Graphical user interface programming

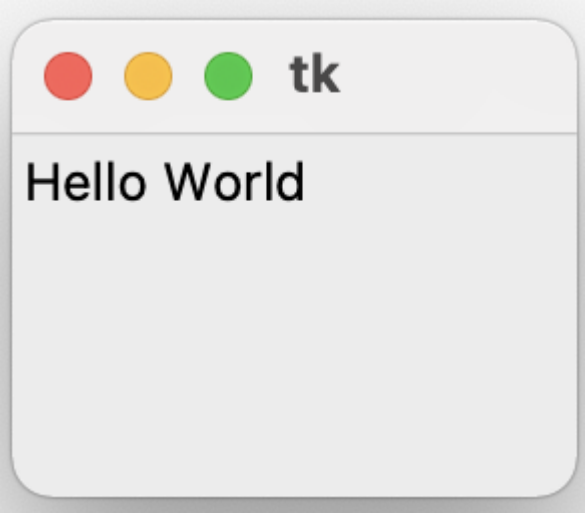
Graphical User Interface (GUI) programming is very different from the kind of programming we've been discussing so far. We might call the programming we've done so far **Sequential Driven Programming**. In this style of programming, the computer does something, then does something else, perhaps waiting for the user to type something in, or getting input from somewhere else, but essentially it proceeds in a sequential fashion. GUI programs are different because they are usually **Event Driven**. In this sort of program, there is usually a lot more user interaction, and the program almost always has to do something (redrawing itself on the screen if nothing else).

Tkinter introduction

TK (TK interface) is Python's dafault GUI toolkit, although there are many others. It is reasonably simple and lightweight and should be perfectly adequate for GUI programs of small to medium complexity. Let's have a look at a simple TKinter program:

```
from tkinter import *

class HelloWorld:
    """
    A Hello World App.
    """
    def __init__(self, parent):
        self.label = Label(parent, text="Hello World")
        self.label.grid(column=0, row=0)
# create the root window
root = Tk()
# create the application
app = HelloWorld(root)
# enter the main loop
root.mainloop()
```

If you run the above program, you should see a window which looks like Figure 1. (Note: please save the script as helloWorld.py and run it in the terminal by `python3 helloWorld.py`. Don't run the script in Spyder console since it sometimes conflict with tkinter.)



This structure of the program above is typical of a Tkinter program and has the following general steps:

1. define a class which encapsulates the user interface components of the application:
    1. create the specific user interface components
    2. specify where in the interface the elements should go (by calling the grid method)
2. create the root window (root=Tk())
3. instantiate a member of the application class
4. enter the main loop.

Note that root and label are objects of class Tk and Lable respectively. These classes are defined by Tkinter.

One piece of notation you should be familiar with is the term **widget**. A widget is a GUI element. Every GUI element can be called a widget. Tkinter has several different types of widgets. The most useful ones are:

- Tk: the root widget,
- Label: a widget containing fixed text,
- Button: a button widget,
- Entry: a widget for entry of single lines of text,
- Canvas: a widget for drawing shapes onto screen.

