



Computer Programming in Python

Chapter 11

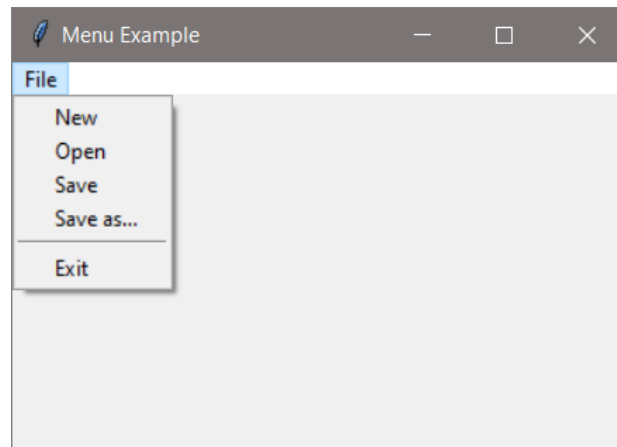
Menus, Images, and Windows

Chapter 11 Menus, Images, and Windows

- GUI User interaction
 - Menu
 - Provide program-level operations
 - Images
 - Enhance the user experience
 - Provide information
 - Charts and Plotted Data
 - Enhance information presentation
 - Interface Operation Control
 - Window icons, centering, resizing, closing

Chapter 11 Menus, Images, and Windows

- Drop-down Menus
 - Provide program-level operations
 - File handling - Open, Save, and Save as
 - Exit the program
 - Rests on the window frame, and drops down to reveal the options

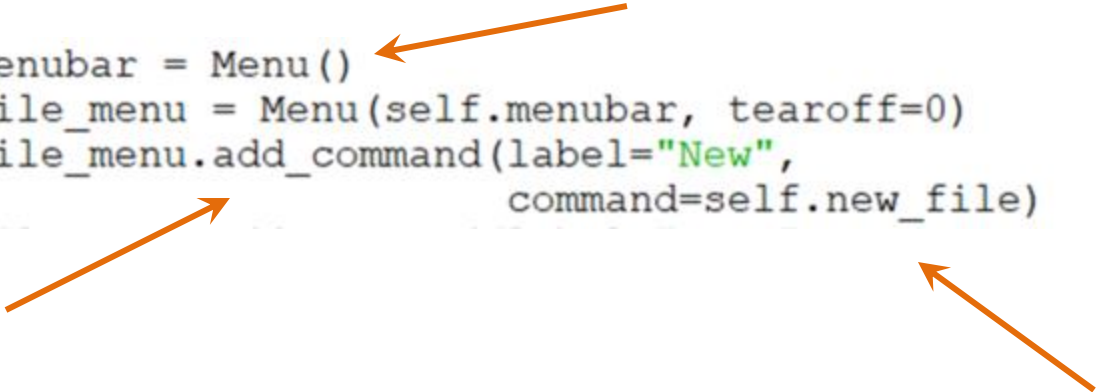


Chapter 11 Menus, Images, and Windows

- Drop-down Menus

- A drop-down is created by assigning a *Menu()*
- The items listed on the menu are added using *add_command* a label (text), and a function to respond to the selection
 - Similar to the way the callback function is assigned to a button

```
self.menubar = Menu()  
self.file_menu = Menu(self.menubar, tearoff=0)  
self.file_menu.add_command(label="New",  
                           command=self.new_file)
```



Chapter 11 Menus, Images, and Windows

- Drop-down Menus
 - The other menu items and commands are added the same way

```
self.file_menu.add_command(label="New",  
                           command=self.new_file)  
self.file_menu.add_command(label="Open",  
                           command=self.open_file)  
self.file_menu.add_command(label="Save",  
                           command=self.save_file)  
self.file_menu.add_command(label="Save as...",  
                           command=self.save_as_file)
```

Chapter 11 Menus, Images, and Windows

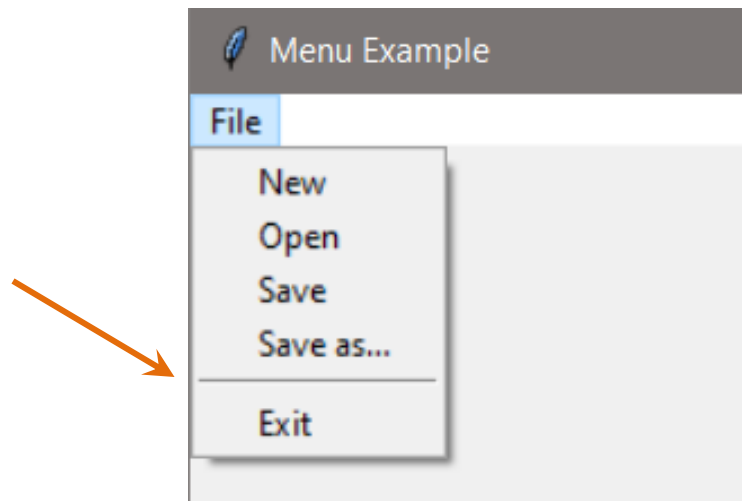
- Drop-down Menus
 - The functions handle the dialogs and operations
- The open file function with a test loop

```
def open_file(self):  
    print("Open clicked.")  
    infile = tkFileDialog.askopenfile()  
    for line in infile:  
        print(line)
```

