejecutar sus instrucciones permanentemente.

En Python un bucle infinito se puede crear con la instrucción **while** para repetir un bloque de código mientras se cumple una condición.

Dejadme enseñaros como crear un cronómetro en Python porque es un ejemplo muy ilustrativo:

```
cronometro.py
import time

segundero = 0

while True:
    time.sleep(1)    # esperamos un segundo
    segundero += 1   # segundero = segundero + 1
    print(segundero)
```

El problema de los bucles infinitos es que nunca dejan de ejecutarse, para finalizarlos hay que cerrar el proceso forzosamente con Control+C.

La forma de finalizar un bucle infinito es romper su ejecución desde dentro, algo que en Python se puede hacer condicionando el código y utilizando la instrucción **break**:

```
import time

segundero = 0
maximo = 5

while True:
    time.sleep(1)    # esperamos un segundo
    segundero += 1   # segundero = segundero + 1
    print(segundero)

    if segundero == maximo:
        print("Se ha llegado al límite, rompiendo el bucle infinito")
        break
```

En resumen, mientras que un script es una serie de instrucciones que se ejecutan y finalizan de forma natural, un programa son unas instrucciones que se repiten de forma indefinida hasta que se acciona en su interior el mecanismo que las finaliza.

Sin los bucles infinitos no tendríamos programas gráficos, ni servicios web y mucho menos videojuegos, por eso son tan importantes.

Menú interactivo

Quiero consolidar este concepto del bucle infinito con un ejemplo interactivo, así que vamos a hacer un programa muy sencillo para la terminal.

En Python hay una instrucción que permite leer una cadena de caracteres por teclado a través del prompt y asignarla a una variable, se llama **input**:

```
menu.py
nombre = input("Introduce tu nombre\n> ")
print("Hola", nombre)
```

Usando esta función en conjunto con un bucle infinito y unas condiciones, podemos crear un programa con un menú interactivo que haga varias cosas, incluyendo una opción para romper el bucle:

```
import time

while True:
    print("[1] Mostrar la hora\t[2] Mostrar la fecha\t[3] Salir")
    opcion = input("> ")

    if opcion == "1":
        print(time.strftime("%H:%M:%S"))

    if opcion == "2":
        print(time.strftime("%d/%m/%Y"))

    if opcion == "3":
        break
```

¿Queda más clara la idea no? Este menú se ejecuta indefinidamente pero como hemos incluido la opción 3 para romper el bucle podemos salir del programa en cualquier momento.

Pues los programas gráficos funcionan de forma similar.

Modularización del código

En las lecciones anteriores hemos visto que Python incluye un módulo llamado **time** para manipular la información del tiempo del sistema, pero no es el único.

Hay muchos otros módulos internos para realizar todo tipo de tareas y también contamos con un repositorio público llamado Python Package Index (PyPI.org) donde hay decenas de miles de módulos externos creados por la comunidad.

Un módulo es un fichero con definiciones de código que se puede importar en otros scripts y programas para utilizarlas. Si un programa cuenta con varios módulos que trabajan en conjunto para resolver un problema, a este conjunto de módulos se le denomina paquete, pero este concepto no lo trataré en el curso.

Por ejemplo supongamos que tenemos un módulo llamado **operaciones.py** con varias funciones para realizar operaciones aritméticas como sumas, restas y multiplicaciones.

```
def suma(a, b):
    return a+b

def resta(a, b):
    return a-b

def producto(a, b):
    return a*b


# Funciones para probar el módulo
print("La suma de 5 y 10 es", suma(5, 10))
print("La resta de -9 y 8 es", resta(-9, 8))
print("El producto de 3 y -6 es", producto(3, -6))
```

Ahora supongamos que necesitamos usar alguna de estas operaciones en otro script llamado **programa.py**:

```
import operaciones

print("4*6=", operaciones.producto(4, 6))
```

Al ejecutar el programa veremos que efectivamente se llama la función **producto**, pero antes se han ejecutado las pruebas del módulo, aquellas instrucciones que pusimos abajo.

Esto sucede porque **import** a parte de cargar las definiciones del módulo también ejecuta todas las instrucciones sueltas del fichero.

Para evitar este comportamiento hay que definir una condición en los módulos que ejecute sólo las instrucciones sueltas cuando se ejecuta específicamente el propio fichero. Eso se hace así:

```
# Funciones para probar el módulo
if __name__ == "__main__":
    print("La suma de 5 y 11 es", suma(5, 11))
    print("La resta de -9 y 8 es", resta(-9, 8))
    print("El producto de -3 y -6 es", producto(-3, -6))
```

La variable especial **__name__** siempre contiene el nombre del propio fichero menos cuando concuerda con el script que ejecuta el programa, en ese caso su valor cambia a **__main__**, que en contexto indica que es el fichero "principal" del programa.

