Part II

Graphics

Chapter 15

GUI Programming with Tkinter

Up until now, the only way our programs have been able to interact with the user is through keyboard input via the `input` statement. But most real programs use windows, buttons, scrollbars, and various other things. These *widgets* are part of what is called a *Graphical User Interface* or GUI. This chapter is about GUI programming in Python with Tkinter.

All of the widgets we will be looking at have far more options than we could possibly cover here. An excellent reference is Fredrik Lundh's *Introduction to Tkinter* [2].

15.1 Basics

Nearly every GUI program we will write will contain the following three lines:

```
from tkinter import *
root = Tk()
mainloop()
```

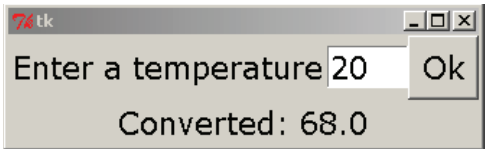
The first line imports all of the GUI stuff from the `tkinter` module. The second line creates a window on the screen, which we call `root`. The third line puts the program into what is essentially a long-running while loop called the *event loop*. This loop runs, waiting for keypresses, button clicks, etc., and it exits when the user closes the window.

Here is a working GUI program that converts temperatures from Fahrenheit to Celsius.

```
from tkinter import *

def calculate():
    temp = int(entry.get())
    temp = 9/5*temp+32
    output_label.configure(text = 'Converted: {:.1f}'.format(temp))
    entry.delete(0,END)
```

```
root = Tk()
message_label = Label(text='Enter a temperature',
                      font=('Verdana', 16))
output_label = Label(font=('Verdana', 16))
entry = Entry(font=('Verdana', 16), width=4)
calc_button = Button(text='Ok', font=('Verdana', 16),
                     command=calculate)

message_label.grid(row=0, column=0)
entry.grid(row=0, column=1)
calc_button.grid(row=0, column=2)
output_label.grid(row=1, column=0, columnspan=3)

mainloop()
```

Here is what the program looks like:



We now will examine the components of the program separately.

15.2 Labels

A label is a place for your program to place some text on the screen. The following code creates a label and places it on the screen.

```
hello_label = Label(text='hello')
hello_label.grid(row=0, column=0)
```

We call `Label` to create a new label. The capital `L` is required. Our label's name is `hello_label`. Once created, use the `grid` method to place the label on the screen. We will explain `grid` in the next section.

Options There are a number of options you can change including font size and color. Here are some examples:

```
hello_label = Label(text='hello', font=('Verdana', 24, 'bold'),
                    bg='blue', fg='white')
```

Note the use of keyword arguments. Here are a few common options:

- `font` — The basic structure is `font=` (*font name, font size, style*). You can leave out the font size or the style. The choices for style are `'bold'`, `'italic'`, `'underline'`, `'overstrike'`, `'roman'`, and `'normal'` (which is the default). You can combine multiple styles like this: `'bold italic'`.

- `fg` and `bg` — These stand for `foreground` and `background`. Many common color names can be used, like `'blue'`, `'green'`, etc. Section 16.2 describes how to get essentially any color.
- `width` — This is how many characters long the label should be. If you leave this out, Tkinter will base the width off of the text you put in the label. This can make for unpredictable results, so it is good to decide ahead of time how long you want your label to be and set the `width` accordingly.
- `height` — This is how many rows high the label should be. You can use this for multi-line labels. Use newline characters in the text to get it to span multiple lines. For example, `text='hi\nthere'`.

There are dozens more options. The aforementioned *Introduction to Tkinter* [2] has a nice list of the others and what they do.

Changing label properties Later in your program, after you've created a label, you may want to change something about it. To do that, use its `configure` method. Here are two examples that change the properties of a label called `label`:

```
label.configure(text='Bye')
label.configure(bg='white', fg='black')
```

Setting `text` to something using the `configure` method is kind of like the GUI equivalent of a **`print`** statement. However, in calls to `configure` we cannot use commas to separate multiple things to print. We instead need to use string formatting. Here is a **`print`** statement and its equivalent using the `configure` method.

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

The `configure` method works with most of the other widgets we will see.

15.3 `grid`

The `grid` method is used to place things on the screen. It lays out the screen as a rectangular grid of rows and columns. The first few rows and columns are shown below.

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

Spanning multiple rows or columns There are optional arguments, `rowspan` and `columnspan`, that allow a widget to take up more than one row or column. Here is an example of several grid statements followed by what the layout will look like:

```
label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
label3.grid(row=1, column=0, columnspan=2)
label4.grid(row=1, column=2)
label5.grid(row=2, column=2)
```

label1	label2	
label 3		label4
		label5

Spacing To add extra space between widgets, there are optional arguments `padx` and `pady`.

Important note Any time you create a widget, to place it on the screen you need to use `grid` (or one of its cousins, like `pack`, which we will talk about later). Otherwise it will not be visible.

15.4 Entry boxes

Entry boxes are a way for your GUI to get text input. The following example creates a simple entry box and places it on the screen.

```
entry = Entry()
entry.grid(row=0, column=0)
```

Most of the same options that work with labels work with entry boxes (and most of the other widgets we will talk about). The `width` option is particularly helpful because the entry box will often be wider than you need.

- **Getting text** To get the text from an entry box, use its `get` method. This will return a string. If you need numerical data, use **`eval`** (or **`int`** or **`float`**) on the string. Here is a simple example that gets text from an entry box named `entry`.

  ```
  string_value = entry.get()
  num_value = eval(entry.get())
  ```

- **Deleting text** To clear an entry box, use the following:

  ```
  entry.delete(0,END)
  ```

- **Inserting text** To insert text into an entry box, use the following:

  ```
  entry.insert(0, 'hello')
  ```

15.5 Buttons

The following example creates a simple button:

```
ok_button = Button(text='Ok')
```

To get the button to do something when clicked, use the `command` argument. It is set to the name of a function, called a *callback function*. When the button is clicked, the callback function is called. Here is an example:

```
from tkinter import *

def callback():
    label.configure(text='Button clicked')

root = Tk()
label = Label(text='Not clicked')
button = Button(text='Click me', command=callback)

label.grid(row=0, column=0)
button.grid(row=1, column=0)

mainloop()
```

When the program starts, the label says `Click me`. When the button is clicked, the callback function `callback` is called, which changes the label to say `Button clicked`.

`lambda` trick Sometimes we will want to pass information to the callback function, like if we have several buttons that use the same callback function and we want to give the function information about which button is being clicked. Here is an example where we create 26 buttons, one for each letter of the alphabet. Rather than use 26 separate `Button()` statements and 26 different functions, we use a list and one function.

```
from tkinter import *
alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def callback(x):
    label.configure(text='Button {} clicked'.format(alphabet[x]))

root = Tk()

label = Label()
label.grid(row=1, column=0, columnspan=26)

buttons = [0]*26 # create a list to hold 26 buttons
for i in range(26):
    buttons[i] = Button(text=alphabet[i],
                        command = lambda x=i: callback(x))
```

```
    buttons[i].grid(row=0, column=i)

mainloop()
```

We note a few things about this program. First, we set `buttons=[0]*26`. This creates a list with 26 things in it. We don't really care what thoset things are because they will be replaced with buttons. An alternate way to create the list would be to set `buttons=[]` and use the `append` method.

