

Tkinter programming

Jan Bodnar

January 25, 2016

Contents

| | |
|---------------------------------------|-----------|
| Preface | iv |
| About the author | v |
| 1 Introduction | 1 |
| 1.1 Tkinter | 1 |
| 1.2 Simple example | 1 |
| 1.3 Centering a window | 3 |
| 1.4 Colours | 5 |
| 1.5 Fonts | 10 |
| 1.6 Styles | 12 |
| 2 Layout management | 15 |
| 2.1 Absolute positioning | 15 |
| 2.2 Row of buttons | 18 |
| 2.3 Corner buttons | 19 |
| 2.4 Rows of buttons | 20 |
| 2.5 New folder with pack | 23 |
| 2.6 Windows with pack | 25 |
| 2.7 Calculator | 27 |
| 2.8 New folder with grid | 30 |
| 2.9 Windows with grid | 32 |
| 3 Events | 35 |
| 3.1 The command parameter | 35 |
| 3.2 Binding events | 36 |
| 3.3 Unbinding events | 38 |
| 3.4 Multiple event handlers | 40 |
| 3.5 Event object | 41 |
| 3.6 Event source | 43 |
| 3.7 Binding widget class | 45 |
| 3.8 Custom event | 47 |
| 3.9 Protocols | 49 |
| 3.10 Animation | 51 |
| 3.11 Floating window | 53 |
| 3.12 Splash screen | 56 |
| 3.13 Notifications | 59 |

| | | |
|----------|--|------------|
| 4 | Widgets | 64 |
| 4.1 | Button | 64 |
| 4.2 | Label | 65 |
| 4.3 | Message | 67 |
| 4.4 | Separator | 68 |
| 4.5 | Frame | 70 |
| 4.6 | LabelFrame | 71 |
| 4.7 | Checkbutton | 72 |
| 4.8 | Radiobutton | 74 |
| 4.9 | Entry | 76 |
| 4.10 | Scale | 80 |
| 4.11 | Spinbox | 82 |
| 4.12 | OptionMenu | 84 |
| 4.13 | Combobox | 85 |
| 4.14 | Scrollbar | 88 |
| 4.15 | Notebook | 90 |
| 4.16 | PanedWindow | 91 |
| 4.17 | Progressbar | 93 |
| 5 | Menus and toolbars | 96 |
| 5.1 | Simple menu | 96 |
| 5.2 | Submenu | 98 |
| 5.3 | Popup menu | 100 |
| 5.4 | Check menu button | 102 |
| 5.5 | Toolbar | 104 |
| 6 | Listbox | 107 |
| 6.1 | Item selection | 107 |
| 6.2 | Multiple selection | 109 |
| 6.3 | Attaching a scrollbar | 111 |
| 6.4 | Adding and removing items | 113 |
| 6.5 | Sorting items | 115 |
| 6.6 | Reordering items by dragging | 117 |
| 7 | Text | 121 |
| 7.1 | Simple example | 121 |
| 7.2 | Fonts | 123 |
| 7.3 | Selecting text | 125 |
| 7.4 | Image | 126 |
| 7.5 | Undo, redo | 128 |
| 7.6 | Cut, copy, and paste | 130 |
| 7.7 | Searching text | 132 |
| 7.8 | Spell checking | 134 |
| 7.9 | Opening, saving files | 138 |
| 8 | Treeview | 146 |
| 8.1 | Simple example | 146 |
| 8.2 | Row colours | 148 |
| 8.3 | Hierarchy | 150 |
| 8.4 | Images | 152 |

| | | |
|----------|--|------------|
| 8.5 | Selection | 154 |
| 8.6 | Inserting and deleting items | 156 |
| 8.7 | Double clicking a row | 159 |
| 8.8 | Sorting | 161 |
| 8.9 | File browser | 164 |
| 9 | Canvas | 171 |
| 9.1 | Lines | 171 |
| 9.2 | Line joins | 172 |
| 9.3 | Line caps | 173 |
| 9.4 | Cubic line | 174 |
| 9.5 | Colours | 175 |
| 9.6 | Shapes | 176 |
| 9.7 | Image | 177 |
| 9.8 | Text | 178 |
| 9.9 | Dragging items | 179 |
| 9.10 | Arkanoid | 182 |
| | Bibliography | 191 |

Preface

This is Tkinter programming e-book. It is a detailed introduction to the Tkinter toolkit. The e-book uses Python 3 and Tk 8.6.

This e-book contains 9 chapters:

1. Introduction
2. Layout management
3. Events
4. Widgets
5. Menus and toolbars
6. Listbox
7. Text
8. Treeview
9. Canvas

The Introduction chapter introduces Tkinter and explains some fundamentals of the toolkit. The Layout management chapter explains the layout management of Tkinter widgets. This area is considered to be one of the most difficult areas of GUI programming. Therefore, this e-book gives particular emphasis on the layout management process.

The Events chapter is an in-depth coverage of the event handling system of Tkinter. The Widgets chapter covers 17 Tkinter widgets; each widget has a small working example. In the Menus and toolbars chapter, we show how to work with menus and toolbars. The Listbox chapter is dedicated to the listbox widget. The Text chapter covers the text widget in a detail; it contains 9 examples including searching for text and spell checking. The Treeview widget dedicates 9 examples to the treeview widget. In the Canvas chapter we do some drawing. At the end of the chapter, we create an Arkanoid game.

About the author

My name is Jan Bodnar. I am Hungarian and I come from Slovakia. Currently I live in Bratislava. Apart from working with computers I read a lot. Mostly belles-letters and history. I study several foreign languages. Being aware of that healthy soul lives in a healthy body, I exercise regularly, work in the garden, and often take tours. Recently I became an aspiring dancer.

I run the zetcode.com site. *Tkinter programming* is my sixth e-book. I have also written the following e-books: *Swing layout management*, *Advanced PyQt4*, *Advanced wxPython*, *SQLite Python*, and *Advanced Java Swing*. I hope my new e-book will be useful to you.



Thank you for buying this e-book.

Chapter 1

Introduction

This chapter is an introduction to Tkinter. We present the Tkinter toolkit and cover some of its fundamentals.

1.1 Tkinter

Tkinter is a default Python GUI toolkit. It is a Python binding to Tk, which is the original GUI library for the Tcl language. Tkinter is implemented as a Python wrapper around a complete Tcl interpreter embedded in the Python interpreter. Python's tkinter module talks to Tk, and the Tk API in turn interfaces with the underlying window system: Microsoft Windows, X Windows on Unix or Linux, or Macintosh.

There are several other popular Python GUI toolkits. The most popular are wxPython, PyQt, and PyGTK.¹

1.2 Simple example

In our first example, we show a small window on the screen.

Listing 1.1: simple.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script shows a simple window
on the screen.

Author: Jan Bodnar
Last modified: January 2016
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame
```

¹There are tutorials for all three toolkits on zetcode.com

```

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

        self.initUI()

    def initUI(self):

        self.parent.title("Simple")
        self.pack(fill=BOTH, expand=True)

def main():

    root = Tk()
    root.geometry("250x150+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

While this code is very small, the application window can do quite a lot. It can be resized, maximized, or minimized. All the complexity that comes with it has been hidden from the application programmer.

```

from tkinter import Tk, BOTH
from tkinter.ttk import Frame

```

Here we import `Tk` and `Frame` classes. The first class is used to create a root window. The latter is a container for other widgets. The `tkinter.ttk` module provides access to the Tk themed widget set.

```

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

```

Our `Example` class inherits from the `Frame` container widget. In the `__init__()` constructor method we call the constructor of our inherited class.

```

self.parent = parent

```

We save a reference to the parent widget. The parent widget is the Tk root window in our case.

```

self.initUI()

```

The creation of the user interface is delegated to the `initUI()` method.

```

self.parent.title("Simple")

```


We set the title of the window using the `title()` method.

```
self.pack(fill=BOTH, expand=True)
```

The `pack()` method is one of the three geometry managers in Tkinter. It organizes widgets into horizontal and vertical boxes. Here we put the **Frame** widget, accessed via the `self` attribute, to the Tk root window. It is expanded in both directions. In other words, it takes the whole client space of the root window.

```
root = Tk()
```

The root window is created. The root window is a main application window in Tkinter programs. It has a title bar and borders. These are provided by the window manager. The root window must be created before any other widgets.

```
root.geometry("250x150+300+300")
```

The `geometry()` method sets the size for the window and positions it on the screen. The first two parameters are the width and height of the window. The last two parameters are x and y screen coordinates.

```
app = Example(root)
```

An instance of the application class is created.

```
root.mainloop()
```

Finally, we enter the mainloop. The event handling starts from this point. The mainloop receives events from the window system and dispatches them to the application widgets. It is terminated when we click on the close button of the titlebar or call the `quit()` method.

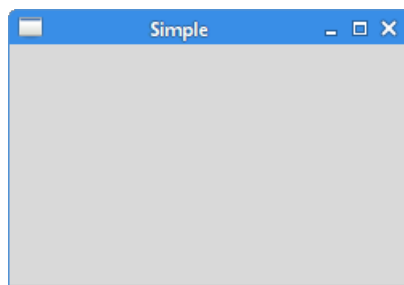


Figure 1.1: Simple example

Figure 1.1 shows a small window created with the code example.

1.3 Centering a window

In the second code example, we center an application window on the screen.

Listing 1.2: `center_window.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script centers a small
window on the screen.

Author: Jan Bodnar
Last modified: January 2016
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

        self.initUI()
        self.centerWindow()

    def initUI(self):

        self.parent.title("Centered window")
        self.pack(fill=BOTH, expand=True)

    def centerWindow(self):

        w = 290
        h = 150

        sw = self.parent.winfo_screenwidth()
        sh = self.parent.winfo_screenheight()

        x = (sw - w)/2
        y = (sh - h)/2
        self.parent.geometry('%dx%d+%d+%d' % (w, h, x, y))

def main():

    root = Tk()
    ex = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

We need to determine the size of the window and the size of the screen to position a window in the center of the screen.

```
w = 290
```

```
h = 150
```

These are the width and height values of the application window.

```
sw = self.parent.winfo_screenwidth()
sh = self.parent.winfo_screenheight()
```

We determine the width and height of the screen.

```
x = (sw - w)/2
y = (sh - h)/2
```

We calculate the required x and y coordinates.

```
self.parent.geometry('%dx%d+%d+%d' % (w, h, x, y))
```

Finally, the `geometry()` method is used to place the window in the center of the screen.

1.4 Colours

In computers, colours are represented by various colour models. The most common are RGB and HSV models. Tkinter uses RGB model in which colours are represented by combinations of red, green, and blue values. In addition, Tkinter has built-in named colours.

Listing 1.3: `initUI()` from `linecaps.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we use Tkinter's RGB model
to create various colour values.

Author: Jan Bodnar
Last modified: January 2016
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Label

COLOURS = [ "#f45", "#ee5", "#aa4", "#a1e433", "#e34412", "#116611",
            "#111eeeff", "#3aa922191", "#abbabbaaa" ]

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()
```

```

def initUI(self):

    self.parent.title("Colours")
    self.pack(fill=BOTH, expand=True)

    i = 0
    for col in COLOURS:
        lbl = Label(self, text=col, background=col)
        lbl.grid(row=0, column=i)
        i += 1

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we display various colour values as backgrounds for labels.

```

COLOURS = [ "#f45", "#ee5", "#aa4", "#a1e433", "#e34412", "#116611",
            "#111eeefff", "#3aa922191", "#abbabbaaa" ]

```

In the COLOURS list, we specify nine colour values in RGB model. The red, green, and blue parts of the colour value can be specified in four, eight, or twelve bits.

```

i = 0
for col in COLOURS:
    lbl = Label(self, text=col, background=col)
    lbl.grid(row=0, column=i)
    i += 1

```

We set these colours for backgrounds of twelve label widgets.



Figure 1.2: Colours

Figure 1.2 shows twelve colour values.

Another option for specifying colours is to use named colours. Colours are given names such as aquamarine, lavender, snow, or misty rose.

Listing 1.4: named_colours.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we use Tkinter's RGB model

```

to create various colour values.

Author: Jan Bodnar

Last modified: January 2016

Website: www.zetcode.com

"""

```
from tkinter import Tk, BOTH
```

```
from tkinter.ttk import Frame, Label
```

```
COLOURS = ['snow', 'ghost white', 'white smoke', 'gainsboro',
            'floral white', 'old lace', 'linen', 'antique white',
            'papaya whip', 'blanched almond', 'bisque', 'peach puff',
            'navajo white', 'lemon chiffon', 'mint cream', 'azure',
            'alice blue', 'lavender', 'lavender blush', 'misty rose',
            'dark slate gray', 'dim gray', 'slate gray', 'light slate gray',
            'gray', 'light gray', 'midnight blue', 'navy', 'cornflower blue',
            'dark slate blue', 'slate blue', 'medium slate blue',
            'light slate blue', 'medium blue', 'royal blue', 'blue',
            'dodger blue', 'deep sky blue', 'sky blue', 'light sky blue',
            'steel blue', 'light steel blue', 'light blue', 'powder blue',
            'pale turquoise', 'dark turquoise', 'medium turquoise',
            'turquoise', 'cyan', 'light cyan', 'cadet blue',
            'medium aquamarine', 'aquamarine', 'dark green',
            'dark olive green', 'dark sea green', 'sea green',
            'medium sea green', 'light sea green', 'pale green',
            'spring green', 'lawn green', 'medium spring green',
            'green yellow', 'lime green', 'yellow green', 'forest green',
            'olive drab', 'dark khaki', 'khaki', 'pale goldenrod',
            'light goldenrod yellow', 'light yellow', 'yellow', 'gold',
            'light goldenrod', 'goldenrod', 'dark goldenrod', 'rosy brown',
            'indian red', 'saddle brown', 'sandy brown', 'dark salmon',
            'salmon', 'light salmon', 'orange', 'dark orange', 'coral',
            'light coral', 'tomato', 'orange red', 'red', 'hot pink',
            'deep pink', 'pink', 'light pink', 'pale violet red', 'maroon',
            'medium violet red', 'violet red', 'medium orchid', 'dark orchid',
            'dark violet', 'blue violet', 'purple', 'medium purple', 'thistle',
            'snow2', 'snow3', 'snow4', 'seashell2', 'seashell3', 'seashell4',
            'AntiqueWhite1', 'AntiqueWhite2', 'AntiqueWhite3', 'AntiqueWhite4',
            'bisque2', 'bisque3', 'bisque4', 'PeachPuff2', 'PeachPuff3',
            'PeachPuff4', 'NavajoWhite2', 'NavajoWhite3', 'NavajoWhite4',
            'LemonChiffon2', 'LemonChiffon3', 'LemonChiffon4', 'cornsilk2',
            'cornsilk3', 'cornsilk4', 'ivory2', 'ivory3', 'ivory4',
            'honeydew2', 'honeydew3', 'honeydew4', 'LavenderBlush2',
            'LavenderBlush3', 'LavenderBlush4', 'MistyRose2', 'MistyRose3',
            'MistyRose4', 'azure2', 'azure3', 'azure4', 'SlateBlue1',
            'SlateBlue2', 'SlateBlue3', 'SlateBlue4', 'RoyalBlue1',
            'RoyalBlue2', 'RoyalBlue3', 'RoyalBlue4', 'blue2', 'blue4',
            'DodgerBlue2', 'DodgerBlue3', 'DodgerBlue4', 'SteelBlue1',
            'SteelBlue2', 'SteelBlue3', 'SteelBlue4', 'DeepSkyBlue2',
            'DeepSkyBlue3', 'DeepSkyBlue4', 'SkyBlue1', 'SkyBlue2', 'SkyBlue3',
            'SkyBlue4', 'LightSkyBlue1', 'LightSkyBlue2', 'LightSkyBlue3',
            'LightSkyBlue4', 'SlateGray1', 'SlateGray2', 'SlateGray3',
            'SlateGray4', 'LightSteelBlue1', 'LightSteelBlue2',
            'LightSteelBlue3', 'LightSteelBlue4', 'LightBlue1', 'LightBlue2',
            'LightBlue3', 'LightBlue4', 'LightCyan2', 'LightCyan3',
            'LightCyan4', 'PaleTurquoise1', 'PaleTurquoise2', 'PaleTurquoise3',
            'PaleTurquoise4', 'CadetBlue1', 'CadetBlue2', 'CadetBlue3',
            'CadetBlue4', 'turquoise1', 'turquoise2', 'turquoise3',
            'turquoise4', 'cyan2', 'cyan3', 'cyan4', 'DarkSlateGray1',
            'DarkSlateGray2', 'DarkSlateGray3', 'DarkSlateGray4',
            'aquamarine2', 'aquamarine4', 'DarkSeaGreen1', 'DarkSeaGreen2',
```

```

'DarkSeaGreen3', 'DarkSeaGreen4', 'SeaGreen1', 'SeaGreen2',
'SeaGreen3', 'PaleGreen1', 'PaleGreen2', 'PaleGreen3',
'PaleGreen4', 'SpringGreen2', 'SpringGreen3', 'SpringGreen4',
'green2', 'green3', 'green4', 'chartreuse2', 'chartreuse3',
'chartreuse4', 'OliveDrab1', 'OliveDrab2', 'OliveDrab4',
'DarkOliveGreen1', 'DarkOliveGreen2', 'DarkOliveGreen3',
'DarkOliveGreen4', 'khaki1', 'khaki2', 'khaki3', 'khaki4',
'LightGoldenrod1', 'LightGoldenrod2', 'LightGoldenrod3',
'LightGoldenrod4', 'LightYellow2', 'LightYellow3', 'LightYellow4',
'yellow2', 'yellow3', 'yellow4', 'gold2', 'gold3', 'gold4',
'goldenrod1', 'goldenrod2', 'goldenrod3', 'goldenrod4',
'DarkGoldenrod1', 'DarkGoldenrod2', 'DarkGoldenrod3',
'DarkGoldenrod4', 'RosyBrown1', 'RosyBrown2', 'RosyBrown3',
'RosyBrown4', 'IndianRed1', 'IndianRed2', 'IndianRed3',
'IndianRed4', 'sienna1', 'sienna2', 'sienna3', 'sienna4',
'burlywood1', 'burlywood2', 'burlywood3', 'burlywood4', 'wheat1',
'wheat2', 'wheat3', 'wheat4', 'tan1', 'tan2', 'tan4', 'chocolate1',
'chocolate2', 'chocolate3', 'firebrick1', 'firebrick2',
'firebrick3', 'firebrick4', 'brown1', 'brown2', 'brown3', 'brown4',
'salmon1', 'salmon2', 'salmon3', 'salmon4', 'LightSalmon2',
'LightSalmon3', 'LightSalmon4', 'orange2', 'orange3', 'orange4',
'DarkOrange1', 'DarkOrange2', 'DarkOrange3', 'DarkOrange4',
'coral1', 'coral2', 'coral3', 'coral4', 'tomato2', 'tomato3',
'tomato4', 'OrangeRed2', 'OrangeRed3', 'OrangeRed4', 'red2',
'red3', 'red4', 'DeepPink2', 'DeepPink3', 'DeepPink4', 'HotPink1',
'HotPink2', 'HotPink3', 'HotPink4', 'pink1', 'pink2', 'pink3',
'pink4', 'LightPink1', 'LightPink2', 'LightPink3', 'LightPink4',
'PaleVioletRed1', 'PaleVioletRed2', 'PaleVioletRed3',
'PaleVioletRed4', 'maroon1', 'maroon2', 'maroon3', 'maroon4',
'VioletRed1', 'VioletRed2', 'VioletRed3', 'VioletRed4', 'magenta2',
'magenta3', 'magenta4', 'orchid1', 'orchid2', 'orchid3', 'orchid4',
'plum1', 'plum2', 'plum3', 'plum4', 'MediumOrchid1',
'MediumOrchid2', 'MediumOrchid3', 'MediumOrchid4', 'DarkOrchid1',
'DarkOrchid2', 'DarkOrchid3', 'DarkOrchid4', 'purple1', 'purple2',
'purple3', 'purple4', 'MediumPurple1', 'MediumPurple2',
'MediumPurple3', 'MediumPurple4', 'thistle1', 'thistle2',
'thistle3', 'thistle4', 'gray1', 'gray2', 'gray3', 'gray4',
'gray5', 'gray6', 'gray7', 'gray8', 'gray9', 'gray10', 'gray11',
'gray12', 'gray13', 'gray14', 'gray15', 'gray16', 'gray17',
'gray18', 'gray19', 'gray20', 'gray21', 'gray22', 'gray23',
'gray24', 'gray25', 'gray26', 'gray27', 'gray28', 'gray29',
'gray30', 'gray31', 'gray32', 'gray33', 'gray34', 'gray35',
'gray36', 'gray37', 'gray38', 'gray39', 'gray40', 'gray42',
'gray43', 'gray44', 'gray45', 'gray46', 'gray47', 'gray48',
'gray49', 'gray50', 'gray51', 'gray52', 'gray53', 'gray54',
'gray55', 'gray56', 'gray57', 'gray58', 'gray59', 'gray60',
'gray61', 'gray62', 'gray63', 'gray64', 'gray65', 'gray66',
'gray67', 'gray68', 'gray69', 'gray70', 'gray71', 'gray72',
'gray73', 'gray74', 'gray75', 'gray76', 'gray77', 'gray78',
'gray79', 'gray80', 'gray81', 'gray82', 'gray83', 'gray84',
'gray85', 'gray86', 'gray87', 'gray88', 'gray89', 'gray90',
'gray91', 'gray92', 'gray93', 'gray94', 'gray95', 'gray97',
'gray98', 'gray99']

```

```

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

```

```

def initUI(self):

    self.parent.title("Named colours")
    self.pack(fill=BOTH, expand=True)

    r = 0
    c = 0

    for col in COLOURS:

        label = Label(self, text=col, background=col)
        label.grid(row=r, column=c, sticky="ew")
        r += 1

        if r > 36:
            r = 0
            c += 1

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+30+30")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example shows 479 named colours on the window.

```

COLOURS = ['snow', 'ghost white', 'white smoke', 'gainsboro',
           'floral white', 'old lace', 'linen', 'antique white',
           ...

```

The COLOURS is a list of Tkinter named colours.

```

r = 0
c = 0

for col in COLOURS:

    label = Label(self, text=col, background=col)
    label.grid(row=r, column=c, sticky="ew")
    r += 1

    if r > 36:
        r = 0
        c += 1

```

We go through the colour list and apply them on the labels.



Figure 1.3: Named colours

Figure 1.3 shows a portion of the window displaying named colours.

1.5 Fonts

Tkinter has a `tkinter.font` module for working with fonts. It has some built-in fonts such as `TkToolTipFont`, `TkDefaultFont`, or `TkTextFont`. The `tkinter.font` module has a `names()` method to get the names of all defined fonts and a `families()` method to retrieve all font families on the current platform.

Listing 1.5: fonts.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we display text in three
different fonts.

Author: Jan Bodnar
Last modified: January 2016
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Label, Notebook, Style

from tkinter.font import Font

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)
```



```

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Fonts")
        self.pack(fill=BOTH, expand=True)

        txt = "Tkinter is a default Python GUI toolkit"

        myfont = Font(family="Ubuntu Mono", size=16)
        label1 = Label(self, text=txt, font=myfont)
        label1.grid(row=0, column=0)

        label2 = Label(self, text=txt, font="TkTextFont")
        label2.grid(row=1, column=0)

        label3 = Label(self, text=txt, font=('Times', '18', 'italic'))
        label3.grid(row=2, column=0)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example shows three labels with three different fonts.

```
from tkinter.font import Font
```

We import the `Font` class from the `tkinter.font` module.

```
myfont = Font(family="Ubuntu Mono", size=16)
label1 = Label(self, text=txt, font=myfont)
label1.grid(row=0, column=0)
```

A specific font is created with the `Font` class. If the font is not available on the platform, Tkinter reverts to some default font.

```
label2 = Label(self, text=txt, font="TkTextFont")
label2.grid(row=1, column=0)
```

In the second case, we use a built-in font name.

```
label3 = Label(self, text=txt, font=('Times', '18', 'italic'))
label3.grid(row=2, column=0)
```

A font can also be specified as a tuple of strings.



Figure 1.4: Fonts

Figure 1.4 displays three sentences in three different fonts.

1.6 Styles

Tk 8.5 introduced widget styles; a new `tkinter.ttk` module is available. It contains all widgets on which it is possible to apply styles. (Some widgets, including `Text`, `Listbox`, `Canvas`, and `Spinbox` do not support styles yet.)

A *style* is the description of the appearance of a widget class. A widget class is used by Tk to identify the type of a particular widget. Each widget class has a default style. The default style name of a widget is ‘T’ prefixed to the widget name; for example, `TButton` for a button widget, `TLabel` for a label widget, or `TRadioButton` for a radio button. (The `Treeview` widget is an exception, its default style is called `Treeview`.)

A theme is a complete look and feel, customizing the appearance of all the widgets. Each theme comes with a predefined set of styles, but we can customize the built-in styles or create our own new styles.

Listing 1.6: `default_style.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we change the default style
for the label widget class.

Author: Jan Bodnar
Last modified: January 2016
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Label, Style

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):
```

```

self.parent.title("Default style")
self.pack(fill=BOTH, expand=True)

s = Style()
s.configure('TLabel', background='dark sea green',
            font=('Helvetica', '16'))

lbl1 = Label(self, text="zetcode.com")
lbl2 = Label(self, text="spoznaj.sk")

lbl1.pack()
lbl2.pack()

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("250x100+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we change the default style of the label widget class.

```
from tkinter.ttk import Frame, Label, Style
```

From the `tkinter.ttk` module, we import the `Label` widget and the `Style` class. The `Style` class is used to work with Tkinter styles.

```
s = Style()
```

A style object is created.

```
s.configure('TLabel', background='dark sea green',
            font=('Helvetica', '16'))
```

We modify the `TLabel` style; it is a default style for all label widgets.

```
lbl1 = Label(self, text="zetcode.com")
lbl2 = Label(self, text="spoznaj.sk")
```

Two labels are created; both automatically have the modified style.

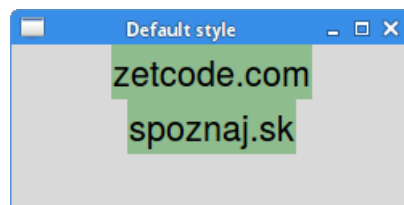


Figure 1.5: Default style

Figure 1.5 shows two labels having a modified default style.

In the following example, we create custom styles.

Listing 1.7: `initUI()` from `custom_styles.py`

```
def initUI(self):

    self.parent.title("Custom styles")
    self.pack(fill=BOTH, expand=True)

    notebook = Notebook(self)

    s = Style()
    s.configure('Tab1.TFrame', background='midnight blue')
    s.configure('Tab2.TFrame', background='lime green')
    s.configure('Tab3.TFrame', background='khaki')

    frame1 = Frame(width=400, height=300, style='Tab1.TFrame')
    frame2 = Frame(width=400, height=300, style='Tab2.TFrame')
    frame3 = Frame(width=400, height=300, style='Tab3.TFrame')

    notebook.add(frame1, text="Tab 1")
    notebook.add(frame2, text="Tab 2")
    notebook.add(frame3, text="Tab 3")
    notebook.pack(pady=5)
```

A `Notebook` widget is created; it contains three frames in three tabs. Each frame has a different background specified with a custom style.

```
s = Style()
s.configure('Tab1.TFrame', background='midnight blue')
s.configure('Tab2.TFrame', background='lime green')
s.configure('Tab3.TFrame', background='khaki')
```

Three custom styles are defined. By prepending another name followed by a dot onto an existing style, we are implicitly creating a new style derived from the existing one.

```
frame1 = Frame(width=400, height=300, style='Tab1.TFrame')
```

A `Frame` widget is created. The `style` option specifies the custom style used.

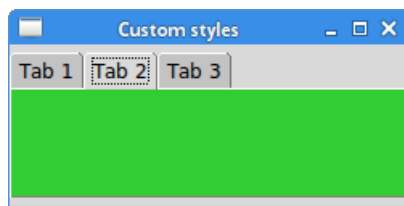


Figure 1.6: Custom styles

Figure 1.6 shows a `Notebook` widget with a frame having a lime green background. The colour was set with a custom style.

Chapter 2

Layout management

When we design the GUI of our application, we decide what widgets we will use and how we will organize those widgets in the application. To organize our widgets, we use specialized non-visible objects called layout managers.

There are two kinds of widgets: containers and their children. The containers group children into suitable layouts.

Tkinter has three built-in layout managers: the *place*, *pack*, and *grid* managers. The *place* geometry manager positions widgets in absolute coordinates. The *pack* geometry manager organizes widgets in horizontal and vertical boxes. It is a very simple manager; to create more complex layouts, we can combine several frames each having some partial layout solved with the *pack* manager. The *grid* geometry manager puts widgets in a two-dimensional grid. It is the most complex and most capable layout manager in Tkinter.

It is possible to use the *pack* and the *grid* manager together, but each must have its own parent.

2.1 Absolute positioning

In most cases, programmers should use layout managers. There are a few situations where we could use absolute positioning. In absolute positioning, the programmer specifies the position and the size of each widget in pixels. The size and the position of a widget do not change if we resize a window. Applications look different on various platforms, and what looks OK on Linux, might not look OK on Mac OS. Changing fonts in our application might spoil the layout. If we translate our application into another language, we must redo our layout.

Absolute positioning is done with the *place* manager.

Listing 2.1: absolute.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we lay out images
using absolute positioning.
```

```

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from PIL import Image, ImageTk
from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Label, Style

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Absolute positioning")
        self.pack(fill=BOTH, expand=True)

        style = Style()
        style.configure("TFrame", background="#333")

        bard = Image.open("bardejov.jpg")
        bardejov = ImageTk.PhotoImage(bard)
        label1 = Label(self, image=bardejov)
        label1.image = bardejov
        label1.place(x=20, y=20)

        rot = Image.open("rotunda.jpg")
        rotunda = ImageTk.PhotoImage(rot)
        label2 = Label(self, image=rotunda)
        label2.image = rotunda
        label2.place(x=40, y=160)

        minc = Image.open("mincol.jpg")
        mincol = ImageTk.PhotoImage(minc)
        label3 = Label(self, image=mincol)
        label3.image = mincol
        label3.place(x=170, y=50)

def main():

    root = Tk()
    root.geometry("300x280+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we place three images using absolute positioning.

```

from PIL import Image, ImageTk

```

We use `Image` and `ImageTk` from the Python Imaging Library (PIL) module.

```
self.pack(fill=BOTH, expand=True)
```

This line makes the parent widget, the frame, take the whole area of its toplevel window. The three labels with their images are placed on the frame.

```
style = Style()
style.configure("TFrame", background="#333")
```

We configure our frame to have a dark gray background.

```
bard = Image.open("bardejov.jpg")
bardejov = ImageTk.PhotoImage(bard)
```

We create an image object and a photo image object from an image in the current working directory.

```
label1 = Label(self, image=bardejov)
```

We create a label with an image. Labels can contain text or images.

```
label1.image = bardejov
```

We must keep the reference to the image to prevent image from being garbage collected.

```
label1.place(x=20, y=20)
```

The label is placed on the frame at `x=20` and `y=20` coordinates.

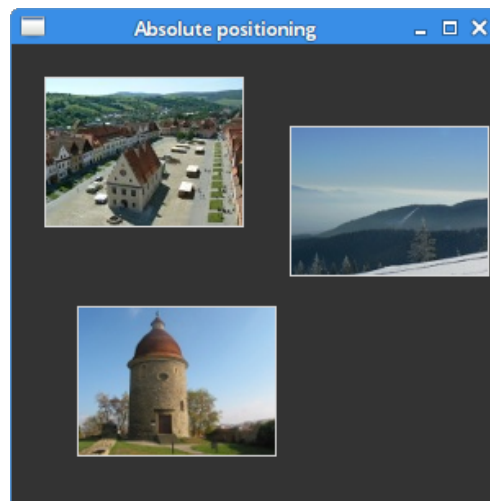


Figure 2.1: Absolute positioning

Figure 2.1 shows three images on a frame positioned in absolute coordinates.

2.2 Row of buttons

In the following example, we create a row of buttons with the pack manager. We put some space between the buttons.

Listing 2.2: rowofbuttons.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, four buttons into one row
using the pack manager.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, LEFT
from tkinter.ttk import Frame, Button

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Row of buttons")
        self.pack(fill=BOTH, expand=True)

        btn1 = Button(self, text="Button")
        btn2 = Button(self, text="Button")
        btn3 = Button(self, text="Button")
        btn4 = Button(self, text="Button")

        btn1.pack(side=LEFT, padx=5)
        btn2.pack(side=LEFT)
        btn3.pack(side=LEFT, padx=5)
        btn4.pack(side=LEFT)

def main():

    root = Tk()
    root.geometry("380x120+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

The example puts four buttons into a row.

```
btn1.pack(side=LEFT, padx=5)
btn2.pack(side=LEFT)
btn3.pack(side=LEFT, padx=5)
btn4.pack(side=LEFT)
```

The `side` parameter of each button is set to `LEFT`; the buttons form a line. The `padx` parameter puts some horizontal space between the buttons.

