4

Events and Event Handlers

In this Chapter:

- ❑ Keyboard and Mouse Events
- ❑ Bind Methods
- ❑ The Bindtags List
- ❑ Event Names
- ❑ Timed Events
- ❑ The Event Class
- ❑ Virtual Events
- ❑ Forcing Events

Keyboard Events

Introduction

So far we have made use of the `command` option when constructing a button or radio button to link a widget event to an event handler. And, although this approach is fine for very simple programs, it has some limitations.

For example, we cannot get other widgets, such as a label, to react to an event nor can we get buttons to react to any other event other than being pressed.

Another problem occurs when we link more than one widget to the same event handler - we have no simple way of knowing which of the widgets has invoked the event handler when it begins to execute.

Binding

If we want a widget to react to an event which cannot be specified by setting that widget's `command` option, then we need to **bind** the new event to that widget. One way to do this is to use the `bind()` method which is available to every widget and has the following format:

bind(event_name, handler [,+])

This method is used to 'bind' the widget to one or more events. When the widget has focus and that event occurs the specified event-handler is run.

event_name	is a string giving the name of the event to which the widget is to be bound.
handler	is the name of the event handler to be executed when the event occurs.
'+'	if the + symbol is included, *handler* is added to the list of event handlers to be executed when the specified event occurs. If the + is missing, then *handler* replaces all existing event handlers for the specified event.

Let's say we want a button, *but1*, to react to the *a* key being pressed on the keyboard by executing an event handler called `handle()`, then we would write

```
but1.bind('a', handle)
```

When dealing with standard, printable characters from the keyboard (other than the space or < characters), the event name is simply a string containing the key involved.

The code for the event handler itself is a little different from those we linked to the `command` option of a button. These new event handler must take a parameter (traditionally called *e* or *event*). We'll discuss the purpose of this parameter later. For the moment, all we need to know is that it must be included.

The program in FIG-4.1 gives a simple demonstration of event binding by linking a `Button` widget to the pressing of the letter *b* on the keyboard. This causes the contents of a second, label widget to change.

FIG-4.1

Event Binding

```
#****** Keyboard Events ******

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***        Event Handlers          ***
# ***************************************

def change_label(e):
    """Change the label contents
    """
    lab1['text'] = "A key pressed"

#****************************************
# ***          GUI Layout            ***
# ***************************************
#*** Create window ***
root = Tk()

#*** Add Label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()
#*** Add button ***
but1 = ttk.Button(root, text = "Give Focus")
but1.pack()

#*** Make button react to the a key being pressed ***
but1.bind('b',change_label)

#*** Wait for events ***
root.mainloop()
```

Activity 4.1

In the *GUI* project, create a new file called *KeyboardBinding.py* and implement the code given in FIG-4.1.

Run the program. What happens when you press the **b** key?

If you start by pressing the b *key, nothing will happen. This is because the button does not have focus.*

Click on the button to give it focus. Now, press the **b** key again. What happens this time?

Modify your program so that it reacts to the **a** key rather than the **b** key.

Test and save your program.

If we want the event to be triggered by a capital **A** being pressed, then we need to change the `bind()` statement to read

```
but1.bind('A',change_label)
```

and when the program is run, we need to press **Shift-A** to get the event to occur.

Activity 4.2

Modify *KeyboardBinding.py* so that the event is triggered by a capital *A* being entered.

Test and save your program.

For the remainder of the examples, we'll stick with lowercase characters.

If we would like to run our event handler when either the *a* or *b* key is pressed, then we might try writing

```
but1.bind('ab', change_label)
```

but this does not have the desired effect.

Activity 4.3

Modify *KeyboardBinding.py* so that the call to `but1.bind()` matches the line given above.

What is required to cause the specified event to occur?

Save your program.

As we can see from Activity 4.3, binding to `'ab'` means that these to keys must be pressed in that order - **a** followed by **b**. If we want either key to cause the event handler to execute, then we need to write two separate bind statements:

```
but1.bind('a', change_label)
but1.bind('b', change_label)
```

Activity 4.4

Modify *KeyboardBinding.py* so that the event handler is executed when either *a* or *b* are pressed.

Test and save your program.

For the next example, we'll have the **a** key not only change the contents of the label but also the background colour of the window.

Now, although we could simply achieve this by adding another statement to existing event handler, we'll learn a bit more if we perform the colour change in a second handler which we'll call `change_screen()`.

If we were to link the **a** key to this new handler with the line

```
but1.bind('a', change_screen)
```

we would overide the **a** key's link to the `label_change()` handler. To have this single event (the pressing of the **a** key) cause both handlers to be executed, we must make use of the `bind()` method's third parameter and write

```
 but1.bind('a', change_screen, '+')
```

forcing the new handler to be executed along with the original handler.

Activity 4.5

Modify *KeyboardBinding.py* adding a new handler (called *change_screen()*) for the *a* key which changes the screen background colour to red. Remove the handler for the *b* key.

Test the program with and without the '+' option in the second call to `bind()`.

Save your program.

Return Values

When we call `bind()` to set up an event and its handler, the method returns a reference to the event-handler. For example, changing the last version of *KeyboardBinding.py* so that the bind statements become

```
h1 = but1.bind('a', change_label)
h2 = but1.bind('a', change_screen, '+')
```

and this is followed by the statement

```
print(h1, "and", h2)
```

then we get the following display:

```
1805128change_label and 3598656change_screen
```

We can find out which events a widget has been bound to previously by calling `bind()` without parameters. For example, adding the line

```
print(but1.bind())
```

to the previous program would display the tuple

```
('a',)
```

Activity 4.6

Modify *KeyboardBinding.py* so that pressing *a* calls `change_label()` and pressing *b* calls `change_screen()`.

In addition, display in the console window, the events to which *but1* is linked.

Save your program.

Key Names

Not all keys on the keyboard are so easily specified when naming an event. For example, how do we specify that an event is to occur when the **up arrow** key is pressed?

To get round this problem, all the troublesome keys are given a symbolic name (called **keysym**) as well as a numeric code equivalent to the symbolic name, (**keysym_num**) and a keycode number (**keycode**). These values are shown in FIG-4.2.

FIG-4.2

Symbolic Key Names

keysym	keysym_num	keycode	Physical Key
Alt_L	65513	64	Left-hand Alt key
Alt_R	65514	113	Right-hand Alt key
BackSpace	65288	22	Backspace
Cancel	65387	110	Break
Caps_Lock	65549	66	CapsLock
Control_L	65507	37	Left-hand Ctrl key
Control_R	65508	109	Right-hand Ctrl key
Delete	65535	107	Delete
Down	65364	104	↓
End	65367	103	End
Escape	65307	9	Esc
Execute	65378	111	SysRq
F1	65470	67	F1 key
F2	65471	68	F2 key
Fi	65469+i	66+i	F_i key (i = 1 to 11)
F12	65481	96	F12 key
Home	65360	97	Home
Insert	65379	106	Insert
Left	65361	100	→
Linefeed	106	54	Linefeed (Ctrl-J)
KP_0	65438	90	Keypad 0
KP_1	65436	87	Keypad 1
KP_2	65433	88	Keypad 2
KP_3	65435	89	Keypad 3
KP_4	65430	83	Keypad 4
KP_5	65437	84	Keypad 5
KP_6	65432	85	Keypad 6
KP_7	65429	79	Keypad 7
KP_8	65431	80	Keypad 8
KP_9	65434	81	Keypad 9

FIG-4.2
(continued)

Symbolic Key Names

keysym	keysym_num	keycode	Physical Key
KP_Add	65451	86	Keypad +
KP_Begin	65437	84	Keypad centre (5)
KP_Decimal	65439	91	Keypad . (decimal point)
KP_Delete	65439	91	Keypad delete
KP_Divide	65455	112	Keypad /
KP_Down	65433	88	Keypad ↓
KP_End	65436	87	Keypad End
KP_Enter	65421	108	Keypad Enter
KP_Home	65429	79	Keypad Home
KP_Insert	65438	90	Keypad Insert
KP_Left	65430	83	Keypad ←
KP_Multiply	65450	63	Keypad *
KP_Next	65435	89	Keypad PgDn
KP_Prior	65434	81	Keypad PgUp
KP_Right	65432	85	Keypad →
KP_Subtract	65453	82	Keypad - (minus)
KP_Up	65431	80	Keypad ↑
Next	65366	105	PageDown
Num_Lock	65407	77	NumLock
Pause	65299	110	Pause
Print	65377	111	PrtScrn
Prior	65365	99	PageUp
Return	65293	36	Enter (Return)
Right	65363	102	→
Scroll_Lock	65300	78	ScrLk
Shift_L	65505	50	Left-hand shift
Shift_R	65506	62	Right-hand shift
Tab	65289	23	Tab
Up	65362	98	↑

When specifying an event involving any of these keys, we may make use of the symbolic name.

Two other keys that make use of a symbolic name are the space bar (‘space’) and the less than key (‘less’).

All symbolic names must be enclosed in angled brackets (`< >`) when referenced in a call to `bind()`. For example, to have the window background colour change when the button (*but1*) in *KeyboardBinding.py* has focus and the **backspace** key is pressed, we would write

```
but1.bind('<BackSpace>',change_screen)
```

Activity 4.7

In *KeyboardBinding.py* begin by deleting the line which displays the events to which *but1* is linked.

Now change the binding so that the window background colour is changed by pressing the **Delete** key.

Test and save your program.

The key specified in an event name is known as the **event detail**.

So what if we want all keys to initiate execution of a specific event-handler? In that case, we use the term ‘`<Key>`’.

Activity 4.8

Modify *KeyboardBinding.py* so that the window background changes to red when *but1* has focus and any key is pressed.

What happens when the **a** key is pressed first?

Save your program.

As we can see from the results of Activity 4.8, using `'<Key>'` only assigns those keys not previously assigned.

Event Types

There will be times when we want an event to occur, not when we press a key, but when we release it. On these occasions we need to add the term `KeyRelease`. Hence, to have the *change_screen()* handler executed when we release the **Backspace** key, we would write

```
but1.bind('<KeyRelease BackSpace>', change_screen)
```

In fact, we could have included the term `KeyPress` when we want to react to a key being pressed:

```
but1.bind('<KeyPress Backspace>', change_screen)
```

but as you’ve found out from the previous examples, `KeyPress` is assumed, if you don’t include it.

The terms `KeyPress` and `KeyRelease` are classified as **event types** There are other event types, as well see later in this chapter.

Activity 4.9

Change *KeyboardBinding.py* so that the window background changes to white when *but1* has focus and the **Delete** key is released (do this by adding another event handler called `change_screen_white()`).

Test and save your program.

If we use `KeyRelease` on its own as in the line

```
but1.bind('<KeyRelease>', change_screen)
```

the event will occur when any key is released.

Note, that although we can detect a key combination being pressed, it is only possible to detect the release of a single key.

Using `KeyPress` on its own as in

```
but1.bind('<KeyPress>', change_screen)
```

has exactly the same effect as `'<Key>'` activating the event when any key is pressed.

Event Modifiers

Many window based commands have keyboard shortcuts which often involve holding down the *Ctrl* or *Alt* keys while another key or set of keys are pressed. This can be handled by adding an **event modifier** to the event name. Valid modifiers for a keyboard event are

- Alt
- Control
- Lock (caps lock on)
- Shift
- Double
- Triple
- Any

For example, we could have the window colour change linked to the key combination *Ctrl-C* using the line

```
but1.bind('<Control c>', change_screen)
```

Note that it is important to use the lowercase c.

Alternatively, the modifier and the key can be separated by a hyphen:

```
but1.bind('<Control-c>', change_screen)
```

Although primarily for use with the mouse buttons, the `Double` and `Triple` options can also be used for keyboard events. So, for example, the line

```
but1.bind('<Triple-a>', change_screen)
```

would activate when *but1* had focus and the a key was pressed three times in quick succession.

`Any` is used exclusively with the mouse buttons to mean that the event is activated by pressing any mouse button.

Activity 4.10

Change *KeyboardBinding.py* so that the window background changes to red only when the key combination *Alt-z* is pressed.

Test and save your program.