We only use one callback function and it has one argument, which indicates which button was clicked. As far as the **lambda** trick goes, without getting into the details, `command=callback(i)` does not work, and that is why we resort to the **lambda** trick. You can read more about **lambda** in Section 23.2. An alternate approach is to use classes.

15.6 Global variables

Let's say we want to keep track of how many times a button is clicked. An easy way to do this is to use a global variable as shown below.

```
from tkinter import *

def callback():
    global num_clicks
    num_clicks = num_clicks + 1
    label.configure(text='Clicked {} times.'.format(num_clicks))

num_clicks = 0
root = Tk()

label = Label(text='Not clicked')
button = Button(text='Click me', command=callback)

label.grid(row=0, column=0)
button.grid(row=1, column=0)

mainloop()
```

We will be using a few global variables in our GUI programs. Using global variables unnecessarily, especially in long programs, can cause difficult to find errors that make programs hard to maintain,

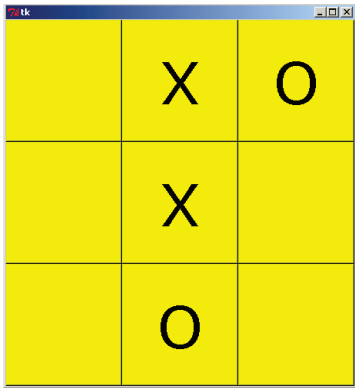
but in the short programs that we will be writing, we should be okay. Object-oriented programming provides an alternative to global variables.

15.7 Tic-tac-toe

Using Tkinter, in only about 20 lines we can make a working tic-tac-toe program:

```
from tkinter import *

def callback(r,c):
    global player
    if player == 'X':
        b[r][c].configure(text = 'X')
        player = 'O'
    else:
        b[r][c].configure(text = 'O')
        player = 'X'

root = Tk()

b = [[0,0,0],
     [0,0,0],
     [0,0,0]]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                         command = lambda r=i,c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)

player = 'X'

mainloop()
```

The program works, though it does have a few problems, like letting you change a cell that already has something in it. We will fix this shortly. First, let's look at how the program does what it does. Starting at the bottom, we have a variable `player` that keeps track of whose turn it is. Above that we create the board, which consists of nine buttons stored in a two-dimensional list. We use the `lambda` trick to pass the row and column of the clicked button to the callback function. In the callback function we write an *X* or an *O* into the button that was clicked and change the value of the global variable `player`.



Correcting the problems To correct the problem about being able to change a cell that already has something in it, we need to have a way of knowing which cells have *X*'s, which have *O*'s, and which are empty. One way is to use a `Button` method to ask the button what its text is. Another way, which we will do here is to create a new two-dimensional list, which we will call `states`, that will keep track of things. Here is the code.

```
from tkinter import *

def callback(r,c):
    global player

    if player == 'X' and states[r][c] == 0:
        b[r][c].configure(text='X')
        states[r][c] = 'X'
        player = 'O'

    if player == 'O' and states[r][c] == 0:
        b[r][c].configure(text='O')
        states[r][c] = 'O'
        player = 'X'


root = Tk()

states = [[0,0,0],
          [0,0,0],
          [0,0,0]]

b = [[0,0,0],
     [0,0,0],
     [0,0,0]]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                         command = lambda r=i,c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)
```

```
player = 'X'

mainloop()
```

We have not added much to the program. Most of the new action happens in the callback function. Every time someone clicks on a cell, we first check to see if it is empty (that the corresponding index in `states` is 0), and if it is, we display an *X* or *O* on the screen and record the new value in `states`. Many games have a variable like `states` that keeps track of what is on the board.

Checking for a winner We have a winner when there are three *X*'s or three *O*'s in a row, either vertically, horizontally, or diagonally. To check if there are three in a row across the top row, we can use the following if statement:

```
if states[0][0]==states[0][1]==states[0][2]!=0:
    stop_game=True
    b[0][0].configure(bg='grey')
    b[0][1].configure(bg='grey')
    b[0][2].configure(bg='grey')
```

This checks to see if each of the cells has the same nonzero entry. We are using the shortcut from Section 10.3 here in the if statement. There are more verbose if statements that would work. If we do find a winner, we highlight the winning cells and then set a global variable `stop_game` equal to **`True`**. This variable will be used in the callback function. Whenever the variable is **`True`** we should not allow any moves to take place.

Next, to check if there are three in a row across the middle row, change the first coordinate from 0 to 1 in all three references, and to check if there are three in a row across the bottom, change the 0's to 2's. Since we will have three very similar if statements that only differ in one location, a for loop can be used to keep the code short:

```
for i in range(3):
    if states[i][0]==states[i][1]==states[i][2]!=0:
        b[i][0].configure(bg='grey')
        b[i][1].configure(bg='grey')
        b[i][2].configure(bg='grey')
        stop_game = True
```

Next, checking for vertical winners is pretty much the same except we vary the second coordinate instead of the first. Finally, we have two further if statements to take care of the diagonals. The full program is at the end of this chapter. We have also added a few color options to the `configure` statements to make the game look a little nicer.

Further improvements From here it would be easy to add a restart button. The callback function for that variable should set `stop_game` back to false, it should set `states` back to all zeroes, and it should configure all the buttons back to `text=''` and `bg='yellow'`.

To add a computer player would also not be too difficult, if you don't mind it being a simple com-

puter player that moves randomly. That would take about 10 lines of code. To make an intelligent computer player is not too difficult. Such a computer player should look for two *O*'s or *X*'s in a row in order to try to win or block, as well avoid getting put into a no-win situation.

```
from tkinter import *

def callback(r,c):
    global player

    if player == 'X' and states[x][y] == 0 and stop_game==False:
        b[r][c].configure(text='X', fg='blue', bg='white')
        states[r][c] = 'X'
        player = 'O'

    if player == 'O' and states[r][c] == 0 and stop_game==False:
        b[r][c].configure(text='O', fg='orange', bg='black')
        states[r][c] = 'O'
        player = 'X'

    check_for_winner()

def check_for_winner():
    global stop_game
    for i in range(3):
        if states[i][0]==states[i][1]==states[i][2]!=0:
            b[i][0].configure(bg='grey')
            b[i][1].configure(bg='grey')
            b[i][2].configure(bg='grey')
            stop_game = True

    for i in range(3):
        if states[0][i]==states[1][i]==states[2][i]!=0:
            b[0][i].configure(bg='grey')
            b[1][i].configure(bg='grey')
            b[2][i].configure(bg='grey')
            stop_game = True

    if states[0][0]==states[1][1]==states[2][2]!=0:
        b[0][0].configure(bg='grey')
        b[1][1].configure(bg='grey')
        b[2][2].configure(bg='grey')
        stop_game = True

    if states[2][0]==states[1][1]==states[0][2]!=0:
        b[2][0].configure(bg='grey')
        b[1][1].configure(bg='grey')
        b[0][2].configure(bg='grey')
        stop_game = True

root = Tk()

b = [[0,0,0],
     [0,0,0],
```

```
     [0,0,0]]


states = [[0,0,0],
          [0,0,0],
          [0,0,0]]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(font=('Verdana', 56), width=3, bg='yellow',
                         command = lambda r=i,c=j: callback(r,c))
        b[i][j].grid(row = i, column = j)

player = 'X'
stop_game = False

mainloop()
```

Chapter 16

GUI Programming II

In this chapter we cover more basic GUI concepts.