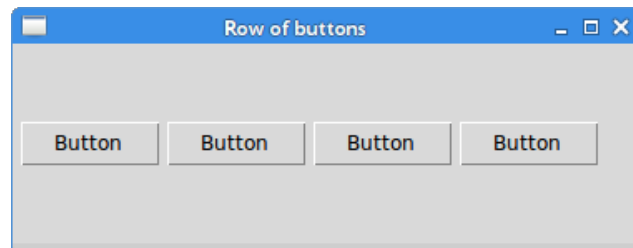


Figure 2.2: Row of buttons

Figure 2.2 shows four buttons placed in one row.

2.3 Corner buttons

The following example places two buttons in the bottom-right corner of the window. The buttons are anchored with the pack manager.

Listing 2.3: buttons.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we use the pack manager
to position two buttons in the
bottom-right corner of the window.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, RIGHT, BOTH, S
from tkinter.ttk import Frame, Button

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):
```

```
def initUI(self):

    self.parent.title("Buttons")
    self.pack(fill=BOTH, expand=True)

    closeButton = Button(self, text="Close")
    closeButton.pack(anchor=S, side=RIGHT, padx=5, pady=5)
    okButton = Button(self, text="OK")
    okButton.pack(side=RIGHT, pady=5, anchor=S)

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

The buttons are placed in the corner using pack's `side` and `anchor` parameters.

```
closeButton.pack(anchor=S, side=RIGHT, padx=5, pady=5)
```

The Close button is anchored to the south with the `anchor` parameter and put to the right with the `side` parameter.

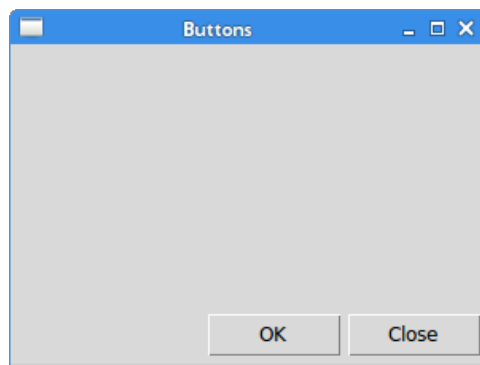


Figure 2.3

Figure 2.3 shows two push buttons in the bottom-right corner of the window.

2.4 Rows of buttons

The pack geometry manager is suitable for small layout tasks. To create more complicated layouts, we combine several frames each having its partial layout created with the pack manager.

Listing 2.4: button_rows.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we create two
rows of buttons.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, LEFT
from tkinter.ttk import Frame, Button

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Rows of buttons")
        self.pack(fill=BOTH, expand=True)

        frame1 = Frame(self)
        frame1.pack(padx=5, pady=5)

        btn1 = Button(frame1, text="Button")
        btn2 = Button(frame1, text="Button")
        btn3 = Button(frame1, text="Button")
        btn4 = Button(frame1, text="Button")

        btn1.pack(side=LEFT, padx=5)
        btn2.pack(side=LEFT)
        btn3.pack(side=LEFT, padx=5)
        btn4.pack(side=LEFT)

        frame2 = Frame(self)
        frame2.pack()

        btn1 = Button(frame2, text="Button")
        btn2 = Button(frame2, text="Button")
        btn3 = Button(frame2, text="Button")
        btn4 = Button(frame2, text="Button")

        btn1.pack(side=LEFT, padx=5)
        btn2.pack(side=LEFT)
        btn3.pack(side=LEFT, padx=5)
        btn4.pack(side=LEFT)

def main():
```

```
root = Tk()
root.geometry("380x120+300+300")
app = Example(root)
root.mainloop()

if __name__ == '__main__':
    main()
```

The example creates two rows of buttons. Each row is placed in a single frame widget. The frames are placed on the base parent frame.

```
frame1 = Frame(self)
frame1.pack(padx=5, pady=5)

btn1 = Button(frame1, text="Button")
btn2 = Button(frame1, text="Button")
btn3 = Button(frame1, text="Button")
btn4 = Button(frame1, text="Button")

btn1.pack(side=LEFT, padx=5)
btn2.pack(side=LEFT)
btn3.pack(side=LEFT, padx=5)
btn4.pack(side=LEFT)
```

The first four buttons are organized within the first frame widget.

```
frame2 = Frame(self)
frame2.pack()

btn1 = Button(frame2, text="Button")
btn2 = Button(frame2, text="Button")
btn3 = Button(frame2, text="Button")
btn4 = Button(frame2, text="Button")

btn1.pack(side=LEFT, padx=5)
btn2.pack(side=LEFT)
btn3.pack(side=LEFT, padx=5)
btn4.pack(side=LEFT)
```

The second row of buttons is managed with the second frame.

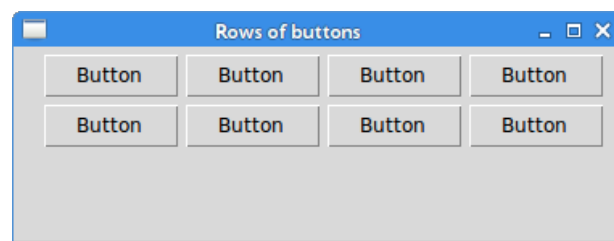


Figure 2.4: Rows of buttons

Figure 2.4 shows two rows of buttons managed by two pack managers in two frame widgets.

2.5 New folder with pack

The next example is a practical real-world layout created with the pack manager.

Listing 2.5: new_folder_pack.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we create a New folder
example with the pack manager.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Text, BOTH, LEFT, X, RIGHT
from tkinter.ttk import Frame, Button, Label, Entry

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("New folder")
        self.pack(fill=BOTH, expand=True)

        frame1 = Frame(self)
        frame1.pack(fill=X)

        lbl = Label(frame1, text="Name:")
        lbl.pack(side=LEFT, padx=5, pady=5)

        entry = Entry(frame1)
        entry.pack(side=LEFT, fill=X, padx=5, expand=True)

        frame2 = Frame(self)
        frame2.pack(fill=BOTH, expand=True)

        txt = Text(frame2, width=20, height=10)
        txt.pack(fill=BOTH, expand=True, padx=5)

        frame3 = Frame(self)
        frame3.pack(fill=X)

        closeBtn = Button(frame3, text="Close")
        closeBtn.pack(side=RIGHT, pady=5)
        okBtn = Button(frame3, text="OK")
        okBtn.pack(side=RIGHT, padx=5)
```

```
def main():

    root = Tk()
    root.geometry("330x300+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

To create the layout, we need four frames. Note that in addition to managing the children inside their frames, we also manage the frames within its base parent frame.

```
self.pack(fill=BOTH, expand=True)
```

The base frame occupies the whole area of the root window.

```
frame1 = Frame(self)
frame1.pack(fill=X)

lbl = Label(frame1, text="Name:")
lbl.pack(side=LEFT, padx=5, pady=5)

entry = Entry(frame1)
entry.pack(side=LEFT, fill=X, padx=5, expand=True)
```

In the first frame we put the label and the entry. The combination of the `fill` and the `expand` parameters makes the entry widget stretch horizontally.

```
frame2 = Frame(self)
frame2.pack(fill=BOTH, expand=True)

txt = Text(frame2, width=20, height=10)
txt.pack(fill=BOTH, expand=True, padx=5)
```

The second frame is occupied by the text widget. Both the frame and its child fill their allotted area in both directions.

```
frame3 = Frame(self)
frame3.pack(fill=X)

closeBtn = Button(frame3, text="Close")
closeBtn.pack(side=RIGHT, pady=5)
okBtn = Button(frame3, text="OK")
okBtn.pack(side=RIGHT, padx=5)
```

The third frame contains two buttons. The buttons are placed at the right side of their container.

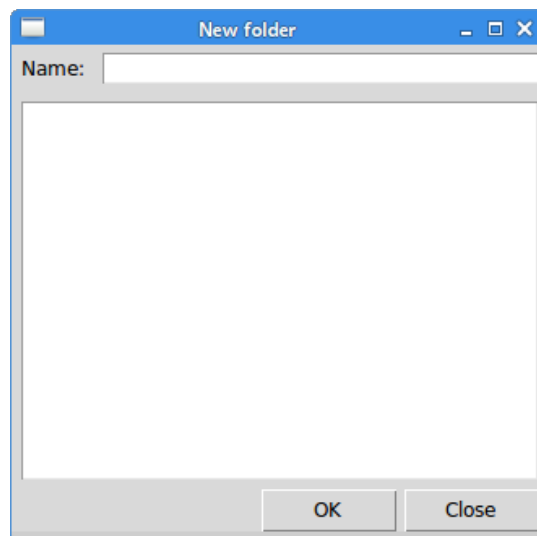


Figure 2.5: New folder with pack

Figure 2.5 shows the New folder layout example.

2.6 Windows with pack

The following example creates the Windows layout using the pack manager. The layout comes from the JDeveloper application.

Listing 2.6: windows_pack.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we create a New folder
example with the pack manager.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Text, BOTH, LEFT, X, Y, TOP, RIGHT
from tkinter.ttk import Frame, Button, Label

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()
```

```

def initUI(self):

    self.parent.title("Windows")
    self.pack(fill=BOTH, expand=True)

    frame1 = Frame(self)
    frame1.pack(fill=X)

    lbl = Label(frame1, text="Windows")
    lbl.pack(side=LEFT, padx=5, pady=5)

    frame2 = Frame(self)
    frame2.pack(fill=BOTH, expand=True)

    txt = Text(frame2, width=20, height=10)
    txt.pack(side=LEFT, fill=BOTH, expand=True, padx=5)

    frame22 = Frame(frame2)
    frame22.pack(side=LEFT, fill=Y)

    actBtn = Button(frame22, text="Activate")
    actBtn.pack(side=TOP, padx=5)

    actBtn = Button(frame22, text="Close")
    actBtn.pack(side=TOP, pady=5)

    frame3 = Frame(self)
    frame3.pack(fill=X)

    hlpBtn = Button(frame3, text="Help")
    hlpBtn.pack(side=LEFT, pady=5, padx=5)
    okBtn = Button(frame3, text="OK")
    okBtn.pack(side=RIGHT, padx=5)

def main():

    root = Tk()
    root.geometry("330x300+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In this example, we need four additional frames.

```

frame1 = Frame(self)
frame1.pack(fill=X)

lbl = Label(frame1, text="Windows")
lbl.pack(side=LEFT, padx=5, pady=5)

```

The first frame has one single label widget. It is positioned to the left and has some spacing around its borders.

```

frame2 = Frame(self)
frame2.pack(fill=BOTH, expand=True)

txt = Text(frame2, width=20, height=10)

```



```
txt.pack(side=LEFT, fill=BOTH, expand=True, padx=5)
```

The text widget goes into the left part of the second frame. It fills the bulk of the layout's area.

```
frame22 = Frame(frame2)
frame22.pack(side=LEFT, fill=Y)

actBtn = Button(frame22, text="Activate")
actBtn.pack(side=TOP, padx=5)

actBtn = Button(frame22, text="Close")
actBtn.pack(side=TOP, pady=5)
```

The two buttons go into the right part of the central frame. They go to the top of the container.

```
frame3 = Frame(self)
frame3.pack(fill=X)

hlpBtn = Button(frame3, text="Help")
hlpBtn.pack(side=LEFT, pady=5, padx=5)
okBtn = Button(frame3, text="OK")
okBtn.pack(side=RIGHT, padx=5)
```

Finally, two buttons are placed at the bottom frame. One is placed to the left side of the container, the other one to the right side.

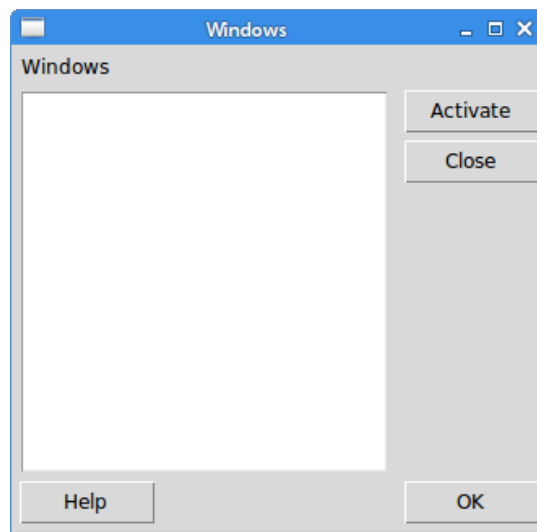


Figure 2.6: Windows with pack

Figure 2.6 shows the Windows layout.

2.7 Calculator

In the following example, the grid geometry manager is used to create a simple calculator layout.

Listing 2.7: calculator.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we use the grid manager
to create a skeleton of a calculator.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, W, E
from tkinter.ttk import Frame, Button, Entry, Style

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Calculator")

        Style().configure("TButton", padding=(0, 5, 0, 5),
                           font='serif 10')

        self.columnconfigure(0, pad=3)
        self.columnconfigure(1, pad=3)
        self.columnconfigure(2, pad=3)
        self.columnconfigure(3, pad=3)

        self.rowconfigure(0, pad=3)
        self.rowconfigure(1, pad=3)
        self.rowconfigure(2, pad=3)
        self.rowconfigure(3, pad=3)
        self.rowconfigure(4, pad=3)

        entry = Entry(self)
        entry.grid(row=0, column=0, columnspan=4, sticky=W+E)
        cls = Button(self, text="Cls")
        cls.grid(row=1, column=0)
        bck = Button(self, text="Back")
        bck.grid(row=1, column=1)
        lbl = Button(self)
        lbl.grid(row=1, column=2)
        clo = Button(self, text="Close")
        clo.grid(row=1, column=3)
        sev = Button(self, text="7")
        sev.grid(row=2, column=0)
        eig = Button(self, text="8")
        eig.grid(row=2, column=1)
        nin = Button(self, text="9")
```

```

        nin.grid(row=2, column=2)
        div = Button(self, text="/")
        div.grid(row=2, column=3)

        fou = Button(self, text="4")
        fou.grid(row=3, column=0)
        fiv = Button(self, text="5")
        fiv.grid(row=3, column=1)
        six = Button(self, text="6")
        six.grid(row=3, column=2)
        mul = Button(self, text="*")
        mul.grid(row=3, column=3)

        one = Button(self, text="1")
        one.grid(row=4, column=0)
        two = Button(self, text="2")
        two.grid(row=4, column=1)
        thr = Button(self, text="3")
        thr.grid(row=4, column=2)
        mns = Button(self, text="-")
        mns.grid(row=4, column=3)

        zer = Button(self, text="0")
        zer.grid(row=5, column=0)
        dot = Button(self, text=".")
        dot.grid(row=5, column=1)
        equ = Button(self, text "=")
        equ.grid(row=5, column=2)
        pls = Button(self, text="+")
        pls.grid(row=5, column=3)

        self.pack()

def main():

    root = Tk()
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, the grid manager organizes twenty buttons and one entry widget.

```

Style().configure("TButton", padding=(0, 5, 0, 5),
    font='serif 10')

```

We configure the `Button` widget to have a specific font and to have some internal padding.

```

self.columnconfigure(0, pad=3)
...
self.rowconfigure(0, pad=3)

```

We use the `columnconfigure()` and the `rowconfigure()` methods to define some space in grid columns and rows. This way we achieve that the buttons are

separated by some space.

```
entry = Entry(self)
entry.grid(row=0, column=0, columnspan=4, sticky=W+E)
```

The `Entry` widget is placed in the first row and column and it spans all four columns. Widgets may not occupy all the space allotted by cells in the grid. The `sticky` parameter expands the widget in a given direction. In our case we ensure that the entry widget is expanded from left to right.

```
cls = Button(self, text="Cls")
cls.grid(row=1, column=0)
```

The `Cls` button is placed in the second row and first column. Note that the rows and columns start from zero.

```
self.pack()
```

The `pack()` method shows the frame widget and gives it initial size. If no other parameters are given, the size will be just enough to show all children. This method packs the frame widget to the toplevel root window, which is also a container. The grid geometry manager is used to organize buttons inside the frame widget.

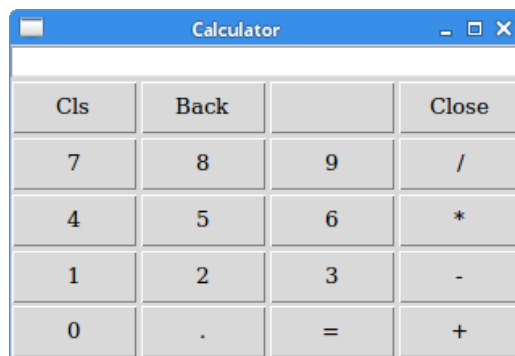


Figure 2.7: Calculator layout

Figure 2.7 shows the layout of a calculator.

2.8 New folder with grid

The following example presents a New folder layout created with the grid manager.

Listing 2.8: new_folder_grid.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book
```

In this script, we create a New folder example with the grid manager.

Author: Jan Bodnar
 Last modified: December 2015
 Website: www.zetcode.com
 """

```
from tkinter import Tk, Text, BOTH, E, W, S, N
from tkinter.ttk import Frame, Button, Label, Entry

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("New folder")
        self.pack(fill=BOTH, expand=True)

        self.columnconfigure(1, weight=1)
        self.rowconfigure(1, weight=1)
        self.rowconfigure(0, pad=10)

        lbl = Label(self, text="Name:")
        lbl.grid(row=0, column=0, padx=5)

        entry = Entry(self)
        entry.grid(row=0, column=1, columnspan=4, padx=5, sticky=W+E)

        txt = Text(self, width=20, height=10)
        txt.grid(row=1, column=0, columnspan=5, padx=5, pady=5,
                sticky=E+W+N+S)

        okBtn = Button(self, text="OK")
        okBtn.grid(row=3, column=3, sticky=E)
        closeBtn = Button(self, text="Close")
        closeBtn.grid(row=3, column=4, padx=5, sticky=E)

def main():

    root = Tk()
    root.geometry("330x300+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

The layout consists of one label, one entry, one text widget, and two buttons.

```
self.columnconfigure(1, weight=1)
self.rowconfigure(1, weight=1)
```

These two lines make the second column and row take all the extra space. This way the text widget occupies the bulk area of the window.

```
lbl = Label(self, text="Name:")
lbl.grid(row=0, column=0, padx=5)
```

The label widget is placed in the top-left cell of the grid. The `padx` option puts some horizontal space after the label.

```
entry = Entry(self)
entry.grid(row=0, column=1, columnspan=4, padx=5, sticky=W+E)
```

The entry widget goes next to the label. With the `columnspan` option, we span the cell in which the widget resides across four columns. The `sticky` parameter makes the entry horizontally fill the enlarged cell.

```
txt = Text(self, width=20, height=10)
txt.grid(row=1, column=0, columnspan=5, padx=5, pady=5,
        sticky=E+W+N+S)
```

The text widgets goes into the second row. It spans five columns and is stretched within its cell in all sides. The stretching is done with the `sticky` parameter. The `width` and `height` parameters specify the size of the text widget in characters, measured according to the current font size. The `weight` option set previously makes it grow and shrink horizontally and vertically.

```
okBtn = Button(self, text="OK")
okBtn.grid(row=3, column=3, sticky=E)
closeBtn = Button(self, text="Close")
closeBtn.grid(row=3, column=4, padx=5, sticky=E)
```

The two buttons go below the text widget, into the right side of the window.

2.9 Windows with grid

The following example presents a Windows layout created with the grid manager.

Listing 2.9: windows_grid.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we use the grid
manager to create a Windows layout.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Text, BOTH, W, N, E, S
from tkinter.ttk import Frame, Button, Label, Style
```

```

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Windows")
        self.pack(fill=BOTH, expand=True)

        self.columnconfigure(1, weight=1)
        self.columnconfigure(3, pad=7)
        self.rowconfigure(3, weight=1)
        self.rowconfigure(5, pad=7)

        lbl = Label(self, text="Windows")
        lbl.grid(sticky=W, pady=4, padx=5)

        area = Text(self)
        area.grid(row=1, column=0, columnspan=2, rowspan=4,
                  padx=5, sticky=E+W+S+N)

        abtn = Button(self, text="Activate")
        abtn.grid(row=1, column=3)

        cbtn = Button(self, text="Close")
        cbtn.grid(row=2, column=3, pady=4)

        hbtn = Button(self, text="Help")
        hbtn.grid(row=5, column=0, padx=5)

        obtn = Button(self, text="OK")
        obtn.grid(row=5, column=3)

def main():

    root = Tk()
    root.geometry("350x300+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In this example, we use a label widget, a text widget, and four buttons.

```

self.columnconfigure(1, weight=1)
self.columnconfigure(3, pad=7)
self.rowconfigure(3, weight=1)
self.rowconfigure(5, pad=7)

```

We define some space among widgets in the grid. The `weight` parameter makes the second column and fourth row growable. This row and column is occupied

by the text widget, so all the extra space is taken by it.

```
lbl = Label(self, text="Windows")
lbl.grid(sticky=W, pady=4, padx=5)
```

The label widget is created and put into the grid. If no column and row is specified, then the first column or row is assumed. The label sticks to the west and it has some padding around its borders.

```
area = Text(self)
area.grid(row=1, column=0, columnspan=2, rowspan=4,
          padx=5, sticky=E+W+S+N)
```

The text widget is created and starts from the second row and first column. It spans two columns and four rows. There is a 4 px space between the widget and the left border of the root window. Finally, the widget sticks to all the four sides. So when the window is resized, the text widget grows in all directions.

```
abtn = Button(self, text="Activate")
abtn.grid(row=1, column=3)

cbtn = Button(self, text="Close")
cbtn.grid(row=2, column=3, pady=4)
```

These two buttons go next to the text widget.

```
hbtn = Button(self, text="Help")
hbtn.grid(row=5, column=0, padx=5)

obtn = Button(self, text="OK")
obtn.grid(row=5, column=3)
```

These two buttons go below the text widget; the Help button takes the first column, the Ok Button takes the last column.

Chapter 3

Events

GUI applications are event-driven. An application reacts to different event types which are generated during its lifetime. Events are generated by a user (a mouse click), an application (a timer), or the system (a clock).

Event is some occurrence that an application needs to know about. *Event source* is the object whose state changes; it generates events. *Event object* encapsulates the state changes of the event source. It is the single argument to the callback methods. *Event handler* is a function that is called in reaction to an event. *Event binding* is attaching an event handler to the triggered event. It is essential that the event handlers are fast, because while they are executing, no other GUI is being updated. Lengthy tasks must be placed in different processes.

Events come in two flavours: low-level events and semantic events. Mouse events and key events are examples of low-level events. Button action events or item selections belong to semantic events. Semantic events are called *virtual events* in Tkinter.

3.1 The command parameter

`Button` and `Checkbutton` widgets have a `command` parameter which invokes a provided function. It is a shortcut to the event binding process.

Listing 3.1: commands.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This program presents the command
parameter.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""
```

```

from tkinter import Tk, BOTH, LEFT
from tkinter.ttk import Frame, Button, Checkbutton

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Commands")
        self.pack(fill=BOTH, expand=True)

        btn = Button(self, text="Button",
                     command=self.onButton1Click)
        btn.pack(side=LEFT, padx=15)

        cb = Checkbutton(self, text="Checkbutton",
                         command=self.onButton2Click)
        cb.pack(side=LEFT)

    def onButton1Click(self):

        print("Push Button clicked")

    def onButton2Click(self):

        print("Checkbutton clicked")

def main():

    root = Tk()
    root.geometry("250x150+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, `Button` and `Checkbutton` widgets use the `command` parameter to execute a function.

```

cb = Checkbutton(self, text="Checkbutton",
                 command=self.onButton2Click)

```

The `Checkbutton`'s `command` parameter points to the `onButton2Click()` method.

3.2 Binding events

The `bind()` method attaches an event handler to an event.

Listing 3.2: bind_method.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This program binds the Esc key
to a quit action.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame
from tkinter import messagebox

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("The bind() method")
        self.pack(fill=BOTH, expand=True)

        self.parent.bind("<Escape>", self.quitApp)

    def quitApp(self, e):

        ret = messagebox.askquestion("Question",
                                     "Are you sure to quit?", default=messagebox.NO)

        if (ret == "yes"):
            self.quit()
        else:
            return

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

In the example, we bind the Esc key to the `quitApp()`. The method shows a confirmation dialog.

```
self.parent.bind("<Escape>", self.quitApp)
```

The `<Escape>` event is triggered when the Esc key is pressed. The event is bound to the `quitApp()` method.

```
def quitApp(self, e):
    ret = messagebox.askquestion("Question",
                                "Are you sure to quit?", default=messagebox.NO)

    if (ret == "yes"):
        self.quit()
    else:
        return
```

The `quitApp()` method shows a dialog with Yes and No buttons. Clicking the Yes button the application is terminated with the `quit()` method.

3.3 Unbinding events

The `unbind()` method removes the specified event handler from the event. It takes the name of the event as a parameter.

Listing 3.3: `bind_unbind.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script shows how to bind and unbind
a specific event.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, BooleanVar
from tkinter.ttk import Frame, Button, Checkbutton

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Binding, unbinding event")
        self.pack(fill=BOTH, expand=True)

        btn = Button(self, text="Click")
        btn.grid(row=0, column=0, padx=10, pady=10)
```

```

self.var = BooleanVar()
cb = Checkbutton(self, text="Bind event", variable=self.var,
                 command=lambda : self.onBind(btn))
cb.grid(row=0, column=1)

def onBind(self, w):

    if (self.var.get() == True):
        w.bind("<Button-1>", self.onClick)
    else:
        w.unbind("<Button-1>")

def onClick(self, e):

    print("clicked")

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

There are two widgets in the example: a `Button` and a `Checkbutton`. The check button binds and unbinds the `<Button-1>` event of the button.

```

cb = Checkbutton(self, text="Bind event", variable=self.var,
                 command=lambda : self.onBind(btn))

```

The `lambda` expression creates a small anonymous function. It allows us to pass a reference to the button.

```

def onBind(self, w):

    if (self.var.get() == True):
        w.bind("<Button-1>", self.onClick)
    else:
        w.unbind("<Button-1>")

```

Depending on the state of the check button, we bind or unbind the `<Button-1>` event.

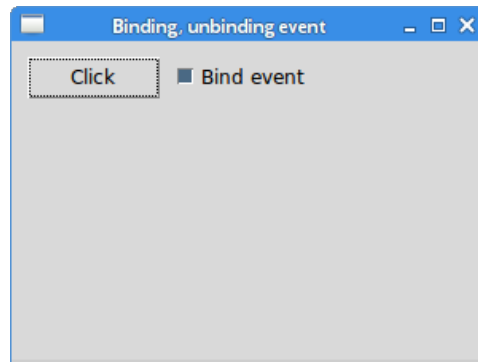


Figure 3.1: Unbinding event

Figure 3.1 shows a selected check box. Clicking on the button prints a message to the terminal.

3.4 Multiple event handlers

The `bind()` method has an `add` parameter which determines whether the event handler is replacing any existing event handlers. It can take either “+” or “-”. The first value adds an event handler to the list of functions bound with the event, the second value replaces existing event handlers with the new event handler.

Listing 3.4: multiple_callbacks.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script plugs two event handlers
to a Button's event.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Button

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):
```

```

        self.parent.title("Button")
        self.pack(fill=BOTH, expand=True)

        btn = Button(self, text="Button")
        btn.grid(row=0, column=0, padx=10, pady=10)
        btn.bind("<Button-1>", self.onClick)
        btn.bind("<Button-1>", self.onClick2, add="+")

    def onClick(self, e):

        print("onClick() method called")

    def onClick2(self, e):

        print("onClick2() method called")

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we plug two event handlers to a `<Button-1>` event. A single button click executes two methods.

```
btn.bind("<Button-1>", self.onClick)
```

The first event handler is bound to the `<Button-1>` event.

```
btn.bind("<Button-1>", self.onClick2, add="+")
```

Second event handler is bound to the `<Button-1>` event. Both methods are called when the button is pressed.

3.5 Event object

Event object is a standard Python object instance with a number of attributes describing the event. The event object is specific to the event type.

Listing 3.5: mouse_motion.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script tracks mouse motion events.

```

It writes the x and y coordinates of
a mouse pointer into a label.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

```
from tkinter import Tk, BOTH, StringVar
from tkinter.ttk import Frame, Label

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Mouse motion event")
        self.pack(fill=BOTH, expand=True)

        x = 0
        y = 0

        text = "x:{0}, y:{1}".format(x, y)
        self.lblv = StringVar()
        self.lblv.set(text)

        lbl = Label(self, textvariable=self.lblv)
        lbl.grid(row=0, column=0, padx=10, pady=10)

        self.bind("<Motion>", self.onMotion)

    def updateLabel(self, x, y):

        val = "x:{0}, y:{1}".format(x, y)
        self.lblv.set(val)

    def onMotion(self, e):

        x = e.x
        y = e.y
        self.updateLabel(x, y)

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

In the example, we examine a mouse motion event. It is an event triggered when we move a mouse pointer. From the event object we determine the x and y coordinates of the mouse pointer and write them to a label widget.

```
self.bind("<Motion>", self.onMotion)
```

We bind the `<Motion>` event to the `onMotion()` method.

```
def onMotion(self, e):
    x = e.x
    y = e.y
    self.updateLabel(x, y)
```

The second parameter of the `onMotion()` method is the event object. We determine the x and y coordinates and pass them to the `updateLabel()` method. The coordinates track the distance of the mouse pointer from the upper-left corner of the application window.

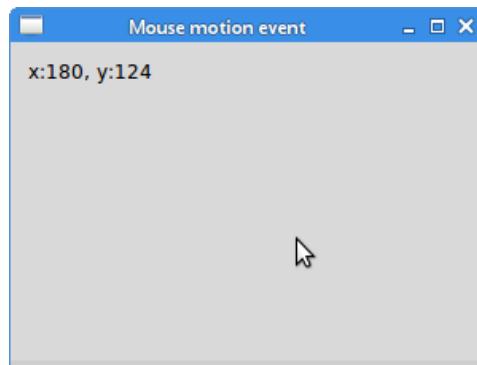


Figure 3.2: Mouse motion event

Figure 3.2 shows the coordinates of the mouse pointer in the upper-left corner of the window. The coordinates are relative to the application window.

3.6 Event source

Event source is the object that generates an event.

Listing 3.6: event_source.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script identifies the event sources.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
```

```
"""

from tkinter import Tk, BOTH, StringVar, W, E
from tkinter.ttk import Frame, Button, Label, Style

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Event source")
        self.pack(fill=BOTH, expand=True)

        s = Style()
        s.configure("Stat.TLabel", padding=5)

        self.grid_columnconfigure(2, weight=1)
        self.grid_rowconfigure(2, weight=1)

        okBtn = Button(self, text="OK", name="okbtn")
        okBtn.grid(row=0, column=0, padx=15, pady=15)
        okBtn.bind("<Button-1>", self.onClick)

        applyBtn = Button(self, text="Apply", name="applybtn")
        applyBtn.bind("<Button-1>", self.onClick)
        applyBtn.grid(row=0, column=1)

        self.svar = StringVar()

        lbl = Label(self, style="Stat.TLabel", textvariable=self.svar)
        lbl.grid(row=5, column=0, columnspan=3, sticky=W+E)

    def onClick(self, e):

        w = e.widget

        if (w._name == "okbtn"):
            self.svar.set("OK button clicked")
        elif (w._name == "applybtn"):
            self.svar.set("Apply button clicked")

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

In the example, we have two buttons and a label. The label serves as a statusbar. Both buttons launch the same event handler. Inside the handler, we identify which button was pressed.

```
okBtn = Button(self, text="OK", name="okbtn")
```

With the `name` parameter, we give a widget a name. It is used to identify the event source.

```
okBtn.bind("<Button-1>", self.onClick)
...
applyBtn.bind("<Button-1>", self.onClick)
```

Both buttons have the same event handler.

```
def onClick(self, e):
    w = e.widget
    ...
```

From the event object, we identify the widget—the event source.

```
if (w._name == "okbtn"):
    self.svar.set("OK button clicked")
elif (w._name == "applybtn"):
    self.svar.set("Apply button clicked")
```

We get the widget's name and set it to the statusbar.

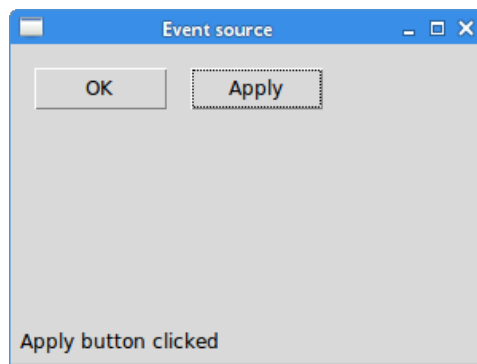


Figure 3.3: Event source

Figure 3.3 shows a message in the statusbar. It says that the Apply button is the source of the event.

3.7 Binding widget class

So far, we have seen individual widgets being attached to event handlers. Tkinter allows to attach an event handler to a widget class. For this, we use the `bind_class()` method.

Listing 3.7: `class_bind.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script binds a TButton class to
an event handler.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH
from tkinter.ttk import Frame, Button

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Class bind")
        self.pack(fill=BOTH, expand=True)

        btn1 = Button(self, text="Button 1", name="btn1",
            command=self.onButton1Click)
        btn1.grid(row=0, column=0, pady=5, padx=5)

        btn2 = Button(self, text="Button 2", name="btn2",
            command=self.onButton2Click)
        btn2.grid(row=0, column=1)

        btn3 = Button(self, text="Button 3", name="btn3",
            command=self.onButton3Click)
        btn3.grid(row=0, column=2, padx=5)

        self.bind_class("TButton", "<Button-1>", self.onButtonsClick)

    def onButtonsClick(self, e):

        print(e.widget)

    def onButton1Click(self):

        print("Doing task 1")

    def onButton2Click(self):

        print("Doing task 2")
```

```

def onButton3Click(self):

    print("Doing task 3")

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example attaches an event handler to a button class. A click on a button gives two messages: an individual message and a class message.

