UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

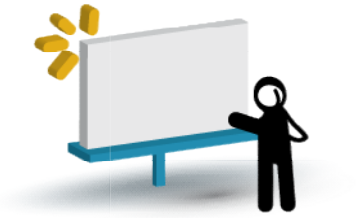
UNIDAD DIDÁCTICA V

DIPLOMATURA EN PYTHON

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo II – Nivel Inicial II

Unidad I – Funciones II, ámbito de variable, estructura de control y bucles.



Presentación:

En esta unidad seguiremos viendo temas relacionados con funciones bucles y estructuras de control, y comenzaremos a utilizarlas dentro de la plataforma tkinter.

Objetivos:

Que los participantes:

Aprendan a trabajar con argumentos de funciones.

Adopten el concepto de recursividad y puedan aplicarlo.

Apliquen temas aprendidos en tkinter.

Bloques temáticos:

1.- Funciones – Sintaxis de asignación de argumentos.

2.- Funciones avanzadas.

3.- GUI – Funciones y funcionalidades.

Consignas para el aprendizaje colaborativo

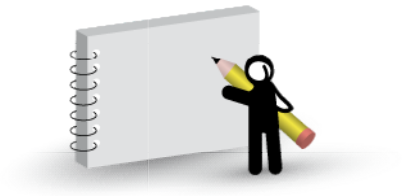
En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Funciones - Sintaxis de asignación de argumentos.

Un tema interesante que vamos a analizar ahora es cómo indicar los argumentos pasados a una función, veámoslo punto por punto.

Asignación por posición

La forma más común de pasar argumentos, es respetando la posición, en el siguiente ejemplo notemos cómo pasamos a argumentos dos objetos totalmente distintos (un número y una lista). El primer elemento pasado es el número y el segundo corresponde a la lista.

argumentos1.py

```
def modificar(a, b):
    a = 2
    b[0]="Manzana"
A=1
L=[1,2]
modificar(A,L)
print('',end='\n##################\n' )
print(A, L)
```

Dado que las listas no son INMUTABLES, podemos modificar uno de sus valores, en este caso el primero, por lo que el resultado del script queda como sigue:

```
##################
1 ['Manzana', 2]
```

Asignación por nombre

Es posible indicar el nombre del argumento a pasar, e incluso realizar una mezcla entre asignación por posición y por nombre, veamos tres ejemplos para clarificar este punto.

argumentos2.py

```
# ################################################
# Por posición ####
# ################################################
def f(x, y, z):print(x,y,z)
f(1,2,3)
print('',end='\n################\n' )
# ################################################
# Por nombre ####
# ################################################
def f(x, y, z):print(x,y,z)
f(z=3, y=2, x=1)
print('',end='\n################\n' )
# ################################################
# Por mixto - primero de izquierda a derecha ##
# y luego por nombre ####
# ################################################
def f(x, y, z):print(x,y,z)
f(1 ,z=3, y=2)
```

Asignación por defecto

Algo que vamos a utilizar muy a menudo, es establecer un valor por defecto, es decir que si no se especifica un valor, el valor que toma una variable es el que establecemos en la declaración de la función. Como ejemplo pensemos en los estilos de una aplicación que por defecto viene seteado con color de fondo blanco y letras azules y que posteriormente podemos modificar desde un menú.

Un ejemplo nos va a permitir fijar el conocimiento:

argumentos3.py

```
def f(a, b=2, c=3): print(a, b, c)
f(1)
```

Nota: En este caso el único valor que debemos pasarle de forma obligada es el valor de "a", sin embargo tanto en el caso de "b" como de "c" no es necesario pasar un valor a no ser que queramos pisar el valor determinado por defecto.

Uso de * y **

Mediante el uso de * podemos pasar tuplas o listas a una función, si dentro de la función usamos "args" podemos manejar los elementos como si fueran una tupla.

argumentos4.py

```
def f(a, *args):
    print("Tupla de valores: ", args)
        for arg in args:
        print("Elemento de la tupla: " , arg)

f(0, 1, 2, "Manzana")
```

La salida queda:

```
Tupla de valores: (1, 2, 'Manzana')
Elemento de la tupla: 1
Elemento de la tupla: 2
Elemento de la tupla: Manzana
```

Nota: El uso del nombre args es arbitrario podría usar otro nombre.

Si en lugar de un asterisco ponemos dos, python interpreta los datos pasados como un diccionario.

argumentos5.py

```
def funcion(**kwargs):
    if kwargs is not None:
        for clave, valor in kwargs.items():
            print( "%s == %s" %(clave, valor))
funcion(nombre="Juan", edad=1, sexo="Masculino")
```

Retorna:

```
edad == 1
nombre == Juan
sexo == Masculino
```

Un ejemplo mixto de lo visto hasta ahora nos puede evidenciar lo amplio del lenguaje de Python en la asignación de argumentos.

argumentos6.py

```
def funcion(a, *pargs, **kwargs):
    print(a, pargs,kwargs)
    print("-----------")
    print(a)
    print("-----------")
    print(pargs)
    print("-----------")
    if kwargs is not None:
        for clave, valor in kwargs.items():
            print( "%s == %s" %(clave, valor))

funcion(1, 2, 3, nombre="Juan", edad=1, sexo="Masculino")
```

Retorna:

```
1 (2, 3) {'sexo': 'Masculino', 'edad': 1, 'nombre': 'Juan'}
-----------
1
-----------
(2, 3)
-----------
sexo == Masculino
edad == 1
nombre == Juan
```

Nota: Los nombres pargs, y kwargs son totalmente arbitrarios.