En otras palabras, al hacer:

```
python programa.py
```

El **__name__** de **operaciones.py** es "operaciones" y no se ejecutan las instrucciones sueltas, pero si ejecutamos directamente:

```
python operaciones.py
```

Su **__name__** cambia a **__main__** y entonces sí se ejecutan.

Dejando esto de banda, también es posible especificar qué definiciones queremos importar:

```
from operaciones import producto, suma

print("4*6=", producto(4, 6))
print("5+3=", suma(5, 3))
```

Usando la importación específica no tenemos que llamar las funciones con el módulo delante, pero por contra no podremos usar las definiciones que no importamos, como resta.

Sea como sea con lo que hemos aprendido queda clara la idea de que los módulos nos permiten organizar el código de nuestros proyectos en diferentes ficheros para hacerlos más fáciles de mantener.

Como las bibliotecas gráficas están repletas de componentes y definiciones, conocer como trabajar con los módulos es un concepto esencial para utilizarlas.

Clases y objetos

Para entender el desarrollo de interfaces gráficas es obligatorio conocer el paradigma de la programación orientada a objetos.

Este concepto se basa en la idea de simular los datos de la vida real en unas estructuras independientes y reutilizables. Los objetos tienen sus propias variables y funciones internas para manipular su contenido, conocidas como atributos y métodos respectivamente.

Python incluye un constructor base para crear objetos, se trata de la clase **object**:

```
objetos.py
objeto = object()
print(type(objeto).__name__)
```

La clase **object** por sí misma no provee ninguna funcionalidad relevante pero es la base de todas la clases en Python.

¿Pero a todo esto qué es una clase?

Pues una clase es una definición que explica como crear un nuevo tipo de dato. Esta idea se entiende mejor con el ejemplo del molde y las galletas.

Una clase es un molde para hacer galletas y las galletas son los objetos que se crean con el molde. Mirad, vamos a crear una nueva clase en Python llamada **Molde**, no tendrá nada todavía así que usaremos la sentencia **pass** para dejarla en blanco:

```
class Molde:
    pass
```

Ahora podemos crear un objeto galleta a partir de la definición de la clase Molde:

```
galleta = Molde()
print(type(galleta).__name__)
```

Al consultar el tipo de nuestra galleta vemos que se devuelve "Molde", eso es porque se ha creado siguiendo las instrucciones de la clase Molde. ¿Pero esto no tiene mucha lógica verdad? Es por eso que normalmente las clases tienen el nombre del objeto que definen, así que en lugar de Molde vamos a llamar a la clase Galleta:

```
class Galleta:
    pass


galleta = Galleta()
print(type(galleta).__name__)
```

¿Váis captando la idea? Una clase es la definición de un tipo de dato, mientras que un objeto es el resultado de crear ese dato siguiendo las instrucciones de la clase.

Los números, las cadenas, las listas... en Python todo son clases que definen estructuras de datos.

Hemos estado creando objetos todo el tiempo sin darnos cuenta y la única diferencia respecto a nuestra clase Galleta, es que ésta la hemos creado nosotros y podemos extenderla para que realice todo lo que nosotros queramos.

Instancias de clase

Cada vez que asignamos datos a una variable se está reservando un espacio en la memoria de nuestro ordenador para almacenar ahí su información.

A este proceso de reservar el espacio en la memoria se le conoce como instanciación y podemos comprobar la dirección de memoria de cualquier objeto haciendo uso de un par de funciones de Python:

```
objetos.py
numero = 10
cadena = "Hola mundo"
lista = [3, 5, 6]

print(hex(id(numero)))
print(hex(id(cadena)))
print(hex(id(lista)))
```

Evidentemente ocurre lo mismo con los objetos creados con nuestras propias clases:

```
class Galleta:
    pass


galleta = Galleta()
print(hex(id(galleta)))
```

Incluso por muy extraño que parezca, la propia definición de la clase existe en la memoria:

```
print(hex(id(Galleta)))
```

Y es que si no existieran esas instrucciones no podríamos utilizarlas para crear nuestros objetos, es incluso lógico.

