Tkinter\_GUI\_Application\_Development\_**Bhaskar\_**c01

**Tkinter GUI Application Development H TSHOT**

Develop exciting and engaging GUI applications in Python and Tkinter by working on 10 real-world applications

**Bhaskar Chaudhary**

p20

**Project 1 Meet Tkinter**

Welcome to the exciting world of GUI programming with Tkinter. This project aims at getting you acquainted with Tkinter, the built-in graphical user interface (GUI) interface for all standard Python distributions.

Bienvenido al apasionante mundo de la programación GUI con Tkinter. Este proyecto tiene como objetivo familiarizarlo con Tkinter, la interfaz gráfica de usuario (GUI) incorporada para todas las distribuciones estándar de Python.

**Tkinter** (pronounced tea-kay-inter) is the Python interface to Tk, the GUI toolkit for Tcl/Tk.

**Tcl** (pronounced "tickle" and is an acronym for Tool Command Language) is a popular scripting language in the domains of embedded applications, testing, prototyping, and GUI development.

**Tk** on the other hand is an open source, multiplatform widget toolkit that is used by many different languages for building GUI programs.

Tk, por otro lado, es un conjunto de herramientas de widgets multiplataforma de código abierto que se utiliza en muchos lenguajes diferentes para crear programas GUI.

The Tkinter interface is implemented as a Python module, Tkinter.py, which is just a wrapper around a C-extension that uses Tcl/Tk libraries.

La interfaz Tkinter se implementa como un módulo de Python, Tkinter.py, que es solo un contenedor de una extensión C que usa bibliotecas Tcl/Tk.

Tkinter is suited for application to a wide variety of areas ranging from small desktop applications, to use in scientific modeling and research endeavors across various disciplines.

Tkinter es adecuado para su aplicación en una amplia variedad de áreas que van desde pequeñas aplicaciones de escritorio hasta su uso en modelado científico y proyectos de investigación en diversas disciplinas.

We believe that the concepts you will develop here will enable you to apply and develop GUI applications in your area of interest. Let's get started!

Creemos que los conceptos que desarrollará aquí le permitirán aplicar y desarrollar aplicaciones GUI en su área de interés. ¡Empecemos!

**Mission Briefing**

The purpose of this project is to make you comfortable with Tkinter. It aims at introducing you to various components of GUI programming with Tkinter.

El propósito de este proyecto es hacer que se sienta cómodo con Tkinter. Su objetivo es presentarle varios componentes de la programación GUI con Tkinter.

By the end of this project, you will have developed several partly functional dummy applications such as the one shown as follows:

Al final de este proyecto, habrá desarrollado varias aplicaciones ficticias parcialmente funcionales, como la que se muestra a continuación:

The applications developed in this project are "dummy applications" because they are not fully functional. In fact, the purpose of each small dummy application is to introduce you to some specific aspects of programming with Tkinter. This will set up the context for developing some fun and fully functional project ideas from *Project 2*, *Making a Text Editor*, onwards.

Las aplicaciones desarrolladas en este proyecto son "aplicaciones ficticias" porque no son completamente funcionales. De hecho, el propósito de cada pequeña aplicación ficticia es presentarle algunos aspectos específicos de la programación con Tkinter. Esto establecerá el contexto para desarrollar algunas ideas de proyectos divertidas y completamente funcionales desde el Proyecto 2, Creación de un editor de texto, en adelante.

**Why Is It Awesome?**

¿Por qué es impresionante?

The ability to program a GUI application (as opposed to a simple console application) opens a whole world of possibilities for a programmer. It shifts the focus of the program from the programmer to the end user, enabling the programmer to reach out to a wider audience.

La capacidad de programar una aplicación GUI (a diferencia de una simple aplicación de consola) abre todo un mundo de posibilidades para un programador. Cambia el enfoque del programa del programador al usuario final, permitiendo al programador llegar a una audiencia más amplia.

When a person learning Python needs to graduate to GUI programming, Tkinter seems to be the easiest and fastest way to get the work done. Tkinter is a great tool for programming GUI applications in Python.

Cuando una persona que aprende Python necesita pasar a la programación GUI, Tkinter parece ser la forma más fácil y rápida de realizar el trabajo. Tkinter es una gran herramienta para programar aplicaciones GUI en Python.

The features that make Tkinter a great choice for GUI programming include:

* It is simple to learn (simpler than any other GUI package for Python)
* Relatively little code can produce powerful GUI applications
* Layered design ensures that it is easy to grasp

El diseño en capas garantiza que sea fácil de agarrar

* It is portable across all operating systems
* It is easily accessible as it comes pre-installed with standard Python distribution

None of the other GUI toolkits has all of these features at the same time.

**Your Hotshot Objectives**

Tus objetivos destacados

The key concepts that we want you to take from this project include:

* Understanding the concept of root window and main loop

Comprender el concepto de ventana raíz y bucle principal

* Understanding widgets—the building blocks for your programs

Comprender los widgets: los componentes básicos de sus programas

* Acquainting yourself with a list of available widgets
* Developing layouts using three geometry managers: pack, grid, and place

Desarrollar diseños utilizando tres administradores de geometría: empaquetar, cuadrícula y colocar

* Learning to apply events and callbacks to make your program functional

Aprender a aplicar eventos y devoluciones de llamadas para que su programa sea funcional

* Styling your widgets with styling options and configuring the root widget

Diseñe sus widgets con opciones de estilo y configure el widget raíz

**Mission Checklist**

Lista de verificación de la misión

An elementary knowledge of data structures, syntax, and semantics of Python is assumed. To work along with this project, you must have a working copy of Python 2.7.3 installed on your computer.

Se supone un conocimiento elemental de las estructuras de datos, la sintaxis y la semántica de Python. Para trabajar con este proyecto, debe tener una copia funcional de Python 2.7.3 instalada en su computadora.

The Python download package and instructions for downloading for different platforms are available at http://www.Python.org/getit/releases/2.7.3/.

We will develop our application on the Windows 7 platform. However, since Tkinter is truly cross-platform, you can follow along on Mac or Linux distributions without any modifications to our code.

After the installation, open the IDLE window and type:

**>>>from Tkinter import \***

If you have installed Python 2.7, this shell command should execute without any errors.

If there are no error messages the Tkinter module is installed in your Python distribution. When working with examples from this book, we do not support any other Python version except for Python 2.7, which comes bundled with Tkinter Tcl/Tk Version 8.5.

To test if you have the correct Tkinter version on your Python installation, type the following commands in your IDLE or interactive shell:

**>>> import Tkinter**

**>>>Tkinter.\_test()**

This should pop up a window where the first line in the window reads This is Tcl/Tk version 8.5. Make sure it is not 8.4 or any earlier version, as Version 8.5 is a vast improvement over its previous versions.

You are ready to code your Tkinter GUI applications if your version test confirms it as Tcl/Tk version 8.5. Let's get started!

**The root window – your drawing board**

La ventana raíz: tu mesa de dibujo

GUI programming is an art, and like all art, you need a drawing board to capture your ideas. The drawing board you will use is called the root window. Our first goal is to get the root window ready.

