Chapter 9: GUI Programming with Tkinter

Contents

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself

Tutorials

- Tkinter Course - Create Graphic User Interfaces in Python Tutorial (FreeCodeCamp Video Tutorial)
- https://realpython.com/python-gui-tkinter/

Python GUI - Tkinter

Up until now, the only way our programs have been able to interact with the user is through keyboard input via the input statement. These types of programs determine the order in which things happen.

GUI programs are event driven; the user is in control. The user can click or enter data in any order. A mouse click or keyboard event is registered: the program responds to that event with the appropriate action.

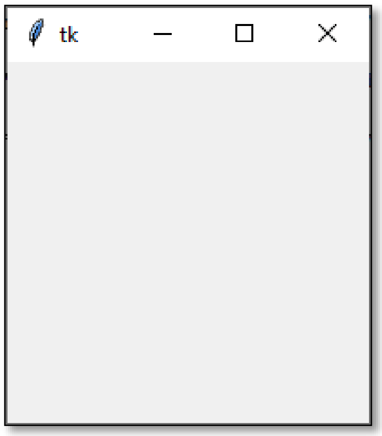
Most real programs use windows, buttons, scrollbars, and various other things. These widgets are part of what is called a Graphical User Interface or GUI.

This chapter is about GUI programming in Python with **Tkinter**. GUI programming is not built into Python. Python comes with a built-in module called **tkinter** that you can import.

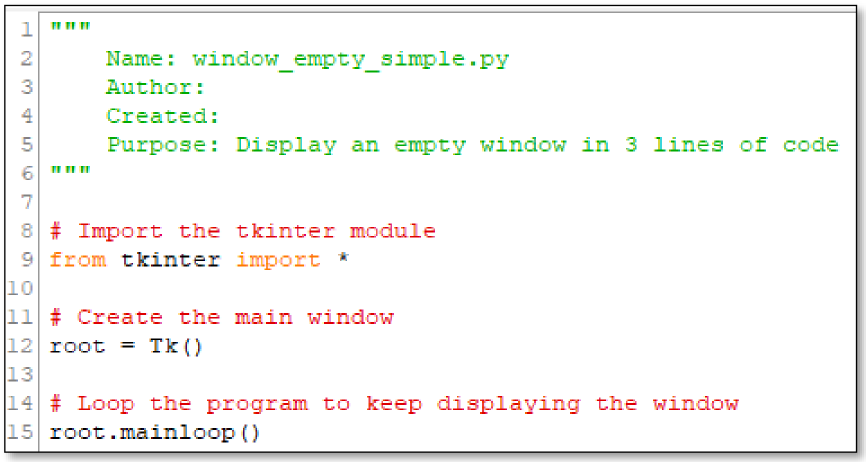
There are a couple approaches to using **tkinter**. This chapter will take the object-oriented approach, as is more flexible and reusable. Ttinter code can be written using only functions, it's much better to use a class to keep track of all individual widgets which may need to reference each other. Without doing this, you need to rely on global or nonlocal variables, which gets ugly as your app grows and is not a secure method of programming. It allows for much finer controls once your app gets more complex, allowing you to override default behaviors of Tkinter's own objects.

A GUI program starts with a window with various widgets for the user to interact with. The widgets we will be looking at have far more options than we could possibly cover.

Tutorial 9.1 – A Window



Not very impressive, is it. The code behind it isn't very complicated either.

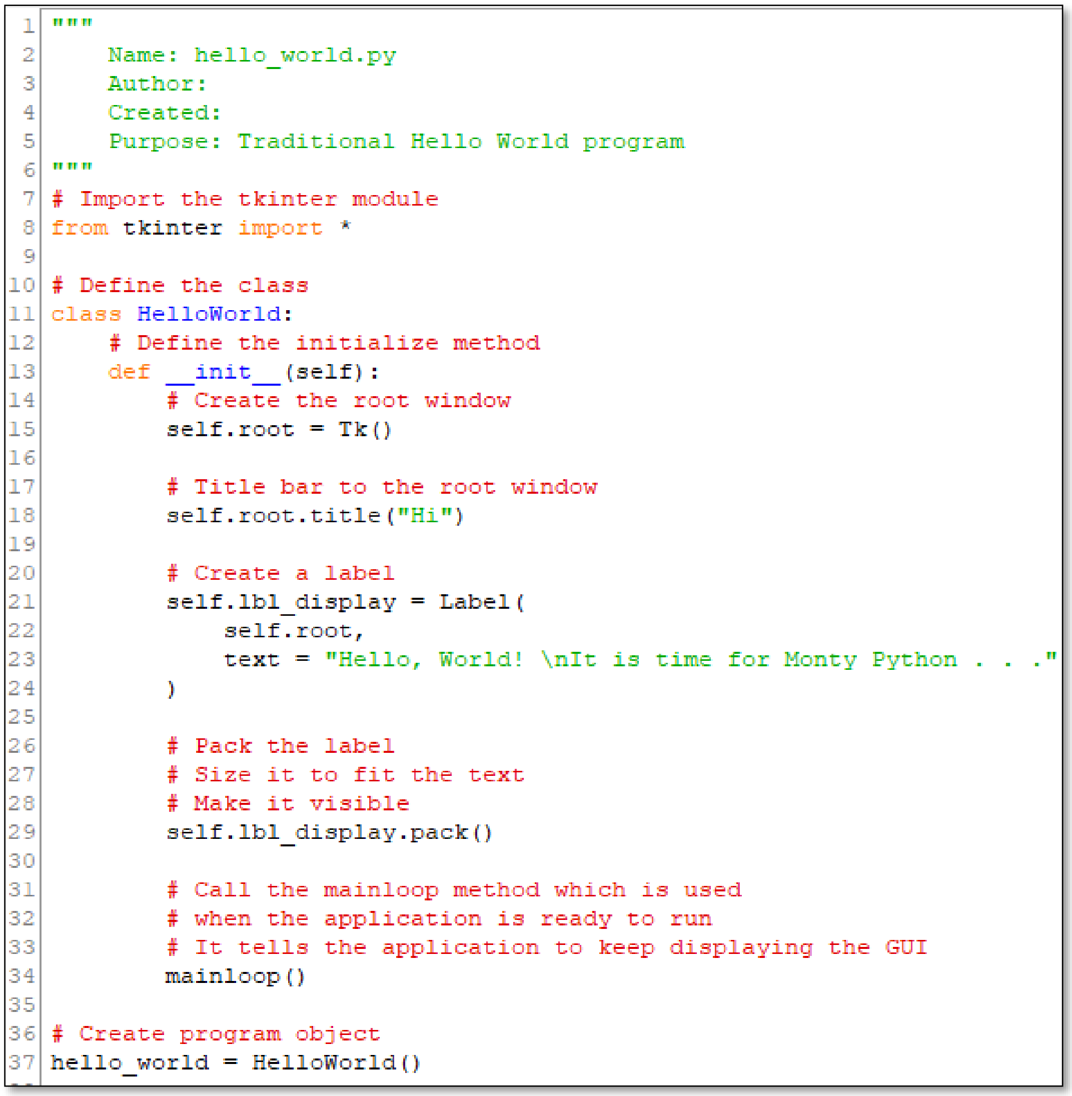


```
"""
    Name: window_empty_simple.py
    Author:
    Created:
    Purpose: Display an empty window in 3 lines of code
"""

# Import the tkinter module
from tkinter import *

# Create the main window
root = Tk()

# Loop the program to keep displaying the window
root.mainloop()
```

Of course, the first real Tkinter program we create is the traditional **Hello World** program. We create a root window, put a title in the title bar, and display a single label.

