tk tutorial Documentation

Release 2020

Raphael Holzer

Mar 14, 2020

Contents:

Tk is a **graphical user interface** (GUI) library. It allows to create windows, buttons and all the other graphical elements. This tutorial shows how to use **object-oriented programming** (OOP) for making applications with the **Tk** framework.

CHAPTER 1

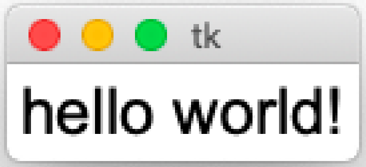
Introduction

In this tutorial you will learn to

- understand the original Tk classes
- redefine them into more powerful classes
- master object-oriented programming

1.1 Our first program

It is somewhat of a tradition for programming tutorials to start with a program which writes **hello world** to the computer screen. Here is the result of this standard initiation ritual:



So how do we do this ? First we have to **import the module** `tkinter` (Tk interface) and give it the shortcut `tk`:

```
import tkinter as tk
```

Then we create the **root widget** with class `tk.Tk` which becomes the root for all other widgets:

```
root = tk.Tk()
```

Then we create a **label widget** with class `tk.Label` which has *root* as parent and *hello world* as text attribute:

```
label = tk.Label(root, text='hello world!', font='Arial 24')
```

Then we have to call a **placement method** such as `grid()`, to make the label appear inside the window:

```
label.grid()
```

Finally we call the **main loop** method `tk.mainloop()` which runs continually until the *window close button* is clicked or the *quit Python menu* is chosen:

```
root.mainloop()
```

This method call is usully the last one in the program, after all the graphical elements and callback functions have been defined.

`intro1.py`

```
import tkinter as tk

root = tk.Tk()
label = tk.Label(master=root, text='hello world!', font='Arial 24')
label.grid()
root.mainloop()
```

1.2 The same in OOP

Now we rewrite this first program in OOP manner. We start by defining the new `App` class:

```
class App:
```

This class has two methods:

- the **constructor** method `__init__()`
- the **main loop** method `run()`

The constructor method creates the instance attributes `self.root` and `self.label`. It then calls the placement method on the label object:

```
def __init__(self):
    self.root = tk.Tk()
    self.label = tk.Label(self.root, text='hello world!', font='Arial 24')
    self.label.grid()
```

In case we do not need to keep a reference to the label object, we can shorten the last two lines to:

```
tk.Label(self.root, text='hello world!', font='Arial 24').grid()
```

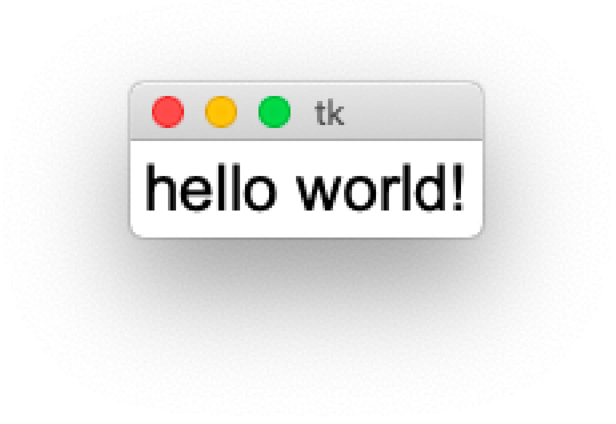
The `run` method starts the main loop:

```
def run(self):
    """Run the main loop."""
    self.root.mainloop()
```

Finally we instantiate the App and run it:

```
App().run()
```

This is the result:



```
import tkinter as tk

class App:
    """Define the application class."""
    def __init__(self):

        self.root = tk.Tk()
        self.label = tk.Label(self.root, text='hello world!', font='Arial 24')
        self.label.grid()

    def run(self):
        """Run the main loop."""
        self.root.mainloop()

App().run()
```

intro2.py

1.3 Classic and themed widgets

The elements of a graphical user interface are called **widgets**. In Tk there are two generations of widgets:

- the old **classic** `tk` widgets, originally introduced in 1991
- the new **themed** `ttk` widgets, added in 2007 with version 8.5

The new themed widgets can be found in the submodule `tkinter.ttk`. Whenever a newer themed widget is available, we will use it. We import the classic and the new themed widgets with this import statement:

```
import tkinter as tk
import tkinter.ttk as ttk
```

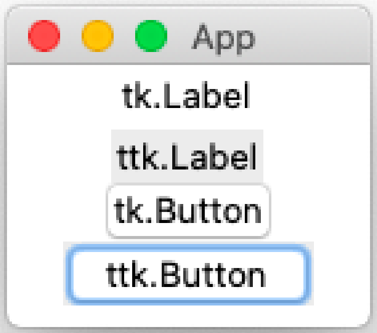
This code creates a **classic label** and a new **themed label**:

```
tk.Label(self.root, text='tk.Label').pack()
ttk.Label(self.root, text='ttk.Label').pack()
```

This code creates a **classic button** and a new **themed button**:

```
tk.Button(self.root, text='tk.Button').pack()
ttk.Button(self.root, text='ttk.Button').pack()
```

The screen capture below shows the difference in appearance. The new themed widgets (ttk) have a gray background and the buttons have uniform size.



```
import tkinter as tk
import tkinter.ttk as ttk

class App:
    """Define the application class."""
    def __init__(self):
        self.root = tk.Tk()
        self.root.title('App')

        tk.Label(self.root, text='old classic tk.Label').grid()
        ttk.Label(self.root, text='new themed ttk.Label').grid()

        tk.Button(self.root, text='tk.Button').grid()
        ttk.Button(self.root, text='ttk.Button').grid()

    def run(self):
        """Run the main loop."""
        self.root.mainloop()

App().run()
```

intro3.py

1.4 Setting options

Options can be set in three ways:

- at object creation, using **keyword arguments**
- after object creation, using a **dictionary index**
- use the **config() method** with keyword attributes

```python
# setting options
import tkinter as tk
import tkinter.ttk as ttk

root = tk.Tk()
ttk.Button(root, text='Set options at creation').grid()

button = ttk.Button(root)
button['text'] = 'Set options as dict'
button.grid()

button = ttk.Button(root)
button.config(text='Set options with config() method')
button.grid()

options = button.config()
for x in options:
    print(x, options[x])

print(button.keys())

root.mainloop()
```

intro4.py

1.5 Let's define our own widget class

We are now going to redefine the original tk and ttk classes to make our own Tk widget classes. This new classes have the following advantages:

- the **text** option has a default value (*Label, Button, etc.*)
- the **parent** object is automatically set (root)
- all **keyword** arguments are passed on (kwargs)
- the **themed** version is used when available (ttk)

This is our new `Label` class based on the original themed class:

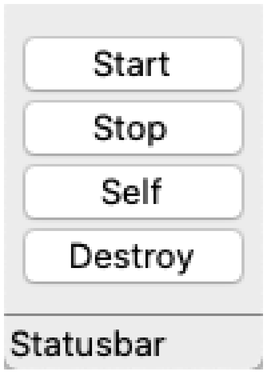
```python
class Label(ttk.Label):
    """Create a Label object."""
    def __init__(self, text='Label', **kwargs):
        super().__init__(App.stack[-1], text=text, **kwargs)
        self.grid()
```

This is the new `Button` class:

```
class Button(ttk.Button):
    """Create a Button object."""
    def __init__(self, text='Button', **kwargs):
        super().__init__(App.stack[-1], text=text, **kwargs)
        self.grid()
```

1.6 Button

Buttons can be clicked and are used to execute a command associated with them. The following demo creates 4 buttons.



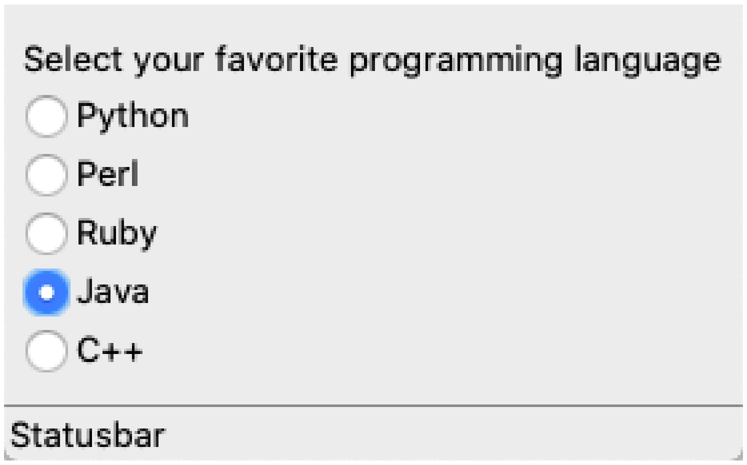
- The **Start** button prints *Start* to the console
- The **Stop** button prints *Stop* to the console
- The **Self** button prints the button object string to the console
- The **Destroy** button removes the button from the window

```
from tklib import *
app = App('Buttons')

Button('Start', 'print("Start")')
Button('Stop', 'print("Stop")')
Button('Self', 'print(self)')
Button('Destroy', 'self.destroy()')

app.run()
```

intro5.py

1.7 Radiobutton

Radiobuttons are active elements which can be clicked and execute actions. Only one button is active at any one time.

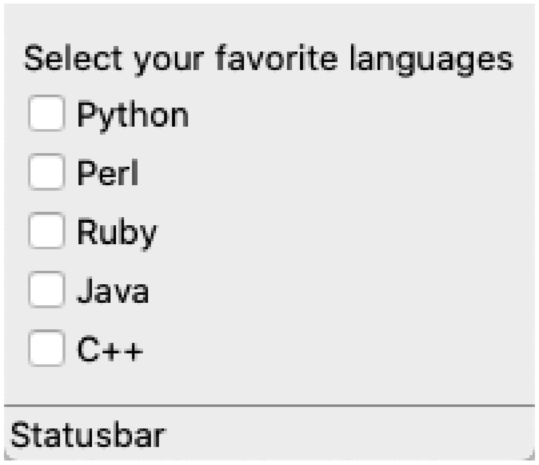


```
from tklib import *
app = App('Radio buttons')

Label('Select your favorite programming language')
Radiobutton('Python;Perl;Ruby;Java;C++', 'print(self.item)')

app.run()
```

intro6.py

1.8 Checkbutton

Checkbuttons are active elements which can be clicked and execute actions. Multiple checkbuttons can be selected simultaneously.

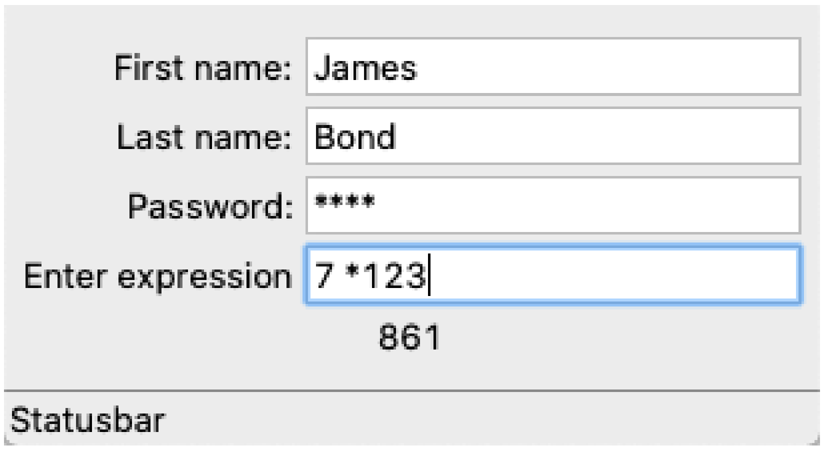


```
from tklib import *
app = App('Checkbuttons')

Label('Select your favorite languages')
Checkbutton('Python;Perl;Ruby;Java;C++', 'print(self.selection)')

app.run()
```

intro7.py

1.9 Entry field

Entry **entry** field presents the user with a single line text field where he can enter a string value.



```
from tklib import *
app = App('Entry fields')

Entry('First name:', 'print(self.var.get())')
Entry('Last name:', 'print(self.var.get())')
Entry('Password:', 'print(self.var.get())', show='*')

Entry('Enter expression', 'App.res["text"] = eval(self.var.get())')
App.res = Label('Result')
app.run()
```

intro8.py

1.10 The `tklib` module

In the following section we are going to redefine the `tk` and `ttk` objects. To make it easier to use them, we follow these design principles:

- we keep the excact same class names
- the parent object is chosen automatically
- all keyword arguments are forwarded

The first widget to consider is the `Label` which just places static text. Where it makes sense, a label will be combined with a button or entry widget.

There are three types of buttons:

- Button
- Checkbutton
- Radiobutton

There are four types of entry widgets for text or numbers, which allow to input text with the keyboard or making a choice uint the mouse:

- Entry
- Combobox
- Spinbox
- Scale

Then there are the two complex display widgets for text and graphics:

- Text
- Canvas

These two widgets present lists:

- Listbox
- Treeview

Finally these widgets are helper widgets:

- Frame
- Separator
- Window
- Scrollbar
- SizeGrip
- Progressbar
- Menu
- ContextMenu
- Panedwindow

CHAPTER 2

Button

Let's start with a simple example of creating a button which displays some text.

2.1 A one-button program

First we import the `tkinter` module and give it the shorter name `tk`:

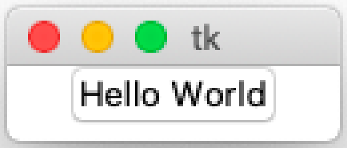
```
import tkinter as tk
```

Then we create a `Tk` root object which represents the window and will be the parent for all the other widgets:

```
root = tk.Tk()
```

Then we create a button object which has root as parent, and to which we give the text *Hello world*. This object calls the method `pack()` which makes the button visible in the window. Finally root calls the `mainloop()` method to start the program:

```
tk.Button(root, text="Hello World").pack()
root.mainloop()
```

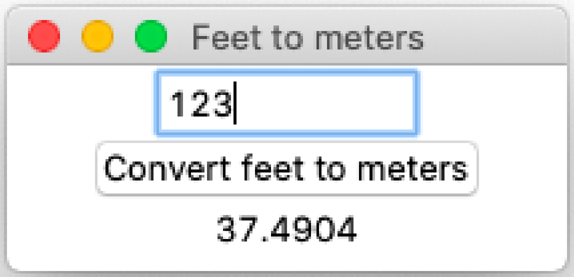


```
"""Create a simple button."""
import tkinter as tk

root = tk.Tk()
tk.Button(root, text="Hello World").pack()
root.mainloop()
```

button1.py

2.2 Convert feet to meters

Now let's create a real application which does something useful. The following program has an **input** entry field, a **button** and an **output** label. When you press the button (or hit the return key) it converts feet to meters.



After importing the **classic Tk** module as `tk` we create the `root` object and set a descriptive window title:

```
import tkinter as tk

root = tk.Tk()
root.title("Feet to meters")
```

Two of the widgets, the entry widget and the label widget, have special `StringVar` variables:

```
feet = tk.StringVar()
meters = tk.StringVar()
```

Now it's time to create the three widgets:

- an entry widget with the text variable `feet`
- a button widget with the command function `calculate`
- a label widget with the text variable `meters`

All three widgets are placed with the `pack()` method:

```
tk.Entry(root, width=10, textvariable=feet).pack()
tk.Button(root, text='Convert feet to meters', command=calculate).pack()
tk.Label(root, textvariable=meters).pack()
```

Finally we bind the `calculate` function also to the **Return** key and start the main loop:

```
root.bind('<Return>', calculate)
root.mainloop()
```

The conversion is done by calling the `calculate` function which gets the feet value from the `StringVar` **feet**, converts the value to meters and sets the `StringVar` **meters**. We enclose the calculation inside a `try-except` statement to account for value errors, in case the input string is not numeric.

```
def calculate(*args):
    try:
        value = float(feet.get())
        meters.set(0.3048 * value)
    except ValueError:
        pass
```

`button2.py`

```
import tkinter as tk

root = tk.Tk()
root.title("Feet to meters")

feet = tk.StringVar()
meters = tk.StringVar()

def calculate(*args):
    try:
        value = float(feet.get())
        meters.set(0.3048 * value)
    except ValueError:
        pass

tk.Entry(root, width=10, textvariable=feet).pack()
tk.Button(root, text='Convert feet to meters', command=calculate).pack()
tk.Label(root, textvariable=meters).pack()

root.bind('<Return>', calculate)
root.mainloop()
```

2.3 Concepts

To understand Tk you need to understand:

- widgets
- geometry management
- events

Widgets are the things you can see on the screen, for example a label, an entry field or a button. Later you will see checkboxes, radiobuttons, and listboxes. Widgets are sometimes referred to as controls.

Widgets are objects, instances of classes. In the example above we had the following 2-level hierarchy:

- root (Tk)
  - entry (Entry class)
  - button (Button class)
  - label (Label class)

When you create a widget, you must pass its parent as the first argument. Whether you save the object under a variable is up to you. It depends if you need to refer to it later on. Because it is inserted into the widget hierarchy, it won't be garbage collected.

Just creating the widget does not yet display it in the window. You need to make a call to a geometry manager.

2.4 Event bindings

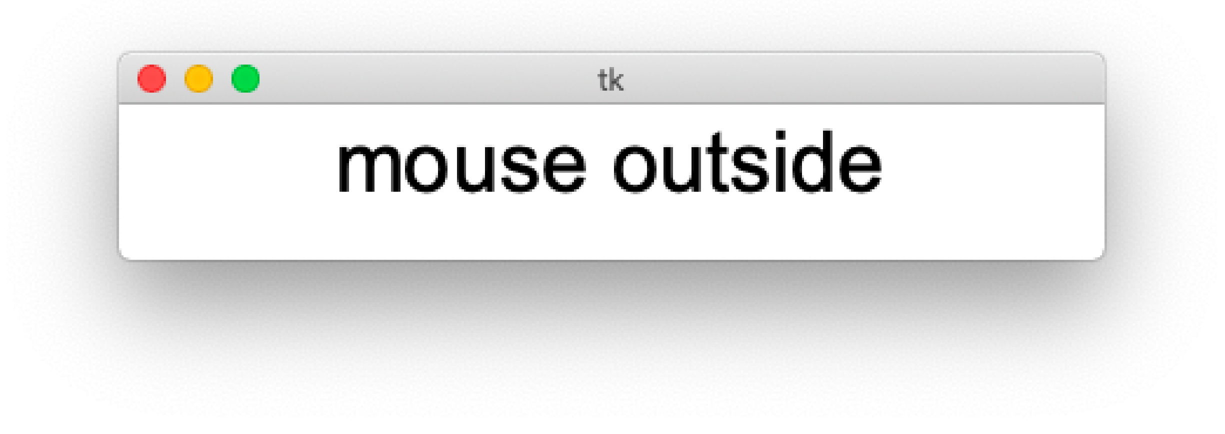
Each widget in Tk can have event bindings. In the following example we bind a function to these events:

- enter the widget
- leave the widget
- click the mouse button

Tk expects an event callback function which has an **event object** as its first argument. Here we define a `lambda` function with an event argument `e`, but we do not use it here.

Event handlers can be set up for:

- the individual widget
- a class of widgets
- the toplevel window



```
import tkinter as tk

root = tk.Tk()
label = tk.Label(root, text='starting...', font='Arial 36')
label.pack()

label.bind('<Enter>', lambda e: l.configure(text='mouse inside'))
label.bind('<Leave>', lambda e: l.configure(text='mouse outside'))
label.bind('<1>', lambda e: l.configure(text='mouse click'))

root.mainloop()
```

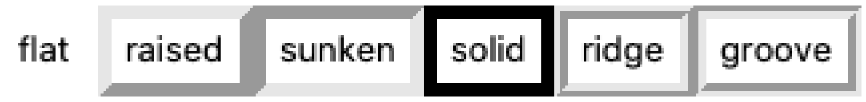
bind1.py

2.5 Frame

The frame widget displays just a rectangle. It primarily is used as a container for other widgets. A frame has these common options:

- padding - extra space inside the frame
- borderwidth
- relief (flat, raised, sunken, solid, ridge, grove)
- width
- height

The example below shows the 6 different relief types and uses a borderwidth of 5.



```
import tkinter as tk

root = tk.Tk()
```

(continues on next page)

(continued from previous page)

```
for x in ('flat', 'raised', 'sunken', 'solid', 'ridge', 'groove'):
    frame = tk.Frame(root, relief=x, borderwidth=5)
    frame.pack(side='left')
    tk.Label(frame, text=x).pack()

root.mainloop()
```

frame1.py

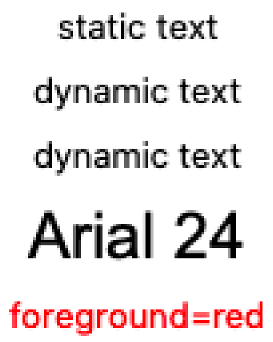
2.6 Label

The label widget displays a static text, for example the text next to an entry. User do normally not interact with a label.

Common options are:

- `text` - a static text
- `textvariable` - a dynamic text from a variable
- `image` - an image to be displayed
- `compound` - center, top, bottom, left, right (text position in in relation to image)
- `justifiy` - left, center, right
- `wraplength` - linelength for long labels

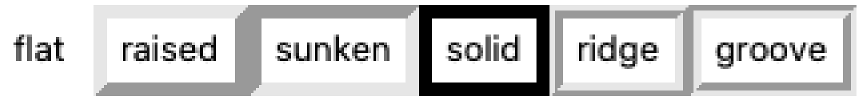
The following example shows a label with a static text and another with a dynamic text. If both are defined, the dynamic text will have precedence. The `font` keyword takes a string with the font family and font size. The `foreground` keyword takes a color string.



```
import tkinter as tk

root = tk.Tk()
str = tk.StringVar(value='dynamic text')

tk.Label(root, text='static text').pack()
tk.Label(root, textvariable=str).pack()
tk.Label(root, text='both', textvariable=str).pack()
tk.Label(root, text='Arial 24', font='Arial 24').pack()
tk.Label(root, text='foreground=red', foreground='red').pack()

root.mainloop()
```

`label1.py`

Labels also accept the `relief` keyword.



```
import tkinter as tk

root = tk.Tk()

for x in ('flat', 'raised', 'sunken', 'solid', 'ridge', 'groove'):
    tk.Label(root, text=x, relief=x, borderwidth=5).pack(side='left')

root.mainloop()
```

`label2.py`

2.7 Button

Buttons have a `command` keyword which allows to specify a function. This function is called, but without an argument. We can use the `lambda` function to create a function on the fly and provide an argument.

```
import tkinter as tk
root = tk.Tk()

def callback(x):
    print('button', x)

tk.Button(text='Button 1', command=lambda : callback(1)).pack()
tk.Button(text='Button 2', command=lambda : callback(2)).pack()
tk.Button(text='Button 3', command=lambda : callback(3)).pack()

root.mainloop()
```

`button3.py`

Pressing the 3 buttons one after another writes this to the console:

```
button 1
button 2
button 3
```

CHAPTER 3

Radio Button

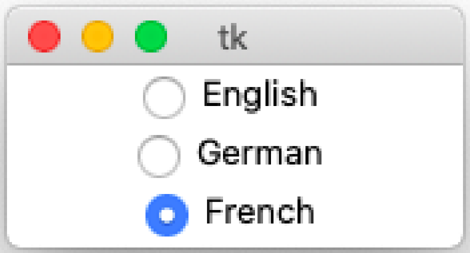
A **radiobutton** lets you choose among a number of mutually exclusive options. Radiobuttons are used together in a set and are appropriate when the number of choices is fairly small (about 3-5).