Lo interesante en este punto es que todos los objetos creados a partir de la clase Galleta comparten la misma estructura. Eso no se aprecia porque no hemos definido nada dentro, pero mirad qué ocurre si añadimos un atributo a la clase:

```
class Galleta:
    sabor = "dulce"


galleta_dulce = Galleta()
print(galleta_dulce.sabor)

galleta_salada = Galleta()
print(galleta_salada.sabor)
```

```
galleta_picante = Galleta()
print(galleta_picante.sabor)
```

Todas nuestras galletas por defecto son dulces, pero eso podemos cambiarlo fácilmente redefiniendo su sabor:

```
galleta_salada = Galleta()
galleta_salada.sabor = "salada"
print(galleta_salada.sabor)

galleta_picante = Galleta()
galleta_picante.sabor = "picante"
print(galleta_picante.sabor)
```

¿Interesante verdad? Como cada galleta se almacena en una instancia diferente de la memoria, sus atributos son independientes de las otras y pueden almacenar diferentes valores.

En conclusión, una instancia es un espacio en la memoria donde se almacenan los datos individuales de un objeto tal como define su clase.

Métodos de clase

Más allá de los atributos, una clase también puede definir métodos. Los métodos son funciones que encontramos en los objetos para interactuar con ellos.

Hay un ejemplo que me encanta para ilustrar esto. Supongamos que nuestra galleta tiene un atributo **chocolate** que es falso por defecto, indicando que la galleta no tiene chocolate:

```
objetos.py
class Galleta:
    sabor = "dulce"
    chocolate = False
```

Ahora si quisiéramos chocolatear nuestras galletas sin modificar directamente el atributo chocolate, lo que podemos hacer es crear un método **chocolatear**:

```
class Galleta:
    sabor = "dulce"
    chocolate = False

    def chocolatear():
        chocolate = True

galleta = Galleta()
galleta.chocolatear()
print(galleta.chocolate)
```

Esta es la lógica básica, sin embargo tendremos un problema al ejecutarlo, Python nos devolverá un error indicando:

```
TypeError: chocolatear() takes 0 positional arguments but 1 was given
```

Se está quejando de que el método **chocolatear** requiere definir por lo menos un argumento, ¿cuál es la razón? Ahora mismo lo vais a entender.

Haced esto:

```
class Galleta:
    sabor = "dulce"
    chocolate = False

    def chocolatear(argumento_misterioso):
        chocolate = True

galleta = Galleta()
galleta.chocolatear()
print(galleta.chocolate)
```

Al hacer este cambio veremos que el código se ejecuta, sin embargo el atributo **chocolate** no se ha modificado al llamar al método chocolate.

Esto se debe a que el atributo **chocolate** que hay definido fuera del método y el que hay dentro no son el mismo, forman parte de ámbitos diferentes. El primero pertenece a la clase y el otro al método, algo que podemos constatar consultando sus posiciones en la memoria:

```
class Galleta:
    sabor = "dulce"
    chocolate = False

    def chocolatear(argumento_misterioso):
        chocolate = True
        print("Posición en la memoria de 'chocolate' en chocolatear -
>",
              hex(id(chocolate)))


galleta = Galleta()
galleta.chocolatear()
# print(galleta.chocolate)
print("Posición en la memoria de 'chocolate' en la galleta ->",
      hex(id(galleta.chocolate)))
```

Como véis se almacenan en lugares diferentes y eso significa que no son lo mismo.

¿Entonces como vamos a interactuar con los atributos de nuestras instancias? Tranquilos, aquí entra en juego ese argumento misterioso. ¿Qué debe de ser? Vamos a mostrarlo:

```
class Galleta:
    sabor = "dulce"
    chocolate = False

    def chocolatear(argumento_misterioso):
        chocolate = True
        print("El valor del argumento misterioso es ->",
argumento_misterioso)
```

```
galleta = Galleta()
galleta.chocolatear()
# print(galleta.chocolate)
```

Y nos devolverá:

```
El valor del argumento misterioso es <__main__.Galleta object at
0x01170830>
```

¡Un objeto galleta! ¿Podría ser la propia instancia del objeto galleta? Vamos a comparar la posición de sus instancias en la memoria:

```
class Galleta:
    sabor = "dulce"
    chocolate = False

    def chocolatear(argumento_misterioso):
        chocolate = True
        print("Instancia en la memoria del argumento misterioso ->",
              hex(id(argumento_misterioso)))


galleta = Galleta()
galleta.chocolatear()
# print(galleta.chocolate)
print("Instancia en la memoria de la galleta ->", hex(id(galleta)))
```

Pues efectivamente, este argumento misterioso es en realidad la propia instancia de la galleta, la posición en la memoria donde se ha creado y se guardan sus datos. En otras palabras, si hacemos referencia a él debería ser lo mismo que hacer referencia a la galleta, así que podríamos asignar su atributo chocolate:

```
class Galleta:
    sabor = "dulce"
    chocolate = False

    def chocolatear(instancia):     # <-- cambiar nombre a instancia
        instancia.chocolate = True  # <-- cambiar nombre a instancia


galleta = Galleta()
galleta.chocolatear()
print(galleta.chocolate)
```

¡Por fin hemos llegado al kit de la cuestión! Ese argumento que necesitan los métodos es una referencia a la propia instancia para poder acceder a sus atributos y métodos internamente.

