# Tkinter (Part One)

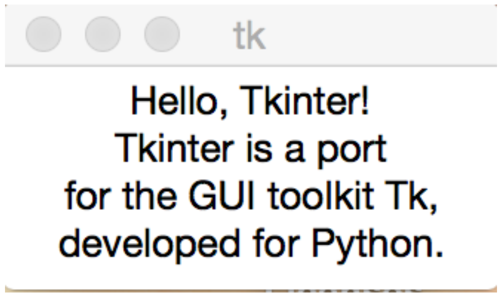
Tk is a GUI extension developed for Tcl, a scripting language. Because it became very popular in the 1990s, and many programmers wanted to use it outside the scripting language it had been developed for, ports for various other languages were developed, such as TASH (for Ada), Tkinter (for Pascal), and others for Perl, Ruby, and Common Lisp.

## The Hello, Tkinter! Widget

One of the simplest Tkinter widgets is the label. A **label** is a Tkinter Widget class, which can display text or an image, and can be viewed, but is not interactive. The following example implements this widget:

```
#!/usr/bin/python

from Tkinter import *

root=Tk()

myString="Hello, Tkinter!\n"+"Tkinter is a port\n"+"for the GUI
toolkit Tk,\n"+"developed for Python."
widget=Label(root, text=myString)
widget.pack()

root.mainloop()
```

This produces the following widget:



```
Hello, Tkinter!
Tkinter is a port
for the GUI toolkit Tk,
developed for Python.
```

The root is the **Tk root widget**, a window with a title bar, provided by the window manager. The root widget must be created before any other widgets, and there can only be one root widget. The second widget is the **Label widget**, generated with a call to the constructor to the class Label, which takes two args:

the parent window—in this case `root`
the text to show—in this case, the string `myString`

Finally, the `pack()` method sizes the window to the minimum available size that still fits the text provided. The call to the `mainloop()` method of object root will generate the event loop that keeps the window open as long as the user doesn't close it.

## Tkinter Widgets

Our simple Hello, Tkinter! widget is only one example of the many such interfaces available from Tkinter. Tkinter produces several graphical controls to be used for applications written for the windowing type of graphical interface. These are summarized in the following table:

| Graphical Control | Behavior |
|---|---|
| **`Button`** | The Button widget displays a button in the application. |
| **`Canvas`** | The Canvas widget draws shapes (lines, ovals, rectangles) in the application. |
| **`Checkbutton`** | The Checkbutton widget allows the selection of options through checkboxes. The user may select multiple options at the same time. |
| **`Entry`** | The Entry widget displays a single line textfield which accepts input text values from the user. |
| **`Frame`** | The Frame widget is used as a container for organizing other widgets. |
| **`Label`** | The Label widget produces a single line caption for other widgets. It can also contain images. |
| **`Listbox`** | The Listbox widgets produces a list of options, of which the user can select one. |
| **`Menubutton`** | The Menubutton widget displays menus in the application. |
| **`Menu`** | The Menu widget provides the menus inside the Menubutton widget. |

| Graphical Control | Behavior |
|---|---|
| **`Message`** | The Message widget produces multiline text fields for accepting user input values. |
| **`Radiobutton`** | The Radiobutton widget produces multiple options as radio buttons. Unlike the Checkbutton widget, the Radiobutton only allows the selection of a single option. |
| **`Scale`** | The Scale widget provides a slider to scale things. |
| **`Scrollbar`** | The Scrollbar widget adds scrolling capability to widgets, such as to list boxes. |
| **`Text`** | The Text widget produces text on multiple lines. |
| **`Toplevel`** | The Toplevel widget provides a separate window container. |
| **`Spinbox`** | The Spinbox widget is a variant of the Tkinter Entry widget, which can be used to select from a fixed number of values. |
| **`PanedWindow`** | The PanedWindow widget is a container widget which may contain a number of panes, arranged horizontally or vertically. |
| **`LabelFrame`** | The LabelFrame widget is a container widget, whose purpose is to act as a spacer or container for complex window layouts. |
| **`txMessageBox`** | The txMessageBox widget displays message boxes in applications. |

