Chapter 9: Python GUI

Contents

Red light: No AI

Time required: 120 minutes

DRY

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

Tutorials

- 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. Mouse clicks or keyboard events are registered: the program responds to that event with the appropriate action.

Graphical 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.

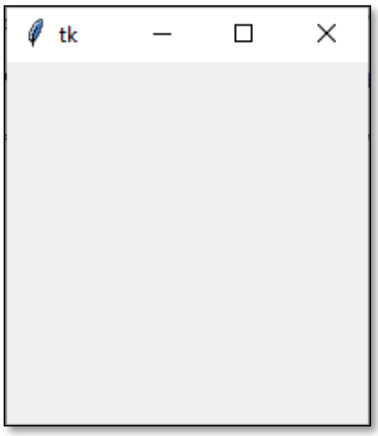
There are a couple approaches to using **tkinter**. This chapter will take the object-oriented approach, as is more flexible and reusable. Although tktinter code can be written using only functions, it's much better to use a class to keep track of all individual widgets which may need to reference each 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.

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: Hello Tkinter World!

Let's start out with the minimum code needed to create an empty window.

```
"""
    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()
```



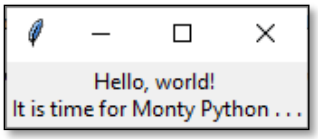
Not very impressive, is it. The code behind it isn't very complicated either. Let's modify this program into the traditional **Hello World** program. We create a root window, put a title in the title bar, and display a single label.

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

Create a python program named **hello_world.py**

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


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, pack is a window/layout manager
        # Pack makes the widget visible in the window
        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:



How It Works

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

This line imports the **tkinter** module. It contains all classes, functions and other items needed to work with the **Tk** GUI 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 dunder **__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, 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. The application window will not appear before you enter the main loop.

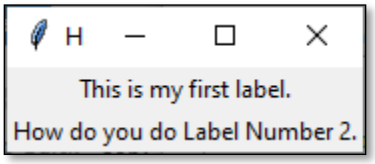
Tutorial 9.2: Labels

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. window and root are conventions.

```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:



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:

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:

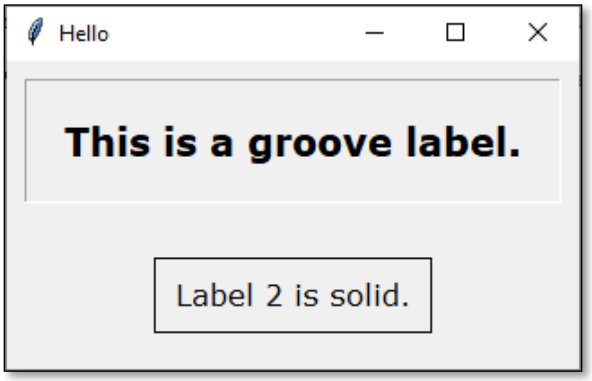
Hi
there

Tutorial 9.3: Label Options

This tutorial combines some of the methods and properties that labels can have.

```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 layout manager displays widgets in the window
        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 runs the program
        mainloop()


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

Example run:



Changing Label Properties

After you've created a label or any other widget, 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.

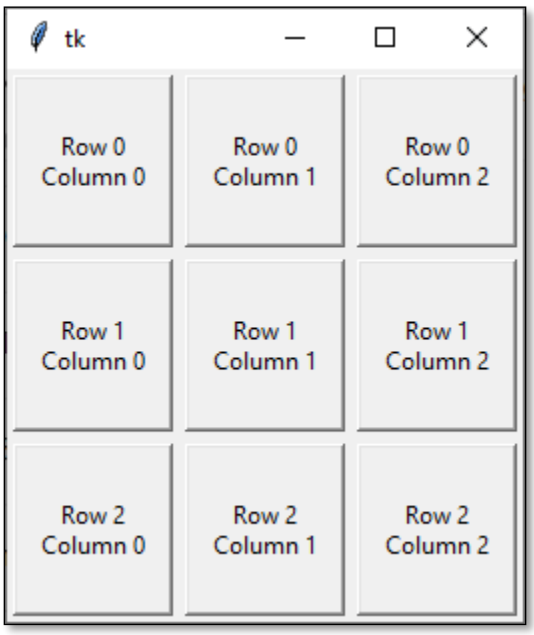
```
print('a =', a, 'and b =', b)
label.configure(text='a = {}, and b = {}'.format(a,b))
```