```

btn1 = Button(self, text="Button 1", name="btn1",
              command=self.onButton1Click)

```

Each of the buttons has its own single event handler.

```

self.bind_class("TButton", "<Button-1>", self.onButtonsClick)

```

The `bind_class()` method binds the `TButton` class to the `onButtonsClick()` event handler.

3.8 Custom event

Custom events are triggered with the `event_generate()` method. A custom event is a high-level event; these events are called *virtual events* in Tkinter. Virtual event names are placed between two angle brackets.

Listing 3.8: custom_event.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script creates a custom event.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, StringVar
from tkinter.ttk import Frame, Button, Label
from random import randint

class MyRandom(object):

    def __init__(self):

```

```

        self.r = 0

    def generate(self):

        self.r = randint(0, 900)

    def getRandom(self):

        return self.r

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.mr = MyRandom()
        self.initUI()

    def initUI(self):

        self.parent.title("Custom event")
        self.pack(fill=BOTH, expand=True)

        self.parent.bind("<<Random>>", self.updateLabel)

        btn = Button(self, text="Random", command=self.onClick)
        btn.grid(row=0, column=0, padx=10, pady=10)

        self.lvar = StringVar()

        lbl = Label(self, textvariable=self.lvar)
        lbl.grid(row=0, column=1, padx=50)

    def onClick(self):

        self.mr.generate()
        self.event_generate('<<Random>>')

    def updateLabel(self, e):

        self.lvar.set(self.mr.getRandom())

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example has a button which generates a random number. Each time a number is generated, a custom event called <<Random>> is triggered. The <<Random>>

event is bound to the `updateLabel()` method, which displays the number in the label widget.

```
class MyRandom(object):

    def __init__(self):
        self.r = 0

    def generate(self):

        self.r = randint(0, 900)

    def getRandom(self):

        return self.r
```

The `MyRandom` object generates a random number. Its `getRandom()` method returns the generated random number.

```
self.parent.bind("<<Random>>", self.updateLabel)
```

The custom `<<Random>>` event is bound to the `updateLabel()` method.

```
def onClick(self):

    self.mr.generate()
    self.event_generate('<<Random>>')
```

The `onClick()` method generates a random number. The `event_generate()` method triggers the `<<Random>>` event. It would be more efficient to send the random number with the event as its attribute; however, due to a programming overlook, Tkinter does not support this.¹

```
def updateLabel(self, e):

    self.lvar.set(self.mr.getRandom())
```

In the `updateLabel()` method, we retrieve the random number with the help of the `getRandom()` method and set it to the label's control variable.

3.9 Protocols

Protocols are interactions between the application and the window manager. The `WM_DELETE_WINDOW` protocol defines what happens when the user explicitly closes a window using the window manager. The `protocol()` method installs a handler for the given protocol.

Listing 3.9: protocol.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book
```

¹stackoverflow.com/a/23195921/2008247

This script presents the WM_DELETE_WINDOW protocol handler.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

```
from tkinter import Tk, BOTH
from tkinter.ttk import Frame
from tkinter import messagebox

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.parent.protocol("WM_DELETE_WINDOW", self.onDeleteWindow)

        self.initUI()

    def initUI(self):

        self.parent.title("Protocol")
        self.pack(fill=BOTH, expand=True)

    def onDeleteWindow(self):

        ret = messagebox.askquestion(title="Question",
                                     message="Are you sure to quit?", default=messagebox.NO)

        if (ret == "yes"):
            self.quit()
        else:
            return

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

The example shows a confirmation dialog when the application is terminated by clicking on the Close button of the titlebar.

```
self.parent.protocol("WM_DELETE_WINDOW", self.onDeleteWindow)
```

We plug the onDeleteWindow() method to the WM_DELETE_WINDOW protocol.

3.10 Animation

The `after()` method registers a callback that is invoked after given delay. The delay is expressed in milliseconds. The method can be used to create animations.

Listing 3.10: animation.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script creates a simple animation.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from PIL import Image, ImageTk
from tkinter import Tk, BOTH, CENTER
from tkinter.ttk import Frame, Label

DELAY = 850

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Animation")
        self.pack(fill=BOTH, expand=True)

        self.i = 0
        self.img_names = ("one.png", "two.png", "three.png",
                           "four.png", "five.png", "six.png", "seven.png",
                           "eight.png", "nine.png")

        img = Image.open(self.img_names[self.i])
        num = ImageTk.PhotoImage(img)
        self.label = Label(self, image=num)
        self.label.pack(pady=30)

        # reference must be stored
        self.label.image = num

        self.after(DELAY, self.doCycle)

    def doCycle(self):

        self.i += 1

        if (self.i >= 9):
```

```

        return

        img = ImageTk.PhotoImage(Image.open(self.img_names[self.i]))
        self.label.configure(image=img)
        self.label.image = img

        self.after(DELAY, self.doCycle)

def main():

    root = Tk()
    root.geometry("300x200+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example plays a simple animation. The animation is a sequence of nine numbers.

```
self.i = 0
```

The `i` variable is a counter.

```

self.img_names = ("one.png", "two.png", "three.png",
                  "four.png", "five.png", "six.png", "seven.png",
                  "eight.png", "nine.png")

```

We have nine PNG images representing numbers.

```

img = Image.open(self.img_names[self.i])
num = ImageTk.PhotoImage(img)
self.label = Label(self, image=num)
self.label.pack(pady=30)

```

The first number is displayed on the window. A `PhotoImage` class is used to create an image object.

```
self.after(DELAY, self.doCycle)
```

We register the `doCycle()` callback, called after `DELAY` milliseconds.

```

def doCycle(self):

    self.i += 1

    img = ImageTk.PhotoImage(Image.open(self.img_names[self.i]))
    self.label.configure(image=img)
    self.label.image = img
    ...

```

The counter is increased and the next image is displayed.

```

if (self.i >= 9):
    return

```

The animation stops after showing nine numbers.

```
self.after(DELAY, self.doCycle)
```

The `after()` does not repeatedly call its callback; therefore, we need to register a new callback inside the callback.

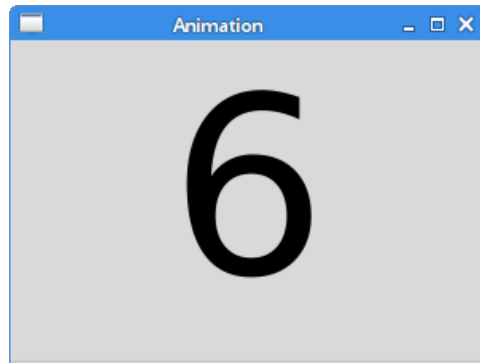


Figure 3.4: Animation

Figure 3.4 shows a still picture from the animation.

3.11 Floating window

The following example creates a floating window without a titlebar. We work with mouse events.

Listing 3.11: floating_window.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from tkinter import Tk, Toplevel, BitmapImage
from tkinter.ttk import Label

"""
ZetCode Tkinter e-book

This script creates a floating window.
The window has a grip by which we can
drag it and move it.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

BITMAP = """
#define grip_width 15
#define grip_height 29
static unsigned char grip_bits[] = {
    0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55,
    0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00,
    0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55,
}
```

```

0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00,
0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55,
0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55 };
"""

class Example(Toplevel):

    def __init__(self, parent):
        Toplevel.__init__(self, parent)

        self.initUI()

    def initUI(self):

        self.overrideredirect(True)

        bitmap = BitmapImage(data=BITMAP)

        msg = "Click on the grip to move\nRight click to terminate"
        self.label = Label(self, text=msg)
        self.grip = Label(self, image=bitmap)
        self.grip.image=bitmap

        self.grip.pack(side="left", fill="y")
        self.label.pack(side="right", fill="both", padx=3, expand=True)

        self.grip.bind("<ButtonPress-1>", self.startMove)
        self.grip.bind("<ButtonRelease-1>", self.stopMove)
        self.grip.bind("<B1-Motion>", self.onMotion)

        self.bind("<ButtonPress-3>", self.onRightClick)

        self.geometry("+300+300")

    def startMove(self, e):

        self.x = e.x
        self.y = e.y

    def onMotion(self, e):

        deltax = e.x - self.x
        deltay = e.y - self.y
        x = self.winfo_x() + deltax
        y = self.winfo_y() + deltay
        self.geometry("+%s+%s" % (x, y))

    def stopMove(self, e):

        e.x = None
        e.y = None

    def onRightClick(self, e):

        self.quit()

```

```
def main():

    root = Tk()
    app = Example(root)
    root.withdraw()
    root.mainloop()

if __name__ == '__main__':
    main()
```

The example shows a toplevel window without a titlebar. The window is moved by its grip located in the left corner. We move it by left clicking on the grip and moving the mouse. A right click closes the window.

```
BITMAP = """
#define grip_width 15
#define grip_height 29
static unsigned char grip_bits[] = {
    0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55,
    0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00,
    0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55,
    0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00,
    0x55, 0x55, 0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55,
    0x00, 0x00, 0x55, 0x55, 0x00, 0x00, 0x55, 0x55 };
"""
```

A grip is a bitmap image. This bitmap is a plain text binary image format.

```
self.overrideRedirect(True)
```

The `overrideRedirect()` method removes the window decoration.

```
bitmap = BitmapImage(data=BITMAP)
```

A `BitmapImage` is created from the data.

```
msg = "Click on the grip to move\nRight click to terminate"
self.label = Label(self, text=msg)
self.grip = Label(self, image=bitmap)
self.grip.image=bitmap
```

The toplevel window contains two labels: one label holds a bitmap and the other one a text message.

```
self.grip.bind("<ButtonPress-1>", self.startMove)
self.grip.bind("<ButtonRelease-1>", self.stopMove)
self.grip.bind("<B1-Motion>", self.onMotion)
```

In order to be able to drag a window, we need to track mouse press, mouse release, and mouse motion events.

```
self.bind("<ButtonPress-3>", self.onRightClick)
```

In addition, we listen for the right mouse button event.

```
def startMove(self, e):
```

```
self.x = e.x
self.y = e.y
```

In the `startMove()` method, we store the x and y coordinates of the window at the time of a left mouse click.

```
def onMotion(self, e):

    deltax = e.x - self.x
    deltay = e.y - self.y
    x = self.winfo_x() + deltax
    y = self.winfo_y() + deltay
    self.geometry("+%s+%s" % (x, y))
```

In the `onMotion()` method, we react to mouse motion events. We compute the delta values of the mouse pointer movement and reposition the window accordingly.

```
root.withdraw()
```

The `withdraw()` method removes the root window from the screen without destroying it.

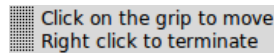


Figure 3.5: Floating window

Figure 3.5 shows how the floating window looks like. The left part of the window is the grip which is used to move the window.

3.12 Splash screen

A splash screen is a window that is shown at the start of more complex applications, which take longer time to load. With a splash screen the user is informed that the application has already started.

Listing 3.12: splash_screen.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from tkinter import Tk, Toplevel, PhotoImage, BOTH
from tkinter.ttk import Frame, Label, Button

"""
ZetCode Tkinter e-book

This script creates a splash screen before
displaying the main application window.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""
```

```

class MySplash(Toplevel):

    def __init__(self, parent):
        Toplevel.__init__(self)

        self.delay = 3000
        self.parent = parent
        self.initUI()

    def initUI(self):

        self.splash_image = PhotoImage(file="splash.png")
        w = self.splash_image.width()
        h = self.splash_image.height()

        self.overrideredirect(True)
        x = (self.parent.winfo_screenwidth() - w) / 2
        y = (self.parent.winfo_screenheight() - h) / 2
        self.geometry('{0}x{1}+{2}+{3}'.format(w, h, int(x), int(y)))

        self.splash_label = Label(self, image=self.splash_image)
        self.splash_label.pack()

        self.after(self.delay, self.close)

    def close(self):

        self.parent.build_app()
        self.destroy()

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.pack(fill=BOTH, expand=True)
        self.parent.title('Application')
        self.parent.withdraw()
        self.showSplashScreen()

    def build_app(self):

        self.quitButton = Button(self, text='Quit', command=self.quit)
        self.quitButton.pack(padx=10, pady=10)

        self.parent.deiconify()

    def showSplashScreen(self):

        MySplash(self)

```

```
def main():

    root = Tk()
    root.geometry('250x200+300+300')
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

A splash screen contains a PNG image which is displayed before the main application window.

```
class MySplash(Toplevel):

    def __init__(self, parent):
        Toplevel.__init__(self)

        self.delay = 3000
        self.parent = parent
        self.initUI()
    ...
```

A splash screen is a `Toplevel` window. It displays a PNG image in a `Label` and has its decoration removed.

```
def initUI(self):

    self.splash_image = PhotoImage(file="splash.png")
    w = self.splash_image.width()
    h = self.splash_image.height()
    ...
```

A `PhotoImage` is created. Its parameter is the name of the image file. We determine the image's size. We need to know the size of the image in order to centre it on the screen.

```
self.overrideRedirect(True)
```

The `overrideRedirect()` method removes the decoration of the `Toplevel` window.

```
x = (self.parent.winfo_screenwidth() - w) / 2
y = (self.parent.winfo_screenheight() - h) / 2
self.geometry('{0}x{1}+{2}+{3}'.format(w, h, int(x), int(y)))
```

The window is placed in the centre of the screen.

```
self.splash_label = Label(self, image=self.splash_image)
self.splash_label.pack()
```

The splash image is placed on the label.

```
self.after(self.delay, self.close)
```


With the `after()` method, we create a single-shot timer. After `delay` ms, we call the `close()` method.

```
def close(self):  
    self.parent.build_app()  
    self.destroy()
```

Inside the `close()` method, we call the parent's `build_app()` method and destroy the splash window.

```
self.parent.withdraw()  
self.showSplashScreen()
```

The root window is displayed first. With the `withdraw()` method we remove it from the screen and create and show the splash screen window.

```
def build_app(self):  
    self.quitButton = Button(self, text='Quit', command=self.quit)  
    self.quitButton.pack(padx=10, pady=10)  
  
    self.parent.deiconify()
```

At the end of the splash screen's lifetime, the `build_app()` method is invoked. The GUI of the main application is built. The `deiconify()` method shows the main window, removed with the `withdraw()` method, back on the screen.

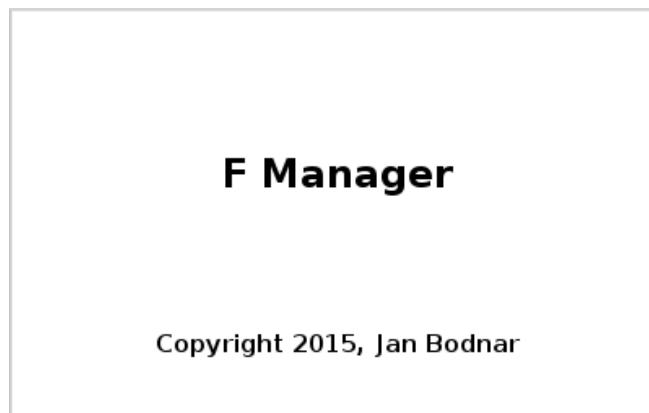


Figure 3.6: Splash screen

Figure 3.6 shows the splash screen.

3.13 Notifications

The following program shows small notification windows. It works with these two events: `<Button-1>` and `<ButtonRelease-1>`.

Listing 3.13: notify.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script creates notification windows
from mouse press and mouse release events.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, Toplevel
from tkinter.ttk import Frame, Label, Style
from tkinter import messagebox
import time

class NotifyWindow(Toplevel):

    def __init__(self, parent, elapsed, screen_x, screen_y):
        Toplevel.__init__(self, parent, background="sky blue")

        self.overridereDIRECT(True)
        self.el = elapsed
        self.screen_x = screen_x
        self.screen_y = screen_y

        self.initUI()

    def initUI(self):

        s = Style()
        s.configure('TLabel', background='sky blue')

        lbl = Label(self, text=str(self.el) + " ms")
        lbl.pack(padx=5, pady=5)
        self.update()
        h = self.winfo_height()
        self.geometry('+{0}+{1}'.format(self.screen_x,
                                         self.screen_y-h))

        self.after(self.el, self.destroyWin)

    def destroyWin(self):

        self.destroy()

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()
```

```

def initUI(self):

    self.parent.title("Notify")
    self.pack(fill=BOTH, expand=True)

    self.bind("<Button-1>", self.onPress)
    self.bind("<ButtonRelease-1>", self.onRelease)

def onPress(self, e):

    self.start = time.time()

def onRelease(self, e):

    screen_x = e.x_root
    screen_y = e.y_root

    self.stop = time.time()

    el = self.stop - self.start
    el_ms = el * 1000
    el_r = round(el_ms)
    self.createNotificationWindow(el_r, screen_x, screen_y)

def createNotificationWindow(self, elapsed, screen_x, screen_y):

    nwin = NotifyWindow(self, elapsed, screen_x, screen_y)

def main():

    root = Tk()
    root.geometry("350x250+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

The program tracks the time between button press and button release events. The elapsed time is shown in a small notification window. The window is displayed just above the mouse pointer and its duration reflects the time of a mouse click. (A mouse click consists of a mouse press and mouse release actions.)

```

class NotifyWindow(Toplevel):

    def __init__(self, parent, elapsed, screen_x, screen_y):
        Toplevel.__init__(self, parent, background="sky blue")

        self.overrideredirect(True)
        self.el = elapsed
        self.screen_x = screen_x
        self.screen_y = screen_y

```

```

        self.initUI()
    ...

```

A notification window is a `Toplevel` window, whose decoration was removed with the `overrideredirect()` method. Its background is sky blue. The duration of a mouse click is one of its passed parameters.

```

lbl = Label(self, text=str(self.el) + " ms")
lbl.pack(padx=5, pady=5)

```

The elapsed time is shown in a label widget.

```

self.update()
h = self.winfo_height()

```

With the `winfo_height()` method we get the height of the notification window. Since the method is called in its constructor, the object creation is not yet finished and we do not have the dimensions set. Therefore, we call the `update()` method before calling the `winfo_height()` method.

```

self.geometry('{0}{1}'.format(self.screen_x,
                               self.screen_y-h))

```

This line positions the notification window just above the mouse pointer.

```

self.after(self.el, self.destroyWin)

```

After some delay, we call the `destroyWin()` method, which destroys the notification window. The delay is equal to the duration of a mouse click that triggered the notification window.

```

self.bind("<Button-1>", self.onPress)
self.bind("<ButtonRelease-1>", self.onRelease)

```

In the main application, we listen to mouse press and mouse release events.

```

def onPress(self, e):
    self.start = time.time()

```

In the `onPress()` method, we store the start time of the mouse click.

```

def onRelease(self, e):
    screen_x = e.x_root
    screen_y = e.y_root

    self.stop = time.time()

    el = self.stop - self.start
    el_ms = el * 1000
    el_r = round(el_ms)
    self.createNotificationWindow(el_r, screen_x, screen_y)

```

In the `onRelease()` method, we compute the elapsed time between a mouse press and a mouse release. We also get the x and y screen coordinates of a mouse pointer. These values are passed to the `createNotificationWindow()` method,

which shows the notification window displaying the elapsed time.

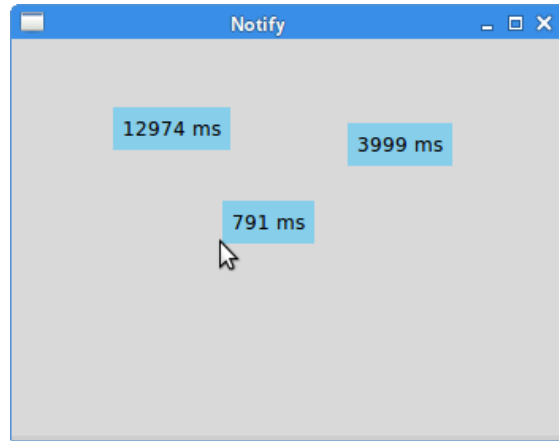


Figure 3.7: Notify

Figure 3.7 shows small, blue notifications which display the elapsed time of recent mouse clicks.

Chapter 4

Widgets

Widgets are basic building blocks of a GUI application. Over the years, several widgets became a standard in all toolkits on all OS platforms; for example a button, a check box, or a scroll bar. Some of them might have different names. For instance, a check box is called a check button in Tkinter. Tkinter has a small set of widgets which cover basic programming needs.

4.1 Button

Button is a rectangular widget used to perform an action. The action is triggered by a mouse click. **Button** displays text or an image.

Listing 4.1: `initUI()` from `button.py`

```
def initUI(self):  
  
    self.parent.title("Button")  
    self.pack(fill=BOTH, expand=1)  
  
    quitButton = Button(self, text="Quit",  
                        command=self.quit)  
    quitButton.pack(anchor=N+W, padx=25, pady=25)
```

In the example we show one button on the window. Clicking on the button terminates the applications.

```
quitButton = Button(self, text="Quit",  
                    command=self.quit)
```

A **Button** widget is created. The `text` option specifies the string displayed by the button. The `command` option points to the function or method which is invoked when the button is pressed. The `quit()` method ends the application's mainloop.

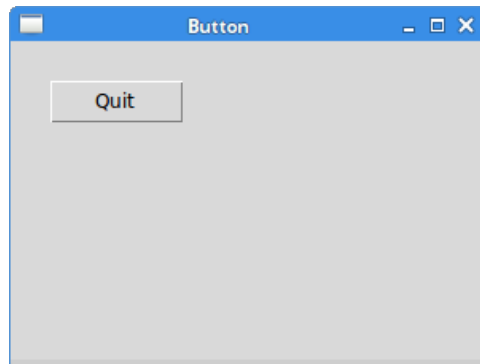


Figure 4.1: Button

Figure 4.1 shows a `Button` widget in the top-left area of the window.

4.2 Label

`Label` is a standard widget used to display a text or image.

Listing 4.2: `initUI()` from `label.py`

```
def initUI(self):  
    self.parent.title("Label")  
    self.pack(fill=BOTH, expand=1)  
  
    self.label = Label(self,  
        text="This is the end of the world as we know it.")  
    self.label.pack(pady=15)
```

In this example, we display text on the window.

```
self.label = Label(self,  
    text="This is the end of the world as we know it.")
```

An instance of a `Label` widget is created. The `text` option specifies the text to be displayed by the label.

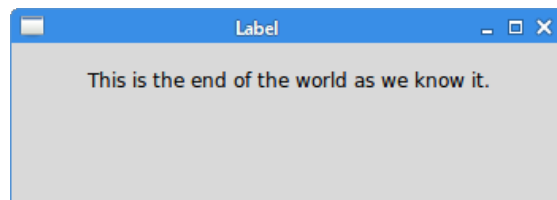


Figure 4.2: Label

Figure 4.2 shows a `Label` in the upper area of the window.

In the next example, we use a label to display an image.

Listing 4.3: label2.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we use the Label
widget to show an image.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from PIL import Image, ImageTk
from tkinter import Tk
from tkinter.ttk import Frame, Label

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

        self.initUI()

    def initUI(self):

        self.parent.title("Label")

        self.img = Image.open("tatra.jpg")
        tatra = ImageTk.PhotoImage(self.img)
        label = Label(self, image=tatra)

        # reference must be stored
        label.image = tatra

        label.pack()
        self.pack()

    def setGeometry(self):

        w, h = self.img.size
        self.parent.geometry("(%dx%d+300+300" % (w, h))

def main():

    root = Tk()
    ex = Example(root)
    ex.setGeometry()
    root.mainloop()

if __name__ == '__main__':
    main()
```


The example shows an image on the window.

```
from PIL import Image, ImageTk
```

By default, the `Label` widget can display only a limited set of image types. To display a JPG image, we must use the Python Imaging Library (PIL) module.

```
self.img = Image.open("tatra.jpg")  
tatra = ImageTk.PhotoImage(self.img)
```

We create an image from the image file in the current working directory. Later we create a photo image from the image.

```
label = Label(self, image=tatra)
```

The photoimage is given to the `image` option of the label widget.

```
label.image = tatra
```

In order not to be garbage collected, the image reference must be stored.

```
def setGeometry(self):  
    w, h = self.img.size  
    self.parent.geometry("%dx%d+300+300" % (w, h))
```

We make the size of the window to exactly fit the image size.

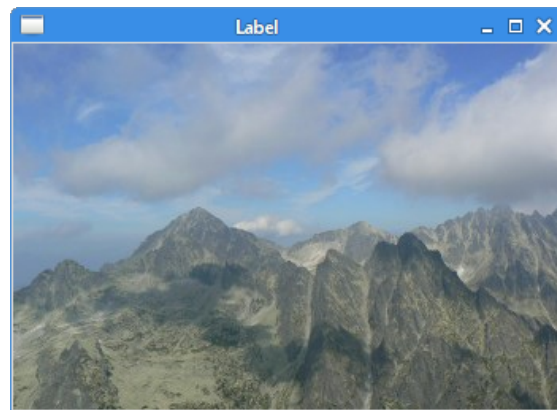


Figure 4.3: Label

Figure 4.3 shows an image on the window.

4.3 Message

`Message` is a variant of the `Label`, designed to display multiline messages.

Listing 4.4: `initUI()` from `message.py`

```
def initUI(self):

    self.parent.title("Message")
    self.pack(fill=BOTH, expand=1)

    quote = """You have within you right now everything
you need to deal with whatever the world can throw at \
you.\nBrian Tracy"""

    msg = Message(self, text=quote)
    msg.config(bg='lightblue', font=('sans', 20, 'italic'))
    msg.pack()
```

The example shows a quote from Brian Tracy on the window. The text is displayed using bigger italic font.

```
msg = Message(self, text=quote)
```

A `Message` widget is created. The quote is passed to the `text` option.

```
msg.config(bg='lightblue', font=('sans', 20, 'italic'))
```

With the `config()` method, we change the background colour and set a specific font for the text.

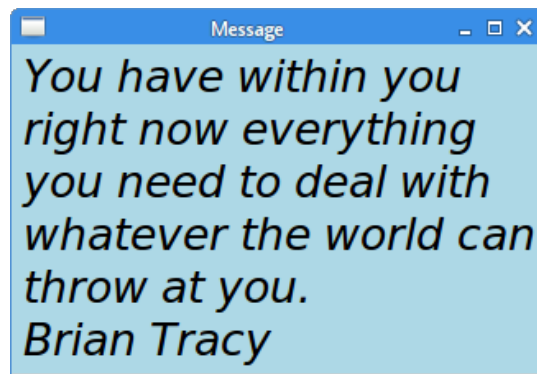


Figure 4.4: Message

Figure 4.4 shows a multiline text.

4.4 Separator

`Separator` is a horizontal or vertical bar that separates other widgets.

Listing 4.5: `initUI()` from `separator.py`

```
def initUI(self):

    self.parent.title("Separator")
    self.pack(fill=BOTH, expand=1)
```

```

quote = """You have within you right now everything
you need to deal with whatever the world can throw at you.
Brian Tracy"""

msg = Message(self, text=quote)
msg.config(bg='lightgreen', font=('sans', 20, 'italic'))
msg.pack(padx=5, pady=10)

sep = Separator(self, orient=HORIZONTAL)
sep.pack(fill=X, padx=25, expand=1)

quote2 = """It is better to light one small candle than to
curse the darkness.\nEleanor Roosevelt"""

msg2 = Message(self, text=quote2)
msg2.config(bg='lightgreen', font=('sans', 20, 'italic'))
msg2.pack(padx=5, pady=10)

```

The example uses a `Separator` to visually divide two `Message` widgets.

```
sep = Separator(self, orient=HORIZONTAL)
```

A horizontal `Separator` is created.

```
sep.pack(fill=X, padx=25, expand=1)
```

We use the `pack` manager to expand the widget horizontally.

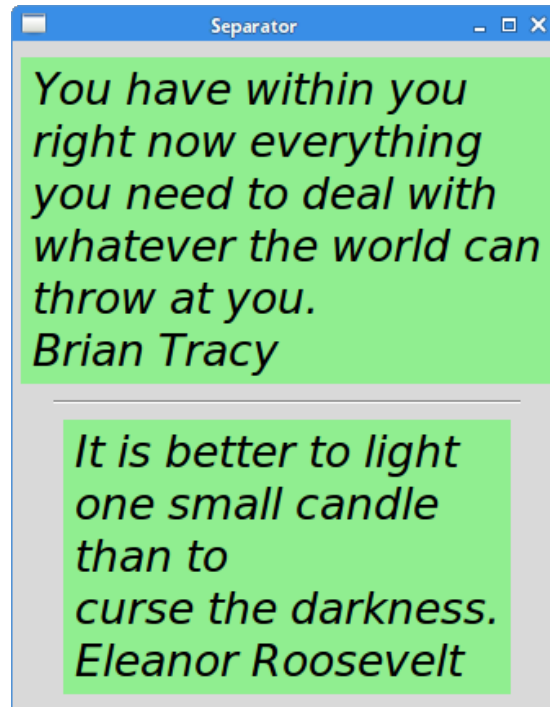


Figure 4.5: Separator

Figure 4.5 shows two messages divided by a horizontal bar.

4.5 Frame

Frame is a basic container widget. It serves as a parent widget and is used mainly to group other widgets into more complex layouts.

Listing 4.6: `initUI()` from `frames.py`

```
def initUI(self):

    self.style = Style()
    self.style.theme_use("alt")

    self.parent.title("Frames")

    self.pack(fill=BOTH, expand=1)
    self.grid(padx=5, pady=5)

    frame1 = Frame(self, width=100, height=100, relief=SUNKEN)
    frame1.grid(row=0, column=0, padx=5, pady=5)

    frame2 = Frame(self, width=100, height=100, relief=RAISED)
    frame2.grid(row=0, column=1)

    frame3 = Frame(self, width=100, height=100, relief=GROOVE)
    frame3.grid(row=1, column=0)

    frame4 = Frame(self, width=100, height=100, relief=RIDGE)
    frame4.grid(row=1, column=1)
```

In the example, we show four frames. Each of them has a different decoration.

```
self.style = Style()
self.style.theme_use("alt")
```

The default theme does not support all reliefs; therefore, we choose the `alt` theme which does.

```
self.grid(padx=5, pady=5)
```

A grid geometry manager is used to organize the four frames.

```
frame1 = Frame(self, width=100, height=100, relief=SUNKEN)
```

A **Frame** widget is created. The `relief` is set to `SUNKEN`. The frames are empty by default and in order to visually separate them, we choose different reliefs.

```
frame1.grid(row=0, column=0, padx=5, pady=5)
```

The first frame is set to the first row and column of the grid manager.

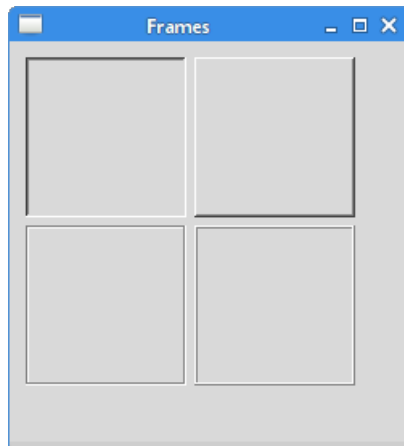


Figure 4.6: Frames

Figure 4.6 shows four `Frame` widgets on the window.

4.6 LabelFrame

`LabelFrame` is a frame widget which draws a border around its child widgets. It is used to group a number of related widgets.

Listing 4.7: `initUI()` from `labelframe.py`

```
def initUI(self):
    self.parent.title("LabelFrame")
    self.pack(fill=BOTH, expand=True)

    labFrame = LabelFrame(self, text="Personal info")

    nameLbl = Label(labFrame, text="Name:")
    nameLbl.grid(row=0, column=0, padx=10, pady=10, sticky=E)
    nameEntry = Entry(labFrame)
    nameEntry.grid(row=0, column=1)

    addrLbl = Label(labFrame, text="Address:")
    addrLbl.grid(row=1, column=0, padx=10, sticky=E)
    addrEntry = Entry(labFrame)
    addrEntry.grid(row=1, column=1)

    labFrame.pack(anchor=N+W, ipadx=20, ipady=20, padx=15, pady=15)
```

In the example, there are two labels and two entry widgets inside a `LabelFrame`.

```
labFrame = LabelFrame(self, text="Personal info")
```

A `LabelFrame` widget is created. Its `text` option specifies the description of the frame.