¿Y sabéis qué? Por conveniencia a este argumento deberíamos llamarlo **self** haciendo referencia a "la instancia en sí misma". Esta es la razón por la cuál VSC se lleva quejando un montón de rato, así que vamos a escribirlo bien:

```
def chocolatear(self):
    self.chocolate = True
```

En resumen, para manipular los atributos de los objetos desde sus métodos internos es necesario tener la referencia a sus instancias en la memoria y esa es la razón por la que se envía automáticamente como primer argumento.

Método constructor

Una peculiaridad interesante que tienen los objetos, es que al crearse pueden recibir unos valores iniciales para establecerlos en sus posibles atributos internos. Esto es gracias a un método especial de las clases llamado constructor:

```
objetos.py
class Galleta:
    sabor = "dulce"
    chocolate = False

    def __init__(self):
        print("Soy una galletita recién horneada")

galleta = Galleta()
```

Si quisiéramos establecer los valores iniciales de los atributos **sabor** y **chocolate** sería tan sencillo como definirlos como parámetros y enviarlos durante la instanciación:

```
class Galleta:
    sabor = "dulce"
    chocolate = False

    def __init__(self, sabor, chocolate):
        self.sabor = sabor
        self.chocolate = chocolate
        print("Soy una galletita recién horneada")

    def chocolatear(self):
        self.chocolate = True


# galleta = Galleta("Salada", False)
galleta = Galleta(sabor="Salada", chocolate=False)
print(galleta.sabor)
print(galleta.chocolate)
```

Si utilizamos este método tendremos que enviar siempre los dos valores u ocurrirá un error:

```
galleta = Galleta()
```

Podemos solucionarlo añadiendo un valor por defecto igual que ocurre con los parámetos de las funciones:

```
def __init__(self, sabor="dulce", chocolate=False):
```

Sin embargo al hacer esto los atributos de clase pierden su sentido, por eso no casi nunca se suelen utilizar:

```
class Galleta:

    def __init__(self, sabor="dulce", chocolate=False):
        self.sabor = sabor
        self.chocolate = chocolate
        print("Soy una galletita recién horneada")

    def chocolatear(self):
        self.chocolate = True


galleta = Galleta()
print(galleta.sabor)
print(galleta.chocolate)
```

Método strings

Otro método especial que vale la pena comentar es **__str__**, sirve para dar una representación textual a un objeto en forma de cadena.

En nuestro caso nos sería muy útil para mostrar un resumen de la galleta al imprimirla por pantalla con **print()**:

```
class Galleta:

    def __init__(self, sabor="dulce", chocolate=False):
        self.sabor = sabor
        self.chocolate = chocolate

    def __str__(self):
        resumen = "Soy una galletita " + self.sabor
        if self.chocolate:
            resumen += " con chocolate"
        return resumen  # debe devolver una cadena

    def chocolatear(self):
        self.chocolate = True


galleta = Galleta()
print(galleta)
```

Como vemos **print()** ejecuta implícitamente el metodo string del objeto galleta.

Este valor también podemos conseguirlo llamándolo directamente o utilizando la función **str()**:

```
galleta = Galleta()
print(galleta)

resumen_galleta = galleta.__str__()
print(resumen_galleta)

resumen_galleta = str(galleta)
print(resumen_galleta)
```

Bastante útil como véis.

Herencia y mixins

La herencia de clases es la capacidad que tiene una clase, llamada subclase, de heredar los atributos y métodos de una o varias clases, llamadas superclases.

Muchas veces para explicar este concepto he usado de ejemplo una clases Vehículo que sirve como base de dos clases Coche y Camión. O también una clase Animal como base para dos clases Gato y Perro. Todos los vehículo tienen ruedas, todos los animales tienen patas, etc. Se trata de encontar atributos comunes en las clases padre y hacer que las clases hijas los hereden, bla, bla, bla.

Pero sinceramente, en todos los años que llevo programando, que son como 12 o 13, sólo he diseñado un sistema alrededor de este concepto y ni siquiera salió bien. No me malinterpretéis, no quiero decir que la herencia sea una tontería, sino que raramente es aplicable para resolver problemas del mundo real. Donde sale a relucir su verdadero potencial es en la ingeniería de software, cuando se utiliza para crear bibliotecas que tienen como objetivo desarrollar más software, como por ejemplo los frameworks web, los motores de videojuegos y las interfaces gráficas, que son entornos puramente virtuales.