Grid Layout Manager

Almost all the programs you use are laid out in a grid. The Tkinter **grid** method is used to place things on the screen as a rectangular grid of rows and columns. The first few rows and columns are shown below.

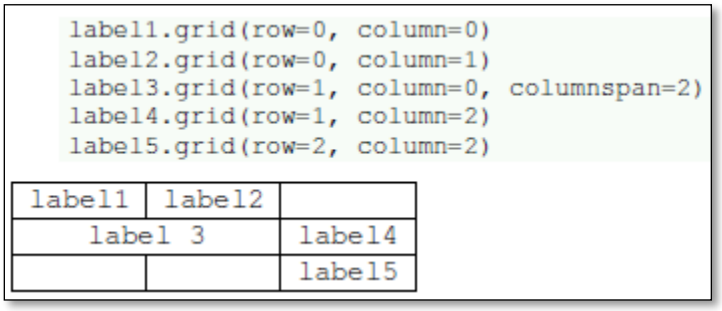
(row=0, column=0)	(row=0, column=1)	(row=0, column=2)
(row=1, column=0)	(row=1, column=1)	(row=1, column=2)
(row=2, column=0)	(row=2, column=1)	(row=2, column=2)

```
from tkinter import *
# Create the Tkinter Window
root = Tk()

for i in range(3):
    root.columnconfigure(i, weight=1)
    root.rowconfigure(i, weight=1)

    for j in range(3):
        frame = Frame(
            master=root,

            borderwidth=1
        )
        frame.grid(row=i, column=j, padx=1, pady=1)
        button = Button(
            master=frame,
            text=f"Row {i}\nColumn {j}",
            height=5,
            width=10
        )
        button.pack(padx=1, pady=1)
root.mainloop()
```



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)
```



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.

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)
```

Most 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, ... )
```

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')
```

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, it calls back the **callback** method.

Tutorial 9.4: Data Entry Program

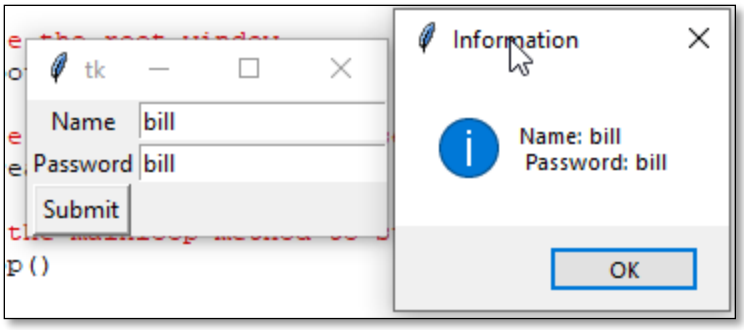
Example entry program.

```
"""
    Name: entry_demo.py
    Author:
    Created:
    Purpose: Demonstrate data entry and display
"""
from tkinter import *
from tkinter import messagebox
```

```
class EntryDemo:
    def __init__(self):
        self.root = Tk()

        # Create and layout all widgets
        self.lbl_name = Label(self.root, text="Name")
        self.lbl_name.grid(row=0, column=0)

        self.entry_name = Entry(self.root)
        self.entry_name.grid(row=0, column=1)

        self.lbl_password = Label(self.root, text="Password")
        self.lbl_password.grid(row=1, column=0)

        self.entry_password = Entry(self.root)
        self.entry_password.grid(row=1, column=1)

        self.btn_submit = Button(
            self.root,
            text="Submit",
            command=self.get_entry,
        )
        self.btn_submit.grid(row=4, column=0)

        mainloop()
```

```python
    # ---------------------------- GET ENTRY ---------------------------------#
    def get_entry(self):
        """Define a callback method called get_entry within a class"""
        # Retrieve and store the text content of entry widgets
        # named entry_name and entry_password.
        name = self.entry_name.get()
        password = self.entry_password.get()

        # Display an information message box containing
        # the retrieved name and password.
        messagebox.showinfo(
            "Information",
            f"Name: {name}\n Password: {password}"
        )


entry_demo = EntryDemo()
```

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.