La programación GUI es un arte y, como todo arte, necesitas una mesa de dibujo para plasmar tus ideas. El tablero de dibujo que utilizará se llama ventana raíz. Nuestro primer objetivo es preparar la ventana raíz.

**Engage Thrusters**

Activar propulsores

The following screenshot depicts the root window we are going to create:

La siguiente captura de pantalla muestra la ventana raíz que vamos a crear:

Drawing the root window is easy. You just need the following three lines of code:

f**rom Tkinter import \***

**root = Tk()**

**root.mainloop()**

Save this with the .py file extension or check out the code 1.01.py. Open it in the IDLE window and run the program from the Run menu (*F5* in IDLE). Running this program should generate a blank root window as shown in the preceding screenshot. This window is furnished with functional minimize, maximize, and close buttons, and a blank frame.

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub. com/support and register to have the files e-mailed directly to you.

The description of the code is as follows:

* The first line imports all (\*) classes, attributes, and methods of Tkinter into the current workspace.
* The second line creates an instance of the class Tkinter.Tk. This creates what is called the "root" window that you see in the screenshot provided. By convention, the root window in Tkinter is usually called "root", but you are free to call it by any other name.
* The third line executes the mainloop (that is, the event loop) method of the root object. The mainloop method is what keeps the root window visible. If you remove the third line, the window created in line 2 will disappear immediately as the script stops running. This will happen so fast that you will not even see the window appearing on your screen. Keeping the mainloop running also lets you keep the program running until you press the close button, which exits the main loop.

**Objective Complete – Mini Debriefing**

Objetivo completado – Mini informe

Congratulations! You have completed your first objective, which was to draw the root window. You have now prepared your drawing canvas (root window). Now get ready to paint it with your imagination!

Commit the three lines of code (shown in *code 1.01.py*) to memory. These three lines generate your root window, which will accommodate all other graphical components. These lines constitute the skeleton of any GUI application that you will develop in Tkinter. All code that will make your GUI application functional will go between line 2 (new object creation) and line 3 (mainloop) of this code.

Confirme las tres líneas de código (que se muestran en el código 1.01.py) en la memoria. Estas tres líneas generan su ventana raíz, que acomodará todos los demás componentes gráficos. Estas líneas constituyen el esqueleto de cualquier aplicación GUI que desarrollarás en Tkinter. Todo el código que hará que su aplicación GUI sea funcional irá entre la línea 2 (creación de nuevo objeto) y la línea 3 (bucle principal) de este código.

**Classified Intel**

This section describes the different styles of importing Tkinter modules.

In the preceding example, we imported Tkinter using the following command:

**from Tkinter import \***

This method of import eases the handling of methods defined in the module. That is to say, you can simply access the methods directly. Generally, it is considered a bad practice to import all (\*) methods of a module like we did here. This is because if you import all methods from some other module with a common method name, it would lead to the overwriting of methods.

Este método de importación facilita el manejo de los métodos definidos en el módulo. Es decir, simplemente puedes acceder a los métodos directamente. Generalmente, se considera una mala práctica importar todos los métodos (\*) de un módulo como lo hicimos aquí. Esto se debe a que si importa todos los métodos de algún otro módulo con un nombre de método común, se sobrescribirán los métodos.

There are several ways to import Tkinter in which this overlapping can be avoided, a common one being:

**import Tkinter**

This style of importing does not pollute the namespace with a list of all methods defined within Tkinter. However, every method within Tkinter will now have to be called using the format Tkinter.methodA instead of directly calling the method.

Another commonly used import style is as follows:

**import Tkinter as Tk**

Here too, you do not pollute the current namespace with all Tkinter methods and now you can access methods such as Tk.methodA. "Tk" is a convenient, easy-to-type alias commonly used by many developers for importing Tkinter.

Aquí tampoco contamina el espacio de nombres actual con todos los métodos de Tkinter y ahora puede acceder a métodos como Tk.methodA. "Tk" es un alias conveniente y fácil de escribir comúnmente utilizado por muchos desarrolladores para importar Tkinter.

**The big picture**

As a GUI programmer, you will generally be responsible for deciding three aspects of your program:

* What components should appear on screen?: This involves choosing the components that make the user interface. Typical components include things such as buttons, entry fields, checkboxes, radio buttons, scroll bars, and the like. In Tkinter, the components that you add to your GUI are called widgets.

¿Qué componentes deben aparecer en pantalla?: Esto implica elegir los componentes que componen la interfaz de usuario. Los componentes típicos incluyen elementos como botones, campos de entrada, casillas de verificación, botones de opción, barras de desplazamiento y similares. En Tkinter, los componentes que agregas a tu GUI se llaman widgets.

* Where should the components go?: This involves deciding the positioning or placement of each component in the overall design structure. This includes decisions to be made on issues of positioning and the structural layout of various components. In Tkinter, this is referred to as geometry management.

¿Dónde deben ir los componentes?: Esto implica decidir el posicionamiento o ubicación de cada componente en la estructura de diseño general. Esto incluye decisiones que se deben tomar sobre cuestiones de posicionamiento y disposición estructural de varios componentes. En Tkinter, esto se conoce como gestión de geometría.

* How do components interact and behave?: This involves adding functionality to each component. Each component or widget does some work. For example, a button, when clicked on, does something in response; a scrollbar handles scrolling; and checkboxes and radio buttons enable the user to make some choices. In Tkinter, the functionality of various widgets is managed by command binding or event binding using callback functions.

¿Cómo interactúan y se comportan los componentes?: Esto implica agregar funcionalidad a cada componente. Cada componente o widget hace algún trabajo. Por ejemplo, cuando se hace clic en un botón, hace algo en respuesta; una barra de desplazamiento maneja el desplazamiento; y las casillas de verificación y los botones de opción permiten al usuario tomar algunas decisiones. En Tkinter, la funcionalidad de varios widgets se administra mediante enlace de comandos o enlace de eventos mediante funciones de devolución de llamada.

Let us delve deeper into each of these three components in the context of Tkinter.

**Widgets – building blocks for your GUI program**

Now that we have our Toplevel window ready, it is time to think over the question, what components should appear in the window? In Tkinter jargon, these components are called widgets.

**Engage Thrusters**

The syntax for adding a widget is as follows:

**mywidget = Widget-name (its container window,\*\*configuration options)**

In the following example (refer to the code 01.02.py), we add two widgets, a label and a button, to the root frame. Notice how all widgets are added in between the skeleton code we defined in the first example.

En el siguiente ejemplo (consulte el código 01.02.py), agregamos dos widgets, una etiqueta y un botón, al marco raíz. Observe cómo se agregan todos los widgets entre el código esqueleto que definimos en el primer ejemplo.

**from Tkinter import \***

**root = Tk()**

**mylabel = Label(root,text="I am a label widget")**

**mybutton = Button(root,text="I am a button")**

**mylabel.pack()**

**mybutton.pack()**

**root.mainloop()**

The description of the code is listed as follows:

* This code adds a new instance, mylabel, for the **Label** widget. The first parameter defines root as its parent or container. The second parameter configures its text option as "I am a label widget".
* We similarly define an instance of a **Button** widget. This is also bound to the root window as its parent.
* We use the pack() method, which is essentially required to position the label and button widgets within the window. We will discuss the pack() method and several other related concepts under the Geometry management task. However, you must note that some sort of geometry specification is essential for the widgets to display within the Toplevel window.
* Running this code will generate a window as shown in the following screenshot. It will have a custom label and a custom button:

**Objective Complete – Mini Debriefing**

In this iteration, we have learned the following:

* What widgets are.
* How widgets are instantiated and displayed within a container window frame.
* How to set options for the widgets at the time of instantiation.
* The importance of specifying a geometry option such as pack() to display a widget. We will discuss more about this in a subsequent task.