```
nameEntry = Entry(labFrame)
```

The children take the `LabelFrame` as its parent widget.

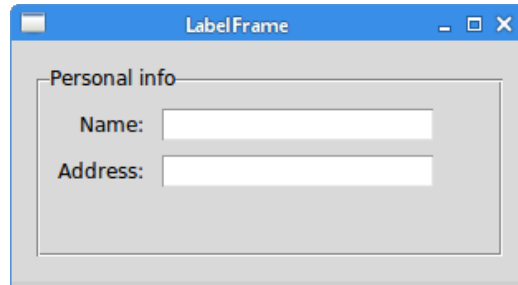


Figure 4.7: `LabelFrame`

Figure 4.7 shows four widgets inside a `LabelFrame`.

4.7 Checkbutton

`Checkbutton` is a widget that has two states: on and off. It is used to denote some boolean property. The `Checkbutton` widget provides a check box with a text label. Each `Checkbutton` widget should be associated with a variable.

Listing 4.8: `checkboxbutton.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This program toggles the title of the
window with the Checkbutton widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BooleanVar, BOTH
from tkinter.ttk import Frame, Checkbutton

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

        self.initUI()

    def initUI(self):

        self.parent.title("Checkbutton")
        self.pack(fill=BOTH, expand=True)
```

```
self.var = BooleanVar()

cb = Checkbutton(self, text="Show title",
                 variable=self.var, command=self.onClick)
self.var.set(True)
cb.place(x=50, y=50)

def onClick(self):

    if self.var.get() == True:
        self.master.title("Checkbutton")
    else:
        self.master.title("")

def main():

    root = Tk()
    root.geometry("250x150+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

A single `Checkbutton` is placed on the window. It shows or hides the title of the window.

```
self.var = BooleanVar()
```

A `BooleanVar` is created. It is a value holder for Boolean values for widgets in Tkinter.

```
cb = Checkbutton(self, text="Show title",
                 variable=self.var, command=self.onClick)
```

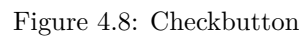
An instance of the `Checkbutton` is created. The value holder is connected to the widget via the `variable` parameter. When we click on the check button, the `onClick()` method is called. This is done with the `command` parameter.

```
self.var.set(True)
```

Initially, the title is shown in the titlebar. Therefore, we make it selected by calling the `set()` method on the associated variable.

```
cb.place(x=50, y=50)
```

The widget is located on the window with the `place()` method at the given coordinates.



4.8 Radiobutton

Listing 4.9: radiobutton.py

[illegible]


```

radio1.pack(anchor="nw", padx=10, pady=15)
radio2 = Radiobutton(self, text="Female", variable=self.rvar,
                     value=2, command=self.onRadioSelect)

radio2.pack(padx=10, anchor="nw")

self.lvar = StringVar()
self.lvar.set("...")
lbl = Label(self, textvariable=self.lvar)
lbl.pack(pady=35, anchor="s")

def onRadioSelect(self):

    val = self.rvar.get()

    if (val == 1):
        self.lvar.set("Male")
    else:
        self.lvar.set("Female")

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x200+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example we have two `Radiobutton` widgets. The text of the selected radio button is displayed in a label widget.

```
self.rvar = IntVar()
```

An `IntVar` is created. This variable is shared by both `Radiobuttons`.

```
radio1 = Radiobutton(self, text="Male", variable=self.rvar,
                     value=1, command=self.onRadioSelect)
```

The first `Radiobutton` is created. The `value` parameter specifies the value associated with the radio button. It is later used to tell between the radio buttons.

```
self.lvar = StringVar()
self.lvar.set("...")
lbl = Label(self, textvariable=self.lvar)
```

This is the `Label` widget which displays the text of the currently selected radio button.

```
def onRadioSelect(self):

    val = self.rvar.get()

    if (val == 1):

```

```

        self.lvar.set("Male")
    else:
        self.lvar.set("Female")

```

In the `onRadioSelect()` method we find out the currently selected radio button. We query the radio button's value. The label widget is updated with its `set()` method.

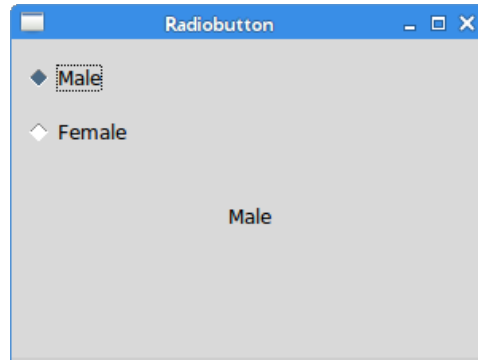


Figure 4.9: Radiobutton

Figure 4.9 shows two `Radiobutton` widgets on the window. The selected radio button's value is displayed in the label.

4.9 Entry

`Entry` allows the user to enter or edit a single line of plain text.

Listing 4.10: entry.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, the text entered
in the Entry widget is shown in
the Label.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, LEFT, BOTH, StringVar
from tkinter.ttk import Entry, Frame, Label

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

```

```

        self.initUI()

    def initUI(self):

        self.parent.title("Entry")
        self.pack(fill=BOTH, expand=1)

        evar = StringVar()
        evar.trace("w", self.onChangeed)
        self.entry = Entry(self, textvariable=evar)
        self.entry.pack(side=LEFT, padx=15)

        self.lvar = StringVar()
        self.lvar.set("...")
        lbl = Label(self, text="...", textvariable=self.lvar)
        lbl.pack(side=LEFT)

    def onChangeed(self, a, b, c):

        self.lvar.set(self.entry.get())

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("250x100+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example shows the entered text in an adjacent label. The label is updated immediately as the letters are typed.

```

evar = StringVar()
evar.trace("w", self.onChangeed)
self.entry = Entry(self, textvariable=evar)

```

An `Entry` is created with an associated control variable. The changes to the widget are followed with the `trace()` method.

```

def onChangeed(self, a, b, c):

    self.lvar.set(self.entry.get())

```

In the `onChangeed()` method, we update the label's control variable. (The parameters of the method are not important for us.) The `Entry`'s `get()` method retrieves the entered data.

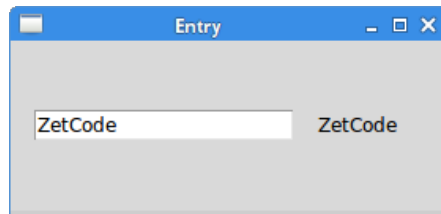


Figure 4.10: Entry

Figure 4.10 shows the `Entry` widget. The nearby label reflects the content of the `Entry` widget.

The following example explores a few methods of the `Entry` widget.

Listing 4.11: entry2.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, the text entered
in the Entry widget is shown in
the Label.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, W, END, NORMAL, DISABLED, BooleanVar
from tkinter.ttk import Entry, Frame, Button, Checkbutton

class Example(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Entry")
        self.pack(fill=BOTH, expand=True)

        self.entry = Entry(self)
        self.entry.grid(row=0, column=0, columnspan=2,
                        padx=10, pady=15, sticky=W)

        clearBtn = Button(self, text="Clear", command=self.onClear)
        clearBtn.grid(row=1, column=0, pady=15, padx=10, sticky=W)

        selBtn = Button(self, text="Select all",
                        command=self.onSelectAll)
        selBtn.grid(row=1, column=1, sticky=W)
```

```

deselBtn = Button(self, text="Deselect",
                  command=self.onDeselect)
deselBtn.grid(row=1, column=2, padx=10)

self.chvar = BooleanVar()
self.chvar.set(False)

checkBtn = Checkbutton(self, text="readonly",
                       variable=self.chvar, command=self.onChangeReadability)
checkBtn.grid(row=2, column=0)

def onClear(self):

    self.entry.delete(0, END)

def onSelectAll(self):

    self.entry.select_range(0, END)

def onDeselect(self):

    self.entry.select_clear()

def onChangeReadability(self):

    if self.chvar.get() == True:
        self.entry.config(state=DISABLED)
    else:
        self.entry.config(state=NORMAL)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x140+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we have an `Entry` widget and four buttons. The buttons clear the text, select and deselect the text, and disable the widget, making it read-only.

```

def onClear(self):

    self.entry.delete(0, END)

```

The `delete()` method clears the text. The `END` flag corresponds to the position just after the last character in the entry.

```

def onSelectAll(self):

    self.entry.select_range(0, END)

```

The `select_range()` method selects the specified range of the entry's text.

```
def onDeselect(self):
    self.entry.select_clear()
```

The `select_clear()` method clears the selection.

```
def onChangeReadability(self):
    if self.chvar.get() == True:
        self.entry.config(state=DISABLED)
    else:
        self.entry.config(state=NORMAL)
```

We make the `Entry` widget read-only by changing its state to `DISABLED`.

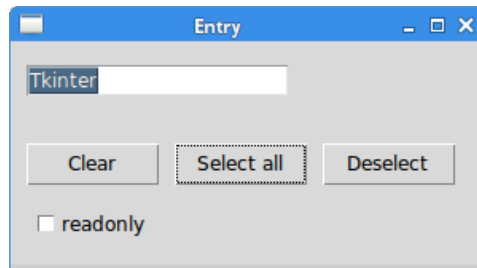


Figure 4.11: Entry 2

Figure 4.11 shows selected text of the `Entry` widget.

4.10 Scale

`Scale` is a widget that lets the user graphically select a value by sliding a knob within a bounded interval. A `Scale` can be horizontal or vertical. The programmer can set the minimum and maximum value of the `Scale` and set its resolution.

Listing 4.12: scale.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we show how to
use the Scale widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, IntVar, LEFT
from tkinter.ttk import Frame, Label, Scale
```

```

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Scale")
        self.pack(fill=BOTH, expand=1)

        scale = Scale(self, from_=0, to=100, command=self.onScale)
        scale.pack(side=LEFT, padx=15)

        self.var = IntVar()
        self.label = Label(self, text=0, textvariable=self.var)
        self.label.pack(side=LEFT)

    def onScale(self, val):

        v = int(float(val))
        self.var.set(v)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("250x100+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

We have two widgets in the above script: a scale and a label. A value from the scale widget is shown in the label widget.

```
scale = Scale(self, from_=0, to=100, command=self.onScale)
```

A Scale widget is created. We provide the lower and upper bounds. The from is a regular Python keyword that is why there is an underscore. When we move the knob of the scale, the `onScale()` method is called.

```
self.var = IntVar()
self.label = Label(self, text=0, textvariable=self.var)
```

An integer value holder and label widget are created. The value from the holder is shown in the label widget.

```
def onScale(self, val):

    v = int(float(val))

```

```
self.var.set(v)
```

The `onScale()` method receives a currently selected value from the scale widget as a parameter. The value is first converted to a float and then to integer. Finally, the value is set to the value holder of the label widget.

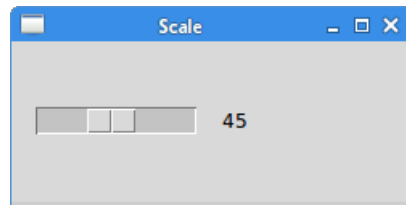


Figure 4.12: Scale

Figure 4.12 shows a label with a value chosen by the adjacent scale widget.

4.11 Spinbox

`Spinbox` allows the user to choose a value by clicking the up/down buttons or pressing up/down on the keyboard to increase/decrease the value currently displayed. The user can also type the value in manually.

Listing 4.13: `spinbox.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we show how to
use the Scale widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Spinbox, BOTH, IntVar, LEFT
from tkinter.ttk import Frame, Label

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Spinbox")
```



```

        self.pack(fill=BOTH, expand=1)

        self.sbox = Spinbox(self, from_=0, to=100,
                             command=self.onClick)
        self.sbox.pack(side=LEFT, padx=15)

        self.var = IntVar()
        self.label = Label(self, text=0, textvariable=self.var)
        self.label.pack(side=LEFT)

    def onClick(self):

        val = self.sbox.get()
        self.var.set(val)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("250x100+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, the selected value from the `Spinbox` is shown in the label.

```

self.sbox = Spinbox(self, from_=0, to=100,
                    command=self.onClick)

```

A `Spinbox` widget is created. We set its range with the `from_` and `to` parameters.

```

def onClick(self):

    val = self.sbox.get()
    self.var.set(val)

```

We retrieve the currently selected `Spinbox`'s value with the `get()` method and set it to the label widget with the `set()` method of its associated value holder.

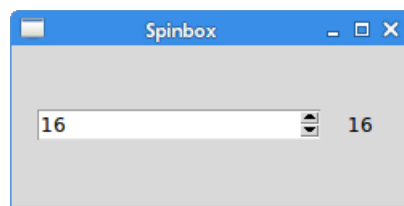


Figure 4.13: Spinbox

Figure 4.13 shows the selected value of a spin box in the adjacent label.

4.12 OptionMenu

OptionMenu offers a fixed set of choices to the user in a drop-down menu.

Listing 4.14: optionmenu.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we present the
OptionMenu widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, LEFT, BOTH, StringVar
from tkinter.ttk import Frame, Style, Label, OptionMenu

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("OptionMenu")
        self.pack(fill=BOTH, expand=1)

        self.ovar = StringVar()
        options = ["OpenBSD", "NetBSD", "FreeBSD"]

        om = OptionMenu(self, self.ovar, options[0],
                        *options, command=self.onSelect)
        om.pack(side=LEFT, padx=15)

        self.lvar = StringVar()
        self.lvar.set("OpenBSD")
        lbl = Label(self, textvariable=self.lvar)
        lbl.pack(side=LEFT)

    def onSelect(self, a):

        self.lvar.set(a)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("250x150+300+300")
    root.mainloop()
```

```
if __name__ == '__main__':
    main()
```

In the example, we have an `OptionMenu` widget with three options. The selected option is displayed in the adjacent label.

```
self.ovar = StringVar()
```

The `OptionMenu` widget has an associated control variable.

```
options = ["OpenBSD", "NetBSD", "FreeBSD"]
```

The list groups the values displayed in the drop-down menu of the `OptionMenu` widget.

```
om = OptionMenu(self, self.ovar, options[0],
                 *options, command=self.onSelect)
```

The `OptionMenu` widget is created. The second parameter is the associated control variable, the third is the default value. Then comes the list of values.

```
def onSelect(self, a):
    self.lvar.set(a)
```

In the `onSelect()` method, the currently selected value is set to the control variable of the label.

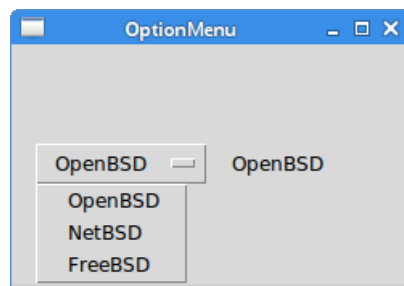


Figure 4.14: `OptionMenu`

Figure 4.14 shows an `OptionMenu` with its drop-down menu.

4.13 Combobox

`Combobox` is a combination of an `Entry` and a drop-down menu. `Combobox` is similar to `OptionMenu`.

Listing 4.15: `combobox.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
```

```
"""
ZetCode Tkinter e-book

In this script, we present the
Combobox widget. A value selected from the
Combobox is shown in a label.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, LEFT, BOTH, StringVar
from tkinter.ttk import Frame, Style, Label, Combobox

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Combobox")
        self.pack(fill=BOTH, expand=1)

        self.cvar = StringVar()

        combo = Combobox(self, textvariable=self.cvar)
        combo.bind("<<ComboboxSelected>>", self.onComboSelect)
        combo['values'] = ('OpenBSD', 'NetBSD', 'FreeBSD')
        combo.current(0)
        combo.pack(side=LEFT, padx=15)

        self.lvar = StringVar()
        self.lvar.set("OpenBSD")
        lbl = Label(self, textvariable=self.lvar)
        lbl.pack(side=LEFT)

    def onComboSelect(self, e):

        w = e.widget
        self.lvar.set(w.get())

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x150+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()
```

In the example, there are two widgets: a `Combobox` and a `Label`. The value selected from the combo box is shown in the label.

```
self.cvar = StringVar()
```

This `StringVar` control variable is associated with the `Combobox`.

```
combo = Combobox(self, textvariable=self.cvar)
```

A `Combobox` widget is created. The `textvariable` option associates the widget with a control variable.

```
combo.bind("<<ComboboxSelected>>", self.onComboSelect)
```

We bind the `<<ComboboxSelected>>` event to the `self.onComboSelect()` method. The `Combobox` widget generates a `<<ComboboxSelected>>` event when the user selects an element from the list of values.

```
combo['values'] = ('OpenBSD', 'NetBSD', 'FreeBSD')
```

We add values to the combo box's drop-down menu.

```
combo.current(0)
```

The `current()` method selects an option, which is initially visible.

```
self.lvar = StringVar()
self.lvar.set("OpenBSD")
lbl = Label(self, textvariable=self.lvar)
```

A label and its associated control variable are created.

```
def onComboSelect(self, e):
    w = e.widget
    self.lvar.set(w.get())
```

The `onComboSelect()` method is fired when the user selects an option from the drop-down menu. From the event object, we get the reference to the event source (our combo box), and retrieve its currently selected value with the `get()` method. The returned value is set to the label's control variable.

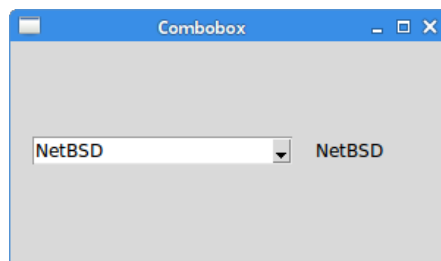


Figure 4.15: Combobox

Figure 4.15 shows a `Combobox` widget. The selected value is shown in the adjacent label.

4.14 Scrollbar

`Scrollbar` provides a scrolling functionality. Sometimes the functional area of a widget is bigger than the physical area represented by a GUI widget. `Scrollbar` gives access to the whole functional area via scrolling. The `Scrollbar` is used to implement scrolled listboxes, canvases, and text fields.

Listing 4.16: scrollbar.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, attach a vertical Scrollbar
to the Canvas widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, RIGHT, LEFT, Y, ALL, Canvas, VERTICAL
from tkinter.ttk import Frame, Scrollbar, Label

FONT_SIZE = 12
YOFFSET = 10
XOFFSET = 20

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Scrollbar")
        self.pack(fill=BOTH, expand=True)

        self.canvas = Canvas(self)

        sbar = Scrollbar(self, orient=VERTICAL,
            command=self.canvas.yview)

        self.canvas.configure(yscrollcommand=sbar.set)

        sbar.pack(side=RIGHT, fill=Y)
        self.canvas.pack(side=LEFT, fill=BOTH, expand=True)

        for i in range(100):
            x = XOFFSET
            y = i*(YOFFSET + FONT_SIZE) + YOFFSET
            self.canvas.create_text(x, y, font=("times", FONT_SIZE),
                text="Item " + str(i))

        self.bind("<Configure>", self.onConfigure)
```

```

def onConfigure(self, e):

    self.canvas.configure(scrollregion=self.canvas.bbox(ALL))

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x150+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, a `Scrollbar` is attached to a `Canvas` widget. The canvas has one hundred text items, which cannot be shown at once.

```
self.canvas = Canvas(self)
```

A `Canvas` widget is created. It is possible to place more items on a `Canvas` than any monitor can display in one application window.

```
sbar = Scrollbar(self, orient=VERTICAL,
                 command=self.canvas.yview)
```

A `Scrollbar` widget is created. The `orient` parameter sets a horizontal orientation for the widget. The `command` parameter specifies the method to be called whenever the scrollbar is moved. The `yview` method of the canvas controls its vertical position.

```
self.canvas.configure(yscrollcommand=sbar.set)
```

The `yscrollcommand` is used to connect a canvas to a vertical scrollbar. The command is called when the view is changed (for example, when new items are added, or the widget is resized).

```

for i in range(100):
    x = XOFFSET
    y = i*(YOFFSET + FONT_SIZE) + YOFFSET
    self.canvas.create_text(x, y, font=("times", FONT_SIZE),
                           text="Item " + str(i))

```

In the for loop, we add one hundred text items to the canvas.

```
self.bind("<Configure>", self.onConfigure)
```

```

def onConfigure(self, e):

    self.canvas.configure(scrollregion=self.canvas.bbox(ALL))

```

It is necessary to provide the size of a scrolling area to the canvas. We bind the `<Configure>` event, which is triggered when the frame is resized, to the

`onConfigure()` method. Inside this method, we set the `scrollregion` to the actual size of the canvas. The `bbox()` method returns a bounding box for the given object. Passing the `ALL` value, we get the bounding box for all items of the canvas.

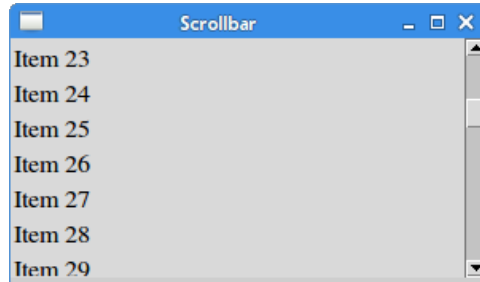


Figure 4.16: Scrollbar

Figure 4.16 shows a `Scrollbar` widget attached to a `Canvas` widget.

4.15 Notebook

`Notebook` is a container widget which consists of tabs. Each time the user clicks on one of these tabs, the widget will display the child pane associated with that tab.

Listing 4.17: `initUI()` from `notebook.py`

```
def initUI(self):

    self.parent.title("Notebook")
    self.pack(fill=BOTH, expand=True)

    notebook = Notebook(self)

    frame1 = Frame()
    frame2 = Frame()
    frame3 = Frame()

    lbl1 = Label(frame1, text="Pane 1")
    lbl1.place(x=30, y=40)

    lbl2 = Label(frame2, text="Pane 2")
    lbl2.place(x=30, y=40)

    lbl3 = Label(frame3, text="Pane 3")
    lbl3.place(x=30, y=40)

    notebook.add(frame1, text="Tab 1")
    notebook.add(frame2, text="Tab 2")
    notebook.add(frame3, text="Tab 3")
    notebook.pack(pady=5, fill=BOTH, expand=True)
```

In the example, we create a `Notebook` widget with three tabs. Each of these tabs is associated with a `Frame` widget. We place one label on each of the frames.


```
notebook = Notebook(self)
```

An instance of the `Notebook` widget is created.

```
frame1 = Frame()
frame2 = Frame()
frame3 = Frame()
```

Three frames are created.

```
lbl1 = Label(frame1, text="Pane 1")
lbl1.place(x=30, y=40)
```

A `Label` widget is created with the first frame as its parent. The `place` manager locates the label on the frame.

```
notebook.add(frame1, text="Tab 1")
```

With the `add()` method, we add a new tab to the notebook. The `text` option specifies the tab's description.

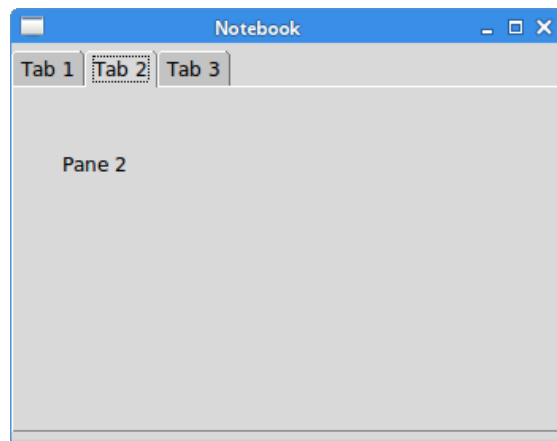


Figure 4.17: Notebook

Figure 4.17 shows a `Notebook` widget with three tabs.

4.16 PanedWindow

`PanedWindow` is a container widget that separates the window into resizable parts. The child widgets can be resized by the user by moving separator lines using the mouse. Note that the themed `PanedWindow` does not support theming of the separator line, the sash. It is expected that the children provide some decorations that will tell them apart.

Listing 4.18: `initUI()` from `panedwindow.py`

```
def initUI(self):
```

```
self.parent.title("PanedWindow")
self.pack(fill=BOTH, expand=True)

pwin = PanedWindow(self, orient=HORIZONTAL)

leftFrame = LabelFrame(pwin, text="Left pane",
                        width=150, height=200)
pwin.add(leftFrame, weight=5)

mdlFrame = LabelFrame(pwin, text="Middle pane", width=150)
pwin.add(mdlFrame, weight=3)

rightFrame = LabelFrame(pwin, text="Right pane", width=150)
pwin.add(rightFrame, weight=1)

pwin.pack(fill=BOTH, expand=True, padx=5, pady=5)
```

The example creates a `PanedWindow` with three panes; the panes are `LabelFrames`. The panes have different weights which make them shrink or grow at a different speed.

```
pwin = PanedWindow(self, orient=HORIZONTAL)
```

A horizontal `PanedWindow` is created. The panes are place in a row.

```
leftFrame = LabelFrame(pwin, text="Left pane",
                        width=150, height=200)
```

A `LabelFrame` widget is created. We put label frames into the paned window. An initial size is provided with the `width` and `height` parameters.

```
pwin.add(leftFrame, weight=5)
```

The `add()` method adds a pane to the paned window. The `weight` parameter sets the proportion of the pane's resizing.

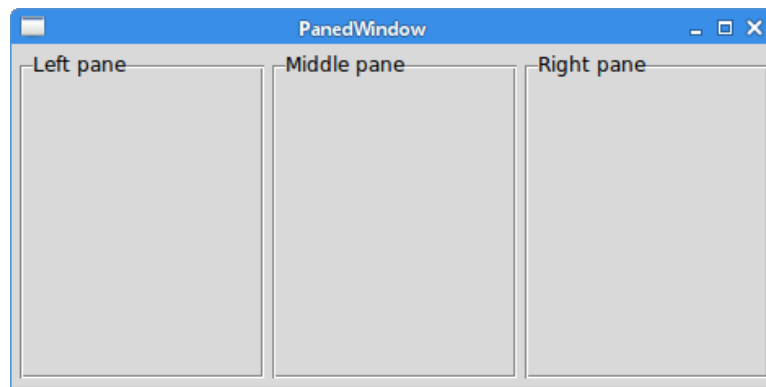


Figure 4.18: `PanedWindow`

Figure 4.18 shows a window separated into three resizable areas.

4.17 Progressbar

`Progressbar` gives the user an indication of the progress of an operation and reassures them that the application is still running.

Listing 4.19: progressbars.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we present the Progressbar
widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, E, W
from tkinter.ttk import Frame, Button, Progressbar

DELAY1 = 30
DELAY2 = 20

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Progressbar")
        self.pack(fill=BOTH, expand=True)

        pb1 = Progressbar(self, mode='determinate', name='pb1')
        pb2 = Progressbar(self, mode='indeterminate', name='pb2')

        startBtn = Button(self, text='Start',
                           command=lambda: self.updateBars('start'))
        stopBtn = Button(self, text='Stop',
                           command=lambda: self.updateBars('stop'))

        pb1.grid(row=0, column=0, columnspan=2, pady=15,
                  padx=10, sticky=W+E)
        pb2.grid(row=1, column=0, columnspan=2, padx=10, sticky=W+E)

        startBtn.grid(row=2, column=0, pady=15, padx=10, sticky=E)
        stopBtn.grid(row=2, column=1, padx=10, sticky=W)

    def updateBars(self, op):

        pb1 = self.nametowidget('pb1')
        pb2 = self.nametowidget('pb2')
```

```

        if op == 'start':
            pb1.start(DELAY1)
            pb2.start(DELAY2)
        else:
            pb1.stop()
            pb2.stop()

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x200+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The application displays two **Progressbar** widgets in two different modes: determinate and indeterminate. Two buttons are used to start and stop the progressbars.

```
pb1 = Progressbar(self, mode='determinate', name='pb1')
```

A **Progressbar** widget is created in the determinate mode. In this mode, an indicator moves from beginning to end of the bar. This mode is used when we know the duration of the task beforehand. Giving a name to the widget with the **name** option enables to refer to the widget by its name.

```
pb2 = Progressbar(self, mode='indeterminate', name='pb2')
```

The second **Progressbar** is created in the indeterminate mode. In this mode, an indicator bounces back and forth between the ends of the widget. This mode is used when we do not know the exact duration of the task.

```

def updateBars(self, op):

    pb1 = self.nametowidget('pb1')
    pb2 = self.nametowidget('pb2')
    ...

```

In the **updateBars()** method, we get the reference to the **Progressbars** with the **nametowidget()** method.

```

if op == 'start':
    pb1.start(DELAY1)
    pb2.start(DELAY2)
else:
    pb1.stop()
    pb2.stop()

```

The **start()** method starts the indicator of a **Progressbar**. Its parameter is the animation interval in milliseconds; the default is 50 ms. The **stop()** method stops the progress.

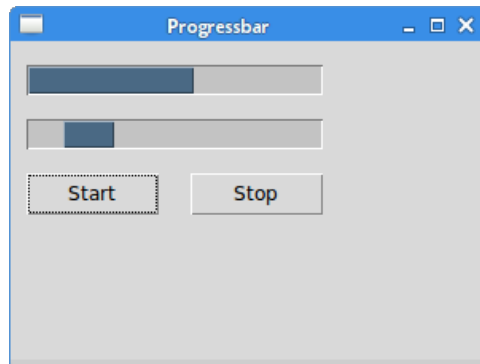


Figure 4.19: Progressbar

Figure 4.19 shows two `Progressbar` widgets controlled with two buttons.

Chapter 5

Menus and toolbars

A menu is a group of commands located in a menubar. A toolbar has buttons with some common commands of the application. In Tkinter, the `Menu` widget is used to create drop-down and popup menus.

5.1 Simple menu

The first example shows a simple menu.

Listing 5.1: `simplemenu.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This program shows a simple menu.
It has one action, which terminates
the program when selected.

Author: Jan Bodar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Menu, BOTH
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Simple menu")
```

```

        self.pack(fill=BOTH, expand=True)

        menubar = Menu(self.parent)
        self.parent.config(menu=menubar)

        fileMenu = Menu(menubar, tearoff=False)
        fileMenu.add_command(label="Exit", command=self.onExit)
        menubar.add_cascade(label="File", menu=fileMenu)

    def onExit(self):

        self.quit()

def main():

    root = Tk()
    root.geometry("300x150+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example shows a menu with one item. By selecting the Exit menu item, we close the application.

```

menubar = (self.parent)
self.parent.config(menu=menubar)

```

A menubar is created. It is a regular `Menu` widget configured to be the menubar of the root window.

```

fileMenu = Menu(menubar, tearoff=False)

```

A File menu object is created. A menu is a drop-down window containing commands. The `tearoff` option disables the possibility to remove the menu from the menubar into a separate window; this is visually represented by a dashed, horizontal line.

```

fileMenu.add_command(label="Exit", command=self.onExit)

```

A command is added to the File menu with the `add_command()` method. The command calls the `onExit()` method.

```

menubar.add_cascade(label="File", menu=fileMenu)

```

The File menu is added to the menubar using the `add_cascade()` method.

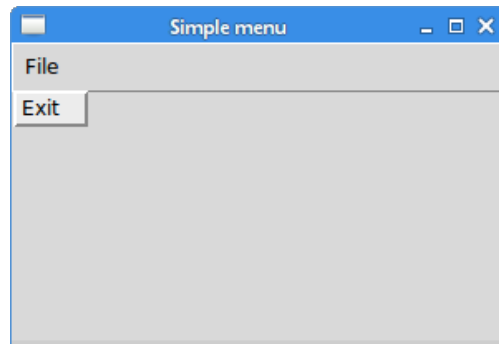


Figure 5.1: Simple menu

Figure 5.1 shows an Exit menu item in the File menu.

5.2 Submenu

A submenu is a menu plugged into another menu object. The next example demonstrates this.

Listing 5.2: submenu.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script we create a submenu
a separator and keyboard shortcuts to menus.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Menu, BOTH
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Submenu")
        self.pack(fill=BOTH, expand=True)

        menubar = Menu(self.parent)
        self.parent.config(menu=menubar)
```



```

fileMenu = Menu(menubar, tearoff=False)

submenu = Menu(fileMenu, tearoff=False)
submenu.add_command(label="New feed")
submenu.add_command(label="Bookmarks")
submenu.add_command(label="Mail")
fileMenu.add_cascade(label='Import', menu=submenu, underline=0)

fileMenu.add_separator()

fileMenu.add_command(label="Exit", underline=0,
                     command=self.onExit)
menubar.add_cascade(label="File", underline=0, menu=fileMenu)

def onExit(self):

    self.quit()

def main():

    root = Tk()
    root.geometry("250x150+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we have three options in a submenu of a File menu. We also create a separator and keyboard shortcuts.

```

submenu = Menu(fileMenu)
submenu.add_command(label="New feed")
submenu.add_command(label="Bookmarks")
submenu.add_command(label="Mail")

```

We have a submenu with three commands. The submenu is a regular menu.

```
fileMenu.add_cascade(label='Import', menu=submenu, underline=0)
```

By adding the menu to the File menu and not to the menubar, we create a submenu. The `underline` parameter creates a keyboard shortcut. It provides the character position which should be underlined. In our case it is the first; the positions start from zero. The shortcuts enable to navigate to menu items with a keyboard. First, we activate the File menu with Alt+F and then we can activate the submenu with Alt+I. The Exit menu is activated with Alt+E.

```
fileMenu.add_separator()
```

A separator is a horizontal line that visually separates menu commands.