Nota: Lo que sí es importante es en el caso mixto respetar el orden a, *pargs, **kwargs

2. Funciones avanzadas

Recursividad.

Recurrencia, recursión o recursividad es la forma en la cual se especifica un proceso basado en su propia definición. Nada mejor que un ejemplo gráfico para comenzar a darnos una idea de qué se trata.

https://www.quora.com/Is-there-a-blank-mirror-at-the-end-of-an-Infinity-Mirror-reflection

Este concepto es muy útil en programación, en donde muchas veces tomamos el resultado de la ejecución de una función como entrada de la misma función, veamos un ejemplo para clarificar el concepto:

Supongamos que tenemos una función como la que sigue, que lo que hace es imprimir el valor de la lista que le pasamos, luego si la lista es vacía retorna cero, si no nos da el primer elementos de la lista y ejecuta nuevamente la función, pero esta vez quitándole el primer elemento y sumando el resultado de la nueva ejecución al primer elementos guardado.

recursividad.py

```
def misuma(L):
    print(L)
    if not L:
        return 0
    else:
        return L[0] + misuma(L[1:])
print(misuma([1, 2, 3, 4, 5]))
```

Como resultado lo que obtenemos es la impresión de los valores obtenidos en cada vuelta y al final cuando misuma() ya no se ejecuta el valor de la suma de cada elemento.

Retorna:

```
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
[]
15
```

Nota: Podemos utilizar una función ternaria para obtener la suma de los elementos con una notación un poco más escueta, con la ayuda de if/else.

recursividad_ternarias.py

```
def miSuma(L):
    return 0 if not L else L[0] + miSuma(L[1:])
print(miSuma([1, 2, 3, 4, 5]))
```

Recursividad indirecta.

En algunas ocasiones mientras trabajamos podemos definir que una parte de las cuentas o mejor dicho de la lógica de un script lo lleve a cabo otra función, para facilitar la lectura de código y poder encontrar errores más rápidamente.

Veamos cómo queda el ejercicio anterior, si utilizamos una segunda función que se encarga de la suma de los elementos.

recursividad_indirecta.py

```
def misuma(L):
    if not L:
        return 0
    return novacia(L)            # llama a otra función que llama a misuma

def novacia(L):
    return L[0] + misuma(L[1:]) # Recursividad indirecta

print(misuma([1.1, 2.2, 3.3, 4.4]))
```

Recursividad vs loop.

Lo que podemos hacer con una recursión, lo podemos realizar con un loop como por ejemplo un while

recursividad_loop.py

```
L = [1, 2, 3, 4, 5]
suma = 0
while L:
    suma += L[0]
    L=L[1:]
print(suma)
```

Datos de la función.

A continuación ejecutamos algunas líneas de código para obtener datos de la función ejecutada.

datos_funcion.py

```
def misuma(L):
    print(L)
    if not L:
        return 0
    else:
        return L[0] + misuma(L[1:])

print(misuma([1, 2, 3, 4, 5]))
print('',end='\n##################\n' )
print(misuma.__name__) #Obtengo el nombre
print('',end='\n##################\n' )
print(dir(misuma)) #Obtengo sus atributos
print('',end='\n##################\n' )
print(misuma.__code__) #Datos de la función
print('',end='\n##################\n' )
print(dir(misuma.__code__)) #Datos de la función
print('',end='\n##################\n' )
print(misuma.__code__.co_varnames) #Nombre de variables utilizadas
print('',end='\n##################\n' )
print(misuma.__code__.co_argcount) #Cantidad de argumentos
```

Uso de __annotations__.

Consideremos la siguiente función:

anotations.py

```
def func1(a, b, c):
    return a + b + c

print(func1(1, 2, 3))
```

Puedo agregar comentarios a los atributos si en la declaración le pongo dos puntos y luego el comentario

Con __annotations__ recupero los comentarios

Con -> índico un comentario para el tipo de retorno de la función

```
def func2(a: 'spam', b: (1, 10), c: float) -> int:
    return a + b + c

print(func2(1, 2, 3))
print(func2.__annotations__)
```

Retorna:

```
6
{'b': (1, 10), 'c': <class 'float'>, 'a': 'spam', 'return': <class 'int'>}
```

También puedo usar valores por defecto junto con los comentarios

```
def func3(a: 'spam' = 4, b: (1, 10) = 5, c: float = 6) -> int:
    return a + b + c

print(func3(1, 2, 3))
print(func3.__annotations__)
```

Función Lambda.