We are not limited to a single modifier in the event name. For example, we could have the `Button` widget respond to the key combination **Alt-Shift-Home** using the line

```
but1.bind('<Alt-Shift-Home>', change_screen)
```

However, be careful about the combination you choose, since there is always the possibility that the operating system will intercept the keys. For example, it would be a bad idea to try to catch **Ctrl-Alt-Delete** on a Microsoft Windows system.

The Event Class

The parameter we've been obliged to add to our latest event handlers is, in fact, an object of class `Event`. An object of this type is automatically passed to the event handler.

The `Event` class object contains many attributes from which we can harvest details of the event that has just occurred. The attributes that relate to keyboard events and more general attributes are listed below (some are only set by specific events):

char	a string containing the key pressed (but only if it is a standard printable ASCII character)
keycode	an integer giving the *keycode* value for the key (see FIG-4.2 for the *keycode* values).
keysym	a string giving the *keysym* name for the key (see FIG-4.2 for the *keysym* values).
keysym_num	an integer giving the *keysym num* value (see FIG-4.2 for the *keysym num* values).
serial	the current value of an integer value that is incremented every time an event occurs within your system. Be aware that many events happen outside your own program, so this number will not simply increment by one for each event within your program.
time	the current value of an integer value which is incremented every millisecond.
type	a integer code describing the type of event (a full set of values is given later).
widget	a reference to the widget that initiated the event.

The program in FIG-4.3 creates a text box, and executes an event handler on every key press. The handler then displays the event class attributes listed above.

FIG-4.3

Displaying Event Details

```
#*** Event Details ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***        Event Handlers          ***
# **************************************
def event_details(e):
    """Displays event details
    """
    data[0]['text'] = e.char
    data[1]['text'] = e.keycode
    data[2]['text'] = e.keysym
    data[3]['text'] = e.keysym_num
    data[4]['text'] = e.serial
    data[5]['text'] = e.time
    data[6]['text'] = e.type
    data[7]['text'] = e.widget
#****************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create text box ***
ent1 = ttk.Entry(root)
ent1.pack(side = 'left')

#*** Run handler when any key pressed ***
ent1.bind('<Key>', event_details)

#*** Add a frame ***
f1 = ttk.Frame(root)
f1.pack(side = 'right', ipadx = 50)

#*** Add labels to frame ***
label_texts = ('char     :', 'keycode   :', 'keysym    :',
↳'keysym_num :', 'serial   :' , 'time    :', 'type    :',
↳'widget   :')
data = []
for r in range(8):
    ttk.Label(f1, text = label_texts[r]).grid(column = 0, row = r,
    ↳sticky = E)
    data.append(ttk.Label(f1, text = "..."))
    data[r].grid(column = 1, row = r, sticky = W)

#*** Wait for events ***
root.mainloop()
```

Activity 4.11

Create a new file called *EventDetails.py* and enter the code given in FIG-4.3. Press various keys and observe the values returned.

Save the program.

Activity 4.12

Create a new file called *Disabling.py* and produce a program which displays six buttons, each button should be changed to the disabled state when clicked on by the left mouse button.

(HINT: the event handler should make use of the widget `Event` object's `widget` attribute and set the widget's `state` attribute to *disabled*.)

Test and save the program.

Other Bind Methods

It seems a bit contrived to be forced to give the button in *KeyboardBinding.py* focus before we can change the window's background colour. After all, many programs will produce an effect irrespective of what GUI element has focus when a key is pressed - for example, pressing the **F1** key to access the Help features of an application.

Luckily, Tkinter has other bind methods available.

bind_all(event, handler [,'+'])

This method binds all widgets in an application to the specified *event* and *handler*. For example, in *KeyboardBinding.py*, we could link all widgets to the **a** key being pressed and `change_screen()` using the line

```
root.bind_all('a', change_screen)
```

It is unimportant which widget is used to call the method.

The third parameter, '+' can be added if this handler is to be added to existing handlers for this event.

Activity 4.13

Modify *KeyboardBinding.py* so that the window background changes to red when **Alt-z** is pressed and to white when the keys are released irrespective of the button having focus.

Test and save your program.

bind_class(class, event, handler[,'+'])

Back in Chapter 2 when we were examining the attributes of our first widgets, we saw that every widget can be assigned to a named group using the `class` attribute (not to be confused with a class in the object-oriented sense).

This method binds all widgets of a specified group to an event and handler. For example, in Tkinter, every button belongs to the grouping `TButton` (that is to say, that every button's `class` attribute has the value `TButton`). If we were to use the line

```
root.bind_class('TButton','a',handle)
```

every button within the application would, when in focus, react to the **a** key being pressed by executing the function called `handle()`.

It is unimportant which widget is used to call the method.

A final parameter, `'+'`, can be added if this handler is to be added to existing handlers for this event.

We are free to assign a new value of our own to a widget's class attribute. This should be done when the widget is first created. The group name must be assigned using the term `class_` since `class` is a reserved term (we had the same problem earlier using `in_` rather than `in`). So we might put a button (*but1*) and a text box (*ent1*) in the same grouping using the lines

```
but1 = ttk.Button(root, text = "TG button", class_='TestGroup')
ent1 = ttk.Entry(root, class_='TestGroup')
```

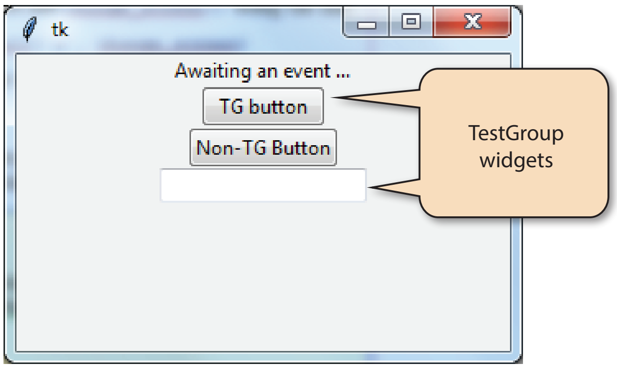
Now, if we assign an event to the *TestGroup* group, both of these widgets will execute the associated handler under the correct circumstances.

However, changing a widget's default `class` value can have undesired consequences as we'll see in the next program.

The layout shown in FIG-4.4 contains a text box and a button belonging to a grouping called *TestGroup* (as indicated). The other widgets do not belong to a grouping.

FIG-4.4

Setting Class Names: Display



The two group widgets are coded to change the contents of the label when the **Alt-z** key combination is pressed. The program code is given in FIG-4.5.

FIG-4.5

Setting Class Names: Code

```
#*** Binding Groupings ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***         Event Handlers          ***
# ***************************************
def change_label(e):
    """Changes the label contents
    """
    lab1['text'] = "Changed by a TestGroup widget"

#****************************************
# ***           GUI Layout            ***
# ***************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')
```

FIG-4.5
(continued)

Setting Class Names: Code

```
#*** Add ungrouped Label ***
lab1 = ttk.Label(root, text = "Awaiting an event ...")
lab1.pack()

#*** Add TestGroup button ***
but1 = ttk.Button(root, text = "TG button", class_ = 'TestGroup')
but1.pack()

#*** Add ungrouped button ***
but2 = ttk.Button(root, text = "Non-TG Button")
but2.pack()

#*** Add TestGroup text box ***
ent1 = ttk.Entry(root, class_ = 'TestGroup')
ent1.pack()

#*** Change label if Alt-z pressed when a ***
#*** TestGroup widget has focus           ***
but1.bind_class('TestGroup','<Alt-z>', change_label)

#*** Wait for events ***
root.mainloop()
```

Activity 4.14

Create a new *GroupBinding.py* and implement the code given in FIG-4.5.

When you run the program, can the widgets in group *TestGroup* be given focus by clicking on them? Can the widgets in the group be given focus using the *Tab* key?

Can data be typed into the text box?

Give focus to either of the widgets in group *TestGroup* and press **Alt-z**. Is the event handler executed?

Save your program.

The results of Activity 4.14 shows us that, although there may be times when we'd like a set of different widgets to belong to the same grouping, changing the class name causes too many problems. Luckily, there is another way to achieve the grouping.

Bindtags

There is an "under the hood" detail on how events are linked to widgets that can be worth knowing about.

Every widget maintains a list of values called **bindtags**. A typical bindtags list will contain exactly four entries initially (although the toplevel window only has three). Below is the bindtags lists for a `Label` widget (*lab1*) and two `Button` widgets (*but1* and *but2*) from a program.

```
('.2734064',  'TLabel',   '.',   'all')
('.36138800',  'TButton',  '.',   'all')
('.44461872',  'TButton',   '.',   'all')
```

- The first value in the bindtags list represents the ID of that specific widget. The bindtag name for a specific widget always starts with a full stop.
- The second value in each list represents the grouping (`class`) to which the widget belongs.
- The third entry is just a single full stop. This is the ID for the main, `Toplevel` window.
- The final entry in the list is the fixed term `all`.

The last two entries are the same for all widgets (other than the main window).

The bindtags list for any main window is ('.', 'Tk', 'all') which is the window's ID (`.`), the class to which it belongs (`Tk`) and the fixed term `all`.

We can discover the contents of a widget's bindtags list using the `bindtags()` method. For example, we could display the bindtags list for widget *lab1* using the line

```
print(lab1.bindtags())
```

Activity 4.15

Create a new program called *UnderstandingBindtags.py* containing a label and two buttons and a text box.

Each widget should store its variable name in its `text` property. For example, if we store the label's ID in *lab1*, then the label's text should be *'lab1'*.

Have the program display the bindtags list for the main window and each of its four widgets.

Test and save your program.

So what's so important about the bindtags list? When we add an event binding to a widget, that binding is actually linked to an entry in the bindtags list.

When we bind an event to a widget, with a statement that starts

```
but1.bind(...
```

what we are doing is actually binding that event to the first entry in that widget's taglist - its ID. Since every widget has a unique ID, only the calling widget is linked to the event (see FIG-4.6).

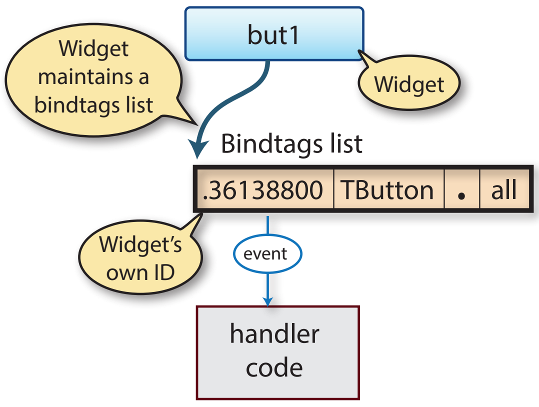


FIG-4.6

Binding a Widget Name

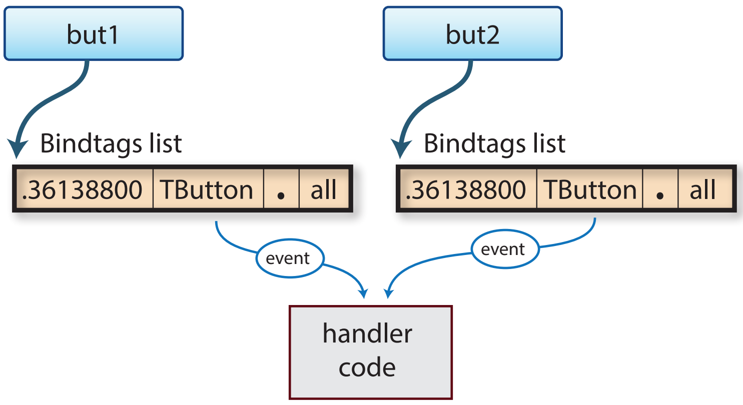
When we start a statement with

```
but1.bind_class('TButton',...
```

we are binding the event to EVERY widget that contains the term *TButton* in its bindtags list. In effect, this means to every button in our application (see FIG-4.7).

FIG-4.7

Binding the Widget Class Name



The fact that the method is called by widget *but1* is irrelevant - in fact, we could have set up an event for all widgets in the *TButton* class by making the call from a `Label` widget or the main window (*root*) neither of which belong to the *TButton* class

When we use

```
bind_all(...
```

we are binding the event to every widget that contains the term '*all*' in its bindtags list.