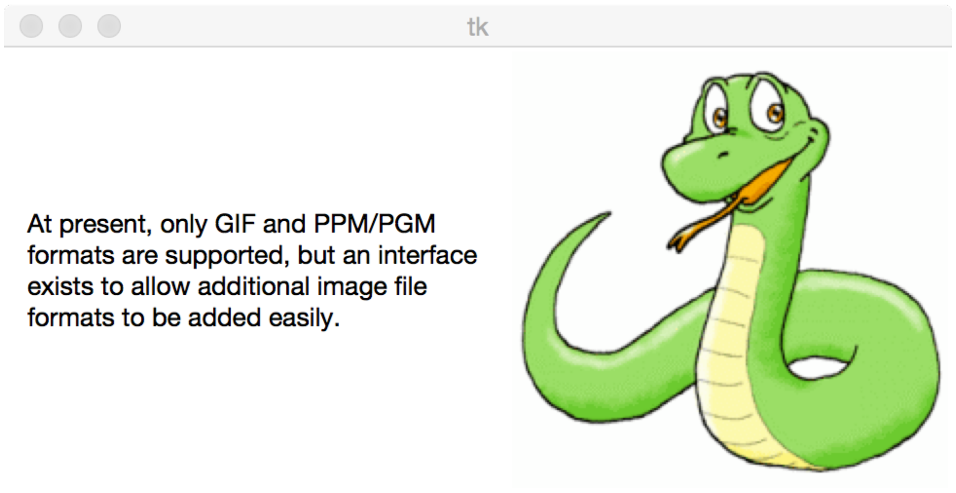
## The Label Widget

We used the label widget in our previous simples Tkinter example. In the following example, we will add an image to the label:

```
#!/usr/bin/python

from Tkinter import *

root=Tk()
```

```
logo=PhotoImage(file="/Users/awise/Python/Tkinter/images/
python.gif")
w1=Label(root, image=logo).pack(side="right")
explanation="""At present, only GIF and PPM/PGM
formats are supported, but an interface
exists to allow additional image file
formats to be added easily."""
w2=Label(root, justify=LEFT, padx = 10,
text=explanation).pack(side="left")
root.mainloop()
```

In this code, there are two Label widgets, `w1` and `w2`, one image, one text. The output is shown below:



At present, only GIF and PPM/PGM
formats are supported, but an interface
exists to allow additional image file
formats to be added easily.

The syntax for the Label widget is as follows:

```
w=Label(master, option, …)
```

where the arguments are:
`master`—the parent window
`options`—one of the options listed in the following table:

| Option | Description |
|---|---|
| **anchor** | Controls where the text is positioned if the widget (Label) has more space than the text needs. The default is anchor=CENTER |

| Option | Description |
|---|---|
| **`bg`** | The background color displayed behind the label . |
| **`bitmap`** | If this option is set to a bitmap or an image file, the label will display it. |
| **`bd`** | The size of the border around the indicator. The default is 2 pixels. |
| **`cursor`** | If this option is set to a cursor name (arrow, dot etc.), the mouse cursor will change to that pattern when it goes over the check button. |
| **`font`** | If text is displayed in the label (with the text or the textvariable option), this option sets the font for it. |
| **`fg`** | If text is displayed in the label (with the text or the textvariable option), this option sets the color for it. |
| **`height`** | The vertical dimension of the new frame. |
| **`image`** | This option sets the image to be displayed. |
| **`justify`** | Sets the justification of text. The options are: CENTER, LEFT, or RIGHT. The default it CENTER. |
| **`padx`** | Allows horizontal padding around the text within the label widget. The default is 1 space. |
| **`pady`** | Allows vertical padding around the text within the label widget. The default is 1 space. |
| **`relief`** | The decorative border around the label. The default is FLAT. |
| **`text`** | This option is set to the text string to be displayed by the label, if the label displays static (hardcoded) text. |
| **`textvariable`** | This option is set to the text string to be displayed by the label, if the label displays dynamic (sourced from a variable) text. |
| **`underline`** | Displays an underline between the 0th and nth character positions within the text. The default is -1, meaning no underline. |

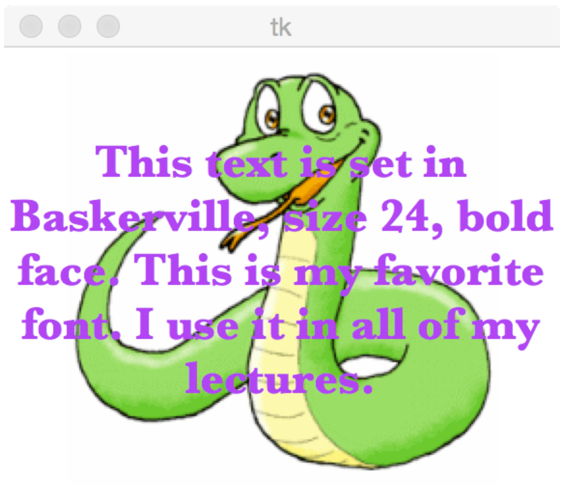
| Option | Description |
|---|---|
| **`width`** | The width of the label in # of characters. If not set, the widget will be sized to fit its contents (text+images). |
| **`wraplength`** | The max limit of # of characters per line. |

Another option not specified in this table is `compound`. We can use this to display both text and image within the same Label, to create an overlapping effect:

```
from Tkinter import *

root=Tk()
logo=PhotoImage(file="/Users/awise/Python/Tkinter/images/
python.gif")
explanation="This text is set in Baskerville, size 24, bold
face. This is my favorite font. I use it in all of my lectures."
w=Label(root,
          compound=CENTER,
          text=explanation,
          font='Baskerville 24 bold',
          fg='purple',
          wraplength=300,
          image=logo).pack(side="right")

root.mainloop()
```

The text fonts can be specified either in system-supplied available fonts (e.g. Times, Courier, Helvetica, Baskerville etc.), or in a structured format (e.g. TkDefaultFont, TkTextFont, TkFixedFont, TkMenuFont, TkHeadingFont etc.). There are online references with available fonts for various systems.

The code above produces the following application window with a label widget:



This text is set in Baskerville, size 24, bold face. This is my favorite font. I use it in all of my lectures.

## The Button Widget

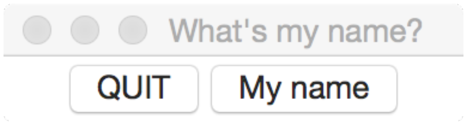
The Button widget can be associated with some action, which should be executed when the button is pressed. The action is specified as a Python function or method.