**Classified Intel**

Intel clasificado

* All widgets are actually objects derived from their respective **widget class**. So, a statement such as mybutton = Button(myContainer), actually creates the button instance from the Button class.
* Each widget has a set of options that decides its behavior and appearance. This includes attributes such as text labels, colors, font size, and many more. For example, the Button widget has attributes to manage its label, control its size, change its foreground and background colors, change the size of the border, and so on.
* To set these attributes, you can set the values directly at the time of creation of the widget as we have done in the preceding example. Alternatively, you can later set or change the options of the widget by using the .config() or .configure() method. Note that the .config() or .configure() method are interchangeable and provide the same functionality.

You can also add the pack() method on the same line in which you create a new instance of the widget. For example, consider the following code:

mylabel = Label(root,text="I am a label widget")

mylabel.pack()

If you are instantiating the widget directly, you can write both the lines together as follows:

Label(root,text="I am a label widget").pack()

You may keep a reference to the widget created (as in the first example, mylabel) or you can create a widget without keeping any reference to it (as in the second example).

You should ideally keep the reference if the widget content is likely to be modified by some action at a later stage in the program. If the widget state is to remain static after its creation, you need not keep a reference for the widget.

Also, note that calls to pack() (or other geometry managers) always returns None. So, consider you create a widget keeping a reference to it and add the geometry manager (say pack()) on the same line as shown:

mylabel = Label(...).pack()

In this case, you are actually not creating a reference to the widget but instead creating a None type object for the variable mylabel.

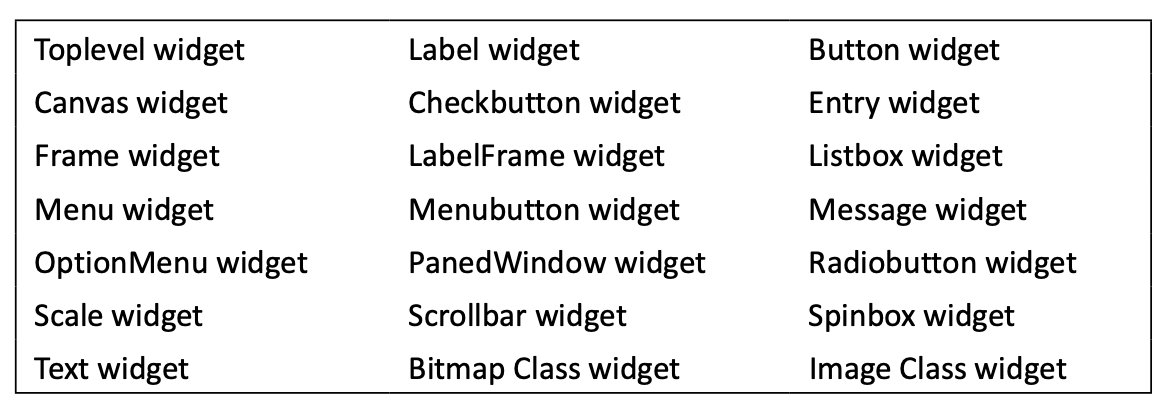
So, when you later try to modify the widget through the reference, you get an error as you are actually trying to work on a None type object.

This is one of the most common errors committed by beginners.

In this iteration, we will get to know all core Tkinter widgets. We have already seen two of them in the previous example—the Label and Button widgets. Let's now see all other core Tkinter widgets.

**Prepare for Lift Off**

Tkinter includes 21 core widgets. These are as follows:



Let's write a program to include these widgets on our root window.

**Engage Thrusters**

The format for adding widgets is the same as we discussed in the previous task. To give you a flavor, here's some sample code for adding some common widgets:

**Label(parent, text=" Enter your Password:")**

**Button(parent, text="Search")**

**Checkbutton(parent, text='RememberMe', variable=v, value=True)**

**Entry(parent, width=30)**

**Radiobutton(parent, text=Male, variable=v, value=1)**

**Radiobutton(parent, text=Female, variable=v, value=2)**

**OptionMenu(parent, var, "Select Country", "USA", "UK", "India", Others")**

**Scrollbar(parent, orient=VERTICAL, command=mytext.yview)**

Can you spot the pattern common to each widget? Can you spot the differences? As a reminder, the syntax for adding a widget is:

**Widget-name (its container window, \*configuration options)**

The method for creating all the previously mentioned widgets is the same. Most of the configuration options will also be similar. However, a few configuration options vary from widget to widget.

For example, the Button and Label widgets will have an option to configure their text, but scrollbars do not have a text-configuration option.

Using the same pattern, let's now add all the 21 core Tkinter widgets into a dummy application (code 01.03.py).

Do not be intimidated by the size of the program. Instead look for a common pattern that is used to initialize and display all the widgets. To reiterate, the syntax for adding a widget is:

*Project 1*

mywidget = Widget-name (container, all widget-options)

Notice how the configuration options for each widget differ slightly from each other depending on the type of widget being initialized.

Refer to the code 1.03.py for a demo of all Tkinter widgets. A summarized code description for 1.03.py is as follows:

f We create a Toplevel window and create a main loop as seen in the earlier examples.

f We add a Frame widget that we named menubar. Note that Frame widgets are just holder widgets that hold other widgets. Frame widgets are great for grouping widgets together. The syntax for adding a frame is the same as that of all other widgets:

myframe = Frame(root)

myframe.pack()

f Keeping the menubar frame as the container, we add two widgets to it, the Menubutton and Menu widgets.

f We create another frame and name it myframe1. Keeping myframe1 as the container/parent widget, we add seven widgets to it:

The Label, Entry, Button, Checkbutton, Radiobutton, OptionMenu, and Bitmap Class widgets.

f We then proceed to create myframe2, another Frame widget. We add six more widgets to it:

The Image Class, Listbox, Spinbox, Scale, LabelFrame, and Message widgets.

We then create myframe3, another Frame widget. We add two more widgets to it, the Text and Scrollbar widgets.

f Finally we create the last frame, myframe4, another Frame widget. We add two more widgets to it, the Canvas and PanedWindow widgets.

All these widgets constitute the 21 core widgets of Tkinter.

Read through the code explanation, and find the corresponding piece of code in the example code 01.03.py. Look at how each widget is created. Try to identify each widget's class name as used in Tkinter. Look what remains the same in all widgets, and what changes between one widget and another?

A few minutes spent reading and understanding the code in 1.03.py will really help you appreciate the simplicity and overall structure of a Tkinter program.

Finally, note that we have used .pack() on each widget to display it inside its container frame. We discuss .pack() in the next task. However, for now just note that we have used something called pack(), without which the widgets would not have displayed at all.