Chapter 11 Menus, Images, and Windows

- Drop-down Menus
 - A separator can be added between selections using *add_separator()*

```
self.file_menu.add_separator()
```



Chapter 11 Menus, Images, and Windows

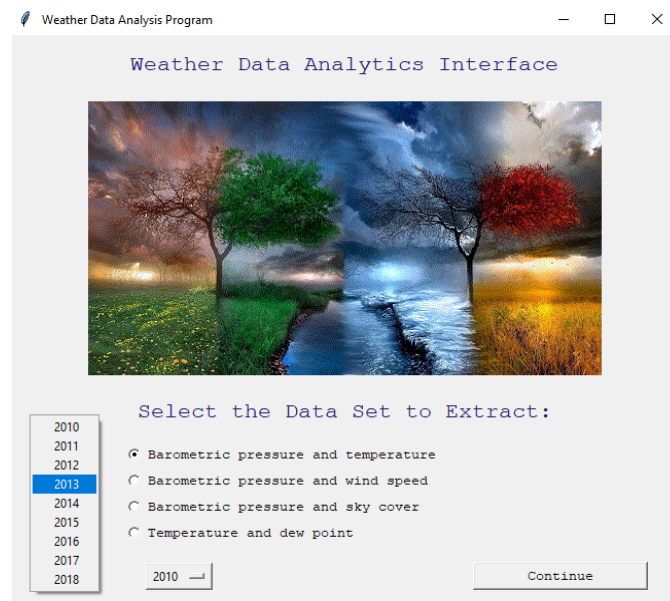
- Drop-down Menus
 - The *destroy* method closes the window and ends the program

```
self.file_menu.add_separator()  
self.file_menu.add_command(label="Exit",  
                           command=self.main_win.destroy)
```



Chapter 11 Menus, Images, and Windows

- Images
 - An image on an interface can enhance the user experience and provide information



Chapter 11 Menus, Images, and Windows

- Images
 - Tkinter has a *PhotoImage* class for handling images
 - Supports the GIF, PGM/PPM, and PNG formats
 - The image file can be located with the program files, which is the default directory, or a path to the file can be used

```
from tkinter import PhotoImage # import PhotoImage
import tkinter as tk          # imports tkinter as tk
```

Chapter 11 Menus, Images, and Windows

- Wildcard Note

- Some programmers may use an import statement with a wildcard (asterisk) as shown here which imports the entire tkinter module

```
from tkinter import *    # NOT RECOMMENDED
```

- Avoid using **wildcard import statements** especially when multiple modules are imported
 - Name clashes can occur when modules have functions or classes with the same name

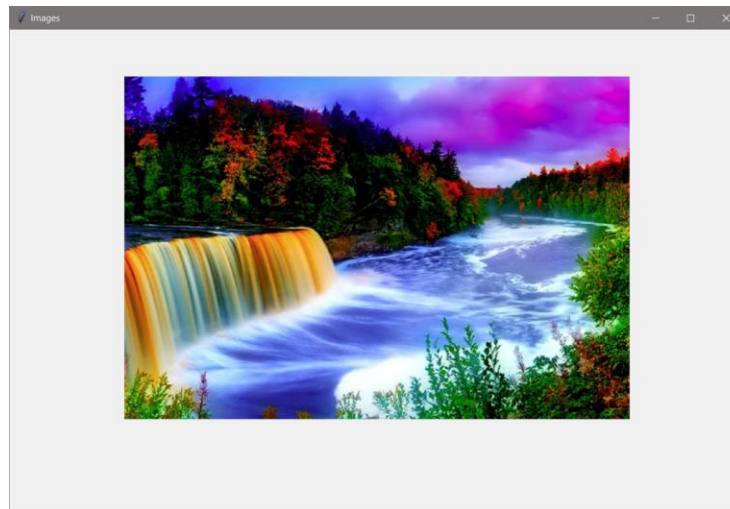
Chapter 11 Menus, Images, and Windows

- Images
 - The code to apply an image consists of three lines
 - First, assign the file to a *PhotoImage* object
 - Second, place the image on a label
 - A canvas or frame can also be used
 - Third - retain a reference to the image

```
photo = PhotoImage(file="WaterFall2.PNG")  
  
self.main_win.image_label = tk.Label(image=photo)  
  
self.main_win.image_label.image=photo    # retain a reference
```