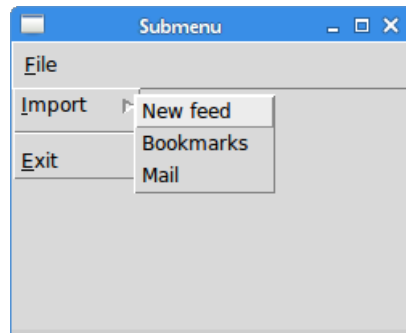


Figure 5.2: Submenu

Figure 5.2 shows an activated submenu with three menu items.

5.3 Popup menu

In the next example, we create a popup menu. A popup menu is also called a context menu. It can be shown anywhere on the client area of a window.

Listing 5.3: popupmenu.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this program, we create
a popup menu.

Author: Jan Bodar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Menu, BOTH
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

        self.initUI()

    def initUI(self):

        self.parent.title("Popup menu")
        self.pack(fill=BOTH, expand=True)

        self.menu = Menu(self.parent, tearoff=False)
```

```

        self.menu.add_command(label="Minimize",
                               command=self.doMinimize)
        self.menu.add_command(label="Exit", command=self.onExit)

        self.parent.bind("<Button-3>", self.showMenu)
        self.pack()

    def showMenu(self, e):

        self.menu.post(e.x_root, e.y_root)

    def doMinimize(self):

        self.parent.iconify()

    def onExit(self):

        self.quit()

def main():

    root = Tk()
    root.geometry("250x150+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we create a popup menu with two commands. One command minimizes the application, the other one terminates it.

```
self.menu = Menu(self.parent, tearoff=False)
```

A context menu is a regular `Menu` widget.

```
self.parent.bind("<Button-3>", self.showMenu)
```

We bind the `<Button-3>` event to the `showMenu()` method. The event is generated when we right click on the client area of the window.

```
def showMenu(self, e):

    self.menu.post(e.x_root, e.y_root)
```

The `showMenu()` method shows the context menu. The popup menu is shown at the x and y coordinates of the mouse click.

```
def doMinimize(self):

    self.parent.iconify()
```

The `iconify()` method minimizes the application.

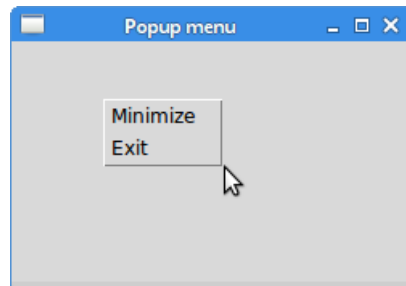


Figure 5.3: Popup menu

Figure 5.3 shows a popup menu showing two menu items.

5.4 Check menu button

It is possible to have a check menu button in a menu object. Such a button is added to a menu with the `add_checkbutton()` method.

Listing 5.4: checkmenubutton.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This program shows or hides a statusbar
with a check menu button.

Author: Jan Bodar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import (Tk, Menu, BooleanVar, StringVar, BOTTOM, X,
                     BOTH, RIDGE)
from tkinter.ttk import Frame, Label, Style

class Example(Frame):
    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Check menu button")
        self.pack(fill=BOTH, expand=True)

        s = Style()
        s.configure("Statusbar.TLabel", borderwidth=2, relief=RIDGE)
```

```

menubar = Menu(self.parent)
self.parent.config(menu=menubar)

fileMenu = Menu(menubar, tearoff=False)
fileMenu.add_command(label="Exit", command=self.onExit)
menubar.add_cascade(label="File", menu=fileMenu)

self.showStat = BooleanVar()
self.showStat.set(True)

viewMenu = Menu(menubar, tearoff=False)
viewMenu.add_checkbutton(label="Show statusbar",
    command=self.onClick, variable=self.showStat, onvalue=True,
    offvalue=False)
menubar.add_cascade(label="View", menu=viewMenu)

self.svar = StringVar()
self.svar.set("Ready")

self.sb = Label(self, textvariable=self.svar,
    style="Statusbar.TLabel")
self.sb.pack(side=BOTTOM, fill=X)

def onClick(self):

    if (self.showStat.get() == True):
        self.sb.pack(side=BOTTOM, fill=X)
    else:
        self.sb.pack_forget()

def onExit(self):

    self.quit()

def main():

    root = Tk()
    root.geometry("300x150+300+300")
    app = Example(root)
    root.mainloop()

if __name__ == '__main__':
    main()

```

With the check menu button, we control the visibility of the application's statusbar.

```

s = Style()
s.configure("Statusbar.TLabel", borderwidth=2, relief=RIDGE)

```

We provide a custom style for the statusbar, which is a `Label` widget.

```

self.showStat = BooleanVar()
self.showStat.set(True)

```

The check menu button is associated with a `BooleanVar`.

```
viewMenu = Menu(menubar, tearoff=False)
viewMenu.add_checkbutton(label="Show statusbar",
    command=self.onClick, variable=self.showStat, onvalue=True,
    offvalue=False)
```

A check button is added to the View menu using the `add_checkbutton()` method.

```
self.sb = Label(self, textvariable=self.svar,
    style="Statusbar.TLabel")
self.sb.pack(side=BOTTOM, fill=X)
```

The `Label` widget serves as a statusbar. With the `style` option, we set the custom style for the label.

```
def onClick(self):
    if (self.showStat.get() == True):
        self.sb.pack(side=BOTTOM, fill=X)
    else:
        self.sb.pack_forget()
```

In the `onClick()` method, we toggle the visibility of the statusbar. The label is removed from the layout with the `pack_forget()` method.

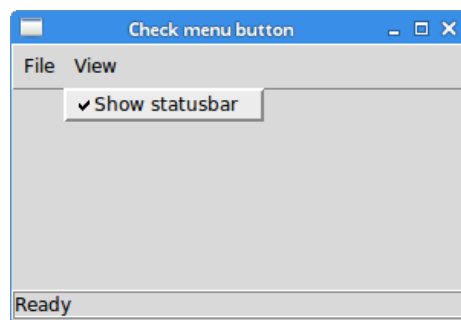


Figure 5.4: Check menu button

Figure 5.4 shows a selected check menu button; the statusbar is visible.

5.5 Toolbar

Menus group commands that we can use in an application. Toolbars provide a quick access to the most frequently used commands. There is no toolbar widget in Tkinter; we use a `Frame` widget to create a toolbar.

Listing 5.5: toolbar.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this program, we create a toolbar.
```

```
Author: Jan Bodar
Last modified: November 2015
Website: www.zetcode.com
"""

from PIL import Image, ImageTk
from tkinter import Tk, Menu, LEFT, TOP, BOTH, X, FLAT, RAISED
from tkinter.ttk import Button, Frame, Style

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

        self.initUI()

    def initUI(self):

        self.parent.title("Toolbar")
        self.pack(fill=BOTH, expand=True)

        s = Style()
        s.configure('TFrame', relief=RAISED)
        s.configure('TButton', relief=FLAT)

        menubar = Menu(self.parent)
        self.fileMenu = Menu(self.parent, tearoff=0)
        self.fileMenu.add_command(label="Exit", command=self.onExit)
        menubar.add_cascade(label="File", menu=self.fileMenu)

        toolbar = Frame(self)

        self.img = Image.open("exit.png")
        eimg = ImageTk.PhotoImage(self.img)

        exitButton = Button(toolbar, image=eimg,
                             command=self.quit)
        exitButton.image = eimg
        exitButton.pack(side=LEFT, padx=2, pady=2)

        toolbar.pack(side=TOP, fill=X)
        self.parent.config(menu=menubar)
        self.pack()

    def onExit(self):

        self.quit()

def main():

    root = Tk()
    root.geometry("300x150+300+300")
    app = Example(root)
    root.mainloop()
```

```
if __name__ == '__main__':  
    main()
```

Our toolbar contains one push button.

```
s = Style()  
s.configure('TFrame', relief=RAISED)  
s.configure('TButton', relief=FLAT)
```

We create a style for the toolbar's frame and button.

```
toolbar = Frame(self)
```

A toolbar is a `Frame` widget.

```
exitButton = Button(toolbar, image=eimg,  
                    command=self.quit)  
exitButton.image = eimg  
exitButton.pack(side=LEFT, padx=2, pady=2)
```

An Exit button is created and placed on the left side of the toolbar.

```
toolbar.pack(side=TOP, fill=X)
```

The toolbar is placed at the top of the window; it is horizontally stretched.

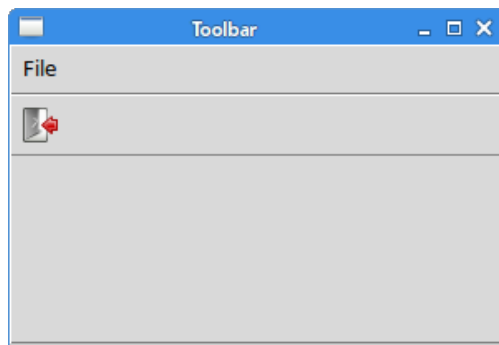


Figure 5.5: Toolbar

Figure 5.5 shows a toolbar with an Exit button.

Chapter 6

Listbox

Listbox is a widget that displays a list of objects. It allows the user to select one or more items.

6.1 Item selection

When a selection of a Listbox has changed, a <<ListboxSelect>> virtual event is generated.

Listing 6.1: listbox__selection.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we display a selected
item from a Listbox in a Label widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, Listbox, StringVar, END
from tkinter.ttk import Frame, Label

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Item selection")
        self.pack(fill=BOTH, expand=True)
```

```

    acts = ['Scarlett Johansson', 'Rachel Weiss',
            'Natalie Portman', 'Jessica Alba']

    lb = Listbox(self)

    for i in acts:
        lb.insert(END, i)

    lb.bind("<<ListboxSelect>>", self.onSelect)

    lb.pack(padx=10, pady=10, fill=BOTH, expand=True)

    self.var = StringVar()
    self.label = Label(self, text=0, textvariable=self.var)
    self.label.pack(pady=10)

    def onSelect(self, val):

        sender = val.widget
        idx = sender.curselection()
        value = sender.get(idx)

        self.var.set(value)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, a `Listbox`'s selected item is displayed in a label widget.

```

acts = ['Scarlett Johansson', 'Rachel Weiss',
        'Natalie Portman', 'Jessica Alba']

```

These items fill the listbox widget.

```
lb = Listbox(self)
```

An instance of a `Listbox` widget is created.

```

for i in acts:
    lb.insert(END, i)

```

We use the `insert()` method to fill the listbox with data.

```
lb.bind("<<ListboxSelect>>", self.onSelect)
```

We listen for the `<<ListboxSelect>>` event.

```
def onSelect(self, val):
```

```

sender = val.widget
idx = sender.curselection()
value = sender.get(idx)

self.var.set(value)

```

The currently selected item's index is retrieved with the `curselection()` method. The index is passed to the `get()` method to fetch the item.

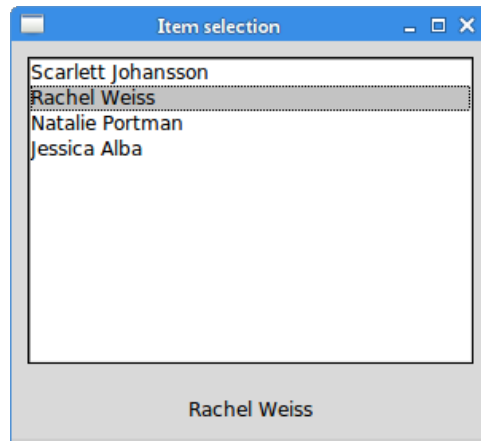


Figure 6.1: Item selection

Figure 6.1 shows a currently selected item in a label located below the listbox.

6.2 Multiple selection

Listbox supports several selection modes; the default is the **BROWSE** mode, where only one item can be selected at a time with a mouse or cursor keys. The **EXTENDED** mode is a multiple selection mode, where items can be selected with Shift and Ctrl keys.

Listing 6.2: multiple_selection.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we print all selected items
of the Listbox to the console.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, Listbox, StringVar, END, EXTENDED
from tkinter.ttk import Frame, Label

```

```

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Multiple selection")
        self.pack(fill=BOTH, expand=True)

        acts = ['Scarlett Johansson', 'Rachel Weiss',
                'Natalie Portman', 'Jessica Alba', 'Meryl Streep']

        lb = Listbox(self, selectmode=EXTENDED)

        for i in acts:
            lb.insert(END, i)

        lb.bind("<<ListboxSelect>>", self.onSelect)

        lb.pack(padx=10, pady=10, fill=BOTH, expand=True)

    def onSelect(self, e):

        sender = e.widget
        idx = sender.curselection()

        for i in idx:
            print(sender.get(i))

        print("*****")

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example prints all selected items to the console.

```
lb = Listbox(self, selectmode=EXTENDED)
```

A Listbox widget is created with selection mode set to EXTENDED.

```

def onSelect(self, e):

    sender = e.widget
    idx = sender.curselection()

    for i in idx:

```

```
print(sender.get(i))

print("*****")
```

The current selection is determined with the `curselection()` method. In the for loop, we go through the returned tuple of selected indexes and print the selected items to the console.

6.3 Attaching a scrollbar

By default, the `ListBox` widget does not have a scrollbar. We have to add the scrollbar manually. The scrollbar is needed when there are many items in a listbox.

Listing 6.3: `listbox_scrollbar.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we attach a Scrollbar
to the ListBox widget.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, ListBox, Y, BOTH, RIGHT, LEFT, END
from tkinter.ttk import Frame, Scrollbar

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Scrollbar")
        self.pack(fill=BOTH, expand=True)

        sbar = Scrollbar(self)
        sbar.pack(side = RIGHT, fill=Y)

        mylist = ListBox(self, yscrollcommand = sbar.set)

        for line in range(100):
            mylist.insert(END, "Item " + str(line))

        mylist.pack(side = LEFT, fill = BOTH, expand=True)
        sbar.config(command = mylist.yview)
```

```
def main():
    root = Tk()
    ex = Example(root)
    root.geometry("250x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()
```

The example glues a `Scrollbar` widget with the listbox.

```
sbar = Scrollbar(self)
sbar.pack(side = RIGHT, fill=Y)
```

An instance of a `Scrollbar` widget is created and packed to the right of the window.

```
mylist = Listbox(self, yscrollcommand = sbar.set)
```

A `Listbox` widget is created. The listbox's `yscrollcommand` points to the scrollbar's `set()` method.

```
for line in range(100):
    mylist.insert(END, "Item " + str(line))
```

We fill the listbox with one hundred items.

```
sbar.config(command = mylist.yview)
```

We make the listbox vertically scrollable.

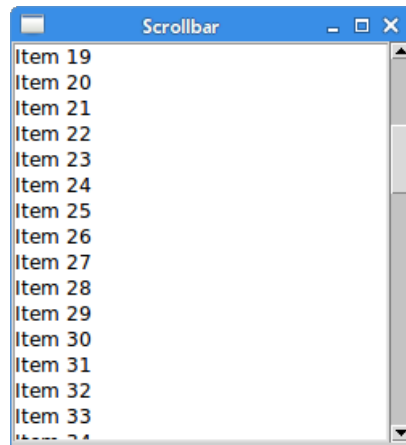


Figure 6.2: Listbox scrollbar

Figure 6.2 shows a listbox with more items than its window can display. Therefore, we need a scrollbar to reach more items of the listbox.

6.4 Adding and removing items

A new item is added to a listbox with the `insert()` method. An item is removed with the `remove()` method.

Listing 6.4: `listbox__modify.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we dynamically add and remove Listbox
items.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, Listbox, BooleanVar, END, E, W, S, N
from tkinter.ttk import Frame, Button, Entry

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Adding, removing items")
        self.pack(fill=BOTH, expand=True)

        self.grid_columnconfigure(2, weight=1)
        self.grid_rowconfigure(1, weight=1)

        self.entry = Entry(self)
        self.entry.grid(row=0, column=0, padx=10)

        addBtn = Button(self, text="Add", command=self.addItem)
        addBtn.grid(row=0, column=1, pady=10)

        self.lbox = Listbox(self)

        self.lbox.grid(row=1, column=0, rowspan=2, columnspan=3,
            padx=10, sticky=E+W+N+S)

        remBtn = Button(self, text="Remove", command=self.removeItem)
        remBtn.grid(row=3, column=0, padx=10, pady=10, sticky=W)

    def addItem(self):

        val = self.entry.get()
```

```

        if (len(val.strip()) == 0):
            return

        self.entry.delete(0, END)
        self.lbox.insert(END, val)

    def removeItem(self):

        idx = self.lbox.curselection()

        if (len(idx) == 0):
            return

        self.lbox.delete(idx, idx)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we have a listbox, an entry, and two buttons. The buttons add and remove items.

```

def addItem(self):

    val = self.entry.get()

    if (len(val.strip()) == 0):
        return

    self.entry.delete(0, END)
    self.lbox.insert(END, val)

```

In the `addItem()` method, we get the value from the entry widget. We ensure that it is not empty and does not contain only white characters. We delete the entry's value and insert the retrieved value into the listbox using the `insert()` method. The value is added at the end of the listbox.

```

def removeItem(self):

    idx = self.lbox.curselection()

    if (len(idx) == 0):
        return

    self.lbox.delete(idx, idx)

```

The `removeItem()` method removes a selected item. Nothing is done if there is no active selection. The item is removed using listbox's `delete()` method.

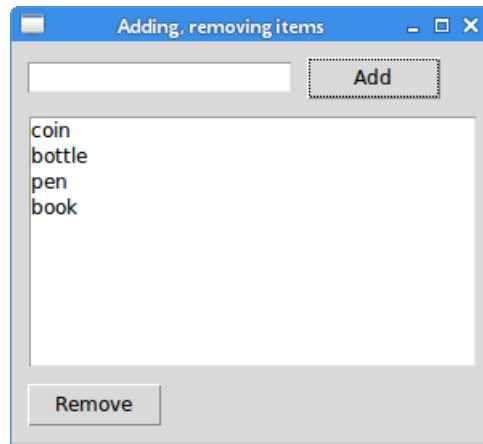


Figure 6.3: Adding and removing items

Figure 6.3 shows four items in a listbox. They were dynamically inserted.

6.5 Sorting items

In the following example, we show how to sort listbox's items.

Listing 6.5: sorting_items.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we sort items of
a Listbox.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

from tkinter import Tk, BOTH, Listbox, BooleanVar, END, LEFT
from tkinter.ttk import Frame, Label, Button, Checkbutton

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Sorting items")
        self.pack(fill=BOTH, expand=True)
```

```

        items = ['flower', 'pen', 'cup', 'chair', 'envelope', 'valet',
                  'coin', 'book', 'paper', 'bottle']

        self.lb = Listbox(self)
        for i in items:
            self.lb.insert(END, i)

        self.lb.pack(padx=10, pady=10, fill=BOTH, expand=True)

        sortBtn = Button(self, text="Sort", command=self.onSortItems)
        sortBtn.pack(side=LEFT, padx=10, pady=5)

        self.var = BooleanVar()
        cb = Checkbutton(self, variable=self.var, text="Ascending")
        cb.pack(side=LEFT)

    def onSortItems(self):

        if (self.var.get() == True):
            rev = False
        else:
            rev = True

        temp_list = list(self.lb.get(0, END))
        sorted_list = sorted(temp_list, reverse = rev)
        self.lb.delete(0, END)

        for item in sorted_list:
            self.lb.insert(END, item)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we have a listbox, a push button, and a check button. Depending on the state of the check button, pressing the push button sorts the listbox's items in ascending or descending order.

```

self.var = BooleanVar()
cb = Checkbutton(self, variable=self.var, text="Ascending")
cb.pack(side=LEFT)

```

The sorting order is determined by the `Checkbutton`.

```

def onSortItems(self):

    if (self.var.get() == True):
        rev = False
    else:
        rev = True

```

...

We get the value from the check button.

```
temp_list = list(self.lb.get(0, END))
```

All the values from the listbox are stored in a temporary list.

```
sorted_list = sorted(temp_list, reverse = rev)
```

The temporary list is sorted with the `sorted()` function. The second parameter is the sorting order.

```
self.lb.delete(0, END)
```

We delete all the items from the listbox.

```
for item in sorted_list:
    self.lb.insert(END, item)
```

New sorted items are added to the listbox.

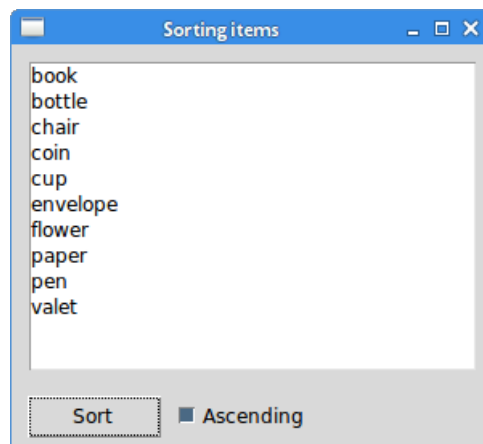


Figure 6.4: Sorting items

Figure 6.4 shows items of a listbox sorted in ascending order.

6.6 Reordering items by dragging

In the following example, we show how listbox items can be reordered by dragging.

Listing 6.6: listbox_dragging.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book
```

In this script, we reorder items by dragging.

Author: Jan Bodnar
Last modified: November 2015
Website: www.zetcode.com
"""

```
from tkinter import Tk, BOTH, Listbox, END
from tkinter.ttk import Frame

class MyList(Listbox):

    def __init__(self, parent):

        Listbox.__init__(self, parent)
        self.bind('<Button-1>', self.setCurrent)
        self.bind('<B1-Motion>', self.moveSelection)

    def setCurrent(self, e):

        self.curIndex = self.nearest(e.y)

    def moveSelection(self, e):

        i = self.nearest(e.y)

        if i < self.curIndex:

            x = self.get(i)
            self.delete(i)
            self.insert(i+1, x)
            self.curIndex = i

        elif i > self.curIndex:

            x = self.get(i)
            self.delete(i)
            self.insert(i-1, x)
            self.curIndex = i

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Reordering items by dragging")
        self.pack(fill=BOTH, expand=True)

        acts = ['Scarlett Johansson', 'Rachel Weiss',
                'Natalie Portman', 'Jessica Alba', 'Meryl Streep']
```

```

        lb = MyList(self)

        for i in acts:
            lb.insert(END, i)

        lb.pack(padx=10, pady=10, fill=BOTH, expand=True)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

To do the dragging, we have to listen to mouse press and mouse motion events.

```

class MyList(Listbox):

    def __init__(self, parent):

        Listbox.__init__(self, parent)
        self.bind('<Button-1>', self.setCurrent)
        self.bind('<B1-Motion>', self.moveSelection)
    ...

```

We create our custom Listbox, where we bind event handlers for <Button-1> and <B1-Motion> events.

```

def setCurrent(self, e):

    self.curIndex = self.nearest(e.y)

```

The `setCurrent()` method receives a mouse press event object. We translate the y coordinate of the mouse pointer to the listbox's index with the `nearest()` method.

```

def moveSelection(self, e):

    i = self.nearest(e.y)

    if i < self.curIndex:

        x = self.get(i)
        self.delete(i)
        self.insert(i+1, x)
        self.curIndex = i

    elif i > self.curIndex:

        x = self.get(i)
        self.delete(i)
        self.insert(i-1, x)
        self.curIndex = i

```

In the `moveSelection()` method, we again translate the y coordinate into the index of the listbox. Depending on the index value, we switch the dragged item with its previous or next neighbour.

Chapter 7

Text

`Text` provides a widget that is used to edit and display both plain and formatted text. The widget also supports embedded images and windows. While the `Entry` allows to edit one line of text, `Text` allows to edit multiple lines of text.

`ScrolledText` is `Text` widget with built-in scrolling functionality.

7.1 Simple example

The following is a simple demonstration of the `Text` widget.

Listing 7.1: `simple_text.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script is a simple demonstration
of the Text widget.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Text, BOTH, END
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Simple example")
```

```
self.pack(fill=BOTH, expand=True)

txt = Text(self)
txt.pack(fill=BOTH, expand=True)

txt.insert(END, "Beyond Sing the Woods.\n")
txt.insert(END, "For Whom the Bell Tolls.\n")
txt.insert(END, "The Whale.\n")
txt.focus()

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x200+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()
```

The example displays a text widget and inserts three lines into it.

```
txt = Text(self)
txt.pack(fill=BOTH, expand=True)
```

A `Text` widget is created.

```
txt.insert(END, "Beyond Sing the Woods.\n")
```

The `insert()` method sets text to the text widget.

```
txt.focus()
```

With the `focus()` method, the text widget receives initial focus.

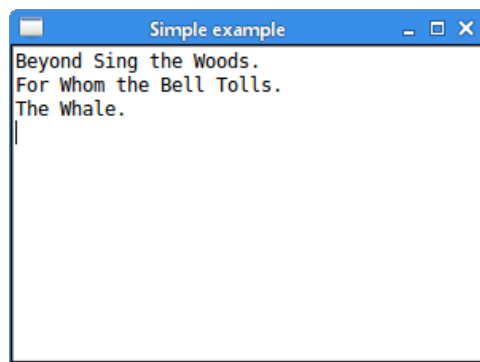


Figure 7.1: Simple Text

Figure 7.1 shows a `Text` widget with three lines of text.

7.2 Fonts

It is possible to use different fonts in a `Text` widget. Font support is located in the Tkinter's font module.

Listing 7.2: fontexample.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from tkinter import Tk, Text, END, BOTH, TOP, StringVar
from tkinter.ttk import Frame, OptionMenu
from tkinter.font import Font

"""
ZetCode Tkinter e-book

In this script, we change the font of
a Text widget using an OptionMenu widget.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Fonts")
        self.pack(fill=BOTH, expand=True)

        self.ovar = StringVar()
        options = ["Arial 10", "Times 12", "Courier 10"]

        om = OptionMenu(self, self.ovar, options[0],
                        *options, command=self.onSelect)
        om.pack(pady=10)

        self.txt = Text(self)

        self.txt.pack(fill=BOTH, expand=True)
        self.txt.focus_set()

    def onSelect(self, val):

        fname, fsize = val.split()

        f = Font(family=fname, size=fsize)
        self.txt.configure(font=f)

def main():
```

```
root = Tk()
ex = Example(root)
root.geometry("300x250+300+300")
root.mainloop()

if __name__ == '__main__':
    main()
```

The example uses an `OptionMenu` widget with three options. These options are font names and sizes. The selected font is applied on the text widget.

```
options = ["Arial 10", "Times 12", "Courier 10"]
```

We have three font descriptions.

```
om = OptionMenu(self, self.ovar, options[0],
                *options, command=self.onSelect)
om.pack(pady=10)
```

These fonts are included as options in the `OptionMenu` widget.

```
def onSelect(self, val):
    fname, fsize = val.split()
    ...
```

We split the font description into a font name and font size.

```
f = Font(family=fname, size=fsize)
```

From the supplied values, a `Font` object is created.

```
self.txt.configure(font=f)
```

With the `configure()` method, we apply the chosen font on the text widget.

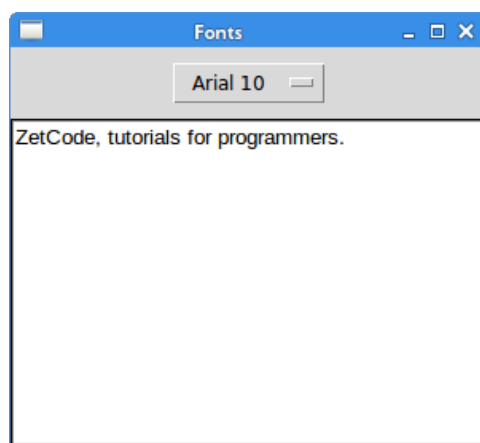


Figure 7.2: Fonts

Figure 7.2 shows a line of text with Arial font applied.

7.3 Selecting text

The `get()` method is used to retrieve text from the text widget. The method receives indexes of the selected text.

Listing 7.3: selecting_text.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we show selected text
in a message dialog.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import (Tk, Text, BOTH, END, TOP, SEL_FIRST,
                     SEL_LAST, TclError)
from tkinter.ttk import Frame, Button
from tkinter.messagebox import showinfo

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Text selection")
        self.pack(fill=BOTH, expand=True)

        selBtn = Button(self, text="Get selection",
                        command=self.onGetSelection)
        selBtn.pack(side=TOP, pady=5)

        self.txt = Text(self)
        self.txt.pack(fill=BOTH, expand=True)

        self.txt.insert(END, "Beyond Sing the Woods.")

    def onGetSelection(self):

        try:
            selText = self.txt.get(SEL_FIRST, SEL_LAST)
            showinfo("You have selected", message=selText)

        except TclError:
            pass
```

```
def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x200+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()
```

The example shows the selected text in a message box.

```
selText = self.txt.get(SEL_FIRST, SEL_LAST)
```

The `SEL_FIRST` index points to the start of the selection; the `SEL_LAST` index points to the end of the selection.

```
showinfo("You have selected", message=selText)
```

The selected text is displayed in a message box.

```
except TclError:
    pass
```

A `TclError` exception is raised when no text is selected. This is a no operation for us.

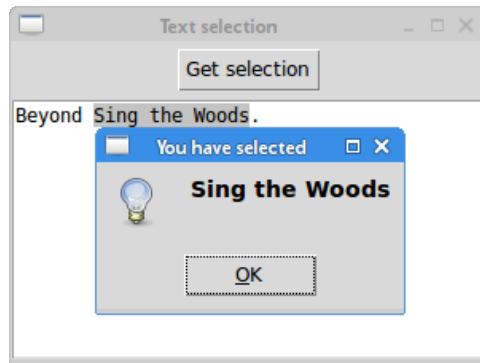


Figure 7.3: Text selection

Figure 7.3 shows a message box containing the selected text from the text widget.

7.4 Image

The `Text` widget is capable of containing an image. It is treated as a single character whose size is the natural size of the object. To insert an image, we use the `image_create()` method.

Listing 7.4: text_image.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from PIL import Image, ImageTk
from tkinter import Tk, Text, END, BOTH, TOP
from tkinter.ttk import Frame

"""
ZetCode Tkinter e-book

In this script, we insert an image into
the Text widget.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Image")
        self.pack(fill=BOTH, expand=True)

        self.bard = Image.open("bardejov.jpg")
        self.photo = ImageTk.PhotoImage(self.bard)

        txt = Text(self)

        txt.image_create(END, image=self.photo)
        txt.insert(END, '\n')
        txt.insert(END, "Bardejov is a small town in east Slovakia.")

        txt.pack(fill=BOTH, expand=True)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()
```

A small image is inserted into the text widget.

```
self.bard = Image.open("bardejov.jpg")
self.photo = ImageTk.PhotoImage(self.bard)
```

A Tkinter compatible image is created from the supplied JPG image.

```
txt.image_create(END, image=self.photo)
```

The image is inserted into the text widget.

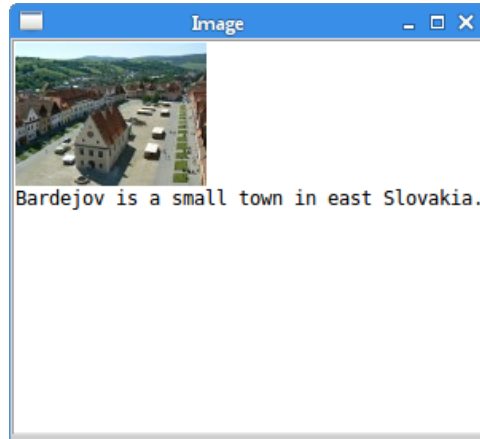


Figure 7.4: Image in a Text widget

Figure 7.4 shows a small image inside a text widget.

7.5 Undo, redo

The Text widget has some simple undo and redo mechanism. We need to explicitly enable the undo and redo operations, they are disabled by default. The undo and redo operations work for text insertions and text deletions.

Listing 7.5: undoredo.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we enable undo and redo
operations.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Text, Menu, BOTH, TclError
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
```

```

    Frame.__init__(self, parent)

    self.parent = parent
    self.initUI()

def initUI(self):

    self.parent.title("Undo, redo")
    self.pack(fill=BOTH, expand=True)

    self.txt = Text(self, undo=True)
    self.txt.bind("<Button-3>", self.showContextmenu)
    self.txt.focus()

    self.txt.pack(fill=BOTH, expand=True)

    self.cmenu = Menu(self, tearoff=False)
    self.cmenu.add_command(label="Undo", command=self.doUndo)
    self.cmenu.add_command(label="Redo", command=self.doRedo)

def showContextmenu(self, e):

    self.cmenu.post(e.x_root, e.y_root)

def doUndo(self):

    try:
        self.txt.edit_undo()

    except TclError:
        pass

def doRedo(self):

    try:
        self.txt.edit_redo()
    except TclError:
        pass

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x200+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The built-in undo and redo operations are tied to the Ctrl+Z and Ctrl+Shift+Z key combinations. In addition, we create undo and redo context menu options.