Since every widget contains that value, the link is to every widget in the application, including the main window (see FIG-4.8).

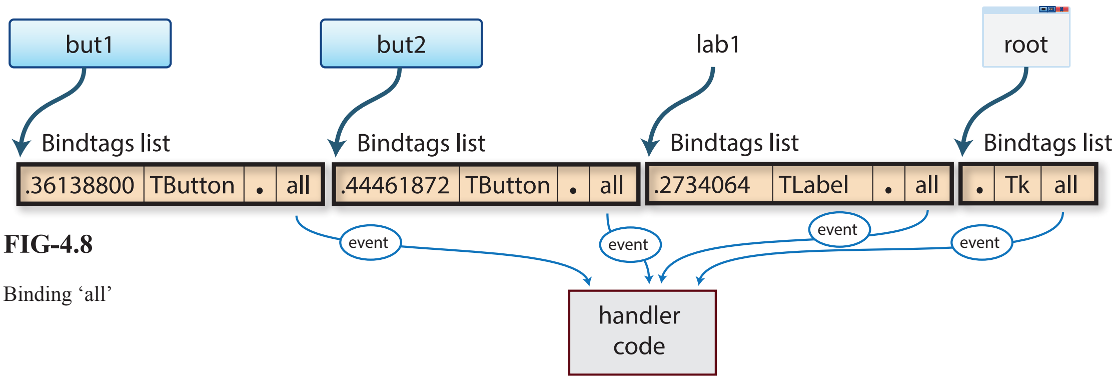


FIG-4.8

Binding 'all'

Using

```
root.bind(...
```

links an event to the main window (assuming we've called it *root*).

Now, since the main window's tag name is '.', and since every widget contains that name in its taglist, we are effectively linking the event to every widget (see FIG-4.9).

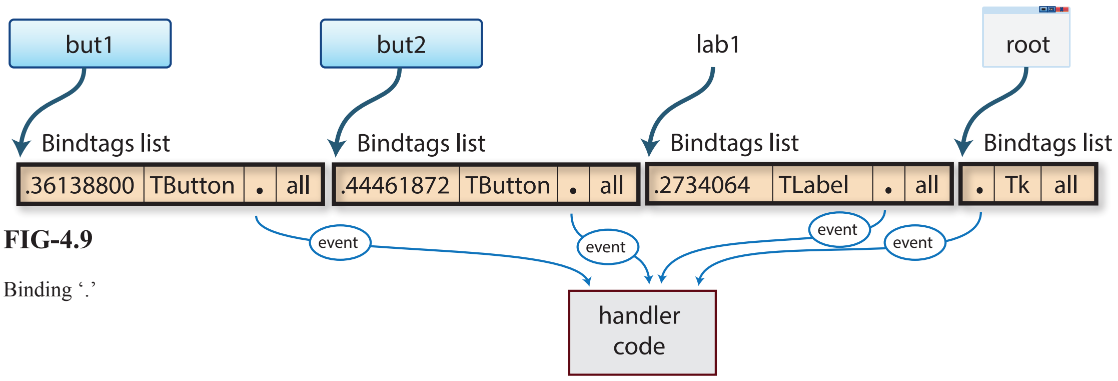


FIG-4.9

Binding '.'

The difference between *'all'* and *'.'* only arises when an application has more than one main window.

When an event occurs, the widget in focus performs the following logic:

```
FOR each tag in bindtags list DO
    IF the current event is defined for this tag THEN
        Execute handler
    ENDIF
ENDFOR
```

We can easily demonstrate this effect by modifying our previous program to display the name of the widget that executes an event handler in the label. For example, adding the lines

```
#****************************************
# ***          Event Handlers         ***
# ***************************************
def show_name(e):
    """Displays widget name in label
    """
    lab1['text'] = e.widget['text']
```

and

```
#*** Add event to first button only ***
but1.bind('<Alt-z>', show_name)
```

defines an event, *Alt-z*, and a handler linked to the first button in the program.

Activity 4.16

Modify *UnderstandingBindtags.py* by removing the calls to `print()` and adding the lines given above at appropriate points in your code.

Run the program and give the first button focus. What happens when you press **Alt-z**?

Give the second button focus and press **Alt-z**. What happens this time?

What happens when the text box has focus and **Alt-z** is pressed?

Save your program.

As we should have expected, this time only the first button reacts to the event because the event has been linked to the first entry in that button's bindtags - that is, to its own, unique ID.

Activity 4.17

In *UnderstandingBindtags.py* change the binding for the event **Alt-z** so that it is now linked to the *TButton* class rather than *but1*.

Run the program and give the first button focus. What happens when you press **Alt-z**?

Give the second button focus and press **Alt-z**. What happens this time?

What happens when the text box has focus and **Alt-z** is pressed?

Save your program.

This time we have linked all `TButton` class widgets to the event **Alt-z**. So, when a widget of that class (a widget with that term in its bindtags list) gains focus, the event is handled.

In the next Activity we'll link the event **Alt-z** to the '`all`' bindtag.

Activity 4.18

In *UnderstandingBindtags.py* change the existing binding statement the `bind_all()` method.

Before we run the program, there's a slight problem we need to cope with. If the main window executes the event handler, it doesn't have a `text` *attribute so we would get an error situation. To stop this happening, we can change the handler code to deal with the window separately:*

```
def show_name(e):
   """ Displays widget name in label
   """
   if e.widget == root:
     lab1['text'] = 'root'
   else:
     lab1['text'] = e.widget['text']
```

How does each widget react to **Alt-z** when it has focus?

Save your program.

The final option we have is to bind an event to the `'.'` bindtag.

We can bind to the `'.'` bindtag in one of two ways. We could use the `bind_class()` function because, despite its name, the `bind_class()` method can link to any name in a bindtags list, not just the one that represents the widget's class (grouping). This allows us to write.

```
but1.bind_class('.','<Alt-z>', show_name)
```

but the more accepted way of achieving the result is to use *root* (the main window name in our examples) and `bind()`:

```
        root.bind('<Alt-z>', show_name)
```

Remember the main window's ID in the bindtags list is `'.'` - the only ID that's listed in every other widget's bindtags list - so it links the event to every other widget.

Activity 4.19

In *UnderstandingBindtags.py* change the binding for the event **Alt-z** so that it is now linked to the main window.

How does each widget react to **Alt-z** when it has focus?

Save your program.

Linking an Event to More Than One Entry in the BindTag List

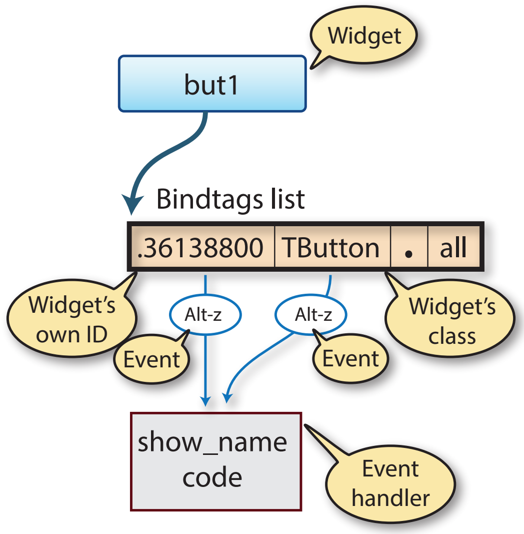
Another option we have is to link an event to more than one entry in a widget's bindtags list. For example, if we were to use the line

```
    but1.bind('<Alt-z>', show_name)
    but1.bind_class('TButton', '<Alt-z>',show_name
```

we have created the situation which is represented visually in FIG-4.10

FIG-4.10

Binding Multiple Bindtags: Concept



The diagram highlights the fact that, when widget *but1* is in focus and the **Alt-z** key combination is presssed, that event will trigger the execution of the code in `show_name()` twice since two values in the bindtags list link the same event to the same handler.

The program in FIG-4.11 is a variation on *UnderstandingBindtags.py* and demonstrates the above situation by creating an event handler which increments a global variable every time the handler is executed. The value of the global variable is stored in the program's label widget, so we can see how often the handler is executed for a given event.

In this first version of the program the event **Alt-z** is linked only to the first button's ID.

FIG-4.11

Binding Multiple Bindtags: Code

```
#*** Binding events to Multiple Tags  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#***       Global Variables          ***
count = 0

#****************************************
# ***        Event Handlers          ***
# ***************************************
def add_to_count(e):
    """Label shows value of count
    """
    global count

    #*** Increment count ***
    count += 1
    #*** Display count's value in label ***
    lab1['text'] =str(count)

#****************************************
# ***          GUI Layout            ***
# ***************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label ***
lab1 = ttk.Label(root, text = "0")
lab1.pack()

#*** Create two buttons ***
but1 = ttk.Button(root, text = "but1")
but1.pack()
but2 = ttk.Button(root, text = "but2")
but2.pack()

#*** Create a text box ***
rbut1 = ttk.Entry(root, text = "ent1")
rbut1.pack()

#*** Add event to first button only ***
but1.bind('<Alt-z>', add_to_count)


#*** Wait for events ***
root.mainloop()
```

Activity 4.20

Modify *UnderstandingBindtags.py* to match the code given in FIG-4.11.

What happens this time as each widget gains focus and **Alt-z** is pressed?

Save your program.

So far, we just have a fairly standard response to the event **Alt-z**; only the first button responds to the event and each event increments the count by 1.

Activity 4.21

Modify *UnderstandingBindtags.py* by adding a second binding for **Alt-z** (don't remove the original bind statement). This time, bind the event to the class `TButton`.

What happens this time as each widget gains focus and **Alt-z** is pressed?

Now add a third binding for **Alt-z**, this time to '.'.

What happens this time as each widget gains focus and **Alt-z** is pressed?

Save your program.

BindTag Ordering

The order in which the bindtags are interrogated can be important and we can see this if we try copying the contents of a text box to a label as demonstrated by the program in FIG-4.12 where every keystroke within the text causes its contents to be copied to the label.

FIG-4.12

Reordering the Bindtags List

```
#*** Bindtag Order ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#**************************************
# ***        Event Handlers        ***
# *************************************
def change_label(e):
    """Sets label to contents of text box
    """
    lab1['text'] = ent1.get()

#**************************************
# ***          GUI Layout          ***
# *************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()

#*** Create text box ***
ent1 = ttk.Entry(root)
ent1.pack()

#*** Bind text box to any key press ***
ent1.bind('<Key>', change_label)

#*** Wait for events ***
root.mainloop()
```

Activity 4.22

Create a new file called *TagOrder.py* and enter the code given in FIG-4.12.

When you run the program and enter data in the text box, what is unusual about the value displayed in the label?

Add a statement to display the bindtags for *ent1* in the console window.

Test and save your program.

The `bindtags()` method can be used with parameters to reorder the bindtags. The bindtags for the text box in the last program was with the widget first and it's class second: ('.36208112', 'TEntry', '.', 'all'). We could reorder this tuple using the line

```
ent1.bindtags(('TEntry', ent1,'.','all'))
```

Note that the new order must be given as a tuple (not a list) and that the widget ID in the tuple is given by specifying the variable, used earlier in the program to store the widget ID.

Activity 4.23

Modify *TagOrder.py* changing the order of the bindtags to match that given in the example above. What happens when you run the program this time?

Save your program.

In Activity 4.22 we saw how the contents of the label seemed to be one character behind what had been typed, now the two values match. So what is going on?

The truth is that Tkinter automatically attaches a hidden event handler to a widget's grouping (the value defined by the widget's `class` attribute). This handler does what we would expect for that type of widget. In the case of an `Entry` widget, that means it adds the typed character to the data displayed in the text box; in the case of a button, it creates the depressed look when the button is clicked.

In the original set up for our text box (*ent1*), the handler attached to the widget's ID was run first and then the handler for the `TEntry` grouping was run later. Since it is the handler for `TEntry` that is responsible for adding the character just typed we can see that running the widget's handler first is always going to mean we are one character behind. By reversing the order of these two bindtags, we also reverse the order in which their handlers are executed, meaning that the latest character is added to the text box before we take the contents of that text box and copy it to the label.