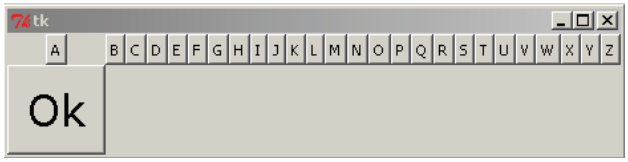
16.1 Frames

Let's say we want 26 small buttons across the top of the screen, and a big Ok button below them, like below:

We try the following code:

```
from tkinter import *

root = Tk()

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
buttons = [0]*26
for i in range(26):
    buttons[i] = Button(text=alphabet[i])
    buttons[i].grid(row=0, column=i)

ok_button = Button(text='Ok', font=('Verdana', 24))
ok_button.grid(row=1, column=0)

mainloop()
```

But we instead get the following unfortunate result:



The problem is with column 0. There are two widgets there, the A button and the Ok button, and Tkinter will make that column big enough to handle the larger widget, the Ok button. One solution to this problem is shown below:

```
ok_button.grid(row=1, column=0, columnspan=26)
```

Another solution to this problem is to use what is called a *frame*. The frame's job is to hold other widgets and essentially combine them into one large widget. In this case, we will create a frame to group all of the letter buttons into one large widget. The code is shown below:

```
from tkinter import *

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
root = Tk()

button_frame = Frame()
buttons = [0]*26
for i in range(26):
    buttons[i] = Button(button_frame, text=alphabet[i])
    buttons[i].grid(row=0, column=i)

ok_button = Button(text='Ok', font=('Verdana', 24))

button_frame.grid(row=0, column=0)
ok_button.grid(row=1, column=0)

mainloop()
```

To create a frame, we use `Frame()` and give it a name. Then, for any widgets we want include in the frame, we include the name of the frame as the first argument in the widget's declaration. We still have to grid the widgets, but now the rows and columns will be relative to the frame. Finally, we have to grid the frame itself.

16.2 Colors

Tkinter defines many common color names, like `'yellow'` and `'red'`. It also provides a way to get access to millions of more colors. We first have to understand how colors are displayed on the screen.

Each color is broken into three components—a red, a green, and a blue component. Each component can have a value from 0 to 255, with 255 being the full amount of that color. Equal parts of red and green create shades of yellow, equal parts of red and blue create shades of purple, and equal

parts of blue and green create shades of turquoise. Equal parts of all three create shades of gray. Black is when all three components have values of 0 and white is when all three components have values of 255. Varying the values of the components can produce up to $256^3 \approx 16$ million colors. There are a number of resources on the web that allow you to vary the amounts of the components and see what color is produced.

To use colors in Tkinter is easy, but with one catch—component values are given in hexadecimal. Hexadecimal is a base 16 number system, where the letters A-F are used to represent the digits 10 through 15. It was widely used in the early days of computing, and it is still used here and there. Here is a table comparing the two number bases:

0	0	8	8	16	10	80	50
1	1	9	9	17	11	100	64
2	2	10	A	18	12	128	80
3	3	11	B	31	1F	160	A0
4	4	12	C	32	20	200	C8
5	5	13	D	33	21	254	FE
6	6	14	E	48	30	255	FF
7	7	15	F	64	40	256	100

Because the color component values run from 0 to 255, they will run from 0 to FF in hexadecimal, and thus are described by two hex digits. A typical color in Tkinter is specified like this: `'#A202FF'`. The color name is prefaced with a pound sign. Then the first two digits are the red component (in this case `A2`, which is 162 in decimal). The next two digits specify the green component (here `02`, which is 2 in decimal), and the last two digits specify the blue component (here `FF`, which is 255 in decimal). This color turns out to be a bluish violet. Here is an example of it in use:

```
label = Label(text='Hi', bg='#A202FF')
```

If you would rather not bother with hexadecimal, you can use the following function which will convert percentages into the hex string that Tkinter uses.

```
def color_convert(r, g, b):
    return '#{:02x}{:02x}{:02x}'.format(int(r*2.55),int(g*2.55),
                                        int(b*2.55))
```

Here is an example of it to create a background color that has 100% of the red component, 85% of green and 80% of blue.

```
label = Label(text='Hi', bg=color_convert(100, 85, 80))
```

16.3 Images

Labels and buttons (and other widgets) can display images instead of text.

To use an image requires a little set-up work. We first have to create a `PhotoImage` object and give it a name. Here is an example:

```
cheetah_image = PhotoImage(file='cheetahs.gif')
```

Here are some examples of putting the image into widgets:

```
label = Label(image=cheetah_image)
button = Button(image=cheetah_image, command=cheetah_callback())
```

You can use the `configure` method to set or change an image:

```
label.configure(image=cheetah_image)
```

File types One unfortunate limitation of Tkinter is the only common image file type it can use is GIF. If you would like to use other types of files, one solution is to use the Python Imaging Library, which will be covered in Section 18.2.

16.4 Canvases

A canvas is a widget on which you can draw things like lines, circles, rectangles. You can also draw text, images, and other widgets on it. It is a very versatile widget, though we will only describe the basics here.

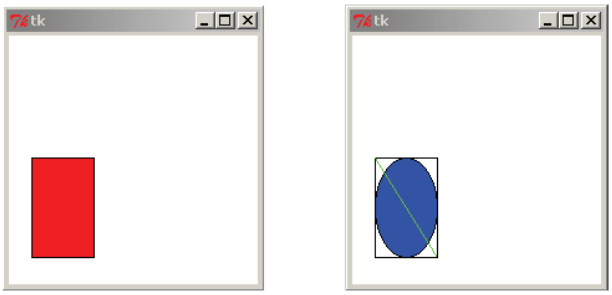
Creating canvases The following line creates a canvas with a white background that is 200 × 200 pixels in size:

```
canvas = Canvas(width=200, height=200, bg='white')
```

Rectangles The following code draws a red rectangle to the canvas:

```
canvas.create_rectangle(20,100,30,150, fill='red')
```

See the image below on the left. The first four arguments specify the coordinates of where to place the rectangle on the canvas. The upper left corner of the canvas is the origin, $(0, 0)$. The upper left of the rectangle is at $(20, 100)$, and the lower right is at $(30, 150)$. If were to leave off `fill='red'`, the result would be a rectangle with a black outline.



Ovals and lines Drawing ovals and lines is similar. The image above on the right is created with the following code:

```
canvas.create_rectangle(20,100,70,180)
canvas.create_oval(20,100,70,180, fill='blue')
canvas.create_line(20,100,70,180, fill='green')
```

The rectangle is here to show that lines and ovals work similarly to rectangles. The first two coordinates are the upper left and the second two are the lower right.

To get a circle with radius `r` and center `(x,y)`, we can create the following function:

```
def create_circle(x,y,r):
    canvas.create_oval(x-r,y-r,x+r,y+r)
```

Images We can add images to a canvas. Here is an example:

```
cheetah_image = PhotoImage(file='cheetahs.gif')
canvas.create_image(50,50, image=cheetah_image)
```

The two coordinates are where the center of the image should be.