```
self.txt = Text(self, undo=True)
```

The undo and redo operations are enabled by setting the text widget's `undo` option to `True`.

```
def doUndo(self):

    try:
        self.txt.edit_undo()

    except TclError:
        pass
```

The `edit_undo()` method performs an undo operation. We also suppress the Nothing to undo error messages; they are irrelevant for us.

```
def doRedo(self):

    try:
        self.txt.edit_redo()
    except TclError:
        pass
```

The `edit_redo()` method performs a redo operation.

7.6 Cut, copy, and paste

Cutting, copying, and pasting text are one of the most common operations with text. In the `Text` widget, these operations are enabled by default for key combinations. We will show how to enable these operations for menu items.

Listing 7.6: `cut_copy_paste.py`

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we enable cut, copy, and
paste operations for menu items.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Text, Menu, BOTH, END, TclError
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()
```



```

def initUI(self):

    self.parent.title("Cut, copy, paste")
    self.pack(fill=BOTH, expand=True)

    self.txt = Text(self)
    self.txt.bind("<Button-3>", self.showContextmenu)
    self.txt.focus()

    self.txt.pack(fill=BOTH, expand=True)
    self.txt.insert(END, "ZetCode, tutorials for programmers")

    self.cmenu = Menu(self, tearoff=False)
    self.cmenu.add_command(label="Cut", command=self.editCut)
    self.cmenu.add_command(label="Copy", command=self.editCopy)
    self.cmenu.add_command(label="Paste", command=self.editPaste)

def editCut(self):

    self.txt.event_generate("<<Cut>>")

def editCopy(self):

    self.txt.event_generate("<<Copy>>")

def editPaste(self):

    self.txt.event_generate("<<Paste>>")

def showContextmenu(self, e):

    self.cmenu.post(e.x_root, e.y_root)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("300x200+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example enables cut, copy, and paste operations for menu items of a context menu. The operations are performed by triggering existing virtual events.

```

self.txt = Text(self)
self.txt.bind("<Button-3>", self.showContextmenu)

```

A context menu creation is bound to a right mouse click.

```

self.cmenu = Menu(self, tearoff=False)
self.cmenu.add_command(label="Cut", command=self.editCut)
self.cmenu.add_command(label="Copy", command=self.editCopy)
self.cmenu.add_command(label="Paste", command=self.editPaste)

```

A context menu with the three operations is created.

```
def editCut(self):

    self.txt.event_generate("<<Cut>>")
```

As we already mentioned, there are built-in operations for cut, copy, and paste operations. For the cut operation triggered from the context menu, we generate the <<Cut>> virtual event.

7.7 Searching text

In text editors, the text that matches the search keyword stands out from the rest of the text by being highlighted. The following example creates such a solution.

Listing 7.7: searching_text.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we search for text. The found text is
highlighted.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

from tkinter import Tk, Text, Menu, BOTH, END, N, S, E, W
from tkinter.ttk import Frame, Entry, Button

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Searching text")
        self.pack(fill=BOTH, expand=True)

        self.columnconfigure(0, weight=1)
        self.rowconfigure(1, weight=1)

        self.entry = Entry(self)
        self.entry.grid(row=0, column=0, padx=5, sticky=W+E)

        searchBtn = Button(self, text="Search", command=self.onSearch)
        searchBtn.grid(row=0, column=1, padx=5, pady=5)
```

```

self.txt = Text(self)
self.txt.focus()

self.txt.grid(row=1, column=0, columnspan=2, padx=5,
              pady=5, sticky=W+E+N+S)

self.txt.tag_configure("wordfound", background="gray88")

def onSearch(self):

    self.txt.tag_remove("wordfound", "1.0", END)

    idx = "1.0"
    myword = self.entry.get()

    if (len(myword.strip()) == 0):
        return

    while True:

        idx = self.txt.search(myword, idx, stopindex=END)

        if (idx == ""):

            return
        else:

            self.txt.tag_add("wordfound", idx,
                            "%s+%dc" % (idx, len(myword)))
            idx = "%s+%dc" % (idx, len(myword))

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("350x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we have a search entry, button, and a text widget. Clicking on the Search button performs a searching operation. All the matching text is highlighted.

```
self.txt.tag_configure("wordfound", background="gray88")
```

A text tag is created for the matching strings; these strings will have a gray background.

```

def onSearch(self):

    self.txt.tag_remove("wordfound", "1.0", END)
    ...

```

When we begin searching, we first remove any previous highlights. The "1.0" is a specific index—the beginning of the text widget's content. The first digit is a line number, the second digit is a column number. Lines start from 1, columns from 0.

```
if (len(myword.strip()) == 0):
    return
```

We skip searching if the entry widget is empty or contains just white spaces.

```
idx = self.txt.search(myword, idx, stopindex=END)
```

The `search()` method is used to search for text. The first parameter is the search string. The second parameter is the index from which the searching progresses. The `stopindex` is the limit of the searching operation. The method returns an empty string if no (other) string was found. The default searching direction is forward.

```
self.txt.tag_add("wordfound", idx,
                "%s+%dc" % (idx, len(myword)))
idx = "%s+%dc" % (idx, len(myword))
```

We apply the `wordfound` tag at the matched word and increase the search index by the length of the word.

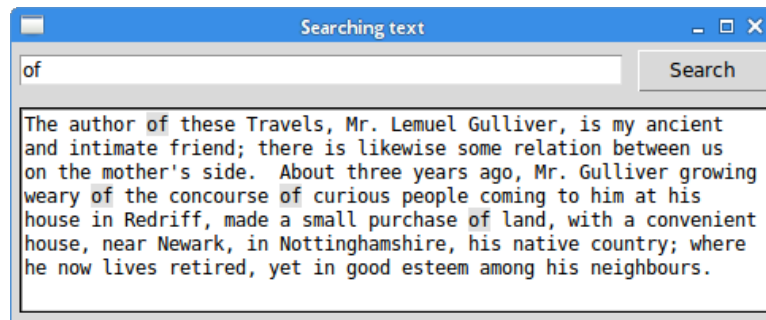


Figure 7.5: Searching text

Figure 7.5 shows a text widget with all occurrences of the “of” word highlighted.

7.8 Spell checking

The next example will do spell checking. In spell checking, we check if the strings available in the text widget are contained within a dictionary of words.

Listing 7.8: spellcheck.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
```

ZetCode Tkinter e-book

This script shows how to do spell checking
in a Text widget.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

```
import re
from tkinter import Tk, Text, BOTH, END, Y, LEFT
from tkinter.ttk import Frame, Button

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Spell checking")
        self.pack(fill=BOTH, expand=True)

        f = Frame(self)
        f.pack(fill=Y, expand=True)

        spellBtn = Button(f, text="Spell check",
                           command=self.onSpellCheck)
        spellBtn.pack(side=LEFT, pady=5)

        clearBtn = Button(f, text="Clear",
                           command=self.onClear)
        clearBtn.pack(side=LEFT, padx=5)

        self.txt = Text(self)
        self.txt.pack(fill=BOTH, expand=True)

        self.txt.tag_configure("misspell", foreground="red",
                               underline=True)

        with open("words", "r") as fwords:
            self.words = fwords.read().split("\n")

    def onSpellCheck(self):

        self.txt.tag_remove("misspell", "1.0", END)

        mywords = self.txt.get("1.0", END).split()

        exclude_chars_start = ("(", "[")
        exclude_chars_end = (".", ",", "?", ")", "]", "!", ":", ";")

        idx = "1.0"

        for myword in mywords:
```

```

        if (len(myword) == 1):
            continue

        if myword[0] in exclude_chars_start:
            myword = myword[1:]

        if myword[-1] in exclude_chars_end:
            myword = myword[:-1]

        if re.match("[0-9]", myword):
            continue

        if (myword not in self.words and myword.lower()
            not in self.words):
            idx = self.txt.search(myword, idx)
            self.txt.tag_add("misspell", idx,
                            "%s+%dc" % (idx, len(myword)))

        idx = "%s+%dc" % (idx, len(myword))

    def onClear(self):

        self.txt.tag_remove("misspell", "1.0", END)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("450x300+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The program contains two buttons and a text widget. We use a dictionary available in Linux.

```

self.txt.tag_configure("misspell", foreground="red",
                        underline=True)

```

A misspell tag is created; this tag makes the text underlined and rendered in red colour.

```

with open("words", "r") as fwords:
    self.words = fwords.read().split("\n")

```

In the current working directory, we have a words file, which is a dictionary of English words. Each of the words is located on a separate line. We load the words and fill a Python list with them.

```

def onSpellCheck(self):

    self.txt.tag_remove("misspell", "1.0", END)
    ...

```

At the beginning of the spell checking process, we remove any misspell tags.

```
mywords = self.txt.get("1.0", END).split()
```

We get the contents of the text widget into a `mywords` list.

```
exclude_chars_start = ("(", "[")
exclude_chars_end = (".", ",", "?", ")", "]", "!", ":", ";")
```

The strings can have many punctuation marks; we will strip them from the words.

```
if (len(myword) == 1):
    continue
```

Strings consisting of one character are dismissed.

```
if myword[0] in exclude_chars_start:
    myword = myword[1:]

if myword[-1] in exclude_chars_end:
    myword = myword[:-1]
```

If a string contains a punctuation character at the beginning or at the end, we strip it.

```
if re.match("[0-9]", myword):
    continue
```

Numeral strings are not checked. A regular expression is used to identify digits in the string.

```
if (myword not in self.words and myword.lower()
    not in self.words):
```

We check if the word from the text widget is not contained in the dictionary. We also check for the lowercase version of the word. This is because first words of a sentence are capitalized but they are available only in lowercase in the dictionary.

```
idx = self.txt.search(myword, idx)
```

If the word is not located in the dictionary, we search for it in the text widget with the `search()` method. The second parameter of the method is the index from which the searching starts. The default search is a forward search.

```
self.txt.tag_add("misspell", idx,
    "%s+%dc" % (idx, len(myword)))
```

We apply the tag at the misspelled word.

```
idx = "%s+%dc" % (idx, len(myword))
```

Finally, the index is increased by the length of the misspelled word. The searching for the next word starts from this place.

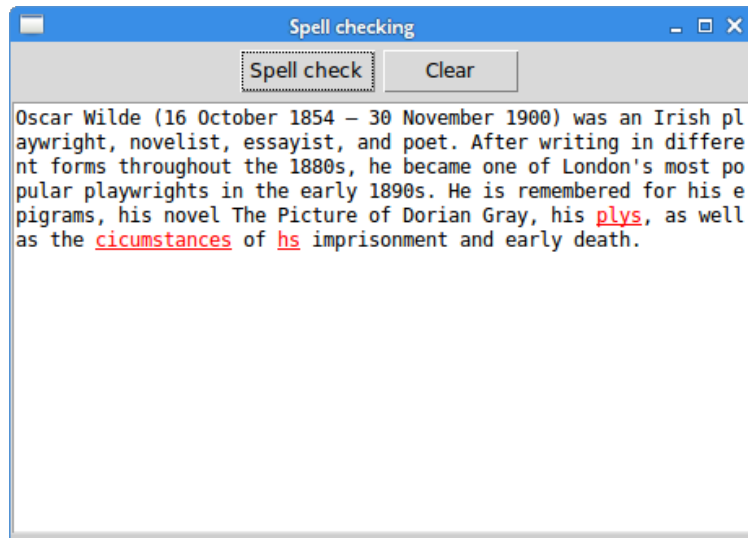


Figure 7.6: Spell checking

Figure 7.6 shows three misspelled words identified by our spell checking.

7.9 Opening, saving files

In the last example, we create a simple text editor. We open and save simple text files. Despite having only a limited functionality, the example is complex; we need to take quite a few considerations into account.

Listing 7.9: open_save_file.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

In this script, we create a simple text
editor. It is possible to load and save
text files.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

import sys
from tkinter import Tk, BOTH, END, N, S, E, W, StringVar
from tkinter.ttk import Frame, Label
from tkinter.scrolledtext import ScrolledText
from tkinter.filedialog import askopenfilename, asksaveasfilename
from tkinter.messagebox import askyesnocancel, showerror

class Example(Frame):
```



```

def __init__(self, parent):
    Frame.__init__(self, parent)

    self.parent = parent
    self.initUI()

def initUI(self):

    self.parent.title("Open, save file")
    self.pack(fill=BOTH, expand=True)

    self.columnconfigure(0, weight=1)
    self.rowconfigure(0, weight=1)

    self.txt = ScrolledText(self)
    self.txt.bind("<Control-o>", self.onOpenFile)
    self.txt.bind("<Control-s>", self.onSaveFile)
    self.txt.bind("<Control-w>", self.onCloseFile)
    self.txt.bind("<Control-Shift-S>", self.onSaveAsFile)
    self.txt.bind("<KeyRelease>", self.onKeyReleased)
    self.txt.focus()

    self.parent.protocol("WM_DELETE_WINDOW", self.onDeleteEvent)

    self.txt.grid(row=0, column=0, columnspan=2, padx=5,
                  pady=5, sticky=W+E+N+S)

    self.lvar = StringVar()
    sbar = Label(self, textvariable=self.lvar)
    sbar.grid(row=1, column=0, sticky=W+E, padx=5)

    self.filename = "Untitled"
    self.lvar.set(self.filename)

def onDeleteEvent(self):

    if (self.txt.edit_modified()):

        msg = "Would you like to save the modified file?"

        ret = askyesnocancel("Question", message=msg)

        if ret == True:

            if (self.filename.startswith("Untitled")):
                self.onSaveAsFile(None)
            else:
                self.doSaveFile()
            self.quit()

        elif ret == False:

            self.quit()
        else:
            return

    else:
        self.quit()

```

```

def onCloseFile(self, e):

    if (not self.txt.edit_modified() and
        self.filename.startswith("Untitled")):
        sys.exit(0)

    if (not self.txt.edit_modified() and not
        self.filename.startswith("Untitled")):

        self.doDeleteContent()
        return

    msg = "Save changes before closing file?"

    ret = askyesnocancel("Question", message=msg)

    if ret == True:

        if (self.filename.startswith("Untitled")):
            ftypes = [('All files', '*'), ('Python files', '*.py')]
            filename = asksaveasfilename(filetypes=ftypes)

            if filename == "":
                return

            self.filename = filename

        self.doSaveFile()
        self.doDeleteContent()

    elif ret == False:
        self.doDeleteContent()
    else:
        return

def onSaveFile(self, e):

    if not self.txt.edit_modified():
        return

    if self.filename.startswith("Untitled"):
        self.onSaveAsFile(e)
    else:
        self.doSaveFile()

def onSaveAsFile(self, e):

    ftypes = [('All files', '*'), ('Python files', '*.py')]
    filename = asksaveasfilename(filetypes=ftypes)

    if filename == "":
        return

    self.filename = filename
    self.doSaveFile()

def doSaveFile(self):

    content = self.txt.get("1.0", END)

```

```

        with open(self.filename, "w") as f:
            f.write(content)

        self.txt.edit_modified(False)
        self.lvar.set(self.filename)

    def doDeleteContent(self):

        self.txt.delete("1.0", END)
        self.txt.edit_modified(False)
        self.filename = "Untitled"
        self.lvar.set(self.filename)

    def onKeyReleased(self, e):

        if self.txt.edit_modified():
            self.lvar.set(self.filename + " *")

    def onOpenFile(self, e):

        if self.txt.edit_modified():
            showerror("Error", "Save file before opening")
            return

        ftypes = [('All files', '*'), ('Python files', '*.py')]
        filename = askopenfilename(parent=self, filetypes=ftypes)

        if filename == "":
            return
        else:
            self.filename = filename

        self.lvar.set(self.filename)

        with open(self.filename, "r") as fname:
            text = fname.read()

        self.txt.delete("1.0", END)
        self.txt.insert(END, text)
        self.txt.edit_modified(False)
        return "break"

    def main():

        root = Tk()
        ex = Example(root)
        root.geometry("450x350+300+300")
        root.mainloop()

if __name__ == '__main__':
    main()

```

There are key combinations for opening, saving, and closing files. Only one file can be opened at a time. Its name is shown in the statusbar. A document that has not yet been given a name is called “Untitled”. An asterisk is given to the

end of the document name to indicate that it has been modified.

```
self.txt = ScrolledText(self)
```

We use the `ScrolledText` widget. This is a `Text` widget with some scrolling functionality enabled.

```
self.txt.bind("<Control-o>", self.onOpenFile)
self.txt.bind("<Control-s>", self.onSaveFile)
self.txt.bind("<Control-w>", self.onCloseFile)
self.txt.bind("<Control-Shift-S>", self.onSaveAsFile)
```

We bind key combinations for opening, saving, and closing files. For instance, the standard `Ctrl+O` combination is used for opening a new file.

```
self.parent.protocol("WM_DELETE_WINDOW", self.onDeleteEvent)
```

To prevent accidental loss of data, we react to window close events. We give a chance to save the modified data before closing the application.

```
def onDeleteEvent(self):
    if (self.txt.edit_modified()):
        msg = "Would you like to save the modified file?"
        ret = askyesnocancel("Question", message=msg)
        ...
    else:
        self.quit()
```

Clicking on the window Close button or pressing the `Alt+F4` combination generates a window close event. If there was some modification of the data, we show a dialog asking for further action. If not, the application is closed. The `edit_modified()` method determines if the document was modified.

```
if ret == True:
    if (self.filename.startswith("Untitled")):
        self.onSaveAsFile(None)
    else:
        self.doSaveFile()
    self.quit()

elif ret == False:
    self.quit()
else:
    return
```

If we click the dialog's Yes button, the data is saved. (If the document has not been named, a Save As dialog is generated.) Clicking on the No button, the changes are discarded. Nothing is done if we click on the Cancel button.

```
def onCloseFile(self, e):
    if (not self.txt.edit_modified() and
        self.filename.startswith("Untitled")):
```

```

        sys.exit(0)
    ...

```

The Ctrl+W key combination triggers the `onCloseFile()` method. The application is terminated if the document was not modified and it has the default name.

```

if (not self.txt.edit_modified() and not
    self.filename.startswith("Untitled")):

    self.doDeleteContent()
    return

```

If the document was not modified (it was saved) and the user has given the document a name, the document's content is erased in the `doDeleteContent()` method.

```

msg = "Save changes before closing file?"

ret = askyesnocancel("Question", message=msg)

```

The other possibility is that the document was modified. For such a case, we show a dialog asking for further actions.

```

if ret == True:

    if (self.filename.startswith("Untitled")):
        ftypes = [('All files', '*'), ('Python files', '*.py')]
        filename = asksaveasfilename(filetypes=ftypes)

        if filename == "":
            return

        self.filename = filename

    self.doSaveFile()
    self.doDeleteContent()

elif ret == False:
    self.doDeleteContent()
else:
    return

```

Choosing Yes from the dialog, a new Save As dialog is generated; it is used to choose a file name for our untitled document. Then we save the file to the chosen file name and delete the text widget's content. Choosing No leads to the deletion of the data; the data is not saved. The third option does nothing.

```

def onSaveFile(self, e):

    if not self.txt.edit_modified():
        return

    if self.filename.startswith("Untitled"):
        self.onSaveAsFile(e)
    else:
        self.doSaveFile()

```

In the `onSaveFile()` we first ensure that the document was modified. If the document has the default name, we first show the Save As dialog for choosing a file name. Otherwise, we call the `doSaveFile()` method which saves the file to the disk.

```
def doSaveFile(self):
    content = self.txt.get("1.0", END)

    with open(self.filename, "w") as f:
        f.write(content)

    self.txt.edit_modified(False)
    self.lvar.set(self.filename)
```

In the `doSaveFile()` method, we get the data from the text widget and save it into the current file name. The file name is stored in the `filename` variable. After the file was saved, the text widget's state is changed to non-modified. This is done by passing `False` to the `edit_modified()` method. The statusbar is updated as well—the asterisk indicating document modification is removed.

```
def doDeleteContent(self):
    self.txt.delete("1.0", END)
    self.txt.edit_modified(False)
    self.filename = "Untitled"
    self.lvar.set(self.filename)
```

The purpose of the `onDeleteContent()` method is to delete the data from the document. The text widget's `delete()` method removes all data from the beginning to the end of the widget. The document's state is changed to non-modified. Finally, we change the `filename` variable to the default “Untitled” value.

```
def onKeyReleased(self, e):
    if self.txt.edit_modified():
        self.lvar.set(self.filename + " *")
```

Each time a key is pressed, we check if the document was modified with the `edit_modified()` method. If so, we add an asterisk at the end of its name shown in the statusbar.

```
def onOpenFile(self, e):
    if self.txt.edit_modified():
        showerror("Error", "Save file before opening")
        return
```

An error message is shown if we try to open a new file while the old one was not saved yet.

```
ftypes = [('All files', '*'), ('Python files', '*.py')]
filename = askopenfilename(parent=self, filetypes=ftypes)

if filename == "":
    return
else:
```

```

        self.filename = filename

self.lvar.set(self.filename)

```

A dialog is shown to select the file name to be opened. There are two categories of files: all types of files and Python files.

```

with open(self.filename, "r") as fname:
    text = fname.read()

self.txt.delete("1.0", END)
self.txt.insert(END, text)
self.txt.edit_modified(False)

```

The file is opened and all the data is read. We delete the existing content from the document and insert the retrieved data. From the perspective of a text editor, the document is not modified at this moment.

```

return "break"

```

Returning the "break" string stops the propagation of the modification events. Without this line the modification state would not be updated correctly.

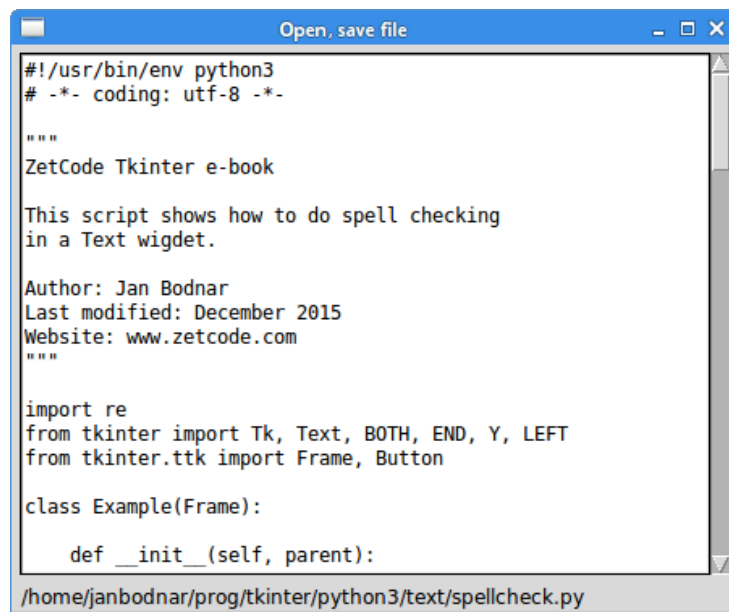


Figure 7.7: Opening, saving files

Figure 7.7 shows a file opened in a simple text editor. The file's name is in the statusbar.

Chapter 8

Treeview

`Treeview` displays a collection of items. The items can be shown in multiple of columns. Also, they can form a hierarchy. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label.

8.1 Simple example

We start with a simple example of the `Treeview` widget.

Listing 8.1: `simple_treeview.py`

```
#!/usr/bin/env python3

from tkinter import Tk, BOTH, E, END
from tkinter.ttk import Treeview, Frame

"""
ZetCode Tkinter e-book

In this script, create a simple Treeview
widget.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Simple Treeview")
        self.pack(fill=BOTH, expand=True)
```



```

tree = Treeview(self, columns=("quant"))

tree.heading("#0", text="Item")
tree.heading("#1", text="Quantity")
tree.column("#0", width=150)
tree.column("#1", width=150)

tree.column("#1", anchor=E)

tree.insert("", index=END, text="coins", values=("10",))
tree.insert("", index=END, text="pens", values=("5",))
tree.insert("", index=END, text="bottles", values=("2",))

tree.pack(fill=BOTH, expand=True)
self.pack()

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example shows some data in two columns.

```
tree = Treeview(self, columns=("quant"))
```

A `Treeview` widget is created. The `columns` parameter sets the number of columns to display and their handlers. The handlers may be used to refer to the columns. Note that these are additional columns created after the first one, which shows the row labels.

```
tree.heading("#0", text="Item")
tree.heading("#1", text="Quantity")
```

With the `heading()` method, we set the column titles. The `#0` refers to the first column, the `#1` to the second.

```
tree.column("#0", width=150)
tree.column("#1", width=150)
```

With the `column()` method, we set the width of the columns. The default width of a column is 200 units.

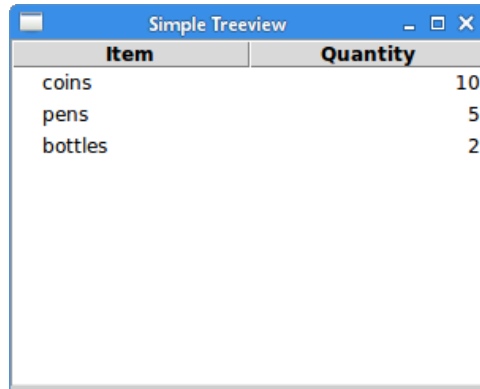
```
tree.column("#1", anchor=E)
```

With the `anchor` option, we align the data inside the second column to the right.

```
tree.insert("", index=END, text="coins", values=("10",))
```

The `insert()` method puts data into the treeview widget. The first parameter is the parent Id; if we are creating a new top-level item, the parameter is an empty string. The `index` parameter sets the position of the item in the treeview.

Passing `END` to the parameter places the item at the end. The `text` option sets the name of the label—the value of the first column. The `values` option sets data for the rest of the columns. Since we have only one additional column, there is only one value.



| Item | Quantity |
|---------|----------|
| coins | 10 |
| pens | 5 |
| bottles | 2 |

Figure 8.1: Simple Treeview

Figure 8.1 shows a simple example of a `Treeview` widget.

8.2 Row colours

It is possible to utilize tags to change the row colours of a treeview.

Listing 8.2: `alternating_row_colours.py`

```
#!/usr/bin/env python3

from PIL import Image, ImageTk
from tkinter import Tk, BOTH, E, END
from tkinter.ttk import Treeview, Frame, Style

"""
ZetCode Tkinter e-book

In this script, we change the colour of every even
row to gray.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):
```

```

self.parent.title("Alternating row colours")
self.pack(fill=BOTH, expand=True)

data = { "coins": "10", "pens": "5", "bottles": "2",
         "bags": "4", "spoons": "6", "brushes": "2",
         "lamps": "7", "corks": "2", "watches": "3" }

tree = Treeview(self, columns=("quant"))
tree.heading('#0', text="Item")
tree.heading('#1', text="Quantity")
tree.column('#1', anchor=E)

tree.tag_configure("evenrow", background="gray88")

i = 0
for itm in data.keys():
    if (i % 2 == 0):
        tree.insert("", index=END, text=itm,
                    values=(data[itm]))
    else:
        tree.insert("", index=END, text=itm,
                    values=(data[itm]), tags = ('evenrow',))
    i += 1

tree.pack(fill=BOTH, expand=True)

self.pack()

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

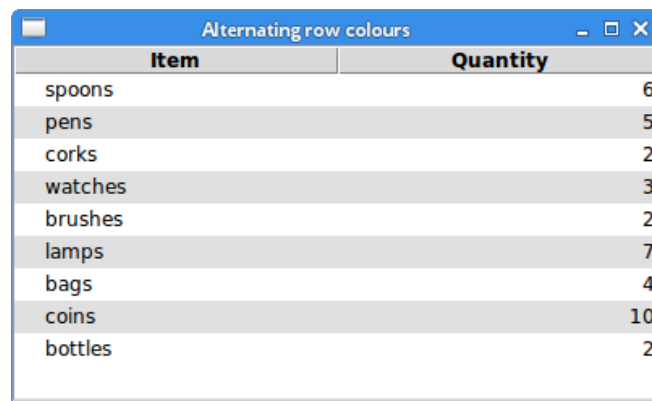
In the example, we change the colour of every even row.

```
tree.tag_configure("evenrow", background="gray88")
```

Each treeview item can have a tag set. The `tag_configure()` method creates an `evenrow` tag, which sets a grey background.

```
tree.insert("", index=END, text=itm,
            values=(data[itm]), tags = ('evenrow',))
```

Every row with an even index has the `evenrow` tag applied.



| Item | Quantity |
|---------|----------|
| spoons | 6 |
| pens | 5 |
| corks | 2 |
| watches | 3 |
| brushes | 2 |
| lamps | 7 |
| bags | 4 |
| coins | 10 |
| bottles | 2 |

Figure 8.2: Alternating row colours

Figure 8.2 shows a `Treeview` widget with alternating row colours.

8.3 Hierarchy

Treeview items can form a hierarchy. The following example demonstrates this.

Listing 8.3: `treeview_hierarchy.py`

```
#!/usr/bin/env python3

from tkinter import Tk, BOTH, E, END
from tkinter.ttk import Treeview, Frame

"""
ZetCode Tkinter e-book

In this script, we organize treeview items in a
hierarchy.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Hierarchy")
        self.pack(fill=BOTH, expand=True)

        asia = { "China": "Beijing", "Mongolia": "Ulaanbaatar",
                  "India": "New Delhi", "Pakistan": "Islamabad"
                }
```

```

    europe = { "Slovakia": "Bratislava", "Hungary": "Budapest",
               "Poland": "Warsaw", "Germany": "Berlin"
    }

    tree = Treeview(self, columns=("capit"))

    tree.heading("#0", text="Country")
    tree.heading("#1", text="Capital")

    idAsia = tree.insert("", index=END, text="Asia")
    idEurope = tree.insert("", index=END, text="Europe")

    for el in asia.keys():
        tree.insert(idAsia, index=END, text=el,
                   values=(asia[el],))

    for el in europe.keys():
        tree.insert(idEurope, index=END, text=el,
                   values=(europe[el],))

    tree.pack(fill=BOTH, expand=True)
    self.pack()

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example uses some geographical data to form a hierarchy in a treeview widget.

```

asia = { "China": "Beijing", "Mongolia": "Ulaanbaatar",
         "India": "New Delhi", "Pakistan": "Islamabad"
    }

europe = { "Slovakia": "Bratislava", "Hungary": "Budapest",
           "Poland": "Warsaw", "Germany": "Berlin"
    }

```

We have two dictionaries of continents and their countries.

```

idAsia = tree.insert("", index=END, text="Asia")
idEurope = tree.insert("", index=END, text="Europe")

```

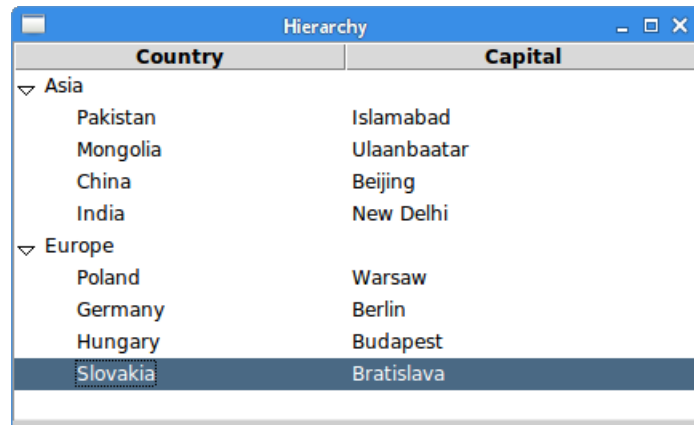
These two lines add two top-level items. Top-level items contain an empty string as their first parameter. The `insert()` method returns an Id of the added item, which is later used to add a subitem.

```

for el in asia.keys():
    tree.insert(idAsia, index=END, text=el,
               values=(asia[el],))

```

This for loop adds Asian countries to the Asia continent. Note that the first parameter of the `insert()` method is the Id of the previously created top-level item.



| Country | Capital |
|----------|-------------|
| Asia | |
| Pakistan | Islamabad |
| Mongolia | Ulaanbaatar |
| China | Beijing |
| India | New Delhi |
| Europe | |
| Poland | Warsaw |
| Germany | Berlin |
| Hungary | Budapest |
| Slovakia | Bratislava |

Figure 8.3: Treeview hierarchy

Figure 8.3 shows geographical data organized in a hierarchy.

8.4 Images

The row label may include an image. The Treeview's `insert()` has an `image` option to specify a photo image.

Listing 8.4: `treeview_images.py`

```
#!/usr/bin/env python3

from PIL import Image, ImageTk
from tkinter import Tk, BOTH, E, END, W
from tkinter.ttk import Treeview, Frame, Style

"""
ZetCode Tkinter e-book

In this script, we display PNG images
in the Treeview widget.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()
```

```

def initUI(self):

    self.parent.title("Images")
    self.pack(fill=BOTH, expand=True)

    Style().configure('Treeview', rowheight=30)

    desc = ("Device used to power utilities",
           "Device used to take photos",
           "Device used to record sounds")

    tree = Treeview(self, columns=("Description"))
    tree.heading('#0', text="Item")
    tree.heading('#1', text="Description")

    self.img1 = Image.open("battery.png")
    self.battery = ImageTk.PhotoImage(self.img1)

    tree.insert("", index=END, text="Battery", image=self.battery,
               values=(desc[0], ))

    self.img2 = Image.open("camera.png")
    self.camera = ImageTk.PhotoImage(self.img2)

    tree.insert("", index=END, text="Camera", image=self.camera,
               values=(desc[1], ))

    self.img3 = Image.open("microphone.png")
    self.microphone = ImageTk.PhotoImage(self.img3)

    tree.insert("", index=END, text="Microphone",
               image=self.microphone, values=(desc[2], ))

    tree.pack(fill=BOTH, expand=True)

    self.pack()

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example displays tree rows. The rows have images in their labels.

```
Style().configure('Treeview', rowheight=30)
```

We increase the height of the rows to put some additional space between the images.

```
self.img1 = Image.open("battery.png")
self.battery = ImageTk.PhotoImage(self.img1)
```

A photo image is created from the supplied PNG image.

```
tree.insert("", index=END, text="Camera", image=self.camera,
            values=(desc[1], ))
```

The `image` parameter of the `insert()` method specifies the photo image to be displayed.