Fuera de estos ámbitos sólo hay una razón por la que la herencia me parece útil, y esa es es para extender las funcionalidades de las clases heredando de otras clases. Estas clases se conocen como **Mixins** y tienen la peculiaridad de ofrecer una funcionalidad pero por si mismas no sirven para nada.

Imaginad que tenemos una clase A y otra clase B sin nada en especial:

```
mixins.py
class A:
    pass


class B:
    pass
```

Nuestro objetivo es implementar en ellas un nuevo método llamado **instancia()** que muestre por pantalla su posición en la memoria.

Usando un mixin esto sería muy fácil de solucionar, simplemente crearemos una clase que haga eso mismo y haremos que A y B hereden de ella para extender su comportamiento:

```
class MixinInstancia:
    def instancia(self):
        print(hex(id(self)))

class A(MixinInstancia):
    pass


class B(MixinInstancia):
    pass
```

```
a = A()
a.instancia()

b = B()
b.instancia()
```

Pero no nos quedemos aquí. Imaginad que necesitamos otra función común para mostrar por pantalla el nombre de la clase, pues podríamos declarar otro mixin y heredar de él, dando como lugar al concepto de herencia múltiple:

```
class MixinInstancia:
    def instancia(self):
        print(hex(id(self)))


class MixinClase:
    def clase(self):
        print(type(self).__name__)


class A(MixinInstancia, MixinClase):
    pass


class B(MixinInstancia, MixinClase):
    pass


a = A()
a.instancia()
a.clase()

b = B()
b.instancia()
b.clase()
```

Como véis usando mixins podemos extender nuestras clases de una forma flexible, sin obligar a que una clase tengan todas las funcionalidades de otra por el hecho de heredar de ella, algo que puede ocasionar problemas trabajando con herencia múltiple.

Hablaremos de ello en la siguiente lección.

Herencia múltiple

Cuando heredamos de varias clases y éstas no son Mixins, la herencia múltiple puede volverse en nuestra contra, ya que las subclases heredan todo el contenido de las superclases. ¿Pero es eso cierto? ¿Lo heredan todo? Pues no, en el caso de que dos o más superclases tengan el mismo método, solo uno de los métodos prevalecerá.

Esto se puede ilustrar con un ejemplo:

```
herencia_multiple.py
class A():
    def hola(self):
```

```
        print("Hola heredado de la clase A")

class B():
    def hola(self):
        print("Hola heredado de la clase B")

class C(A,B):
    pass

c = C()
c.hola()
```

Como véis la superclase dominante es la A, y la única razón para que sea así es que se encuentra más a la izquierda al heredarla. Sí, la posición es lo que determina la prioridad en la herencia cuando tenemos el mismo método.

Esto que a priori parece sin importancia puede ser un problema, porque imaginad el caso excepcional donde tenemos dos métodos solapados y queremos heredar uno de una clase específica, no podemos porque la prioridad lo destruye:

```
class A():
    def hola(self):
        print("Hola heredado de la clase A")

    def adios(self):
        print("Adiós heredado de la clase A")

class B():
    def hola(self):
        print("Hola heredado de la clase B")

    def adios(self):
        print("Adiós heredado de la clase B")

class C(A,B):
    pass

c = C()
c.adios()  # No podemos heredar de B aunque queramos
```

Por suerte hay una forma muy fácil de solucionar el problema. Esa es sobreescribir el método en la subclase y hacer la llamada manualmente al método de la superclase que necesitamos:

```
class C(A,B):

    def adios(self):
        B.adios(self)
```

Extendiendo métodos

En la lección anterior vimos que es posible llamar manualmente un método de una superclase, pero esa no es su única utilidad, también se utiliza para extender un método sin perder la funcionalidad de la clase heredada:

```
herencia_multiple.py
class A():
```

```
    def hola(self):
        print("Hola heredado de la clase A")


class B(A):

    def hola(self):
        print("Hola de la propia clase B")
        A.hola(self)  # <-- aquí la magia


b = B()
b.hola()
```

Python 3 provee un nivel de abstracción para no tener que hacer referencia explícitamente al nombre de la superclase usando la función **super()** para substituir la superclase inmediata:

```
def hola(self):
    print("Hola de la propia clase B")
    super().hola()
```

Esto también se puede escribir con la sintaxis que indica explícitamente buscar la superclase de B, que es A, de la siguiente manera:

```
def hola(self):
    print("Hola de la propia clase B")
    super(B, self).hola()
```

La primera forma es más simple pero en mucha documentación lo veréis escrito de la segunda, así que tenedlo en cuenta.