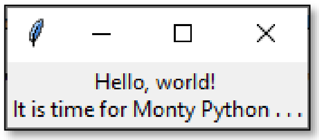
We will program in Tkinter in the OOP methods we learned earlier. This is a common practice in Python. It allows more flexibility as we learn more about Python.

Change your program to the following code.



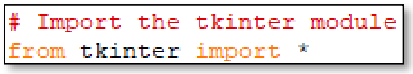
```python
"""
    Name: hello_world.py
    Author:
    Created:
    Purpose: Traditional Hello World program
"""
# Import the tkinter module
from tkinter import *

# Define the class
class HelloWorld:
    # Define the initialize method
    def __init__(self):
        # Create the root window
        self.root = Tk()

        # Title bar to the root window
        self.root.title("Hi")

        # Create a label
        self.lbl_display = Label(
            self.root,
            text = "Hello, World! \nIt is time for Monty Python . . ."
        )

        # Pack the label
        # Size it to fit the text
        # Make it visible
        self.lbl_display.pack()

        # Call the mainloop method which is used
        # when the application is ready to run
        # It tells the application to keep displaying the GUI
        mainloop()

# Create program object
hello_world = HelloWorld()
```

Example run:



```
Hello, world!
It is time for Monty Python . . .
```

How It Works



```python
# Import the tkinter module
from tkinter import *
```

We start by importing the **tkinter** module. It contains all classes, functions and other items needed to work with the **Tk** toolkit.

```
# Create program object
hello_world = HelloWorld()
```

The program starts by creating an object from the program class.

```
class HelloWorld:
    # Define the initialize method
    def __init__(self):
        # Create the root window
        self.root = Tk()
```

After the object is created, we go to the **init** method. To initialize **tkinter**, we create a root window. This is an ordinary window, with a title bar and other decorations provided by your window manager. Create only one root window for each program. It must be created before any other widgets.

```
# Create a label
self.lbl_display = Label(self.root,
    text = "Hello, world! \nIt is time for Monty Python . . .")
```

We create a Label widget named **lbl_display** as a child to the root window:

A **Label** widget can display either text or an icon or other image. In this case, we use the text option to specify which text to display.

```
# Pack the label
# Size it to fit the text
# Make it visible
self.lbl_display.pack()
```

We call the **pack** method on this widget. This tells it to size itself to fit the given text and make itself visible. The window and other GUI elements won't appear until we've entered the **tkinter mainloop()** event loop:

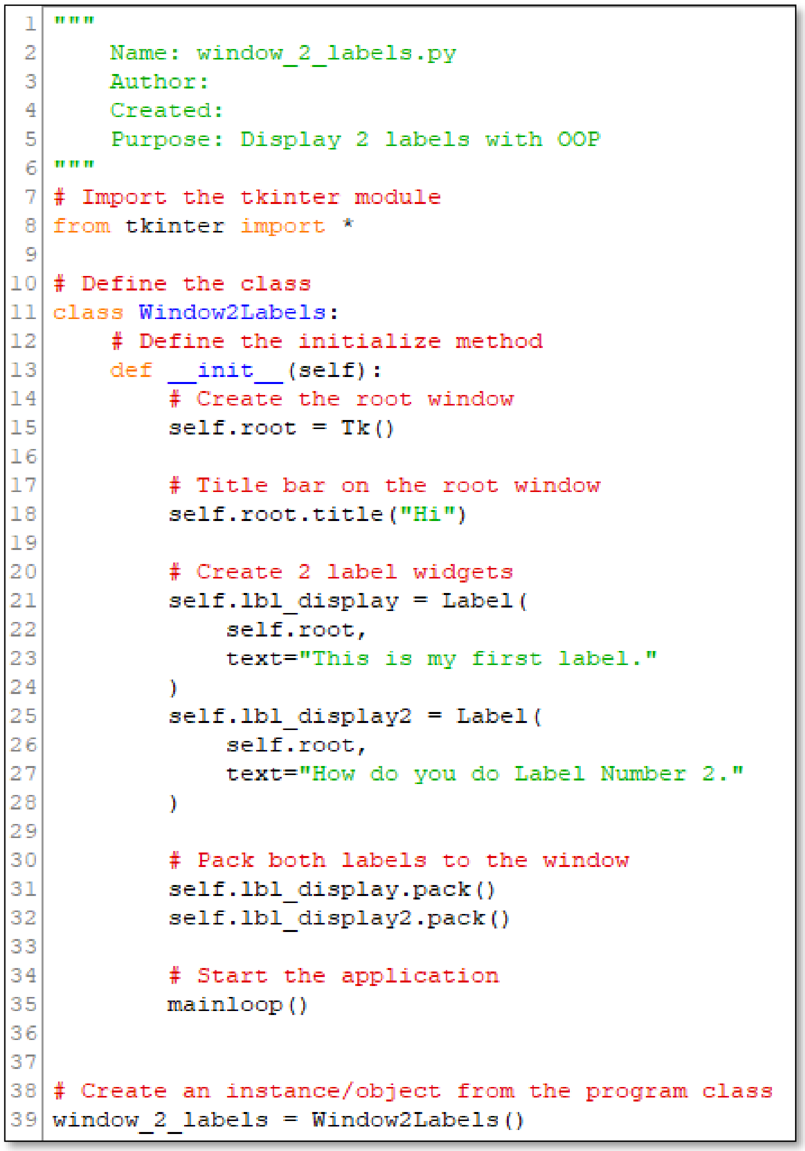
```
# Call the mainloop method which is used
# when the application is ready to run
# It tells the application to keep displaying the GUI
mainloop()
```

The program will stay in the event loop until we close the window. The event loop handles events from the user (such as mouse clicks and key presses) and the windowing system (such as redraw events and window configuration messages). It also handles operations queued by **tkinter** itself. Among these operations are geometry management (queued by the pack method) and display updates. This means the application window will not appear before you enter the main loop.

Tutorial 9.2 – Labels

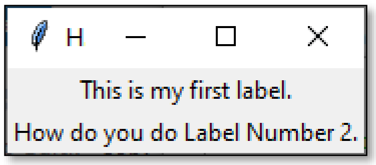
This program is very similar to the last program: we add a second label. The **pack** method must be run on each label and ensures that each label is the correct size and is visible.