Chapter 11 Menus, Images, and Windows

- Images
 - A reference to the image must be retained or Python's interpreter could eliminate it even if it is being displayed

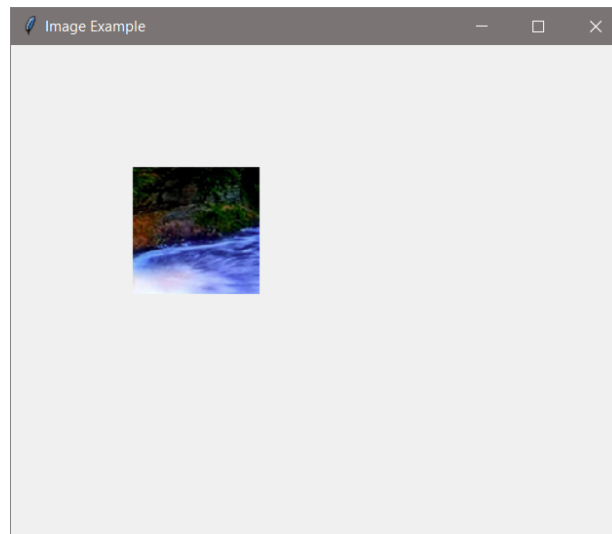


Chapter 11 Menus, Images, and Windows

- Images

- Since the image is attached to a label, it can be positioned using a geometry manager

```
self.main_win.image_label.grid(row=1, column=1)
```



Chapter 11 Menus, Images, and Windows

- Centering the Window
 - A simple method for centering uses *geometry*
 - Changes what tkinter sees as the top-left corner
 - Display resolution dependent

```
# Geometry Arguments - no spaces
#           window size 400 x 300
#           + x position + y position

self.main_win.geometry('400x300+500+300')
```

Resolution dependent centering

Chapter 11 Menus, Images, and Windows

- Centering the Window
 - Use the tkinter *geometry* method
 - Determine the display area size
 - Subtract the window size and divide by two

```
# window size - width = 400, height = 300
# Use screenwidth and screenheight to calculate centering

x_Left = int((self.main_win.wininfo_screenwidth() - 400) / 2)
y_Top = int((self.main_win.wininfo_screenheight() - 300) / 2)

self.main_win.geometry('%dx%d+%d+%d' % (400, 300, x_Left, y_Top))
```


Chapter 11 Menus, Images, and Windows

- Disabling Window Resizing
 - When window controls are positioned using a specific height and width for the window. If a user stretches the window in any direction, the components may move and ruin the desired arrangement
 - The *resizable* function can be set to false
 - Two versions are shown

```
self.main_win.resizable(False, False)
```

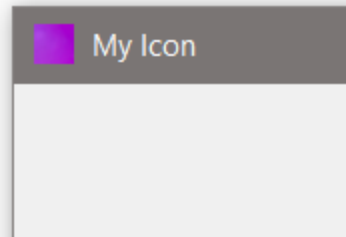
```
self.main_win.resizable(height = False, width = False)
```

Chapter 11 Menus, Images, and Windows

- Window Icons

- Changing the icon requires an image in the .ico format, and using the *iconbitmap()* method
- Once the icon is created, the filename or path is passed to the method

```
self.main_win.iconbitmap('myIcon.ico')
```



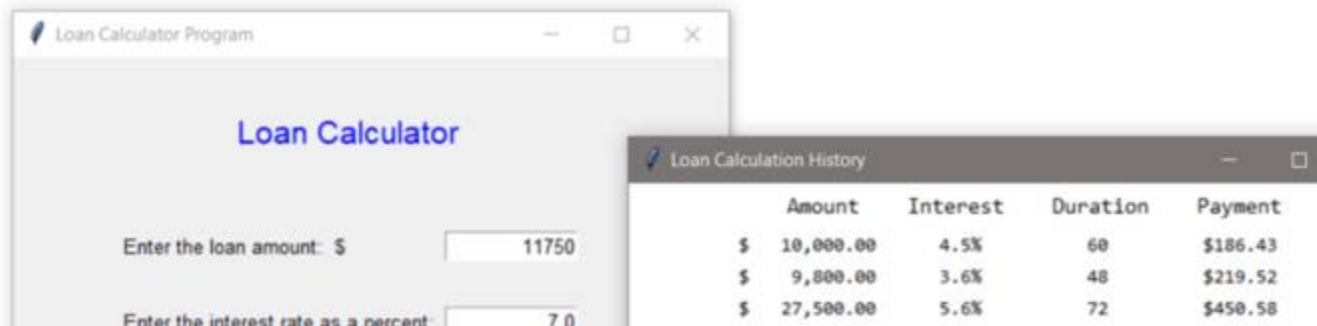
Chapter 11 Menus, Images, and Windows

- Window Icons
 - Another method for changing the icon is *iconphoto()*
 - Accepts other image formats
 - The first argument indicates only this window is to get the icon

```
image = tk.PhotoImage(file = 'myIcon.png')  
self.main_win.iconphoto(False, image)
```