Con todo lo que hemos visto ya deberíais tener una base suficiente sobre la programación orientada a objetos. No os preocupéis si algún concepto no os ha quedado del todo claro, cuando los veáis en la práctica os iréis familiarizando con ellos y siempre estáis a tiempo de volver a estos apuntes si necesitáis refrescar la memoria.

Y ahora sí, ha llegado el momento de empezar con lo bueno.

Primer programa con Tkinter

De las tres bibliotecas que utilizaremos en este curso, Tkinter es la única que en principio viene preinstalada, es por decirlo de alguna forma la biblioteca oficial de Python para crear interfaces gráficas.

Si bien es bastante simple y fácil de empezar a utilizar, por contra ofrece pocos elementos gráficos y un control limitado sobre el comportamiento de la interfaz. Además su principal es que en realidad se trata de un puente para usar otra biblioteca gráfica llamada Tcl/Tk en Python. Eso hace que carezca en sí misma de una documentación en condiciones y si quieres profundizar en sus componentes tienes que acabar consultando la documentación oficial de Tcl/Tk, cuya presentación digamos que no es muy amigable.

En fin, vamos a crear nuestro programa de ejemplo a ver qué podemos aprender de él:

```
hola_tk.py
# Todos los componentes se encuentran en este módulo
import tkinter as tk

# Tk es la clase que crea el componente raíz o ventana principal
app = tk.Tk()
# Su método mainloop() inicia el bucle infinito del programa
app.mainloop()
```

El componente raíz es una ventana en sí misma, pero no tiene ningún contenido. Si queremos añadir algo dentro tendremos que crear otros componentes y añadirlos dentro, por ejemplo un botón:

```
# Al instanciarlo el primer argumento indica el contenedor
button = tk.Button(app, text="Hola")
```

Sin embargo en **tkinter** no es suficiente con crear un componente y añadirlo al contenedor, tenemos que posicionarlo:

```
# Podemos usar el método pack para empaquetarlo en su contenedor
button.pack()
```

Como veis ya aparece nuestro botón, pero curiosamente la ventana se ha estrechado. Eso es porque el componente raíz no tiene tamaño propio.

Para añadirle un tamaño por defecto podemos usar el método geometry:

```
# Esto le dará 100px de ancho por 50 de alto
app.geometry("100x50")
```

Ahora nos falta añadirle la funcionalidad al botón para que al presionarlo se muestre un mensaje en la terminal. Para hacer necesitaremos enlazar una función como comando del botón y tendremos que definirla antes de enlazarla porque sino no se encontrará su referencia:

```
def hola():
    print("¡Hola mundo!")

# ...

button = tk.Button(app, text="Hola", command=hola)
```

Con esto ya tenemos nuestro programa "Hola Mundo" con Tkinter, sin embargo se trata de una estructura poco reutilizable y tener que definir las funciones antes no es muy práctico.

Podemos solucionar estos problemas adaptando el ejemplo para que use programación orientada a objetos bajo nuestra propia lógica:

```
import tkinter as tk
```

```
class MainWindow:
    # Le pasamos el componente raíz al constructor
    def __init__(self, root):
        # Establecemos el tamaño de la raíz
        root.geometry("100x50")
        # Añadimos el botón y lo empaquetamos
        button = tk.Button(root, text="Hola", command=self.hola)
        button.pack()

    # Definimos la función como un método de clase
    def hola(self):
        print("¡Hola mundo!")


# Creamos la aplicación, la ventana e iniciamos el bucle
app = tk.Tk()
window = MainWindow(app)
app.mainloop()
```

Con esto ya lo tenemos.

¿Qué os ha parecido? No os sintáis abrumados por tantos nuevos conceptos, hacedme caso.

Lo importante es que entendáis el flujo del programa. ¿Sabríais determinar cuales son las clases que se están usando? ¿Los métodos y atributos?

Si es así no temáis nada, la mayoría de las veces la programación no se trata de memorizar instrucciones, sino intuir qué necesitas y buscar en la documentación o en algún ejemplo hasta que das con esa instrucción. Leed documentación, tutoriales, ejemplos... haced pequeños experimentos practicando lo que vais aprendiendo, esa es la clave para dominar cualquier tema.