NOTE: Some program examples will use window, some will use root. Either one is fine.



```python
"""
    Name: window_2_labels.py
    Author:
    Created:
    Purpose: Display 2 labels with OOP
"""
# Import the tkinter module
from tkinter import *

# Define the class
class Window2Labels:
    # Define the initialize method
    def __init__(self):
        # Create the root window
        self.root = Tk()

        # Title bar on the root window
        self.root.title("Hi")

        # Create 2 label widgets
        self.lbl_display = Label(
            self.root,
            text="This is my first label."
        )
        self.lbl_display2 = Label(
            self.root,
            text="How do you do Label Number 2."
        )

        # Pack both labels to the window
        self.lbl_display.pack()
        self.lbl_display2.pack()

        # Start the application
        mainloop()


# Create an instance/object from the program class
window_2_labels = Window2Labels()
```

Example run:

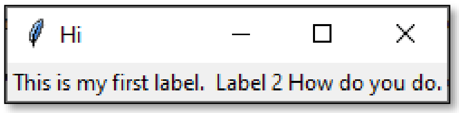


```
This is my first label.
How do you do Label Number 2.
```

We can change the alignment of our widgets by passing arguments to the **pack** method.

```
# Pack both labels aligned to their left side
# Size them to fit the text
# Make themselves visible
self.lbl_display.pack(side='left')
self.lbl_display2.pack(side='left')
```

Example run:



```
This is my first label.  Label 2 How do you do.
```

The other arguments for **side= top**, **bottom**, **left**, and **right**.

Labels

Label Options: There are several options you can change including font size and color. Here are some examples:

```
hello_label = Label(text='hello', font=('Verdana', 24, 'bold'),
                    bg='blue', fg='white')
```

Note the use of keyword arguments. Here are a few common options:

- **font**: The basic structure is **font=(font name, font size, style)**. You can leave out the font size or the style. The choices for style are 'bold', 'italic', 'underline', 'overstrike', 'roman', and 'normal' (which is the default). You can combine multiple styles like this: 'bold italic'.
- **foreground and background:** These stand for foreground and background colors. Many common color names can be used, like 'blue', 'green', etc.
- **width**: This is how many characters long the label should be. If you leave this out, Tkinter will base the width off the text you put in the label. This can make for

unpredictable results. It is good to decide ahead of time how long you want your label to be and set the width accordingly.

- **Height**: This is how many rows high the label should be. You can use this for multiline labels. Use newline characters in the text to get it to span multiple lines.

For example:

text='hi\nthere'

Example run:

Hi
there

Borders and Padding

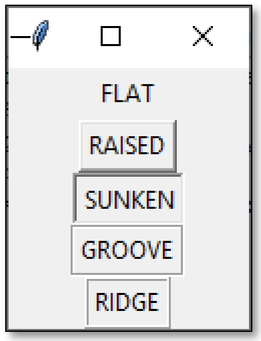
You can also display a border around a label and use internal and external padding. Each method can have more than one argument separated by commas.

Two border arguments are

borderwidth = 1, relief = ’solid’)

borderwidth can have an integer value which specifies the width in pixels.

relief can have **FLAT, RAISED, SUNKEN, RIDGE, SOLID, and GROOVE**.



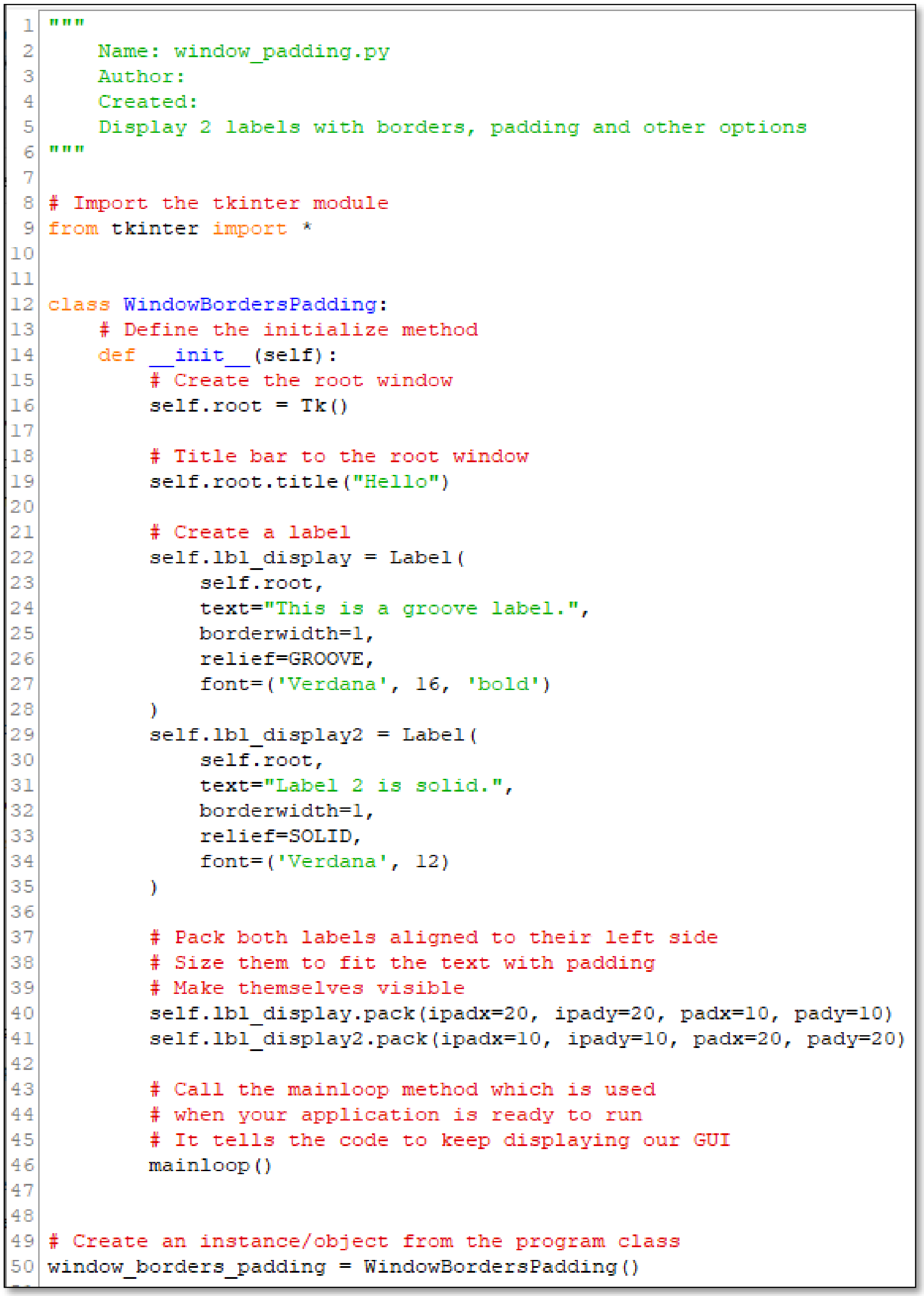
Label Options

anchor	It specifies the exact position of the text within the size provided to the widget. The default value is CENTER, which is used to center the text within the specified space.
bg	The background color displayed behind the widget.

bitmap	It is used to set the bitmap to the graphical object specified so that, the label can represent the graphics instead of text.
bd	It represents the width of the border. The default is 2 pixels.
cursor	The mouse pointer will be changed to the type of the cursor specified, i.e., arrow, dot, etc.
font	The font type of the text written inside the widget.
fg	The foreground color of the text written inside the widget.
height	The height of the widget.
image	The image that is to be shown as the label.
justify	It is used to represent the orientation of the text if the text contains multiple lines. It can be set to LEFT for left justification, RIGHT for right justification, and CENTER for center justification.
padx	The horizontal padding of the text. The default value is 1.
pady	The vertical padding of the text. The default value is 1.
relief	The type of the border. The default value is FLAT. Options are: GROOVE, SOLID, SUNKEN, RIDGE
text	This is set to the string variable which may contain one or more line of text.
textvariable	The text written inside the widget is set to the control variable StringVar so that it can be accessed and changed accordingly.
underline	We can display a line under the specified letter of the text. Set this option to the number of the letter under which the line will be displayed.
width	The width of the widget. It is specified as the number of characters.
wraplength	Instead of having only one line as the label text, we can break it to the number of lines where each line has the number of characters specified to this option.