The following simple example creates two buttons, one of which, when pressed, displays a message on the command line, and the other quits the window:

```
#!/usr/bin/python

from Tkinter import *

class App:
	def __init__(self, master):
		frame=Frame(master)
		frame.pack()
		self.button=Button(frame,
				text="QUIT",
				fg="red",
				command=frame.quit)
		self.button.pack(side=LEFT)
		self.slogan=Button(frame,
				text="My name",
				command=self.write_slogan())
		self.slogan.pack(side=LEFT)
	def write_slogan(self):
		print "My name is Adriana WISE."

root=Tk()
root.title("What's my name?")
app=App(root)
root.mainloop()
```

Class `App` is instantiated with one argument, `master`, which gets passed in as `root` (the root window) when the class constructor is called to create the object `app`. The `Frame` constructor instantiates the object `frame`, which organizes the two Button widgets within the parent (`root`) window with the `pack()` method.

This outputs the following window:

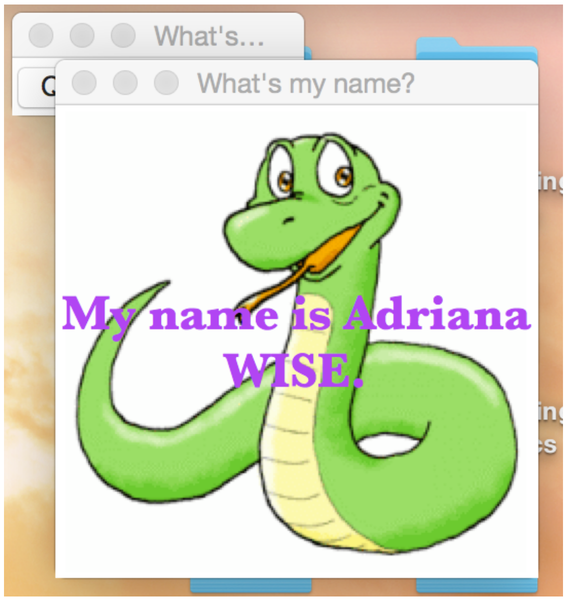


When clicked, the QUIT button quits the window, by having its attribute `command` set to `frame.quit`, while the “My name" button prints the slogan at the command line. The root window is set to display the title “What’s my name?”.

We want to make the button “My name” start a new window instead of writing to the command line, since this is what we would like a button to do in most applications.

```
!/usr/bin/python

from Tkinter import *

class App:
        def __init__(self, master):
                frame=Frame(master)
                frame.pack()
                self.title="Adriana"
                self.button=Button(frame,
                                text="QUIT",
                                fg="red",
                                command=frame.quit)
                self.button.pack(side=LEFT)
                self.slogan=Button(frame,
                                text="My name",
                                command=create_window)
                self.slogan.pack(side=LEFT)

def create_window():
        window=Toplevel(root)
        logo=PhotoImage(file="/Users/awise/Python/Tkinter/
images/python.gif")
        explanation="My name is Adriana WISE."
        widget=Label(window,
                compound=CENTER,
                text=explanation,
                font='Baskerville 24 bold',
                fg='purple',
                wraplength=300,
                image=logo).pack(side=“right")
        widget.logo=logo

root=Tk()
root.title("What's my name?")
app=App(root)
root.mainloop()
```

The output using this code is a root window with the two buttons, and a next level window containing the label widget and the text we want to display:



My name is Adriana WISE.

The syntax for the Button widget is:

```
widget=Button(master, option=value, …)
```

where:

`master`—parent window (window containing the button)
`option`—one of the following:

| Option | Description |
|---|---|
| **`activebackground`** | Background button color when the mouse is rolled over it. |
| **`activeforeground`** | Foreground button color when the mouse is rolled over it. |
| **`bd`** | Border width in pixels. Default is 2. |
| **`bg`** | Passive background color (when mouse is away from button). |
| **`font`** | Text font for the button. |

| Option | Description |
|---|---|
| **`height`** | Height of the button in text lines (for text labels) and in pixels (for image labels). |
| **`highlightcolor`** | The color of the highlight when the widget is in focus. |
| **`image`** | Image to be displayed as button label (instead of text). |
| **`justify`** | Justifies multiple text lines on the button. Choices are LEFT, CENTER, and RIGHT. |
| **`padx`** | Padding left and right of the text label. |
| **`pady`** | Padding above and below the text label. |
| **`relief`** | Type of the border. Choices are SUNKEN, RAISED, GROOVE, and RIDGE. |
| **`state`** | State of the button. Choices are DISABLED (grayed out), ACTIVE (mouse rolls over it), and NORMAL (default). |
| **`underline`** | Displays button label underlined between the specified character positions. Default is -1 (no underline). |
| **`width`** | Width of the button in # of characters (if label is text), or in # of pixels (if label is image). |
| **`wraplength`** | Sets the width in characters that the text lines for the button label should wrap around. |

## Variable Classes

Data entry widgets (text boxes, radio buttons, drop down lists) can be connected directly to application variables by using one of the following options to these widgets:

- `variable`
- `textvariable`
- `onvalue`
- `offvalue`
- `value`

Variables whose values can be passed from the Tkinter GUI to the backend script of the application have to be subclassed from a class called `Variable`, defined in the Tkinter module, and thus available to your script through the `import Tkinter` statement. These variables are declared as shown in the following table:

| Variable Objects | Description |
|---|---|
| **`x=StringVar()`** | Holds a string. Defaults to NULL. |
| **`x=IntVar()`** | Holds an integer. Defaults to 0. |
| **`x=DoubleVar()`** | Holds a float. Defaults to 0.0. |
| **`x=BooleanVar()`** | Holds a Boolean. Defaults to 0 or FALSE. |

To read the value of a variable set through a data entry widget, you call the method `get()`. To set the value of such a variable, call the method `set()`.