Una función lambda, es una función con una sintaxis mínima que puede ser definida en cualquier momento dentro del código.

La forma general de definir una función lambda es:

lamda.py
```
lambda argument1, argument2,... argumentN : expression using arguments
```

Tomemos como ejemplo la siguiente función

```
def func1(a, b, c):return a + b + c
print(func1(1, 2, 3))
```

Puedo escribir una función lambda equivalente como a continuación:

```
func2 = lambda a,b,c: a+b+c
print(func2(1, 2, 3))
```

Nota 1: La lista de argumentos no está entre paréntesis.

Nota 2: La palabra reservada return está implícita, ya que la función entera debe ser una única expresión.

Nota 3: La función no tiene nombre, pero puede ser llamada mediante la variable a que se ha asignado.

Nota4: Es posible definir una función lambda sin asignarla a una variable, pero su aplicación no es muy útil.

Incluso puedo establecer valores por defecto:

```
func3 =(lambda a= 4, b= 5, c= 6 :a+b+c)
print(func3(1, 2, 3))
```

3. GUI – Funciones y funcionalidades.

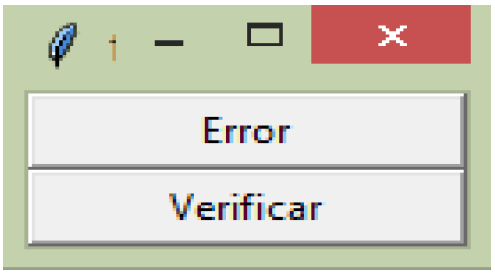
Cajas de diálogo

El módulo tkMessageBox de tkinter nos provee de una interface para representar cuadros de diálogo, mediante el uso de las funciones: showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, o askretrycancel. El nombre de cada una de estas funciones ya nos dice para qué se utilizan, veamos un ejemplo de su implementación.

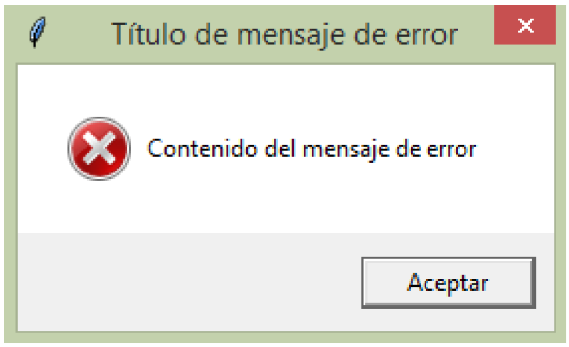
dialogo1.py

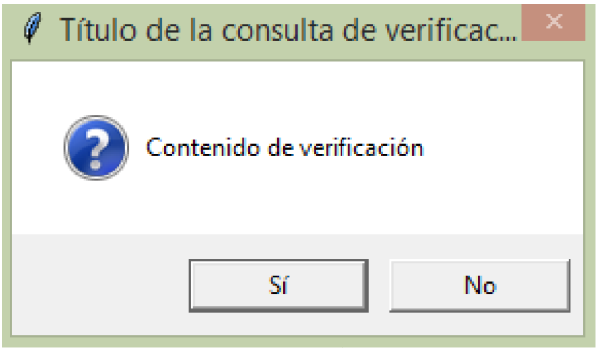
```
from tkinter import *
from tkinter.messagebox import *

def mensaje_error():
    showerror("Título de mensaje de error",
        "Contenido del mensaje de error")

def verificar():
    if askyesno('Título de la consulta de verificación',
        'Contenido de verificación'):
        showinfo('Si', 'Mensaje de información')
    else:
        showinfo('No', 'Esta a punto de salir')

Button(text='Error', command=mensaje_error).pack(fill=X)
Button(text='Verificar', command=verificar).pack(fill=X)

mainloop()
```

Representación gráfica pantalla principal:



Opciones de mensaje de Error y “Verificar”:





Podríamos modificar el script anterior con el uso de una función lambda para ejecutar la función error, según se muestra en el siguiente ejemplo:

dialogo2.py

```
from tkinter import *
from tkinter.messagebox import *

def verificar():
    if askyesno('Título de la consulta de verificación',
        'Contenido de verificación'):
        showinfo('Si', 'Mensaje de información')
    else:
        showinfo('No', 'Esta a punto de salir')
Button(text='Error', command=(lambda: showerror('Título de mensaje de
error',"Contenido del mensaje de error"))).pack(fill=X)
Button(text='Verificar', command=verificar).pack(fill=X)

mainloop()
```