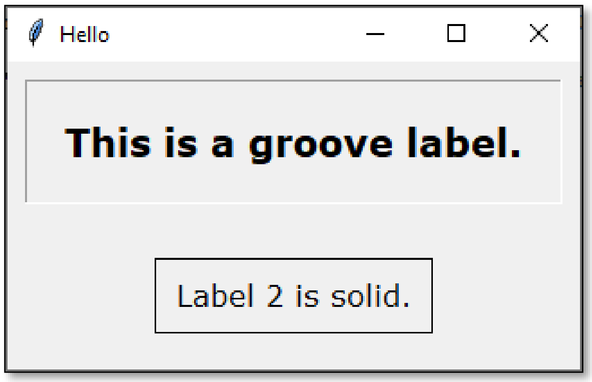
Tutorial 9.3 – Label Options

This tutorial combines some of the methods and properties we have been looking at.



```python
"""
    Name: window_padding.py
    Author:
    Created:
    Display 2 labels with borders, padding and other options
"""

# Import the tkinter module
from tkinter import *


class WindowBordersPadding:
    # Define the initialize method
    def __init__(self):
        # Create the root window
        self.root = Tk()

        # Title bar to the root window
        self.root.title("Hello")

        # Create a label
        self.lbl_display = Label(
            self.root,
            text="This is a groove label.",
            borderwidth=1,
            relief=GROOVE,
            font=('Verdana', 16, 'bold')
        )
        self.lbl_display2 = Label(
            self.root,
            text="Label 2 is solid.",
            borderwidth=1,
            relief=SOLID,
            font=('Verdana', 12)
        )

        # Pack both labels aligned to their left side
        # Size them to fit the text with padding
        # Make themselves visible
        self.lbl_display.pack(ipadx=20, ipady=20, padx=10, pady=10)
        self.lbl_display2.pack(ipadx=10, ipady=10, padx=20, pady=20)

        # Call the mainloop method which is used
        # when your application is ready to run
        # It tells the code to keep displaying our GUI
        mainloop()


# Create an instance/object from the program class
window_borders_padding = WindowBordersPadding()
```

Example run:



This is a groove label.

Label 2 is solid.

Changing Label Properties

After you've created a label, you may want to change something about it. To do that, use the **configure** method. Here are two examples that change the properties of a **label** called **label**:

```
label.configure(text='Bye')
label.configure(bg='white', fg='black')
```

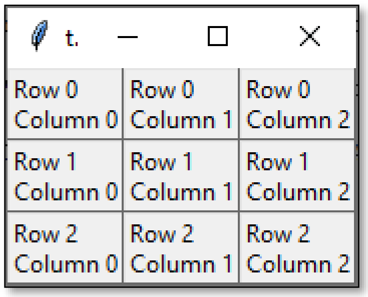
Setting text to something using the **configure** method is kind of like the GUI equivalent of a **print** statement. In calls to **configure** we cannot use commas to separate multiple things to print. We instead need to use **string** formatting. Here is a **print** statement and its equivalent using the **configure** method.

```
print('a =', a, 'and b =', b)
label.configure(text='a = {}, and b = {}'.format(a,b))
```

Grid Layout Manager

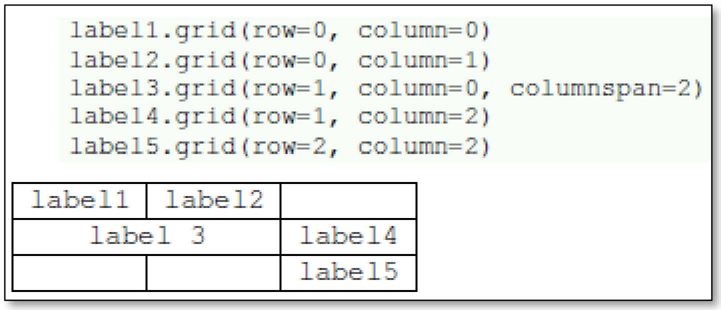
The **grid** method is used to place things on the screen. It lays out the screen as a rectangular grid of rows and columns. The first few rows and columns are shown below.

(row=0, column=0)	(row=0, column=1)	(row=0, column=2)
(row=1, column=0)	(row=1, column=1)	(row=1, column=2)
(row=2, column=0)	(row=2, column=1)	(row=2, column=2)



Row 0 Column 0	Row 0 Column 1	Row 0 Column 2
Row 1 Column 0	Row 1 Column 1	Row 1 Column 2
Row 2 Column 0	Row 2 Column 1	Row 2 Column 2

With multiple rows or columns there are optional arguments, **rowspan** and **columnspan**, that allow a widget to take up more than one row or column. Here is an example of several grid statements followed by what the layout will look like:



```
label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
label3.grid(row=1, column=0, columnspan=2)
label4.grid(row=1, column=2)
label5.grid(row=2, column=2)
```

label1	label2	
label 3		label4
		label5

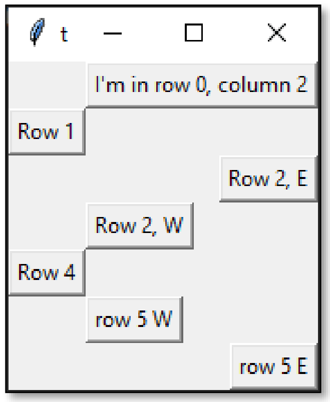
Spacing: To add extra space between widgets, there are optional arguments as follows.

- **ipadx:** How many pixels to pad widget horizontally inside the widget's borders.
- **ipady:** How many pixels to pad widget vertically inside the widget's borders.
- **padx:** How many pixels to pad widget horizontally outside the widget's borders.
- **pady:** How many pixels to pad widget vertically outside the widget's borders.