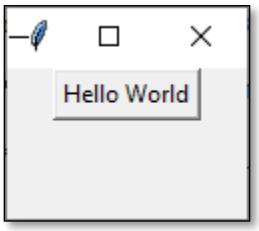
To override the basic Tk widgets, the Ttk import should follow the Tk import:

```
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.

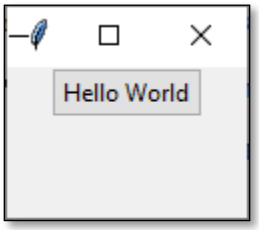
Using the Ttk widgets give the application an improved look and feel. There are differences in how the styling is coded.

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.

Find a free thermometer icon. The extension for Windows icon files is ico.

NOTE: If you have a Mac, do not try to add ico files. They are Windows only.

Go to IconArchive for free ico files.

If you wish to put in a degree symbol,:

1. https://www.degreesymbol.net/
2. Click the Copy button.
3. Paste in your code. °

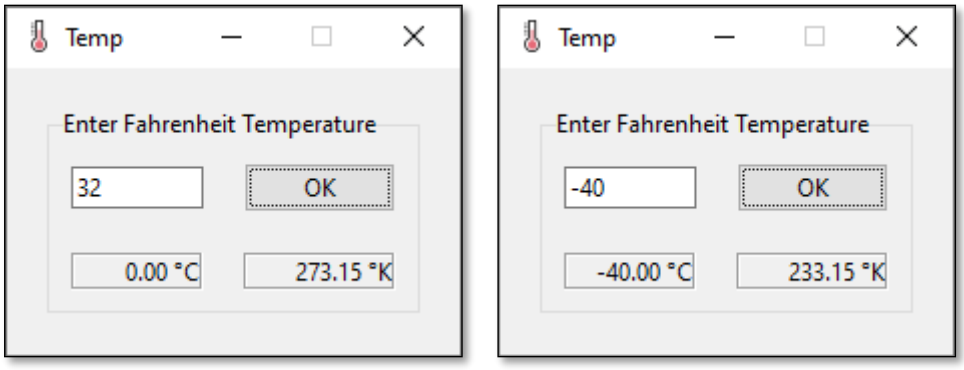
Create a Python program named: **temperature_converter_gui.py**

```python
"""
    Name: temperature_converter_gui.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):
        # 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):
        """Method called by the button click event
           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
            anchor=E             # Anchor text to the East
        )

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

```
        # 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
        )

        # 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 enter 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:



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 import two built-in Python modules. **tkinter** is the standard Python binding to the Tk GUI library. When imported, it loads the Tk library on your system. **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. For example, you may not be able to use color on an individual widget.

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.

```
50          # Create entry widget in the frame to get input from user
51          self.entry = Entry(
52              self.main_frame,
53              text='',
54              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).

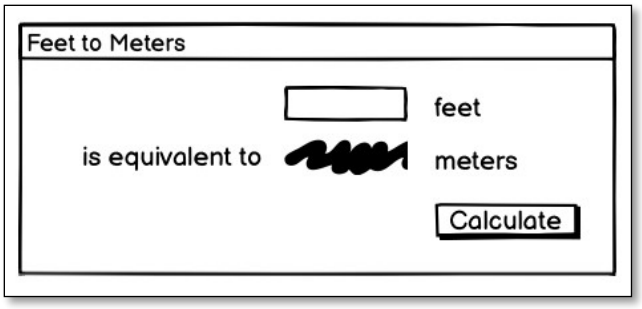
```
77          # Use Grid layout manager to place widgets in the frame
78          self.entry.grid(row=0, column=0)
79          self.btn_calculate.grid(row=0, column=1)
80          self.lbl_celsius.grid(row=1, column=0)
81          self.lbl_kelvin.grid(row=1, column=1, sticky=E)
```

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)`.