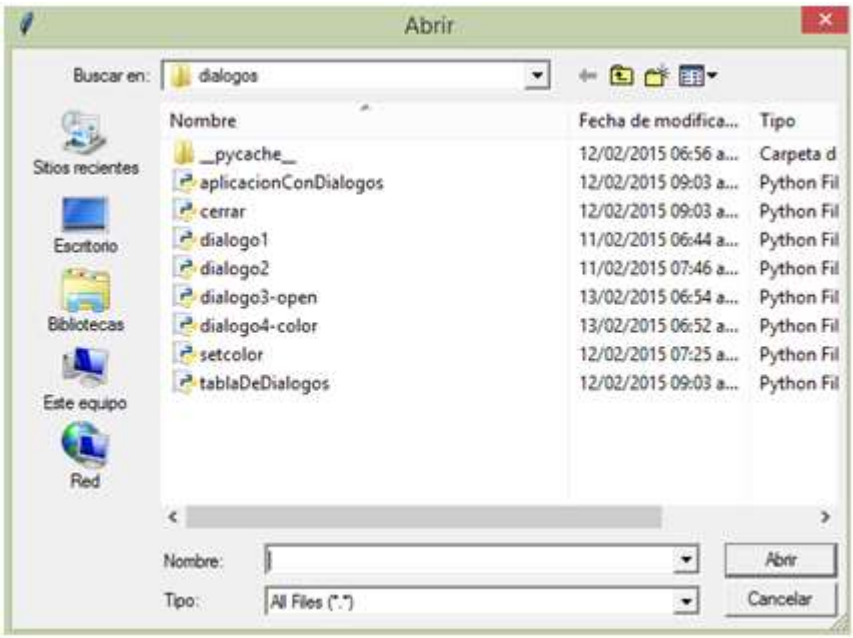
Cajas de diálogo – Seleccionar Archivo

En determinada ocasiones necesitamos poder seleccionar archivos y abrirlos para leerlos o editarlos, para estos casos tkinter nos proporcionar el método askopenfilename () con el cual podemos seleccionar un archivo y obtener su path. El siguiente ejemplo presenta un botón que al presionarlo nos permite seleccionar un archivo e imprimir su path en pantalla.

dialogo3.py

```
from tkinter import *
from tkinter.filedialog import askopenfilename


def callback():
    ruta = askopenfilename()
    print(ruta)

Button(text='Abrir archivo', command=callback).pack(fill=X)
mainloop()
```

Visualización:



Cajas de diálogo – Selección de color

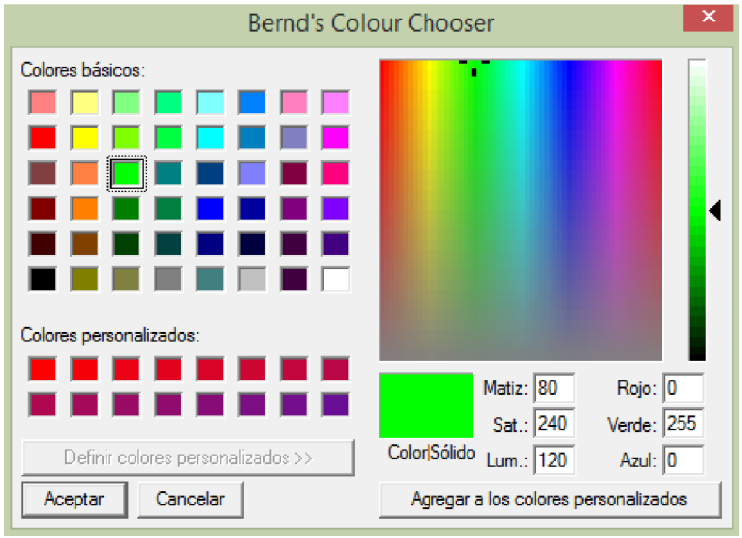
Otro método muy útil es askcolor (), con el cual podemos seleccionar desde una paleta de colores un determinado color, veamos cómo utilizarlo con el siguiente ejemplo:

dialogo4_color.py

```
from tkinter import *
from tkinter.colorchooser import askcolor

def callback():
```

```
    result = askcolor(color="#00ff00",title = "Bernd's Colour
           Chooser")
    print(result)
    print(result[1])

root = Tk()
Button(root, text='Seleccionar color', fg="green",
    command=callback).pack(side=LEFT, padx=10)
Button(text='Cerrar', command=root.quit,
    fg="red").pack(side=LEFT, padx=10)
mainloop()
```

Visualización:



Imágenes

En tkinter, las imágenes pueden ser presentadas dentro de botones, cajas, canvas, etc, al asociarles una imagen o un bitmap mediante el uso de un atributo. En el siguiente ejemplo vemos como dada la ruta a un directorio donde se encuentra una imagen, podemos mediante el método PhotoImage() asociarle la imagen a un botón con el uso del atributo image. Dado que no hemos especificados dimensiones para el botón, este adquiere la dimensión de la imagen.

imagen1_boton.py

```
from tkinter import *
ruta = "img/"
win = Tk()
imagen = PhotoImage(file=ruta + "download.gif")
Button(win, image=imagen).pack()
win.mainloop()
```

Visualización:

Como podemos ver a continuación, es posible utilizar otro elemento de tkinter, en este caso un canvas.