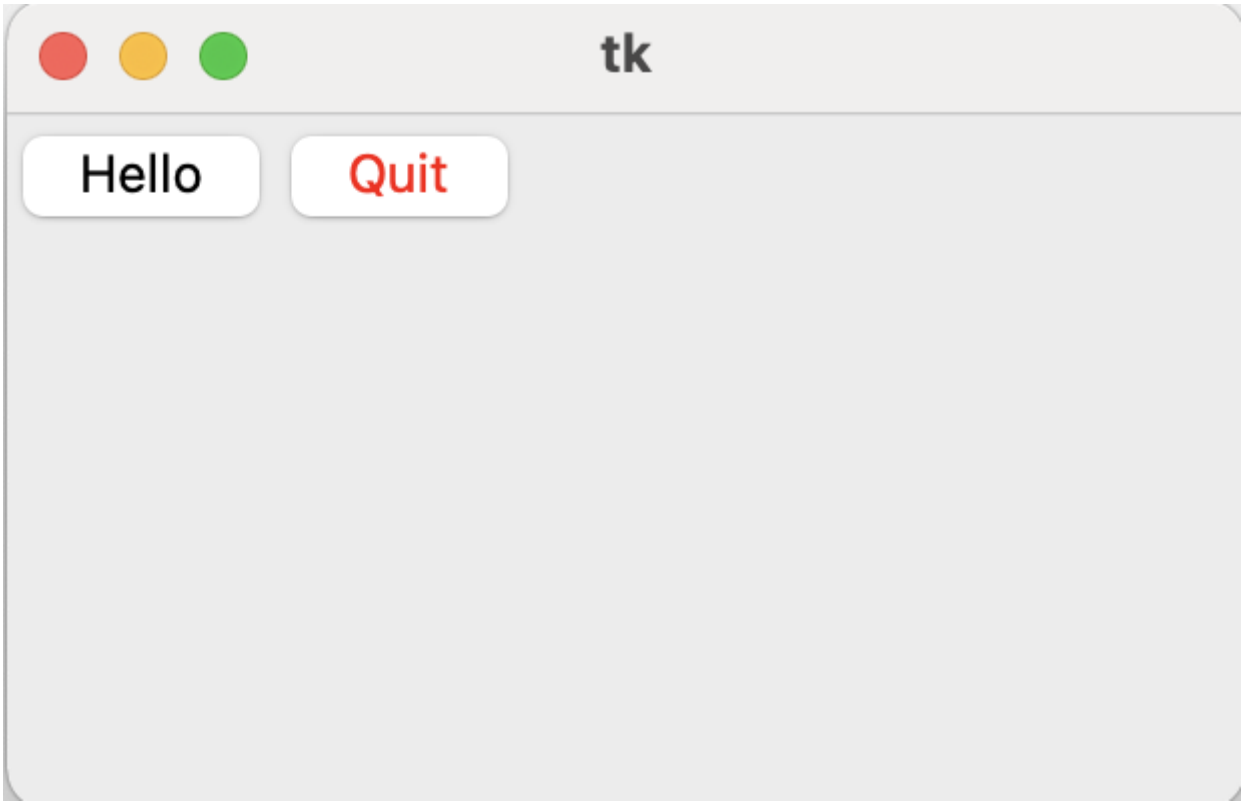
Introducing to callbacks

To do anything really useful with GUI programming, we need some way of associating an action with a widget. Usually, when user presses a button on a GUI she expects something to happen as a result. Let's make our Hello World program a bit more complex, by adding two buttons:

```
from tkinter import *

class HelloWorld:
    """
    A Hello World App.
    """
    def __init__(self, parent):
        self.hello_button = Button(parent, text="Hello",
command=self.say_hi)
```

```
        self.hello_button.grid(column=0, row=0)
        self.quit_button = Button(parent, text="Quit", fg="red",
command=parent.destroy)
        self.quit_button.grid(column=1, row=0)
    def say_hi(self):
        print("hi there")

root = Tk()
app = HelloWorld(root)
root.mainloop()
```

If we run the above program, we will get a window that looks like Figure 2. In both calls to the function Button, there is a parameter called command. The value of the command parameter is the callback or function associated with the action of pressing the button. So, if we press the Hello button, it should print 'hi there' to the system console because that's what the function say_hi does. If we press the Quit button, then the special function parent.destory will be called which, unsurprisingly, causes the window to be destroyed.



Since we're writing GUI program, it would be very unsual (and quite disconcerting) to make use of the console at all. When we want to give a message to the user, it is more usual to use a dialog widget. The module tkinter.messagebox includes numerous useful dialogs. The above program can be modified to use a showinfo dialog box instead:

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

class HelloWorld:
    """
```

```
    A Hello World App.
    """
    def __init__(self, parent):
        self.hello_button = Button(parent, text="Hello",
command=self.say_hi)
        self.hello_button.grid(column=0, row=0)
        self.quit_button = Button(parent, text="Quit", fg="red",
command=parent.destroy)
        self.quit_button.grid(column=1, row=0)
    def say_hi(self):
        showinfo(message="Hi there")