It now becomes clear why the `Button` and `Entry` widgets that we regrouped as *TestGroup* in an earlier program no longer responded to being clicked. This was because the two objects no longer executed the hidden handlers associated with their original groupings `TButton` and `TEntry` respectively.

This is why it's not normally a good idea to change a widget's grouping via the class name.

Adding to a Widget's Bindtag List

Although we now see it's a bad idea to change a widget's class grouping, it would be nice to be able to group widgets without losing their default behaviour. Luckily, we can easily do this by adding an entirely new bindtag entry to a widget's list.

Returning to our *GroupBindings.py* program, we could remove the class definition in the constructor and replace this with a bindtag of the same name. For example, we could replace

```
but1 = ttk.Button(root, text = "TG button", class_='TestGroup')
```

with

```
but1 = ttk.Button(root, text = "TG button")
but1.bindtags(but1,'TButton','.','all', 'TestGroup')
```

Activity 4.24

Reload *GroupBindings.py* (which we last looked at in Activity 14.4) and modify the code so that *but1* and *ent1* no longer have `class_` values in their constructors but both have '*TestGroup*' added to their bindtag lists.

Do both buttons and the text box now take focus when clicked?
Does the **Alt-z** event still work for both these widgets?

Save your program.

We can see from the result of Activity 4.24 the method name `bind_class()` is more than a little misleading since its real purpose is to bind to any widget whose bindtag list contains the term defined in the first parameter.

Stopping BindTag Processing

There will be times when we'd like a greater control over how the bindtag list is processed. For example, let's assume a program contains a text box in which the user is meant to type in numeric values only. One way to ensure that only numeric values are entered is simply not to pass any non-numeric key presses to the hidden `TEntry` handler.

The program in FIG-4.13 represents the first stage in a program that stops non-numeric keys being accepted by a text box. The program contains a label and a text box. The label displays the most recently pressed key, or an error message if that key is not numeric. At this point, the text box displays all keys pressed (numeric and non-numeric).

FIG-4.13

Stopping Bindtags List Processing

```
#*** Controlling Text Box Entry  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#***************************************
# ***        Event Handlers          ***
# ***************************************
```

FIG-4.13
(continued)

Stopping Bindtags List Processing

```
def show_key_pressed(e):
    """Label shows last numeric key pressed or error message
    """
    #*** If numeric display it in label ***
    if e.char >= '0' and e.char <= '9':
        lab1['text'] = e.char
    else: # not numeric
        #*** Display error message ***
        lab1['text'] = "Not numeric"


#****************************************
# ***          GUI Layout            ***
# ***************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label for key ***
lab1 = ttk.Label(root, text = "")
lab1.pack()

#*** Create text box ***
ent1 = ttk.Entry(root)
ent1.pack()

#*** Handle every key press ***
ent1.bind('<Key>', show_key_pressed)


#*** Wait for events ***
root.mainloop()
```

Activity 4.25

Create a new file called *StoppingBindTags.py* which implements the code given in FIG-4.13.

Test the program by pressing both numeric and non-numeric keys.

Save the program.

What's happening here is that the first bindtag in the text box's list is executing the `show_key_pressed()` handler which correctly detects that a non-numeric key has been pressed, but then the `TEntry` bindtag is running the hidden, default handler for that class which adds the pressed key to the text within the box.

break

What we would like to do is to stop the handler for `TEntry` being executed if `show_key_pressed()` detected a non-numeric key. And this is done by having our handler return the string '`break`'.

When a handler returns the term '`break`' this breaks the run through the bindtags list for the widget that called this handler. No other entries in the bindtags list are processed.

Activity 4.26

Modify *StoppingBindTags.py* by adding the line

```
return 'break'
```

as part of the `else:` option in `show_key_pressed()`.

What happens this time when non-numeric keys are pressed?

Save the program.

Unbinding

We can remove an event from a widget, making that widget no longer responsive to that event using the widget's `unbind()` method.

unbind(event)	removes *event* from the calling widget.

The program in FIG-4.14 creates a label, a text box and a button. The label displays a count of how many *a*'s have been entered in the text box. However, when the button is pressed, the count is discontinued by unbinding the *'a'* event defined for the text box.

FIG-4.14

Unbinding

```
#*** Unbinding ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#***         Global Variables        ***
count = 0

#****************************************
# ***        Event Handlers          ***
# **************************************
def add_to_count(e):
    """ Increment count and update label
    """
    global count

    count += 1
    lab1['text'] = str(count)


def deactivate():
    """Unbind text box's e count
    """
    ent1.unbind('a')

#****************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')
```

FIG-4.14
(continued)

Unbinding

```
#*** Create label ***
lab1 = ttk.Label(root, text="0")
lab1.pack()
#*** Create text box ***
ent1 = ttk.Entry(root)
ent1.pack()
ent1.bind('a', add_to_count)

#*** Create button ***
but1 = ttk.Button(root,text="Stop 'a' count",command = deactivate)
but1.pack()


#*** Wait for events ***
root.mainloop()
```

Activity 4.27

Create a new file called *Unbinding.py* and implement the code given in FIG-4.14.

Test and save the program.

Although this section has concentrated on keyboard events, much of what has been written is also relevant to other event types. For example, we can add or rearrange bindtags and unbind events just as easily when dealing with mouse or timed events. The next sections of this chapter will cover the other types of events.

Summary

- Some widgets can link an event to an event handler using their `command` attribute.
- Even when available, using the command option to link an event to a handler has limitations.
- Every widget has a bindtags list.
- The entries in a bindtags list are used when binding an event and a event handler to a widget.
- A bindtags list contains:

  a unique ID for the widget to which it belongs.
  the name of the class to which the widget belongs
  '.' entry signifying the parent widget
  'all' entry

- When an event occurs, Python interrogates the bindtags list for the widget involved, and executes any handlers linked to that event for that bindtag entry.
- Use the widget's `bindtags()` method to change the order of the entries in a widget's bindtag list.
- Use the widget's `bindtags()` method to specify the contents of a widget's bindtag list.
- When using `bindtags()`, give the widget's variable name to specify the widget's ID.

- Using the `bindtags()` method allows new entries to be added to a bindtags list.
- Use `bind()` with arguments to link any widget to any event and to any handler. `bind()` returns a reference to the event handler.
- The `bind()` method links the event and handler specified to the widget's ID given in that widget's bindtags list.
- Use `bind()` without arguments to return a list of which events a widget is linked to.
- Use `bind_all()` to link all widgets to a specific event and handler.
- Using `bind_all()` links the event and handler to the `'all'` entry in every widget's bindtags list.
- Use `bind_class()` to bind all widgets which contain a specified name in their bindtags list to a given event and handler.
- If an event handler returns the string `'break'`, the remainder of the calling widget's bindtags list is not processed, so any other handlers that should be executed for this event are ignored.
- Use the `unbind()` method to make a widget no longer respond to a specific event.
- Some keys are assigned symbolic names. (e.g. 'Down' for the down arrow key)
- Specific key press events are identified by the key's letter (e.g. `'a'`).
- Keys with symbolic names are enclosed in angled brackets (e.g. `'<Down>'`).
- Use `'<Key>'` to react to any key press.
- The full format for identifying an event in the `bind()` argument list is

  event type   event modifier   event detail

- Event types for keyboard events are:

  ```
  KeyPress
  KeyRelease
  ```

- Event modifiers for keyboard events are:

  ```
  Alt
  Control
  Lock
  Shift
  Double
  Triple
  Any
  ```

- Event details for keyboard events are:

  key letter
  key symbolic name

- The event modifier and event detail can be separated by a space or hyphen (e.g. `'<KeyPress Alt-a>'`)
- When an event is linked to an event handler using `bind()`, the event handler must be written with a parameter of class `Event`.
- The `Event` object passed to an event handler, gives details of the widget which has triggered the call to the event handler.

- Details within an `Event` object include:

serial	event number (incremented every time an event occurs)
time	time event occurred in milliseconds
type	code for event’s type
widget	reference to widget calling handler

Keyboard-specific events details:

char	the key pressed (if printable ASCII)
keycode	the *keycode* value for the key pressed
keysym	the symbolic name for the key
keysym_num	the *keysym num* value for the key

Mouse Events

Introduction

As well as triggering events from the keyboard, events can be triggered by the mouse - either simply by moving the mouse pointer or clicking on the mouse buttons. A set of event names and `Event` class attributes are defined for mouse-related events.

Event Names

Like keyboard events, a mouse event term can consists of an event modifier, an event type and an event detail. The simplest of these contains only a type. Mouse movement (without any mouse buttons being pressed) offers three options:

‘<Enter>’	this event occurs when the mouse pointer enters the widget.
‘<Leave>’	this event occurs when the mouse pointer exits the widget.
‘<Motion>’	this event occurs when the mouse pointer moves within the widget.

Mouse events that make use of the mouse buttons are:

‘<1>’, ‘<2>’, ‘<3>’	these event names represent the pressing of the left mouse button (`'<1>'`), the centre button (`'<2>'`), and the right button (`'<3>'`). The term `Button` may be included in these event names, if you wish, giving us `'<Button-1>'`, `'<Button-2>'` and `'<Button-3>'`.
‘<Double-1>’, ‘<Double-2>’, ‘<Double-2>’	these events respond to the double clicking of the left, right or centre buttons. Again you are free to add the term `Button` immediately after the word `Double`.
‘<ButtonRelease-1>’, ‘<ButtonRelease-2>’, ‘<ButtonRelease-3>’	this event occurs when the specified mouse button is released.
‘<B1-Motion>’, ‘<B2-Motion>’,‘<B3-Motion>’	this event occurs when the mouse is dragged with the specified button held down.
‘<MouseWheel>’	this event occurs when the mousewheel is rotated.

Attributes in Event

The `Event` class holds a few additional attributes that reveal the state of the mouse when an event occurs. These are

num	contains the number of the mouse button pressed (1, 2, 3).

x, **y** the coordinates of the mouse pointer measured from the top-left corner of the widget.

x_root, **y_root** the coordinates of the mouse pointer measured from the top-left corner of the screen.

state an integer giving the state of all the modifier keys. The value of this variable can tell us which special keyboard keys and mouse buttons are being pressed when a mouse event occurs. The keys and their codes are (in hexadecimal):

Shift	0x00001
Caps Lock	0x00002
Ctrl	0x00004
Num Lock	0x00008
Left mouse button	0x00100
Centre mouse button	0x00200
Right mouse button	0x00400
Alt (on left)	0x20000
Alt (on right)	0x20004

When more than one of these keys are pressed at the same time, the value of `state`, is determined by ORing the values of those keys. For example, pressing the left mouse button and the **Ctrl** key, sets `state` to 0x00104. The only combination that cannot be detected is right **Alt** and **Ctrl** since 0x20004 OR 0x00004 is 0x20004,

delta this property returns the angle through which the mouse wheel has been turned. A negative value is returned if the wheel is rolled towards the user; a positive value if it is rolled away from the user. Typically, returned values seem to be -28 or +28.

The program in FIG-4.15 demonstrates all three of the mouse-move events (`<Motion>`, `<Enter>` and `<Leave>`) by constantly displaying the mouse coordinates as it moves within the main window and changing an image-based button from greyscale to colour and back again as the mouse pointer moves over and away from the button.

FIG-4.15

Handling Mouse Events

```
#*** Mouse Movement Events ***


#*** Import modules ***
from tkinter import *
from tkinter import ttk


#****************************************
# ***         Event Handlers          ***
# ***************************************
def display_coordinates(e):
    """ Displays the mouse coordinates in label
    """
    lab1['text'] = '('+str(e.x)+','+str(e.y)+')'
```

FIG-4.15
(continued)

Handling Mouse Events

```
def change_to_colour(e):
    """Makes button image colour
    """
    but1['image'] = img1

def change_to_bw(e):
    """Makes button image greyscale
    """
    but1['image'] = img2


#***************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Load images used ***
img1 = PhotoImage(file = 'Triangle.gif')
img2 = PhotoImage(file = 'TriangleBW.gif')

#*** Create label ***
lab1 = ttk.Label(root, text="0")
lab1.pack()

#*** Create button ***
but1 = ttk.Button(root, image = img2,)
but1.pack()

#*** Have button react to mouse pointer ***
but1.bind('<Enter>', change_to_colour)
but1.bind('<Leave>', change_to_bw)

#*** Have window react to mouse movement ***
root.bind('<Motion>', display_coordinates)

#*** Wait for events ***
#***root.mainloop()
```

Activity 4.28

Create a new file called *MouseEvents.py* and implement the code given in FIG-4.15.