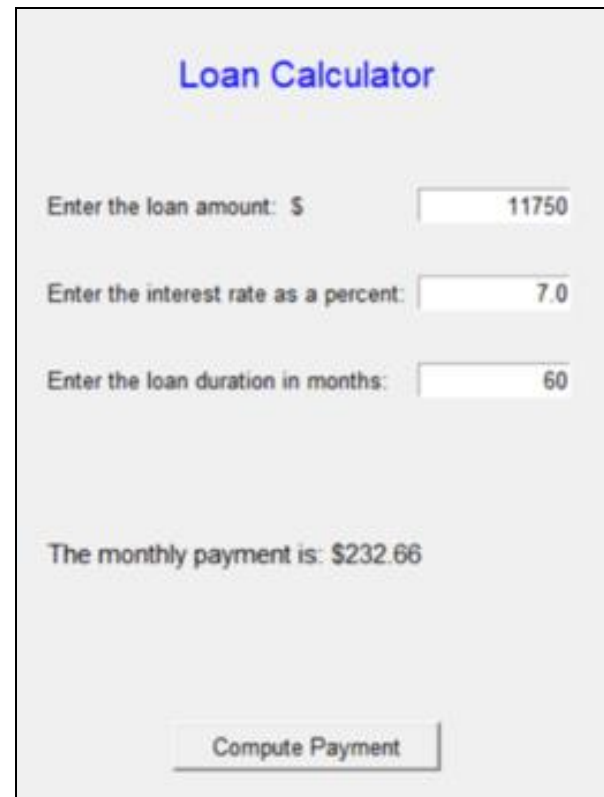
Chapter 11 Menus, Images, and Windows

- Updating a Second Window
 - Many GUI programs display data to the user as it is being computed and display the previous results for comparison
 - This is often in a second display window
 - As an example, consider a program that computes a value when new data is entered and a second window that displays the historical results



Chapter 11 Menus, Images, and Windows

- Updating a Second Window
 - The main GUI includes three prompt labels, three entry components for the data, and a “Compute Payment” button



The screenshot shows a window titled "Loan Calculator" with a light gray background. It contains three input fields with labels to their left: "Enter the loan amount: \$" with the value "11750", "Enter the interest rate as a percent:" with the value "7.0", and "Enter the loan duration in months:" with the value "60". Below these fields, the text "The monthly payment is: \$232.66" is displayed. At the bottom center, there is a button labeled "Compute Payment".

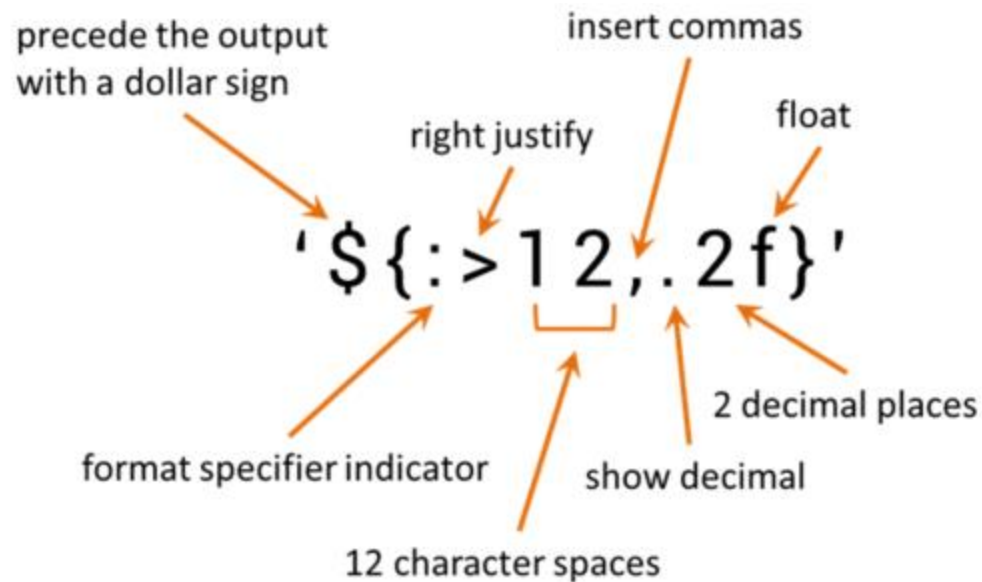
Chapter 11 Menus, Images, and Windows

- Updating a Second Window
 - To display the history of loan computations, once the payment amount has been calculated, the data is formatted for display
 - The example below formats the loan amount for the display window
 - Note that a dollar sign precedes the opening brace and the entire expression is inside quotes followed by a dot and format

```
fltA = float(amt)          # convert the value to float
fltA_string = '${:>12,.2f}'.format(fltA)
```