Sticky: When the widget is smaller than the cell, sticky is used to indicate which sides and corners of the cell the widget sticks to. The direction is defined by compass directions: N, E, S, W, NE, NW, SE, and SW and zero. These could be a string concatenation, for example, NESW make the widget take up the full area of the cell.

```
from tkinter import *

def main():

    root = Tk()
    btn_column = Button(root, text="I'm in row 0, column 2")
    btn_column.grid(row=0, column=1)

    btn_columnspan = Button(root, text="Row 1")
    btn_columnspan.grid(row=1, column=0)

    btn_pady = Button(root, text="Row 2, E")
    btn_pady.grid(row=2, column=1, sticky=E)

    btn_row = Button(root, text="Row 2, W")
    btn_row.grid(row=3, column=1, sticky=W)

    btn_rowspan = Button(root, text="Row 4")
    btn_rowspan.grid(row=4, column=0)

    btn_sticky = Button(root, text="row 5 W")
    btn_sticky.grid(row=5, column=1, sticky=W)

    btn_sticky1 = Button(root, text="row 5 E")
    btn_sticky1.grid(row=6, column=1, sticky=E)

    root.mainloop()
main()
```

Example run:



Important Note: Any time you create a widget, to place it on the screen you need to use **grid** (or one of its cousins, like **pack**). Otherwise it will not be visible.

Entry Boxes

Entry boxes are a way for your GUI to get text input from the user. The following example creates a simple entry box and places it on the screen.

```
entry = Entry()
entry.grid(row=0, column=0)
```

ost of the same options that work with labels work with entry boxes (and most of the other widgets we will talk about). The width option is particularly helpful because the entry box will often be wider than you need.

Syntax

```
w = Entry(master, option=value, ... )
```

Entry Box Options

bg	The normal background color displayed behind the label and indicator.
bd	The size of the border around the indicator. Default is 2 pixels.
cursor	If you set this option to a cursor name (*arrow, dot etc.*), the mouse cursor will change to that pattern when it is over the widget.
exportselection	By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0.
fg	The color used to render the text.
font	The font used for the text.
highlightcolor	The color of the focus highlight when the widget has the focus.
justify	If the text contains multiple lines, this option controls how the text is justified: CENTER, LEFT, or RIGHT.

relief	With the default value, relief=FLAT, the checkbutton does not stand out from its background. You may set this option to any of the other styles like GROOVE, RAISED, RIGID.
selectbackground	The background color to use displaying selected text.
Selectborderwidth	The width of the border to use around selected text. The default is one pixel.
selectforeground	The foreground (text) color of selected text.
show	Normally, the characters that the user types appear in the entry. To make a password entry that echoes each character as an asterisk, set show="*".
state	The default is state=NORMAL, but you can use state=DISABLED to gray out the control and make it unresponsive.
textvariable	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
xscrollcommand	If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar.

Entry Box Methods

delete (first, last=None)	Deletes characters from the widget, starting with the one at index first, up to but not including the character at position last. If the second argument is omitted, only the single character at position first is deleted.
get()	Returns the entry's current text as a string.

icursor(index)	Set the insertion cursor just before the character at the given index.
index(index)	Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry.
insert(index, s)	Inserts string s before the character at the given index.
select_adjust (index)	This method is used to make sure that the selection includes the character at the specified index.
select_clear()	Clears the selection. If there isn't currently a selection, has no effect.
select_from (index)	Sets the ANCHOR index position to the character selected by index, and selects that character.
select_present()	If there is a selection, returns true, else returns false.
select_range (start, end)	Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position.
select_to(index)	Selects all the text from the ANCHOR position up to but not including the character at the given index.
xview (index)	This method is useful in linking the Entry widget to a horizontal scrollbar.
xview_scroll (number, what)	Used to scroll the entry horizontally. The what argument must be either UNITS, to scroll by character widths, or PAGES, to scroll by chunks the size of the entry widget. The number is positive to scroll left to right, negative to scroll right to left.

Getting Text

To get the text from an entry box, use the **get** method. This will return a string. If you need numerical data, use **int** or **float** to cast the string. Here is a simple example that gets text from an entry box named **entry**.

```
string_value = entry.get()
num_value = int(entry.get())
```

Deleting text: To clear an entry box, use the following:

```
entry.delete(0, END)
```

Inserting text: To insert text into an entry box, use the following:

```
entry.insert(0, 'hello')
```

Buttons

The Button widget is used to add buttons in a Python application. These buttons can display text or images that convey the purpose of the buttons. You can attach a function or a method to a button which is called automatically when you click the button.

Syntax to create this widget.

```
w = Button(master, option=value, ... )
```

The following example creates a simple button:

```
btn_ok = Button(text='Ok')
```

The follow options can be used with the Python Button object.

activebackground	Background color when the button is under the cursor.
activeforeground	Foreground color when the button is under the cursor.
bd	Border width in pixels. Default is 2.
bg	Normal background color.
command	Function or method to be called when the button is clicked.
fg	Normal foreground (text) color.
font	Text font to be used for the button's label.
height	Height of the button in text lines (for textual buttons) or pixels (for images).

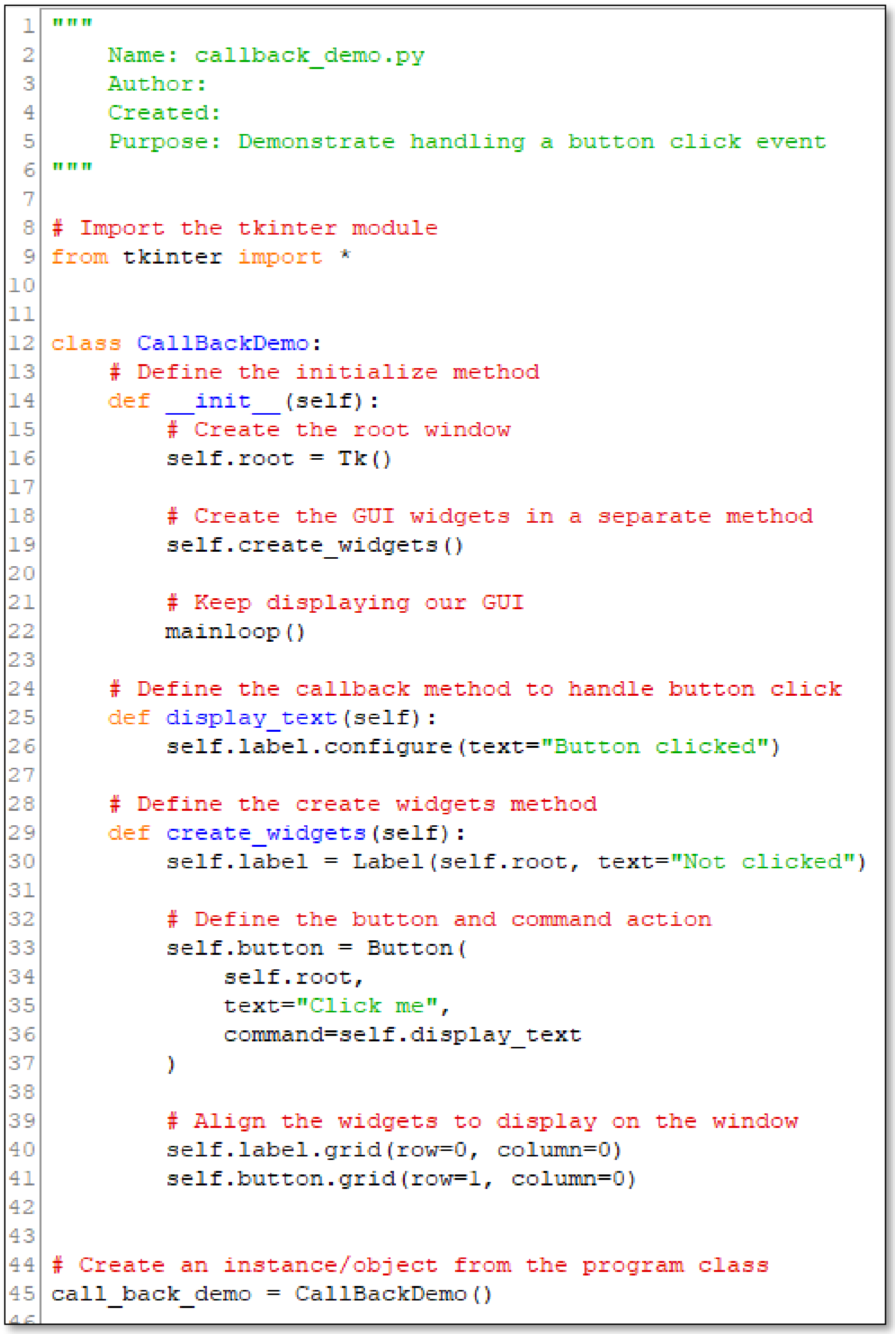
highlightcolor	The color of the focus highlight when the widget has focus.
image	Image to be displayed on the button (instead of text).
justify	How to show multiple text lines: LEFT to left-justify each line; CENTER to center them; or RIGHT to right-justify.
padx	Additional padding left and right of the text.
pady	Additional padding above and below the text.
relief	Relief specifies the type of the border. Some of the values are SUNKEN, RAISED, GROOVE, and RIDGE.
state	Set this option to DISABLED to gray out the button and make it unresponsive. Has the value ACTIVE when the mouse is over it. Default is NORMAL.
underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined.
width	Width of the button in letters (if displaying text) or pixels (if displaying an image).
wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length.