Naming things, changing them, moving them, and deleting them We can give names to the things we put on the canvas. We can then use the name to refer to the object in case we want to move it or remove it from the canvas. Here is an example were we create a rectangle, change its color, move it, and then delete it:

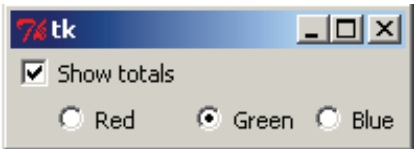
```
rect = canvas.create_rectangle(0,0,20,20)
canvas.itemconfigure(rect, fill='red')
canvas.coords(rect,40,40,60,60)
canvas.delete(rect)
```

The `coords` method is used to move or resize an object and the `delete` method is used to delete it. If you want to delete everything from the canvas, use the following:

```
canvas.delete(ALL)
```

16.5 Check buttons and Radio buttons

In the image below, the top line shows a check button and the bottom line shows a radio button.



Check buttons The code for the above check button is:

```
show_totals = IntVar()
check = Checkbutton(text='Show totals', var=show_totals)
```

The one thing to note here is that we have to tie the check button to a variable, and it can't be just any variable, it has to be a special kind of Tkinter variable, called an `IntVar`. This variable, `show_totals`, will be 0 when the check button is unchecked and 1 when it is checked. To access the value of the variable, you need to use its `get` method, like this:

```
show_totals.get()
```

You can also set the value of the variable using its `set` method. This will automatically check or uncheck the check button on the screen. For instance, if you want the above check button checked at the start of the program, do the following:

```
show_totals = IntVar()
show_totals.set(1)
check = Checkbutton(text='Show totals', var=show_totals)
```

Radio buttons Radio buttons work similarly. The code for the radio buttons shown at the start of the section is:

```
color = IntVar()
redbutton = Radiobutton(text='Red', var=color, value=1)
greenbutton = Radiobutton(text='Green', var=color, value=2)
bluebutton = Radiobutton(text='Blue', var=color, value=3)
```

The value of the `IntVar` object `color` will be 1, 2, or 3, depending on whether the left, middle, or right button is selected. These values are controlled by the `value` option, specified when we create the radio buttons.

Commands Both check buttons and radio buttons have a `command` option, where you can set a callback function to run whenever the button is selected or unselected.

16.6 `Text` widget

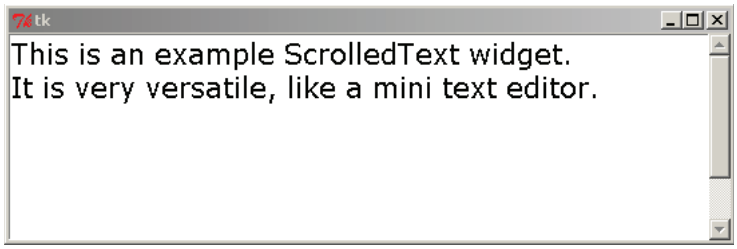
The `Text` widget is a bigger, more powerful version of the `Entry` widget. Here is an example of creating one:

```
textbox = Text(font=('Verdana', 16), height=6, width=40)
```

The widget will be 40 characters wide and 6 rows tall. You can still type past the sixth row; the widget will just display only six rows at a time, and you can use the arrow keys to scroll.

If you want a scrollbar associated with the text box you can use the `ScrolledText` widget. Other than the scrollbar, `ScrolledText` works more or less the same as `Text`. An example is of what it looks like is shown below. To use the `ScrolledText` widget, you will need the following import:

```
from tkinter.scrolledtext import ScrolledText
```



Here are a few common commands:

Statement	Description
`textbox.get(1.0,END)`	returns the contents of the text box
`textbox.delete(1.0,END)`	deletes everything in the text box
`textbox.insert(END,'Hello')`	inserts text at the end of the text box

One nice option when declaring the `Text` widget is `undo=True`, which allows Ctrl+Z and Ctrl+Y to undo and redo edits. There are a ton of other things you can do with the `Text` widget. It is almost like a miniature word processor.

16.7 `Scale` widget

A `Scale` is a widget that you can slide back and forth to select different values. An example is shown below, followed by the code that creates it.


```
scale = Scale(from_=1, to_=100, length=300, orient='horizontal')
```

Here are some of the useful options of the `Scale` widget:

Option	Description
`from_`	minimum value possible by dragging the scale
`to_`	maximum value possible by dragging the scale
`length`	how many pixels long the scale is
`label`	specify a label for the scale
`showvalue='NO'`	gets rid of the number that displays above the scale
`tickinterval=1`	displays tickmarks at every unit (1 can be changed)

There are several ways for your program to interact with the scale. One way is to link it with an `IntVar` just like with check buttons and radio buttons, using the `variable` option. Another option is to use the scale's `get` and `set` methods. A third way is to use the `command` option, which

works just like with buttons.

16.8 GUI Events

Often we will want our programs to do something if the user presses a certain key, drags something on a canvas, uses the mouse wheel, etc. These things are called *events*.

A simple example The first GUI program we looked at back in Section 15.1 was a simple temperature converter. Anytime we wanted to convert a temperature we would type in the temperature in the entry box and click the Calculate button. It would be nice if the user could just press the enter key after they type the temperature instead of having to click to Calculate button. We can accomplish this by adding one line to the program:

```
entry.bind('<Return>', lambda dummy=0:calculate())
```

This line should go right after you declare the entry box. What it does is it takes the event that the enter (return) key is pressed and *binds* it to the `calculate` function.

Well, sort of. The function you bind the event to is supposed to be able to receive a copy of an `Event` object, but the calculate function that we had previously written takes no arguments. Rather than rewrite the function, the line above uses `lambda` trick to essentially throw away the `Event` object.

Common events Here is a list of some common events:

Event	Description
`<Button-1>`	The left mouse button is clicked.
`<Double-Button-1>`	The left mouse button is double-clicked.
`<Button-Release-1>`	The left mouse button is released.
`<B1-Motion>`	A click-and-drag with the left mouse button.
`<MouseWheel>`	The mouse wheel is moved.
`<Motion>`	The mouse is moved.
`<Enter>`	The mouse is now over the widget.
`<Leave>`	The mouse has now left the widget.
`<Key>`	A key is pressed.
<*key name*>	The *key name* key is pressed.

For all of the mouse button examples, the number 1 can be replaced with other numbers. Button 2 is the middle button and button 3 is the right button.