Copy the two files *Trinagle.gif* and *TriangleBW.gif* into the program's folder.

Run the program and check that the button's image changes colour.

What happens to the displayed coordinates when the mouse pointer enters the button?

Modify the program so that the mouse coordinates displayed are always relative to the top left-corner of the main window - even when the mouse is within the button. (HINT: Use `root.winfo-x()` and `root.winfo_y()` to retrieve the window's coordinates on the screen.)

Save the program.

Activity 4.29

Create a new file called *ShowingStates.py* and write a program that displays which `state`-recorded keys have been pressed as the mouse moves within the main window.

The pressed key names should be shown as the text of a label. For example, if the **Ctrl** and **Shift** keys were pressed, the display should read **Shift Ctrl**.

Test and save the program.

The program in FIG-4.16 uses the `Place` geometry manager to allow buttons to be dragged to new positions within the window.

FIG-4.16

Dragging Widgets

```
#*** Dragging Widgets  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#***************************************
# ***        Event Handlers          ***
# **************************************
def move_object(e):
    """Reposition dragged widget
    """
    e.widget.place(x = e.x_root-root.winfo_x()-8,
    ↳y = e.y_root-root.winfo_y()-30)

#***************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create 6 buttons with bindings to ***
#*** mouse drag                        ***
for c in range(6):
    lab = ttk.Button(root, text = str(c))
    lab.place(x = 0, y = 0)
    lab.bind('<B1-Motion>', move_object)

#*** Wait for events ***
root.mainloop()
```

The repositioning makes use of the mouse coordinates as measured from the top-left of the screen minus the position of the window on the screen. The window's position is modified slightly to take into account the size of the window's frame.

Activity 4.30

Create a new file called *DraggingWidgets.py* and implement the code shown in FIG-4.16.

Test the program by dragging the buttons to various points on the screen.

Save the program.

Widget Drawing Order

From the last program we can see that widgets may sit over each other with the most recently created widget always being on top. We can change the order in which widgets are drawn on the screen and thereby modify which widget appears on top of an overlapping pile. If we think of the widgets as a deck of playing cards, this adjustment can be likened to moving a card up (or down) within the deck.

All widgets (labels, buttons, etc.) contain two methods which affect that widget's drawing order. A widget which is drawn last will always appear on top of any other widget(s) which it overlaps; a widget which is drawn first will be at the bottom of any overlapping widgets.

lift()	lifts the widget to the top of the 'deck', meaning that the widget is drawn last.
lower()	lowers the widget to the bottom of the 'deck', meaning the widget is drawn first.

Activity 4.31

Modify *DraggingWidgets.py* by adding two new handlers, *lift_widget()* and *lower_widget()* which modify the position of the calling widget. The handlers should be triggered by the right and centre mouse buttons respectively (if you don't have three buttons, use a key/mouse button combination).

Test the program by overlapping buttons and then raising and lowering their drawing order.

Save the program.

Using the Mouse Wheel

The mouse wheel is most often used to scroll a window or widget which is too small for the elements it contains. We'll see how to do this in a later chapter, but for now, we'll make use of the mouse wheel to resize a selected widget.

Activity 4.32

Modify *DraggingWidgets.py,* to detect the mouse wheel event. The associated event-handler, *resize()*, should use the `delta` value to decide if the in-focus button should be increased in width (positive delta value) or decreased in width (negative delta value). The change in width should be one pixel.

Test and save the program.

Summary

- Use the event term `'<Motion>'` to react to the mouse pointer movement.
- Use the event term `'<Enter>'` to react to the mouse pointer moving into a widget space.
- Use the event term `'<Leave>'` to react to the mouse pointer moving out of a widget space.

- Use `'<1>'` to react to the left mouse button being pressed.
- Use `'<2>'` to react to the centre mouse button being pressed.
- Use `'<3>'` to react to the right mouse button being pressed.
- Use the terms `'<Double-1>'` , `'<Double-2>'` or `'<Double-3>'` to react to double clicks on the left, centre, or right mouse buttons.
- Use the terms `'<ButtonRelease-1>'` , `'<ButtonRelease-2>'` or `'<ButtonRelease-3>'` to react to release of the left, centre, or right mouse buttons.
- Use the terms `'<B1-Motion>'` , `'<B2-Motion>'` or `'<B3-Motion>'` to react to release of the mouse being dragged with the left, centre, or right mouse buttons pressed.
- The `Event` class contains the following details for mouse events:

num	{1, 2, 3} Gives the number of the mouse button pressed.
x, y	Gives the mouse pointer coordinates measured from the top-left of the widget.
x_root, y_root	Gives the mouse pointer coordinates measured from the top-left of the screen.
state	an integer value giving the state of any modifier keys. Values are:
delta	the angle through which the mouse wheel has been moved.

Shift	0x00001
Caps Lock	0x00002
Ctrl	0x00004
Num Lock	0x00008
Left mouse button	0x00100
Centre mouse button	0x00200
Right mouse button	0x00400
Alt (on left)	0x20000
Alt (on right)	0x20004

- A widget's drawing order determines if it appears 'above' or 'below' other, overlapping widgets.
- Use `lift()` to promote a widget's drawing order.
- Use `lower()` to demote a widget's drawing order.

Widget Events

Introduction

Some events are triggered by the user resizing, moving a widget, or changing the stacking order.

Event Names

'<Configure>'	this event occurs if a widget is moved or has a property changed.

Activity 4.33

Modify *DraggingWidgets.py* by adding a label. The label should take on the value *"Reconfigured"* plus the text within the widget when the `Configure` event occurs.

Test the program by moving and resizing the buttons.

Save the program.

'<Expose>'	this event occurs when the visibility of a widget changes. For example, if a change in stacking order results in more or less of the widget being displayed.

Activity 4.34

Modify *DraggingWidgets.py* so that the label takes on the value *"Exposure "* plus the text within the widget when the `Expose` event occurs.

Test and save the program.

'<FocusIn>'	this event occurs when a widget gains focus.

Activity 4.35

Modify *DraggingWidgets.py* so that the label takes on the value *"Focus in : "* plus the text within the widget when the `FocusIn` event occurs.

Test and save the program.

'<FocusOut>'	this event occurs when a widget has just lost focus.

Activity 4.36

Modify *DraggingWidgets.py* so that the label takes on the value *"Focus out : "* plus the text within the widget when the `FocusOut` event occurs.

Test and save the program.

We can see from the last two Activities that these two events can be used to tell us which widget has just gained focus and which has just lost it.

'<Destroy>' this event occurs as a widget is destroyed.

The program in FIG-4.17 contains a label and a button. Pressing the button, deletes the label. This, in turn, displays a message box to say that the label has been deleted.

FIG-4.17

Deleting Widgets

```
#*** Destroy  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk
from tkinter import messagebox


#***************************************
# ***        Event Handlers          ***
# **************************************
def delete_label():
    """ Deletes label
    """
    lab1.destroy()


def destroy_alert(e):
    """ Displays message box when label is deleted
    """
    messagebox.showinfo(message = "Label has been deleted")


#***************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label ***
lab1 = ttk.Label(root, text = "Here today")
lab1.pack()
#*** Let label detect its own destruction ***
lab1.bind('<Destroy>', destroy_alert)

#*** Create button to delete label ***
but1 = ttk.Button(root, text = "Destroy Label", command =
↳delete_label)
but1.pack()

#*** Wait for events ***
root.mainloop()
```

Activity 4.37

Create a new file called *DeletingWidgets.py* which contains the code given in FIG-4.17.

Test and save the program.

Event Types

One of the attributes of the `Event` class is `type` which specifies the type of event that has just occurred. Each code and its meaning is listed in FIG-4.18.

FIG-4.18

Event Codes

Code	Event
2	KeyPress
3	KeyRelease
4	ButtonPress
5	ButtonRelease
6	Motion
7	Enter
8	Leave
9	FocusIn
10	FocusOut
12	Expose
15	Visibility
17	Destroy
18	Unmap
19	Map
21	Reparent
22	Configure
24	Gravity
26	Circulate
28	Property
32	ColorMap
36	Activate
37	Deactivate
38	Mousewheel

Note: Although the type attribute contains only numeric values, it is stored as a string.

Activity 4.38

Modify *DraggingWidgets.py* so that `'<Expose>'`, `'<Configure>'` and `'<FocusIn>'` events are handled as before but using only a single function for all three events. Remove the code for the `'<FocusOut>'` event.

Test and save the program.

Protocols

Tkinter can also bind to events within the operating system's window manager. This is known as protocol handler. There are a set of protocol names, the most common of which is the string `"WM_DELETE_WINDOW"` which occurs when a window is being closed.

Rather than use one of the bind methods to link a protocol to a handler, we use the widget's `protocol()` method:

protocol(name, function)
This method causes *function* to be executed when protocol *name* is flagged by the operating system.

The program in FIG-4.19 displays a dialog box to allow the user to cancel the closure of the main window.

FIG-4.19

Closing a Window

```
#*** Close Window Second Chance  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk
from tkinter import messagebox


#****************************************
# ***        Event Handlers          ***
# ***************************************
def last_chance():
    """ Gives option to cancel window closure
    """
    if messagebox.askokcancel("Quit", "Are you sure you want to
    ↳quit?"):
        root.destroy()

#****************************************
# ***          GUI Layout            ***
# ***************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Link window closure to second chance option ***
root.protocol("WM_DELETE_WINDOW", last_chance)


#*** Wait for events ***
root.mainloop()
```

Activity 4.39

Create a new file called *Closing.py* which contains the code given in FIG-4.19.

Test and save the program.

Summary

- An event can be triggered by a widget being moved, resized, or redrawn.
- The '<Configure>' event occurs when a widget is moved or has a property value changed.

- The `'<Expose>'` event occurs when the visibility of a widget changes.
- The `'<FocusIn>'` event occurs when a widget gains focus.
- The '<FocusOut>' event occurs when a widget loses focus.
- The '<Destroy>' event occurs when a widget is deleted.
- Use the `Event` class's `type` property to discover what type of event has occurred.
- Use a widget's `protocol()` method to link a handler to an operating system's event.

Timer Events

Introduction

The final group of events are triggered by the passage of time. A timed event can be activated at a specific point in time after the program has started or at set time intervals. Every widget class (including the main window) has a set of methods which set up timed events. Timed events are often known as **alarms**.

Timed Methods

after(delay, function[, *paras])

The `after()` method calls a function (*function*) after *delay* milliseconds. If the function being called requires parameters, these can be included in the parameter list after the function name. A typical call to this method could be

```
root.after(1000,change_label,"One second")
```

which states that the function *change_label()* is to be executed after 1 second.

The program in FIG-4.20 changes the contents of a label to "Three seconds" after three seconds have passed.

FIG-4.20

A Times Event

```
#*** A Timed Event ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***         Event Handlers         ***
# **************************************
def change_label(txt):
    """Changes label's text
    """
    lab1['text'] = txt

#****************************************
# ***           GUI Layout           ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Add a label ***
lab1 = ttk.Label(root, text = "Waiting")
lab1.pack()

#*** After 3 seconds, change label's text ***
root.after(3000, change_label, "Three seconds")

#*** Wait for events ***
root.mainloop()
```

Notice that the event-handler does not take an `Event` object parameter.

Activity 4.40

Start a new project *Timing.py* and enter the code given in FIG-4.20.

Test and save your program.

As we can see from Activity 4.40, the event occurs only once after the specified time has elapsed. If we want a timed event to occur at regular intervals, then we need to place another call to `root.after()` inside the event handler.