A `Radiobutton` object has these attributes:

- **parent** - the parent object
- **text** - the text label to display
- **command** - the callback function
- **variable** - the variable shared among radiobuttons of a set
- **value** - the specific value of the radiobutton

3.1 Standard Radiobutton

Let's make a program which displays 3 radiobuttons to choose a language.



```python
import tkinter as tk
root = tk.Tk()

var = tk.StringVar()
var.set('English')

def cb():
    print(var.get())

tk.Radiobutton(root, text='English', variable=var, value='English', command=cb).pack()
tk.Radiobutton(root, text='German', variable=var, value='German', command=cb).pack()
tk.Radiobutton(root, text='French', variable=var, value='French', command=cb).pack()

root.mainloop()
```

radio1.py

The radiobutton code consists of 7 lines and has a lot of repeated parts.

3.2 Using a list

A better way would be to use a list. The code still has 7 lines, but when we increase the number of items, the code length remains constant.

```python
import tkinter as tk
root = tk.Tk()

items = ('English', 'German', 'French', 'Italian', 'Spanish')
var = tk.StringVar()
var.set(items[0])

def cb():
    print(var.get())

for x in items:
    tk.Radiobutton(root, text=x, variable=var, value=x, command=cb).grid()
```

(continues on next page)

(continued from previous page)

```
root.mainloop()
```

radio2.py

3.3 A better Radiobutton class

It's time now to redefine the `Radiobutton` class to create everything in just one line:

```
Radiobutton('English;German;French', 'print(self.item)')
```

- the items are declared as a semicolon-separated list
- the command is a string to be evaluated in the Radiobutton environment
- the selected value is available in `self.item`
- the selection index is available in `self.val` (could be used as a list index)

```
"""Create radiobuttons."""
from tklib import *

app = App()
Radiobutton('English;German;French', 'print(self.item)')
app.run()
```

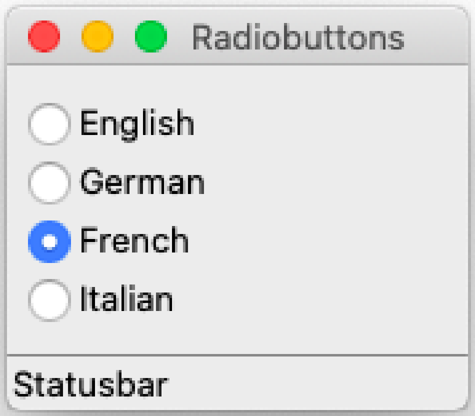
radio3.py

Now let's see how this class is defined

```
class Radiobutton:
    """Create a list-based Radiobutton object."""

    def __init__(self, items='Radiobutton', cmd='', val=0, **kwargs):
        self.items = items.split(';')
        self.cmd = cmd
        self.val = tk.IntVar()
        self.val.set(val)
        for i, item in enumerate(self.items):
            r = ttk.Radiobutton(App.stack[-1], text=item, variable=self.val,
                                value=i, command=self.cb, **kwargs)
            r.grid(sticky='w')

    def cb(self):
        """Evaluate the cmd string in the Radiobutton context."""
        self.item = self.items[self.val.get()]
        exec(self.cmd)
```

The item string is split at the semicolon into a regular list. The shared variable is a `IntVar` object. Each radiobutton has an integer value (0, 1, 2, . . .). The callback function finds the selected item by looking up this integer value in the items list.

Let's look at another exemple. This time we add another language (Italian) and initialize the default button to 2 (French).

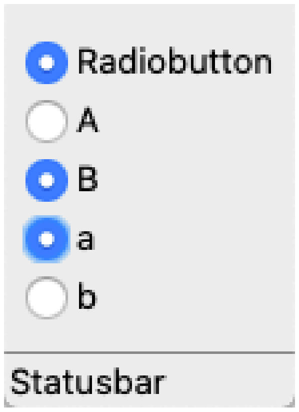


```python
from tklib import *

app = App('Radiobuttons')
Radiobutton('English;German;French;Italian', 'print(self.item)', 2)
app.run()
```

radio4.py

3.4 If something goes wrong

Let’s see what happens if there are errors in the options.



If there is no items list, there will be a single item called `Radiobutton`. If there is an error in the expression, this message is written to the console:

```
File "/Users/raphael/GitHub/tk-tutorial/docs/radio/tklib.py", line 238, in cb
    exec(self.cmd)
File "<string>", line 1
    print(self.item
                  ^
SyntaxError: unexpected EOF while parsing
```

```
from tklib import *

app = App('Radiobuttons')
Radiobutton()                                # no item list
Radiobutton('A;B', 'print(self.item')    # error in command
Radiobutton('a;b', 'print(self.item)')   # correct

app.run()
```

radio5.py

CHAPTER 4

Checkbutton

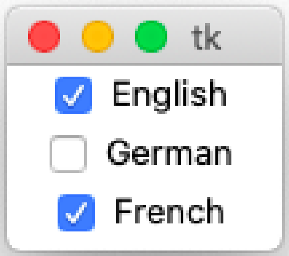
A **checkbutton** is like a regular button, with a label and a callback function, but it also holds and displays a binary state.

The `Checkbutton` object has the following attributes:

- **parent** - the parent object
- **text** - the label to display
- **command** - the callback function
- **variable** - the variable holding the state value
- **onvalue** - the value when in the ON state
- **offvalue** - the value when in the OFF state

4.1 The standard checkbutton

Here is an example of 3 checkbuttons



The callback function `cb` writes this to the console:

```
--- languages ---
English 1
German 0
French fluent
```

We notice that the default offvalue is `0` and the default onvalue is `1`. In our case:

- var0 toggles between `0` and `1`
- var1 toggles between `barely` and `1`
- var2 toggles between `0` and `fluent`

```
import tkinter as tk

root = tk.Tk()

var0 = tk.StringVar(value='1')
var1 = tk.StringVar(value='0')
var2 = tk.StringVar(value='0')

def cb():
    print('--- languages ---')
    print('English', var0.get())
    print('German', var1.get())
    print('French', var2.get())

tk.Checkbutton(root, text='English', variable=var0, command=cb).pack()
tk.Checkbutton(root, text='German', variable=var1, offvalue='barely', command=cb).
↪pack()
tk.Checkbutton(root, text='French', variable=var2, onvalue='fluent', command=cb).
↪pack()

root.mainloop()
```

`check1.py`

Now let us rewrite this program by using lists.

```
import tkinter as tk

root = tk.Tk()

texts = ['English', 'German', 'French']
vars = [tk.StringVar(value='0'), tk.StringVar(value='0'), tk.StringVar(value='0')]

def cb():
    print('--- languages ---')
    for i, s in enumerate(texts):
        print(s, vars[i].get())

tk.Checkbutton(root, text=texts[0], variable=vars[0], command=cb).pack()
tk.Checkbutton(root, text=texts[1], variable=vars[1], command=cb).pack()
tk.Checkbutton(root, text=texts[2], variable=vars[2], command=cb).pack()

root.mainloop()
```

check2.py

4.2 A better Checkbutton class

It’s time now to define a new and better `Checkbutton` class which can do everything in one line:

```
Checkbutton('English;German;French', 'print(self.selection)')
```

- the items are declared as a **semicolon-separated list**
- the command is a string to be evaluated in the Checkbutton environment
- the items are available in `self.items`
- the selected items are available in `self.selection`
- the selection states are available in `self.val`

This is the result written to the console for three consecutive selections:

```
['French']
['German', 'French']
['English', 'German', 'French']
```

```
"""Create checkbuttons."""
from tklib import *

app = App()
Checkbutton('English;German;French', 'print(self.selection)')
app.run()
```

check3.py

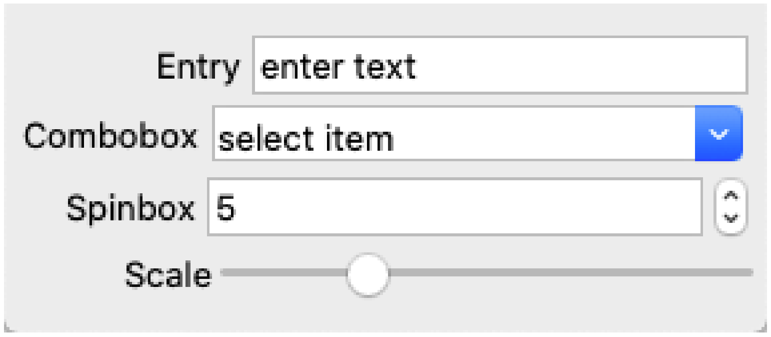
Now let’s see how this class is defined

CHAPTER 5

Entry, Combox, Spinbox

Several widgets allow to enter text or numerical information in a single field.

- the **Entry** widgets allows to type text
- the **Combobox** allows to either type text or select text from a drop-down list
- the **Spinbox** allows to select from a numerical range or from a list
- the **Scale** allows to use a slider to choose a value



`entry0.py`

Since these widgets are very similar, we treat them in the same section.

5.1 Entry widget

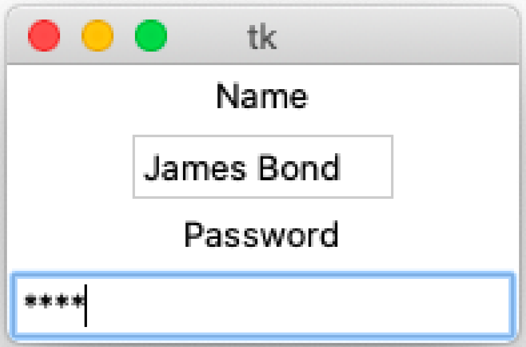
An **entry** widget presents the user an empty field where he can enter a text value.

Each `ttk.Entry` object has these options:

- **parent** - the parent object
- **textvariable** - the text variable to hold the entered string
- **width** - the numbers of characters

- **show** - to indicate * for passwords

The **Entry** widget does not have a `text` or `image` option. You have to use an additional label widget instead.



```
import tkinter as tk
root = tk.Tk()

name = tk.StringVar()
tk.Label(text='Name').pack()
tk.Entry(root, textvariable=name, width=10).pack()

password = tk.StringVar()
tk.Label(text='Password').pack()
tk.Entry(root, textvariable=password, show='*').pack()

root.mainloop()
```

entry1.py

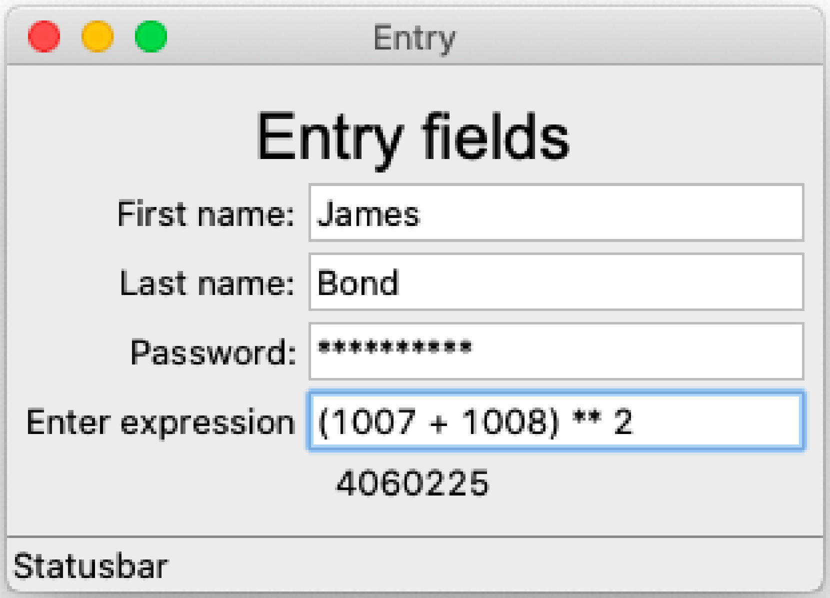
5.2 A better Entry class

It's time now to define a new and better `Entry` class which can do everything in one line:

```
Entry('First name:', 'print(self.var.get())')
```

This new class has the attributes:

- **label** - to automatically add label in front of the entry field
- **cmd** - to execute a command string when hitting the Return key
- **val** - a default value



Entry fields

First name: James

Last name: Bond

Password: ***********

Enter expression (1007 + 1008) ** 2

4060225

Statusbar

The command function evaluates the expression entered and displays the result in the following label widget:

```
Entry('Enter expression', 'App.res["text"] = eval(self.var.get())')
App.res = Label('Result')
```

```
"""Create entry fields."""
from tklib import *
app = App('Entry')

Label('Entry fields', font='Arial 24')
Entry('First name:', 'print(self.var.get())', 'James')
Entry('Last name:', 'print(self.var.get())')
Entry('Password:', 'print(self.var.get())', show='*')

Entry('Enter expression', 'App.res["text"] = eval(self.var.get())')
App.res = Label('Result')

app.run()
```

entry2.py

Now let's see how this class is defined

```
class Entry(ttk.Entry, EntryMixin):
    """Create an Entry object with label and callback."""

    def __init__(self, label='', cmd='', val='',  **kwargs):
```

(continues on next page)

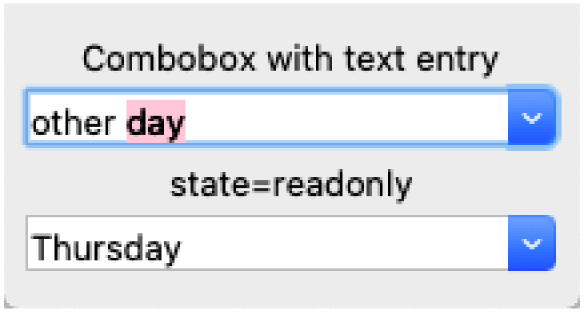
(continued from previous page)

```
        self.var = tk.StringVar()
        self.var.set(val)

        self.add_widget(label, Entry, kwargs)
        self['textvariable'] = self.var
        self.add_cmd(cmd)
```

5.3 Combobox

A **combobox** combines an entry widget with a list of choices. The user can either select text from a drop-down menu or write his own text.

The first combobox allows to also enter your own text, while the second one is restricted to chose an item from the drop-down menu by setting `state='readonly'`.



The **Combobox** class has the options

- **parent** - for the parent object
- **textvariable** - for the variable which stores the value
- **values** - for the items list
- **state** - to indicate `readonly`

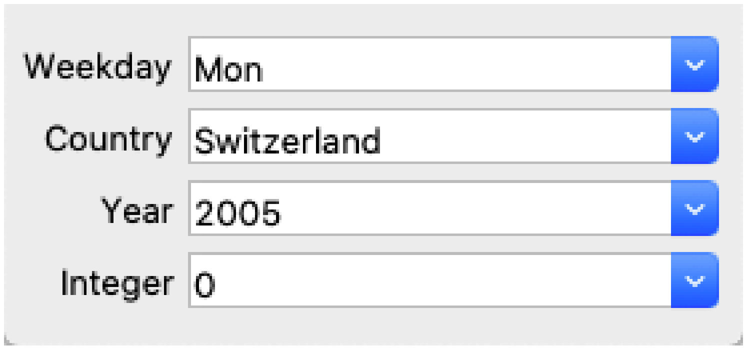
5.4 Standard Combobox

```
from tklib import *
app = App('Combobox')

Label('Combobox with text entry')
days = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday')
day = tk.StringVar()
day.set(days[0])
ttk.Combobox(App.stack[-1], textvariable=day, values=days).grid()

Label('state=readonly')
day2 = tk.StringVar()
day2.set(days[1])
ttk.Combobox(App.stack[-1], textvariable=day2, values=days, state='readonly').grid()

app.run()
```

combo1.py

5.5 A better Combobox class

A **combobox** combines a list of choices with an entry. The user can select from the list, but he can also enter directly a value.



The items list can be a:

- semicolon-separated string
- integer list: [2005, 2006, 2007]
- list expression (list(range10))

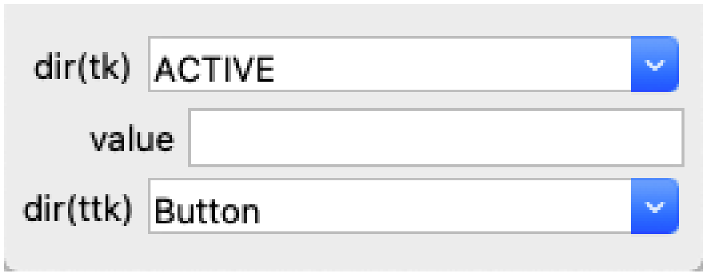
```
from tklib import *
app = App('Combobox')

Combobox('Weekday', 'Mon;Tue;Wed;Thu;Fri', 'print(self.item)')
Combobox('Country', 'Switzerland;France;Italy;Germany', 'print(self.item)')
Combobox('Year', [2005, 2006, 2007], 'print(self.item)')
Combobox('Integer', list(range(10)), 'print(self.item)')

app.run()
```

combo2.py

How is this new class defined ?

```
class Combobox(ttk.Combobox, EntryMixin):
    """Create a Combobox with label and callback."""

    def __init__(self, label='', values='', cmd='', val=0, **kwargs):
        if isinstance(values, str):
            values = values.split(';')

        self.var = tk.StringVar()
        self.var.set(values[val])

        self.add_widget(label, Combobox, kwargs)
        self['textvariable'] = self.var
        self['values'] = values

        self.add_cmd(cmd)
        self.bind('<<ComboboxSelected>>', self.cb)
```

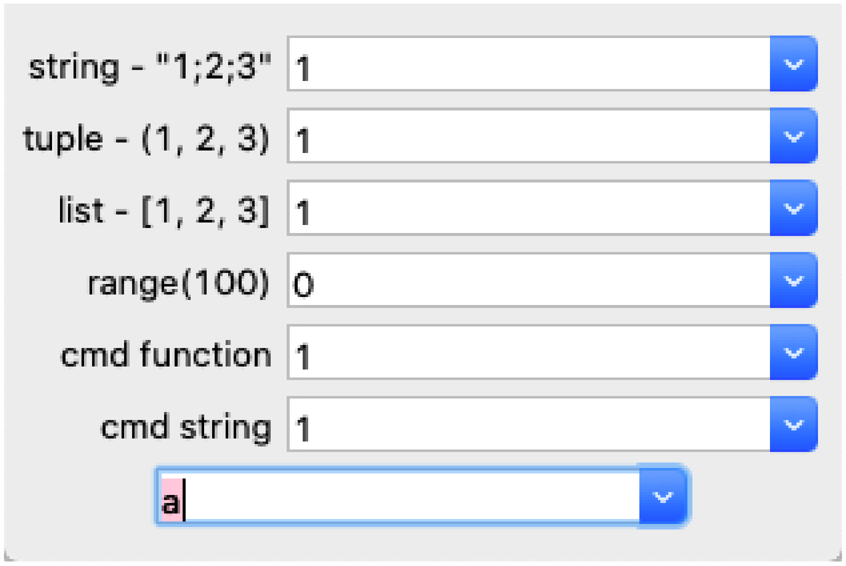
5.6 Exemple



```
from tklib import *
app = App('Combobox')

Combobox('dir(tk)', dir(tk), 'print(self.item); App.entry.var.set(eval("tk."+self.
↪item))')
App.entry = Entry('value')
Combobox('dir(ttk)', dir(ttk), 'print(eval("ttk."+self.item))')

app.run()
```

combo3.py

5.7 Another exemple



The items list can be a:

- semicolon-separated string
- tuple
- list
- list expression (list(range10))

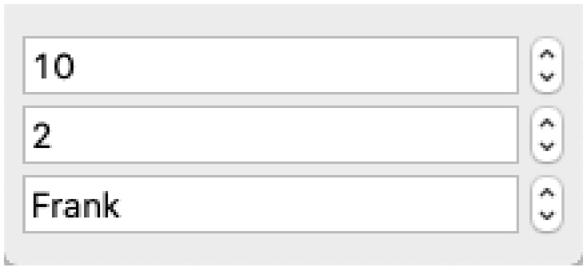
The command function can be either a string to execute or a function.

```
from tklib import *
app = App('Combobox')

def cb(e=None):
    print(c1.var.get())

Combobox('string - "1;2;3"', '1;2;3')
Combobox('tuple - (1, 2, 3)', (1, 2, 3))
Combobox('list - [1, 2, 3]', [1, 2, 3])
Combobox('range(100)', list(range(100)))
Combobox('width=10', '1;2;3', width=10)

c1 = Combobox('cmd function', (1, 2, 3), cb)
Combobox('cmd string', (1, 2, 3), 'print(self.item)')
Combobox(values='a;b;c', cmd='print(self.item)')
Combobox(values='a;b;c', cmd='print(self.item)', width=10)

app.run()
```

combo4.py

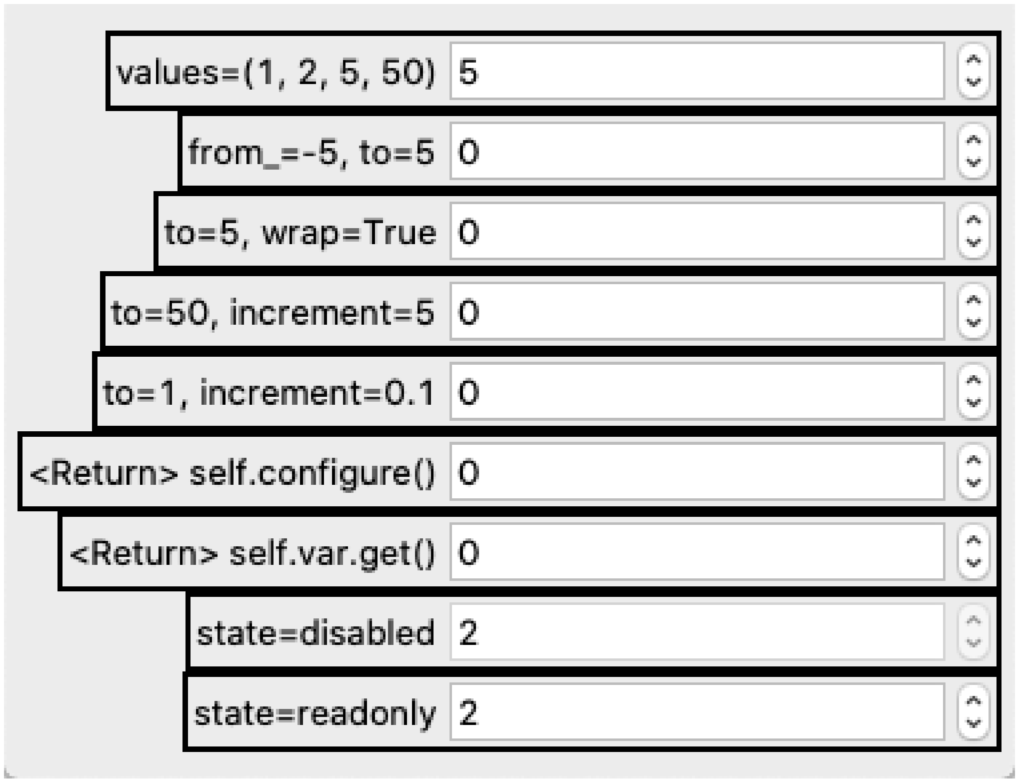
5.8 Spinbox

A **spinbox** widget is an entry widget with built-in up and down buttons that are used to either modify a numeric value or to select among a set of values. The widget implements all the features of the entry widget.



```
from tklib import *

app = App('Spinbox')

var1 = tk.StringVar(value='10')
ttk.Spinbox(App.stack[-1], from_=5.0, to=100.0, increment=25, textvariable=var1).
↪grid()

# default increment=1, from_=0, to=0
var2 = tk.StringVar(value='2')
ttk.Spinbox(App.stack[-1], to=10, textvariable=var2).grid()

# how to use a value list
values = 'Frank;Conny;Jim'.split(';')
var3 = tk.StringVar(value=values[0])
ttk.Spinbox(App.stack[-1], values=values, textvariable=var3).grid()

app.run()
```

spinbox1.py

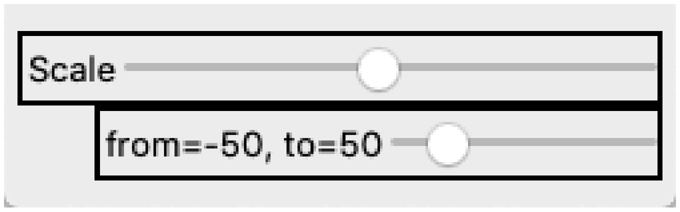


```
from entry import *
app = App('Spinbox')

Spinbox('values=(1, 2, 5, 50)', values=(1, 2, 5, 10), val=5)
Spinbox('from_=-5, to=5', from_=-5, to=5, wrap=True)
Spinbox('to=5, wrap=True', to=5, wrap=True)
Spinbox('to=50, increment=5', to=50, increment=5)
Spinbox('to=1, increment=0.1', to=1, increment=0.1)
Spinbox('<Return> self.configure()', 'print(self.configure())')
Spinbox('<Return> self.var.get()', 'print(self.var.get())', to=5)
Spinbox('state=disabled', to=5, state='disabled', val=2)
Spinbox('state=readonly', to=5, state='readonly', val=2)
app.run()
```

spinbox2.py

5.9 Scale

A scale widget provides a way for users to choose a numeric value through direct manipulation.



```
from entry import *

app = App('Scale')
Scale('Scale', 'print(self.var.get())', to=10, length=200)
Scale('from=-50, to=50', 'print(self.var.get())', from_=-50, to=50)

app.run()
```

scale2.py

5.10 Final implementation

The four classes `Entry`, `Combobox`, `Spinbox` and `Scale` have two common parts:

- adding an optional label in front of the widget
- adding a callback function

This two functions can be placed in a specal class called `EntryMixin`, which serves as second parent class for the 4 entry classes.

```
class EntryMixin:
    """Add label, widget and callback function."""

    def add_widget(self, label, widget, kwargs):
        """Add widget with optional label."""
        if label == '':
            super(widget, self).__init__(App.stack[-1], **kwargs)
            self.grid()
        else:
            d = 2 if App.debug else 0
            frame = ttk.Frame(App.stack[-1], relief='solid', borderwidth=d)
            frame.grid(sticky='e')
            ttk.Label(frame, text=label).grid()
            super(widget, self).__init__(frame, **kwargs)
            self.grid(row=0, column=1)

    def add_cmd(self, cmd):
        # if cmd is a string store it, and replace it 'cb' callback function
        if isinstance(cmd, str):
            self.cmd = cmd
            cmd = self.cb
        self.bind('<Return>', lambda event: cmd(self, event))

    def cb(self, item=None, event=None):
        """Execute the cmd string in the widget context."""
        exec(self.cmd)
```

This allows to make the `Entry` class much shorter.

```
class Entry(ttk.Entry, EntryMixin):
    """Create an Entry object with label and callback."""

    def __init__(self, label='', cmd='', val='',  **kwargs):
        self.var = tk.StringVar()
        self.var.set(val)
```

(continues on next page)

(continued from previous page)

```
        self.add_widget(label, Entry, kwargs)
        self['textvariable'] = self.var
        self.add_cmd(cmd)
```

The other class definitions are as follows:

```
class Combobox(ttk.Combobox, EntryMixin):
    """Create a Combobox with label and callback."""

    def __init__(self, label='', values='', cmd='', val=0, **kwargs):
        if isinstance(values, str):
            values = values.split(';')

        self.var = tk.StringVar()
        self.var.set(values[val])

        self.add_widget(label, Combobox, kwargs)
        self['textvariable'] = self.var
        self['values'] = values

        self.add_cmd(cmd)
        self.bind('<<ComboboxSelected>>', self.cb)
```

```
class Spinbox(ttk.Spinbox, EntryMixin):
    """Create a Spinbox with label and callback."""

    def __init__(self, label='', cmd='', values='', val=0, **kwargs):
        if isinstance(values, str):
            values = values.split(';')
            if len(values) > 1:
                val = values[val]

        self.var = tk.StringVar(value=val)

        self.add_widget(label, Spinbox, kwargs)
        self['textvariable'] = self.var

        if len(values) > 1:
            self['values'] = values
        self.add_cmd(cmd)
```

```
class Scale(ttk.Scale, EntryMixin):
    """Create a Spinbox with label and callback."""

    def __init__(self, label='', cmd='', val=0, **kwargs):
        self.var = tk.IntVar(value=val)

        if not 'length' in kwargs:
            kwargs.update({'length': 200})

        self.add_widget(label, Scale, kwargs)
        self['variable'] = self.var

        self.add_cmd(cmd)
        if isinstance(cmd, str):
```

(continues on next page)

(continued from previous page)

```
            self.cmd = cmd
            cmd = self.cb
        self['command'] = lambda event: cmd(self, event)
```

CHAPTER 6

Grid geometry mangager

The grid manager positions widgets along columns and rows.

CHAPTER 7

Basic widgets

In this section we are going to look in more detail at these basic widgets:

- Button
- Label
- Entry
- Frame

7.1 Button

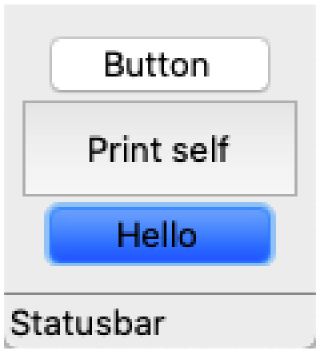
Buttons have text, a command and keyword arguements:

```
Button()
Button('Print 123', 'print(123)', padding=10)
Button('Hello', 'print("Hello " * 3)', default='active')
```

The first button has the default text *Button* and does nothing.

The second button has a padding of 10 pixels and prints `.!frame.!button2`

The third button is active by default and prints:: `Hello Hello Hello`



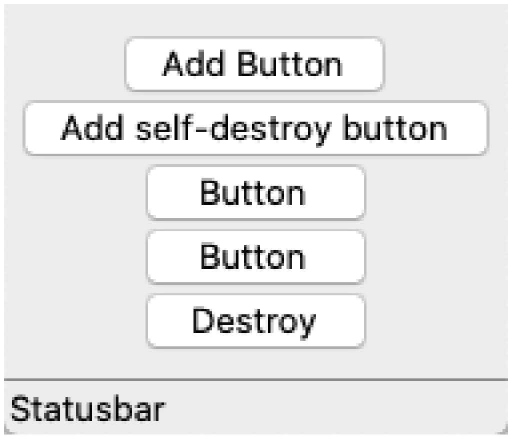
```
"""Create buttons with actions."""
from tklib import *

class Demo(App):
    """Create different buttons."""
    def __init__(self):
        super().__init__()
        App.root.title('Button Demo')

        Button()
        Button('Print self', 'print(self)', padding=10)
        Button('Hello', 'print("Hello " * 3)', default='active')

Demo().run()
```

button1.py

7.2 Buttons which create buttons

Buttons can also be created or deleted dynamically:

- The first button creates another default button.
- The second button creates a self-destroy button.



```
"""Buttons which create other buttons."""
from tklib import *

```

(continues on next page)

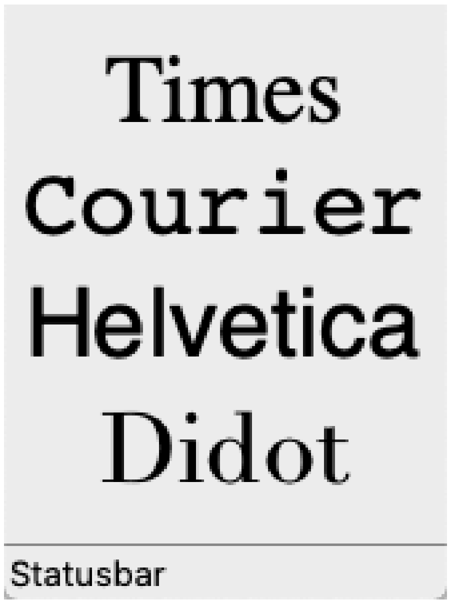
(continued from previous page)

```
class Demo(App):
    """Create different buttons."""
    def __init__(self):
        super().__init__()
        App.root.title('Button Demo')

        Button('Add Button', 'Button()')
        Button('Add self-destroy button', 'Button("Destroy", "self.destroy()")')

Demo().run()
```

button2.py

7.3 Fonts

Labels can use different fonts and size.



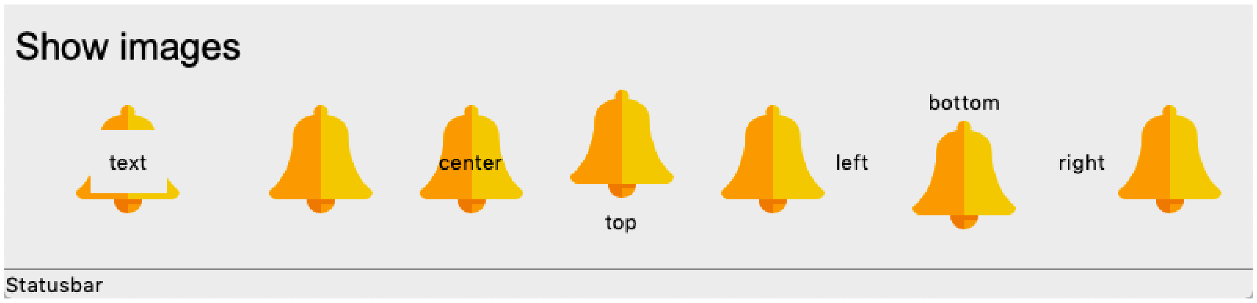
```
"""Show different fonts."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()

        fonts = ['Times', 'Courier', 'Helvetica', 'Didot']
        for x in fonts:
            Label(x, font=x+' 36')

Demo().run()
```

font1.py

7.4 Display a label with an image with

Labels can be displayed with an additional image. The image position can be:

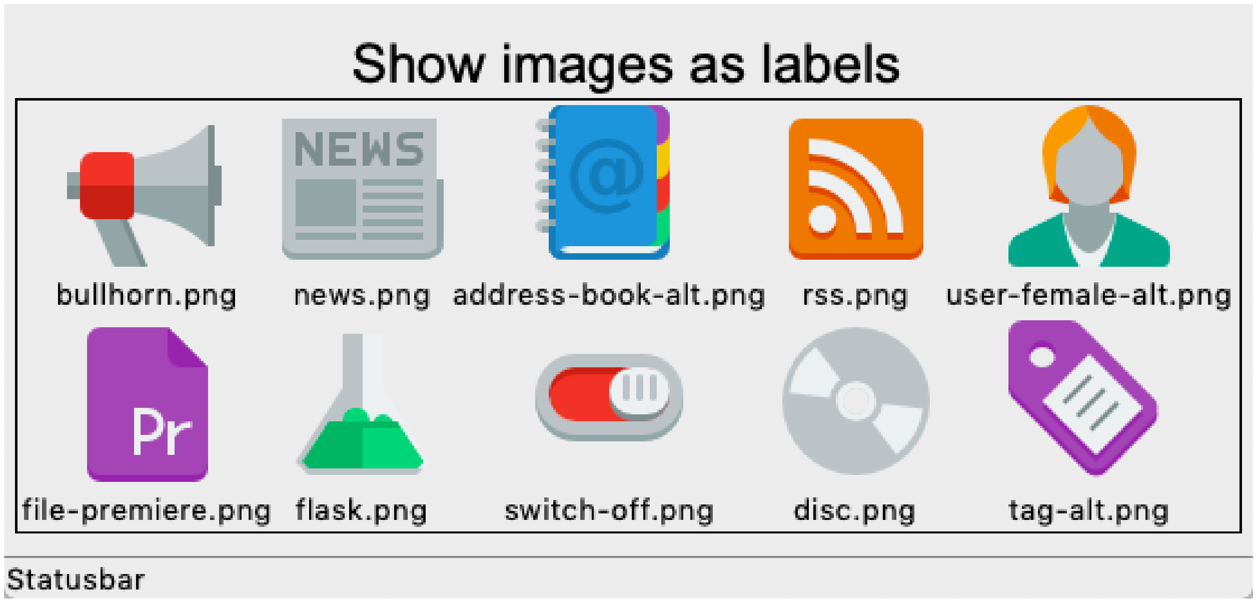
- center
- left/right
- top/bottom



```
"""Display a label with an image."""

import os
from tklib import *
from PIL import Image, ImageTk

class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Show images', font='Arial 24')

        lb = Label()
        App.img0 = Image.open('icons/bell.png')
        App.img = ImageTk.PhotoImage(App.img0)
        lb['image'] = App.img

        L = 'text;image;center;top;left;bottom;right'.split(';')

        i = 0
        for l in L:
            lb = Label(l, compound=l, image=App.img, padding=10)
            lb.grid(column=i, row=1)
            i += 1

Demo().run()
```

label1.py

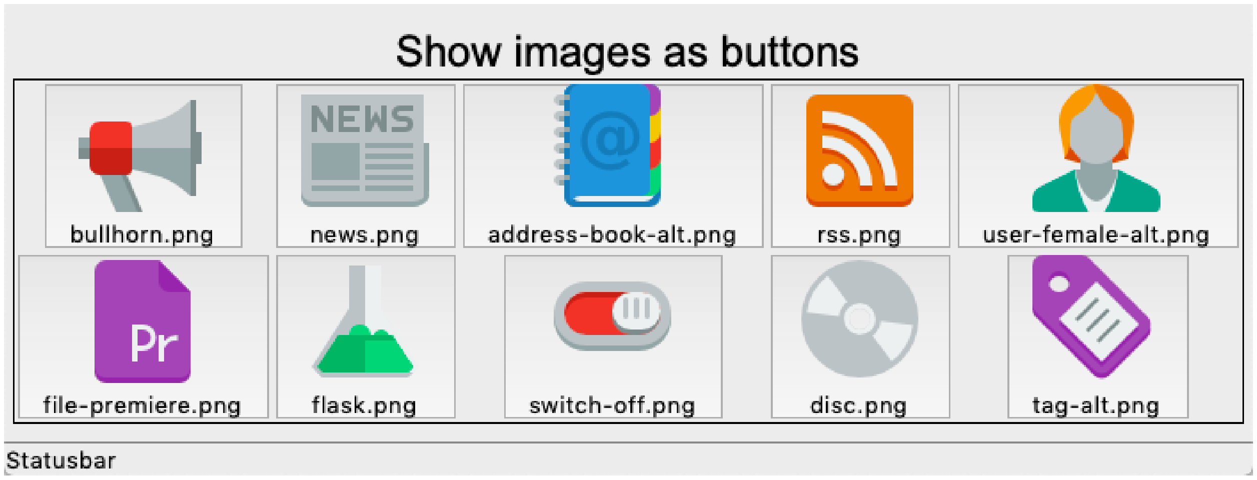
7.5 Display a images as labels



```python
"""Display images as labels."""
from tklib import *
from PIL import Image, ImageTk

class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Show images as labels', font='Arial 24')

        dir = os.listdir('icons')
        n, m = 2, 5
        self.images = []
        Frame()
        for i in range(n):
            for j in range(m):
                k = m*i + j
                label = dir[k]
                path = 'icons/' + label
                img0 = Image.open(path)
                img = ImageTk.PhotoImage(img0)
                self.images.append(img)
                lb = Label(image=img, text=label, compound='top')
                lb.grid(row=i, column=j)

Demo().run()
```

label2.py

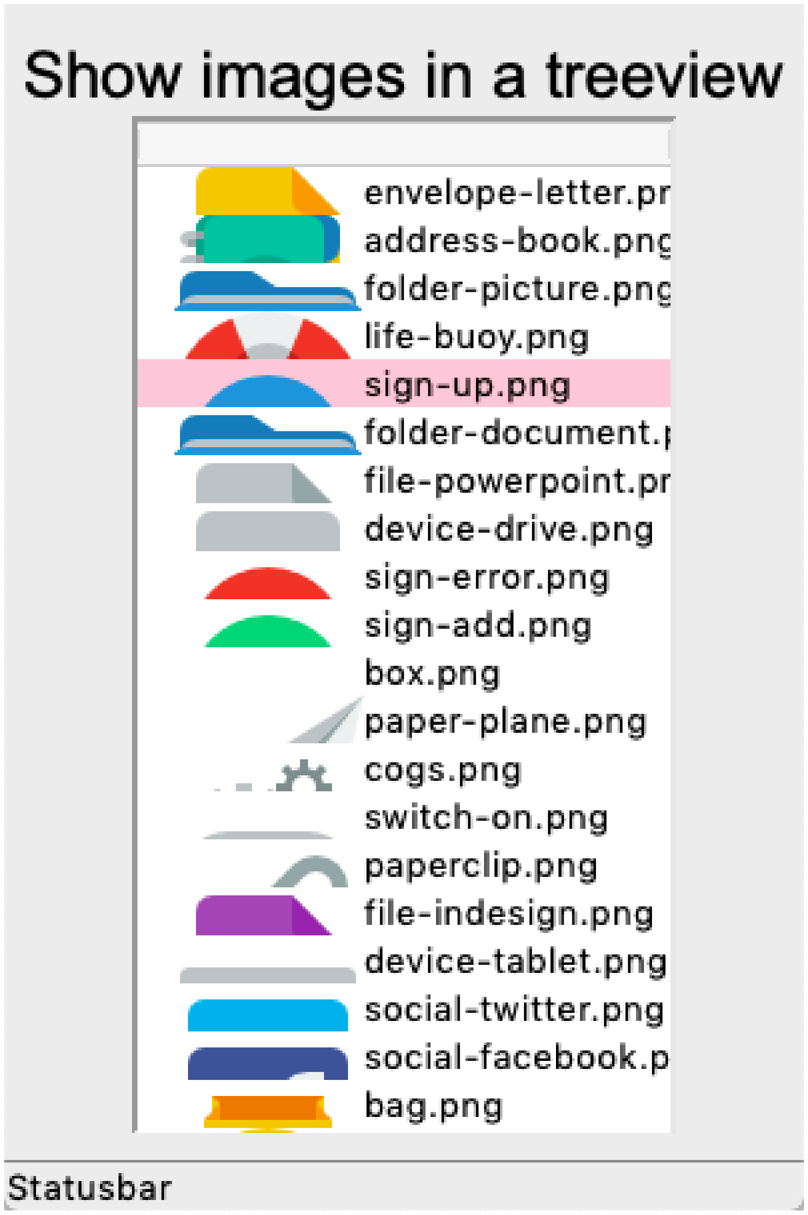
7.6 Display a images as buttons



```python
"""Display images as buttons."""

from tklib import *
from PIL import Image, ImageTk

class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Show images as buttons', font='Arial 24')

        dir = os.listdir('icons')
        n, m = 2, 5
        self.images = []
        Frame()
        for i in range(n):
            for j in range(m):
                k = m*i + j
                label = dir[k]
                path = 'icons/' + label
                img0 = Image.open(path)
                img = ImageTk.PhotoImage(img0)
                self.images.append(img)
                lb = Button(image=img, text=label, compound='top')
                lb.grid(row=i, column=j)

if __name__ == '__main__':
    Demo().run()
```

label3.py

7.7 Display a images in a listbox



```
"""Display images as listbox."""
from tklib import *
from PIL import Image, ImageTk

class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Show images in a treeview', font='Arial 24')

        dir = os.listdir('icons')
        self.tree = Treeview(height=20)
        self.images = []
```

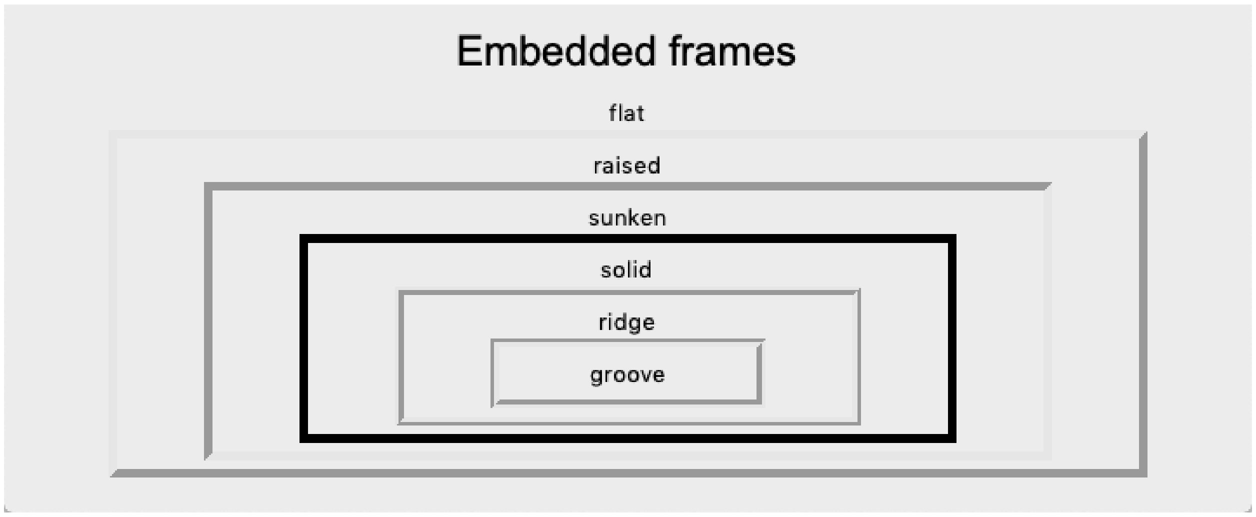
(continues on next page)

(continued from previous page)

```
        for file in dir:
            path = 'icons/' + file
            img0 = Image.open(path)
            img = ImageTk.PhotoImage(img0)
            self.images.append(img)
            self.tree.insert('', 'end', text=file, image=img)

Demo().run()
```

label4.py

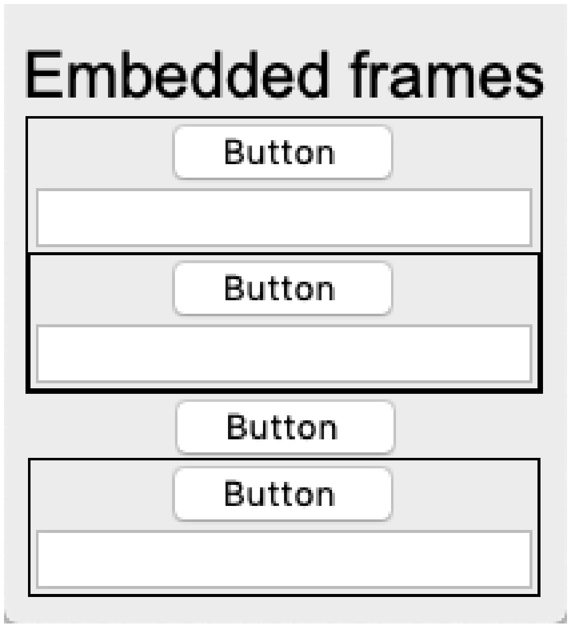
7.8 Embedded frames



```
"""Embedded frames."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Embedded frames', font='Arial 24')

        types = ['flat', 'raised', 'sunken', 'solid', 'ridge', 'groove']
        for t in types:
            Frame(relief=t, padding=(50, 5), borderwidth=5)
            Label(t)

Demo().run()
```

frame1.py

7.9 Embedded frames



```
"""Speparate frames."""
from tklib import *

class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        Label('Embedded frames', font='Arial 24')

        Frame()
        Button()
        Entry()

        Frame()
        Button()
        Entry()

        App.stack.pop()
        App.stack.pop()
        Button()
        Frame()
        Button()
        Entry()



if __name__ == '__main__':
    Demo().run()
```

frame2.py

CHAPTER 8

Events

Tk events can come from various sources including:

- mouse (move, button)
- keyboard
- widget (enter/leave)

Each widget can bind a **handler** method to an event:

```
widget.bind(event, handler)
```

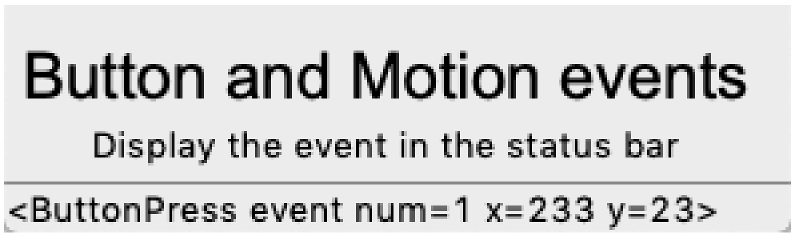
If an event matching the event pattern happens, the corresponing handler is called.