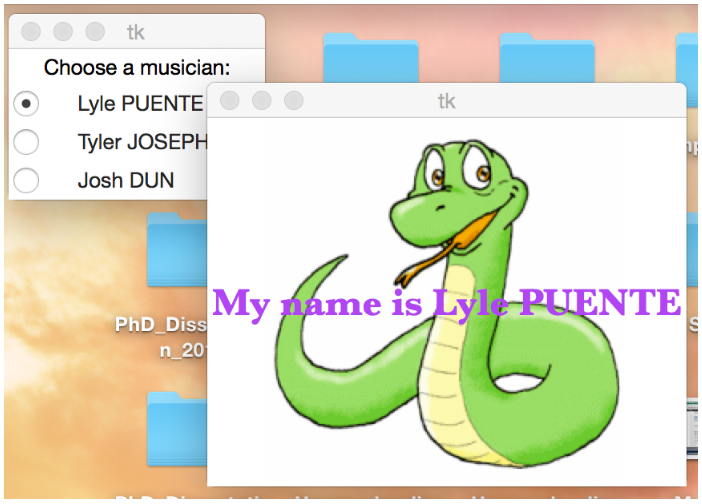
These variable classes were introduced at this point (without examples yet), in order to prepare the terrain for data entry widgets, which follow next.

## The Radiobutton Widget

The Radiobutton Widget is used to accept an option from a list of options, similar to the way you would check a text box on a paper form. In the following example, we select one out of three musician names and capture the value into the value option of the Radiobutton object. This value is then provided to the function `create_window()`, in order to personalize the target pop-up window to the musician name that was chosen:

```
from Tkinter import *

root=Tk()
v=StringVar()

def create_window():
        window=Toplevel(root)
        logo=PhotoImage(file="/Users/awise/Python/Tkinter/
images/python.gif")
        whichsinger=v.get()
        explanation="My name is %s" % whichsinger
        widget=Label(window,
                compound = CENTER,
                text=explanation,
                font='Baskerville 24 bold',
                fg='purple',
                wraplength=300,
                image=logo).pack(side="right")
        window.logo=logo


Label(root,
      text="""Choose a musician:""",
      justify=LEFT,
      padx=20).pack()
button1=Radiobutton(root,
            text="Lyle PUENTE",
            padx=20,
            variable=v,
            value="Lyle PUENTE",
            command=create_window).pack(anchor=W)
button2=Radiobutton(root,
            text="Tyler JOSEPH",
            padx=20,
            variable=v,
            value="Tyler JOSEPH",
            command=create_window).pack(anchor=W)
button3=Radiobutton(root,
            text="Josh DUN",
            padx=20,
            variable=v,
            value="Josh DUN",
            command=create_window).pack(anchor=W)

mainloop()
```

This produces the following windows:



The following table shows the options and their descriptions for the Radiobutton widget:

| Option | Description |
|---|---|
| **activebackground** | Background color when the mouse is rolled over the radiobutton. |
| **activeforeground** | Foreground color when the mouse is rolled over the radiobutton. |
| **anchor** | If the space available is larger than the widget, this option specifies where the radiobutton is placed within that space. The default is CENTER. |
| **bg** | The normal (not under the mouse) background color behind the indicator and the label. |
| **bitmap** | Allows the display of a monochrome image on a radiobutton, if this option is set to a bitmap file. |
| **borderwidth** | Size of the border around the indicator. Default is 2 pixels. |
| **command** | The function called when the radiobutton is selected. |
| **cursor** | Changes the appearance of the mouse cursor when over the radiobutton (e.g. arrow, dot). |
| **font** | The font used for the text. |
| **fg** | The color used to render the text. |
| **height** | The height (in text rows, not pixels) of text around the radiobutton. Default is 1. |
| **highlightbackground** | The color of the focus highlight when the radiobutton is NOT in focus. |
| **highlightcolor** | The color of the focus highlight when the radiobutton IS in focus. |
| **image** | If set to an image file, displays an image, instead of text, for the radiobutton. |

| Option | Description |
|---|---|
| `justify` | Justifies text, when multiline, around the radiobutton. Option values are CENTER (default), LEFT, or RIGHT. |
| `padx` | Horizontal padding around the radiobutton. Default is 1 pixel. |
| `pady` | Vertical padding around the radiobutton. Default is 1 pixel. |
| `relief` | Appearance of the decorative border around the label. Default is FLAT. |
| `selectcolor` | The color of the radiobutton when set. Default is red. |
| `selectimage` | If the radiobutton is used to show an image option, not text (by using the image option), this option specifies the image that the first one will change into, after the radiobutton was pressed. |
| `state` | One of three values: NORMAL (out of focus), DISABLED (grayed out), and ACTIVE (when in focus). |
| `text` | The text label displayed as a user choice next to the radiobutton. Multiple lines of text can be "\n"-separated. |
| `textvariable` | The variable that will hold the string value of the radiobutton label and pass it to a control variable of class **`StringVar`** for further usage. |
| `underline` | Will display an underline between specified character locations of the label text. The default is -1, meaning no underline. |
| `value` | The value of the radiobutton variable that effectively allows the user selection to be passed on to the controlling function and outside the Tkinter script to a backend data processing script. |

| Option | Description |
|---|---|
| **`variable`** | The variable whose value is passed to the controlling function. It can be either an **`IntVar`** or a **`StringVar`**. |
| **`width`** | Width of the label in characters (not pixels). The default is the size that fits the contents. |
| **`wraplength`** | The max # of characters on a line, if text label is multiline. |