root = Tk()
app = HelloWorld(root)
root.mainloop()
```

Now when we press the Hello button, we should see something similar to Figure 3.

User input

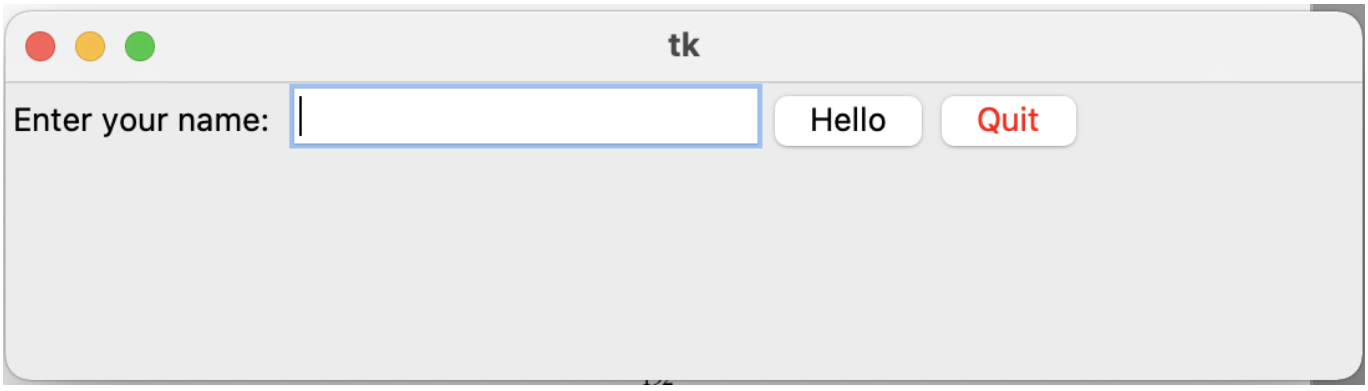
Tkinter has several widgets available for getting input from the user. Dialog boxes allow one sort of input, but most interfaces have some form of input on the main window. There are two main widgets for doing this: Entry and Text. We will use the Entry widget for our simple application.

To make use of user input widgets, we need a mechanism for getting information back out of the widget. The easiest way to do that is to use the get method from the widget and subsequently covert the string to the appropriate variable if needed.

Let's have a look at an example of how to use an Entry widget:

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

class HelloInput:
    """
    An app that gets input from the user and associates Tkinter variable
with the input.
    """
    def __init__(self, parent):
        self.hello_button = Button(parent, text="Hello",
command=self.say_hi)
        self.hello_button.grid(column=2, row=0)
        self.quit_button = Button(parent, text="Quit", fg="red",
command=parent.destroy)
        self.quit_button.grid(column=3, row=0)
        self.input = Entry(parent)
        self.input.grid(column=1, row=0)
        self.label = Label(parent, text="Enter your name: ")
        self.label.grid(column=0, row=0)

    def say_hi(self):
        showinfo(message="Hi there "+self.input.get())

root = Tk()
app = HelloInput(root)
root.mainloop()
```

Not that when the callback say_hi is called, the string associated with Entry widget is retrived and included as part of the dialog message. The main window for this program can be seen in Figure 4.