GUI Design with Pencil

Say we want to create a GUI for a meters to feet calculator.

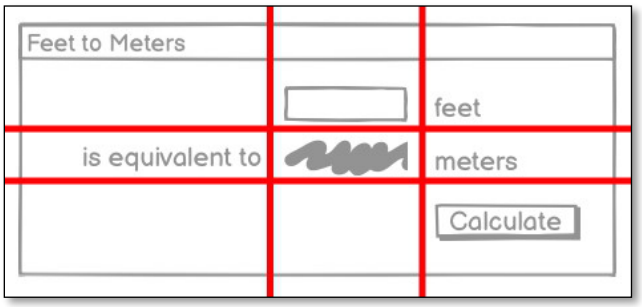
Sketch your interface on paper. It might look something like this:



Here are the widgets we will need.

1. Text entry to type in the number of feet.
2. Calculate button will get the value out of that entry, perform the calculation, and put the result in a label below the entry.
3. Three static labels ("feet," "is equivalent to," and "meters"), which help our user figure out how to run the application.

Let's look at the layout. The grid layout is very flexible for this type of GUI design. The widgets we've included seem to be naturally divided into a grid with three columns and three rows. In terms of layout, things seem to naturally divide into three columns and three rows, as illustrated below:



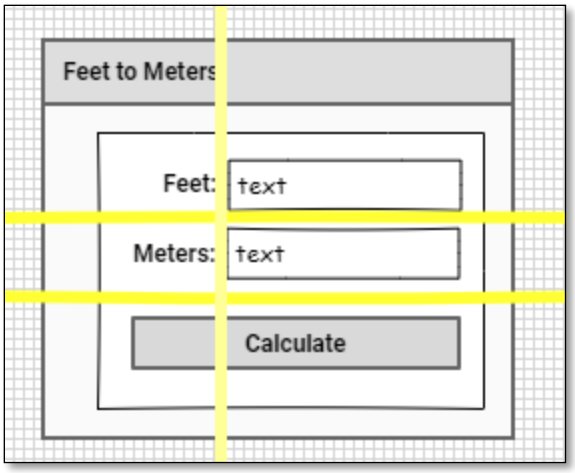
The layout of our user interface with a 3 x 3 grid.

Pencil

Pencil is a free and open-source GUI prototyping tool that people can easily install and use to create mockups in popular desktop platforms. This type of design is called a wireframe.

1. Go to https://pencil.evolus.vn/
2. Download and install Pencil for your OS.

Example wireframe of our evolving GUI:



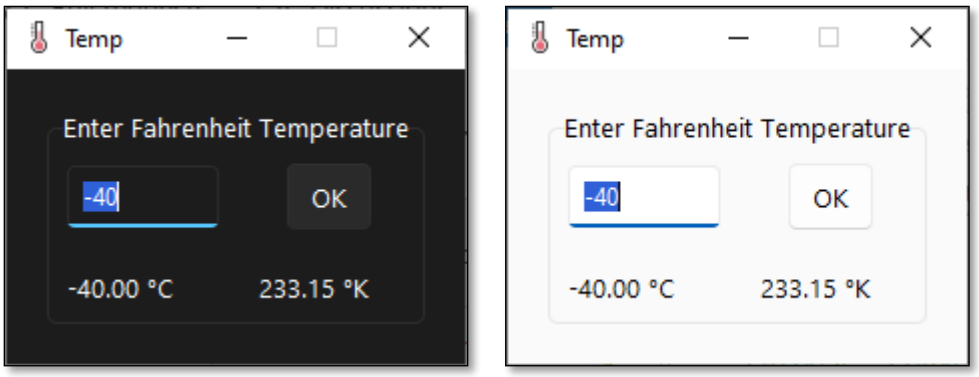
This example uses a grid layout. 2 columns, 3 rows inside a frame.