A radiobutton also has the following methods:

| Method | Description |
|---|---|
| **`deselect()`** | Clears (checks off) the radiobutton. |
| **`flash()`** | Flashes the radiobutton a few times between its active and normal colors, leaving it the way it started. |
| **`invoke()`** | This method can be called to invoke the same actions as if the radiobutton was selected. |
| **`select()`** | Sets (checks on) the radiobutton. |

## The Checkbox Widget

This widget is similar in functionality to, but differs in aspect from, the radiobutton. In the following example we render the same functionality in selecting one of three musicians, using checkboxes instead:

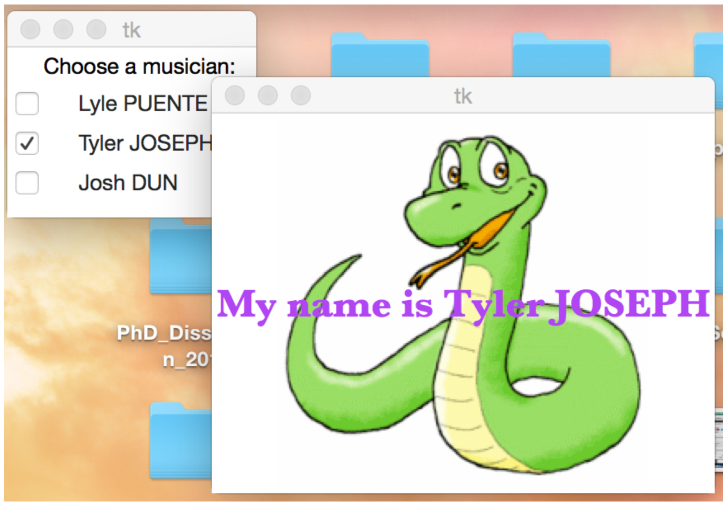
```
from Tkinter import *

root=Tk()
var1=IntVar()
var2=IntVar()
var3=IntVar()

def create_window():
        window=Toplevel(root)
        logo=PhotoImage(file="/Users/awise/Python/Tkinter/
images/python.gif")
```

```
        whichsinger={0: "Lyle PUENTE", 1: "Tyler JOSEPH", 2: 
"Josh DUN"}
        if var1.get()==1:
                explanation="My name is %s" % whichsinger[0]
        elif var2.get()==1:
                explanation="My name is %s" % whichsinger[1]
        elif var3.get()==1:
                explanation="My name is %s" % whichsinger[2]
        widget=Label(window,
                compound = CENTER,
                text=explanation,
                font='Baskerville 24 bold',
                fg='purple',
                wraplength=300,
                image=logo).pack(side="right")
        window.logo=logo

Label(root,
      text="""Choose a musician:""",
      justify=LEFT,
      padx=20).grid(row=0, sticky=W)
checkbox1=Checkbutton(root,
            text="Lyle PUENTE",
            padx=20,
            variable=var1,
            command=create_window).grid(row=1, sticky=W)
checkbox2=Checkbutton(root,
            text="Tyler JOSEPH",
            padx=20,
            variable=var2,
            command=create_window).grid(row=2, sticky=W)
checkbox3=Checkbutton(root,
            text="Josh DUN",
            padx=20,
            variable=var3,
            command=create_window).grid(row=3, sticky=W)

mainloop()
```

The main difference between the Checkbutton and Radiobutton widget is that, while the Radiobutton carries a `value` for the `variable` option, allowing string values to be passed to the controlling (the `command`) function, the Checkbutton DOES NOT. This means that there are only two possible values passed to the command function, `1` if that option was checked, and `0` if it wasn't. It is up to the function, then, to take appropriate action in either case. This makes the

Checkbutton unsuitable for long lists of options, because of how long the code to handle the values associated with each possible option has to be in the command function.

The output of this code is:



My name is Tyler JOSEPH

The checkbutton syntax is:

```
widget=Checkbutton(master, option, …)
```

where:
`master`—the parent window
`option`—one of multiple possible options (not included here, very similar to the Radiobutton)

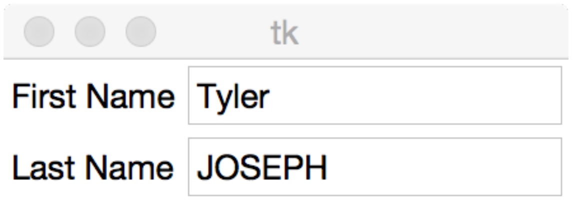
**Text Box ("Entry") Widgets**

The Entry widget allows the user to enter a single line of text. If the string is longer than the box width in number of characters, the contents will be scrolled, which means that the string contents will not be visible in its entirety at any given time.