Os dejaré algunos enlaces de documentación en los apuntes por si queréis profundizar más sobre Tkinter:

- Web oficial
- Tutorialspoint
- Documentación de Tcl/Tk

Primer programa con WxPython

Ahora que hemos visto Tkinter, las otras bibliotecas se entenderán mucho más fácilmente, ya que la estructura de los programas es prácticamente la misma cambiando un poco la sintaxis.

En esta lección vamos a conocer WxPython /wixpython/, un binding para WxWidgets /wixwidgets/, una biblioteca libre y multiplataforma para desarrollar interfaces en C++.

Cuenta con un conjunto de elementos gráficos mucho más extenso que el de Tkinter y a diferencia de éste es mucho más flexible para controlar la interfaz.

Algunas características interesantes que ofrece son por ejemplo una apariencia más nativa en cada sistema operativo, la posibilidad de separar el diseño del código usando ficheros XML y también la de crear tus propios componentes gráficos. Por contra es más difícil de aprender y hay que instalarla a parte.

Así que empecemos por instalarla desde el prompt de Python o una CMD:

```
pip install wxpython
```

Ahora veamos como es aquí la estructura base de un programa:

```
hola_wx.py
import wx

# Creamos una nueva app
app = wx.App()

# Iniciamos el bucle
app.MainLoop()
```

Al ejecutar este código no ocurrirá nada, eso es porque la aplicación raíz no tiene componente gráfico, sirve para manejar el proceso.

Para crear una ventana instanciaremos un componente llamado **Frame**:

```
import wx

app = wx.App()

# Le añadimos un marco contenedor
frame = wx.Frame(None, wx.ID_ANY, "Hola mundo")

app.MainLoop()
```

Pero con esto aún no es suficiente, tenemos que mostrar ese **Frame** porque por defecto está escondido:

```
frame = wx.Frame(None, wx.ID_ANY, "Hola mundo")
frame.Show(True)
```

Para establecerle un tamaño distinto lo haremos con un argumento **size**:

```
frame = wx.Frame(None, wx.ID_ANY, "Hola mundo", size=(200, 100))
```

Esto ya empieza a funcionar, vamos a añadir un botón a la ventana:

```
button = wx.Button(frame, wx.ID_ANY, "Hola")
```

¡Perfecto! Ya sólo nos falta crear una función y llamarla al presionar el botón:

```
def hola(self):
    print("¡Hola mundo!")

# ...

button = wx.Button(frame, wx.ID_ANY, "Hola")
```

```
# Aquí conectamos la función al botón con el método Bind
button.Bind(wx.EVT_BUTTON, hola)
```

Listo, ya tenemos una versión estructurada del programa "Hola mundo, ahora vamos a trasladar esto a clases y objetos:

```
import wx


class MainWindow(wx.Frame):
    # Creamos nuestra ventana a partir de un Frame

    def __init__(self):
        # Con super ejecutamos su propio constructor y le pasamos los
argumentos
        # Así se crea la ventana en su propia instancia self
        super().__init__(None, wx.ID_ANY, size=(200, 100))
        button = wx.Button(self, wx.ID_ANY, "Hola")
        button.Bind(wx.EVT_BUTTON, self.hola)
        # Finalmente mostramos la ventana
        self.Show(True)

    # Se requiere un parámetro de argumentos indeterminados en los
métodos
    def hola(self, *args):
        print("¡Hola mundo!")


# Creamos la aplicación, la ventana e iniciamos el bucle
app = wx.App()
window = MainWindow()
app.MainLoop()
```

En esencia es casi lo mismo que la versión hecha con Tkinter, sólo que aquí como necesitamos utilizar obligatoriamente un **Frame** para hacer de contenedor podemos usarlo en nuestra propia clase.

Igual que antes os dejaré un par de enlaces con documentación en los apuntes:

- Web oficial
- Tutorialspoint

Historia de Qt, PyQt y PySide

Finalmente llegamos a mi biblioteca favorita y en la que se basarán algunos de mi mis futuros cursos de Python, así que me vais a permitir explicaros un poco su historia.

¿Conocéis Qt? Pronunciado como bonito o lindo en inglés.

Qt es un famosísimo framework orientado a objetos utilizado para desarrollar programas con interfaz de usuario usando C++. Empezó en 1995 como un proyecto de la empresa Nokia y desde 2014 es desarrollado por la compañía finlandesa "The Qt Company" como software libre y código abierto bajo un proyecto comunitario llamado "Qt Project".

Muchísimos programas profesionales utilizan Qt, por ejemplo Google Earth, VirtualBox, el reproductor VLC, Autodesk Maya, CryEngine... Incluso el famoso entorno de escritorio KDE y su suite de programas.