Tutorial 9.6: Temperature with Sun Valley Theme

Tkinter isn't a very modern looking interface. One of the most interesting things about Python is how many open-source libraries can be added to Python to solve almost any problem.

This Tkinter themed example is created from rdbende's Sun Valley ttk theme library.

https://github.com/rdbende/Sun-Valley-ttk-theme



Temperature Converter Theme Code Changes

Save **temperature_converter.py** as **temperature_converter_theme.py**

```
"""
    Name: temperature_converter_theme.py
    Author:
    Created:
    Purpose: Convert Fahrenheit to Celsius and Kelvin
    https://github.com/rdbende/Sun-Valley-ttk-theme
"""
# Import the tkinter module with tk standard widgets
from tkinter import *
# Override tk widgets with themed ttk widgets if available
from tkinter.ttk import *
# pip install sv-ttk
# Import Sun Valley Theme
import sv_ttk
```

1. Install the sv-ttk library: **pip install sv-ttk**
2. Import the new library: **import sv_ttk**

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

        # Set the theme to light or dark
        sv_ttk.set_theme("light")

        # Call the mainloop method to start program
        mainloop()
```

3. Set the theme to light or dark as shown. The above example is light.

Tutorial 9.7: ttkbootstrap

This tutorial gives you a nice framework to convert your other CLI API programs to a GUI.

ttkbootstrap is a supercharged theme extension for tkinter that enables on-demand modern flat style themes. https://ttkbootstrap.readthedocs.io/en/latest/

ttkbootstrap is used almost exactly like Tkinter, the only difference is a slight change in the __init__ method. It has pre-defined themes and styles, new widgets, and a theme creator. If you want a normal Tkinter interface, replace the ttkbootstrap import with Tkinter.

Try some of the other themes. This example uses darkly.

https://ttkbootstrap.readthedocs.io/en/latest/themes/

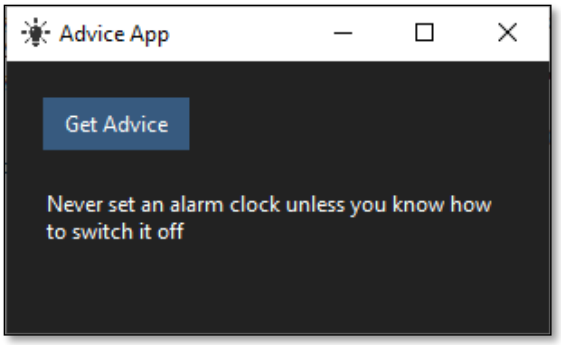
Install **ttkbootstrap**.

```
pip install ttkbootstrap
# Demo styles program: python -m ttkbootstrap
```

```python
"""
    Name: advice_gui.py
    Author: William Loring
    Created: 01/07/2023
    Purpose: Get random advice from API using ttkbootstrap
"""
# pip install ttkbootstrap
# Override tk widgets with ttkbootstrap
import ttkbootstrap as ttk
from ttkbootstrap.constants import *
import requests

# Set this to False to only display the result
IS_DEBUGGING = False

# URL for single random advice
URL = "https://api.adviceslip.com/advice"


class AdviceGUI:
    def __init__(self):
        self.root = ttk.Window(themename="darkly")
        self.root.title("Advice App")
        self.root.minsize(width=300, height=150)
        # Padding to edge of window
        self.root.config(padx=10, pady=10)
        self.root.iconbitmap("idea.ico")
        self.create_widgets()
        self.root.mainloop()

# ------------------------------ CREATE WIDGETS -------------------------#
    def create_widgets(self):
        self.btn_get_advice = ttk.Button(
            text="Get Advice",
            command=self.get_data
        )
        self.lbl_advice = ttk.Label(
            wraplength=250,
            justify="left",
        )
        PAD = 10
        self.btn_get_advice.grid(row=0, column=0, padx=PAD, pady=PAD, sticky=W)
        self.lbl_advice.grid(row=1, column=0, padx=PAD, pady=PAD, sticky=W)