The following example uses the Entry widget to provide the musician name for our Musicians application. In its first implementation, the script will provide just the entry window with the appropriate text boxes. In the second implementation, we will use the data entered in the text box to provide contents to a variable whose value will be displayed in the new window.

```
from Tkinter import *

master=Tk()

Label(master, text="First Name").grid(row=0)
Label(master, text="Last Name").grid(row=1)

e1=Entry(master)
e2=Entry(master)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)

mainloop( )
```

The output is:



First Name Tyler

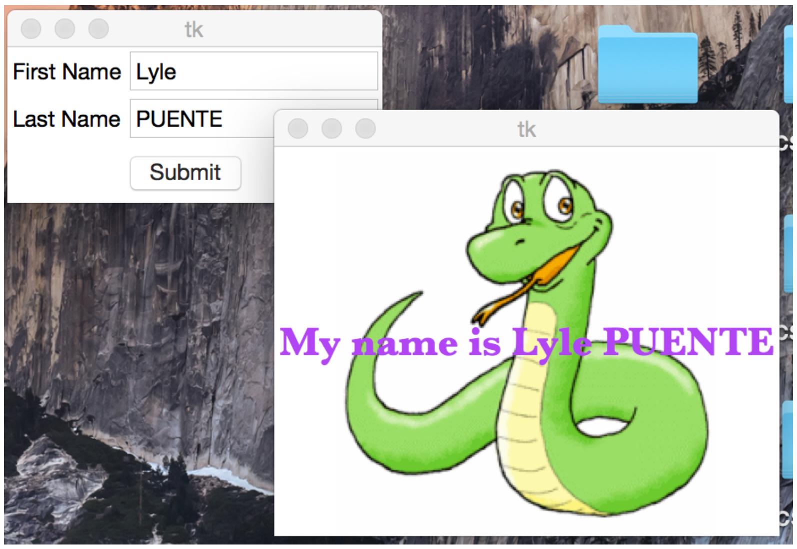
Last Name JOSEPH

Now we want to add the capability to submit this form and capture the input for display in another window:

```
from Tkinter import *

root=Tk()

def create_window():
        window=Toplevel(root)
        logo=PhotoImage(file="/Users/awise/Python/Tkinter/
images/python.gif")
        firstName=entry1.get()
        lastName=entry2.get()
        explanation="My name is %s %s" % (firstName, lastName)
        widget=Label(window,
                compound = CENTER,
                text=explanation,
                font='Baskerville 24 bold',
                fg='purple',
                wraplength=300,
                image=logo).pack(side="right")
        window.logo=logo
```

```
Label(root, text="First Name").grid(row=0)
Label(root, text="Last Name").grid(row=1)

entry1=Entry(root)
entry2=Entry(root)

entry1.grid(row=0, column=1)
entry2.grid(row=1, column=1)

Button(root,
        text='Submit',
        command=create_window).grid(row=3, column=1, sticky=W,
pady=4)

mainloop( )
```

The output is:



The syntax of the Entry widget is:

```
widget=Entry(master, option, ...)
```

where:
`master`—the parent window
`option`—an option from a list of options, not presented here.

## The Canvas Widget

The Canvas widget allows graphics to be displayed inside of a Tkinter window. These graphics can be:

- pre-defined shapes, which can be specified by coordinates (line, circle, oval, rectangle etc.)
- free-hand drawings
- graphs and plots

The following example code (borrowed from python-course.eu) allows free-hand drawing:

```
from Tkinter import *

canvas_width=500
canvas_height=150

def paint(event):
   python_green="#476042"
   x1, y1=(event.x-1), (event.y-1)
   x2, y2=(event.x+1), (event.y+1)
   widget.create_oval(x1, y1, x2, y2, fill=python_green)

root=Tk()
root.title("Painting using Ovals")
widget=Canvas(root,
           width=canvas_width,
           height=canvas_height)
widget.pack(expand=YES, fill=BOTH)
widget.bind("<B1-Motion>", paint)

message=Label(root, text="Press and Drag the mouse to draw")
message.pack(side=BOTTOM)

mainloop()
```

The key to this code is the `widget.bind()` statement, which is an advance introduction into the topic of Events and Binds. In short, a keyboard or mouse event can be associated with a function (in this case, `“<B1-Motion>”`, representing motion with the left mouse button pressed down, is associated with the function `paint()`, which allows the user to create tiny ovals (the dots representing the image), one at a time.

Here's what my daughter, Rosanna, painted with it:

The syntax for declaring a Canvas widget object is:

```
widget=Canvas(master, option=value, ...)
```

where:
`master`—the parent window
`option`—again, one of the many in the following table…

| Option | Description |
|---|---|
| **bd** | Border width in pixels. Default is 2. |
| **bg** | Background color when passive. |
| **confine** | If TRUE, the canvas cannot be scrolled outside of the scroll region. This is the default. |
| **cursor** | One of: arrow, circle, dot etc. |
| **height** | The y size of the canvas. |
| **highlightcolor** | Background color when active (when in focus). |
| **relief** | Type of border: SUNKEN, RAISED, GROOVE, RIDGE. |

| Option | Description |
|---|---|
| **`scrollregion`** | A tuple `(w, n, e, s)` defining how large an area of the canvas can be scrolled, with the 4 elements of the tuple being west, north, east, south. |
| **`width`** | The x size of the canvas. |
| **`xscrollincrement`** | Scroll increment size, if this option is set to a positive value. A scroll unit will be added, for instance, when the user clicks on the bottom arrow of the scroll bar. |
| **`xscrollcommand`** | If canvas is scrollable, this option should be the `set()` method of the horizontal scrollbar. |
| **`yscrollincrement`** | Same as `xscrollincrement`, but in the y direction. |
| **`yscrollcommand`** | Same as `xscrollcommand`, but in the y direction. |