In the next event handler, the label's contents have a plus symbol (+) added every two seconds:

```
def change_label():
   """Adds + symbol to label's string every 2 secs
   """
   lab1['text'] += "+"
   #*** Reschedule event ***
   root.after(2000, change_label)
```

Activity 4.41

Modify *Timing.py* so that the label begins with an empty string and has a plus symbol added every two seconds.

Run the program. Do the string's contents grow longer over time? When does the string stop getting longer?

Save your program.

The `after()` method returns a type of reference to the last event (called an **alarm id**). We can see this if we include a `print()` call in the last part of the handler, changing the final line to

```
print(root.after(2000, change_label))
```

Activity 4.42

Modify *Timing.py* so that the last line of the event handler displays the value returned by the call to `root.after()`.

Run the program and observe the values returned.

Save your program.

after_cancel(event_id)

Our last program would run on forever, continually adding plus characters to the end of the string. However. if we want to eventually disable the timed event, we can do so by executing the object's `after_cancel()` method.

To do this we must save the value returned by the call to `after()` which is required as the parameter to `after_cancel()`.

In FIG-4.21 we have a label showing the time in minutes and seconds. The label's text continues to update until the button is pressed; timing then stops and the time's text turns red.

FIG-4.21

Stopping a Repeating Timed Event

```
# *** Stopping a Timed Event ***

# *** Import Modules ***
from tkinter import *
from tkinter import ttk

# *** global variables ***
time        = 0         # Holds elapsed time
lab_timer   = None      # Timed event id

#*************************************
# ***        Event Handlers         ***
# ************************************
def change_time():
    """Changes the text within the label to show time
    """
    global time, lab_timer

    #*** Add 1 sec to time ***
    time +=1
    #*** Format the label's text ***
    lab1["text"] = "{}:{}{}".format(time // 60, time % 60 // 10,
    ↳time % 60 %10)
    #*** Make another call to timed event saving the ID ***
    lab_timer = lab1.after(1000, change_time)

#*** Event handler for Button pressed ***
def stop_time():
    """Stops timed event handler when button pressed
    """
    global lab_timer

    #*** Cancel label's timer event (ID as parameter) ***
    lab1.after_cancel(lab_timer)
    #*** Change time label text colour to red ***
    lab1['foreground']="red"

# *************************************
# ***          GUI Layout             ***
# *************************************
#*** Create window ***
wnd = Tk()
wnd.geometry("300x120+200+100")

#*** Add a label ***
lab1 = ttk.Label(wnd, text="0:00")
lab1.pack()

#*** Add a button ***
but1 = ttk.Button(wnd, text = "Stop", command = stop_time)
but1.pack()

#*** After 1 second update label ***
lab1.after(1000, change_time)

#*** Wait for events ***
wnd.mainloop()
```

Activity 4.43

Start a new file *StopTime.py* and enter the code given in FIG-4.21

Test and save your project.

Summary

- An event can be triggered by the passage of time.
- Use `after()` to execute an event handler after a given amount of time has passed.
- To have a time-related handler execute repeatedly, the handler itself must make another call to `after()`.
- All event handlers return a reference to the last event. This is known as the **alarm id**.
- Use `after_cancel()` to stop a repeating timed event.

Virtual Events

Introduction

When we make use of the `command` option in a `Button` widget to link a handler to the pressing of that button, we are, in fact, binding the handler to either the clicking of the mouse button when the pointer is over the button, or the pressing of the space bar when the button has focus.

If we want a widget to react to two or more normal events, each event executing the same handler, then it may be worth the effort to create a **virtual event**.

Virtual Event Methods

event_add()

To create a virtual event we must call a widget's `event_add()` method, giving the name of our newly created event. That name must be enclosed in double angled brackets. For example, let's say we want to create a virtual event for a `Button` widget called *but1*. The virtual event is to be called *BPress* and is triggered when the left mouse button is pressed, or the pressing of the **Return** key (when the widget has focus). To set this up, we would use the line

```
root.event_add('<<BPress>>','<Button-1>', '<KeyPress Return>')
```

After it is created, the virtual event can be linked to any widget in the program using one of the standard bind methods:

```
but1.bind('<<BPress>>', press_handler)
```

The program in FIG-4.22 creates a label and three buttons with the buttons responding to the *BPress* virtual event as defined above. Pressing a button changes the colour of the label's text.

FIG-4.22

Using a Virtual Event

```
#*** Using a Virtual Event ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#****************************************
# ***         Event Handlers          ***
# ***************************************
def change_colour(e):
    '''Changes the text colour of lab1 to that
       given as text on button, e
    '''
    lab1['foreground'] = e.widget.cget('text')
#****************************************
# ***           GUI Layout            ***
# ***************************************
#*** Create window ***
root = Tk()
root.geometry('320x180+500+500')

#*** Create a Virtual event ***
#*** for mouse button 1 or return key ***
```

FIG-4.22
(continued)

Using a Virtual Event

```
root.event_add('<<BPress>>', '<Button-1>', '<KeyPress Return>')

#*** Create label ***
lab1 = ttk.Label(root, text = 'Changes colour')
lab1.pack(side = 'top')

# Create three buttons all linked to the virtual event ***
but1 = ttk.Button(root, text = 'red')
but1.pack(side = 'left')
but1.bind('<<BPress>>', change_colour)

but2 = ttk.Button(root, text = 'green')
but2.pack(side = 'left')
but2.bind('<<BPress>>', change_colour)

but3 = ttk.Button(root, text = 'blue')
but3.pack(side = 'left')
but3.bind('<<BPress>>', change_colour)


#*** Wait for events ***
root.mainloop()
```

Activity 4.44

Start a new file *VirtualEvent.py* and enter the code given in FIG-4.22.

Test and save your project.

event_delete()

If we wish to remove one or more of the native events from a virtual event, then we can use the `event_delete()` method. For example, to delete the mouse button option from the virtual event *BPress*, we would use the line

```
but1.event_delete('<<BPress>>','<Button-1>')
```

If all native events are removed from a virtual event, then that event is no longer triggered.

Activity 4.45

Add a new button to *VirtualEvent.py*. This button, when pressed (use the normal `command` option, not *Bpress*), should remove the **Return** key option from the *Bpress* virtual event. Call the new handler *remove_return()*.

Test and save your project.

event_info()

To find out what virtual events have been defined within a program, or what native events are currently linked to a virtual event, use `event_info()`.

When used without an argument, this method returns a list of all virtual events that have been defined.

When the name of a virtual event is supplied as a parameter, the native events currently linked to that virtual event are returned. If the virtual event does not exist, `None` is returned.

Activity 4.46

In the handler created for the fourth button in *VirtualEvent.py*, add instructions to display in the console window the native events linked to *BPress* before and after the removal of the **Return** key press.

Test and save your project.

Summary

- A virtual event links a identifying name to a set of native events.
- Virtual event names must be enclosed in double angled brackets.
- Use a widget's `event_add()` method to create a virtual event.
- Once created a virtual event can be linked to any widget using the standard bind methods.
- Use the `event_delete()` method to remove a native event from those events associated with a named virtual event.
- Without parameters, use `event_info()` to list all virtual events defined within a program.
- With a parameter identifying an existing virtual event, use `event_info()` to retrieve the native events currently linked to that virtual event.

Forcing an Event

event_generate()

We don't have to wait for an event to occur to have it's handler executed. Instead we can call the method `event_generate()` which fools the program into thinking that a specified event has occurred.

The method requires the name of the event that is to be triggered as its first argument.

Let's say we want the *remove_return()* handler in *VirtualEvent.py* to also set the colour of the text to blue, we could do this by forcing a call to the event *BPress* for *but3* within the handler using the line

```
but3.event_generate('<<BPress>>')
```

Activity 4.47

In the handler created for the fourth button in *VirtualEvent.py*, remove the `print()` statements then add the line given above at the end of the function.

What happens when the fourth button is pressed?

Save your project.

We can also set some of the attributes of the `Event` object that is passed to an event handler linked to a bind statement. For example, let's assume that, in *VirtualEvent.py*, we have changed the *change_colour()* handler so that the label's text displays the time at which the event occurred. The new code would be

```
def change_colour(e):
    """Changes the text colour of lab1 to that
       given as text on button, e
    """
    lab1['foreground'] = e.widget.cget('text')
    lab1['text'] = e.time
```

Activity 4.48

In *VirtualEvent.py*, modify *change_colour()* to match the code given above.

Run the program and observe the text displayed when each of the four buttons is pressed.

What value is displayed when the fourth button is pressed?

Save your project.

We can set a `time` value when we make the call to `event_generate()`:

```
but3.event_generate('<<BPress>>', time = 12)
```

This value for `time` will now be passed to *change_colour()* within the `e` parameter.

Activity 4.49

In *VirtualEvent.py*, modify the call to `event_generate()` so that the `time` value is set to 140000.

Run the program and observe the text displayed when each of the four buttons is pressed.

What value is displayed when the fourth button is pressed?

Save your project.

Summary

- Use a widget's `event_generate()` method to trigger an event to which the widget is already bound.
- The `event_generate()` method can be given additional parameters to set various aspects of the Event object being passed to the event handler.

Solutions

Activity 4.1

When you first press the **b** key nothing happens (unless you have previously given the button focus).

After the button has focus, pressing the **b** key, causes the text within the label to change.

Modified code for *KeyboardBinding.py*:

```
#****** Keyboard Events ******

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#***************************************
# ***        Event Handlers          ***
# *************************************

def change_label():
    """Changes the label contents
    """
    lab1['text'] = "'a' key pressed"

#***************************************
# ***          GUI Layout            ***
# *************************************
#*** Create window ***
root = Tk()

#*** Add Label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()
#*** Add button ***
but1 = ttk.Button(root, text = "Give Focus")
but1.pack()

#*** Make button react to a key being pressed ***
but1.bind('a',change_label)


#*** Wait for events ***
root.mainloop()
```

Now the program reacts to the **a** key being pressed rather than the **b** key.

Activity 4.2

To have the event linked to a capital A, we need only change the line

```
but1.bind('a',change_label)
```

to

```
but1.bind('A',change_label)
```

Activity 4.3

If we change the call to `bind()` to read

```
but1.bind('ab',change_label)
```

then we need to press the **a** key followed by the **b** key to have the event take place. Pressing **b** then **a** will not trigger the event.

Activity 4.4

Now the `bind()` calls need to be

```
but1.bind('a',change_label)
but1.bind('b',change_label)
```

With this code, pressing either *a* or *b* will trigger the event.

Activity 4.5

Modified code for *KeyboardBinding.py*:

```
#****** Keyboard Events ******

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#***************************************
# ***        Event Handlers          ***
# *************************************
def change_label(e):
    """Changes the label contents
    """
    lab1['text'] = "'a' key pressed"


def change_screen(e):
    """Changes the background colour of the main
       window to red
    """
    root['background'] = 'red'

#***************************************
# ***          GUI Layout            ***
# *************************************
#*** Create window ***
root = Tk()

#*** Add Label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()
#*** Add button ***
but1 = ttk.Button(root, text = "Give Focus")
but1.pack()

#*** Make key a change label and window colour ***
but1.bind('a', change_label)
but1.bind('a', change_screen, '+')


#*** Wait for events ***
root.mainloop()
```

Without the + symbol in the last `bind()` call, the new event handler for **a** replaces the earlier one, meaning that the **a** key changes only the screen colour but not the label contents.

With the + symbol added, both event handlers are run when the **a** key is pressed.

Activity 4.6

Modified code for *KeyboardBinding.py* :

```
#****** Keyboard Events ******

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#***************************************
# ***        Event Handlers          ***
# *************************************
def change_label(e):
    """Changes the label contents
    """
    lab1['text'] = "'a' key pressed"


def change_screen(e):
    """Changes the background colour of the main
       window to red
    """
    root['background'] = 'red'

#***************************************
# ***          GUI Layout            ***
# *************************************
#*** Create window ***
root = Tk()

#*** Add Label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()
#*** Add button ***
```

```
but1 = ttk.Button(root, text = "Give Focus")
but1.pack()

#*** Make key 'a' change label text and key 'b' the
↳window colour ***
but1.bind('a', change_label)
but1.bind('b', change_screen)

#*** Display events to which but1 is bound ***
print(but1.bind())


#*** Wait for events ***
root.mainloop()
```

The program displays ('b', 'a').