The most useful attributes in the `Event` object are:

Attribute	Description
`keysym`	The name of the key that was pressed
`x, y`	The coordinates of the mouse pointer
`delta`	The value of the mouse wheel

Key events For key events, you can either have specific callbacks for different keys or catch all keypresses and deal with them in the same callback. Here is an example of the latter:

```
from tkinter import *

def callback(event):
    print(event.keysym)

root = Tk()
root.bind('<Key>', callback)

mainloop()
```

The above program prints out the names of the keys that were pressed. You can use those names in if statements to handle several different keypresses in the callback function, like below:

```
if event.keysym = 'percent':
    # percent (shift+5) was pressed, do something about it...
elif event.keysym = 'a':
    # lowercase a was pressed, do something about it...
```

Use the single callback method if you are catching a lot of keypresses and are doing something similar with all of them. On the other hand, if you just want to catch a couple of specific keypresses or if certain keys have very long and specific callbacks, you can catch keypresses separately like below:

```
from tkinter import *

def callback1(event):
    print('You pressed the enter key.')

def callback1(event):
    print('You pressed the up arrow.')

root = Tk()
root.bind('<Return>', callback1)
root.bind('<Up>', callback2)
mainloop()
```

The key names are the same as the names stored in the keysym attribute. You can use the program from earlier in this section to find the names of all the keys. Here are the names for a few common keys:

Tkinter name	Common name
`<Return>`	Enter key
`<Tab>`	Tab key
`<Space>`	Spacebar
`<F1>`, ..., `<F12>`	F1, ..., F12
`<Next>`, `<Prior>`	Page up, Page down
`<Up>`, `<Down>`, `<Left>`, `<Right>`	Arrow keys
`<Home>`, `<End>`	Home, End
`<Insert>`, `<Delete>`	Insert, Delete
`<Caps_Lock>`, `<Num_Lock>`	Caps lock, Number lock
`<Control_L>`, `<Control_R>`	Left and right Control keys
`<Alt_L>`, `<Alt_R>`	Left and right Alt keys
`<Shift_L>`, `<Shift_R>`	Left and right Shift keys

Most printable keys can be captured with their names, like below:

```
root.bind('a', callback)
root.bind('A', callback)
root.bind('-', callback)
```

The exceptions are the spacebar (`<Space>`) and the less than sign (`<Less>`). You can also catch key combinations, such as `<Shift-F5>`, `<Control-Next>`, `<Alt-2>`, or `<Control-Shift-F1>`.

Note These examples all bind keypresses to `root`, which is our name for the main window. You can also bind keypresses to specific widgets. For instance, if you only want the left arrow key to work on a Canvas called `canvas`, you could use the following:

```
canvas.bind(<Left>, callback)
```

One trick here, though, is that the canvas won't recognize the keypress unless it has the GUI's focus. This can be done as below:

```
canvas.focus_set()
```

16.9 Event examples

Example 1 Here is an example where the user can move a rectangle with the left or right arrow keys.

```
from tkinter import *

def callback(event):
    global move
    if event.keysym=='Right':
```

```
        move += 1
    elif event.keysym=='Left':
        move -=1
    canvas.coords(rect,50+move,50,100+move,100)

root = Tk()
root.bind('<Key>', callback)
canvas = Canvas(width=200,height=200)
canvas.grid(row=0,column=0)
rect = canvas.create_rectangle(50,50,100,100,fill='blue')
move = 0

mainloop()
```

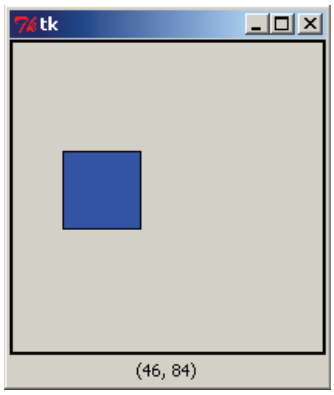
Example 2 Here is an example program demonstrating mouse events. The program starts by drawing a rectangle to the screen. The user can do the following:

- Drag the rectangle with the mouse (`<B1_Motion>`).
- Resize the rectangle with the mouse wheel (`<MouseWheel>`).
- Whenever the user left-clicks, the rectangle will change colors (`<Button-1>`).
- Anytime the mouse is moved, the current coordinates of the mouse are displayed in a label (`<Motion>`).

Here is the code for the program:

```
from tkinter import *

def mouse_motion_event(event):
    label.configure(text='({}, {})'.format(event.x, event.y))

def wheel_event(event):
    global x1, x2, y1, y2
    if event.delta>0:
        diff = 1
    elif event.delta<0:
        diff = -1
    x1+=diff
    x2-=diff
    y1+=diff
    y2-=diff
    canvas.coords(rect,x1,y1,x2,y2)

def b1_event(event):
    global color
    if not b1_drag:
        color = 'Red' if color=='Blue' else 'Blue'
        canvas.itemconfigure(rect, fill=color)
```

```
def b1_motion_event(event):
    global b1_drag, x1, x2, y1, y2, mouse_x, mouse_y
    x = event.x
    y = event.y
    if not b1_drag:
        mouse_x = x
        mouse_y = y
        b1_drag = True
        return
    x1+=(x-mouse_x)
    x2+=(x-mouse_x)
    y1+=(y-mouse_y)
    y2+=(y-mouse_y)
    canvas.coords(rect,x1,y1,x2,y2)
    mouse_x = x
    mouse_y = y

def b1_release_event(event):
    global b1_drag
    b1_drag = False

root=Tk()

label = Label()

canvas = Canvas(width=200, height=200)
canvas.bind('<Motion>', mouse_motion_event)
canvas.bind('<ButtonPress-1>', b1_event)
canvas.bind('<B1-Motion>', b1_motion_event)
canvas.bind('<ButtonRelease-1>', b1_release_event)
canvas.bind('<MouseWheel>', wheel_event)
canvas.focus_set()

canvas.grid(row=0, column=0)
label.grid(row=1, column=0)

mouse_x = 0
mouse_y = 0
b1_drag = False

x1 = y1 = 50
x2 = y2 = 100
color = 'blue'
rect = canvas.create_rectangle(x1,y1,x2,y2,fill=color)

mainloop()
```



Here are a few notes about how the program works:

1. First, every time the mouse is moved over the canvas, the `mouse_motion_event` function is called. This function prints the mouse's current coordinates which are contained in the `Event` attributes `x` and `y`.

2. The `wheel_event` function is called whenever the user uses the mouse (scrolling) wheel. The `Event` attribute `delta` contains information about how quickly and in what direction the wheel was moved. We just stretch or shrink the rectangle based on whether the wheel was moved forward or backward.

3. The `b1_event` function is called whenever the user presses the left mouse button. The function changes the color of the rectangle whenever the rectangle is clicked. There is a global variable here called `b1_drag` that is important. It is set to **`True`** whenever the user is dragging the rectangle. When dragging is going on, the left mouse button is down and the `b1_event` function is continuously being called. We don't want to keep changing the color of the rectangle in that case, hence the if statement.

4. The dragging is accomplished mostly in the `b1_motion_event` function, which is called whenever the left mouse button is down and the mouse is being moved. It uses global variables that keep track of what the mouse's position was the last time the function was called, and then moves the rectangle according to the difference between the new and old position.

   When the dragging is down, the left mouse button will be released. When that happens, the `b1_release_event` function is called, and we set the global `b1_drag` variable accordingly.

5. The `focus_set` method is needed because the canvas will not recognize the mouse wheel events unless the focus is on the canvas.

6. One problem with this program is that the user can modify the rectangle by clicking anywhere on the canvas, not just on rectangle itself. If we only want the changes to happen when the mouse is over the rectangle, we could specifically bind the rectangle instead of the whole canvas, like below:

```
canvas.tag_bind(rect, '<B1-Motion>', b1_motion_event)
```

7. Finally, the use of global variables here is a little messy. If this were part of a larger project, it might make sense to wrap all of this up into a class.

Chapter 17

GUI Programming III

This chapter contains a few more GUI odds and ends.