        # The enter key will activate the calculate method
        self.root.bind("<Return>", self.get_data)
        self.root.bind("<KP_Enter>", self.get_data)
        self.root.bind("<Escape>", self.close_window)
```

```python
# ------------------------------ GET DATA -------------------------------#
    def get_data(self, *args):
        # Use the requests.get() function to get data from API
        response = requests.get(URL)
        # Watch for exception
        response.raise_for_status()

        # If the status_code is 200, successful connection and data
        if (response.status_code == 200):

            # Convert JSON data into Python dictionary with key value pairs
            self._advice_data = response.json()

            # Used to debug process
            if (IS_DEBUGGING == True):

                # Display the status code
                print(
                    f"\nStatus code: {response.status_code} \n")

                # Display the raw JSON data from the API
                print("The raw data from the API:")
                print(response.text)

                # Display the Python dictionary
                print("\nThe JSON data converted to a Python dictionary:")
                print(self._advice_data)

            # Display API data
            self.lbl_advice.config(
                text=self._advice_data.get("slip")["advice"])

        else:
            self.lbl_advice.config(
                text=f"API Unavailable: {response.status_code}")

# ------------------------------ CLOSE WINDOW ---------------------------#
    def close_window(self, *args):
        self.root.destroy()


advice_gui = AdviceGUI()
```

Example run:



Assignment 1: CustomTkinter and AI

 AI can be used as a code helper. Include the prompt and results.

CustomTkinter is a popular UI library for Tkinter. It makes very modern looking apps.

I suggest going through the following tutorial to gain an understanding of how CustomTkinter works.

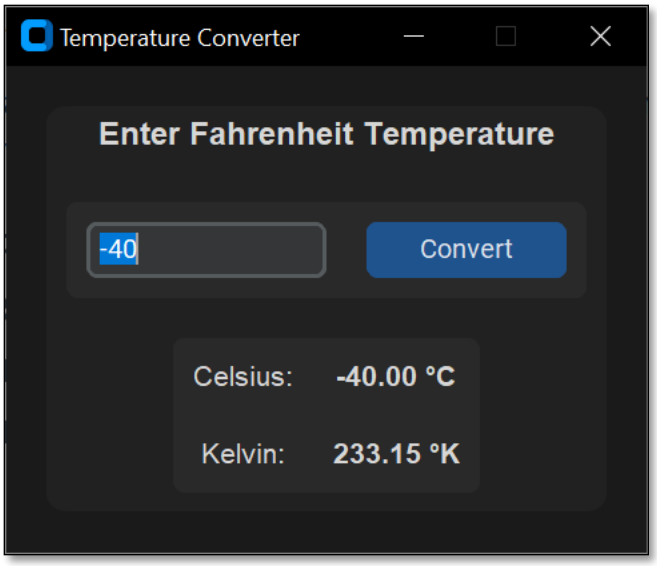
https://customtkinter.tomschimansky.com/tutorial/

This is what Claude.ai gave me when I asked it to convert our Temperature Converter to CustomTkinter. I like this particular appearance_mode and color theme. There are several to choose from. Give it a try with one of your Tkinter or CLI programs.

```
# Set appearance mode and default color theme
ctk.set_appearance_mode("dark")
ctk.set_default_color_theme("dark-blue")
```

A word of advice about using AI: Never use AI to do something that you don't already know how to do.

Requirements: Use an AI of your choice to convert your temperature converter into CustomTkinter.



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.

Install Pillow

Open a command prompt.

```
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("./assets/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.

Assignment Submission

1. Attach the pseudocode or create a TODO.
2. Attach all tutorials and assignments.
3. Attach your AI prompts for the last assignment.
4. Include a screenshot of a CustomTkinter program you converted or built.
5. Attach screenshots showing the successful operation of each tutorial program.
6. Submit in Blackboard.