Siendo Qt un portento en el mundo de las interfaces de usuario no es de extrañar que tenga bindings o puentes para utilizarlo en otros lenguajes y Python no iba a ser una excepción.

En el año 1999 apareció PyQt, un binding de Qt creado por la compañía inglesa Riverbank Computing, distribuido bajo varias licencias diferentes. Una libre y otra para uso comercial con un coste que actualmente es de $550 USD por desarrollador.

En 2009, Nokia intentó negociar con Riverbank Computing la liberación de PyQt bajo una licencia libre LGPL, pero no lo consiguió, seguramente porque esta empresa vive específicamente de la venta de licencias de PyQt, tal como explica en su propia página web. Como consecuencia ese mismo año Nokia lanzó su propio binding llamado PySide, prácticamente un clon de PyQt pero con la licencia libre LGPL.

El problema es que PySide no recibió suficiente apoyo y PyQt era muy superior, así que muchos clientes preferían usar la alternativa de pago. Las cosas empezaron a cambiar lentamente con la revisión PySide2 lanzada en 2015, sin embargo el futuro no era muy alentador debido una vez más a la falta de soporte.

Pero esto cambió hace unos meses.

En diciembre de 2018, "The Qt Company" anunció inesperadamente su pleno apoyo a PySide2, lo rebautizó como proyecto "Qt for Python" y decidió lanzarlo como producto oficial de su plataforma Qt.

¿Cuál podría ser la razón tras este cambio de estrategia? Yo creo que notaron el auge de Python y no han querido perder la oportunidad de subirse al carro, pero en cualquier caso esto es muy positivo para la comunidad, ya que por fin tenemos un binding oficial de Qt distribuido bajo una licencia libre y con buen soporte.

Más no se puede pedir.

Primer programa con PySide

Instalar PySide se puede hacer fácilmente con pip:

```
pip install pyside6
```

La estructura principal de un programa en PySide es muy parecida a la de WxPython, aquí también necesitamos crear un contenedor para la ventana.

Hay varios pero el más común es el widget para la ventana principal:

```
hola_pyside.py
from PySide6 import QtWidgets as QW

# Creamos una nueva app
app = QW.QApplication()

# Creamos la ventana principal y la mostramos
window = QW.QMainWindow()
window.resize(200, 100)
window.show()

# Iniciamos el bucle
app.exec_()
```

Añadir un botón y conectarlo a una función también se hace casi igual que en WxPython:

```
def hola(self):
    print("¡Hola mundo!")


# ...

# Creamos el botón y lo conectamos
button = QW.QPushButton("Hola", window)
button.clicked.connect(hola)
```

Este es el código correcto, sin embargo no veremos el botón. Esto es debido a que en Qt tenemos que mostrar la ventana al final, después de haber añadido sus componentes:

```
# Creamos la ventana principal
window = QW.QMainWindow()
window.resize(200, 100)

# Creamos el botón y lo conectamos
button = QW.QPushButton("Hola", window)
button.clicked.connect(hola)

# Mostramos la ventana principal
window.show()
```

Ya lo tenemos, en mi opinión la sintaxis es más intuitiva que en las otras bibliotecas, no sé qué pensaréis vosotros.

En fin, vamos a llevar este código a nuestra propia clase para dejarlo finiquitado:

```
from PySide6 import QtWidgets as QW


class MainWindow(QW.QMainWindow):
    # Creamos nuestra ventana a partir de un QMainWindow
    def __init__(self):
        # Con super ejecutamos su propio constructor
        # Así se crea la ventana en su propia instancia self
        super().__init__()
        # Redimensionamos y añadimos el botón
```

```
        self.resize(200, 100)
        button = QW.QPushButton("Hola", self)
        button.clicked.connect(self.hola)
        # Finalmente mostramos la ventana
        self.show()

    def hola(self):
        print("¡Hola mundo!")


# Creamos la aplicación, la ventana e iniciamos el bucle
app = QW.QApplication()
window = MainWindow()
app.exec_()
```

¿Qué os ha parecido PySide? ¿Verdad que es prácticamente la misma lógica que WxPython pero cambiando un poco la sintaxis? Ya os dije al comenzar el curso que todas las biblioteas se parecían mucho.

En cualquier caso os dejaré la documentación en los apuntes para que aprendáis un poco más por vuestra cuenta:

- Web oficial
- Tutoriales oficiales

Y con esto acabamos.

Espero que el curso os haya servido, sobretodo la unidad de programación orientada a objetos. Esos conocimientos se utilizan no sólo en bibliotecas de interfaces, sino también en desarrollo web, videojuegos y muchos otros ámbitos, así que repasadlos de tanto en tanto.