8.1 Write events to the status bar

Here is an exemple which prints mouse **Button** and **Motion** events to the status bar:

```
class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Button and Motion events", font="Arial 24")
        Label('Display the event in the status bar')

        App.root.bind('<Button>', self.cb)
        App.root.bind('<Motion>', self.cb)

    def cb(self, event):
        """Callback function."""
        App.status['text'] = event
```

This is a screen capture of the above program.


Button and Motion events

Display the event in the status bar

<ButtonPress event num=1 x=233 y=23>

```
from tklib import *

def cb(event):
    """Callback function."""
    print(event)

app = App('Events and bindings')

Label("Button and Motion events", font="Arial 24")
Label('Display the event in the status bar')

app.root.bind('<Button>', cb)
app.root.bind('<Motion>', cb)

app.run()
```

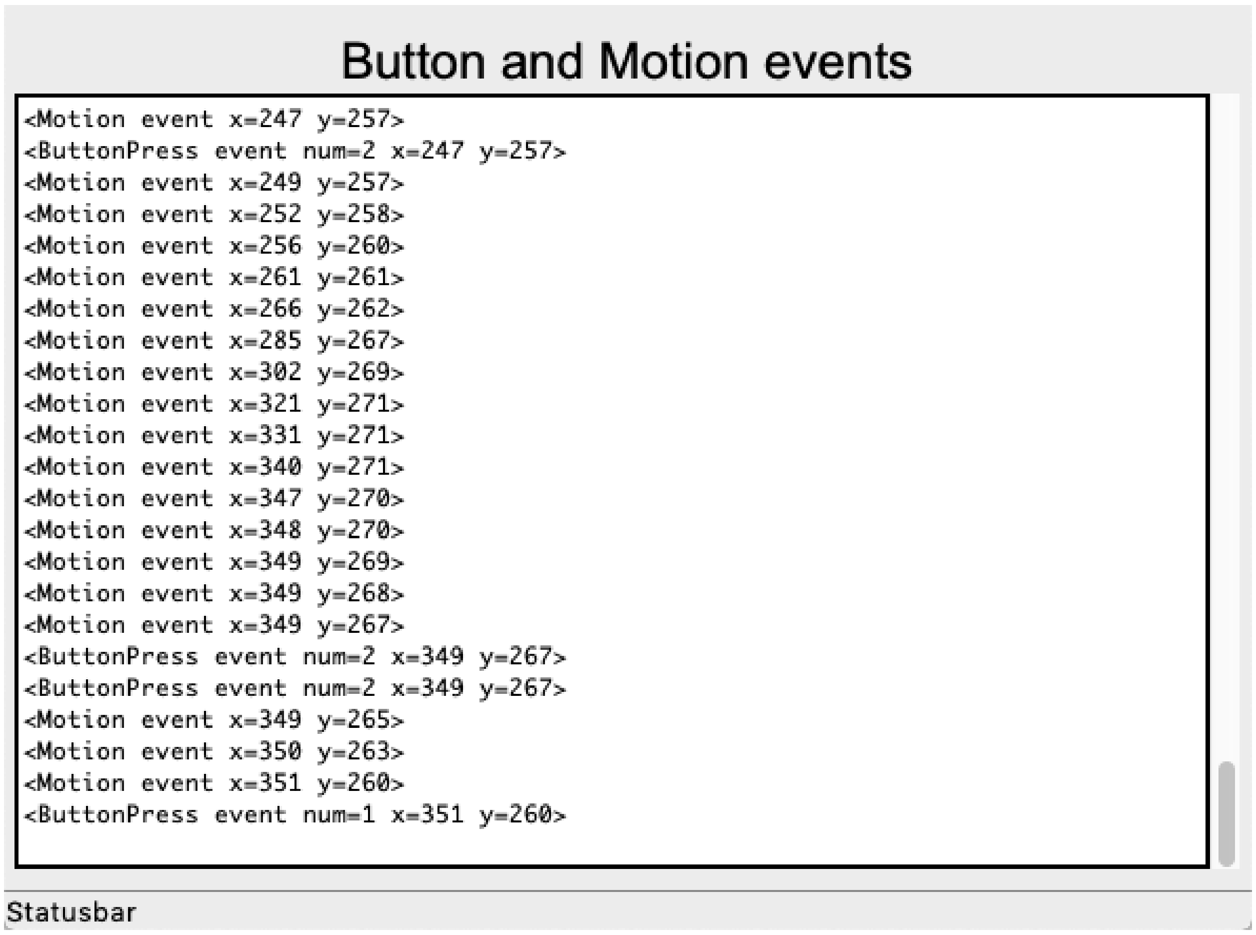
event1.py

8.2 Write events to a text widget

The following program sends **Button** and **Mouse** events to a Text widget, just by changing the callback function:

```
def cb(self, event):
    """Callback function."""
    App.txt.insert('end', str(event) + '\n')
```

This is a screen capture of the above program.



```
<Motion event x=247 y=257>
<ButtonPress event num=2 x=247 y=257>
<Motion event x=249 y=257>
<Motion event x=252 y=258>
<Motion event x=256 y=260>
<Motion event x=261 y=261>
<Motion event x=266 y=262>
<Motion event x=285 y=267>
<Motion event x=302 y=269>
<Motion event x=321 y=271>
<Motion event x=331 y=271>
<Motion event x=340 y=271>
<Motion event x=347 y=270>
<Motion event x=348 y=270>
<Motion event x=349 y=269>
<Motion event x=349 y=268>
<Motion event x=349 y=267>
<ButtonPress event num=2 x=349 y=267>
<ButtonPress event num=2 x=349 y=267>
<Motion event x=349 y=265>
<Motion event x=350 y=263>
<Motion event x=351 y=260>
<ButtonPress event num=1 x=351 y=260>
```

```python
"""Write Button and Mouse events to a Text widget."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Button and Motion events", font="Arial 24")
        App.txt = Text(scroll='y')

        App.root.bind('<Button>', self.cb)
        App.root.bind('<Motion>', self.cb)

    def cb(self, event):
        """Callback function."""
        App.txt.insert('end', str(event) + '\n')

if __name__ == '__main__':
    Demo().run()
```

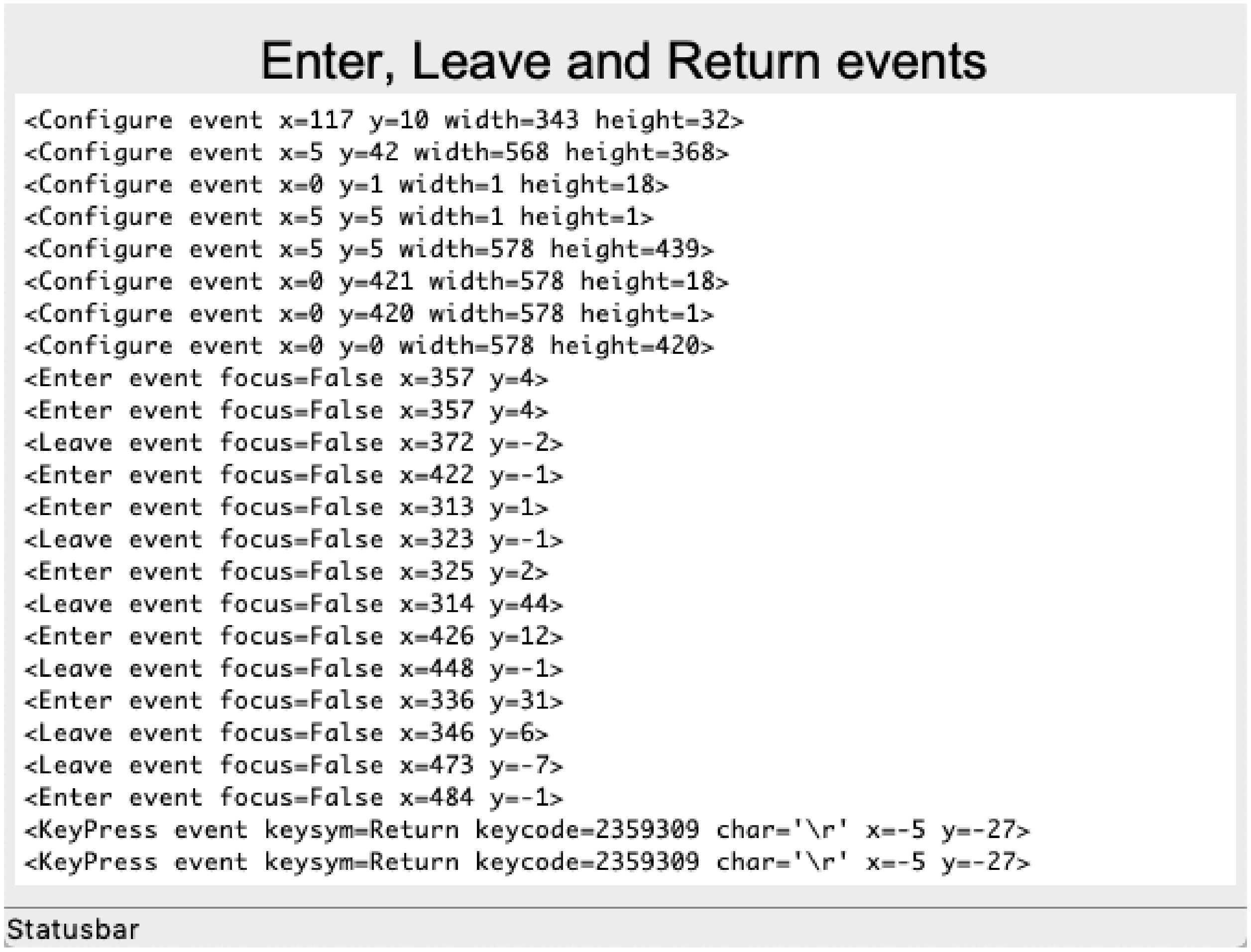
event2.py

8.3 Enter, leave and return events

The following program detects these events:

```
App.root.bind('<Enter>', self.cb)
App.root.bind('<Leave>', self.cb)
App.root.bind('<Return>', self.cb)
App.root.bind('<Configure>', self.cb)
```

This is a screen capture of the above program.


Enter, Leave and Return events

```
<Configure event x=117 y=10 width=343 height=32>
<Configure event x=5 y=42 width=568 height=368>
<Configure event x=0 y=1 width=1 height=18>
<Configure event x=5 y=5 width=1 height=1>
<Configure event x=5 y=5 width=578 height=439>
<Configure event x=0 y=421 width=578 height=18>
<Configure event x=0 y=420 width=578 height=1>
<Configure event x=0 y=0 width=578 height=420>
<Enter event focus=False x=357 y=4>
<Enter event focus=False x=357 y=4>
<Leave event focus=False x=372 y=-2>
<Enter event focus=False x=422 y=-1>
<Enter event focus=False x=313 y=1>
<Leave event focus=False x=323 y=-1>
<Enter event focus=False x=325 y=2>
<Leave event focus=False x=314 y=44>
<Enter event focus=False x=426 y=12>
<Leave event focus=False x=448 y=-1>
<Enter event focus=False x=336 y=31>
<Leave event focus=False x=346 y=6>
<Leave event focus=False x=473 y=-7>
<Enter event focus=False x=484 y=-1>
<KeyPress event keysym=Return keycode=2359309 char='\r' x=-5 y=-27>
<KeyPress event keysym=Return keycode=2359309 char='\r' x=-5 y=-27>
```

Statusbar

```
from tklib import *

class Demo(App):
    """Write Enter, Leave and Return events to a Text widget."""
    def __init__(self):
        super().__init__()
        Label("Enter, Leave and Return events", font="Arial 24")

        App.txt = Text()
        App.txt.grid(sticky='nswe')

        App.root.bind('<Enter>', self.cb)
        App.root.bind('<Leave>', self.cb)
        App.root.bind('<Return>', self.cb)
        App.root.bind('<Configure>', self.cb)

    def cb(self, event):
```

(continues on next page)

(continued from previous page)

```
        """Callback function."""
        App.txt.insert('end', str(event) + '\n')

if __name__ == '__main__':
    Demo().run()
```

event3.py

8.4 Keyboard events

Specific keyboard events can be bound to a specific widget and trigger a callback function. In the example below we bind different keys to the root widget in order to call a callback function. The callback function inserts the event despcriptor or a short text into a **Text** widget.

- **<Key>** - any key
- **a** - a lower-case a (or any other letter)
- **A** - an upper-case A
- **<Return>** - the Return key
- **<Escape>** - the Escape key
- **<Tab>** - the Tab key

Modifier keys can also bind to a callback function. They work equally for the left and the right keys.

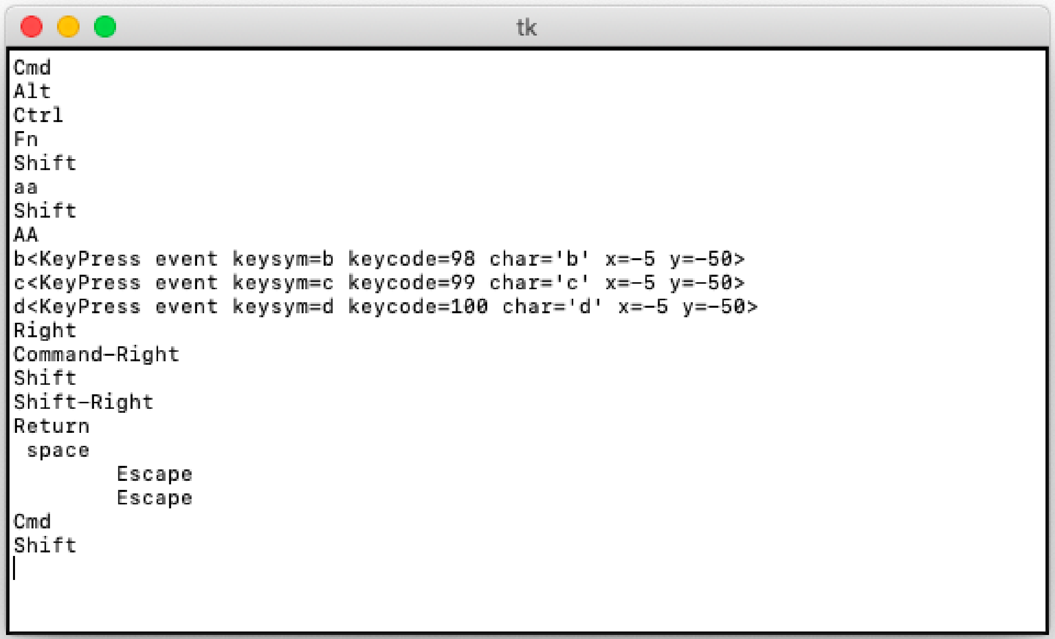
- **<Shift_L>** for the Shift keys
- **<Control_L>** for the Control keys
- **<Alt_L>** for the Alt keys
- **<Meta_L>** for the Command key (on the Mac)
- **<Super_L>** for the Fn key (on the Mac)

Finally we configure the **BackSpace** key to clear the screen:

```
root.bind('<BackSpace>', lambda e: text.delete('1.0', 'end'))
```

The *x=-5, y=-50* coordinates are the position of the widget.

This is a screen capture showing the trace of various key presses.



```
Cmd
Alt
Ctrl
Fn
Shift
aa
Shift
AA
b<KeyPress event keysym=b keycode=98 char='b' x=-5 y=-50>
c<KeyPress event keysym=c keycode=99 char='c' x=-5 y=-50>
d<KeyPress event keysym=d keycode=100 char='d' x=-5 y=-50>
Right
Command-Right
Shift
Shift-Right
Return
 space
        Escape
        Escape
Cmd
Shift
```

```python
# keyboard bindings
import tkinter as tk

def cb(event):
    text.insert('end', str(event) + '\n')

root = tk.Tk()
text = tk.Text(root)
text.grid()

root.bind('<Key>', cb)
root.bind('a', lambda e: cb('a'))
root.bind('A', lambda e: cb('A'))
root.bind('<Return>', lambda e: cb('Return'))
root.bind('<Escape>', lambda e: cb('Escape'))
root.bind('<Tab>', lambda e: cb('Tab'))
root.bind('<space>', lambda e: cb('space'))
root.bind('<F1>', lambda e: cb('F1'))

# modifier keys (left and right)
root.bind('<Shift_L>', lambda e: cb('Shift'))
root.bind('<Control_L>', lambda e: cb('Ctrl'))
root.bind('<Alt_L>', lambda e: cb('Alt'))
root.bind('<Meta_L>', lambda e: cb('Cmd'))
root.bind('<Super_L>', lambda e: cb('Fn'))

# modifier keys for arrows
root.bind('<Right>', lambda e: cb('Right'))
```

(continues on next page)

(continued from previous page)

```
root.bind('<Shift-Right>', lambda e: cb('Shift-Right'))
root.bind('<Command-Right>', lambda e: cb('Command-Right'))

# clear screen
root.bind('<BackSpace>', lambda e: text.delete('1.0', 'end'))
root.mainloop()
```

event4.py

CHAPTER 9

Listbox

A **listbox** displays a list of single-line text items, and allows users to browse through the list, and selecting one or more items.

9.1 Create a listbox the old way

The old way of creating a listbox, was to create an empty listbox widget and then to insert item by item into the listbox.

The listbox methods `insert` and `delete` allow to add or remove items a specific position. The label `tk.END` is used to add items to the end of the listbox. Here three items are added at the top and one item is removed:

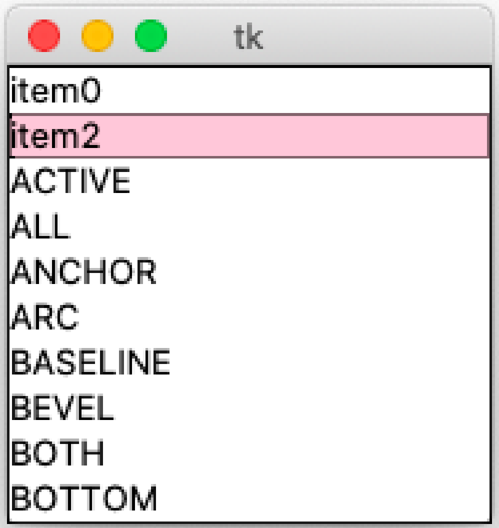
```
lb.insert(0, 'item0', 'item1', 'item2')
lb.delete(1)
```

To insert multiple items of a list one can iterate through the list:

```
for item in items:
    lb.insert(tk.END, item)
```

or hand multiple items directly to the `insert` method:

```
lb.insert(tk.END, *items)
```



```
# Create a listbox the old way, with the insert/delete method
import tkinter as tk

root = tk.Tk()

lb = tk.Listbox(root)
lb.grid()

lb.insert(0, 'item0', 'item1', 'item2')
lb.delete(1)

items = dir(tk)
for item in items:
    lb.insert(tk.END, item)
lb.insert(tk.END, *items)

root.mainloop()
```

lb1.py

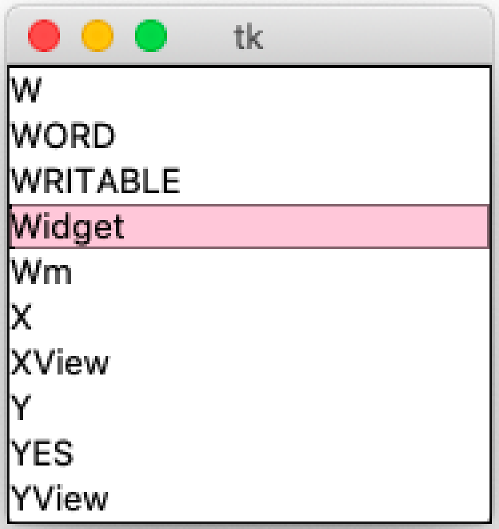
9.2 Create a listbox the new way

The new and much simpler way of handling the listbox content is to use the **listvariable** argument which must be set to a **StringVar** object. The program below displays the attributes of the **Tk** module as a list:

```
items = dir(tk)
var = tk.StringVar()
var.set(items)
```

At creation the listbox sets its **listvariable** to this list:

```
tk.Listbox(root, listvariable=var).grid()
```



```
# Display a simple listbox
import tkinter as tk

root = tk.Tk()

items = dir(tk)
var = tk.StringVar()
var.set(items)
tk.Listbox(root, listvariable=var).grid()

root.mainloop()
```

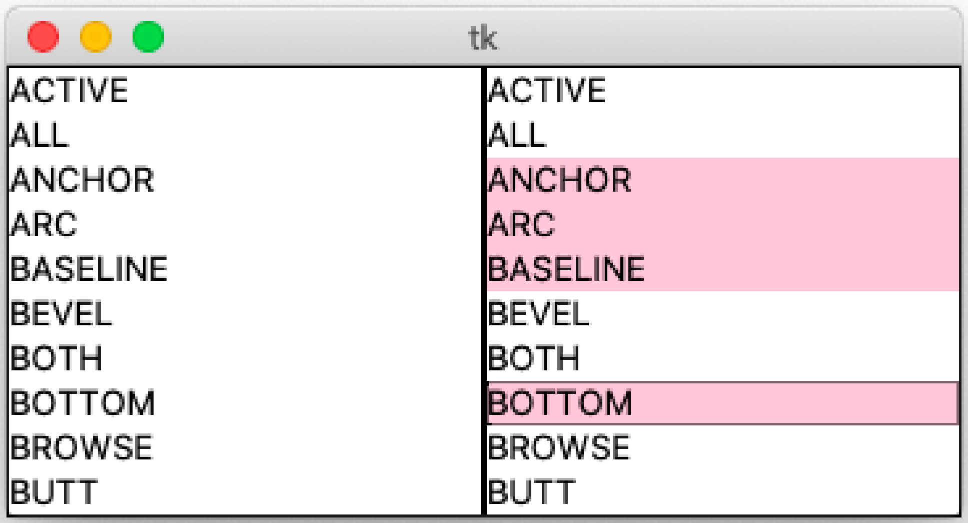
`lb2.py`

9.3 Single and extended selection mode

The `selectmode` argument let's the listbox be configured for

- single item selection (browse)

- multiple item selection (extended)



```
# Show browse/extended selectmode
import tkinter as tk

root = tk.Tk()

var = tk.StringVar(value=dir(tk))

tk.Listbox(root, listvariable=var, selectmode='browse').pack(side='left')
tk.Listbox(root, listvariable=var, selectmode='extended').pack(side='left')

root.mainloop()
root.mainloop()
```

lb3.py

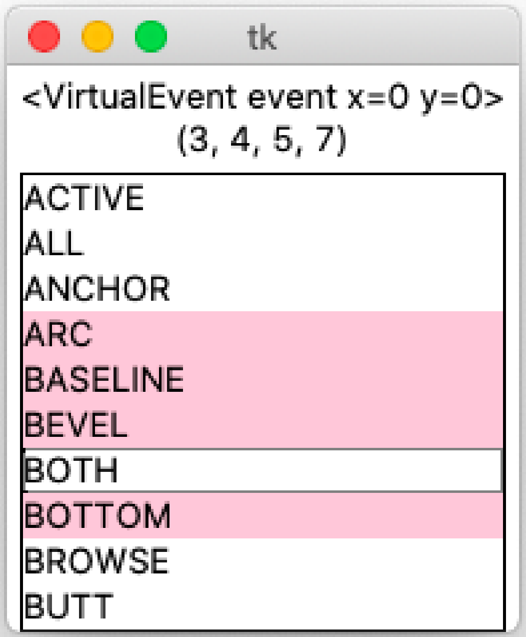
9.4 ListboxSelect callback function

When the user selects an item, either with a mouse click or with the arrow keys, a virtual **<ListboxSelect>** event is generated. You can bind to it to a callback function:

```
lb.bind('<<ListboxSelect>>', cb)
```

The callback function prints the event descriptor and the current selection as an index list to a label:

```
def cb(event):
    label['text'] = str(event) + '\n' + str(lb.curselection())
```



<VirtualEvent event x=0 y=0>
(3, 4, 5, 7)

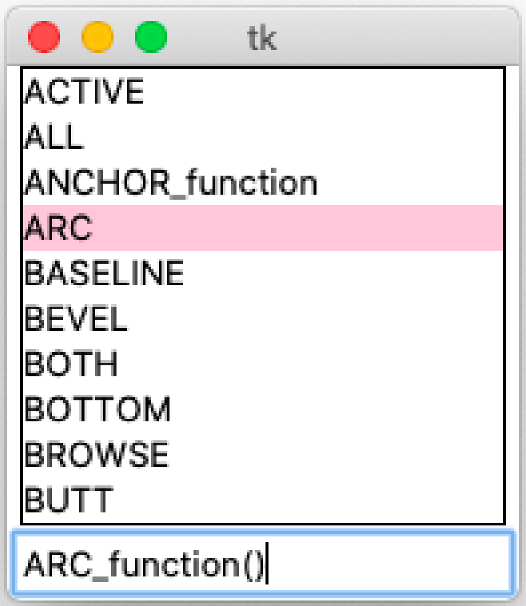
ACTIVE
ALL
ANCHOR
ARC
BASELINE
BEVEL
BOTH
BOTTOM
BROWSE
BUTT

```
# <ListboxSelect> callback function and current selection
import tkinter as tk

def cb(event):
    label['text'] = str(event) + '\n' + str(lb.curselection())

root = tk.Tk()
var = tk.StringVar(value=dir(tk))

label = tk.Label(root)
label.grid()

lb = tk.Listbox(root, listvariable=var, selectmode='extended')
lb.grid()
lb.bind('<<ListboxSelect>>', cb)

root.mainloop()
```

lb4.py

9.5 Edit a listbox item

In the following example we display the selected listbox item in an entry field. The entry field can be edited and hitting the return key writes the new item value to the listbox. The screen capture below shows how **ANCHOR** had been changed to **ANCHOR_function**.