imagen2_boton.py

```
from tkinter import *
ruta = "img/"
win = Tk()
imagen = PhotoImage(file=ruta + "download.gif")
can = Canvas(win)
can.pack(fill=BOTH)
can.create_image(2, 2, image=imagen, anchor=NW)
win.mainloop()
```

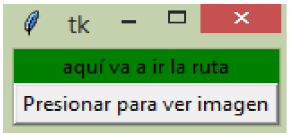
Seleccionar imagen

Vamos ahora a introducir el módulo ***glob***, el cual nos permite seleccionar una serie de archivos que tengan una determinada extensión, y lo vamos a utilizar para presentar una imagen de forma aleatoria seleccionada de un determinado directorio (mediante el uso del módulo random) al presionar en un botón.

imagen3_seleccion.py

```
from tkinter import *
from glob import glob
import random
ruta = 'img/'

def seleccion():
    nombre, foto = random.choice(imagen)
    dialogo.config(text=nombre)
    boton.config(image=foto)

root=Tk()
dialogo = Label(root, text="aquí va a ir la ruta", bg='OrangeRed')
boton = Button(root, text="Presionar para ver imagen", command=seleccion)
dialogo.pack(fill=BOTH)
boton.pack()

archivo = glob(ruta + "*.gif")
imagen = [(x, PhotoImage(file=x)) for x in archivo]
print(archivo)
root.mainloop()
```

Visualización:



Librería PILLOW

Para poder trabajar con una mayor cantidad de formatos de imágenes e incluso editarlas, podemos utilizar la librería de python PILLOW (que es una actualización de la librería PIL) la cual nos permite trabajar con más de 30 formatos, entre los cuales se encuentran por ejemplo: png, jpeg, jpg.

La librería no tiene porqué utilizarse dentro de tkinter, pero si la integramos a tkinter, ya viene con métodos que poseen el mismo nombre que los que tkinter utiliza de forma que su integración requiera únicamente realizar la importación del método correspondiente, en este caso el método PhotoImage de PILLOW el cual pisa al método PhotoImage de tkinter, ya que este únicamente nos permite trabajar con formatos .gif.

imagen4_pillow.py

```
from tkinter import *
from PIL.ImageTk import PhotoImage
from glob import glob
import random
ruta = 'images/'

def seleccion():
    nombre, foto = random.choice(imagen)
    dialogo.config(text=nombre)
    boton.config(image=foto)

root=Tk()
dialogo = Label(root, text="aquí va a ir la ruta", bg='green')
boton = Button(root, text="Presionar para ver imagen", command=seleccion)
dialogo.pack(fill=BOTH)
boton.pack()

archivo = glob(ruta + "*.jpg")
imagen = [(x, PhotoImage(file=x)) for x in archivo]
print(archivo)
root.mainloop()
```

Librería PILLOW - thumbnails

Pillow no solamente nos permite presentar diferentes formatos de imágenes, sino que también nos permite editarlas. A modo de ejemplo creemos un thumbnail de cada imagen dentro de un cierto directorio (ver código completo en el archivo "**imagen5-pillow-thumnail.py**".

Si partimos de un determinado directorio "**images**" en donde tendremos las imágenes a convertir a thumbnails, utilizamos la función "**crearThumbs()**" a la cual le pasamos el directorio.

```
if __name__ == '__main__':
    directorioImagenes = 'images'
    thumbs = crearThumbs(directorioImagenes)
```

y que posee por defecto los parámetros "size" (para determinar el tamaño del thumbnails) y el "subdirectorio" (el cual es el nombre del subdirectorio por defecto en donde se van a guardar los thumbnails)

```
def crearThumbs(directorio, size=(100, 100), subdirectorio='thumbs'):
```

El método crearThumbs(), utiliza el directorio "os" y su método "mkdir" para crear el subdirectorio en donde se van a guardar las imágenes miniatura en caso de que no exista.

```
directorioParaThumb = os.path.join(directorio, subdirectorio)
if not os.path.exists(directorioParaThumb):
        os.mkdir(directorioParaThumb)
```

Luego para cada imagen del directorio de imágenes, abre la imagen mediante:

```
imgobj = Image.open(imgpath)
```

crea la imagen miniatura, a la cual le aplica un filtro:

```
imgobj.thumbnail(size, Image.ANTIALIAS)
```

y la guarda:

```
imgobj.save(thumbpath)
```

El código completo queda:

imagen5_pillow_thumnail.py

```
import os
from tkinter import *
from PIL import Image

def crearThumbs(directorio, size=(100, 100), subdirectorio='thumbs'):

    directorioParaThumb = os.path.join(directorio, subdirectorio)
    if not os.path.exists(directorioParaThumb):
        os.mkdir(directorioParaThumb)

    for imagen in os.listdir(directorio):
        thumbpath = os.path.join(directorioParaThumb, imagen)
        print('Creando', thumbpath)
        imgpath = os.path.join(directorio, imagen)
        try:
            imgobj = Image.open(imgpath)
            imgobj.thumbnail(size, Image.ANTIALIAS)
            imgobj.save(thumbpath)
        except:
            print("Skipping: ", imgpath)

if __name__ == '__main__':
    directorioImagenes = 'images'
    thumbs = crearThumbs(directorioImagenes)
```

El resultado final, es un subdirectorio de imágenes miniatura de las imágenes originales.