## The Text Widget

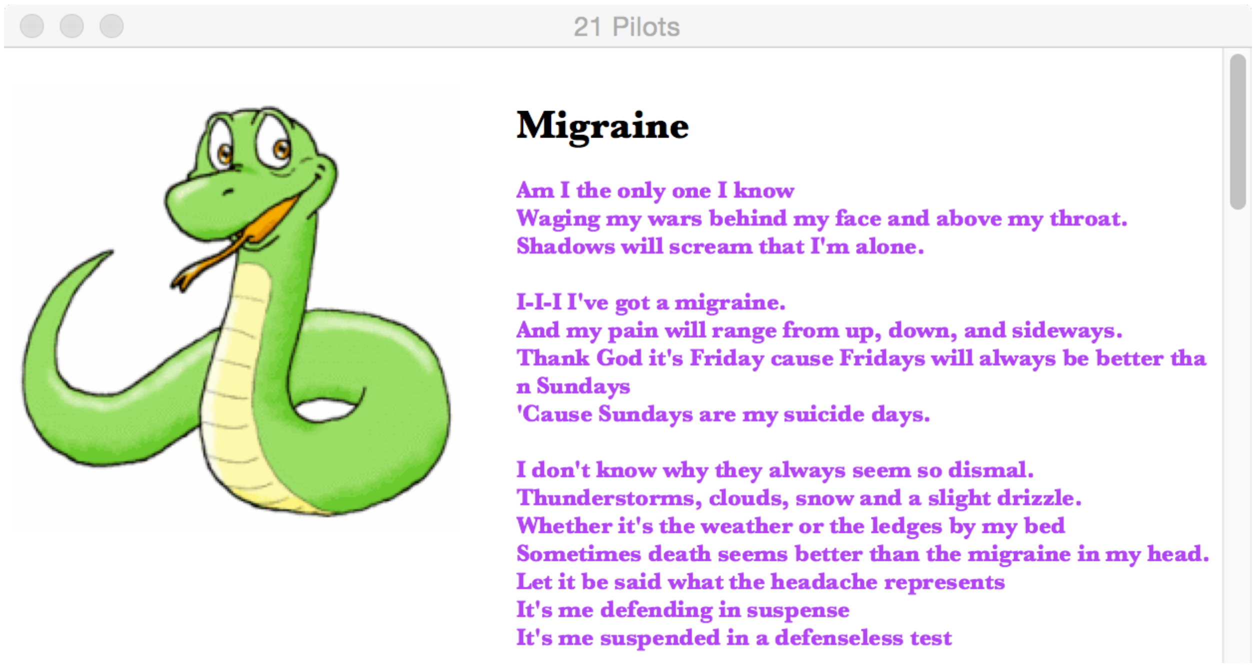
This widget is used to display multiple lines of text, which can also be made scrollable. Text widgets can also be used in forms, as text editors, or even web browsers. The can also display links, images, and CSS/HTML.

The following is a simple example of a Text widget showing song lyrics for one of our 21 Pilots song, long enough to need scroll bars:

```
from Tkinter import *

root=Tk()
root.title("21 Pilots")

text1=Text(root, height=20, width=35)
photo=PhotoImage(file='/Users/awise/Python/Tkinter/images/
python.gif')
text1.insert(END,'\n')
text1.image_create(END, image=photo)

text1.pack(side=LEFT)

text2=Text(root, height=20, width=50)
scroll=Scrollbar(root, command=text2.yview)
text2.configure(yscrollcommand=scroll.set)
```

```
text2.tag_configure('bold_italics', font=('Baskerville', 12,
'bold', 'italic'))
text2.tag_configure('big', font=('Baskerville', 20, 'bold'))
#476042
text2.tag_configure('color', foreground='purple',
                        font=('Baskerville', 12, 'bold'))
text2.tag_bind('follow', '<1>', lambda e, t=text2: t.insert(END,
"Not now, maybe later!"))
text2.insert(END,'\nMigraine\n', 'big')
quote = """
Am I the only one I know
Waging my wars behind my face and above my throat.
Shadows will scream that I'm alone.

I-I-I I've got a migraine.
And my pain will range from up, down, and sideways.
Thank God it's Friday cause Fridays will always be better than
Sundays
'Cause Sundays are my suicide days.

I don't know why they always seem so dismal.
Thunderstorms, clouds, snow and a slight drizzle.
Whether it's the weather or the ledges by my bed
Sometimes death seems better than the migraine in my head.
Let it be said what the headache represents
It's me defending in suspense
It's me suspended in a defenseless test
Being tested by a ruthless examiner
That's represented best by my depressing thoughts.

Am I the only one I know,
Waging my wars behind my face and above my throat.
Shadows will scream that I'm alone.
But I know, we've made it this far, kid.

Made it this far
Made it this fa
"""
text2.insert(END, quote, 'color')
text2.insert(END, 'follow-up\n', 'follow')
text2.pack(side=LEFT)
scroll.pack(side=RIGHT, fill=Y)

root.mainloop()
```

This outputs the following Tkinter window:



Migraine

Am I the only one I know
Waging my wars behind my face and above my throat.
Shadows will scream that I'm alone.

I-I-I I've got a migraine.
And my pain will range from up, down, and sideways.
Thank God it's Friday cause Fridays will always be better tha
n Sundays
'Cause Sundays are my suicide days.