ACTIVE
ALL
ANCHOR_function
ARC
BASELINE
BEVEL
BOTH
BOTTOM
BROWSE
BUTT

ARC_function()

```
# Edit listbox item with an entry widget
import tkinter as tk

def select(event):
    i = lb.curselection()[0]
    item.set(items[i])

def update(event):
    i = lb.curselection()[0]
    items[i] = item.get()
    var.set(items)

root = tk.Tk()
items = dir(tk)
var = tk.StringVar(value=items)

lb = tk.Listbox(root, listvariable=var)
lb.grid()
lb.bind('<<ListboxSelect>>', select)

item = tk.StringVar()
entry = tk.Entry(root, textvariable=item, width=20)
entry.grid()
entry.bind('<Return>', update)
```

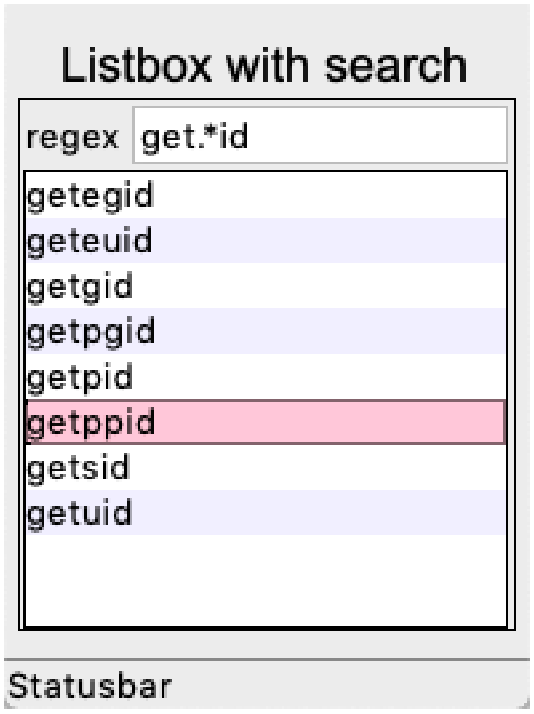
(continues on next page)

(continued from previous page)

```
root.mainloop()
```

lb5.py

9.6 Listbox with search


Listbox with search

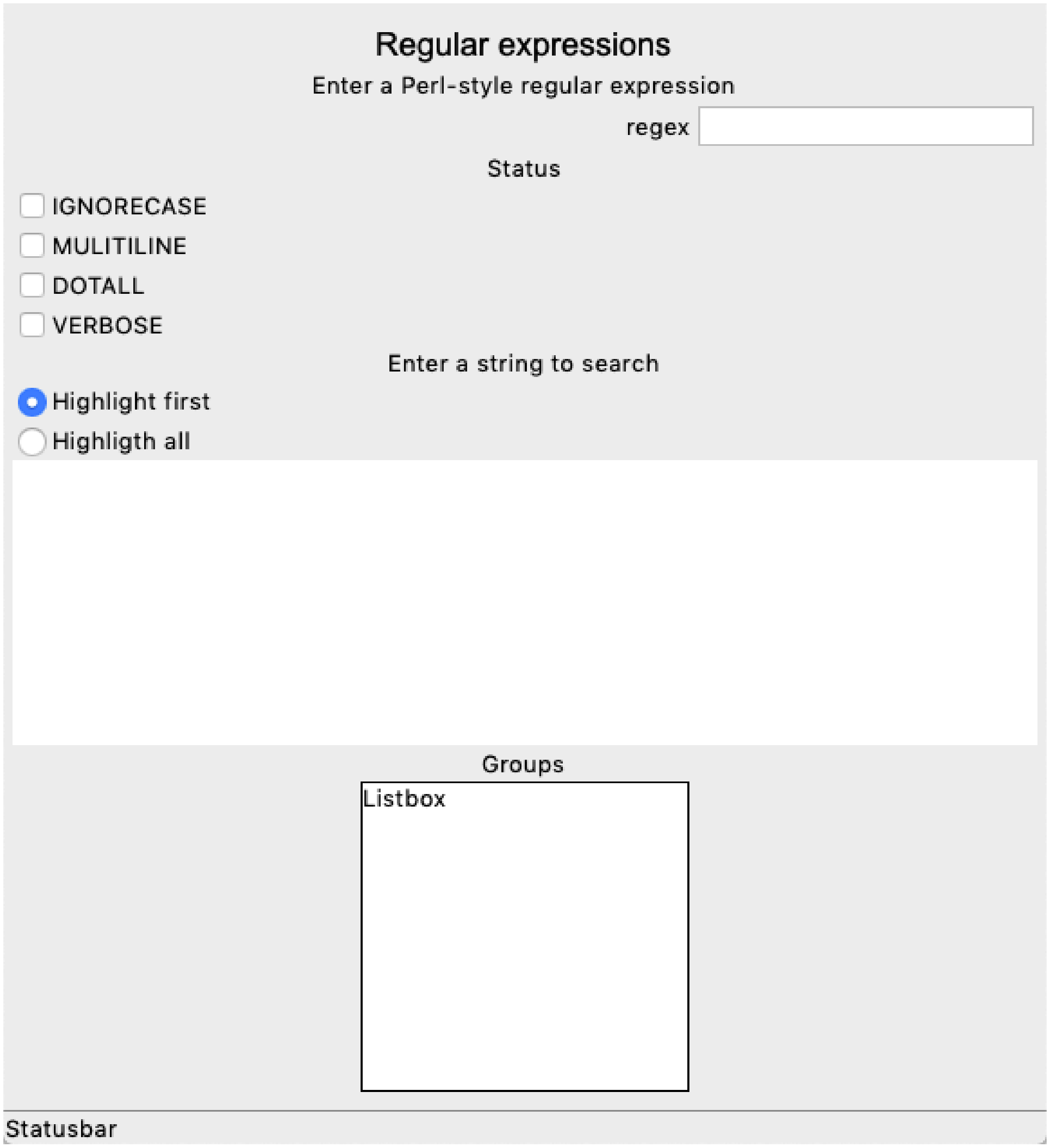
regex get.*id

getegid
geteuid
getgid
getpgid
getpid
getppid
getsid
getuid

Statusbar

listbox4.py

9.7 Regular expression



```
"""Regular expression demo."""
from tkinter import *
from tklib import *
import re

class Browser(Listbox):
    def cb(self, event):
```

(continues on next page)

(continued from previous page)

```
        pass

class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Regular expressions', font='Arial 18')
        Label('Enter a Perl-style regular expression')
        App.re = Entry('regex')
        App.re.bind('<Key>', self.recompile)
        App.status = Label('Status')

        Checkbutton('IGNORECASE;MULITILINE;DOTALL;VERBOSE')

        Label('Enter a string to search')
        Radiobutton('Highlight first;Highligth all')

        App.str = Text(height=10)
        App.str.bind('<Key>', self.reevaluate)
        App.str.tag_configure("hit", background="yellow")

        Label('Groups')
        App.groups = Listbox()

        btags = App.re.bindtags()
        App.re.bindtags(btags[1:] + btags[:1])

        btags = App.str.bindtags()
        App.str.bindtags(btags[1:] + btags[:1])
        self.compiled = None

    def recompile(self, event=None):
        print('recompile')
        try:
            self.compiled = re.compile(App.re.get())
        except re.error as msg:
            self.compiled = None
            App.status.config(
                    text="re.error: %s" % str(msg),
                    background="red")
        self.reevaluate()

    def reevaluate(self, event=None):
        print('reevaluate')
        try:
            self.str.tag_remove("hit", "1.0", END)
        except TclError:
            pass
        try:
            self.str.tag_remove("hit0", "1.0", END)
        except TclError:
            pass
        self.groups.delete(0, END)
        if not self.compiled:
            return
        self.str.tag_configure("hit", background="yellow")
        self.str.tag_configure("hit0", background="orange")
        text = self.str.get("1.0", END)
```

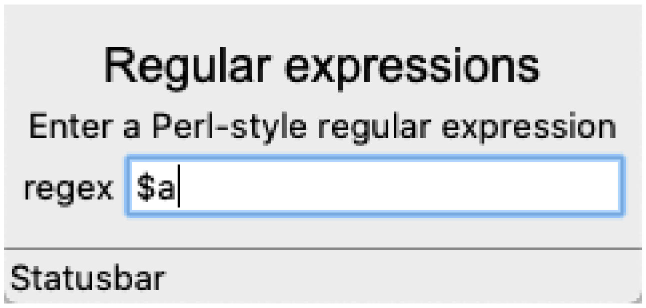
(continues on next page)

(continued from previous page)

```
        last = 0

        nmatches = 0
        while last <= len(text):
            m = self.compiled.search(text, last)
            if m is None:
                break
            first, last = m.span()
            if last == first:
                last = first+1
                tag = "hit0"
            else:
                tag = "hit"
            pfirst = "1.0 + %d chars" % first
            plast = "1.0 + %d chars" % last
            self.str.tag_add(tag, pfirst, plast)
            if nmatches == 0:
                self.str.yview_pickplace(pfirst)
                groups = list(m.groups())
                groups.insert(0, m.group())
                for i in range(len(groups)):
                    g = "%2d: %r" % (i, groups[i])
                    App.groups.insert(END, g)
            nmatches = nmatches + 1
            if self.showvar.get() == "first":
                break

        if nmatches == 0:
            self.status.config(text="(no match)",
                              background="yellow")
        else:
            self.status.config(text="")

Demo().run()
```

re1.py

9.8 Regular expression



Regular expressions

Enter a Perl-style regular expression

regex $a

Statusbar

```
from tklib import *
import re
app = App('Regular expression')
```

(continues on next page)

(continued from previous page)

```
Label('Regular expressions', font='Arial 18')
Label('Enter a Perl-style regular expression')
App.re = Entry('regex')

app.run()
```

re2.py

CHAPTER 10

Menu

This section describes how to create menubars and popup menus. Menus are widgets as well. We add the following code to the App() class. Creating the menu bar:

```
menubar = tk.Menu(root)
```

Adding the menu bar to the root window:

```
root['menu'] = menubar
```

Placing the menu bar on a menus stack for accessing to the menu items later on.

- menus[0] is always the menu bar
- menus[1] is the first menu item created
- menus[-1] is the last menu item created

This initializes the menu stack:

```
menus = [menubar]
```

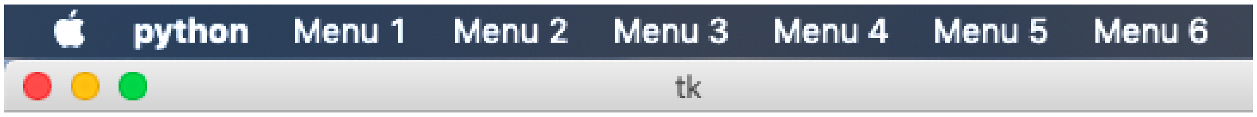
10.1 Adding a menu bar

The Menu() class is defined like this:

```
class Menu(tk.Menu):
    """Add a Menu() node to which a menu Item() can be attached."""
    def __init__(self, label, id=0, **kwargs):
        super(Menu, self).__init__(App.menus[0], **kwargs)
        App.menus[id].add_cascade(menu=self, label=label)
        App.menus.append(self)
```

To create a menubar we just can instantiate a series of Menu() widgets:

```
for i in range(1, 7):
    Menu('Menu ' + str(i))
```



```
"""Add menus to the menubar."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Menu bar without items", font="Arial 24")

        for i in range(1, 7):
            Menu('Menu ' + str(i))

Demo().run()
```

menu1.py

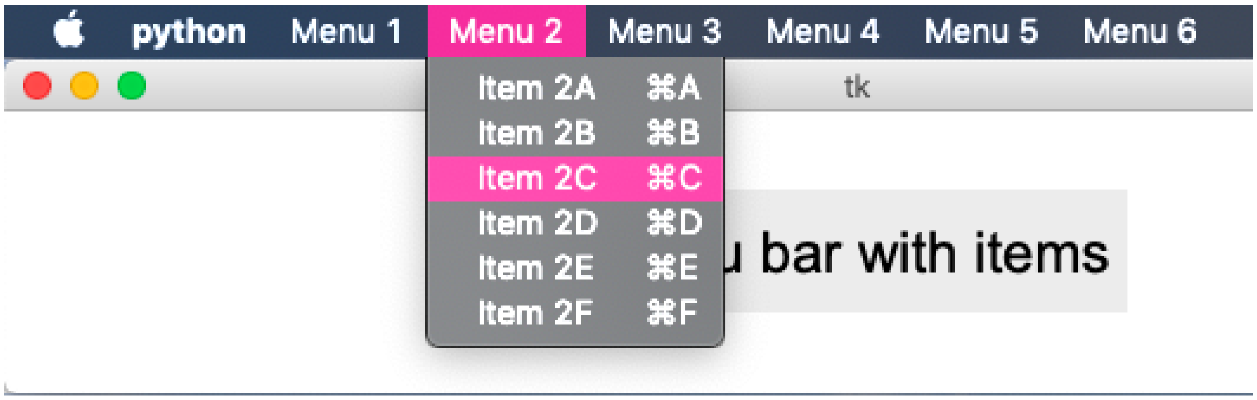
10.2 Adding items to the menu bar

To add items to a menu bar, we first call `Menu()` and then `Item()` for each menu item to add to the menu: The Item object has the following parameters:

- label: the text to appear in the menu item
- cmd: an executable string
- acc (accelerator): a shortcut composed of Control/Command - (hyphen) and a lower-case letter

The following exemple adds 6 items to the 6 menus previously created:

```
for i in range(1, 7):
    Menu('Menu ' + str(i))
    for c in "ABCDEF":
        label = 'Item {}{}'.format(i, c)
        cmd = 'print("{}")'.format(label)
        Item(label, cmd, 'Command-{}'.format(c.lower()))
```



```
"""Add menus and items."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Menu bar with items", font="Arial 24")

        for i in range(1, 7):
            Menu('Menu ' + str(i))
            for c in "ABCDEF":
                label = 'Item {}{}'.format(i, c)
                cmd = 'print("{}")'.format(label)
                Item(label, cmd, 'Command-{}'.format(c.lower()))

Demo().run()
```

menu2.py

10.3 Menus with sub-menus

Each new menu can be attached to the menu bar (default) or can be attached to an existing menu. The parameter id.

The following code adds a new submenu at the end of the previously created menu. Then it adds 3 subitems, and finally an item which is directly attached to the parent Menu():

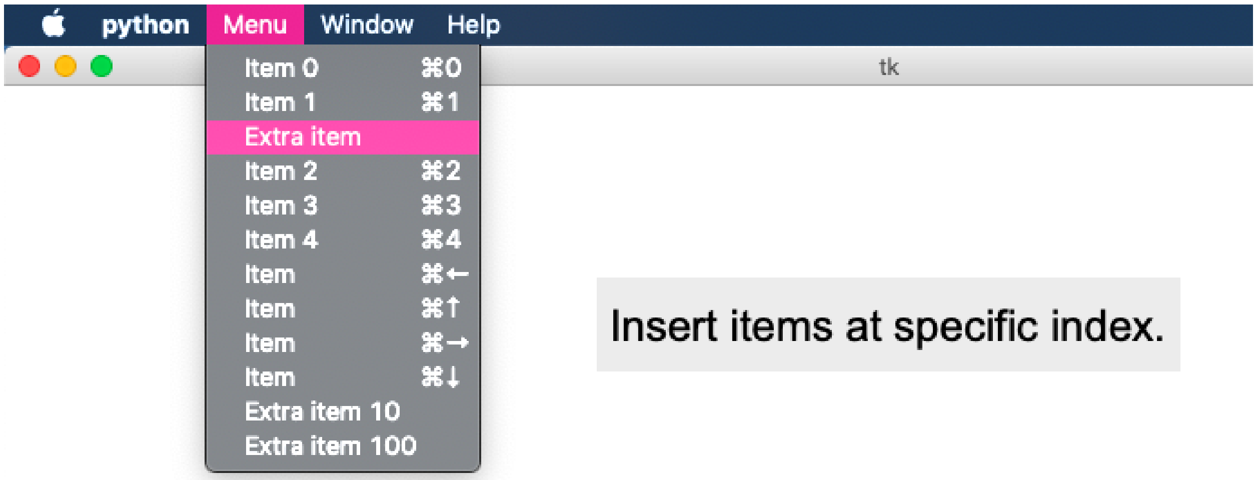
```
for i in range(1, 7):
    Menu('Submenu in {}'.format(i), id=i)
    for c in 'XYZ':
        Item('Item ' + c, 'print(123)', 'Control-' + c.lower())
    Item('Add Item to {}'.format(i), id=i)
```

This is the Menu class definition.

```
class Menu(tk.Menu):
    """Add a Menu() node to which a menu Item() can be attached."""

    def __init__(self, label='Menu', id=0, **kwargs):
        super(Menu, self).__init__(App.menus[0], **kwargs)
        App.menus[id].add_cascade(menu=self, label=label)
        App.menus.append(self)
```

This is the `Item` class definition.

```
class Item(Callback):
    """Add a menu item to a Menu() node. Default is the last menu (id=-1)."""

    def __init__(self, label, cmd='', acc='', id=-1, **kwargs):
        self.cmd = cmd
        if isinstance(cmd, str):
            cmd = self.cb
        if acc != '':
            key = '<{}>'.format(acc)
            App.root.bind(key, self.cb)

        if label == '-':
            App.menus[id].add_separator()
        elif label[0] == '*':
            App.menus[id].add_checkbutton(
                label=label[1:], command=cmd, accelerator=acc, **kwargs)
        elif label[0] == '#':
            App.menus[id].add_radiobutton(
                label=label[1:], command=cmd, accelerator=acc, **kwargs)
        else:
            App.menus[id].add_command(
                label=label, command=cmd, accelerator=acc, **kwargs)
```



Insert items at specific index.

```
"""Insert items at a specific index position."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Insert items at specific index.", font="Arial 24")

        Menu('Menu')
        for i in range(5):
            Item('Item {}'.format(i), cmd=lambda : print(i), acc='Command-'+str(i))

        Item('Item', 'print("Left")', acc='Command-Left')
        Item('Item', 'print("Up")', acc='Command-Up')
        Item('Item', 'print("Right")', acc='Command-Right')
```

(continues on next page)

(continued from previous page)

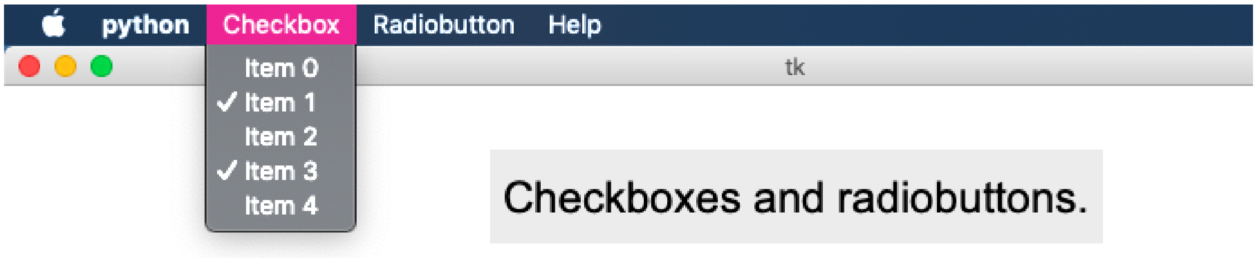
```
        Item('Item', 'print("Down")', acc='Command-Down')

        App.menus[-1].insert(2, 'command', label='Extra item')
        App.menus[-1].insert(100, 'command', label='Extra item 100')
        App.menus[-1].insert(10, 'command', label='Extra item 10')

        Menu('Window', name='window')
        Menu('Help', name='help')

Demo().run()
```

menu6.py 6 Checkbuttones and radiobuttons ————————

Items can be configured as checkboxes or radiobuttons. The first symbol of the label decides if a command, ceckbox or radiobuttion is:

- asterisk (*) : make the item a checkbox
- hashtag (#) : make the item a radiobutton

This is the result:



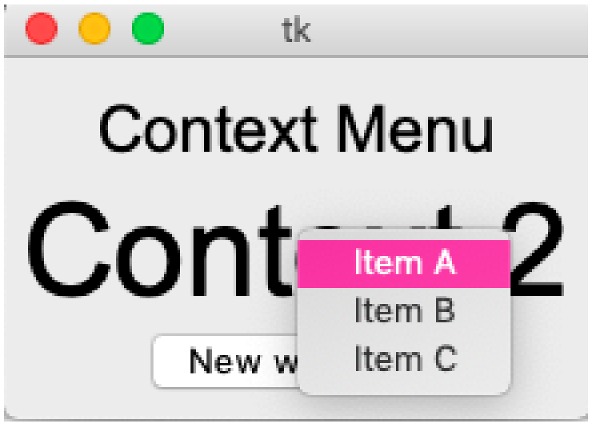
```
"""Menus with items, separators, checkboxes and radiobuttons."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Checkbuttones and radiobuttons.", font="Arial 24")

        Menu('Checkbutton')
        for i in range(5):
            Item('*Item {}'.format(i))

        Menu('Radiobutton')
        radio = tk.StringVar()
        for i in range(5):
            Item('#Item {}'.format(i), variable=radio)

        Menu('Help', name='help')

Demo().run()
```

menu6.py 6 Context menu ————

The same Menu and Item widgets can be used to create context menus. Contrary to the Menu which is attached to the root window, the ContextMenu is attached to a specific widget. When the widget is clicked, a callback function is called.