Menú

Una herramienta fundamental para lograr realizar una aplicación ordenada, es el widget Menu, el cual nos permite crear menús desplegables y popups.

La forma de utilizarlo, es creando un elemento menú, el cual está asociado a una caja

```
menubar = Menu(root)
```

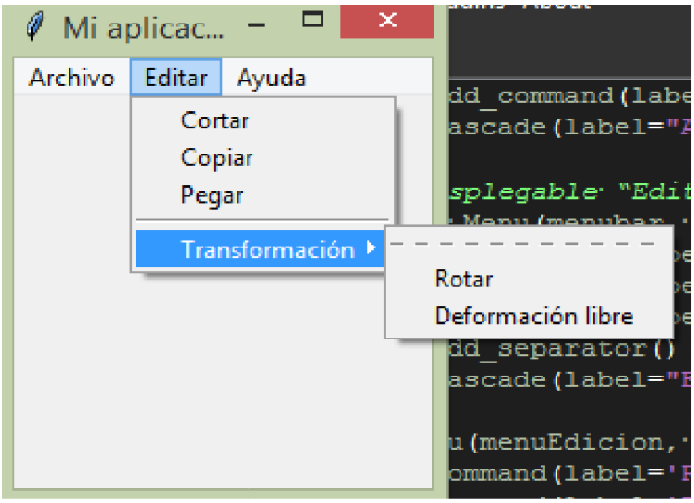
y luego crear un elemento del menú y agregarle los elementos que vamos a utilizar (mediante **.add_command()**), cada elemento puede ser asociado a una función (en este caso todos están asociados a la función "**hola**" que lo único que hace es imprimir un "Hola!" en pantalla. El elemento es asociado al menú mediante "**.add_cascade()**", e incluso podemos agregar un separador entre elementos mediante "**add_separator()**"

```
menuArchivo = Menu(menubar, tearoff=0)
menuArchivo.add_command(label="Abrir", command=hola)
menuArchivo.add_command(label="Guardar", command=hola)
menuArchivo.add_separator()
menuArchivo.add_command(label="Salir", command=root.quit)
menubar.add_cascade(label="Archivo", menu=menuArchivo)
```

En caso de querer agregar un submenú, podemos realizarlo, asociando el submenú al elemento correspondiente:

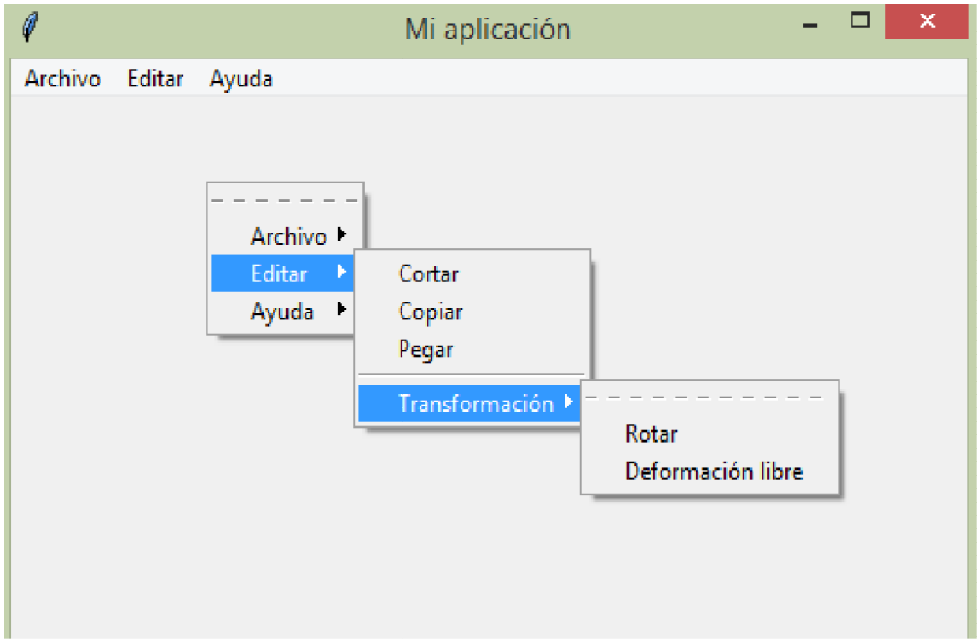
```
submenu = Menu(menuEdicion, tearoff=True)
submenu.add_command(label='Rotar', command=hola)
submenu.add_command(label='Deformación libre', command=hola)
menuEdicion.add_cascade(label='Transformación', menu=submenu)
```

Representación gráfica:



Menú - popups

Algo muy útil y a lo cual estamos acostumbrados en muchas aplicaciones, es al uso de popups para mostrar los menús, como puede verse en la siguiente representación:



En la mayoría de las aplicaciones, esto se realiza mediante el uso del botón derecho del mouse sobre el área de trabajo (en tkinter el botón derecho viene representado por: **<Button-3>**). El método bind() es en este caso el que utilizamos para crear el popup,

asociando en este caso la función “popup()” y ejecutando dentro de la misma, una línea de código en donde indicamos mediante “post()” que un determinado menú sea representado a partir de las coordenadas x e y en donde presionamos con el botón derecho del mouse.

```
##################################################################
# crear una caja
##################################################################
frame = Frame(root, width=512, height=512)
frame.pack()

def popup(event):
    menubar.post(event.x_root, event.y_root)

# asociar el popup a la caja
frame.bind("<Button-3>", popup)
```

Nota: Ver código completo en menu2-popup.py

Menú - Reconfiguración

Algo que es muy útil, es poder llevar un registro de eventos pasados, como puede verse en la herramienta historial de Photoshop, para esto le pasamos al parámetro “**postcommand**” una determinada función que hace algo, en el ejemplo siguiente es asociada a la función “update()”, la cual cada vez que es invocada incrementa el valor de la variable global “**contar**” en una unidad.

```
contar = 0

def update():
    global contar
    contar = contar + 1
    menuPrueba.entryconfig(0, label=str(contar))

menuPrueba = Menu(menubar, tearoff=0, postcommand=update)
menuPrueba.add_command(label=str(contar))
menuPrueba.add_command(label="Salir", command=root.quit)
menubar.add_cascade(label="Prueba", menu=menuPrueba)
```

Nota: Ver código completo en menu3-reconfigurar.py

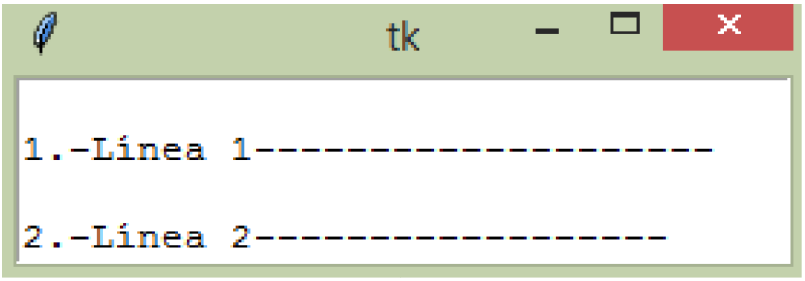
Texto

Para ingresar por pantalla grandes cantidades de texto, de forma que posteriormente se pueda analizar, utilizamos el widget Text(), su utilización es similar al resto de los widgets, en este caso para agregar texto lo logramos mediante el método **"insert()"**. La cantidad de líneas de texto a visualizar está representado por el atributo **"height"**, y la cantidad de caracteres por **"width"**

texto1.py

```
from tkinter import *
root = Tk()

T = Text(root, height=4, width=33)
T.pack()
texto1 = """
1.-Línea 1----------
\n2.-Línea 2----------
\n3.-Línea 3----------
\n4.-Línea 4----------
\n5.-Línea 5----------
\n6.-Línea 6----------
"""
T.insert(END, texto1)
mainloop()
```

Representación gráfica:



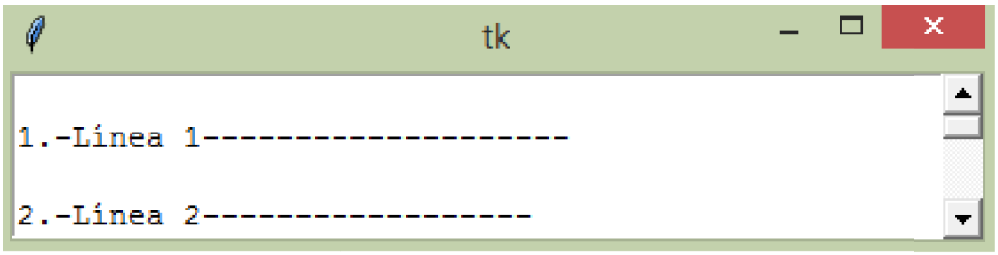
Menú - Scrollbar

En caso de que queramos que el área de trabajo permita visualizar más de las líneas preestablecidas, podemos utilizar el widget "**Scrollbar()**" como se muestra a continuación

texto2-scrollbar.py

```
from tkinter import *
root = Tk()
S = Scrollbar(root)
T = Text(root, height=4, width=50)
S.pack(side=RIGHT, fill=Y)
T.pack(side=LEFT, fill=Y)
S.config(command=T.yview)
T.config(yscrollcommand=S.set)
texto1 = """
1.-Línea 1----------
\n2.-Línea 2----------
\n3.-Línea 3----------
\n4.-Línea 4----------
\n5.-Línea 5----------
\n6.-Línea 6----------
"""
T.insert(END, texto1)
mainloop( )
```