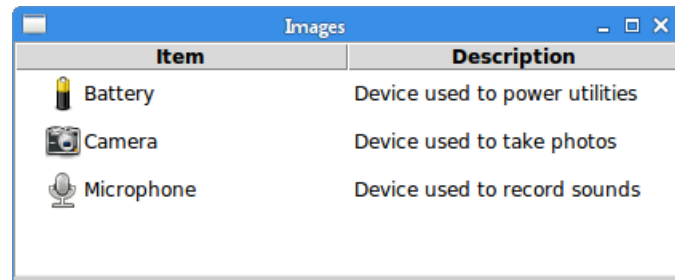


Figure 8.4: Treeview images

Figure 8.4 shows three images in the row labels.

8.5 Selection

There are three selections modes: (a) `BROWSE`, which allows one row to be selected at a time; (b) `EXTENDED`, which allows to select multiple items; (c) `NONE`, which disables selections. The `<<TreeviewSelect>>` virtual event is generated when a selection of rows has changed.

Listing 8.5: `treeview_selection.py`

```
#!/usr/bin/env python3

from tkinter import Tk, BOTH, E, END, LEFT, BROWSE, StringVar
from tkinter.ttk import Frame, Treeview, Label

"""
ZetCode Tkinter e-book

In this script, selected row's values of a Treeview
are displayed in a label widget.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):
```



```

self.parent.title("Selection")
self.pack(fill=BOTH, expand=True)

data = { "coins": "10", "pens": "5", "bottles": "2",
         "bags": "4", "spoons": "6" }

tree = Treeview(self, selectmode=BROWSE, columns=("quant"))

tree.heading("#0", text="Item")
tree.heading("#1", text="Quantity")

tree.column("#1", anchor=E)

for itm in data.keys():
    tree.insert("", index=END, text=itm, values=(data[itm],))

tree.bind("<<TreeviewSelect>>", self.onSelect)

tree.pack(fill=BOTH, expand=True)

self.lvar = StringVar()
lbl = Label(self, textvariable=self.lvar, text="Ready")
lbl.pack(side=LEFT, pady=2)

self.pack()

def onSelect(self, e):

    sender = e.widget
    itm = sender.selection()[0]
    val1 = sender.item(itm, "text")
    val2 = sender.item(itm, "values")[0]
    msg = "{0} : {1}".format(val1, val2)
    self.lvar.set(msg)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example shows the selected row's label text and value in the statusbar.

```
tree = Treeview(self, selectmode=BROWSE, columns=("quant"))
```

We create a single-selection `Treeview`.

```
tree.bind("<<TreeviewSelect>>", self.onSelect)
```

We listen for `<<TreeviewSelect>>` events.

```
def onSelect(self, e):
```

```

        sender = e.widget
        itm = sender.selection()[0]
    ...

```

Inside the `onSelect()` method, we determine the event sender. The `selection()` method returns the iid of the selected row. (The iid is an internal id used by Tkinter.)

```

val1 = sender.item(itm, "text")
val2 = sender.item(itm, "values")[0]

```

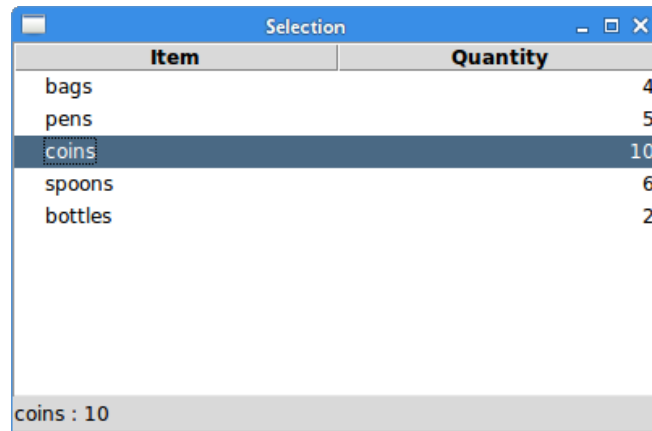
The `item()` method uses the iid to determine the row's label and value.

```

msg = "{0} : {1}".format(val1, val2)
self.lvar.set(msg)

```

The message is built and set to the label's control variable.



| Item | Quantity |
|---------|----------|
| bags | 4 |
| pens | 5 |
| coins | 10 |
| spoons | 6 |
| bottles | 2 |

coins : 10

Figure 8.5: Treeview selection

Figure 8.5 shows a row selected. The row's data is mirrored in the statusbar.

8.6 Inserting and deleting items

The following example allows to insert and delete rows.

Listing 8.6: `treeview_modify.py`

```

#!/usr/bin/env python3

from tkinter import Tk, BOTH, E, W, N, S, END, EXTENDED
from tkinter.ttk import Frame, Treeview, Label, Button, Entry

"""
ZetCode Tkinter e-book

In this script, we insert and delete
treeview items.

```

```

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

```

```

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Modifying rows")
        self.pack(fill=BOTH, expand=True)

        self.rowconfigure(1, weight=1)
        self.columnconfigure(1, weight=1)
        self.columnconfigure(3, weight=1)

        lbl1 = Label(self, text="Item:")
        lbl1.grid(row=0, column=0, padx=5, sticky=W)

        item_entry = Entry(self)
        item_entry.grid(row=0, column=1, sticky=W+E, padx=5)

        lbl1 = Label(self, text="Quantity:")
        lbl1.grid(row=0, column=2)

        quant_entry = Entry(self)
        quant_entry.grid(row=0, column=3, sticky=W+E, padx=5)

        insBtn = Button(self, text="Insert",
                        command=lambda: self.onInsert(item_entry, quant_entry))
        insBtn.grid(row=0, column=4, pady=5, padx=5)

        self.tree = Treeview(self, columns=("quant"),
                             selectmode=EXTENDED)

        self.tree.heading("#0", text="Item")
        self.tree.heading("#1", text="Quantity")

        self.tree.column(0, anchor=E)

        self.tree.grid(row=1, column=0, columnspan=5, padx=5,
                       sticky=E+W+N+S)

        remBtn = Button(self, text="Remove", command=self.onRemove)
        remBtn.grid(row=2, column=2, padx=5, pady=5, sticky=W)

        self.pack()

    def onRemove(self):

        iids = self.tree.selection()

        for iid in iids:

```

```

        self.tree.delete(iid)

    def onInsert(self, item_entry, quant_entry):

        val1 = item_entry.get()
        val2 = quant_entry.get()

        if (len(val1.strip()) == 0):
            return

        if (len(val1.strip()) == 0):
            return

        item_entry.delete(0, END)
        quant_entry.delete(0, END)
        self.tree.insert("", index=END, text=val1, values=(val2,))

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In addition to the treeview widget, the example uses buttons and entry widgets, which are used to modify treeview items.

```

self.tree = Treeview(self, columns=("quant"),
    selectmode=EXTENDED)

```

The treeview widget is created in the `EXTENDED` selection mode, in which it is possible to select multiple rows.

```

def onRemove(self):

    iids = self.tree.selection()

    for iid in iids:
        self.tree.delete(iid)

```

In the `EXTENDED` selection mode, the `selection()` method returns multiple iids. We go through the list of iids and remove the selected rows with the `delete()` method.

```

def onInsert(self, item_entry, quant_entry):

    val1 = item_entry.get()
    val2 = quant_entry.get()
    ...

```

Two entries are passed to the `onInsert()` method. We retrieve their contents with the `get()` method.

```

if (len(val1.strip()) == 0):
    return

if (len(val1.strip()) == 0):
    return

```

We ensure that the returned values are not white spaces or empty strings.

```

item_entry.delete(0, END)
quant_entry.delete(0, END)
self.tree.insert("", index=END, text=val1, values=(val2,))

```

We delete the contents of the entries and add a new row with the `insert()` method.

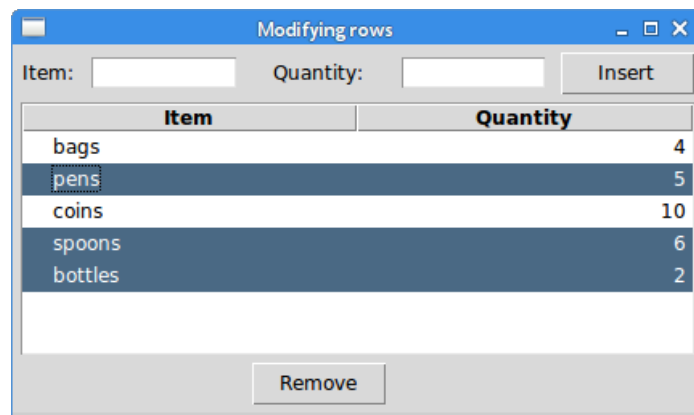


Figure 8.6: Inserting and deleting items

Figure 8.6 shows three selected rows before they are removed.

8.7 Double clicking a row

In this section, we react to a double click on a treeview's row. A double click triggers the `<Double-1>` event.

Listing 8.7: `treeview_doubleclick.py`

```

#!/usr/bin/env python3

from tkinter import (Tk, BOTH, E, END, LEFT, BROWSE,
                     StringVar, messagebox)
from tkinter.ttk import Frame, Treeview, Label

"""
ZetCode Tkinter e-book

In this script, we show the values of the row
on which we double click with a mouse in a
message box.

Author: Jan Bodnar
Last modified: December 2015

```

```

Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent, name="frame")

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Treeview")
        self.pack(fill=BOTH, expand=True)

        data = { "coins": "10", "pens": "5", "bottles": "2",
                  "bags": "4", "spoons": "6" }

        tree = Treeview(self, selectmode=BROWSE, columns=("quant"))

        tree.heading("#0", text="Item")
        tree.heading("#1", text="Quantity")

        tree.column("#1", anchor=E)

        for itm in data.keys():
            tree.insert("", index=END, text=itm, values=(data[itm]), )

        tree.bind("<Double-1>", self.onDoubleClick)

        tree.pack(fill=BOTH, expand=True)
        self.pack()

    def onDoubleClick(self, e):

        sender = e.widget

        iid = sender.identify("item", e.x, e.y)

        val1 = sender.item(iid, "text")
        val2 = sender.item(iid, "values")[0]

        msg = "{0} : {1}".format(val1, val2)
        messagebox.showinfo("Clicked item", msg)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example shows the row's data on which we click in a message box.

```
tree.bind("<Double-1>", self.onDoubleClick)
```

The `onDoubleClick()` method is bound to the `<Double-1>` event.

```
def onDoubleClick(self, e):
    sender = e.widget
    ...
```

First, we determine the event sender.

```
iid = sender.identify("item", e.x, e.y)
```

With the `identify()` method, we determine the row's iid. The method takes the x and y coordinates of the mouse pointer as parameters to find out the row.

```
val1 = sender.item(iid, "text")
val2 = sender.item(iid, "values")[0]
```

With the `item()` method, we get the row's data.

```
msg = "{0} : {1}".format(val1, val2)
messagebox.showinfo("Clicked item", msg)
```

A message is built from the row's data and displayed in a message box.

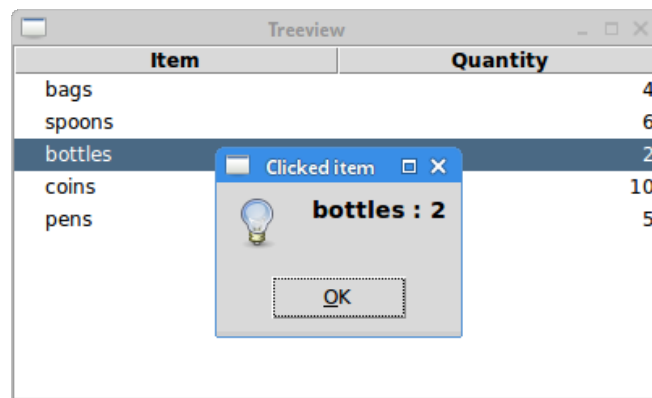


Figure 8.7: Double clicking a row

Figure 8.7 shows a message box displaying the data of the row on which we have double clicked.

8.8 Sorting

In this section, we will be sorting data. A treeview's column can react to mouse clicks. Traditionally, clicking on a column header is bound to a sorting operation.

Listing 8.8: `treeview_sort.py`

```
#!/usr/bin/env python3

from tkinter import Tk, BOTH, N, S, E, W, END, VERTICAL
from tkinter.ttk import Frame, Scrollbar, Treeview

"""
ZetCode Tkinter e-book

In this script, we can sort data by clicking
on the column header.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    sortOrder = True

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Sorting")
        self.pack(fill=BOTH, expand=True)

        self.createTreeview()
        self.loadData()

    def createTreeview(self):

        f = Frame(self)
        f.pack(fill=BOTH, expand=True)

        self.tree = Treeview(f, columns=("Country"), show='headings')

        ysb = Scrollbar(f, orient=VERTICAL, command=self.tree.yview)
        self.tree['yscroll'] = ysb.set

        self.tree.grid(row=0, column=0, sticky=N+S+E+W)
        ysb.grid(row=0, column=1, sticky=N+S)

        f.rowconfigure(0, weight=1)
        f.columnconfigure(0, weight=1)

    def loadData(self):

        self.data = [
            "Poland", "Argentina", "Slovakia", "Italy", "Canada",
            "China", "Germany", "United States", "Brazil",
            "Hungary", "Spain", "Japan", "Mexico", "Australia",
            "South Africa", "United Kingdom", "France", "Russia"
        ]
```



```

        self.tree.heading("#1", text="Country",
                           command=lambda c="Country": self.sortColumn(c,
                               Example.sortOrder))

    for item in self.data:
        self.tree.insert("", index=END, values=(item, ))

    def sortColumn(self, col, sortOrder):

        data = [(self.tree.set(child, col), child)
                 for child in self.tree.get_children('')]

        data.sort(reverse=sortOrder)
        for indx, item in enumerate(data):
            self.tree.move(item[1], '', indx)

        Example.sortOrder = not sortOrder

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In the example, we have one column with names of countries. Clicking on the column sorts the data. Clicking again sorts the data in a reverse order. There is also a vertical scrollbar attached to the treeview.

```

class Example(Frame):

    sortOrder = True

```

The `sortOrder` is a class variable which stores the sorting order. There are two sorting orders: ascending and descending.

```

self.tree = Treeview(f, columns=("Country"), show='headings')

```

A `Treeview` widget is created. The `show` parameter determines which parts of a row are shown. The default values are `tree headings`, which show both the tree labels and the columns. In our case, we show only the columns; e.g. we hide the first column containing the tree labels.

```

ysb = Scrollbar(f, orient=VERTICAL, command=self.tree.yview)
self.tree['yscroll'] = ysb.set

```

A vertical scrollbar is attached to the treeview widget.

```

self.tree.heading("#1", text="Country",
                  command=lambda c="Country": self.sortColumn(c,
                      Example.sortOrder))

```

We give a name to our column. The `command` parameter binds a `sortColumn()` method with a mouse click event on the column.

```
def sortColumn(self, col, sortOrder):
    data = [(self.tree.set(child, col), child)
             for child in self.tree.get_children('')]
    ...
```

In the `sortColumn()` method, we first get the data from the treeview. The `data` is a list of tuples containing the column values and their iids.

```
data.sort(reverse=sortOrder)
for indx, item in enumerate(data):
    self.tree.move(item[1], '', indx)
```

The data is sorted with the `sort()` method. The treeview's `move()` method is used to reposition the treeview items.

```
Example.sortOrder = not sortOrder
```

Finally, the sorting order is reversed.

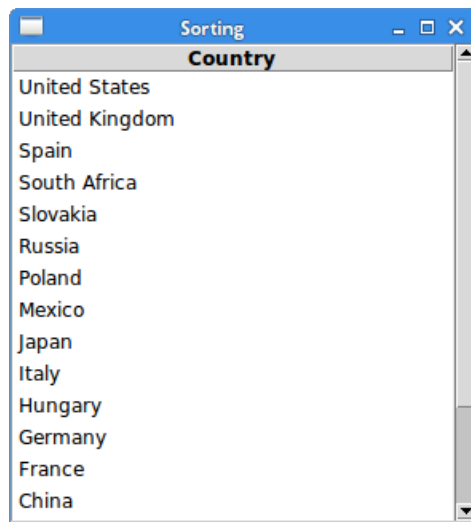


Figure 8.8: Sorting

Figure 8.8 shows a `Treeview` with its items alphabetically sorted. The vertical scrollbar allows to reach items that are not visible given the size of the window.

8.9 File browser

In the final example, we create a simple file browser. It uses a technique called *lazy loading*. *Lazy loading* is the practice of deferring the initialization of an object until the point at which it is needed. It helps the performance of an application.

Lazy loading is created with the help of the <<TreeviewOpen>> virtual event. The event is fired whenever a new node is opened. At this time, the program determines if a node is a directory and loads its files if it is not empty. It is not economical to go through the whole file system because it takes considerable time and because the user is most likely not interested in seeing all files during the lifetime of the application.

Listing 8.9: file_browser.py

```
#!/usr/bin/env python3

import os
import glob

from tkinter import Tk, BOTH, N, S, W, E, END
from tkinter.ttk import Frame, Treeview, Scrollbar

"""
ZetCode Tkinter e-book

In this script, we create a file browser.
It is based on the demo found in Tk.

Author: Jan Bodnar
Last modified: December 2015
Website: www.zetcode.com
"""

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("File browser")
        self.pack(fill=BOTH, expand=True)

        f = Frame(self)
        f.pack(fill=BOTH, expand=True)

        vsb = Scrollbar(f, orient="vertical")
        hsb = Scrollbar(f, orient="horizontal")

        cols = ("path", "type", "size")
        tree = Treeview(f, columns=cols, displaycolumns="size",
            yscrollcommand=lambda f, l:self.autoScroll(vsb, f, l),
            xscrollcommand=lambda f, l:self.autoScroll(hsb, f, l))

        vsb['command'] = tree.yview
        hsb['command'] = tree.xview

        tree.heading("#0", text="Path", anchor='w')
        tree.heading("#1", text="Size", anchor='w')
```

```

self.loadRoot(tree)
tree.bind("<<TreeviewOpen>>", self.updateTree)
tree.bind("<Double-Button-1>", self.changeDir)

tree.grid(column=0, row=0, sticky=N+S+W+E)
vsb.grid(column=1, row=0, sticky=N+S)
hsb.grid(column=0, row=1, sticky=E+W)

f.grid_columnconfigure(0, weight=1)
f.grid_rowconfigure(0, weight=1)

def loadRoot(self, tree):

    mydir = os.path.abspath(".").replace('\\', '/')
    node = tree.insert("", index=END, text=mydir,
        values=[mydir, "directory"])
    self.populateTree(tree, node)

def updateTree(self, e):

    tree = e.widget
    self.populateTree(tree, tree.focus())

def populateTree(self, tree, node):

    if tree.set(node, "type") != 'directory':
        return

    path = tree.set(node, "path")
    tree.delete(*tree.get_children(node))

    parent = tree.parent(node)
    sdirs = [] if parent else glob.glob('.') + glob.glob('..')

    for p in sdirs + os.listdir(path):

        ptype = None
        p = os.path.join(path, p).replace('\\', '/')

        if os.path.isdir(p):
            ptype = "directory"

        elif os.path.isfile(p):
            ptype = "file"

        fname = os.path.split(p)[1]
        itemId = tree.insert(node, index=END, text=fname,
            values=[p, ptype])

        if ptype == 'directory':

            if fname not in ('.', '..'):

                tree.insert(itemId, 0, text="dummy")

        elif ptype == 'file':

            size = os.stat(p).st_size
            tree.set(itemId, "size", "%d bytes" % size)

```

```

def changeDir(self, e):

    tree = e.widget
    node = tree.focus()

    if tree.parent(node):

        path = os.path.abspath(tree.set(node, "path"))

        if os.path.isdir(path):

            os.chdir(path)
            tree.delete(tree.get_children(''))
            self.loadRoot(tree)

def autoScroll(self, sbar, first, last):

    first, last = float(first), float(last)

    if first <= 0 and last >= 1:
        sbar.grid_remove()
    else:
        sbar.grid()
        sbar.set(first, last)

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

The example uses a treeview widget to display files and directories. The size of the files is shown in the second column. It is possible to change directories by clicking on the directory names.

```
cols = ("path", "type", "size")
```

There are three column names, but only the last one is displayed. Other two columns are used for storing values.

```

tree = Treeview(f, columns=cols, displaycolumns="size",
    yscrollcommand=lambda f, l:self.autoScroll(vsb, f, l),
    xscrollcommand=lambda f, l:self.autoScroll(hsb, f, l))

```

A treeview is created. The `displaycolumns` option selects which columns are displayed. We show only the size column. The `autoScroll()` method displays the scrollbars when they are needed; otherwise they are hidden.

```
tree.bind("<<TreeviewOpen>>", self.updateTree)
```

Each time we open a file which is a directory (in the Unix philosophy, everything is a file) a `<<TreeviewOpen>>` event is triggered. To this event, we bind the `updateTree()` method.

```
tree.bind("<Double-Button-1>", self.changeDir)
```

We react on double click events.

```
def loadRoot(self, tree):
    mydir = os.path.abspath(".").replace('\\', '/')
    node = tree.insert("", index=END, text=mydir,
        values=[mydir, "directory"])
    self.populateTree(tree, node)
```

The `loadRoot()` method inserts the root item into the treeview. We start with the current working directory. The root node shows its full path; other files show only the last part of their full path. The `populateTree()` method is called to load the nodes for the current working directory.

```
def updateTree(self, e):
    tree = e.widget
    self.populateTree(tree, tree.focus())
```

This method is called in reaction to the `<<TreeviewOpen>>` event. In this case, we get the currently focused item with the `focus()` method. Again, the nodes are loaded with the `populateTree()` method.

```
def populateTree(self, tree, node):
    if tree.set(node, "type") != 'directory':
        return
    ...
```

The method returns if the file is not a directory. It makes sense to load children for directories only.

```
path = tree.set(node, "path")
```

We get the file name for the currently inspected node.

```
tree.delete(*tree.get_children(node))
```

We delete the previously loaded nodes. Each time we click on an expandable tree node, new children are loaded and old ones (if there are some) are deleted.

```
parent = tree.parent(node)
```

The `parent()` method returns an empty string for top-level nodes. For other nodes, it returns the iid of their parent.

```
sdirs = [] if parent else glob.glob('.') + glob.glob('..')
```

In the `sdirs` list, we store two special directories—the current working directory and the parent directory of the current working directory.

```

if os.path.isdir(p):
    ptype = "directory"

elif os.path.isfile(p):
    ptype = "file"

```

We determine if the file path is a regular file or a directory.

```

fname = os.path.split(p)[1]
itemId = tree.insert(node, index=END, text=fname,
    values=[p, ptype])

```

Except for the root node, all nodes display only the last part of their full path. Note that we also store information in the two columns that are not visible; the stored information is a full path name and the file type.

```

if ptype == 'directory':

    if fname not in ('.', '..'):

        tree.insert(itemId, 0, text="dummy")

```

If a node is a directory, we add a dummy item. This makes the node expandable. The dummy item is never visible; it is deleted when the node is opened.

```

elif ptype == 'file':

    size = os.stat(p).st_size
    tree.set(itemId, "size", "%d bytes" % size)

```

For regular files, we add their size to the size column.

```

def changeDir(self, e):

    tree = e.widget
    node = tree.focus()
    ...

```

The `changeDir()` method is called when we double click on a node. First, we determine the node on which we have clicked.

```

path = os.path.abspath(tree.set(node, "path"))

if os.path.isdir(path):

    os.chdir(path)
    tree.delete(tree.get_children(''))
    self.loadRoot(tree)

```

To change to a new directory, we need its full path name. The full path name is stored in the path column. We change to the new directory, delete the contents of the treeview, and load a new root node.

```

def autoScroll(self, sbar, first, last):

    first, last = float(first), float(last)

    if first <= 0 and last >= 1:
        sbar.grid_remove()

```

```
else:
    sbar.grid()
    sbar.set(first, last)
```

The `autoScroll()` method shows the scrollbars only when they are needed. Both horizontal and vertical scrollbars work with two values; these values fall between 0 and 1. If the scrollbar is not needed, it is hidden with the `grid_remove()` method. The grid options are remembered and are applied later with the `grid()` method.

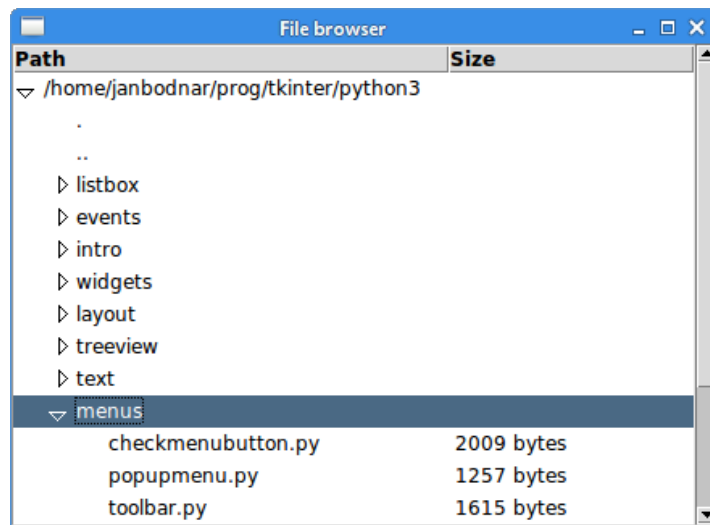


Figure 8.9: File browser

Figure 8.9 shows the File browser program. The top item is the root node. The directories have a triangle image next to them.

Chapter 9

Canvas

In this chapter we will do some drawing. Drawing in Tkinter is done on the `Canvas` widget. `Canvas` is a high-level facility for doing graphics in Tkinter.

9.1 Lines

A line is a simple geometric primitive. The `create_line()` method creates a line item on the `Canvas`.

Listing 9.1: `initUI()` from `lines.py`

```
def initUI(self):  
  
    self.parent.title("Lines")  
    self.pack(fill=BOTH, expand=True)  
  
    canvas = Canvas(self)  
    canvas.create_line(15, 25, 200, 25)  
    canvas.create_line(300, 35, 300, 200, dash=(4, 2))  
    canvas.create_line(55, 85, 155, 85, 105, 180, 55, 85)  
  
    canvas.pack(fill=BOTH, expand=True)
```

In the code example, we draw three simple lines.

```
canvas.create_line(15, 25, 200, 25)
```

The parameters of the `create_line()` method are the x and y coordinates of the start and end points of the line.

```
canvas.create_line(300, 35, 300, 200, dash=(4, 2))
```

A vertical line is drawn. The `dash` option specifies the dash pattern of the line. We have a line consisting of alternating segments of 4 px dash and 2 px space.

```
canvas.create_line(55, 85, 155, 85, 105, 180, 55, 85)
```

The `create_line()` method can take multiple points. This line draws a triangle.

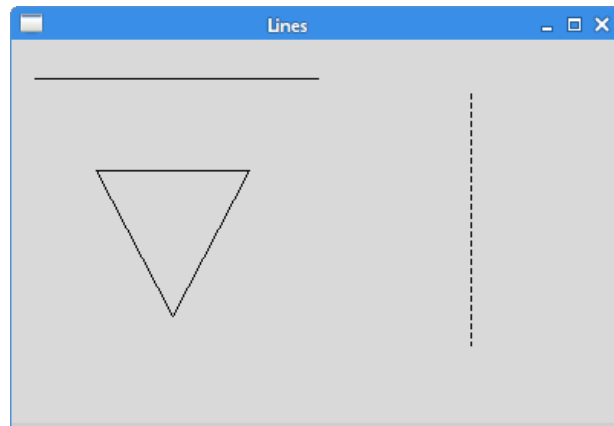


Figure 9.1: Lines

Figure 9.1 shows three lines on the window.

9.2 Line joins

Line joins are decorations applied at the intersection of two path segments and at the intersection of the endpoints of a subpath. There are three decorations: `ROUND`, `BEVEL`, and `MITER`.

Listing 9.2: `initUI()` from `linejoins.py`

```
def initUI(self):
    self.parent.title("Joins")
    self.pack(fill=BOTH, expand=True)

    canvas = Canvas(self)
    canvas.create_line(15, 25, 150, 25, 150, 80, 15, 80, 15, 25,
                      joinstyle=ROUND, width=10)

    canvas.create_line(200, 25, 320, 25, 320, 80, 200, 80, 200, 25,
                      joinstyle=BEVEL, width=10)

    canvas.create_line(15, 120, 150, 120, 150, 200, 15, 200, 15, 120,
                      joinstyle=MITER, width=10)
```

In the example, we show these three types of line joins. We draw three rectangles.

```
canvas.create_line(15, 25, 150, 25, 150, 80, 15, 80, 15, 25,
                  joinstyle=ROUND, width=10)
```

A rectangle is drawn with the `create_line()` method. The line join is specified with the `joinstyle` parameter. In order to see the joins more clearly, we increase the width of the line.

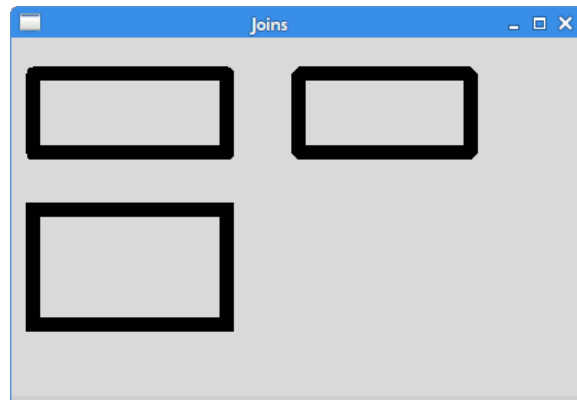


Figure 9.2: Line joins

Figure 9.2 shows three types of line joins in Tkinter.

9.3 Line caps

Caps are decorations applied to the ends of lines. There are three different caps: BUTT, ROUND, and PROJECTING.

Listing 9.3: `initUI()` from `linecaps.py`

```
def initUI(self):

    self.parent.title("Caps")
    self.pack(fill=BOTH, expand=True)

    canvas = Canvas(self)
    canvas.create_line(20, 30, 250, 30, capstyle=BUTT, width=8)
    canvas.create_line(20, 80, 250, 80, capstyle=PROJECTING, width=8)
    canvas.create_line(20, 130, 250, 130, capstyle=ROUND, width=8)

    canvas.create_line(20, 20, 20, 140)
    canvas.create_line(250, 20, 250, 140)
    canvas.create_line(254, 20, 254, 140)

    canvas.pack(fill=BOTH, expand=True)
```

In our example, we show all three types of caps.

```
canvas.create_line(20, 30, 250, 30, capstyle=BUTT, width=8)
```

A cap is specified with the `capstyle` parameter.

```
canvas.create_line(20, 20, 20, 140)
canvas.create_line(250, 20, 250, 140)
canvas.create_line(254, 20, 254, 140)
```

We draw three vertical lines to explain the differences between the caps. Lines with `ROUND` and `PROJECTING` are bigger than the line with `BUTT`. Exactly how much bigger depends on the line size. In our case a line is 8 px thick. Lines are

bigger by 8 px—4 px on the left and 4 px on the right. It should be clear from the picture.

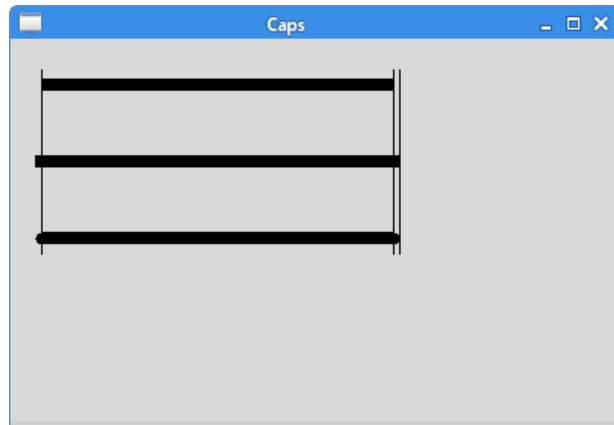


Figure 9.3: Line caps

Figure 9.3 shows three line caps.

9.4 Cubic line

So far, we have worked with straight lines. The canvas also allows to create cubic lines, also called Bézier curves.

Listing 9.4: `initUI()` from `cubic_line.py`

```
def initUI(self):  
    self.parent.title("Cubic line")  
    self.pack(fill=BOTH, expand=True)  
  
    canvas = Canvas(self)  
    canvas.create_line(25, 35, 250, 350, 380, 35, smooth=True)  
  
    canvas.pack(fill=BOTH, expand=True)
```

The example draws a cubic line on the window.