The `Menu.post()` method is used to open the context menu at the current cursor position:

```
class ContextMenu(tk.Menu):
    def __init__(self, widget):
        """Create a context menu attached to a widget."""
        super(ContextMenu, self).__init__(widget)
        App.menus.append(self)

        if (App.root.tk.call('tk', 'windowingsystem') == 'aqua'):
            widget.bind('<2>', self.popup)
            widget.bind('<Control-1>', self.popup)
        else:
            widget.root.bind('<3>', self.popup)

    def popup(self, event):
        """Open a popup menu."""
        self.post(event.x_root, event.y_root)
        return 'break'
```



```
"""Create a context menu."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        l1 = Label("Context Menu", font="Arial 24")
        l2 = Label("Context Menu 2", font="Arial 48")

        ContextMenu(l1)
        Item('Item 1', 'print(1)')
        Item('Item 2', 'print(2)')
        Item('Item 3', 'print(3)')

        ContextMenu(l2)
        for c in 'ABC':
            Item('Item ' + c)

        ContextMenu(App.root)
        Item('Cut', 'print("Cut")')
        Item('Copy', 'print("Copy")')

Demo().run()
```

menu5.py

10.4 Insert an item at a specific position

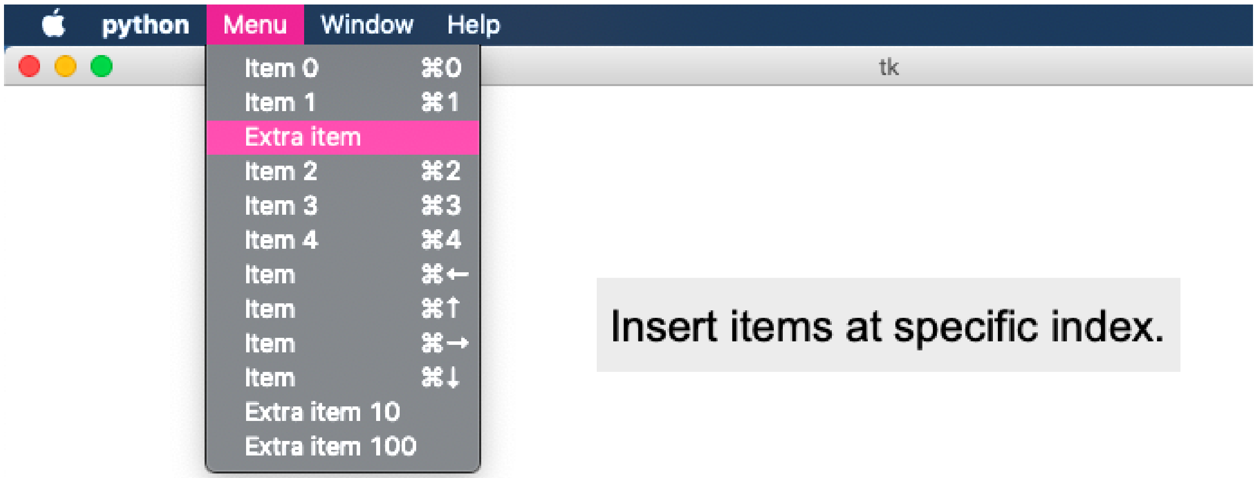
Normally we build menus in order. We create a `Menu()` object, and then we add the `Item()` objects subsequently. With the `insert method` it is possible to attach an item anywhere in the existing menu.

This inserts an new command at index=2:

```
App.menus[-1].insert(2, 'command', label='Extra item')
```

This inserts a new command at the end:

```
App.menus[-1].insert(100, 'command', label='Extra item 100')
App.menus[-1].insert(10, 'command', label='Extra item 10')
```



```
"""Insert items at a specific index position."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Insert items at specific index.", font="Arial 24")

        Menu('Menu')
        for i in range(5):
            Item('Item {}'.format(i), cmd=lambda : print(i), acc='Command-'+str(i))

        Item('Item', 'print("Left")', acc='Command-Left')
        Item('Item', 'print("Up")', acc='Command-Up')
        Item('Item', 'print("Right")', acc='Command-Right')
        Item('Item', 'print("Down")', acc='Command-Down')

        App.menus[-1].insert(2, 'command', label='Extra item')
        App.menus[-1].insert(100, 'command', label='Extra item 100')
        App.menus[-1].insert(10, 'command', label='Extra item 10')

        Menu('Window', name='window')
        Menu('Help', name='help')

Demo().run()
```

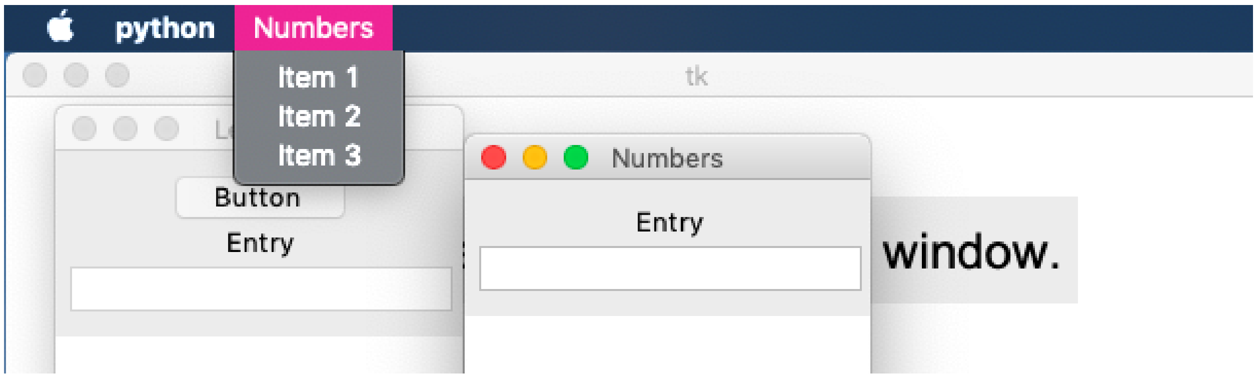
menu6.py

10.5 Different menus for each window

If no new menu is defined for a new window, the root menu will be used for the new window. If a new menu is added after the window is created, this window will have it's own menu. Whenever a window becomes active, it will display it's own menu bar.

This shows the creation of two new windows with their different menubars:

```
Window("Letters")
Button()
Entry()
Menu1()
Menu('Help', name='help')

Window("Numbers")
Entry()
Menu2()
```



```
"""Insert different menus for each window."""
from tklib import *

def Menu1():
    Menu('Letters')
    for c in 'ABC':
        Item('Item ' + c)

def Menu2():
    Menu('Numbers')
    for c in '123':
        Item('Item ' + c)

class Demo(App):
    # constructor
    def __init__(self):
        super().__init__()
        Label("Different menus for each window.", font="Arial 24")

        Menu1()
        Menu2()
```

(continues on next page)

(continued from previous page)

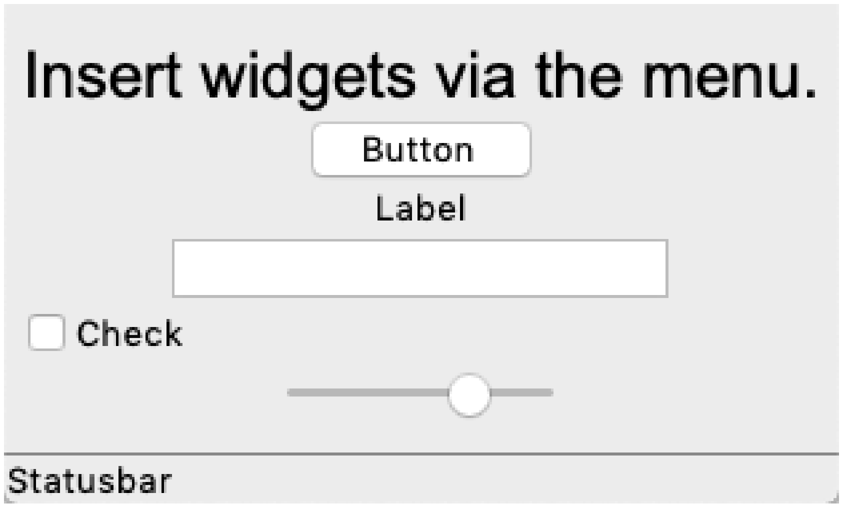
```
        Window("Letters")
        Button()
        Entry()
        Menu1()
        Menu('Help', name='help')

        Window("Numbers")
        Entry()
        Menu2()

Demo().run()
```

menu7.py

10.6 Creating new widgets from the menu

The next example shows how to add widgets dynamically. The `Widget` menu contains specific widgets as items which are added to the current window:

```
Menu('Widgets')
Item('Button', 'Button()', 'Command-b')
Item('Label', 'Label()', 'Command-l')
Item('Entry', 'Entry()', 'Command-e')
Item('Radiobutton', 'Radiobutton()', 'Command-r')
Item('Checkbutton', 'Checkbutton()', 'Command-k')
Item('Canvas', 'Canvas()', 'Command-c')
Item('Listbox', 'Listbox(height=5)', 'Command-i')
Item('Scale', 'Scale()', 'Command-s')
Item('Text', 'Text(width=30, height=5)', 'Command-t')
```



```
"""Insert widgets via a menu."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
```

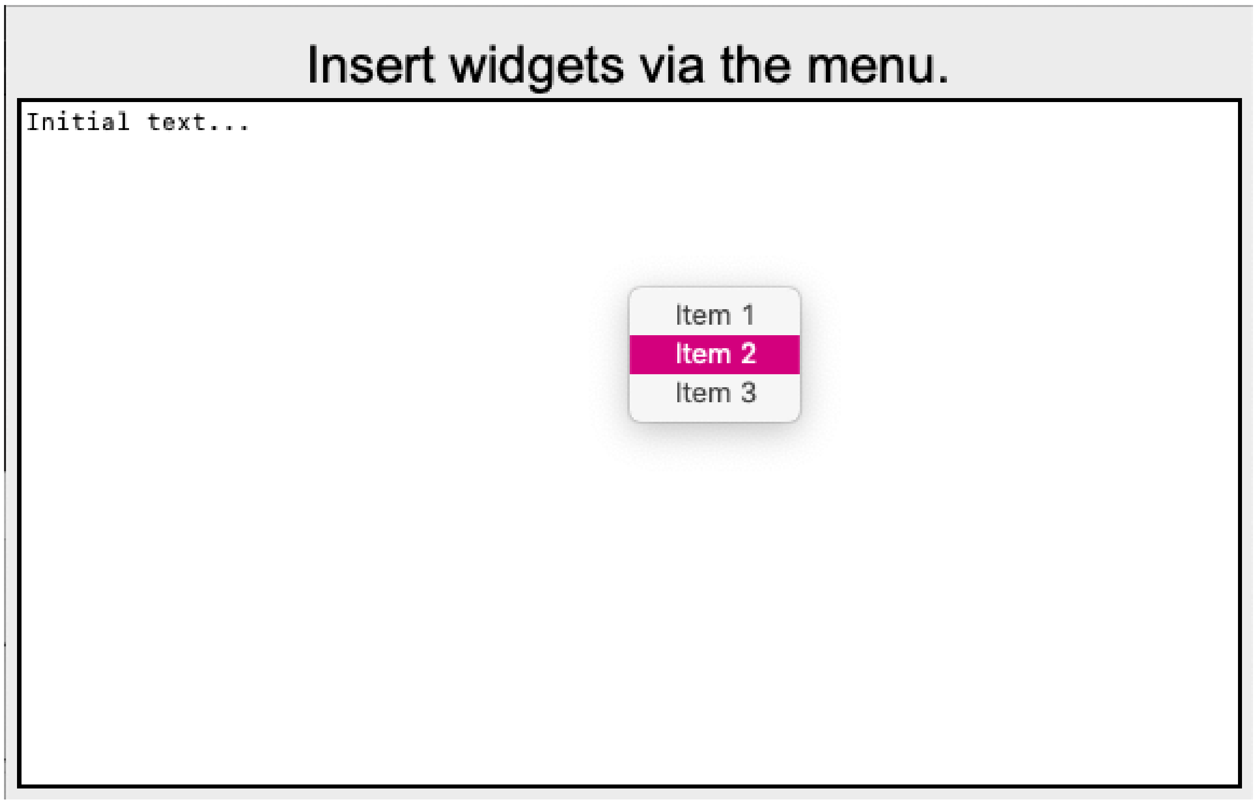
(continues on next page)

(continued from previous page)

```
        Label("Insert widgets via the menu.", font="Arial 24")

        Menu('Widgets')
        Item('Button', 'Button()', 'Command-b')
        Item('Label', 'Label()', 'Command-l')
        Item('Entry', 'Entry()', 'Command-e')
        Item('Radiobutton', 'Radiobutton()', 'Command-r')
        Item('Checkbutton', 'Checkbutton()', 'Command-k')
        Item('Canvas', 'Canvas()', 'Command-c')
        Item('Listbox', 'Listbox(height=5)', 'Command-i')
        Item('Scale', 'Scale()', 'Command-s')
        Item('Text', 'Text(width=30, height=5)', 'Command-t')

Demo().run()
```

menu8.py

10.7 Insert a popup menu


Insert widgets via the menu.

Initial text...

```
"""Insert a pop-up menu."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Insert widgets via the menu.", font="Arial 24")
```

(continues on next page)

(continued from previous page)

```
        text = Text('Initial text...')

        ContextMenu(text)
        Item('Item 1', 'print(1)')
        Item('Item 2', 'print(2)')
        Item('Item 3', 'print(3)')

Demo().run()
```

`menu9.py`

CHAPTER 11

Canvas

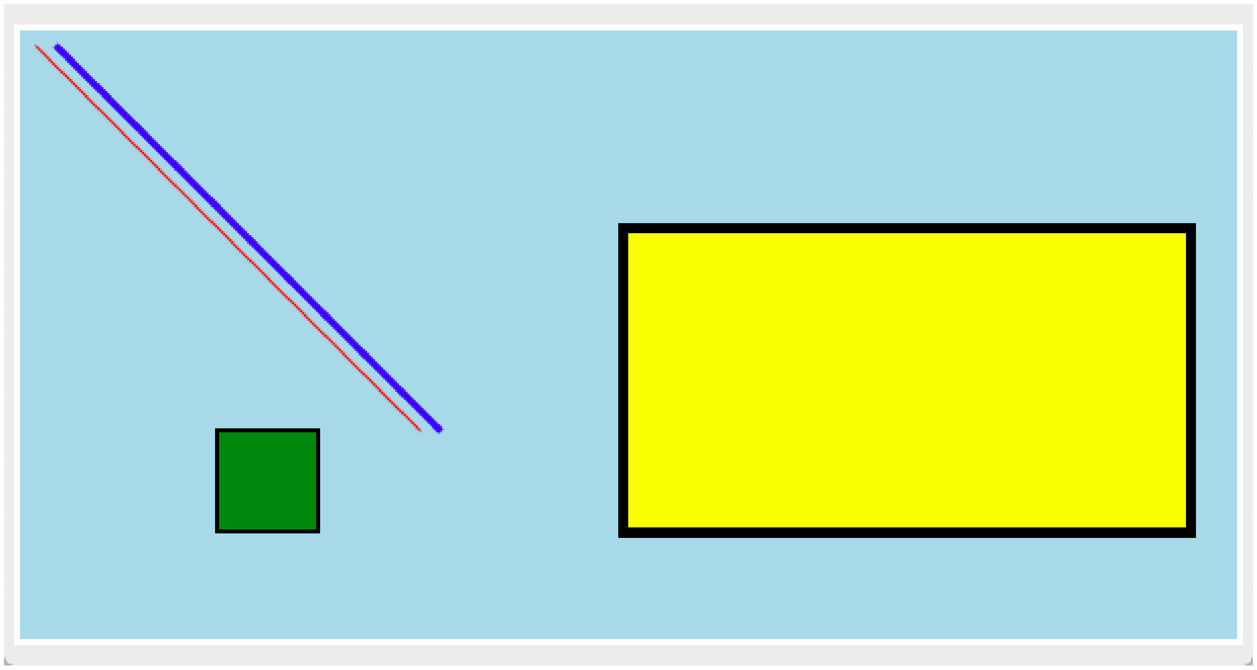
The **Canvas** widget can be used to draw lines, shapes, and text to create complex drawings and graphs. The origin (0, 0) of the canvas is in the top left corner. It has the keyword options:

- background = background color
- borderwidth
- height
- width

11.1 Draw lines and rectangles

To draw lines and rectangles, use these **create** methods:

- create_line()
- create_rectangle()



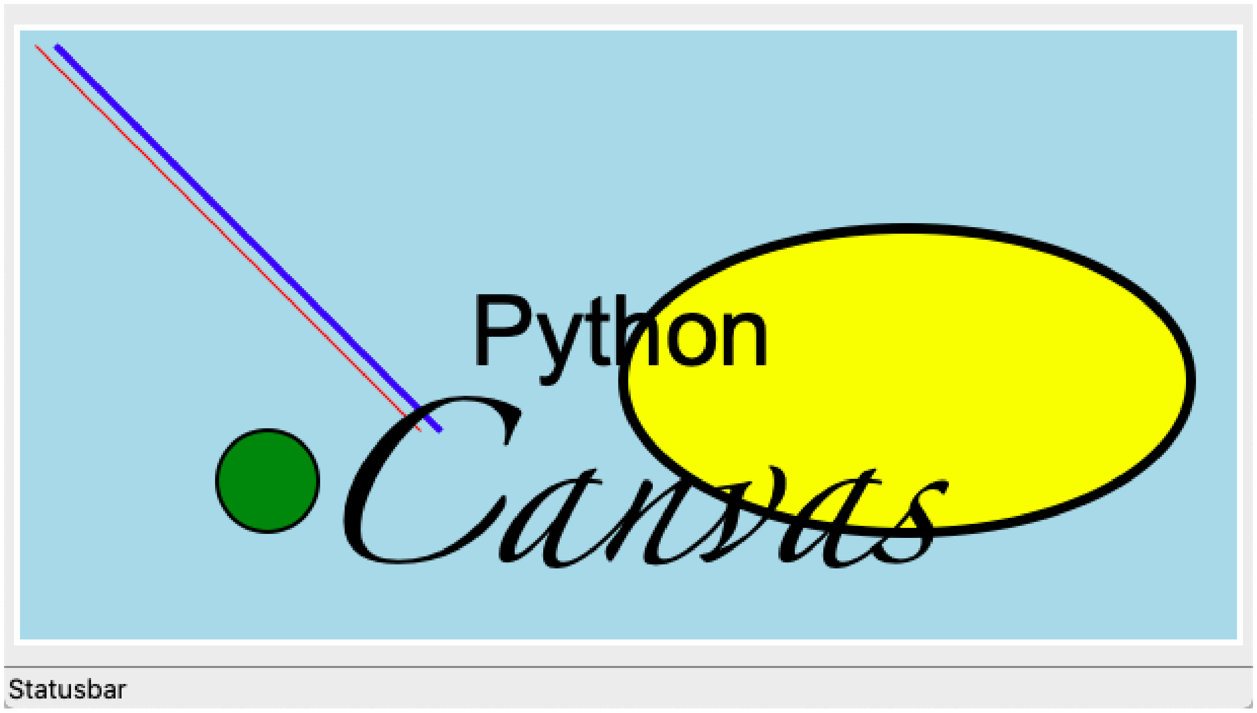
```
from tklib import *
app = App('Canvas with lines and rectangles')

c = Canvas(width=600, height=300, background='lightblue')
c.create_line(10, 10, 200, 200, fill='red')
c.create_line(20, 10, 210, 200, fill='blue', width=3)
c.create_rectangle(100, 200, 150, 250, fill='green', width=2)
c.create_rectangle(300, 100, 580, 250, fill='yellow', width=5)

app.run()
```

canvas1.py

11.2 Create text

Text can be added to a canvas with this function:

- create_text()

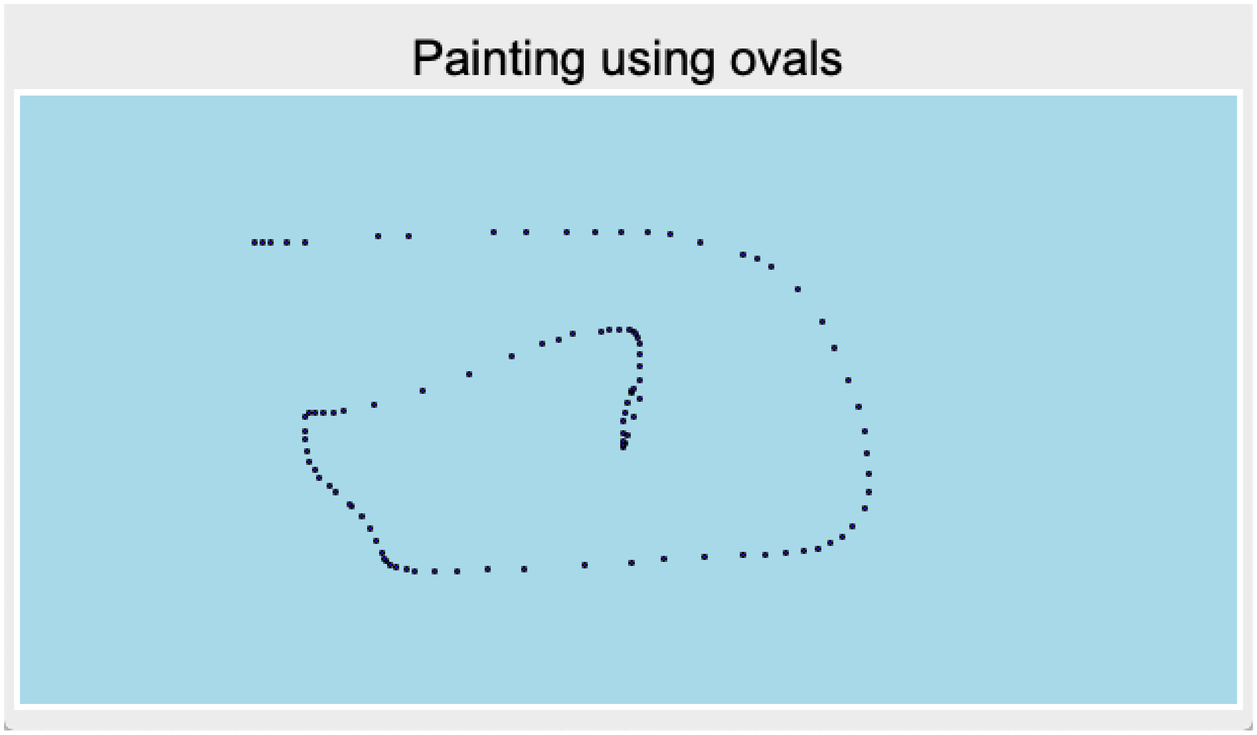


```python
from tklib import *
app = App('Canvas with ovals and text')

c = Canvas(width=600, height=300, background='lightblue')
c.create_line(10, 10, 200, 200, fill='red')
c.create_line(20, 10, 210, 200, fill='blue', width=3)
c.create_oval(100, 200, 150, 250, fill='green', width=2)
c.create_oval(300, 100, 580, 250, fill='yellow', width=5)
c.create_text(300, 150, text="Python", font='Arial 48')
c.create_text(300, 250, text="Canvas", font='Zapfino 72')

app.run()
```

canvas2.py

11.3 Paint with ovals

Small ovals can be used to paint with the mouse, by binding a callback function to the mouse movement.



```
"""Painting with ovals."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Painting using ovals", font="Arial 24")
        self.c = Canvas(width=600, height=300, background='lightblue')
        self.c.bind('<B1-Motion>', self.paint)

    def paint(self, event):
        """Draw ovals on the canvas."""
        x0, y0 = event.x-1, event.y-1
        x1, y1 = event.x+1, event.y+1
        self.c.create_oval(x0, y0, x1, y1, fill='blue')

Demo().run()
```

canvas3.py

11.4 Polygons

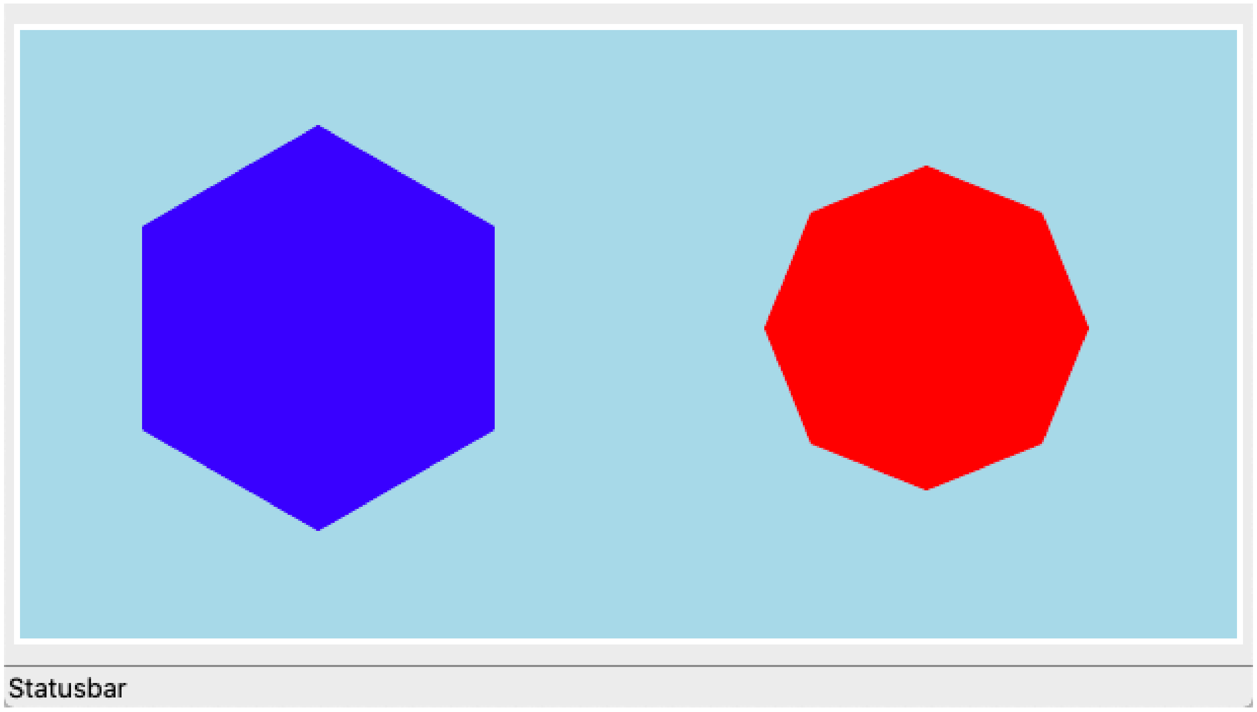
We can add our own methods to the Canvas class. For example we can define a method to add a polygon.

```
    def polygon(self, x0, y0, r, n, **kwargs):
        points = []
        for i in range(n):
            a = 2 * math.pi * i / n
            x = x0 + math.sin(a) * r
            y = y0 + math.cos(a) * r
```