Chapter 11 Menus, Images, and Windows

- Formatting Data
 - The new Python formatting types use placeholders (braces) and additional specifiers



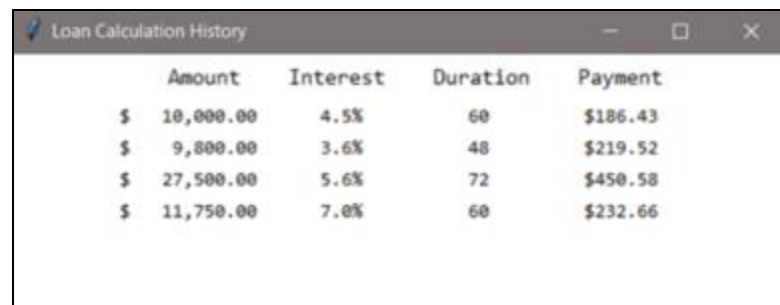
Chapter 11 Menus, Images, and Windows

- Updating a Second Window

- Once the data is formatted, a new label is created and placed on the next row of the output display

```
self.dd_win.data_lbl = tk.Label(self.dd_win,  
                                font=('Consolas',10), bg='white', \  
                                text=fltA_string + fltI_string + \  
                                intD_string + fltMP_string)  
self.dd_win.data_lbl.grid(row=rc, column=1, sticky='W', colspan=4)
```

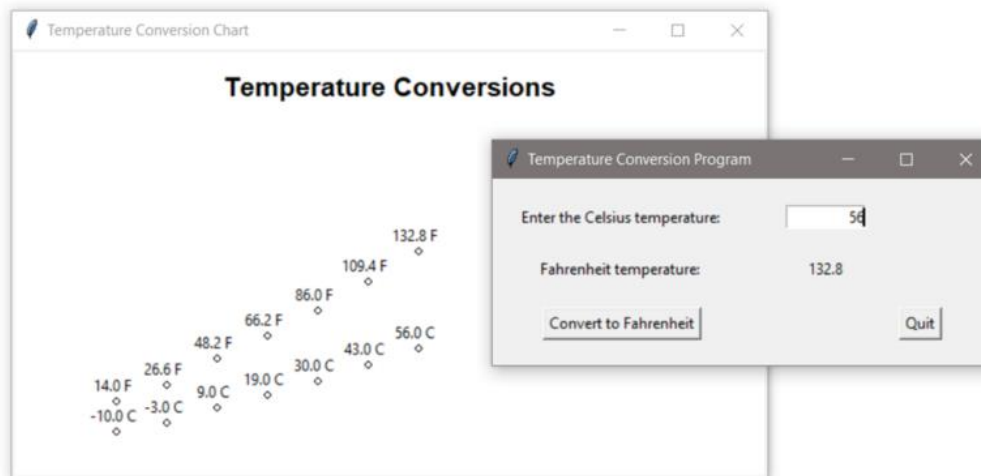
rc is a rowcounter
that is incremented



Amount	Interest	Duration	Payment
\$ 10,000.00	4.5%	60	\$186.43
\$ 9,800.00	3.6%	48	\$219.52
\$ 27,500.00	5.6%	72	\$450.58
\$ 11,750.00	7.0%	60	\$232.66

Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window
 - Plotted data is often shown in a separate window
 - Data can be plotted on a Canvas added to a display window



Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window

- This example will compute a Fahrenheit temperature from a Celsius input and plot both values in a separate display
- The window owning the canvas is the first argument when it is created

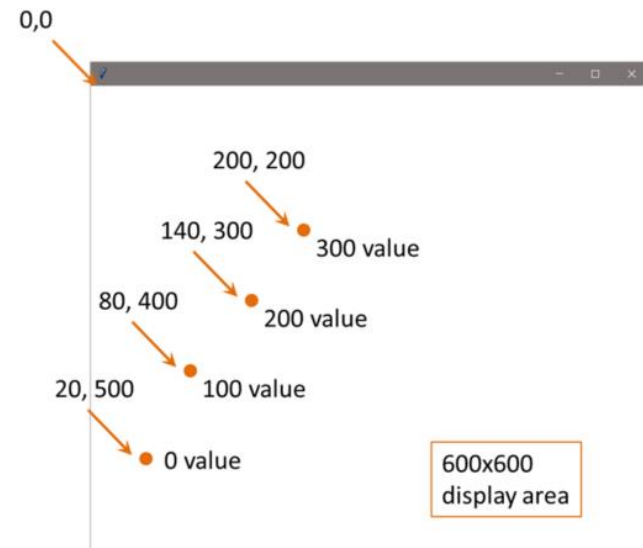
```
self.plot_win = tk.Tk()
self.plot_win.geometry('600x340')
self.plot_win.title('Temperature Conversion Chart')

self.canvas = tk.Canvas(self.plot_win,width=600,height=340,
                        bg='white')
self.canvas.create_text(300, 30, font='Helvetica 16 bold',
                        text='Temperature Conversions')
self.canvas.pack()
```

Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window

- Drawing on a canvas uses the x, y coordinate system
- Working relative to the top-left corner which is 0, 0
- The “baseline” for data values is down from the top



Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window - Design
 - The plotting coordinates require determining an optimum size for the window based upon:
 - The possible range of values that will be displayed
 - The scaling factor (pixels)
 - And moving the “x” coordinate for each set of values computed

Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window - Design
 - The program accepts a range of Celsius inputs from -10 to 100 degrees Celsius
 - There are 110 Celsius data points
 - The conversion range for this set of values would be 14 to 212 degrees Fahrenheit
 - There are 198 Fahrenheit data points
 - The total range to be plotted is then -10 to 212 which is 222 data points

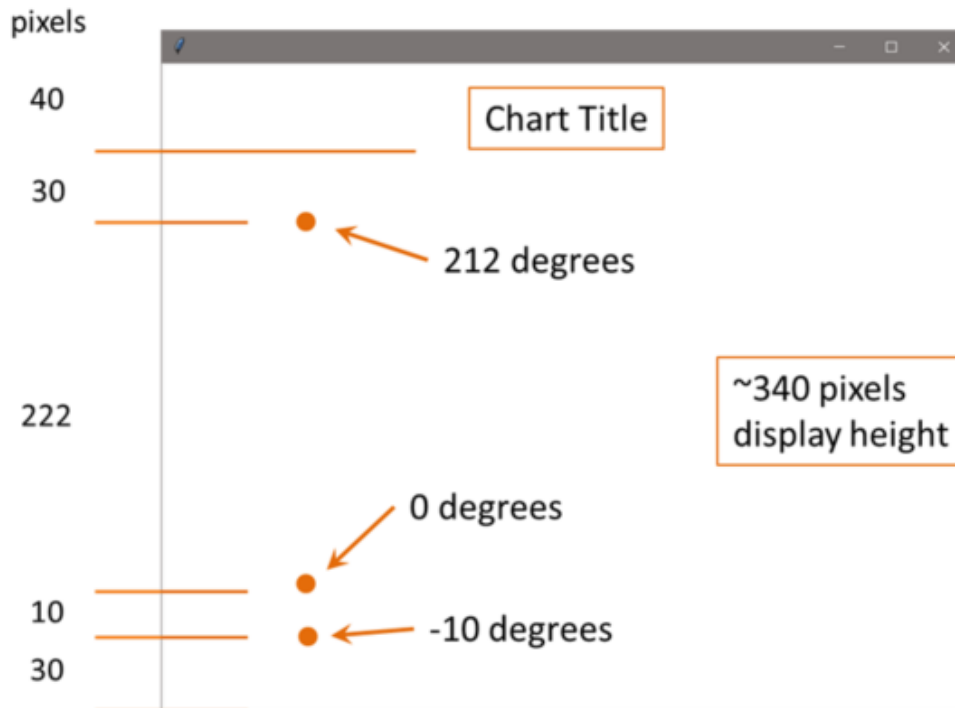
Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window
 - One pixel could represent one degree, so the window needs to be at least 222 pixels in height
 - Consider that a title for the chart and spacing requires additional height

A few minutes designing can save hours programming

Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window
 - A design sketch makes it easier to approximate locations



Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window
 - An offset of 5 pixels is used for the text
 - Ovals are used to create the circles

```
self.canvas.create_text(self.data_num*40, 280-self.fahrenheit, \
                        text= str(self.fahrenheit)+ ' F')

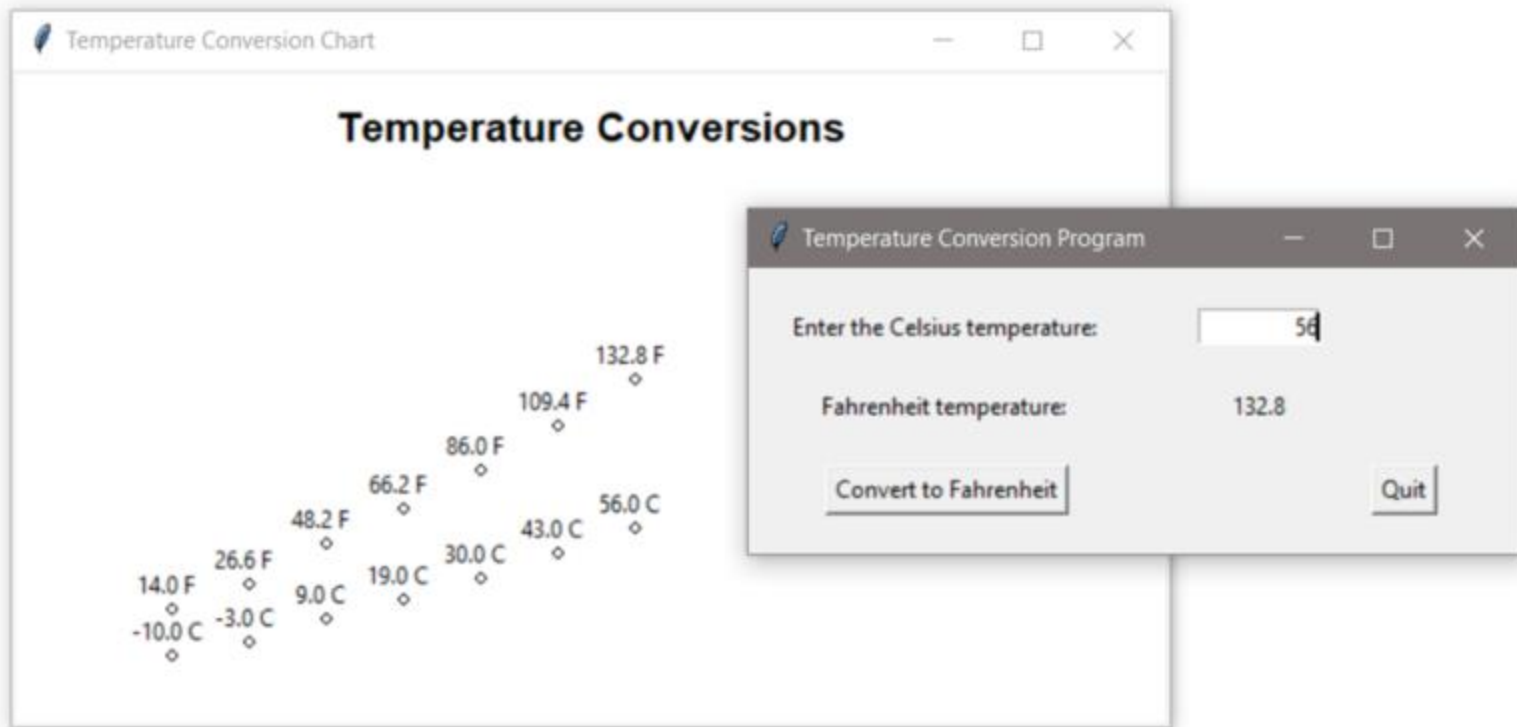
self.canvas.create_oval(self.data_num*40, 290-self.fahrenheit, \
                        5 + self.data_num*40, 295-self.fahrenheit)

self.canvas.create_text(self.data_num*40, 280-self.celsius, \
                        text=str(self.celsius)+ ' C')

self.canvas.create_oval(self.data_num*40, 290-self.celsius, \
                        5 + self.data_num*40, 295-self.celsius)
```