17.1 Title bar

The GUI window that Tkinter creates says Tk by default. Here is how to change it:

```
root.title('Your title')
```

17.2 Disabling things

Sometimes you want to disable a button so it can't be clicked. Buttons have an attribute `state` that allows you to disable the widget. Use `state=DISABLED` to disable the button and `state=NORMAL` to enable it. Here is an example that creates a button that starts out disabled and then enables it:

```
button = Button(text='Hi', state=DISABLED, command=function)
button.configure(state=NORMAL)
```

You can use the `state` attribute to disable many other types of widgets, too.

17.3 Getting the state of a widget

Sometimes, you need to know things about a widget, like exactly what text is in it or what its background color is. The `cget` method is used for this. For example, the following gets the text of a label called `label`:

```
label.cget('text')
```

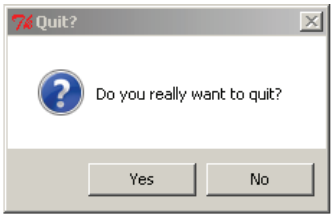
This can be used with buttons, canvases, etc., and it can be used with any of their properties, like `bg`, `fg`, `state`, etc. As a shortcut, Tkinter overrides the `[]` operators, so that `label['text']` accomplishes the same thing as the example above.

17.4 Message boxes

Message boxes are windows that pop up to ask you a question or say something and then go away. To use them, we need an import statement:

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

There are a variety of different types of message boxes. For each of them you can specify the message the user will see as well as the title of the message box. Here are three types of message boxes, followed by the code that generates them:




```
showinfo(title='Message for you', message='Hi There!')
askquestion(title='Quit?', message='Do you really want to quit?')
showwarning(title='Warning', message='Unsupported format')
```

Below is a list of all the types of message boxes. Each displays a message in its own way.

Message Box	Special properties
`showinfo`	`OK` button
`askokcancel`	`OK` and `Cancel` buttons
`askquestion`	`Yes` and `No` buttons
`askretrycancel`	`Retry` and a `Cancel` buttons
`askyesnocancel`	`Yes`, `No`, and `Cancel` buttons
`showerror`	An error icon and an `OK` button
`showwarning`	A warning icon an an `OK` button

Each of these functions returns a value indicating what the user clicked. See the next section for a simple example of using the return value. Here is a table of the return values:

Function	Return value (based on what user clicks)
`showinfo`	Always returns `'ok'`
`askokcancel`	`OK`—**`True`** `Cancel` or window closed—**`False`**
`askquestion`	`Yes`—`'yes'` `No`—`'no'`
`askretrycancel`	`Retry`—**`True`** `Cancel` or window closed—**`False`**
`askyesnocancel`	`Yes`—**`True`** `No`—**`False`** anything else—**`None`**
`showerror`	Always returns `'ok'`
`showwarning`	Always returns `'ok'`

17.5 Destroying things

To get rid of a widget, use its `destroy` method. For instance, to get rid of a button called `button`, do the following:

```
button.destroy()
```

To get rid of the entire GUI window, use the following:

```
root.destroy()
```

Stopping a window from being closed When your user tries to close the main window, you may want to do something, like ask them if they really want to quit. Here is a way to do that:

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

def quitter_function():
    answer = askquestion(title='Quit?', message='Really quit?')
    if answer=='yes':
        root.destroy()

root = Tk()
root.protocol('WM_DELETE_WINDOW', quitter_function)
mainloop()
```

The key is the following line, which cause `quitter_function` to be called whenever the user tries to close the window.

```
root.protocol('WM_DELETE_WINDOW', quitter_function)
```

17.6 Updating

Tkinter updates the screen every so often, but sometimes that is not often enough. For instance, in a function triggered by a button press, Tkinter will not update the screen until the function is done.

If, in that function, you want to change something on the screen, pause for a short while, and then change something else, you will need to tell Tkinter to update the screen before the pause. To do that, just use this:

```
root.update()
```

If you only want to update a certain widget, and nothing else, you can use the `update` method of that widget. For example,

```
canvas.update()
```

A related thing that is occasionally useful is to have something happen after a scheduled time interval. For instance, you might have a timer in your program. For this, you can use the `after` method. Its first argument is the time in milliseconds to wait before updating and the second argument is the function to call when the time is right. Here is an example that implements a timer:

```
from time import time
from tkinter import *

def update_timer():
    time_left = int(90 - (time()-start))
    minutes = time_left // 60
    seconds = time_left % 60
    time_label.configure(text='{}:{:02d}'.format(minutes, seconds))
    root.after(100, update_timer)

root = Tk()
time_label = Label()
time_label.grid(row=0, column=0)

start = time()
update_timer()

mainloop()
```

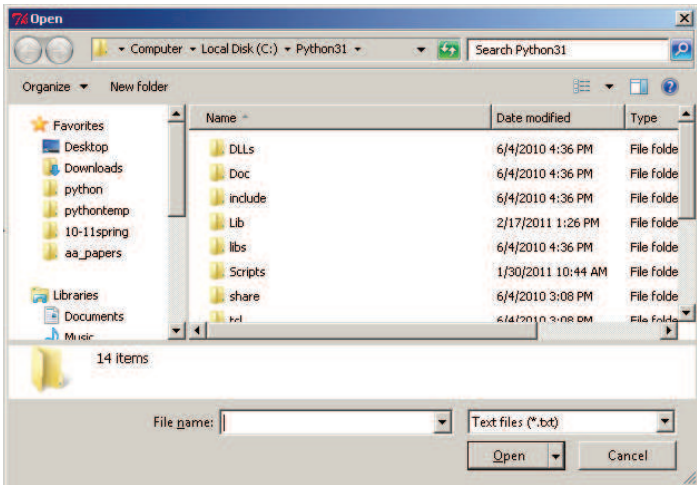
This example uses the `time` module, which is covered in Section 20.2.

17.7 Dialogs

Many programs have dialog boxes that allow the user to pick a file to open or to save a file. To use them in Tkinter, we need the following import statement:

```
from tkinter.filedialog import *
```

Tkinter dialogs usually look like the ones that are native to the operating system.



Here are the most useful dialogs:

Dialog	Description
`askopenfilename`	Opens a typical file chooser dialog
`askopenfilenames`	Like previous, but user can pick more than one file
`asksaveasfilename`	Opens a typical file save dialog
`askdirectory`	Opens a directory chooser dialog

The return value of `askopenfilename` and `asksaveasfilename` is the name of the file selected. There is no return value if the user does not pick a value. The return value of `askopenfilenames` is a list of files, which is empty if no files are selected. The `askdirectory` function returns the name of the directory chosen.

There are some options you can pass to these functions. You can set `initialdir` to the directory you want the dialog to start in. You can also specify the file types. Here is an example:

```
filename=askopenfilename(initialdir='c:\\python31\\',
                         filetypes=[('Image files', '.jpg .png .gif'),
                                    ('All files', '*')])
```

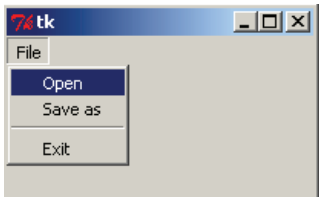
A short example Here is an example that opens a file dialog that allows you to select a text file. The program then displays the contents of the file in a textbox.