To get the button to do something when clicked, use the **command** argument. It is set to the name of a method, called a **callback** method. When the button is clicked, the **callback** method is called.

Tutorial 9.4 – Buttons

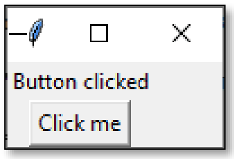
Create and save the following program as **callback_demo.py**

When the program starts, the label says **Click me**. When the button is clicked, the **callback** method is called, which changes the label to say Button clicked. A **callback** method can be named anything, we just choose callback for this example. If you have other buttons, you can create other methods for them.



```python
"""
    Name: callback_demo.py
    Author:
    Created:
    Purpose: Demonstrate handling a button click event
"""

# Import the tkinter module
from tkinter import *


class CallBackDemo:
    # Define the initialize method
    def __init__(self):
        # Create the root window
        self.root = Tk()

        # Create the GUI widgets in a separate method
        self.create_widgets()

        # Keep displaying our GUI
        mainloop()

    # Define the callback method to handle button click
    def display_text(self):
        self.label.configure(text="Button clicked")

    # Define the create widgets method
    def create_widgets(self):
        self.label = Label(self.root, text="Not clicked")

        # Define the button and command action
        self.button = Button(
            self.root,
            text="Click me",
            command=self.display_text
        )

        # Align the widgets to display on the window
        self.label.grid(row=0, column=0)
        self.button.grid(row=1, column=0)


# Create an instance/object from the program class
call_back_demo = CallBackDemo()
```

Example run:



Colors

Tkinter defines many common color names, like 'yellow' and 'red'.

```
label = Label(text = 'Hi', background = 'blue')
```

There are color names built into **tkinter**. The **display_colors.py** attached to this assignment will display all the built-in color names. The following is an example.

```
label = Label(text = 'Hi', background = 'cyan')
```

Ttk Themed Widgets

The tkinter.ttk module provides access to the Tk themed widget set, added in 2007 with Tk 8.5. Ttk stands for Tk themed.

To override the basic Tk widgets, the Ttk import should follow the Tk import:

```
from tkinter import *
from tkinter.ttk import *
```

This code causes several tkinter.ttk widgets (Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale and Scrollbar) to automatically replace the Tk widgets.

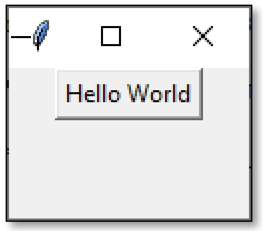
The new widgets give a better look and feel across platforms. Tthe replacement widget optioins are not completely compatible. The main difference is that widget options such as “fg”, “bg” and others related to widget styling are no longer present in Ttk widgets. Instead, you would use the ttk.Style class for improved styling effects.

Ttk Widgets

Ttk comes with 18 widgets, twelve of which already existed in tkinter: Button, Checkbutton, Entry, Frame, Label, LabelFrame, Menubutton, PanedWindow, Radiobutton, Scale, Scrollbar, and Spinbox. The other six are new: Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview.

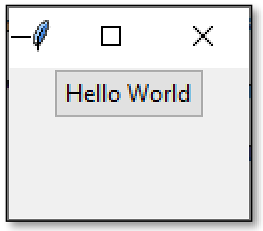
Using the Ttk widgets give the application an improved look and feel. There are differences in how the styling is coded.

Tk code for Hello World:



```
from tkinter import *
root = Tk()
Button(root, text="Hello World").grid()
mainloop()
```

Ttk code for Hello World:



```
from tkinter import *
from tkinter.ttk import *
root = Tk()
Button(root, text="Hello World").grid()
mainloop()
```