**Objective Complete – Mini Debriefing**

You have reached a major milestone in your GUI programming effort.

You now know all the 21 core widgets of Tkinter. You can identify them by their class names, and you can create them on a root frame or on a subframe within the root. You now know how to configure options of widgets.

With this you have now seen the first and the most important building block of a Tkinter program. You have mastered Tkinter widgets.

**Classified Intel**

Widget options can be set at instantiation time as we have done in the examples so far. Alternatively, the options can be configured after instantiation using the following syntax:

widget.configure(\*\*options)

This is a very handy tool that lets you change widget options dynamically after the widget has been created. We will be using this very often in all our projects.

For common widget configuration options, refer to the *Options common to widgets* section in *Appendix B*, *Quick Reference Sheets*.

**Geometry management**

Having seen all the core Tkinter widgets, let us now turn our attention to the second component of GUI programming—the question of where to place those widgets.

This is taken care of by the geometry manager options of Tkinter. This component of GUI programming involves deciding the position of the widget, overall layout, and relative placement of various widgets on the screen.

**Prepare for Lift Off**

Recall that we used the pack() method for adding widgets to the dummy application we developed in the previous section. pack() is an example of geometry management in Tkinter.

pack() is not the only way you can manage the geometry in your interface. In fact, there are three geometry managers in Tkinter that let you specify the position

of widgets inside a Toplevel or parent window.

The geometry managers are as follows:

f pack: This is the one we have used so far. Simple to use for simpler layouts but may get very complex for slightly complex layouts.

f grid: This is the most commonly used geometry manager that provides a table-like layout of management features for easy layout management.

f place: This is least popular, but provides the best control for absolute positioning of widgets.

**Engage Thrusters**

Let us now see examples of all three geometry managers in action.

**The pack geometry manager**

The pack geometry derives its name from the fact that it literally packs widgets on a first- come-first-serve basis in the space available in the master frame in which widgets are pushed.

The pack geometry manager fits "slave widgets" into "parent spaces". When packing the slave widgets, the pack manager distinguishes between three kinds of spaces:

**Geometry management**

Having seen all the core Tkinter widgets, let us now turn our attention to the second component of GUI programming—the question of where to place those widgets.

f The unclaimed space

f The claimed but unused space f The claimed and used space

The most commonly used options in pack() include:

f side: LEFT, TOP, RIGHT, and BOTTOM (these decide the alignment of the widget) f fill: X, Y, BOTH, and NONE (these decide whether the widget can grow in size)

f expand :1/0 or Yes/No (corresponding to values respectively)

f anchor: NW, N, NE, E, SE, S, SW, W, and CENTER (corresponding to the

cardinal directions)

f Internal padding (ipadx and ipady) and external padding (padx and pady), which all defaulted to a value of zero

Let's take a look at some demo code that illustrates some of the pack features. Here's the code snippet (code 1.04.py) that generates a GUI like the following screenshot:

**from Tkinter import \***

**root = Tk()**

**Button(root, text="A").pack(side=LEFT, expand=YES, fill=Y)**

**Button(root, text="B").pack(side=TOP, expand=YES, fill=BOTH)**

**Button(root, text="C").pack(side=RIGHT, expand=YES, fill=NONE, anchor=NE)**

**Button(root, text="D").pack(side=LEFT, expand=NO, fill=Y)**

**Button(root, text="E").pack(side=TOP, expand=NO, fill=BOTH)**

**Button(root, text="F").pack(side=RIGHT, expand=NO, fill=NONE)**

**Button(root, text="G").pack(side=BOTTOM, expand=YES, fill=Y)**

**Button(root, text="H").pack(side=TOP, expand=NO, fill=BOTH)**

**Button(root, text="I").pack(side=RIGHT, expand=NO)**

**Button(root, text="J").pack(anchor=SE)**

**root.mainloop()**

The description of the code is listed as follows:

When you insert button **A** in the root frame, it captures the left-most area of the frame, it expands, and fills the *Y* dimension. Because expand and fill options are specified in affirmative, it claims all the area it wants and fills the *Y* dimension. If you increase the size of the root window pulling it down, you will notice that the button **A** expands in the downward direction (along the *Y* coordinate) but a side-wise increase in the window does not result in a horizontal increase in the size of button **A**.

f When you insert the next button, **B**, into the root window, it picks up space from the remaining area but aligns itself to TOP, expand-fills the available area, and fills both *X* and *Y* coordinates of the available space.

f The third button, **C**, adjusts to the right-hand side of the remaining space. But because fill is specified as NONE, it takes up only that much space as is required to accommodate the text inside the button. If you expand the root window, the button **C** will not change its size.

f The anchor attribute used in some lines provides a means to position a widget relative to a reference point. If the anchor attribute is not specified, the pack manager places the widget in the center of the available space or the **packing box**. Other allowed options include the four cardinal directions (N, S, E, and W) and a combination of any two directions. Therefore, valid values for the anchor attribute are: CENTER (default), N, S, E, W, NW, NE, SW, and SE.

The description for the rest of the lines is left as an exercise for you to explore. The best way to study this piece of code would be to comment out all lines of code and introduce each successive button one after another. At each step, try to resize the window to see the

effect it has on various buttons.

We will use the pack geometry manager in some of our projects, so it would be a worthwhile exercise to get acquainted with pack and its options.

Note that the value for most of the Tkinter geometry manager attributes can either be specified in capital letters without quotes (like side=TOP, anchor=SE) or in small letters but within quotes (like side='top', anchor='se').

For a complete pack manager reference refer to the *The pack manager* section in *Appendix B*, *Quick Reference Sheets*.

**Where should you use the pack() geometry manager?**

Using the pack manager is somewhat complicated compared to the grid method that we will discuss next, but it is a great choice in situations such as:

f Having a widget fill the complete container frame

f Placing several widgets on top of each other or in a side by side

position (as in the previous screenshot). See code 1.05.py.

While you can create complicated layouts by nesting widgets in multiple frames, you can find

the grid geometry manager more suitable for most of the complex layouts. **The grid geometry manager**

The grid geometry manager is most easy to understand and, perhaps, the most useful geometry manager in Tkinter. The central idea of the grid geometry manager is to divide the container frame into a two-dimensional table divided into a number of rows and columns. Each cell in the table can then be targeted to hold a widget. In this context, a **cell** is an intersection of imaginary rows and columns. Note that in the grid method, each cell can hold only one widget. However, widgets can be made to span multiple cells.

Within each cell you can further align the position of the widget using the STICKY option. The sticky option decides how the widget is expanded, if its container cell is larger than the size of the widget it contains. The sticky option can be specified using one or more of the N, S, E, and W, or NW, NE, SW, and SE options.