(continues on next page)

(continued from previous page)

```
            points.append(x)
            points.append(y)
        self.create_polygon(points, **kwargs)
```

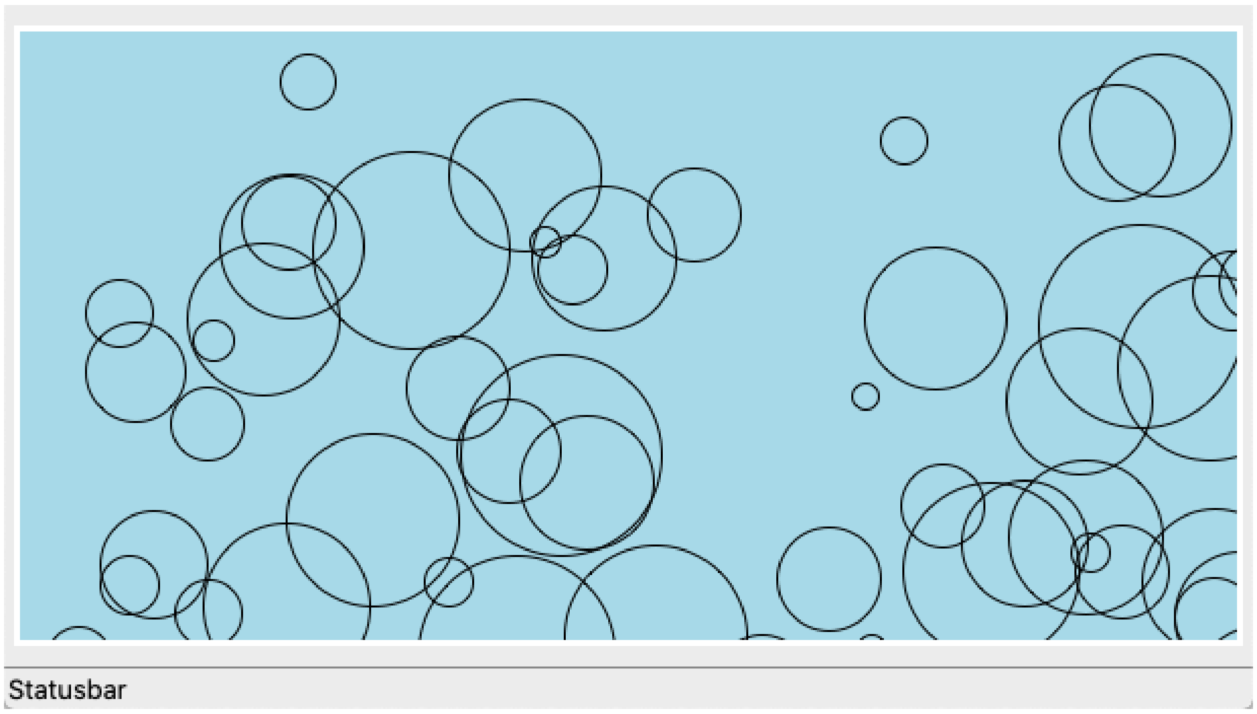


```
from tklib import *
app = App('Draw a polygon')

c = Canvas(width=600, height=300, background='lightblue')
c.polygon(150, 150, 100, 6, fill='blue')
c.polygon(450, 150, 80, 8, fill='red', width=5)

app.run()
```

canvas4.py

11.5 Random circles

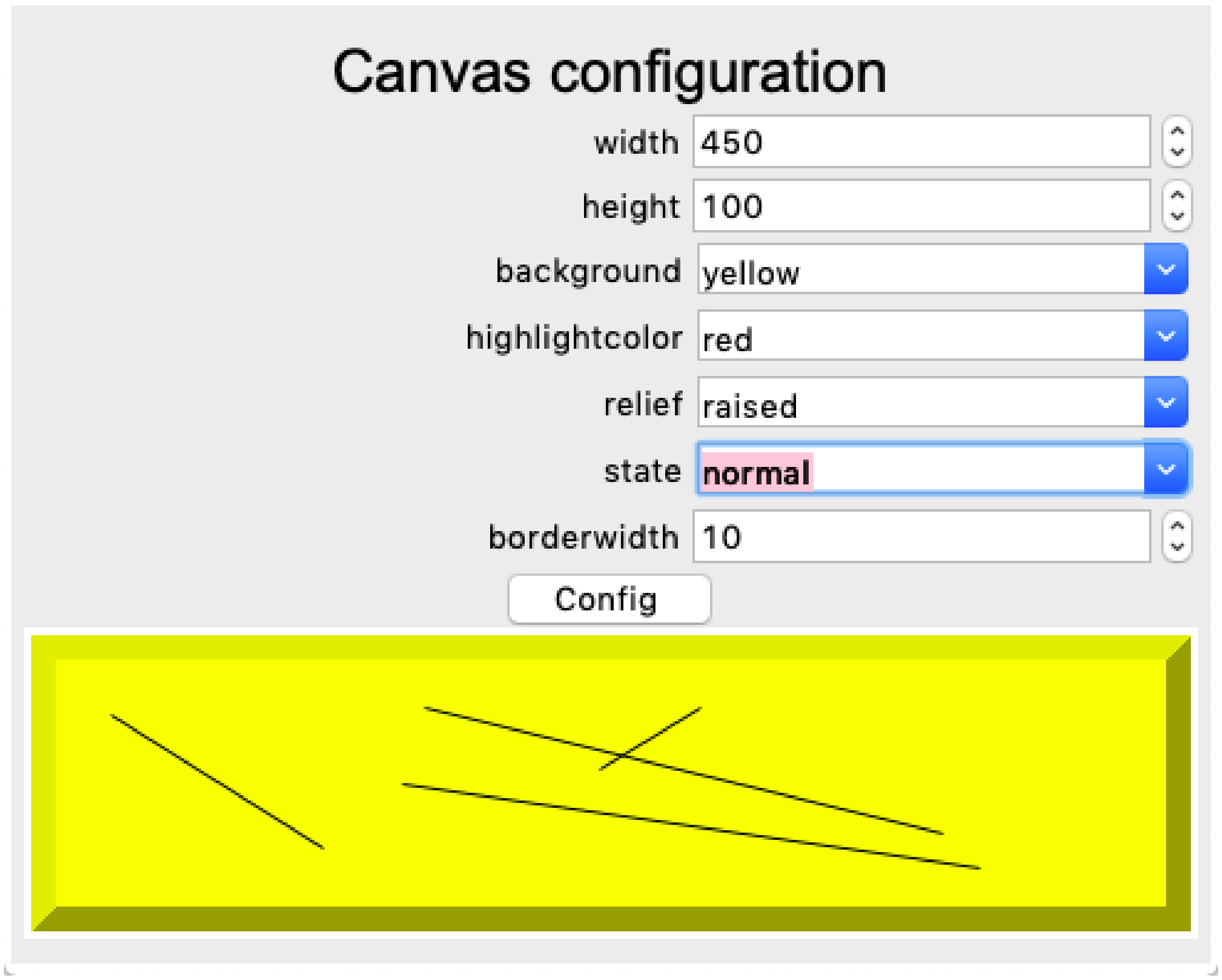
The following program places circles of random size at random locations.



```
from tklib import *
app = App('Random circles')

w, h = 600, 300
c = Canvas(width=w, height=h, background='lightblue')

for i in range(50):
    x = random.randint(0, w)
    y = random.randint(0, h)
    r = random.randint(10, 100)
    c.create_oval(x, y, x+r, y+r)

app.run()
```

canvas5.py

11.6 Canvas configuration



```
"""Canvas config."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Canvas configuration", font="Arial 24")

        Spinbox('width', 'App.c["width"]=self.var.get()', inc=50, to=1000)
        Spinbox('height', 'App.c["height"]=self.var.get()', inc=50, to=1000)
        Combobox('background', 'white;yellow;pink;light blue', 'App.c.
↪config(background=self.var.get())')
        Combobox('highlightcolor', 'black;red;blue;green', 'App.c.
↪config(highlightcolor=self.var.get())')
        Combobox('relief', 'flat;sunken;raised;groove', 'App.c.config(relief=self.var.
↪get())')
        Combobox('state', 'normal;disabled;hidden', 'App.c.config(state=self.var.
↪get())')
        Spinbox('borderwidth', 'App.c.config(borderwidth=self.var.get())')
```

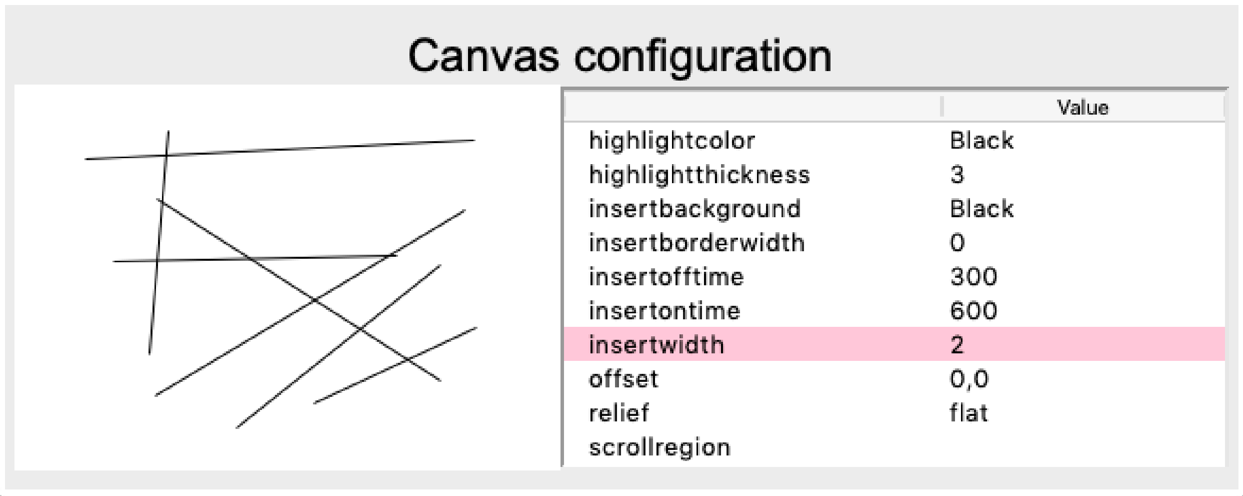
(continues on next page)

(continued from previous page)

```
        Button('Config', 'print(App.c.config())')

        App.c = Canvas()

Demo().run()
```

canvas6.py

11.7 Canvas configuration with tree view



Canvas configuration

	Value
highlightcolor	Black
highlightthickness	3
insertbackground	Black
insertborderwidth	0
insertofftime	300
insertontime	600
insertwidth	2
offset	0,0
relief	flat
scrollregion	

```
"""Canvas configuration with Treeview"""
from tklib import *

class Option:
    def __init__(self, widget):
        self.w = widget
        tree = Treeview(columns=(0))
        tree.column(0, width=150)
        tree.heading(0, text='Value')
        tree.grid(row=0, column=2)

        d = self.w.config()
        print(d)
        for (k, v) in d.items():
            if len(v)>2:
                tree.insert('', 'end', text=k, values=v[-1])


class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Canvas configuration", font="Arial 24")

        App.stack[-1]=Frame()
        App.c = Canvas(background='lightblue',
```

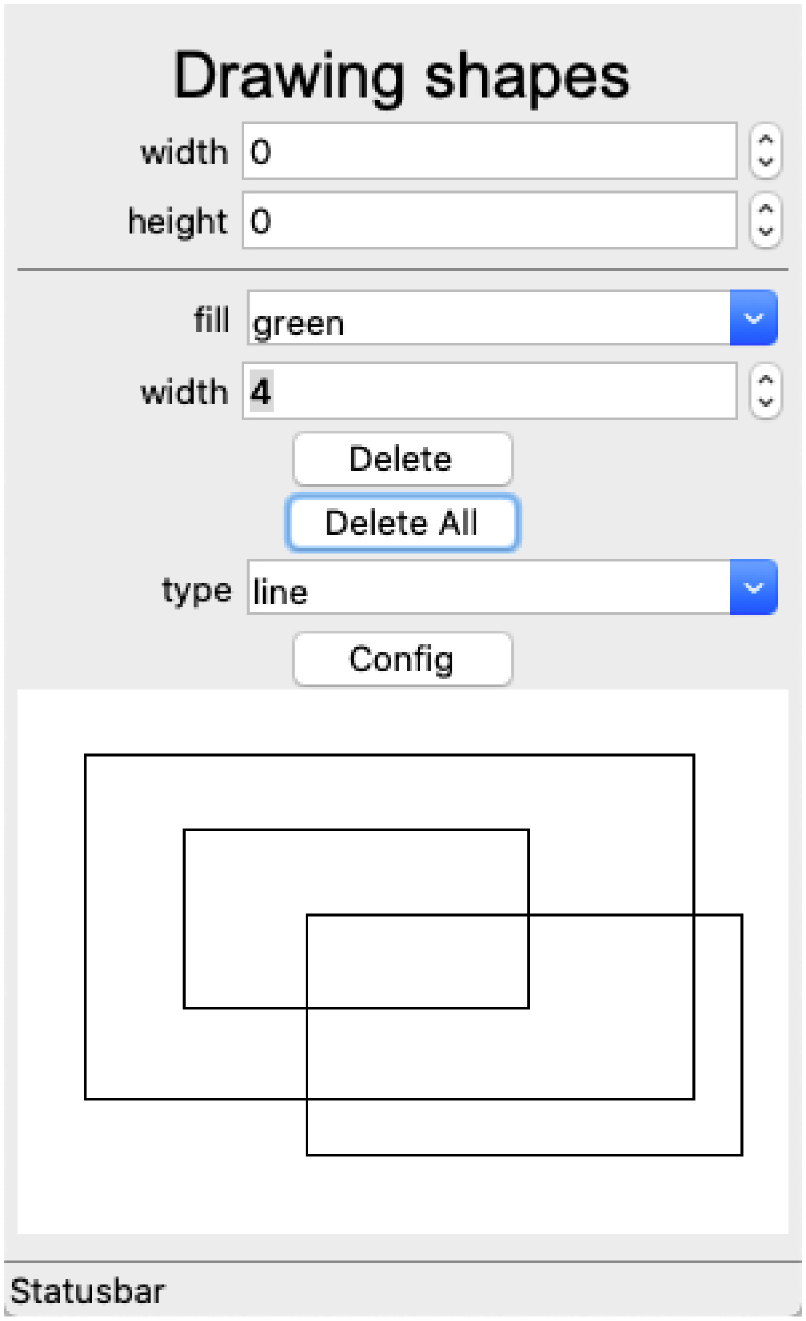
(continues on next page)

(continued from previous page)

```
            borderwidth=10,
            height=250)
        d = App.c.config()

        tree = Treeview(columns=(0))
        tree.column(0, width=150)
        tree.heading(0, text='Value')
        tree.grid(row=0, column=1)

        for (k, v) in d.items():
            if len(v)>2:
                tree.insert('', 'end', text=k, values=v[-1])

        Option(App.c)

Demo().run()
```

canvas7.py

11.8 Draw shapes with the mouse



```
"""Draw shapes with the mouse."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Drawing shapes", font="Arial 24")

        Spinbox('width', 'App.c["width"]=self.val.get()', inc=50, to=1000)
```

(continues on next page)

(continued from previous page)

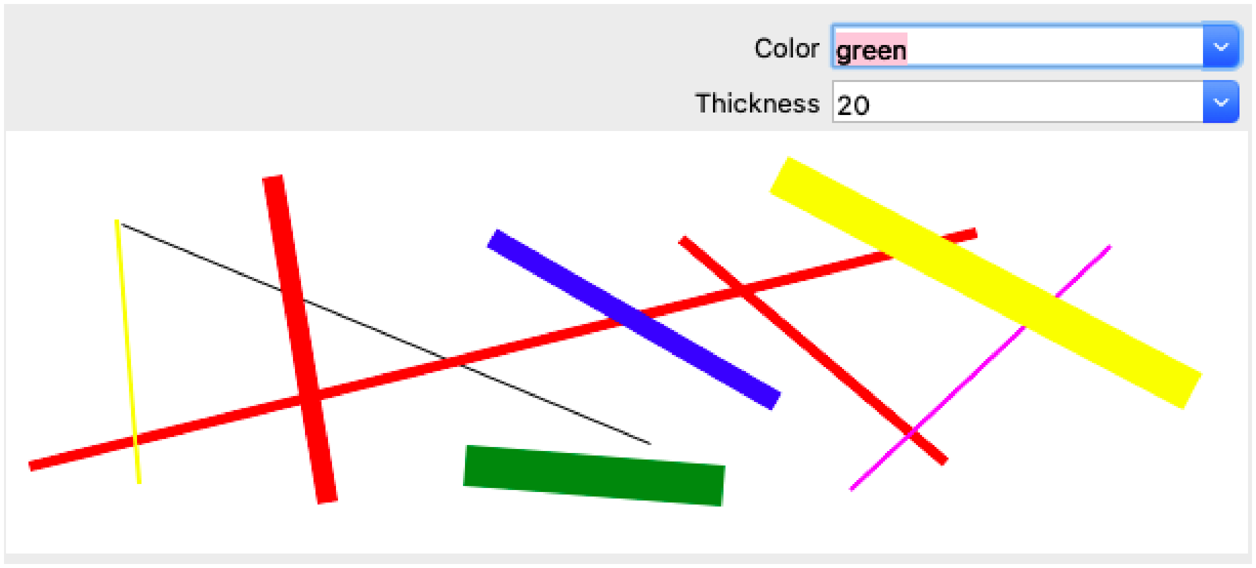
```
        Spinbox('height', 'App.c["height"]=self.val.get()', inc=50, to=1000)
        Separator()

        Combobox('fill', 'black;red;green;blue;orange;cyan',
            'App.c.itemconfig(App.c.id, fill=self.val.get())')
        Spinbox('width', 'App.c.itemconfig(App.c.id, width=self.val.get())', from_=1,␣
↪to=20)
        Button('Delete', 'App.c.delete(App.c.id)')
        Button('Delete All', 'App.c.delete("all")')
        Combobox('type', 'arc;line;rectangle;oval',
            'App.c.itemconfig(App.c.id, fill=self.val.get())')

        Button('Config', 'print(App.c.itemconfig(1))')

        App.c = Canvas()
        print(vars(App.c))
        print()
        print(dir(App.c))

        App.c.create_rectangle(20, 20, 150, 100)

Demo().run()
```

canvas9.py

11.9 Draw straight lines with the mouse

In order to draw with the mouse we have to add two bindings to the canvas:

- **<Button-1>** to initiate the drawing, calling the `start()` method
- **<B1_Motion>** to update the current drawing, calling the `move()` method



```
# draw straight lines on a canvas
from tklib import *
```

(continues on next page)

(continued from previous page)

```
class Canvas(tk.Canvas):
    """Define a canvas with line drawing"""

    def __init__(self, **kwargs):
        # super(Canvas, self).__init__(App.stack[-1], width=w, height=h, bg='light␣
↪blue')
        super(Canvas, self).__init__(App.stack[-1], **kwargs)
        self.grid()
        self.bind('<Button-1>', self.start)
        self.bind('<B1-Motion>', self.move)
        self.bind('<D>', self.print_msg)
        self.bind('<Key>', self.print_msg)
        self.bind('<Return>', self.print_msg)


    def start(self, event=None):
        # Execute a callback function.
        self.x0 = event.x
        self.y0 = event.y
        self.id = self.create_line(self.x0, self.y0, self.x0, self.y0)
        self.itemconfig(self.id, fill=color.var.get())
        self.itemconfig(self.id, width=thickness.var.get())

    def move(self, event=None):
        self.x1 = event.x
        self.y1 = event.y
        self.coords(self.id, self.x0, self.y0, self.x1, self.y1)

    def print_msg(self, event):
        print(event)

app = App('Drawing lines on a canvas')
color = Combobox('Color', 'black;red;green;blue;yellow;cyan;magenta')
thickness = Combobox('Thickness', '1;2;5;10;20')
Canvas(width=600, height=200)

app.run()
```

draw1.py

CHAPTER 12

Scrollbar

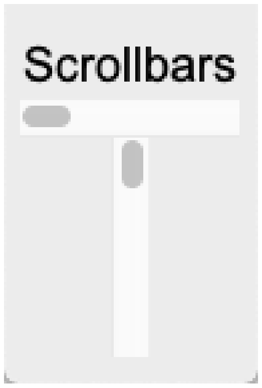
There are 4 widgets who can have a scrollbar:

- Text()
- Treeview()
- Canvas()
- Listbox()
- Entry()

In order to add scrollbars to these widgets we define a class which adds them and use multiple inheritance.

In order for scrollbar and widget to interact, callback functions are assigned on both sides.