```
from tkinter import *
from tkinter.filedialog import *
from tkinter.scrolledtext import ScrolledText

root = Tk()
textbox = ScrolledText()
textbox.grid()

filename=askopenfilename(initialdir='c:\\python31\\',
                         filetypes=[('Text files', '.txt'),
```

```
                                        ('All files', '*')])
s = open(filename).read()
textbox.insert(1.0, s)

mainloop()
```

17.8 Menu bars

We can create a menu bar, like the one below, across the top of a window.



Here is an example that uses some of the dialogs from the previous section:

```
from tkinter import *
from tkinter.filedialog import *

def open_callback():
    filename = askopenfilename()
    # add code here to do something with filename

def saveas_callback():
    filename = asksaveasfilename()
    # add code here to do something with filename

root = Tk()
menu = Menu()
root.config(menu=menu)
file_menu = Menu(menu, tearoff=0)
file_menu.add_command(label='Open', command=open_callback)
file_menu.add_command(label='Save as', command=saveas_callback)
file_menu.add_separator()
file_menu.add_command(label='Exit', command=root.destroy)
menu.add_cascade(label='File', menu=file_menu)

mainloop()
```

17.9 New windows

Creating a new window is easy. Use the `Toplevel` function:

```
window = Toplevel()
```

You can add widgets to the new window. The first argument when you create the widget needs to be the name of the window, like below

```
new_window = Toplevel()
label = Label(new_window, text='Hi')
label.grid(row=0, column=0)
```

17.10 pack

There is an alternative to `grid` called `pack`. It is not as versatile as `grid`, but there are some places where it is useful. It uses an argument called `side`, which allows you to specify four locations for your widgets: `TOP`, `BOTTOM`, `LEFT`, and `RIGHT`. There are two useful optional arguments, `fill` and `expand`. Here is an example.

```
button1=Button(text='Hi')
button1.pack(side=TOP, fill=X)
button2=Button(text='Hi')
button2.pack(side=BOTTOM)
```

The `fill` option causes the widget to fill up the available space given to it. It can be either `X`, `Y` or `BOTH`. The `expand` option is used to allow the widget to expand when its window is resized. To enable it, use `expand=YES`.

Note You can use `pack` for some frames, and `grid` for others; just don't mix `pack` and `grid` within the same frame, or Tkinter won't know quite what to do.

17.11 StringVar

In Section 16.5 we saw how to tie a Tkinter variable, called an `IntVar`, to a check button or a radio button. Tkinter has another type of variable called a `StringVar` that holds strings. This type of variable can be used to change the text in a label or a button or in some other widgets. We already know how to change text using the `configure` method, and a `StringVar` provides another way to do it.

To tie a widget to a `StringVar`, use the `textvariable` option of the widget. A `StringVar` has `get` and `set` methods, just like an `IntVar`, and whenever you set the variable, any widgets that are tied to it are automatically updated.

Here is a simple example that ties two labels to the same `StringVar`. There is also a button that when clicked will alternate the value of the StringVar (and hence the text in the labels).

```
from tkinter import *

def callback():
    global count
    s.set('Goodbye' if count%2==0 else 'Hello')
    count +=1

root = Tk()

count = 0
s = StringVar()
s.set('Hello')

label1 = Label(textvariable = s, width=10)
label2 = Label(textvariable = s, width=10)
button = Button(text = 'Click me', command = callback)

label1.grid(row=0, column=0)
label2.grid(row=0, column=1)
button.grid(row=1, column=0)

mainloop()
```

17.12 More with GUIs

We have left out quite a lot about Tkinter. See Lundh's *Introduction to Tkinter* [2] for more. Tkinter is versatile and simple to work with, but if you need something more powerful, there are other third-party GUIs for Python.

Chapter 18

Further Graphical Programming

18.1 Python 2 vs Python 3

As of this writing, the most recent version of Python is 3.2, and all the code in this book is designed to run in Python 3.2. The tricky thing is that as of version 3.0, Python broke compatibility with older versions of Python. Code written in those older versions will not always work in Python 3. The problem with this is there were a number of useful libraries written for Python 2 that, as of this writing, have not yet been ported to Python 3. We want to use these libraries, so we will have to learn a little about Python 2. Fortunately, there are only a few big differences that we have to worry about.

Division The division operator, `/`, in Python 2, when used with integers, behaves like `//`. For instance, `5/4` in Python 2 evaluates to 1, whereas `5/4` in Python 3 evaluates to 1.2. This is the way the division operator behaves in a number of other programming languages. In Python 3, the decision was made to make the division operator behave the way we are used from math.

In Python 2, if you want to get 1.25 by dividing 5 and 4, you need to do `5/4.0`. At least one of the arguments has to be a float in order for the result to be a float. If you are dividing two variables, then instead of `x/y`, you may need to do `x/float(y)`.

`print` The `print` function in Python 3 was actually the `print` *statement* in Python 2. So in Python 2, you would write

```
print 'Hello'
```

without any parentheses. This code will no longer work in Python 3 because the `print` statement is now the `print` function, and functions need parentheses. Also, the current `print` function has those useful optional arguments, `sep` and `end`, that are not available in Python 2.

input The Python 2 equivalent of the `input` function is `raw_input`.

range The `range` function can be inefficient with very large ranges in Python 2. The reason is that in Python 2, if you use `range(10000000)`, Python will create a list of 10 million numbers. The `range` statement in Python 3 is more efficient and instead of generating all 10 million things at once, it only generates the numbers as it needs them. The Python 2 function that acts like the Python 3 `range` is `xrange`.

String formatting String formatting in Python 2 is a little different than in Python 3. When using the formatting codes inside curly braces, in Python 2, you need to specify an argument number. Compare the examples below:

```
Python 2: 'x={0:3d},y={1:3d},z={2:3d}'.format(x,y,z)
Python 3: 'x={:3d},y={:3d},z={:3d}'.format(x,y,z)
```

As of Python 3.1, specifying the argument numbers was made optional.

There is also an older style of formatting that you may see from time to time that uses the `%` operator. An example is shown below along with the corresponding new style.

```
Python 2: 'x=%3d, y=%6.2f, z=%3s' % (x,y,z)
Python 3: 'x={:3d},y={:6.2f},z={:3s}'.format(x,y,z)
```

Module names Some modules were renamed and reorganized. Here are a few Tkinter name changes:

Python 2	Python 3
Tkinter	tkinter
ScrolledText	tkinter.scrolledtext
tkMessageBox	tkinter.messagebox
tkFileDialog	tkinter.filedialog

There are a number of other modules we'll see later that were renamed, mostly just changed to lowercase. For instance, `Queue` in Python 2 is now `queue` in Python 3.

Dictionary comprehensions Dictionary comprehensions are not present in Python 2.

Other changes There are quite a few other changes in the language, but most of them are with features more advanced than we consider here.

Importing future behavior The following import allows us to use Python 3's division behavior in Python 2.