Not specifying stickiness defaults to stickiness to the center of the widget in the cell. Let us now see a demo code that illustrates some of the features of the grid geometry

manager. The code in 1.06.py generates a GUI-like figure as shown:

**from Tkinter import \***

**root = Tk()**

**Label(root, text="Username").grid(row=0, sticky=W)**

**Label(root, text="Password").grid(row=1, sticky=W)**

**Entry(root).grid(row=0, column=1, sticky=E)**

**Entry(root).grid(row=1, column=1, sticky=E)**

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

**root.mainloop()**

The description of the code is listed as follows:

Take a look at the grid position defined in terms of rows and column positions for an imaginary grid table spanning the entire frame. See how the use of sticky=W on both labels makes them stick to the west or left-hand side, resulting in a clean layout.

f The width of each column (or height of each row) is automatically decided by the height or width of the widgets contained in the cell. Therefore, you need not worry about specifying the row or column width as equal. You may specify the width for widgets, if you need that extra bit of control.

f You can use the argument sticky=N+S+E+W to make the widget expandable to fill the entire cell of the grid.

In a more complex scenario, your widgets may span across multiple cells in the grid. To enable a grid to span multiple cells, the grid method offers very handy options such as rowspan and columnspan.

Furthermore, you may often need to provide some padding between cells in the grid. The grid manager provides padx and pady options to provide padding to place around the widget in a cell.

Similarly, there are ipadx and ipady options for internal padding. The default value of external and internal padding is 0.

Let us see an example of the grid manager, where we use most of the common arguments to the grid method such as row, column, padx, pady, rowspan, and columnspan in action.

The code 1.08.py is a demonstration of grid() geometry manager options:

**from Tkinter import \***

**top = Tk()**

**top.title('Find & Replace')**

**Label(top,text="Find:").grid(row=0, column=0, sticky='e')**

**Entry(top).grid(row=0,column=1,padx=2,pady=2,sticky='we',columnspan=9)**

**Label(top, text="Replace:").grid(row=1, column=0, sticky='e')**

**Entry(top).grid(row=1,column=1,padx=2,pady=2,sticky='we',columnspan=9)**

**Button(top, text="Find").grid(row=0, column=10, sticky='ew', padx=2,**

**pady=2)**

**Button(top, text="Find All").grid(row=1, column=10, sticky='ew', padx=2)**

**Button(top, text="Replace").grid(row=2, column=10, sticky='ew', padx=2)**

**Button(top, text="Replace All").grid(row=3, column=10, sticky='ew',**

**padx=2)**

**Checkbutton(top, text='Match whole word only').grid(row =2, column=1,**

**columnspan=4, sticky='w')**

**Checkbutton(top, text='Match Case').grid(row =3, column=1, columnspan=4,**

**sticky='w')**

**Checkbutton(top, text='Wrap around').grid(row =4, column=1, columnspan=4,**

**sticky='w')**

**Label(top, text="Direction:").grid(row=2, column=6, sticky='w')**

**Radiobutton(top, text='Up', value=1).grid(row=3, column=6, columnspan=6,**

**sticky='w')**

**Radiobutton(top, text='Down', value=2).grid(row=3, column=7,**

**columnspan=2, sticky='e')**

**top.mainloop()**

Notice how just 14 lines of core grid manager code generates a complex layout such

as the one shown in the following screenshot. In contrast, developing this with the pack manager would have been much more tedious:

Another grid option that you can sometimes use is the widget.grid\_forget()

method. This method can be used to hide the widget from the screen. When you use this option, the widget exists in its place but becomes invisible. The hidden widget may be made visible again but any grid options that you had originally assigned to the widget will be lost.

Similarly, there is a widget.grid\_remove() method that removes the widget, except that in this case when you make the widget visible again, all its grid options will be restored.

For a complete grid() reference, refer to the the *The grid manager* section in *Appendix B*, *Quick Reference Sheets*.

**Where should you use the grid() geometry manager?**

The grid manager is a great tool for developing complex layouts. Complex structures can be easily achieved by breaking the container widget into grids of rows and columns and then placing the widgets in grids where they are wanted.

It is also commonly used in developing different kinds of dialog boxes.

Now we will delve into configuring grid column and row sizes.

Different widgets have different heights and widths. So when you specify the position

of a widget in terms of rows and columns, the cell automatically expands to accommodate the widget.

Normally the height of all grid rows is automatically adjusted to be the height of its tallest cell. Similarly, the width of all grid columns is adjusted to be equal to the width of the widest widget cell.

If you then want a smaller widget to fill a larger cell or to stay at any one side of the cell, you use the sticky attribute on the widget to control that.

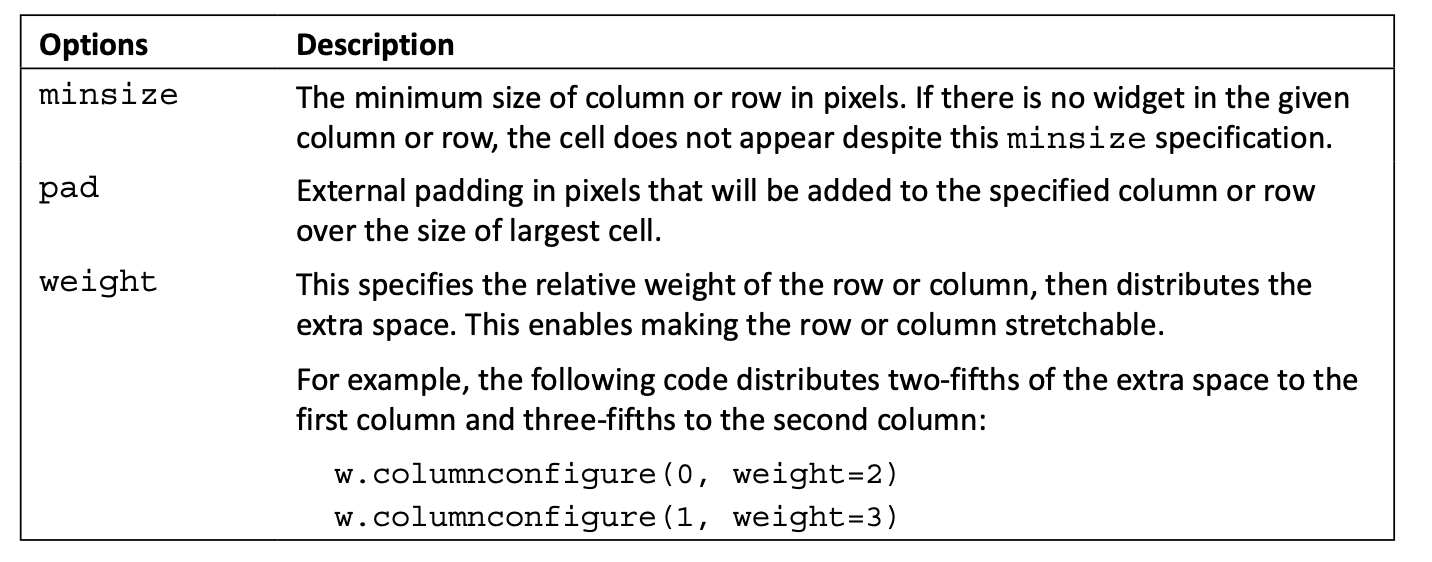
You can, however, override this automatic sizing of columns and rows using the following code:

w.columnconfigure(n, option=value, ...) AND

w.rowconfigure(N, option=value, ...)

Use these to configure the options for a given widget, w, in the column, n, specifying values for the options, minsize, pad, and weight.

The options available here are as mentioned in the following table:



The columnconfigure() and rowconfigure() methods are often used to implement dynamic resizing of widgets, especially on resizing the root window.