Activity 4.7

Modified code for *KeyboardBinding.py*:

```
#****** Keyboard Events ******

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***        Event Handlers           ***
# **************************************
def change_label(e):
    """Changes the label contents
    """
    lab1['text'] = "A key pressed"


def change_screen(e):
    """Changes the background colour of the main
       window to red
    """
    root['background'] = 'red'

#****************************************
# ***          GUI Layout             ***
# **************************************
#*** Create window ***
root = Tk()

#*** Add Label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()
#*** Add button ***
but1 = ttk.Button(root, text = "Give Focus")
but1.pack()

#*** Make key a change label ***
#*** and Delete key change window colour ***
but1.bind('a', change_label)
but1.bind('<Delete>', change_screen)


#*** Wait for events ***
root.mainloop()
```

Activity 4.8

Binding code should be changed to

```
#*** Make key a change label ***
#*** and any key change window colour ***
but1.bind('a', change_label)
but1.bind('<Key>', change_screen)
```

Using `'<Key>'` binds all keys - except those previously assigned to this object. So pressing **a** changes the label's contents, not the window background colour. Every other key will change the colour.

Activity 4.9

Modified code for *KeyboardBinding.py*:

```
#****** Keyboard Events ******

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***        Event Handlers           ***
# **************************************
def change_label(e):
    """Changes the label contents
    """
    lab1['text'] = "A key pressed"


def change_screen(e):
    """Changes the background colour of the main
       window to red
    """
    root['background'] = 'red'


def change_screen_white(e):
    """Changes the background colour of the main
       window to white
    """
    root['background'] = 'white'

#****************************************
# ***          GUI Layout             ***
# **************************************
#*** Create window ***
root = Tk()

#*** Add Label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()
#*** Add button ***
but1 = ttk.Button(root, text = "Give Focus")
but1.pack()

#*** Make key a change label ***
#*** and any key change window colour ***
but1.bind('a', change_label)
but1.bind('<Key>', change_screen)
but1.bind('<KeyRelease Delete>',
↳change_screen_white)


#*** Wait for events ***
root.mainloop()
```

Activity 4.10

In *KeyboardBinding.py*, change the line

```
but1.bind('<Key>', change_screen)
```

to

```
but1.bind('<Alt-z>', change_screen)
```

Activity 4.11

No solution required.

Activity 4.12

Code for *Disabling.py*:

```
#*** Using Attributes from the Event class ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***        Event Handlers           ***
# **************************************
def disable(e):
    """Disables the calling widget
    """
    e.widget['state']= 'disabled'

#****************************************
# ***          GUI Layout             ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create six buttons ***
for c in range(6):
    temp = ttk.Button(root, text = c)
    temp.pack()
    temp.bind('<Button-1>',disable)

#*** Wait for events ***
root.mainloop()
```

Activity 4.13

In *KeyboardBinding.py*, have the window colour change to red irrespective of the button having focus, change the line

```
but1.bind('<Alt-z>', change_screen)
```

to

```
but1.bind_all('<Alt-z>', change_screen)
```

Activity 4.14

Only the second button gains focus when clicked on.

All widgets can gain focus using the **Tab** key.

Data cannot be entered in the textbox.

The event handler executes when the **Alt-z** keys are pressed and either the first button or textbox have focus.

Activity 4.15

Code for *UnderstnadingBindtags.py*:

```
#*** Binding Groupings ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#***************************************
# ***          GUI Layout           ***
# *************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Add ungrouped Label ***
lab1 = ttk.Label(root, text="Awaiting an event ...")
lab1.pack()

#*** Add TestGroup button ***
but1 = ttk.Button(root, text = 'but1')
but1.pack()

#*** Add ungrouped button ***
but2 = ttk.Button(root, text = 'but2')
but2.pack()

#*** Add TestGroup text box ***
ent1 = ttk.Entry(root, text = 'ent1')
ent1.pack()

#*** Display bindtags for window and each widget ***
print ("Bindtags for root : ",root.bindtags())
print ("Bindtags for lab1 : ",lab1.bindtags())
print ("Bindtags for but1 : ",but1.bindtags())
print ("Bindtags for but2 : ",but2.bindtags())
print ("Bindtags for ent1 : ",ent1.bindtags())

#*** Wait for events ***
root.mainloop()
```

This produces the following results (the first value in your lists may be different).

```
Bindtags for root :  ('.', 'Tk', 'all')
Bindtags for lab1 :  ('.6795664', 'TLabel', '.', 'all')
Bindtags for but1 :  ('.36396720', 'TestGroup', '.', 'all')
Bindtags for but2 :  ('.44649360', 'TButton', '.', 'all')
Bindtags for ent1 :  ('.44649392', 'TestGroup', '.', 'all')
```

Activity 4.16

Modified code for *UnderstnadingBindtags.py*:

```
#*** Binding Groupings ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#***************************************
# ***        Event Handlers         ***
# *************************************
def show_name(e):
    """Displays widget name in label
    """
    lab1['text'] = e.widget['text']

#***************************************
# ***          GUI Layout           ***
# *************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Add ungrouped Label ***
lab1 = ttk.Label(root, text="Awaiting an event ...")
lab1.pack()

#*** Add TestGroup button ***
but1 = ttk.Button(root, text = 'but1')
but1.pack()

#*** Add ungrouped button ***
but2 = ttk.Button(root, text = 'but2')
but2.pack()

#*** Add TestGroup text box ***
ent1 = ttk.Entry(root, text = 'ent1')
ent1.pack()

#*** Add event to first button only ***
but1.bind('<Alt-z>', show_name)

#*** Wait for events ***
root.mainloop()
```

The **Alt-z** combination if only effective when the first button (*but1*) has focus.

Activity 4.17

The only change to the program is that the line

```
but1.bind('<Alt-z>', show_name)
```

is changed to

```
but1.bind_class('TButton','<Alt-z>', show_name)
```

Now both buttons (which are members of the `TButton` class) react to the **Alt-z** key press by changing the contents of the label.

The text box does not react to **Alt-z**.

Activity 4.18

Modified code for *UnderstnadingBindtags.py*:

```
#*** Binding Groupings ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#***************************************
# ***        Event Handlers         ***
# *************************************
def show_name(e):
    """ Displays widget name in label
    """
    if e.widget == root:
        lab1['text'] = "root"
    else:
        lab1['text'] = e.widget['text']

#***************************************
# ***          GUI Layout           ***
# *************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')
```

```
#*** Add ungrouped Label ***
lab1 = ttk.Label(root, text="Awaiting an event ...")
lab1.pack()

#*** Add TestGroup button ***
but1 = ttk.Button(root, text = 'but1')
but1.pack()

#*** Add ungrouped button ***
but2 = ttk.Button(root, text = 'but2')
but2.pack()

#*** Add TestGroup text box ***
ent1 = ttk.Entry(root, text = 'ent1')
ent1.pack()

#*** Add event to all widgets ***
but1.bind_all('<Alt-z>', show_name)

#*** Wait for events ***
root.mainloop()
```

Now all widgets (excluding the label, which cannot take focus) respond to the **Alt-z** keys.

Activity 4.19

To link the **Alt-z** event to the main window we need to change the line

```
but1.bind_all('<Alt-z>', show_name)
```

to

```
but1.bind_class('.','<Alt-z>', show_name)
```

Since all the widgets are children of the main window (and have '.' in their bindtags list) they still respond to **Alt-z**.

Activity 4.20

When the first button has focus, every time Alt-z is pressed, the count displayed in the label is incremented.

Activity 4.21

The first modification means that the bind statements should now read

```
#*** Add event to first button and TButton class ***
but1.bind('<Alt-z>', add_to_count)
but1.bind_class('TButton', '<Alt-z>', add_to_count)
```

The event is now linked to two entries in *but1*'s bindtags list: the widget's name and `TButton`, so when that button has focus, and **Alt-z** is pressed, the event handler is executed twice - once for each link.

The second button, *but2*, links the event only to `TButton`, so when it has focus, pressing **Alt-z** causes the event handler to be executed only once.

The second modification adds another line to the bind statements:

```
#*** Add event to first button, TButton class and the
main window ***
but1.bind('<Alt-z>', add_to_count)
but1.bind_class('TButton', '<Alt-z>', add_to_count)
but1.bind_class('.', '<Alt-z>', add_to_count)
```

Now the first button has three entries in its bindtags list that are linked to the event **Alt-z**, the second button has two, the text box one, and the main window one.

The count is incremented by the number of links the in-focus widget has to **Alt-z**.

Activity 4.22

Although the handler copies the contents of the text box to the label, the label is always one character 'behind' the text box.

To display the text box's bindtags list we require the code

```
#*** Display the text box's bindtags list ***
print(ent1.bindtags())
```

This displays a list similar to

('.36057008', 'TEntry', '.', 'all')

Activity 4.23

Modified code for *TagOrder.py*:

```
#*** Bindtag Order ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#**************************************
# ***        Event Handlers         ***
# *************************************
def change_label(e):
    """Sets label to contents of text box
    """
    lab1['text'] = ent1.get()

#**************************************
# ***          GUI Layout           ***
# *************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.pack()

#*** Create text box ***
ent1 = ttk.Entry(root)
ent1.pack()
#*** Reorder bindtags ***
ent1.bindtags(('TEntry', ent1, '.', 'all'))

#*** Bind text box to any key press ***
ent1.bind('<Key>', change_label)

#*** Display the text box's bindtags list ***
print(ent1.bindtags())

#*** Wait for events ***
root.mainloop()
```

When the program is run this time the contents of the label are an exact match for that of the text box.

In the console window we can see the new order of the bindtags list.

Activity 4.24

Modified code for *GroupBindings.py*:

```
#*** Binding Groupings ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#**************************************
# ***        Event Handlers         ***
# *************************************
def change_label(e):
    """Changes the label contents
    """
    lab1['text'] = "Changed by a TestGroup widget"

#**************************************
# ***          GUI Layout           ***
# *************************************
```

```
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Add ungrouped Label ***
lab1 = ttk.Label(root, text="Awaiting an event ...")
lab1.pack()

#*** Add TestGroup button ***
but1 = ttk.Button(root, text = "TG button")
but1.pack()
#*** Add TestGroup to bindtags list ***
but1.bindtags((but1, 'TButton', '.', 'all',
↳'TestGroup'))

#*** Add ungrouped button ***
but2 = ttk.Button(root, text = "Non-TG Button")
but2.pack()

#*** Add TestGroup text box ***
ent1 = ttk.Entry(root)
ent1.pack()
#*** Add TestGroup to bindtags list ***
ent1.bindtags((ent1, 'TEntry', '.', 'all',
↳'TestGroup'))

#*** Change label if Alt-z pressed when a ***
#*** TestGroup widget has focus           ***
but1.bind_class('TestGroup','<Alt-z>', change_label)

#*** Wait for events ***
root.mainloop()
```

The button and text box now operate normally but also react to the **Alt-z** event.

Activity 4.25

The program correctly displays a copy of each numeric key pressed and an error message if any non-numeric character is entered.

Activity 4.26

The handler should now be coded as:

```
def show_key_pressed(e):
    """Label shows last numeric key pressed or error
      message
    """
    #*** If numeric display it in label ***
    if e.char >= '0' and e.char <= '9':
        lab1['text'] = e.char
    else: # not numeric
        #*** Display error message ***
        lab1['text'] = "Not numeric"
        return 'break'
```

No non-numeric characters appear within the text box.

Activity 4.27

No solution required.

Activity 4.28

Mouse coordinates are relative to the top-left corner of the widget in which it is moving. Initially, this is the main window, but when the mouse pointer moves over the button, the coordinates are from the top-left corner of that button and so the numbers, become much smaller.

To have the coordinates always measured from the top-left of the main window, irrespective of the mouse's position we need to use the screen coordinates (from *x_root* and *y_root*) minus the window's coordinates (returned by `root.winfo_x()` and `root.winfo_y()`).