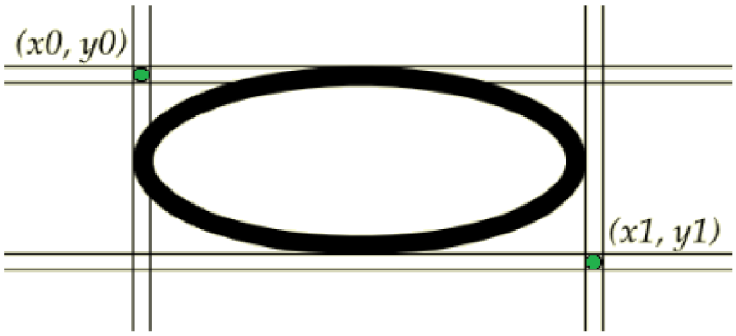
Representación gráfica:



Canvas

En caso de que queramos implementar una aplicación tipo “Paint” necesitamos trabajar con el widget “**Canvas()**”, en el cual podemos agregar imágenes y figuras como pueden ser arcos, polígonos, etc.

Para agregar un óvalo, debemos tener presente cómo se establecen sus parámetros, los dos primeros parámetros de “**create_oval**” son (x0, y0), los siguiente dos son (x1, y1) y luego vienen los parámetros para configurar espesor de línea, color, etc



Las siguientes líneas muestran una representación de tres óvalos.

```
canvas.create_oval(70, 100, 140, 200, width=2, fill='blue')
canvas.create_oval(80, 140, 120, 200, width=2, fill='white')
canvas.create_oval(90, 160, 110, 180, width=2, fill='black')
```

Para agregar una línea, le pasamos como parámetros a “create_line”:

canvas.create_line(x0, y0, x1, y1, otros parámetros)

```
canvas.create_line(200, 100, 300, 200, width=3, fill='OrangeRed')
```

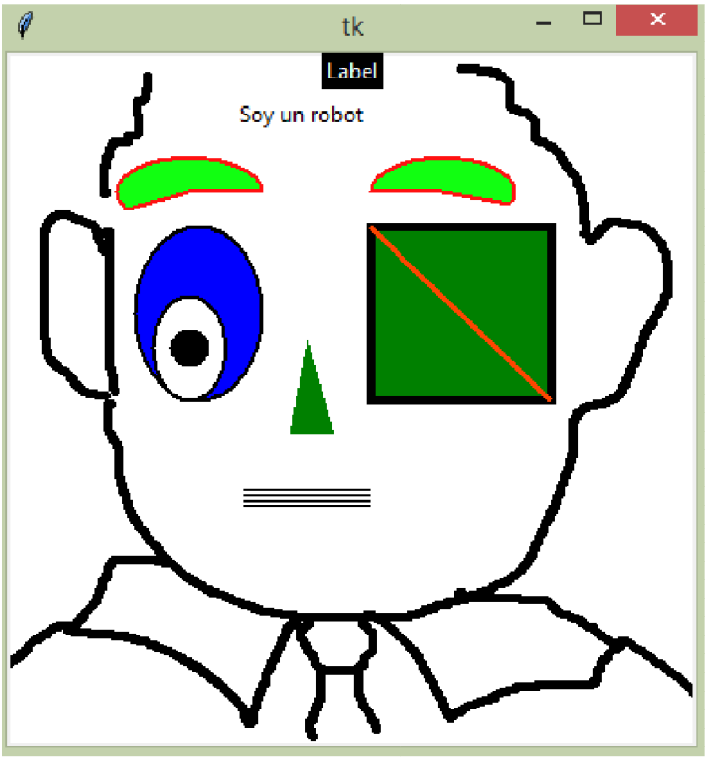
Para agregar un polígono, introducimos sus puntos de a pares

```
canvas.create_polygon(165, 165, 155, 220,180, 220, width=2, fill='green')
```

Quizás el elemento más complejo de representar es el arco, en donde los cuatro primeros parámetros se utilizan análogamente que en el óvalo, luego determinamos el ángulo inicial del arco con start (en grados) y el final con “extent”

```
canvas.create_arc(60, 60, 140, 100, start=0,
extent=210, outline="#f11", fill="#1f1", width=2)
canvas.create_arc(200, 60, 280, 100, start=-20,
extent=200, outline="#f11", fill="#1f1", width=2)
```

La siguiente es la representación del ejemplo “canvas1.py”, en donde el contorno es de una imagen en formato. gif:



Bibliografía utilizada y sugerida

Libros

Programming Python 5th Edition – Mark Lutz – O'Reilly 2013

Programming Python 4th Edition – Mark Lutz – O'Reilly 2011

Manual online

https://docs.python.org/3.7/tutorial/

https://docs.python.org/3.7/library/index.html

Lo que vimos

En esta unidad trabajamos con funciones y asignación de parámetros.

Lo que viene:

En la siguiente unidad comenzaremos a trabajar con formato de archivos y bases de datos.