```
canvas.create_line(25, 35, 250, 350, 380, 35, smooth=True)
```

We pass three pairs of coordinates to the `create_line()` method. The `smooth` parameter turns the structure into a curve.

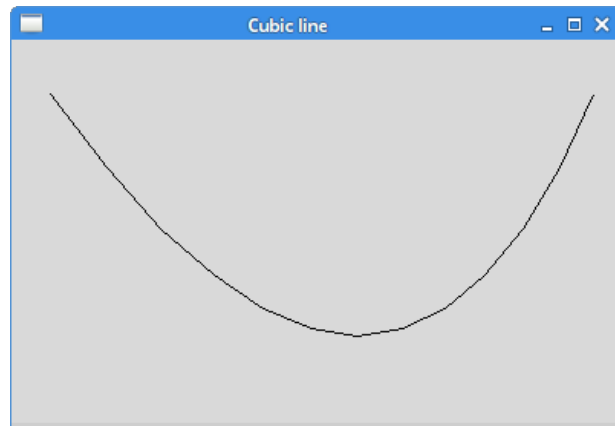


Figure 9.4: Cubic line

Figure 9.4 shows a cubic line on the canvas.

9.5 Colours

A colour is an object representing a combination of Red, Green, and Blue (RGB) intensity values.

Listing 9.5: `initUI()` from `colours.py`

```
def initUI(self):
    self.parent.title("Colours")
    self.pack(fill=BOTH, expand=1)

    canvas = Canvas(self)
    canvas.create_rectangle(30, 10, 120, 80,
        outline="#fb0", fill="#fb0")
    canvas.create_rectangle(150, 10, 240, 80,
        outline="#f50", fill="#f50")
    canvas.create_rectangle(270, 10, 370, 80,
        outline="#05f", fill="#05f")
    canvas.pack(fill=BOTH, expand=1)
```

In the code example, we draw three rectangles and fill them with different colour values.

```
canvas = Canvas(self)
```

A `Canvas` widget is created.

```
canvas.create_rectangle(30, 10, 120, 80,
    outline="#fb0", fill="#fb0")
```

The `create_rectangle()` creates a rectangle item on the canvas. The first four parameters are the x and y coordinates of the two bounding points: the top-left and bottom-right points. With the `outline` parameter we control the colour of

the outline of the rectangle. Likewise, the `fill` parameter provides a colour for the inside of the rectangle.

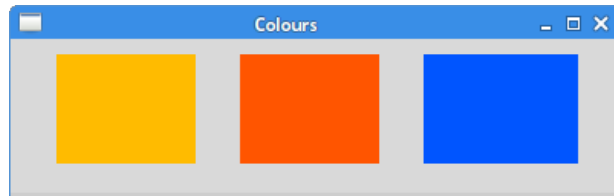


Figure 9.5: Colours

Figure 9.5 shows a three rectangles with three different colour fills.

9.6 Shapes

It is possible to draw various shapes on the canvas.

Listing 9.6: `initUI()` from `shapes.py`

```
def initUI(self):

    self.parent.title("Shapes")
    self.pack(fill=BOTH, expand=True)

    canvas = Canvas(self)
    canvas.create_oval(10, 10, 80, 80, outline="gray",
                      fill="gray", width=2)
    canvas.create_oval(110, 10, 210, 80, outline="gray",
                      fill="gray", width=2)
    canvas.create_rectangle(230, 10, 290, 60,
                           outline="gray", fill="gray", width=2)
    canvas.create_arc(30, 200, 90, 100, start=0,
                     extent=210, outline="gray", fill="gray", width=2)

    points = [150, 100, 200, 120, 240, 180, 210,
              200, 150, 150, 100, 200]
    canvas.create_polygon(points, outline='gray',
                          fill='gray', width=2)

    canvas.pack(fill=BOTH, expand=True)
```

We draw five different shapes on the window: a circle, an ellipse, a rectangle, an arc, and a polygon. Outlines and insides are drawn in gray. The width of the outline is 2 px.

```
canvas.create_oval(10, 10, 80, 80, outline="gray",
                  fill="gray", width=2)
```

Here the `create_oval()` method is used to create a circle item. The first four parameters are the bounding box coordinates of the circle. In other words, they are x and y coordinates of the top-left and bottom-right points of the box, in which the circle is drawn.

```
canvas.create_rectangle(230, 10, 290, 60,
    outline="gray", fill="gray", width=2)
```

We create a rectangle item. The coordinates are again the bounding box of the rectangle to be drawn.

```
canvas.create_arc(30, 200, 90, 100, start=0,
    extent=210, outline="gray", fill="gray", width=2)
```

This code line creates an arc. An arc is a part of the circumference of the circle. We provide its bounding box. The `start` parameter is the start angle of the arc. The `extent` is the angle size.

```
points = [150, 100, 200, 120, 240, 180, 210,
    200, 150, 150, 100, 200]
canvas.create_polygon(points, outline='gray',
    fill='gray', width=2)
```

A polygon is created. It is a shape with multiple corners. To create a polygon in Tkinter, we provide the list of polygon coordinates to the `create_polygon()` method.

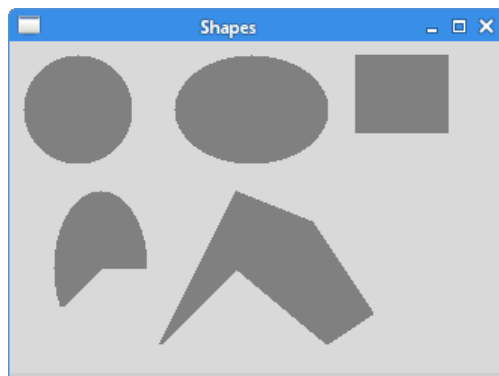


Figure 9.6: Shapes

Figure 9.6 shows five different shapes created on the canvas.

9.7 Image

In the following example we draw an image item on the canvas.

Listing 9.7: `initUI()` from `image.py`

```
def initUI(self):
    self.parent.title("High Tatras")
    self.pack(fill=BOTH, expand=True)

    self.img = Image.open("tatras.jpg")
    self.tatras = ImageTk.PhotoImage(self.img)

    canvas = Canvas(self, width=self.img.size[0]+20,
```

```
height=self.img.size[1]+20)
canvas.create_image(10, 10, anchor=NW, image=self.tatras)
canvas.pack(fill=BOTH, expand=True)
```

The example displays an image on the canvas.

```
self.img = Image.open("tatras.jpg")
self.tatras = ImageTk.PhotoImage(self.img)
```

Tkinter does not support JPG images internally. As a workaround, we use the `Image` and `ImageTk` modules.

```
canvas = Canvas(self, width=self.img.size[0]+20,
                 height=self.img.size[1]+20)
```

We create the `Canvas` widget. It takes the size of the image into account. It is 20 px wider and 20 px higher than the actual image size.

```
canvas.create_image(10, 10, anchor=NW, image=self.tatras)
```

We use the `create_image()` method to create an image item on the canvas. The image is anchored to the north-west. The `image` parameter provides the photo image to display.

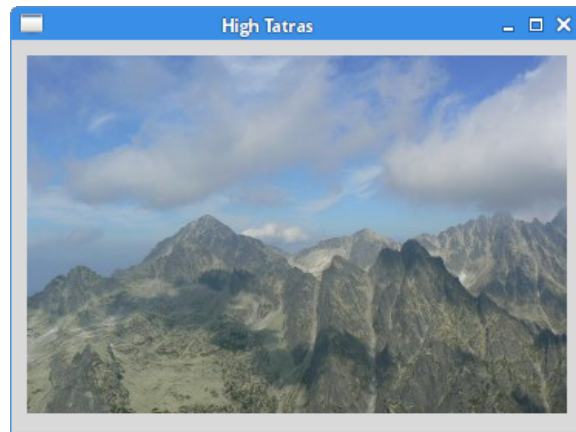


Figure 9.7: Image

Figure 9.7 shows an image on the window.

9.8 Text

Text is drawn on canvas with the `create_text()` method.

Listing 9.8: `initUI()` from `lyrics.py`

```
def initUI(self):
    self.parent.title("Lyrics")
```



```

self.pack(fill=BOTH, expand=True)

canvas = Canvas(self)
canvas.create_text(20, 30, anchor=W, font="Purisa",
    text="Most relationships seem so transitory")
canvas.create_text(20, 60, anchor=W, font="Purisa",
    text="They're good but not the permanent one")
canvas.create_text(20, 130, anchor=W, font="Purisa",
    text="Who doesn't long for someone to hold")
canvas.create_text(20, 160, anchor=W, font="Purisa",
    text="Who knows how to love without being told")
canvas.create_text(20, 190, anchor=W, font="Purisa",
    text="Somebody tell me why I'm on my own")
canvas.create_text(20, 220, anchor=W, font="Purisa",
    text="If there's a soulmate for everyone")
canvas.pack(fill=BOTH, expand=True)

```

We draw lyrics of a song on the window.

```

canvas.create_text(20, 30, anchor=W, font="Purisa",
    text="Most relationships seem so transitory")

```

The first two parameters are the x and y coordinates of the centre point of the text. If we anchor the text item to the west, the text starts from this position. The `font` parameter provides the font of the text and the `text` parameter is the text to be displayed.

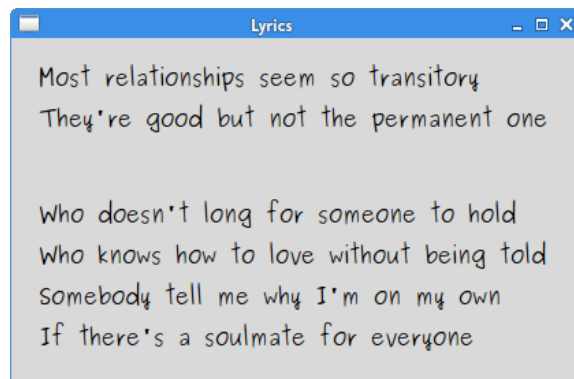


Figure 9.8: Lyrics

Figure 9.8 shows lyrics of a song on the window.

9.9 Dragging items

In the following example, we show how to drag and drop items on the canvas.

Listing 9.9: dragging.py

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

```

```

"""
ZetCode Tkinter e-book

This script enables to drag and drop
canvas items.

Author: Jan Bodnar
Last modified: January 2016
Website: www.zetcode.com
"""

from tkinter import Tk, Canvas, BOTH
from tkinter.ttk import Frame

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent
        self.initUI()

    def initUI(self):

        self.parent.title("Drag and drop items")
        self.pack(fill=BOTH, expand=True)

        self.canvas = Canvas(self)

        self.drag_data = {"x": 0, "y": 0, "item": None}

        self.canvas.create_rectangle(50, 50, 150, 150,
            fill='SteelBlue1', tag="draggable")
        self.canvas.create_rectangle(200, 150, 350, 200, fill='wheat',
            tag="draggable")

        self.canvas.tag_bind("draggable", "<ButtonPress-1>",
            self.OnItemButtonPress)
        self.canvas.tag_bind("draggable", "<B1-Motion>",
            self.OnItemMotion)

        self.canvas.pack(fill=BOTH, expand=True)

    def OnItemButtonPress(self, e):

        self.drag_data["item"] = self.canvas.find_closest(e.x, e.y)[0]
        self.drag_data["x"] = e.x
        self.drag_data["y"] = e.y

    def OnItemMotion(self, e):

        delta_x = e.x - self.drag_data["x"]
        delta_y = e.y - self.drag_data["y"]

        self.canvas.move(self.drag_data["item"], delta_x, delta_y)

        self.drag_data["x"] = e.x
        self.drag_data["y"] = e.y

```

```
def main():
    root = Tk()
    ex = Example(root)
    root.geometry("400x250+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()
```

Two rectangle objects are created. It is possible to relocate them with a mouse.

```
self.drag_data = {"x": 0, "y": 0, "item": None}
```

The `drag_data` keeps track of the item being dragged.

```
self.canvas.create_rectangle(50, 50, 150, 150,
    fill='SteelBlue1', tag="draggable")
self.canvas.create_rectangle(200, 150, 350, 200, fill='wheat',
    tag="draggable")
```

Two rectangles are created on the canvas. The items have a `draggable` tag defined.

```
self.canvas.tag_bind("draggable", "<ButtonPress-1>",
    self.OnItemButtonPress)
self.canvas.tag_bind("draggable", "<B1-Motion>",
    self.OnItemMotion)
```

We add bindings for clicking and moving any draggable item on the canvas.

```
def OnItemButtonPress(self, e):
    self.drag_data["item"] = self.canvas.find_closest(e.x, e.y)[0]
    self.drag_data["x"] = e.x
    self.drag_data["y"] = e.y
```

Inside the `OnItemButtonPress()` method we store the object to be dragged and its coordinates. The `find_closest()` method returns the item closest to the given position.

```
def OnItemMotion(self, e):
    delta_x = e.x - self.drag_data["x"]
    delta_y = e.y - self.drag_data["y"]
    ...
```

The first two lines of the `OnItemMotion()` method compute how much the current object has moved.

```
self.canvas.move(self.drag_data["item"], delta_x, delta_y)
```

The object is moved by the computed distance.

```
self.drag_data["x"] = e.x
self.drag_data["y"] = e.y
```

A new position is recorded.

9.10 Arkanoid

Arkanoid is an arcade game developed by Atari Inc. In this game, the player moves a bar and bounces a ball. The objective is to destroy bricks in the top of the window.

Listing 9.10: arkanoid.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

"""
ZetCode Tkinter e-book

This script creates an Arkanoid game.

Author: Jan Bodnar
Last modified: January 2016
Website: www.zetcode.com
"""

from tkinter import Tk, Canvas, BOTH, ALL
from tkinter.ttk import Frame
import colorsys

BAR_WIDTH = 60
BOTTOM_EDGE = 270
RIGHT_EDGE = 400
NEAR_BAR_Y = 248
BALL_INIT_X = 200
BALL_INIT_Y = 150

INIT_DELAY = 800
DELAY = 30

class Example(Frame):

    def __init__(self, parent):
        Frame.__init__(self, parent)

        self.parent = parent

        self.initVariables()
        self.initBoard()
        self.after(INIT_DELAY, self.onTimer)

    def initVariables(self):

        self.bricks = []
        self.ballvx = self.ballvy = 3
        self.ball_x = BALL_INIT_X
        self.ball_y = BALL_INIT_Y
        self.inGame = True

        self.bar_x = 170
        self.bar_y = 250
```

```

self.lives = 3

def initBoard(self):

    self.parent.title("Arkanoid")
    self.pack(fill=BOTH, expand=True)

    self.parent.config(cursor="none")

    self.canvas = Canvas(self, width=400, height=300,
                          background="#000000")
    self.canvas.bind("<Motion>", self.onMotion)

    self.bar = self.canvas.create_line(self.bar_x, self.bar_y,
                                       self.bar_x+BAR_WIDTH, self.bar_y,
                                       fill="ffffff")

    self.lives_item = self.canvas.create_text(15, 270,
                                              text=self.lives, fill="white")

    k = 0.0
    for j in range(10):
        for i in range(10):

            c = colorsys.hsv_to_rgb(k, 0.5, 0.4)
            d = hex(int(c[0]*256)<<16 | int(c[1]*256)<<8
                    | int(c[2]*256))
            d = "#" + d[2:len(d)]
            k += 0.01

            brick = self.canvas.create_rectangle(40*i,
                                                  (j*10)+20, 40+(40*i), 30+(j*10), fill=d)
            self.bricks.append(brick)

    self.ball = self.canvas.create_oval(self.ball_x-3,
                                       self.ball_y-3, self.ball_x+3, self.ball_y+3,
                                       fill="#cccccc")

    self.canvas.pack(fill=BOTH, expand=True)

def onMotion(self, e):

    self.bar_x = e.x

def onTimer(self):

    if self.inGame:
        self.doCycle()
        self.checkCollisions()
        self.after(DELAY, self.onTimer)
    else:
        self.gameOver()

def doCycle(self):

    self.ball_x += self.ballvx
    self.ball_y += self.ballvy

```

```

self.canvas.coords(self.ball, self.ball_x-3, self.ball_y-3,
                    self.ball_x+3, self.ball_y+3)

if(self.bar_x < RIGHT_EDGE - BAR_WIDTH):
    self.canvas.coords(self.bar, self.bar_x, self.bar_y,
                        self.bar_x+BAR_WIDTH, self.bar_y)

if (len(self.bricks) == 0):
    self.msg = "Game won"
    self.inGame = False

def checkCollisions(self):

    if (self.ball_x >= RIGHT_EDGE or self.ball_x <= 0):
        self.ballvx *= -1

    if (self.ball_y <= 0):
        self.ballvy *= -1

    for brick in self.bricks:

        hit = 0
        co = self.canvas.coords(brick)

        if (self.ball_x > co[0] and self.ball_x < co[2]
            and self.ball_y + self.ballvy > co[1]
            and self.ball_y + self.ballvy < co[3]):

            hit = 1
            self.ballvy *= -1

        if (self.ball_x + self.ballvx > co[0]
            and self.ball_x + self.ballvx < co[2]
            and self.ball_y > co[1]
            and self.ball_y < co[3]):

            hit = 1
            self.ballvx *= -1

        if (hit == 1):

            self.bricks.remove(brick)
            self.canvas.delete(brick)

    if ((self.ball_y > NEAR_BAR_Y and self.ball_y < self.bar_y)
        and (self.ball_x > self.bar_x
            and self.ball_x < self.bar_x + BAR_WIDTH)):

        self.ballvy *= -1

    if (self.ball_y > NEAR_BAR_Y
        and self.ball_x < self.bar_x + BAR_WIDTH/2
        and self.ballvx > 0):

        self.ballvx *= -1

    if (self.ball_y > NEAR_BAR_Y
        and self.ball_x > self.bar_x + BAR_WIDTH/2
        and self.ballvx < 0):

```

```

        self.ballvx *= -1

    if (self.ball_y > BOTTOM_EDGE):

        self.lives -= 1
        self.canvas.delete(self.lives_item)
        self.lives_item = self.canvas.create_text(15, 270,
                                                    text=self.lives, fill="white")

    if self.lives == 0:

        self.inGame = False
        self.msg = "Game lost"
    else:

        self.ball_x = BALL_INIT_X
        self.ball_y = BALL_INIT_Y

    def gameOver(self):

        self.canvas.delete(ALL)
        self.canvas.create_text(self.winfo_width()/2,
                                self.winfo_height()/2, text=self.msg, fill="white")

def main():

    root = Tk()
    ex = Example(root)
    root.geometry("+300+300")
    root.mainloop()

if __name__ == '__main__':
    main()

```

In our game, we have one bar, one ball, and one hundred bricks. A timer is used to create a game cycle. The player controls the bar with a mouse.

```

BAR_WIDTH = 60
BOTTOM_EDGE = 270
RIGHT_EDGE = 400
NEAR_BAR_Y = 248
BALL_INIT_X = 200
BALL_INIT_Y = 150

INIT_DELAY = 800
DELAY = 30

```

In the beginning, we define some constants. The `BAR_WIDTH` sets the size of the bar object. The `BOTTOM_EDGE` and the `RIGHT_EDGE` determine the boundaries of the game. The ball will bounce from this edges. The `NEAR_BAR_Y` is a place where the ball hits the bar. The `BALL_INIT_X` and `BALL_INIT_Y` are initial coordinates of the ball object. Finally, the `INIT_DELAY` and `DELAY` are initial delay and normal delay of the game's timer.

```

self.initVariables()
self.initBoard()

```

```
self.after(INIT_DELAY, self.onTimer)
```

In the `Example`'s constructor, we initiate game variables and the board and start the timer. The timer is started with an initial delay using the `after()` method.

```
def initVariables(self):

    self.bricks = []
    self.ballvx = self.ballvy = 3
    self.ball_x = BALL_INIT_X
    self.ball_y = BALL_INIT_Y
    self.inGame = True

    self.bar_x = 170
    self.bar_y = 250

    self.lives = 3
```

The `initVariables()` method initiates important variables. The `bricks` list holds the brick objects. The `ballvx` and `ballvy` are the horizontal and vertical speeds of the ball. The `ball_x` and `ball_y` are the x and y coordinates of the ball's top-left point. The `inGame` controls whether we are playing the game. The `bar_x` and `bar_y` are the x and y coordinates of the bar's top-left point. Finally, the `lives` determines how many lives we have; i.e. how many times we can miss the ball.

```
def initBoard(self):

    self.parent.title("Arkanoid")
    self.pack(fill=BOTH, expand=True)

    self.parent.config(cursor="none")
    ...
```

Inside the `initBoard()` method, we initiate the game board. With the `config()` method, we hide the cursor. The bar is controlled with the mouse and the icon of the cursor would be misleading.

```
self.canvas = Canvas(self, width=400, height=300,
                     background="#000000")
self.canvas.bind("<Motion>", self.onMotion)
```

A `Canvas` is created; its background is set to black colour. We listen for mouse motion events on the canvas.

```
self.bar = self.canvas.create_line(self.bar_x, self.bar_y,
                                   self.bar_x+BAR_WIDTH, self.bar_y,
                                   fill="ffffff")
```

A bar object is created. It is a simple line in white colour. The line is created with the `create_line()` method.

```
self.lives_item = self.canvas.create_text(15, 270,
                                          text=self.lives, fill="white")
```

A text item is created at the bottom-left corner of the window. The item shows the lives. It is created with the `create_text()` method.


```

k = 0.0
for j in range(10):
    for i in range(10):

        c = colorsys.hsv_to_rgb(k, 0.5, 0.4)
        d = hex(int(c[0]*256)<<16 | int(c[1]*256)<<8
                | int(c[2]*256))
        d = "#"+d[2:len(d)]
        k += 0.01

        brick = self.canvas.create_rectangle(40*i,
            (j*10)+20, 40+(40*i), 30+(j*10), fill=d)
        self.bricks.append(brick)

```

These lines create one hundred bricks in different colours. They are stored in the `bricks` list. A brick is a rectangle item created with the `create_rectangle()` method. The colour values are generated with the HSV (Hue, Saturation, Value) colour model.

```

self.ball = self.canvas.create_oval(self.ball_x-3,
    self.ball_y-3, self.ball_x+3, self.ball_y+3,
    fill="#ccccc")

```

A ball object is created with the `create_oval` method.

```

def onMotion(self, e):

    self.bar_x = e.x

```

The `onMotion()` method is created in reaction to the mouse move event. Inside the method, we update the x coordinate of the bar.

```

def onTimer(self):

    if self.inGame:
        self.doCycle()
        self.checkCollisions()
        self.after(DELAY, self.onTimer)
    else:
        self.gameOver()

```

Each `DELAY` ms, the `onTimer()` method is called; it controls the cycles of the game, checks for collisions, or ends the game.

```

def doCycle(self):

    self.ball_x += self.ballvx
    self.ball_y += self.ballvy

    self.canvas.coords(self.ball, self.ball_x-3, self.ball_y-3,
        self.ball_x+3, self.ball_y+3)

    if(self.bar_x < RIGHT_EDGE - BAR_WIDTH):
        self.canvas.coords(self.bar, self.bar_x, self.bar_y,
            self.bar_x+BAR_WIDTH, self.bar_y)

    if (len(self.bricks) == 0):
        self.msg = "Game won"
        self.inGame = False

```

Inside the `doCycle()` method, we update the ball's coordinates and move the ball and the bar with the `coords()` method. If there are no more bricks left, we set the `inGame` variable to false.

```
def checkCollisions(self):

    if (self.ball_x >= RIGHT_EDGE or self.ball_x <= 0):
        self.ballvx *= -1

    if (self.ball_y <= 0):
        self.ballvy *= -1
    ...
```

In the `checkCollisions()` method we check for collisions in the game. The ball changes its direction when it collides with the top, left, and right edges.

```
for brick in self.bricks:

    hit = 0
    co = self.canvas.coords(brick)
    ...
```

In this for loop, we check for collisions between the bricks and the ball. The `coords()` method can be used to move the canvas item or to determine its coordinates; in this case, we get the coordinates of the currently examined brick.

```
if (self.ball_x > co[0] and self.ball_x < co[2]
    and self.ball_y + self.ballvy > co[1]
    and self.ball_y + self.ballvy < co[3]):

    hit = 1
    self.ballvy *= -1
```

If the ball hits the top or the bottom edge of the brick, the `hit` variable is set and the vertical direction of the ball changes.

```
if (self.ball_x + self.ballvx > co[0]
    and self.ball_x + self.ballvx < co[2]
    and self.ball_y > co[1]
    and self.ball_y < co[3]):

    hit = 1
    self.ballvx *= -1
```

Likewise, if the ball hits the right or left edge of the brick, the `hit` variable is set and the horizontal direction of the ball changes.

```
if (hit == 1):

    self.bricks.remove(brick)
    self.canvas.delete(brick)
```

The brick that was hit is removed.

```
if ((self.ball_y > NEAR_BAR_Y and self.ball_y < self.bar_y)
    and (self.ball_x > self.bar_x
        and self.ball_x < self.bar_x + BAR_WIDTH)):

    self.ballvy *= -1
```

If the ball hits the bar, its vertical direction changes; it bounces off the bar.

```
if (self.ball_y > NEAR_BAR_Y
    and self.ball_x < self.bar_x + BAR_WIDTH/2
    and self.ballvy > 0):

    self.ballvy *= -1

if (self.ball_y > NEAR_BAR_Y
    and self.ball_x > self.bar_x + BAR_WIDTH/2
    and self.ballvy < 0):

    self.ballvy *= -1
```

The horizontal changes of the direction of the ball depend on what part of the bar the ball lands.

```
if (self.ball_y > BOTTOM_EDGE):

    self.lives -= 1
    self.canvas.delete(self.lives_item)
    self.lives_item = self.canvas.create_text(15, 270,
                                              text=self.lives, fill="white")

    if self.lives == 0:

        self.inGame = False
        self.msg = "Game lost"
    else:

        self.ball_x = BALL_INIT_X
        self.ball_y = BALL_INIT_Y
```

If the ball passes the bottom edge, we loose a life. The lives text item is updated. If there are no more lives left, the game is over. Otherwise, the ball reappears in its initial position and the game continues.

```
def gameOver(self):

    self.canvas.delete(ALL)
    self.canvas.create_text(self.winfo_width()/2,
                          self.winfo_height()/2, text=self.msg, fill="white")
```

In the `gameOver()` method, we delete all canvas items and create a final text message. The message is "Game won" or "Game lost", depending on our performance in the game.

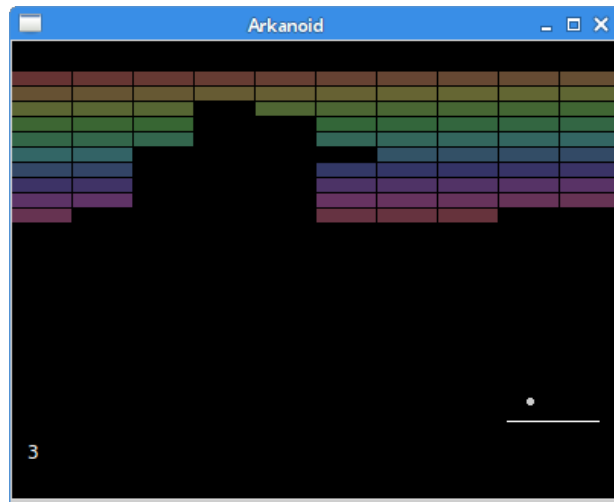


Figure 9.9: Arkanoid

Figure 9.9 shows the Arkanoid game.

Bibliography

1. Tkinter Online Reference Documentation, John W. Shipman, 2013
2. An Introduction to Tkinter, Fredrik Lundh, 2005

Index

- «ListboxSelect», 107, 108
- «TreeviewOpen», 165, 168
- «TreeviewSelect», 154, 155
- <B1-Motion>, 119
- <Button-1>, 119
- <Double-1>, 159, 161

- after(), 51, 53, 59

- BEVEL, 172
- BooleanVar, 73, 103
- BUTT, 173

- canvas, 171–190
 - Arkanoid, 182–190
 - colours, 175–176
 - cubic line, 174–175
 - dragging items, 179–182
 - image, 177–178
 - line caps, 173–174
 - line joins, 172–173
 - lines, 171–172
 - shapes, 176–177
 - text, 178–179
- capstyle, 173
- Checkbutton, 116
- colour models, 5
- column(), 147
- columnconfigure(), 29
- columnspan, 32
- command, 64
- config(), 186
- configure(), 124
- coords(), 188
- create_image(), 178
- create_line(), 171, 172, 174, 186
- create_oval(), 176, 187
- create_polygon(), 177
- create_rectangle(), 175, 187
- create_text(), 178, 186
- curselection(), 109
- custom style, 14

- default style, 12, 13
- deiconify(), 59
- delete(), 79

- edit_modified(), 142, 144
- edit_redo(), 130
- edit_undo(), 130
- Entry, 30
- event binding, 35
- event handler, 35
- event object, 35
- event source, 35
- event_generate(), 47, 49, 132
- events, 35–63
 - animation, 51–53
 - binding events, 36–38
 - binding widget class, 45–47
 - command parameter, 35–36
 - custom event, 47–49
 - event object, 41–43
 - event source, 43–45
 - floating window, 53–56
 - mouse motion event, 43
 - multiple event handlers, 40–41
 - notifications, 59–63
 - protocols, 49–50
 - splash screen, 56–59
 - unbinding events, 38–40
- extent, 177

- fill, 176
- find_closest(), 181
- focus(), 168
- Font, 11, 124
- font, 179
- Frame, 2, 3, 14, 104

- geometry(), 3, 5
- grid manager, 15, 27, 30, 32
- grid(), 170
- grid_remove(), 170

- heading(), 147

- identify(), 161
- Image, 178
- image_create(), 126
- ImageTk, 178
- introduction, 1–14
 - centering a window, 3–5
 - colours, 5–10
 - fonts, 10–12
 - simple example, 1–3
 - styles, 12
- IntVar, 75
- joinstyle, 172
- Label, 103, 104
- layout management, 15–34
 - absolute positioning, 15–17
 - calculator layout, 27–30
 - corner buttons, 19–20
 - new folder with grid, 30–32
 - new folder with pack, 23–25
 - row of buttons, 18–19
 - rows of buttons, 20–22
 - window with grid, 32
 - windows with grid, 34
 - windows with pack, 25–27
- lazy loading, 164
- Listbox, 107–120
 - adding and removing items, 113–115
 - item selection, 107–109
 - multiple selection, 109–111
 - reordering items by dragging, 117–120
 - scrolling, 111–112
 - sorting items, 115–117
- menus and toolbars, 96–106
 - check menu button, 102–104
 - popup menu, 100–102
 - simple menu, 96–98
 - submenu, 98–100
 - toolbar, 104–106
- MITER, 172
- named colours, 6
- nametowidget(), 94
- nearest(), 119
- Notebook, 14
- OptionMenu, 124
- outline, 175
- overrideredirect(), 58, 62
- pack manager, 15, 18–20, 23, 25
- pack(), 3
- pack_forget(), 104
- PhotoImage, 52, 58, 67, 153
- PROJECTING, 173
- protocol(), 49
- quit(), 3, 64
- ROUND, 172, 173
- rowconfigure(), 29
- select_clear(), 80
- select_range(), 80
- start, 177
- sticky, 30, 32
- StringVar, 87
- Style, 13, 103, 106
- Text, 121–145
 - cut, copy, paste, 130–132
 - fonts, 123–125
 - image, 126–128
 - open, save files, 138–145
 - searching text, 132–134
 - selecting text, 125–126
 - simple example, 121–122
 - spell checking, 134–138
 - undo, redo, 128
- title(), 3
- Tk, 2, 3
- Treeview, 146–170
 - double clicking a row, 159–161
 - file browser, 164–170
 - hierarchy, 150–152
 - images, 152–154
 - inserting and deleting items, 156–159
 - row colours, 148–150
 - selection, 154–156
 - simple example, 146–148
 - sorting, 161–164
- update(), 62
- virtual event, 131, 132, 154, 165
- virtual events, 35, 47
- weight, 32, 33
- widgets, 64–95
 - Button, 64–65
 - Checkbutton, 72–74
 - Combobox, 85–87
 - Entry, 76–80
 - Frame, 70–71
 - Label, 65–67
 - LabelFrame, 71–72
 - Message, 67–68

Notebook, 90–91
OptionMenu, 84–85
PanedWindow, 91–92
Progressbar, 93–95
Radiobutton, 74–76
Scale, 80–82
Scrollbar, 88–90

Separator, 68–70
Spinbox, 82–83
winfo_height(), 62
withdraw(), 59
WM_DELETE_WINDOW protocol,
49