You cannot use grid and pack methods together in the same container window. If you try doing that, your program will enter into an infinite negotiation loop.

**The place geometry manager**

The place geometry manager is the most rarely used geometry manager in Tkinter. Nevertheless, it has its uses in that it lets you precisely position widgets within

its parent frame using the *X*-*Y* coordinate system.

The place manager can be assessed using the place() method on all standard widgets. The important options for place geometry include:

f Absolute positioning (specified in terms of x=N or y=N)

f Relative positioning (key options include relx, rely, relwidth, and relheight)

Other options commonly used with place() include width and anchor (the default is NW). Refer to the code in 1.09.py for a demonstration of the common place option:

from Tkinter import \*

root = Tk()

# Absolute positioning

Button(root,text="Absolute Placement").place(x=20, y=10)

# Relative positioning

Button(root, text="Relative").place(relx=0.8, rely=0.2, relwidth=0.5,

width=10, anchor = NE)

root.mainloop()

You may not see much of a difference between absolute and relative positions simply

by looking at the code or the window frame. If, however, you try resizing the window,

you will notice that the button placed absolutely does not change its coordinates, while the relative button changes its coordinates and size to fit the new size of the root window.

For a complete place() reference, check out the *The place manager* section in *Appendix B*, *Quick Reference Sheets*.

**When should you use the place manager?**

The place manager is useful in situations where you have to implement the custom geometry managers where the widget placement is decided by the end user.

While pack() and grid() managers cannot be used together in the same frame, the place() manager can be used with any other geometry manager within the same container frame.

The place manager is rarely used. This is because if you use it you have to worry about the exact coordinates. If say you make a minor change for one widget, it is very likely that you will have to change the *X*-*Y* values for other widgets as well, which can be very cumbersome.

We will not use the place manager in our projects. However, knowing that options for coordinate-based placement exist can be helpful in certain situations.

**Objective Complete – Mini Debriefing**

This concludes our discussion on geometry management in Tkinter.

In this section you implemented examples of pack, grid, and place geometry managers. You also understood the strength and weaknesses of each geometry manager.

You learned that pack is best for a simple side-wise or top-down widget placement. You also saw that the grid manager is best suited for handling complex layouts. You saw examples of the place geometry manager and the reasons why it is rarely used.

You should now be in a position to plan and execute different layouts for your programs using these geometry managers of Tkinter.

**Events and callbacks – adding life to**

**programs**

Now that we have learned how to add widgets to our screen and how to position them where we want, let's turn our attention to the third component of GUI programming. This addresses the question of how to make the widgets functional.

Making the widgets functional involves making them responsive to events such as the pressing of buttons, the pressing keys on keyboards, mouse clicks, and the like. This requires associating callbacks to specific events.

**Engage Thrusters**

Callbacks are normally associated with specific widget events using the command binding the rules, which is elaborated on in the following section.

**Command binding**

The simplest way to add functionality to a button is called command binding, whereby the callback function is mentioned in the form of command = some\_callback in the widget option.

Take a look at the following sample code:

def my\_callback ():

# do something

Button(root,text="Click",command= my\_callback)

Note that my\_callback is called without parentheses () from within the widget command option. This is because when the callback functions are set, it is necessary to pass a reference to a function rather than actually calling it.

**Passing arguments to the callback**

If the callback does not take any argument, it can be handled with a simple function like the one we just used. However, if the callback needs to take some arguments, we can use the lambda function as shown in the following code snippet:

def my\_callback (somearg):

#do something with argument

Button(root,text="Click",command=lambda: my\_callback

('some argument'))

Python borrows syntax from a functional program called the lambda function. The lambda function lets you define a single-line, nameless function on the fly.

The format for using lambda is lambda arg: #do something with arg in a single line, for instance:

lambda x: return x^2

Please note that the command option available with the Button widget is really an alternative function to ease programming the Button event. Many other widgets do not provide any equivalent command binding option.

The command button binds by default to the left mouse click and the Space bar. It does not bind to the Return key. Therefore, if you bind a button using the command function, it will react to the Space bar and not the Return key. This is counter-intuitive to many Windows users. What's worse is you cannot change this binding of the command function. The moral is that command binding, though a very handy tool, does not provide you the the independence to decide your own bindings.

**Event binding**

Fortunately, Tkinter provides an alternative form of event binding mechanism called bind() to let you deal with different events. The standard syntax for binding an event is as follows:

widget.bind(event, handler)

When an event corresponding to the event description occurs in the widget, it calls the associated handle passing an instance of the event object as the argument, with the event details.

Let us look at an example of the bind() method (refer to the code file 1.10.py):

*Project 1*

from Tkinter import \*

root = Tk()

Label(root, text='Click at different\n locations in the frame below'). pack()

**def mycallback(event):**

**print dir(event)**

**print "you clicked at", event.x, event.y**

myframe = Frame(root, bg='khaki', width=130, height=80)

**myframe.bind("<Button-1>", mycallback)**

myframe.pack()

root.mainloop()

The description of the code is listed as follows:

f We bind the Frame widget to the event, <Button-1>, which corresponds to left-click of the mouse. On the occurrence of this event, it calls the function mycallback, passing along an object instance as its argument.

f We define the function mycallback(event). Notice that it takes the event object generated by the event as the argument.

f We inspect the event object using dir(event), which returns a sorted list of attribute names for the event object passed to it. This prints the list:

['\_\_doc\_\_', '\_\_module\_\_', 'char', 'delta', 'height', 'keycode', 'keysym', 'keysym\_num', 'num', 'send\_event', 'serial', 'state', 'time', 'type', 'widget', 'width', 'x', 'x\_root', 'y', 'y\_root'].

f Out of the attributes list generated by the object, we use two attributes, event.x and event.y, to print the coordinates of the point of click.

When you run this code, it produces a window like the one shown. When you left-click anywhere in the frame, it outputs messages to the console. A sample message passed to the console is as follows:

**['\_\_doc\_\_', '\_\_module\_\_', 'char', 'delta', 'height', 'keycode', 'keysym',**

**'keysym\_num', 'num', 'send\_event', 'serial', 'state', 'time', 'type',**

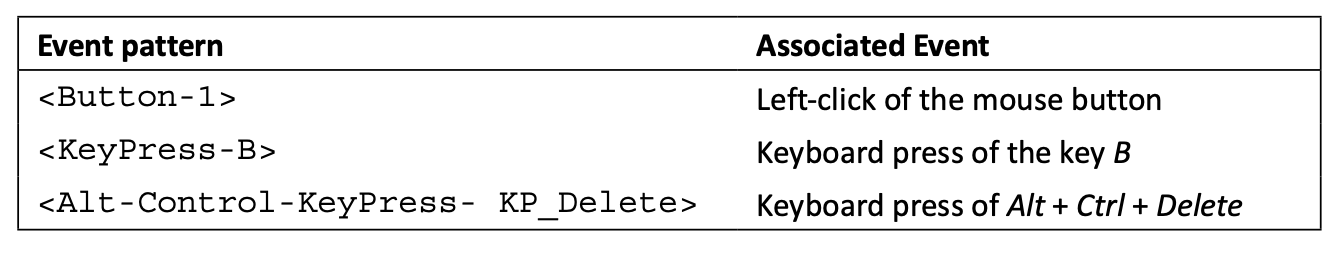
**'widget', 'width', 'x', 'x\_root', 'y', 'y\_root']**

You clicked at 63 36.