The `display_coordinates()` handler's code becomes:

```
lab1['text'] = '('+str(e.x_root-root.winfo_x())
↳+','+str(e.y_root-root.winfo_y())+')'
```

This gives us the coordinates as measured from outside the window's borders. To produce coordinates where (0,0) is the top-left corner within the window borders, the we need to subtract 8 pixels from the *x* value and 30 from the *y* value (as measured using Windows 7).

Activity 4.29

Code for *ShowingStates.py*:

```
#*** Mouse Key State Events ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#***************************************
# ***        Event Handlers          ***
# *************************************
def display_states(e):
    """ Displays the keyboard button pressed when
        a mouse motion event occurs
    """
    #*** Create an empty string ***
    txt = ""
    #*** Add text for any key found ***
    if e.state & 0x00001:
        txt = txt + "Shift"
    if e.state & 0x00002:
        txt = txt + " Caps Lock"
    if e.state & 0x00004:
        txt = txt + " Ctrl"
    if e.state & 0x00008:
        txt = txt + " Num Lock"
    if e.state & 0x00100:
        txt = txt + " Left Mouse"
    if e.state & 0x00200:
        txt = txt + " Centre Mouse"
    if e.state & 0x00400:
        txt = txt + " Right Mouse"
    if e.state & 0x20000:
        txt = txt + "  Alt"
    #*** Copy final text to label ***
    lab1['text'] = txt

#***************************************
# ***          GUI Layout            ***
# *************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label ***
lab1 = ttk.Label(root, text="Waiting...")
lab1.pack()

#*** Have window react to mouse movement ***
root.bind('<Motion>', display_states)

#*** Wait for events ***
root.mainloop()
```

Activity 4.30

Widgets can be dragged to any position within the window. They can even overlap each other.

Activity 4.31

Modified code for *DraggingWidgets.py*:

```
#*** Dragging Widgets  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#***************************************
# ***        Event Handlers          ***
# *************************************
def move_object(e):
    """Reposition dragged widget
    """
    e.widget.place(x = e.x_root-root.winfo_x()-8,
    ↳y = e.y_root-root.winfo_y()-30)

def lift_widget(e):
   """ Lift widget to top level
   """
    e.widget.lift()

def lower_widget(e):
    """Lower widget to bottom level
    """
    e.widget.lower()

#****************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create 6 buttons with bindings to       ***
#*** mouse drag and centre and right buttons ***
for c in range(6):
    lab = ttk.Button(root, text = str(c),width = 10)
    lab.place(x = 0, y = 0)
    lab.bind('<B1-Motion>', move_object)
    lab.bind('<Button-3>', lift_widget)
    lab.bind('<Button-2>', lower_widget)

#*** Wait for events ***
root.mainloop()
```

Activity 4.32

Modified code for *DraggingWidgets.py*:

```
#*** Dragging Widgets  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#****************************************
# ***        Event Handlers          ***
# **************************************
def move_object(e):
    """Reposition dragged widget
    """
    e.widget.place(x = e.x_root-root.winfo_x()-8,
    y = e.y_root-root.winfo_y()-30)

def lift_widget(e):
    """ Lift widget to top level
    """
    e.widget.lift()

def lower_widget(e):
    """Lowers widget to bottom level
    """
    e.widget.lower()

def resize(e):
    """Resizes width of widget
    """
    if e.delta > 0:
        e.widget['width'] += 1
    else:
        e.widget['width'] -= 1


#****************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create 6 buttons with bindings to ***
#*** mouse drag                      ***
for c in range(6):
    lab = ttk.Button(root, text = str(c),width = 10)
    lab.place(x = 0, y = 0)
    lab.bind('<B1-Motion>', move_object)
    lab.bind('<3>', lift_widget)
    lab.bind('<2>', lower_widget)
    lab.bind('<MouseWheel>', resize)

#*** Wait for events ***
root.mainloop()
```

Activity 4.33

Modified code for *DraggingWidgets.py*:

```
#*** Dragging Widgets  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#****************************************
# ***        Event Handlers          ***
# **************************************
def move_object(e):
    """Reposition dragged widget
    """
    e.widget.place(x = e.x_root-root.winfo_x()-8,
    ↳y = e.y_root-root.winfo_y()-30)

def lift_widget(e):
    """ Lift widget to top level
    """
    e.widget.lift()

def lower_widget(e):
    """Lowers widget to bottom level
    """
    e.widget.lower()

def resize(e):
    """Resizes width of widget
    """
    if e.delta > 0:
        e.widget['width'] += 1
    else:
        e.widget['width'] -= 1

def reconfig(e):
    """ Changes label to show 'Reconfigured'
    """
    lab1['text'] = "Reconfigured " + e.widget['text']

#****************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.place(x = 120, y = 10)

#*** Create 6 buttons with bindings to ***
#*** mouse drag, mouse buttons, wheel and size/
position ***
for c in range(6):
    but = ttk.Button(root, text = str(c),width = 10)
    but.place(x = 0, y = 0)
    but.bind('<B1-Motion>', move_object)
    but.bind('<3>', lift_widget)
    but.bind('<2>', lower_widget)
    but.bind('<MouseWheel>', resize)
    but.bind('<Configure>', reconfig )


#*** Wait for events ***
root.mainloop()
```

The label should change when a widget is moved or resized.

Activity 4.34

The modifications require a new handler:

```
def exposed(e):
    """ Changes label to show 'Exposed'
    """
    lab1['text'] = "Exposed " + e.widget['text']
```

and the following line added within the `for` loop structure:

```
but.bind('<Expose>', exposed)
```

The label should show this option only when the widget needs to be redrawn.

Activity 4.35

The modifications require a new handler:

```
def focus(e):
    """ Changes label to show 'FocusIn'
    """
    lab1['text'] = "FocusIn " + e.widget['text']]
```

and the following line added within the `for` loop structure:

```
but.bind('<FocusIn>', focus)
```

The label should show this option only when the widget gains focus.

Activity 4.36

The modifications require a new handler:

```
def defocus(e):
    """ Changes label to show 'FocusOut'
    """
    lab1['text'] = "FocusOut " + e.widget['text']
```

and the following line added within the `for` loop structure:

```
but.bind('<FocusOut>', defocus)
```

To have the label show this option, you need to comment out the `but.bind('<FocusIn>' focus)` statement.

Activity 4.37

No solution required.

Activity 4.38

Modified code for *DraggingWidgets.py*:

```
#*** Dragging Widgets  ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#****************************************
# ***        Event Handlers           ***
# **************************************
def move_object(e):
    """Reposition dragged widget
    """
    e.widget.place(x = e.x_root-root.winfo_x()-8,
    ↳y = e.y_root-root.winfo_y()-30)

def lift_widget(e):
    """ Lift widget to top level
    """
    e.widget.lift()

def lower_widget(e):
    """Lowers widget to bottom level
    """
    e.widget.lower()

def resize(e):
    """Resizes width of widget
    """
    if e.delta > 0:
        e.widget['width'] += 1
    else:
        e.widget['width'] -= 1

def reconfig(e):
    """ Changes label to show as appropriate
    """
    if e.type == '22':
        lab1['text'] = "Reconfigured " +
        ↳e.widget['text']
    elif e.type == '12':
        lab1['text'] = "Exposed " + e.widget['text']
    elif e.type == '9':
        lab1['text'] = "FocusIn " + e.widget['text']
    else:
        lab1['text'] = e.type
#****************************************
# ***          GUI Layout             ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Create label ***
lab1 = ttk.Label(root, text = "Waiting...")
lab1.place(x = 120, y = 10)

#*** Create 6 buttons with bindings to ***
#*** mouse drag, mouse buttons, wheel and size/
position ***
for c in range(6):
    but = ttk.Button(root, text = str(c), width =
10)
    but.place(x = 0, y = 0)
    but.bind('<B1-Motion>', move_object)
    but.bind('<3>', lift_widget)
    but.bind('<2>', lower_widget)
    but.bind('<MouseWheel>', resize)
    but.bind('<Configure>', reconfig )
    but.bind('<Expose>', reconfig)
    but.bind('<FocusIn>', reconfig)


#*** Wait for events ***
root.mainloop()
```

Activity 4.39

No solution required.

Activity 4.40

No solution required.

Activity 4.41

Modified code for *Timing.py*:

```
#*** A Timed Event ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk

#****************************************
# ***        Event Handlers           ***
# **************************************
def change_label():
    """Adds + symbol to label's string every 2 secs
    """
    lab1['text'] += "+"
    #*** Reschedule event ***
    root.after(2000, change_label)

#****************************************
# ***          GUI Layout             ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('300x180+500+500')

#*** Add a label ***
lab1 = ttk.Label(root, text = "")
lab1.pack()

#*** After 2 seconds, change label's text ***
root.after(2000, change_label)

#*** Wait for events ***
root.mainloop()
```

A + character is added to the string every two seconds.

The string continues to grow indefinitely.

Activity 4.42

The code for the event handler should now be:

```
def change_label():
    """Adds + symbol to label's string every 2 secs
    """
```

```
        lab1['text'] += "+"
        #*** Reschedule event ***
        root.after(2000, change_label)
        print(root.after(2000, change_label))
```

The values displayed in the console window will be

after#2
after#4
after#6
etc.

Activity 4.43

No solution required.

Activity 4.44

No solution required.

Activity 4.45

Modified code for *VirtualEvent.py*:

```
#*** Using a Virtual Event ***

#*** Import modules ***
from tkinter import *
from tkinter import ttk


#****************************************
# ***        Event Handlers          ***
# **************************************
def change_colour(e):
    """Changes the text colour of lab1 to that
       given as text on button, e
    """
    lab1['foreground'] = e.widget.cget('text')

def remove_return():
    """Removes the native Return key press from
       the BPress virtual event
    """
    root.event_delete('<<BPress>>',
    ↳'<KeyPress Return>')

#****************************************
# ***          GUI Layout            ***
# **************************************
#*** Create window ***
root = Tk()
root.geometry('320x180+500+500')

#*** Create a Virtual event *** for mouse button 1
↳or return key ***
root.event_add('<<BPress>>', '<Button-1>',
↳'<KeyPress Return>')

#*** Create label ***
lab1 = ttk.Label(root, text = 'Changes colour')
lab1.pack(side = 'top')

# Create three buttons all linked to the virtual
↳event ***
but1 = ttk.Button(root, text = 'red')
but1.pack(side = 'left')
but1.bind('<<BPress>>', change_colour)

but2 = ttk.Button(root, text = 'green')
but2.pack(side = 'left')
but2.bind('<<BPress>>', change_colour)

but3 = ttk.Button(root, text = 'blue')
but3.pack(side = 'left')
but3.bind('<<BPress>>', change_colour)

but4 = ttk.Button(root, text = 'Remove Return',
↳command = remove_return)
but4.pack(side = 'left')


#*** Wait for events ***
root.mainloop()
```

Activity 4.46

In *VirtualEvent.py*, the handler *remove_return()* is now coded as:

```
def remove_return():
    """Removes the native Return key press from
       the BPress virtual event
    """
    print(root.event_info('<<BPress>>'))
    root.event_delete('<<BPress>>',
     ↳'<KeyPress Return>')
    print(root.event_info('<<BPress>>'))
```

Activity 4.47

In *VirtualEvent.py*, the handler *remove_return()* is now coded as:

```
def remove_return():
    """Removes the native Return key press from
       the BPress virtual event and turns text blue
    """
    root.event_delete('<<BPress>>',
     ↳'<KeyPress Return>')
     #*** Create blue text ***
     but3.event_generate('<<BPress>>')
```

Activity 4.48

When the fourth button is pressed, it forces a call to *change_colour()*. But because this is a generated `'<<BPress>>'` event rather than a true one, the `Event` object does not have its usual values and hence the *time* attribute has a zero value which is then copied to the label.

Activity 4.49

```
Updated code for remove_return():

    def remove_return():
        """Removes the native Return key press from
           the BPress virtual event and turns text blue
        """
        root.event_delete('<<BPress>>','<KeyPress
        ↳Return>')
        #*** Create blue text ***
        but3.event_generate('<<BPress>>', time = 140000)
```

When the fourth button is pressed, the label displays the value 14000.