I don't know why they always seem so dismal.
Thunderstorms, clouds, snow and a slight drizzle.
Whether it's the weather or the ledges by my bed
Sometimes death seems better than the migraine in my head.
Let it be said what the headache represents
It's me defending in suspense
It's me suspended in a defenseless test

The syntax of the Text widget is:

```
widget=Text(master, option, ...)
```

The options are listed in the following table:

| Option | Description |
|---|---|
| **bg** | The background color when passive. |
| **bd** | Width of the border around the text widget. Default is 2 pixels. |
| **cursor** | Shape of the cursor when the mouse is over the Text widget. |
| **exportselection** | If text is selected within the Text widget, if this option is set to 1, it will go on the clipboard of the window manager. To changes this behavior, set this option to 1. |
| **font** | The default font text in the Text widget. |

| Option | Description |
|---|---|
| **fg** | Foreground (text) color. |
| **height** | Height of the widget in # of text lines. |
| **highlightbackground** | Color of the background when widget is in focus. |
| **highlightcolor** | Color of the foreground (text) when widget is in focus. |
| **highlightthickness** | Thickness of the highlight focus. Default is 1. |
| **insertbackground** | Color of the insertion cursor. Default is BLACK. |
| **insertborderwidth** | Size of the 3D border around the insertion cursor. Default is 0. |
| **insertofftime** | The OFF time [ms] during the blink cycle of the cursor. Default is 300ms. |
| **insertontime** | The ON time [ms] during the blink cycle of the cursor. Default is 600ms. |
| **insertwidth** | Width of the insertion cursor (height determined by the tallest item in its line). Default is 2 pixels. |
| **padx** | Size of internal padding to left and right of the text area. Default is 1 pixel. |
| **pady** | Size of internal padding above and below the text area. Default is 1 pixel. |
| **relief** | The 3D appearance of the text widget. Default is SUNKEN. |
| **selectbackground** | Background color of selected text. |
| **selectborderwidth** | Width of the border around selected text. |
| **spacing1** | Amount of vertical space above each line of text. If a line wraps, only affects the first line in the multiline text. Default is 0. |
| **spacing2** | Amount of vertical space above each line of text for all subsequent lines. Default is 0. |

| Option | Description |
|---|---|
| **`spacing3`** | Amount of vertical space below each line of text. It applies only below last line of multiline wrapping text. Default is 0. |
| **`state`** | When set to NORMAL, Text widgets respond to keyboard and mouse events. To make the widget unresponsive, set this option to DISABLED. |
| **`tabs`** | Controls how tabs indent text. |
| **`width`** | Width of the widget in characters. |
| **`wrap`** | Controls the text wrapping behavior. If set to WORD, text line will wrap after the first complete word that fits. If set to CHAR, text line will wrap at any character when the line hits its max width. |
| **`xscrollcommand`** | To make text scrollable in the x dimension, this option should be set to the **`set()`** method of the horizontal scrollbar. |
| **`yscrollcommand`** | To make text scrollable in the y dimension, this option should be set to the **`set()`** method of the vertical scrollbar. |

There are methods to configure:

- **text objects**—used to process ranges of text from within the Text widget, based on indices of characters within the line of text
- **marks**—used to bookmark positions between two characters within text from a Text widget
- **tags**—used to associate names to regions of text, which makes easy to distinguish between text areas when it comes to different formatting or to calling different methods for them

| Text Object Methods | Description |
|---|---|
| **`delete(startindex[, endindex])`** | Deletes a range of text. |
| **`get(startindex[, endindex])`** | Returns a range of text. |

| Text Object Methods | Description |
|---|---|
| **`index(index)`** | Returns the absolute value of an index, based on the **`index`** passed in. |
| **`insert(index[, string], …)`** | Inserts string starting at **`index`** location. |
| **`see(index)`** | Returns TRUE if text located at **`index`** is visible. |

A **mark** indicates <u>where the cursor is placed on existing text to begin insertion</u>. **Gravity** is a property through which you can specify <u>where the mark will remain after insertion</u> (the opposite of which side of the mark the insertion will take place).

For example, if we want to insert an `“o”` in `“Tyler J|seph”`, and the mark is indicated by the vertical bar, if the gravity is set to RIGHT (the default), the result of the insertion is `“Tyler Jo|seph”`, i.e. *the mark remains to the right of the inserted letter*.

| Mark Object Methods | Description |
|---|---|
| **`index(mark)`** | Returns line and column of a mark. A mark indicates where the cursor is placed on existing text to begin insertion. |
| **`mark_gravity(mark[, gravity])`** | Returns the gravity of **`mark`** passed in. If the 2nd argument is provided, the gravity is set for the given mark. |
| **`mark_names()`** | Returns all marks from the Text widget. |
| **`mark_unset(mark)`** | Removes **`mark`** passed in from the given Text widget. |

**Tags** are used to associated names to regions of text, to make the task of displaying settings for subsets (text areas) of text from a Text widget easier.

| Tag Object Methods | Description |
|---|---|
| **`tag_add(tagname, startindex[, endindex])`** | Assigns a tag name to the range specified either by a start index to the end of the text, or by a pair start index, end index. |
| **`tag_config`** | Configures tag properties, such as justify (options being CENTER, LEFT, or RIGHT), tabs, and underline. The properties apply to the tagged text. |
| **`tag_delete(tagname)`** | Deletes the **`tagname`** passed in. |
| **`tag_remove(tagname[, startindex[, endindex]])`** | Tag passed in as **`tagname`** is removed from the provided area without deleting its definition. |