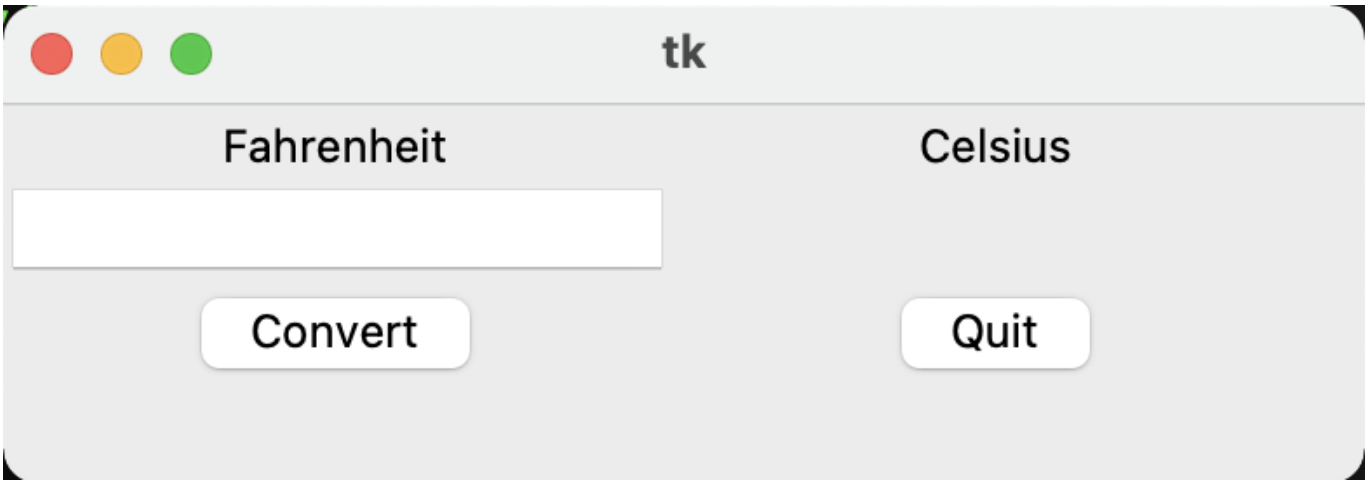
Mini-case study

Let's write a simple GUI program that can convert from Fahrenheit to Celsius. The coversion equation is quite simple: $$c = \frac{5}{9}(f-32)$$ where f is the temperature in Fahrenheit and c is the temperature in Celsius.

For this program, we'll need an Entry widget, an output Label widget, a Button widget to do the conversion and a Button widget to quit the application. The program, tk_converter1.py, is shown below:

```
from tkinter import *

class TempConverter:
    """
    A simple app for converting from Fahrenheit to Celsius
    """
    def __init__(self, parent):
        self.celsius_val = Label(parent, text="", width=20)
        self.celsius_val.grid(column=1, row=1)
        self.celsius_label = Label(parent, text="Celsius", width=20)
        self.celsius_label.grid(column=1, row=0)

        self.fahr_input = Entry(parent)
        self.fahr_input.grid(column=0, row=1)
        self.fahr_label = Label(parent, text="Fahrenheit")
        self.fahr_label.grid(column=0, row=0)

        self.convert_button = Button(parent, text="Convert",
command=self.convert)
        self.convert_button.grid(row=2, column=1)

        self.quit_button = Button(parent, text="Quit",
comman=parent.destroy)
        self.quit_button.grid(row=2, column=1)

    def convert(self):
        # get the fahrenheit value from the widget and convert it from
        # a string to a float number
        dfahr = float(self.fahr_input.get())
        # calculate the celsius value as a float
        celsius = (dfahr-32)*5.0/9.0
        # update the celsius widget with the celsius value by
        # converting from a float to a string
        self.celsius_val.configure(text = str(celsius))

root = Tk()
app = TempConverter(root)
root.mainloop()
```

In this function convert does the conversion. Notice how it first gets the value from self.fahr_input which is what the user entered into the entry field. Then it does the conversion, and finally, it changes the text on the output field by calling the configure method. If you type in and run the above program, you should get an application that looks like Figure 5.



Common Patterns

Typical Program Structure

Here is the usual structure of a GUI program in Python:

```
from tkinter import *
class ClassName:
    def __init__(self, parent):
    # code to create and display widgets goes here as
    # well as any other variables needed for the class
    def some_callback_method(self):
    # code you want to execute in response to an event
    # goes here
    def maybe_another_callback(self):
    # maybe code to respond to a different event
    # goes here
    # main routine
root = Tk()
instance_name = ClassName(root)
root.mainloop()
```

Creating and Displaying Widgets

General pattern:

```
self.widget_name = WidgetType(parent, option1=val1, option2=val2)
self.widget_name.grid(row=2, column=3) # or appropriate vals
```

Examples:

```
self.my_hi_button = Button(parent, text="Hello", bg="red")
self.my_hi_button.grid(row=0, column=0)
```

```
self.entry_label1 = Label(parent, text="Enter birth year:")
self.entry_label1.grid(row=0, column=1)
```

Responding to events

To respond to events you need to define a function that will be caleed when the event happens, and you also need to associate the event with the function. This often happens at the time of creating widgets.

```
from tkinter import *
class ClassName:
    def __init__(self, parent):
        self.my_hi_button = Button(parent, text="Hello", bg="red",
command=self.callback_name)
        self.my_hi_button.grid(row=0,column=0)
    def callback_name(self):
        print 'hello there' # this prints to the console

# main routine
root = Tk()
instance_name = ClassName(root)
root.mainloop()
```