Chapter 11 Menus, Images, and Windows

- Plotting to a Second Window



Chapter 11 Menus, Images, and Windows

- Interacting with a Second (Toplevel) Window
 - Windows created in addition to the main window, are referred to as *Toplevel* windows
 - The following is a simple example that creates a main window with a button, and a second window that reacts to the button click
 - The change is handled through a StringVar
 - Note that the second window is declared as a tk.Toplevel, and that the StringVar is not assigned to a window

```
class TwoWins:
    def __init__(self):

        main_win = tk.Tk()
        main_win.title('Main Win')
        main_win.geometry('300x200')
        main_win.btn = tk.Button(text='Click Here', \
                                  width=18, command=self.update)
        main_win.btn.pack()

        sec_win = tk.Toplevel()
        sec_win.title('Second Win')
        sec_win.geometry('300x200')

        self.update_var = tk.StringVar()
        self.update_var.set('The label')

        sec_win.lbl = tk.Label(sec_win,
                                textvariable=self.update_var)
        sec_win.lbl.pack()

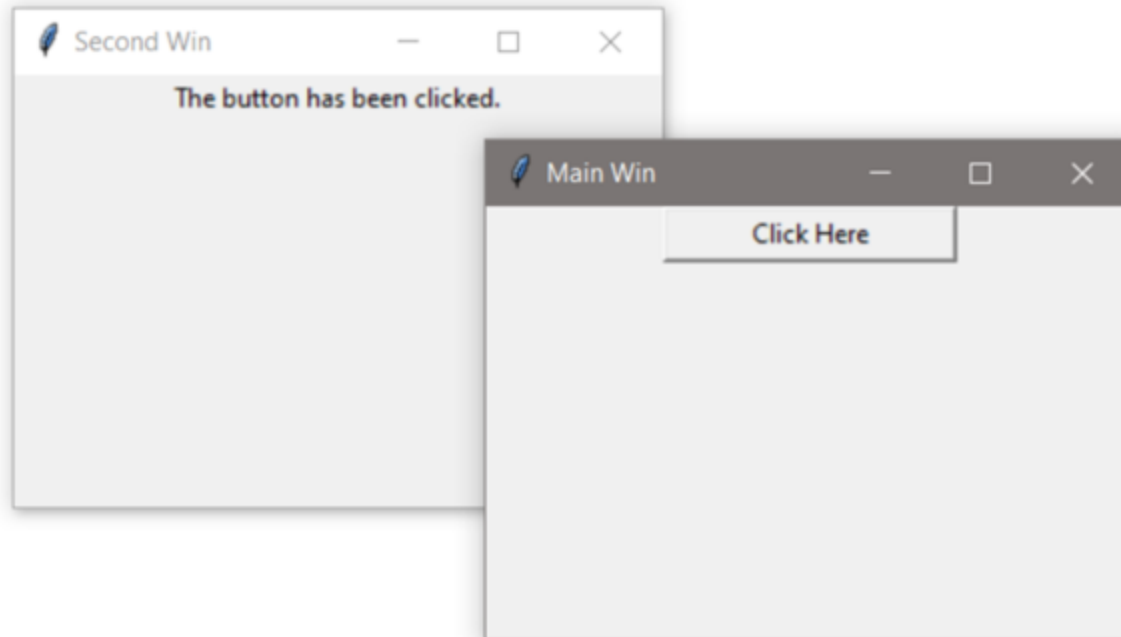
        tk.mainloop()

    def update(self):
        self.update_var.set('The button has been clicked.')
```

```
iWin = TwoWins()
```

Chapter 11 Menus, Images, and Windows

- Interacting with a Second (Toplevel) Window



Chapter 11 Menus, Images, and Windows

- Closing Windows
 - When a user exits a program either by clicking on a “Quit” button (if provided) or by clicking on the “X” at the top right corner of the window, the program should end
 - This includes closing any other windows created by the program
 - There are several ways of handling this

Chapter 11 Menus, Images, and Windows

- Closing Windows

- The first example includes a quit button that calls a function assigned to the command, that uses the destroy method to end the program (and the main loop)

- The destroy method cannot be assigned directly to the command

```
self.quit_button = tk.Button(text = 'Quit', width=12,  
                             command = self.close_prog)  
self.quit_button.grid(row=1, column=1)
```

```
tk.mainloop()
```

```
def close_prog(self):  
    self.main_win.destroy()
```

Chapter 11 Menus, Images, and Windows

- Closing Windows
 - The function can provide any clean-up needed by the program including closing other windows

```
def close_prog(self):  
    self.main_win.destroy()  
    self.second_win.destroy()
```

Chapter 11 Menus, Images, and Windows

- Lambda Expressions
 - A lambda expression is an inline function
 - Lambda expressions are not necessary, but in some situations, they make writing the code easier
 - When a function is simple and will only be called once, a lambda expression makes sense
 - It can be anonymous (no name) and defined where it will execute
 - One frequent use of a lambda is in programming “callbacks” for the command assigned to a button
 - A button requires a function object to be assigned to the command

Chapter 11 Menus, Images, and Windows

- Lambda Expressions

- Since a button requires a function object to be assigned to the command, one way of handling this is to have the command be a call to a function, and have the function perform the operation

```
self.new_button1 = tk.Button(text='Button 1', width=16,  
                             command=self.on_click)  
def on_click(self):  
    print('Button selected')
```

Chapter 11 Menus, Images, and Windows

- Lambda Expressions
 - Using a lambda function would eliminate the call to the function
 - The keyword lambda is followed by a colon and the function

```
self.new_button2 = tk.Button(text='Button 2', width=16,  
                             command=lambda : print('Lambda !'))
```

Chapter 11 Menus, Images, and Windows

- Lambda Expressions
 - The earlier example that used a function that called `destroy` to close the window can be rewritten using a lambda expression as well
 - As long as there is no other clean-up required

```
self.quit_button = tk.Button(text = 'Quit', width=12,  
                             command = lambda:self.main_win.destroy())
```

Chapter 11 Menus, Images, and Windows

- Using Protocol

- Can also use protocol and the event of the window closing so that when a user clicks on the “X”, the program has control and can execute other statements like closing other windows
- Below, both windows react to being closed by the system and call the function that closes them both

```
self.main_win.protocol("WM_DELETE_WINDOW", self.close_prog)  
self.sec_win.protocol("WM_DELETE_WINDOW", self.close_prog)
```

```
def close_prog(self):  
    self.main_win.destroy()  
    self.sec_win.destroy()
```

Chapter 11 Menus, Images, and Windows

Chapter 11 Menus, Images, and Windows