```
from __future__ import division
```

There are many other things you can import from the future.

18.2 The Python Imaging Library

The Python Imaging Library (PIL) contains useful tools for working with images. As of this writing, the PIL is only available for Python 2.7 or earlier. The PIL is not part of the standard Python distribution, so you'll have to download and install it separately. It's easy to install, though.

PIL hasn't been maintained since 2009, but there is a project called Pillow that it nearly compatible with PIL and works in Python 3.0 and later.

We will cover just a few features of the PIL here. A good reference is *The Python Imaging Library Handbook*.

Using images other than GIFs with Tkinter Tkinter, as we've seen, can't use JPEGs and PNGs. But it can if we use it in conjunction with the PIL. Here is a simple example:

```
from Tkinter import *
from PIL import Image, ImageTk

root = Tk()
cheetah_image = ImageTk.PhotoImage(Image.open('cheetah.jpg'))

button = Button(image=cheetah_image)
button.grid(row=0, column=0)

mainloop()
```

The first line imports Tkinter. Remember that in Python 2 it's an uppercase `Tkinter`. The next line imports a few things from the PIL. Next, where we would have used Tkinter's `PhotoImage` to load an image, we instead use a combination of two PIL functions. We can then use the image like normal in our widgets.

Images PIL is the Python *Imaging* Library, and so it contains a lot of facilities for working with images. We will just show a simple example here. The program below displays a photo on a canvas and when the user clicks a button, the image is converted to grayscale.

```
from Tkinter import *
from PIL import Image, ImageTk

def change():
    global image, photo
    pix = image.load()
```

```
    for i in range(photo.width()):
        for j in range(photo.height()):
            red,green,blue = pix[i,j]
            avg = (red+green+blue)//3
            pix[i,j] = (avg, avg, avg)
    photo=ImageTk.PhotoImage(image)
    canvas.create_image(0,0,image=photo,anchor=NW)

def load_file(filename):
    global image, photo
    image=Image.open(filename).convert('RGB')
    photo=ImageTk.PhotoImage(image)
    canvas.configure(width=photo.width(), height=photo.height())
    canvas.create_image(0,0,image=photo,anchor=NW)
    root.title(filename)

root = Tk()
button = Button(text='Change', font=('Verdana', 18), command=change)
canvas = Canvas()
canvas.grid(row=0)
button.grid(row=1)
load_file('pic.jpg')

mainloop()
```

Let's first look at the `load_file` function. Many of the image utilities are in the Image module. We give a name, `image`, to the object created by the `Image.open` statement. We also use the `convert` method to convert the image into RGB (Red-Green-Blue) format. We will see why in a minute. The next line creates an `ImageTk` object called `photo` that gets drawn to the Tkinter canvas. The `photo` object has methods that allow us to get its width and height so we can size the canvas appropriately.

Now look at the `change` function. The `image` object has a method called `load` that gives access to the individual pixels that make up the image. This returns a two-dimensional array of RGB values. For instance, if the pixel in the upper left corner of the image is pure white, then `pix[0,0]` will be `(255,255,255)`. If the next pixel to the right is pure black, `pix[1,0]` will be `(0,0,0)`. To convert the image to grayscale, for each pixel we take the average of its red, green, and blue components, and reset the red, green, and blue components to all equal that average. Remember that if the red, green, and blue are all the same, then the color is a shade of gray. After modifying all the pixels, we create a new `ImageTk` object from the modified pixel data and display it on the canvas.

You can have a lot of fun with this. Try modifying the change function. For instance, if we use the following line in the `change` function, we get an effect that looks like a photo negative:

```
pix[i,j] = (255-red, 255-green, 255-blue)
```

Try seeing what interesting effects you can come up with.

Note, though, that this way of manipulating images is the slow, manual way. PIL has a number of much faster functions for modifying images. You can very easily change the brightness, hue, and contrast of images, resize them, rotate them, and much more. See the PIL reference materials for more on this.

putdata If you are interested drawing mathematical objects like fractals, plotting points pixel-by-pixel can be very slow in Python. One way to speed things up is to use the `putdata` method. The way it works is you supply it with a list of RGB pixel values, and it will copy it into your image. Here is a program that plots a 300 × 300 grid of random colors.

```
from random import randint
from Tkinter import *
from PIL import Image, ImageTk

root = Tk()
canvas = Canvas(width=300, height=300)
canvas.grid()
image=Image.new(mode='RGB',size=(300,300))

L = [(randint(0,255), randint(0,255), randint(0,255))
      for x in range(300) for y in range(300)]

image.putdata(L)

photo=ImageTk.PhotoImage(image)
canvas.create_image(0,0,image=photo,anchor=NW)
mainloop()
```

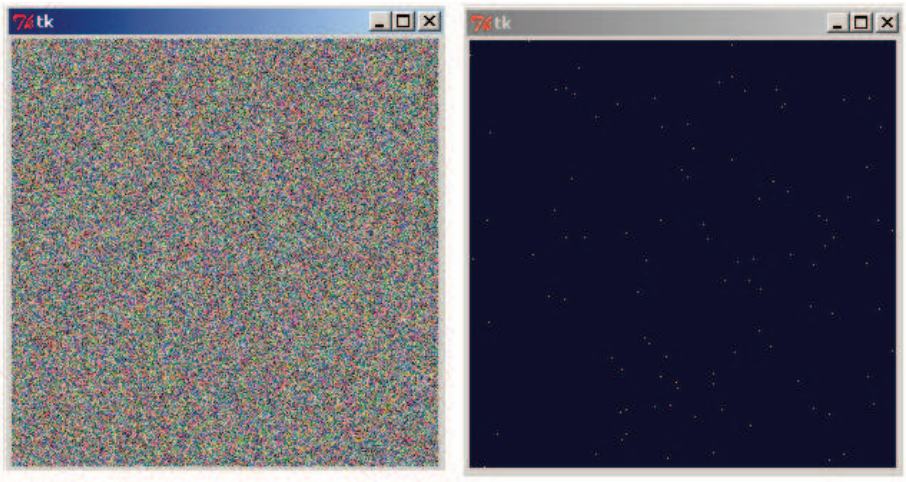


Figure 18.1: (Left) putdata example (Right) ImageDraw example

ImageDraw The `ImageDraw` module gives another way to draw onto images. It can be used to draw rectangles, circles, points, and more, just like Tkinter canvases, but it is faster. Here is a short example that fills the image with a dark blue color and then 100 randomly distributed yellow points.

```
from random import randint
from Tkinter import *
```

```
from PIL import Image, ImageTk, ImageDraw

root = Tk()
canvas = Canvas(width=300, height=300)
canvas.grid()
image=Image.new(mode='RGB',size=(300,300))
draw = ImageDraw.Draw(image)

draw.rectangle([(0,0),(300, 300)],fill='#000030')
L = [(randint(0,299), randint(0, 299)) for i in range(100)]
draw.point(L, fill='yellow')

photo=ImageTk.PhotoImage(image)
canvas.create_image(0,0,image=photo,anchor=NW)
mainloop()
```

To use `ImageDraw`, we have to first create an `ImageDraw` object and tie it to the `Image` object. The `draw.rectangle` method works similarly to the `create_rectangle` method of canvases, except for a few differences with parentheses. The `draw.point` method is used to plot individual pixels. A nice feature of it is we can pass a list of points instead of having to plot each thing in the list separately. Passing a list is also much faster.

18.3 Pygame

Pygame is a library for creating two-dimensional games in Python. It can be used to can make games at the level of old arcade or Nintendo games. It can be downloaded and easily installed from www.pygame.org. There are a number of tutorials there to help you get started. I don't know a whole lot about Pygame, so I won't cover it here, though perhaps in a later edition I will.