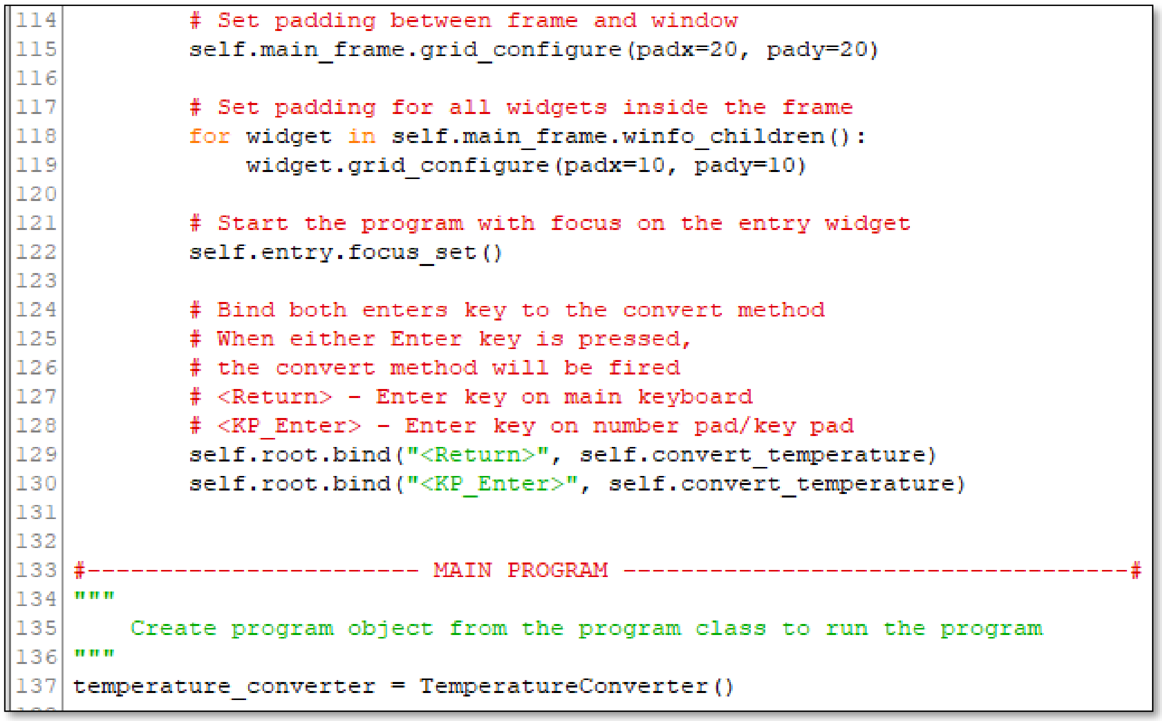
Tutorial 9.5 – Temperature Converter

Let’s create a tkinter ttk version of our old friend, the temperature converter.

Create a Python program named: **temperature_converter.py**

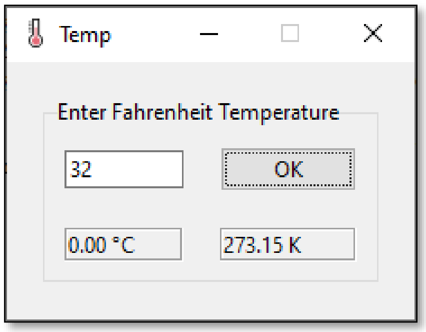
```python
"""
    Name: temperature_converter.py
    Author:
    Created:
    Purpose: Convert Fahrenheit to Celsius and Kelvin
"""
# Import the tkinter module with tk standard widgets
from tkinter import *
# Override tk widgets with themed ttk widgets if available
from tkinter.ttk import *


class TemperatureConverter:

    def __init__(self):
        """
            Define the object initialize method
        """
        # Create the root window
        self.root = Tk()
        self.root.title("Temp")

        # Add icon to window corner
        # Search the web for free thermometer.ico files
        self.root.iconbitmap("thermometer.ico")

        # Prevent window from resizing
        self.root.resizable(False, False)

        # Create the GUI widgets in a separate method
        self.create_widgets()

        # Call the mainloop method to start program
        mainloop()

#----------------------- CONVERT TEMPERATURE -----------------------------#
    def convert_temperature(self, *args):
        """
            Convert fahrenheit to celsius and kelvin
        """
        # Get input from user as a float
        self.__fahrenheit = float(self.entry.get())

        # Convert Fahrenheit to Celsius and Kelvin
        self.__celsius = ((self.__fahrenheit - 32) * 5.0) / 9.0
        self.__kelvin = (((self.__fahrenheit - 32) * 5.0) / 9.0) + 273.15

        # Display output in labels using the configure method
        self.lbl_celsius.configure(text=f"{self.__celsius:,.2f} °C")
        self.lbl_kelvin.configure(text=f"{self.__kelvin:,.2f} K")

        # 0 starts the selection at the beginning of the entry widget text
        # END finishes the selection at the end of the entry widget text
        self.entry.selection_range(0, END)
```

```python
#------------------------CREATE WIDGETS -----------------------------------#
    def create_widgets(self):
        """
            Create and grid widgets
        """
        # Create main label frame to hold widgets
        self.main_frame = LabelFrame(
            self.root,                            # Assign to parent window
            text="Enter Fahrenheit Temperature",  # Text for the frame
            relief=GROOVE                         # Decorative border
        )

        # Create entry widget in the frame to get input from user
        self.entry = Entry(
            self.main_frame,  # Assign to parent frame
            width=10          # Width in characters
        )

        # Create button in the frame to call calulate method
        self.btn_calculate = Button(
            self.main_frame,      # Assign to parent frame
            text="OK",            # Text shown on button
            # Connect convert method to button click
            command=self.convert_temperature
        )

        # Create label in the frame to show celsius
        self.lbl_celsius = Label(
            self.main_frame,      # Assign to parent frame
            width=10,             # Width in characters
            relief=GROOVE         # Decorative border
        )

        # Create label in the frame to show output
        self.lbl_kelvin = Label(
            self.main_frame,
            width=10,
            relief=GROOVE
        )

        # Use Grid layout manager to place widgets in the frame
        self.entry.grid(row=0, column=0)
        self.btn_calculate.grid(row=0, column=1)

        # sticky uses cardinal directions to stick labels to sides of the column
        # sticky=EW expands the label to fit the column
        # based on the largest widget
        self.lbl_celsius.grid(
            row=1,
            column=0,
            sticky=EW
        )
        self.lbl_kelvin.grid(
            row=1,
            column=1,
            sticky=EW
        )
```



```python
        # Set padding between frame and window
        self.main_frame.grid_configure(padx=20, pady=20)

        # Set padding for all widgets inside the frame
        for widget in self.main_frame.winfo_children():
            widget.grid_configure(padx=10, pady=10)

        # Start the program with focus on the entry widget
        self.entry.focus_set()

        # Bind both enters key to the convert method
        # When either Enter key is pressed,
        # the convert method will be fired
        # <Return> - Enter key on main keyboard
        # <KP_Enter> - Enter key on number pad/key pad
        self.root.bind("<Return>", self.convert_temperature)
        self.root.bind("<KP_Enter>", self.convert_temperature)


#----------------------- MAIN PROGRAM -----------------------------------#
"""
    Create program object from the program class to run the program
"""
temperature_converter = TemperatureConverter()
```

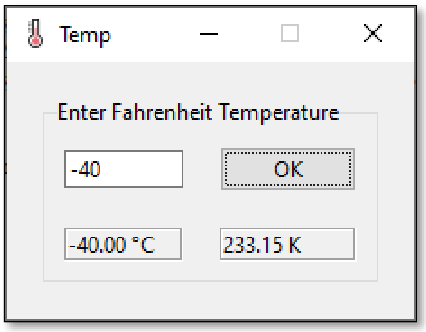
Example run:



32

0.00 °C

273.15 K



-40

-40.00 °C

233.15 K

Using tkinter, ttk, and Frames in Temperature Converter

In temperature_converter.py, we added ttk to tkinter and frames to control the layout.

Our program starts by incorporating Tk as we have done in the past.

```
from tkinter import *
from tkinter.ttk import *
```

These two lines tell Python that our program needs two modules. The first, `tkinter`, is the standard binding to Tk. When imported, it loads the Tk library on your system. The second, `ttk`, is a submodule of `tkinter`. It implements Python's binding to the newer "themed widgets" that were added to Tk in 8.5.

NOTE: One gotcha to watch out for, the newer ttk widgets do not always have the same options as the tk widgets.

Several widgets are defined in both modules. By importing everything from ttk, we don't have to know which is which. If there is not a newer ttk widget, the standard tkinter widget will be used.