**Event pattern**

In the previous example, you saw how we used the event <Button-1> to denote the left- click of a mouse. This is a built-in pattern in Tkinter that maps it to the mouse's left-click event. Tkinter has an exhaustive mapping scheme that exactly identifies events such as this one.

Here are some examples to give you an idea of event patterns:



In general, the mapping pattern takes the following form:

**<[event modifier-]...event type [-event detail]>**

Typically an event pattern will comprise of:

f **An event type** (required): Some common event types include Button, ButtonRelease, KeyRelease, Keypress, FocusIn, FocusOut, Leave (mouse leaves the widget), and MouseWheel. For a complete list of event types, refer to the *The event types* section in *Appendix B*, *Quick Reference Sheets*.

f **An event modifier** (optional): Some common event modifiers include Alt, Any (used like in <Any-KeyPress>), Control, Double (used like in <Double- Button-1> to denote a double-click of the left mouse button), Lock, and Shift. For a complete list of event modifiers, refer to the *The event modifiers* section in *Appendix B*, *Quick Reference Sheets*.

f **The event detail** (optional): The mouse event detail is captured by number 1 for

a left-click and number 2 for a right-click. Similarly, each keyboard keypress is

either represented by the key letter itself (say *B* in <KeyPress-B>) or using a key symbol abbreviated as **keysym**. For example, the up arrow key on the keyboard is represented by the keysym value of KP\_Up. For a complete keysym mapping, refer to the *The event details* section in *Appendix B*, *Quick Reference Sheets*.

Let's take a look at a practical example of the event binding on widgets. (See the code in 1.11.py for the complete working example). The following is a modified snippet of code to give you a flavor of the commonly used the event bindings:

**widget.bind("<Button-1>",callback) #bind widget to left mouse click**

**widget.bind("<Button-2>", callback) # bind to right mouse click**

**widget.bind("<Return>", callback)# bind to Return(Enter) Key**

**widget.bind("<FocusIn>", callback) #bind to Focus in Event**

**widget.bind("<KeyPress-A>", callback)# bind to keypress A**

**widget.bind("<KeyPress-Caps\_Lock>", callback)# bind to CapsLockkeysym**

**widget.bind("<KeyPress-F1>", callback)# bind widget to F1 keysym**

**widget.bind("<KeyPress-KP\_5>", callback)# bind to keypad number 5**

**widget.bind('<Motion>', callback) # bind to motion over widget**

**widget.bind("<Any-KeyPress>", callback) # bind to any keypress**

Rather than binding an event to a particular widget, you can also bind it to the Toplevel window. The syntax remains the same except that now you call it on the root instance of the root window like root.bind().

**Levels of binding**

In the previous section, you saw how to bind an event to an instance of a widget. This can be called **instance level binding**.

However, there might be times when you need to bind events to the entire application. At other times you may want to bind the event to a particular class of widget. Tkinter provides different levels of binding options for this:

f **An application-level binding**: Application-level bindings will let you use the same binding across all windows and widgets of the application, as long as any one window of the application is in focus.

The syntax for application-level bindings is:

w.bind\_all(event, callback)

The typical usage pattern is as follows:

root.bind\_all('<F1>', show\_help)

An application-level binding here means that no matter what widget is under the current focus, a press of the *F1* key will always trigger the show\_help callback as long as the application is under active focus.

**A class-level binding**: You can also bind events at a particular class level. This is normally used to set the same behavior of all instances of a particular widget class.

This syntax for class level binding is as follows:

w.bind\_class(className, event, callback)

The typical usage pattern is as follows:

myentry.bind\_class('Entry', '<Control-V>', paste)

In the preceding example, all entry widgets will be bound to the <Control-V>

event that would call a method called 'paste (event)'.

**Event propagation**

Most of the keyboard events and mouse events occur at the operating system level. It propagates from the source of the event, hierarchically up, until it finds a window that has a corresponding binding. The event propagation

does not stop there. It propagates itself upwards looking for other bindings from other widgets until it reaches the root window. If it does reach the root window and no bindings are discovered by it, the event is disregarded.

**Handling widget-specific variables**

You need variables with a wide variety of widgets. You likely need a string variable to track what the user enters into the entry widget or text widget. You most probably need Boolean variables to track whether the user has checked the Checkbox widget. You need integer variables to track the value entered in a Spinbox or Slider widget.

In order to respond to changes in widget-specific variables, Tkinter offers its own variable class. The variable that you use to track widget-specific values must be subclassed from this Tkinter variable class. Tkinter offers some commonly used predefined variables. They are StringVar, IntVar, BooleanVar, and DoubleVar.

You can use these variables to capture and play with changes in the value of variables from within your callback functions. You can also define your own variable type, if required.

Creating a Tkinter variable is simple. You simply call the required constructor:

**mystring = StringVar()**

**ticked\_yes = BooleanVar()**

**option1 = IntVar()**

**volume = DoubleVar()**

Once the variable is created, you can use it as a widget option, as follows:

**Entry(root, textvariable = mystring)**

**Checkbutton(root, text="Remember Me", variable=ticked\_yes)**

**Radiobutton(root, text="Option1", variable=option1, value="option1")**

**#radiobutton**

**Scale(root, label="Volume Control", variable=volume, from =0, to=10) #**

**slider**

Additionally, Tkinter provides access to the value of variables using set() and get() methods:

**myvar.set("Wassup Dude") # setting value of variable**

**myvar.get() # Assessing the value of variable from say a callback**

A demonstration of the Tkinter variable class is available in the code file 1.12.py. The code generates a window like the following screenshot:

**Objective Complete – Mini Debriefing**

In this lesson, you learned:

The command binding to bind simple widgets to certain functions

Use of the lambda function, if you need to process arguments

The event binding using the widget.bind(event, callback) method to bind keyboard and mouse events to your widgets and to invoke callbacks

on the occurrence of some events

f How to pass extra arguments to a callback

f How to bind events to an entire application or to a particular class of widget

using bind\_all() and bind\_class()

f How to use the Tkinter variable class to set and get values of widget

specific variables

In short you now know how to make your GUI program functional!

**Classified Intel**

In addition to the bind method we previously saw, you might find these two event-related options useful in certain cases:

f unbind: Tkinter provides the unbind options to undo the effect of an earlier binding. The syntax is as follows:

A demonstration of the Tkinter variable class is available in the code file 1.12.py. The code generates a window like the following screenshot:

**Objective Complete – Mini Debriefing**

In this lesson, you learned:

f The command binding to bind simple widgets to certain functions

f Use of the lambda function, if you need to process arguments

widget.unbind(event)

The following are some examples of its usage:

entry.unbind('<Alt-Shift-5>')

root.unbind\_all('<F1>')

root.unbind\_class('Entry', '<KeyPress-Del>')

**Virtual events**: Tkinter also lets you create your own events. You can give these virtual events any name you want.

For example, imagine you want to create a new event called <<commit>>, which is triggered by the *F9* key. To create this virtual event on a given widget, use the syntax:

widget.event\_add('<<commit>>', '<F-9>')

You can then bind <<commit>> to any callback using a normal bind() method like: widget.bind('<<commit>>', callback)

Other event-related methods are listed in the *Other event-related methods* section in *Appendix B*, *Quick Reference Sheets*.

Now that you are ready to dive into real application development with Tkinter, let's spend some time exploring a few custom styling options that Tkinter offers. We will also see some of the configuration options commonly used with the root window.

**Doing it in style**

So far, we have have relied on Tkinter to provide specific platform-based styling for our widgets. However, you can specify your own styling of widgets in terms of their color, font size, border width, and relief. A brief introduction of styling features available in Tkinter is covered in the following task.

**Prepare for Lift Off**

Recall that we could specify widget options at the time of its instantiation as shown:

mybutton = Button(parent, \*\*configuration options) Alternatively, you could specify widget options using configure ():

mybutton.configure(\*\*options)

Styling options are also specified as options to the widgets, either at the time of instantiation or later using the configure option.

**Engage Thrusters**

Under the purview of styling, we will cover how to apply different colors, fonts, border width, relief, cursor, and bitmap icons to our widgets. We will also look at some of the root configurations later in the section.

Let's first see how to specify color options for a widget. You can specify two types of color for most of the widgets:

f Background color f Foreground color

You can specify the color using hexadecimal color codes using the proportion of red, green, and blue. Commonly used representations are #rgb (4 bits), #rrggbb (8 bits), and #rrrgggbbb (12 bits).

For example, #fff is white, #000000 is black, and #fff000000 is red.

Alternatively, Tkinter provides mapping for standard color names. For a list of predefined colors, open the program titled pynche in the Tools folder within your Python installation directory (in my case, C:\Python27\Tools\pynche). Within the program click on **View** | **Color list Window**.

Next, the easiest and the most common way to specify a font is to represent it as a tuple. The standard representation is as follows:

widget.configure( font= 'font family, fontsize, optional style

modifiers like bold, italic, underline and overstrike')

Here are some examples to illustrate the method for specifying fonts:

widget.configure (font='Times, 8')

widget.configure (font = 'Helvetica 24 bold italic')

If you set a Tkinter dimension in a plain integer, the measurements takes place in units of pixel. Alternatively, Tkinter accepts four other measurement units which are: m (millimeters), c (centimeters), i (inches), and p (printer's points, which is about 1/72").

The default border width for most Tkinter widgets is 2 pixels. You can change the border width of the widgets by specifying it explicitly, as shown in the following line:

button.configure (borderwidth=5)

The relief style of a widget refers to the difference between the highest and lowest elevations in a widget. Tkinter offers five possible relief styles: flat, raised, sunken, groove, and ridge.

Let's first see how to specify color options for a widget. You can specify two types of color for most of the widgets:

f Background color f Foreground color

button.configure (relief='raised')

Tkinter lets you change the style of mouse cursor when you hover over a particular widget. This is done using the option cursor as in the following example:

button.configure (cursor='cross')

For a complete list of available cursors, refer to the *List of available cursors* section in *Appendix B*, *Quick Reference Sheets*.

While you can specify styling options at each widget level, sometimes it may be cumbersome to do so individually for each widget. Widget-specific styling has several disadvantages:

f It mixes logic and presentation into one file making the code bulky and difficult to manage

f Any change in styling is to be applied to each widget individually

f It violates the **don't repeat yourself** (**DRY**) principle of effective coding as you

keep specifying the same style for a large number of widgets

Fortunately, Tkinter now offers a way to separate presentation from the logic and to specify styles in what is called the external "option database". This is nothing but a text file where you can specify the common styling options.

A typical option database text file may look like the following:

\*font: Arial 10

\*Label\*font: Times 12 bold

\*background: AntiqueWhite1

\*Text\*background: #454545

\*Button\*foreground:gray55

\*Button\*relief: raised

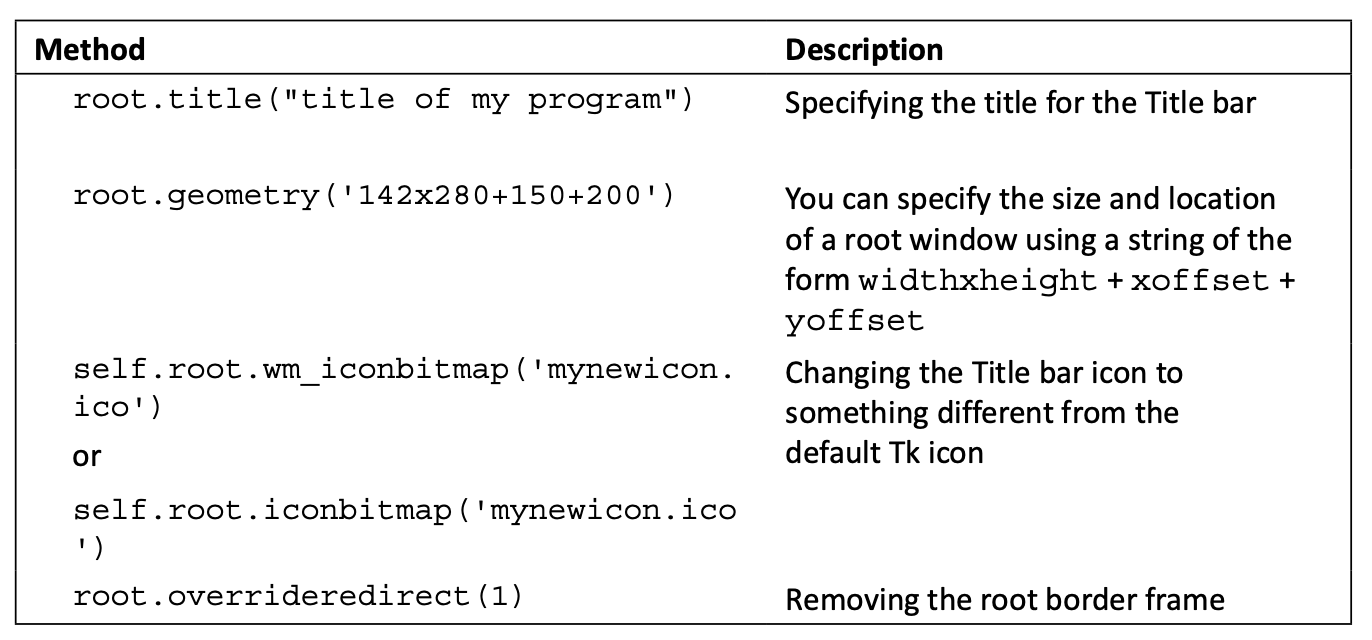
\*Button\*width: 3

The asterisk (\*) symbol here means that the particular style applies to all instances of the given widget.

These entries are placed in an external text (.txt) file. To apply this styling to a particular piece of code, you simply call it using the option\_readfile() call early in your code, as shown here:

root.option\_readfile('optionDB.txt')

Now that we are done discussing styling options, let us wrap up with a discussion on some commonly used options for the root window:



p52