Instead of placing the widgets inside the window, we placed them inside a nice label frame with a border. You can add more than one frame, allowing you to divide up the purpose of various parts of your program.

The other reason is that the main window isn't itself part of the newer "themed" widgets. Its background color doesn't match the themed widgets we will put inside it. Using a "themed" frame widget to hold the content ensures that the background color of the widget is correct.

Using Frames and Widgets with OOP

When using a frame to place our widgets, we need to do two things: create the widget itself and then place it onscreen.

When we create a widget, we need to specify its *parent*. That is the widget that the new widget will be placed inside. In this case, we want our entry placed inside the content frame. Our entry, and other widgets we'll create shortly, are said to be *children* of the content frame. In Python, the `parent` is passed as the first parameter when instantiating a widget object as shown below.

```
        # Create entry widget in the frame to get input from user
        self.entry = Entry(
            self.main_frame,
            text='',
            width=10)
```

When we create a widget, we can provide it with certain *configuration options*. In the example above, we specify how wide we want the entry to appear, i.e., 10 characters.

When widgets are created, they don't automatically appear on the screen; Tk doesn't know where you want them placed relative to other widgets. That's what the `grid` part does. Widgets are placed in the appropriate column (0 or 1) and row (also 0 or 1).

```
        # Use Grid layout manager to place widgets in the frame
        self.entry.grid(row=0, column=0)
        self.btn_calculate.grid(row=0, column=1)
        self.lbl_celsius.grid(row=1, column=0)
        self.lbl_kelvin.grid(row=1, column=1, sticky=E)
```

The `sticky` option to grid describes how the widget should line up within the grid cell, using compass directions. E (east) means to anchor the widget to the right side of the cell, WE (west-east) anchors the widget to both the left and right sides, and so on. Python defines constants for these directional strings, which you can provide as a list, e.g., `W` or `(W, E)`.

place() Layout Manager

NOTE: You cannot mix and match layout managers within a window or a frame. Each window or frame can have its own layout manager.

The **place()** layout manager organizes widgets by placing them in a specific position in the parent widget.

This geometry manager uses the options **anchor**, **bordermode**, **height**, **width**, **relheight**, **relwidth**, **relx**, **rely**, **x** and **y**.

Anchor: Indicates where the widget is anchored to. The options are compass directions: N, E, S, W, NE, NW, SE, or SW, which relate to the sides and corners of the parent widget. The default is NW upper left corner of widget.

bordermode: has two options: **inside**, which indicates that other options refer to the parent's inside, (Ignoring the parent's borders) and **outside**, which is the opposite.

height: Specify the height of a widget in pixels.

width: Specify the width of a widget in pixels.

relheight: Height as a float between 0.0 and 1.0, as a fraction of the height of the parent widget.

relwidth: Width as a float between 0.0 and 1.0, as a fraction of the width of the parent widget.

relx: Horizontal offset as a float between 0.0 and 1.0, as a fraction of the width of the parent widget.

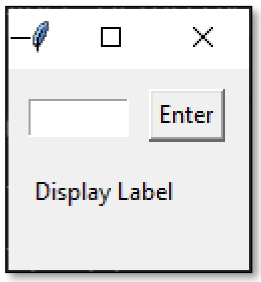
rely: Vertical offset as a float between 0.0 and 1.0, as a fraction of the height of the parent widget.

x: Horizontal offset in pixels.

x: Vertical offset in pixels.

```
from tkinter import *

def main():
    root = Tk()
    root.geometry("100x100")

    entry = Entry()
    entry.place(width=50, x=10, y=15)

    btn_enter = Button(root, text="Enter")
    btn_enter.place(x=70, y=10)

    lbl_label = Label(text="Display Label")
    lbl_label.place(x=10, y=50)

    root.mainloop()
main()
```

Example run:



Images

Labels and buttons (and other widgets) can display images instead of text.

To use an image requires a little set-up work. Create a **PhotoImage** object and give it a name. Here is an example:

```
cheetah_image = PhotoImage(file = 'cheetahs.gif')
```

Here are some examples of putting the image into widgets:

```
label = Label(image = cheetah_image)
button = Button(
    image = cheetah_image,
    command = cheetah_callback()
)
```

You can use the `configure` method to set or change an image:

```
label.configure(image = cheetah_image)
```

One unfortunate limitation of Tkinter is the only common image file type it can use is GIF. If you would like to use other types of files, a good solution is to use the Python Imaging Library.

Python Imaging Library

The Python Imaging Library (PIL) contains useful tools for working with images. There is a project called Pillow that it compatible with PIL and works in Python 3.0 and later.

Install Pillow

Open a command prompt. The installation should be successful.

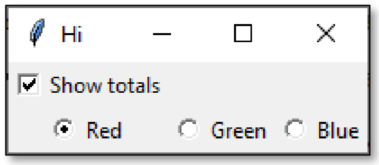
```
pip install Pillow
```

Using images other than GIFs with Tkinter: Tkinter can't use JPEGs and PNGs. But it can if we use it in conjunction with the PIL. Here is a simple example:

```
from tkinter import *
from PIL import Image, ImageTk
root = Tk()
cheetah_image = ImageTk.PhotoImage(Image.open("cheetah.jpg"))
button = Button(image = cheetah_image)
button.grid(row = 0, column=0)
mainloop()
```

The first line imports Tkinter. The next line imports a few things from the PIL. We load an image using a combination of two PIL functions. We can then use the image like normal in our widgets.

Check Buttons and Radio buttons



Check buttons are used when you want to turn something on or off. Radio buttons are used when you want the user to choose from a range of choices. In the program above, the top line shows a check button and the bottom line shows a radio button.

Check Buttons

The code for the above check button is:

```
show_totals = IntVar()
check = Checkbutton(
    text = "Show totals",
    var = show_totals
)
```

Note that we tie the check button to a variable. It can't be just any variable, it has to be a special kind of Tkinter variable, called an `IntVar`. This variable, `show_totals`, will be 0 when the check button is unchecked and 1 when it is checked. To access the value of the variable, you need to use it's get method, like this:

```
show_totals.get()
```

You can also set the value of the variable using its set method. This will automatically check or uncheck the check button on the screen. For instance, if you want the above check button checked at the start of the program, do the following:

```
show_totals = IntVar()
show_totals.set(1)
check = Checkbutton(
    text = "Show totals",
    var = show_totals
)
```

Radio Buttons

Radio buttons are useful when you want the user to choose from a range of choices. Only one radio button can be selected at a time, they are considered mutually exclusive,

The code for the radio buttons shown at the start of the section is:

```
color = IntVar()
rdo_redbutton = Radiobutton(
    text = "Red",
    var = color,
    value = 1
)
rdo_greenbutton = Radiobutton(
    text = "Green",
    var = color,
    value = 2
)
rdo_bluebutton = Radiobutton(
    text = "Blue",
    var = color,
    value = 3
)
```

Commands

Both check buttons and radio buttons have a command option, where you can set a callback function to run whenever the button is selected or unselected.

Assignment Submission

Attach all program files to the assignment in BlackBoard.