- The scrollbar calls the widget's **xview** and **yview** methods.
- The widget has **[x/y]scrollcommmand** options to update the scrollbar



```
"""Display scrollbars."""
from tklib import *

class Demo(App):
```

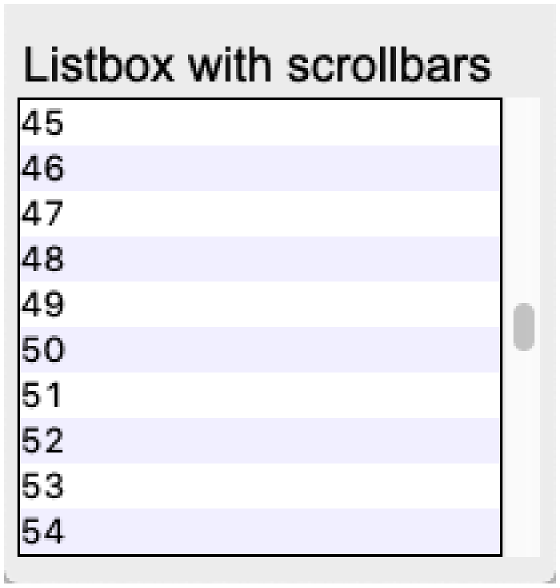
(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()
        Label("Scrollbars", font="Arial 18")

        ttk.Scrollbar(App.stack[-1], orient=tk.HORIZONTAL).grid()
        ttk.Scrollbar(App.stack[-1], orient=tk.VERTICAL).grid()

Demo().run()
```

scrollbar1.py

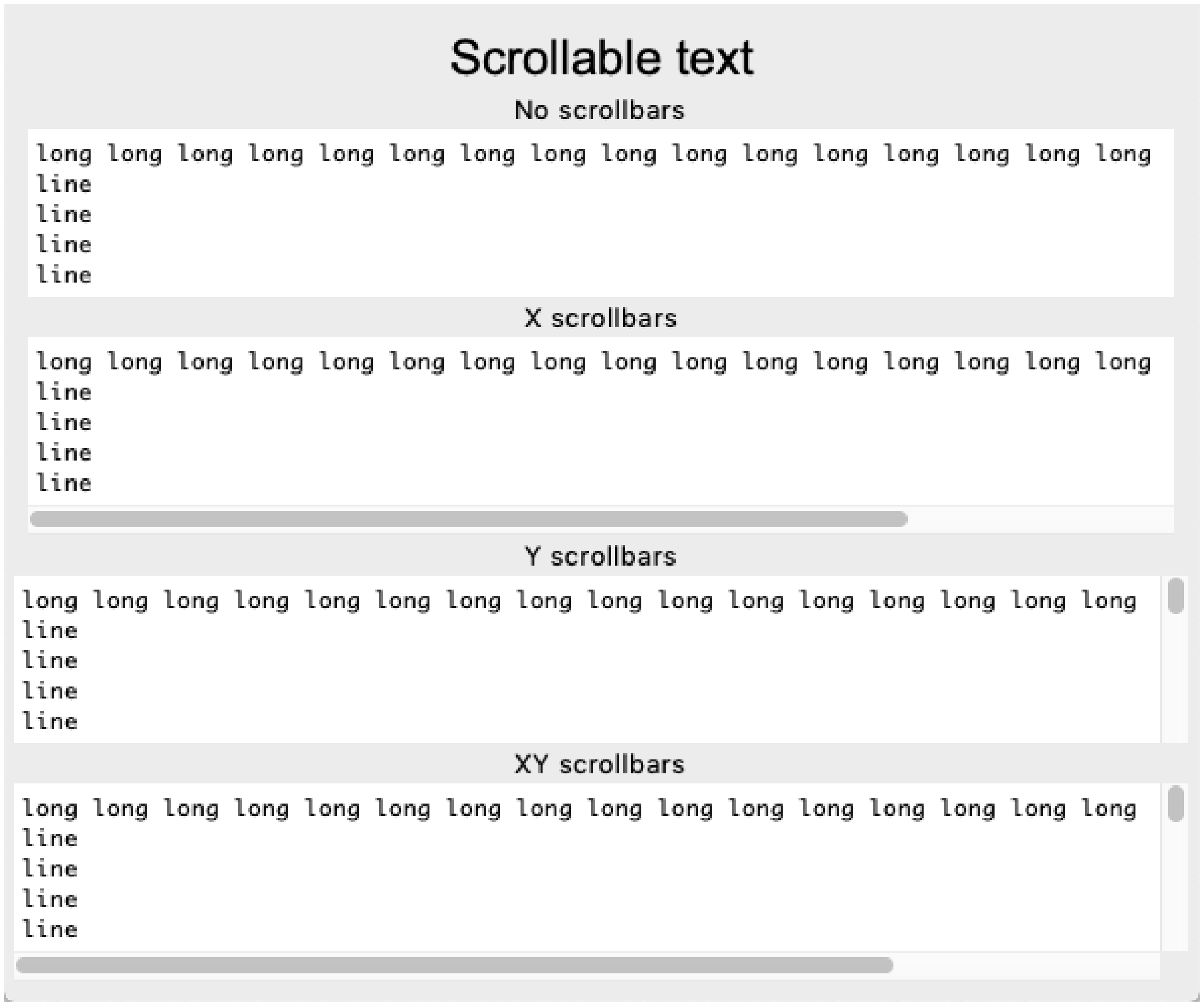
12.1 Listbox with scrollbars



```
"""Listbox with a scrollbar."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Listbox with scrollbars", font="Arial 18")

        lb = Listbox(list(range(100)))
        sb = ttk.Scrollbar(App.stack[-1])
        sb.grid(row=1, column=1, sticky='ns')

        lb.config(yscrollcommand=sb.set)
        sb.config(command=lb.yview)

Demo().run()
```

scrollbar2.py

12.2 Text with scrollbars


Scrollable text

No scrollbars

```
long long long long long long long long long long long long long long long long
line
line
line
line
```

X scrollbars

```
long long long long long long long long long long long long long long long long
line
line
line
line
```

Y scrollbars

```
long long long long long long long long long long long long long long long long
line
line
line
line
```

XY scrollbars

```
long long long long long long long long long long long long long long long long
line
line
line
line
```

```python
"""Text with a scrollbar."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Scrollable text', font='Arial 24')

        text = 'long ' * 20
        text += 'line\n' * 20

        Label("No scrollbars")
        Text(text, height=5, wrap='none')

        Label("X scrollbars")
        Text(text, height=5, scroll='x', wrap='none')

        Label("Y scrollbars")
```

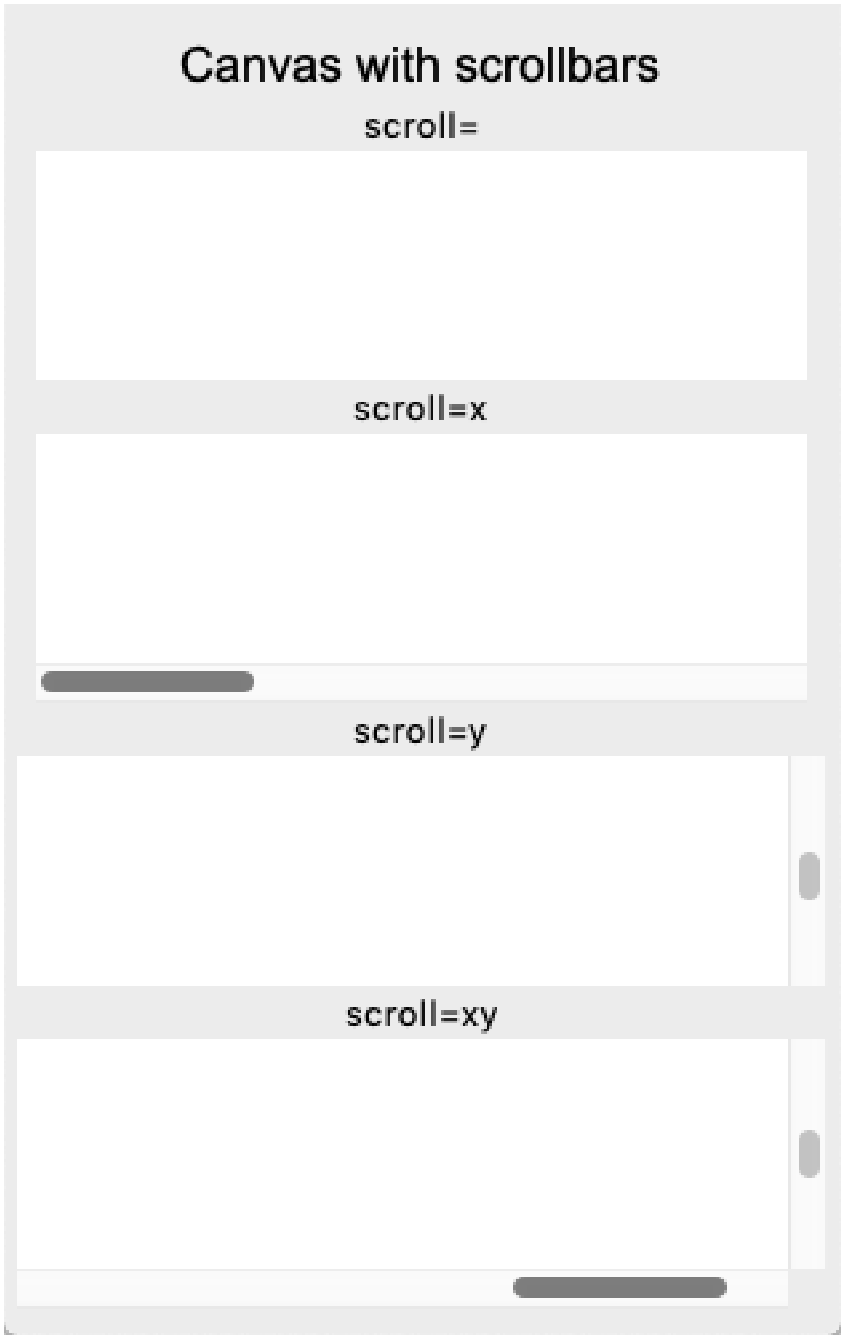
(continues on next page)

(continued from previous page)

```
        Text(text, height=5, scroll='y', wrap='none')

        Label("XY scrollbars")
        Text(text, height=5, scroll='xy', wrap='none')

Demo().run()
```

scrollbar3.py

12.3 Canvas with scrollbars



```
"""Canvas with a scrollbar."""
from tklib import *

class Canvas(tk.Canvas):
    def __init__(self, scroll='', scrollregion=(0, 0, 1000, 1000), **kwargs):
        self = Scrollable(tk.Canvas, scroll=scroll, scrollregion=scrollregion,␣
↪**kwargs)
```

(continues on next page)

(continued from previous page)

```
class Demo(App):
    def __init__(self):
        super().__init__()
        text = 'hello ' * 100
        Label("Canvas with scrollbars", font="Arial 18")
        h = 80

        Label('scroll=')
        Canvas(height=h)

        Label('scroll=x')
        Canvas(height=h, scroll='x')

        Label('scroll=y')
        Canvas(height=h, scroll='y')

        Label('scroll=xy')
        Canvas(height=h, scroll='xy')

Demo().run()
```

scrollbar5.py

12.4 Sources

- http://effbot.org/zone/tkinter-scrollbar-patterns.htm

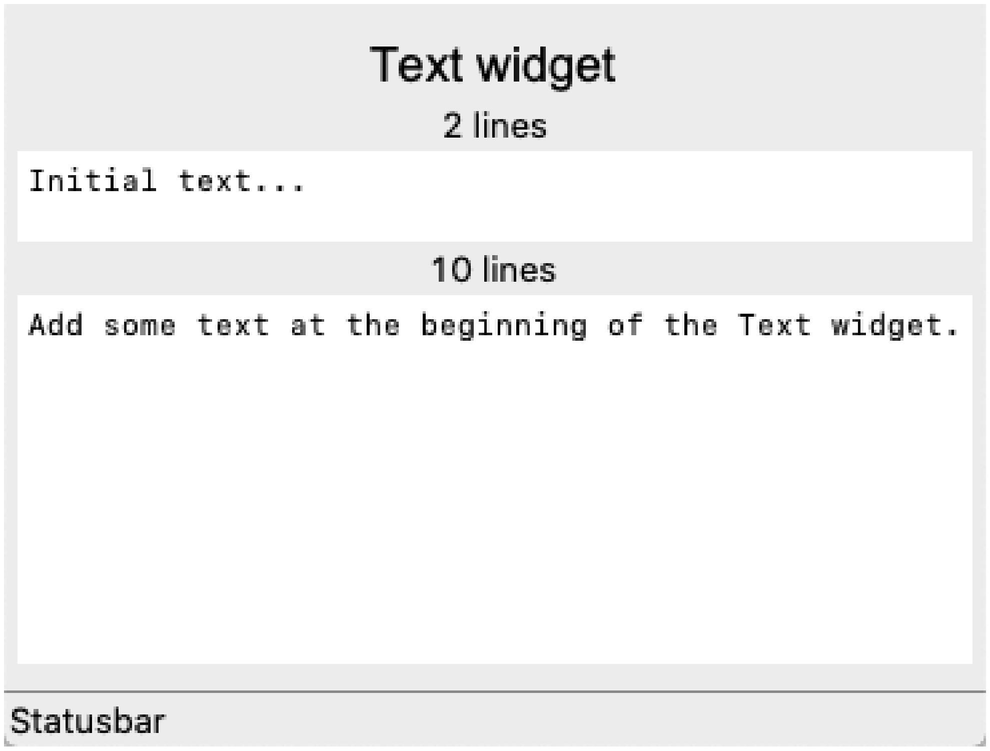
CHAPTER 13

Text

The **Text()** widget manages muliline text.

13.1 Display multi-line text

The `Text` widget displays multi-line text and is user editable. Its size is given in lines and characters.



```
Initial text...
```

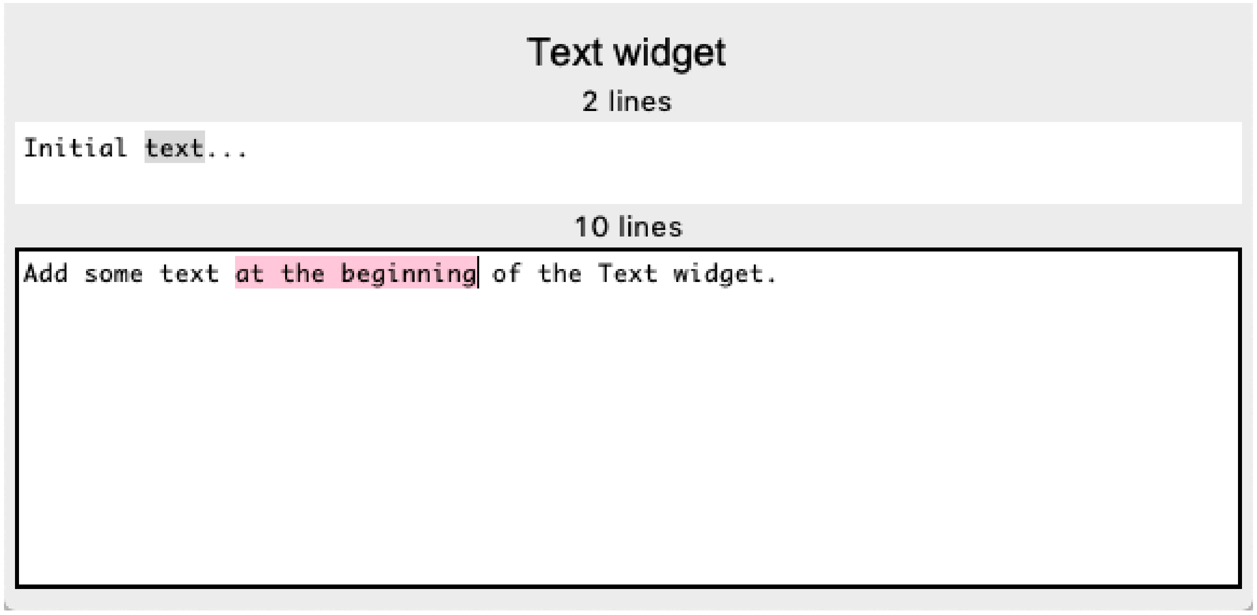
```
Add some text at the beginning of the Text widget.
```

```
"""Display tk Text."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Text widget", font="Arial 18")

        Label('2 lines')
        Text('Initial text...', height=2, width=50)

        Label('10 lines')
        text = Text(height=10, width=50)
        text.insert('1.0', 'Add some text at the beginning of the Text widget.')

Demo().run()
```

text1.py

13.2 Edit text

Text can be edited as usual. Cut, copy, paste is possible. The cursor can be placed with the mouse. Mouse and and arrow key selection are both possible.



Text widget

2 lines

```
Initial text...
```

10 lines

```
Add some text at the beginning of the Text widget.
```

```
"""Display tk Text."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Text widget", font="Arial 18")

        Spinbox('width', 'App.text["width"]=self.val.get()').set(80)
```

(continues on next page)

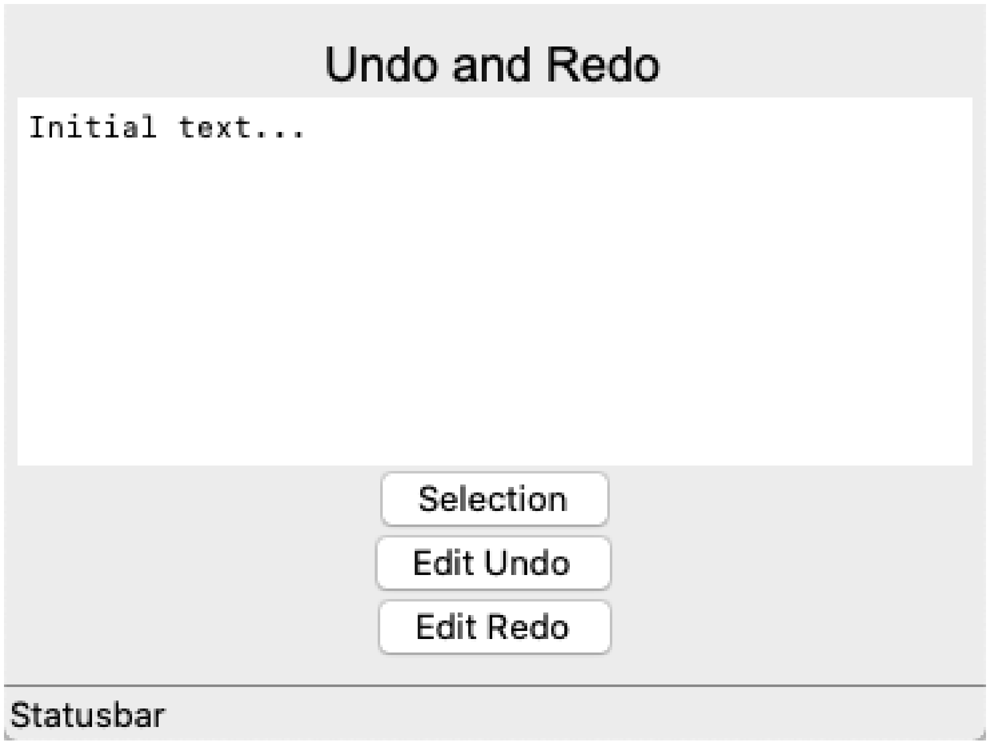
(continued from previous page)

```
        Spinbox('height', 'App.text["height"]=self.val.get()').set(4)

        App.text = Text('Initial text...', height=2)

Demo().run()
```

text2.py

13.3 Undo and redo text edit

Text has an **Undo** and **Redo** function.


Undo and Redo

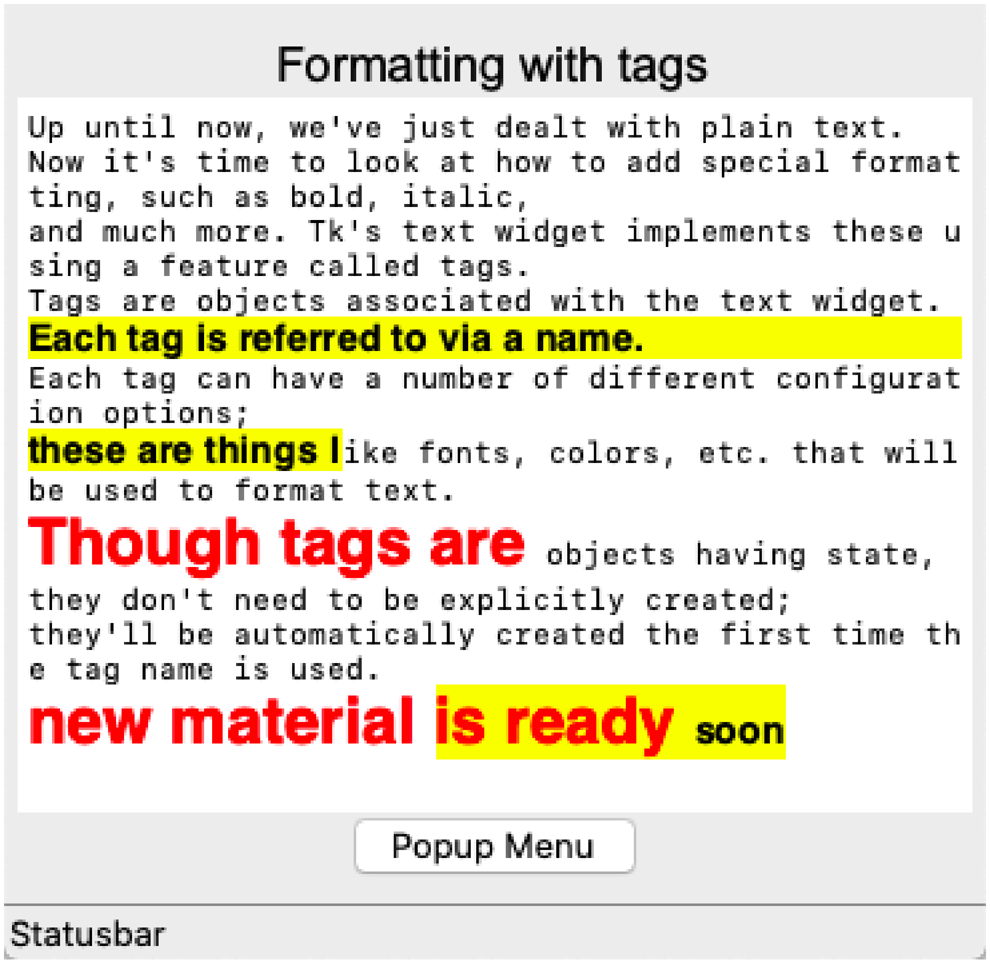
Initial text...

```
"""Undo and Redo."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Undo and Redo", font="Arial 18")

        App.text = Text('Initial text...', height=10, width=50)
        App.text.config(undo=True)

        Button('Selection')
        Button('Edit Undo', 'App.text.edit_undo()')
        Button('Edit Redo', 'App.text.edit_redo()')
        Inspector(App.text, height=20)

Demo().run()
```

text3.py

13.4 Format text with tags

Text can be formatted with tags.



```
Up until now, we've just dealt with plain text.
Now it's time to look at how to add special format
ting, such as bold, italic,
and much more. Tk's text widget implements these u
sing a feature called tags.
Tags are objects associated with the text widget.
Each tag is referred to via a name.
Each tag can have a number of different configurat
ion options;
these are things like fonts, colors, etc. that will
be used to format text.
Though tags are objects having state,
they don't need to be explicitly created;
they'll be automatically created the first time th
e tag name is used.
new material is ready soon
```

```
from tklib import *

str = """Up until now, we've just dealt with plain text.
Now it's time to look at how to add special formatting, such as bold, italic,
and much more. Tk's text widget implements these using a feature called tags.
Tags are objects associated with the text widget.
Each tag is referred to via a name.
Each tag can have a number of different configuration options;
these are things like fonts, colors, etc. that will be used to format text.
Though tags are objects having state, they don't need to be explicitly created;
they'll be automatically created the first time the tag name is used.
"""

class Demo(App):
    def __init__(self):
        super().__init__()
```

(continues on next page)

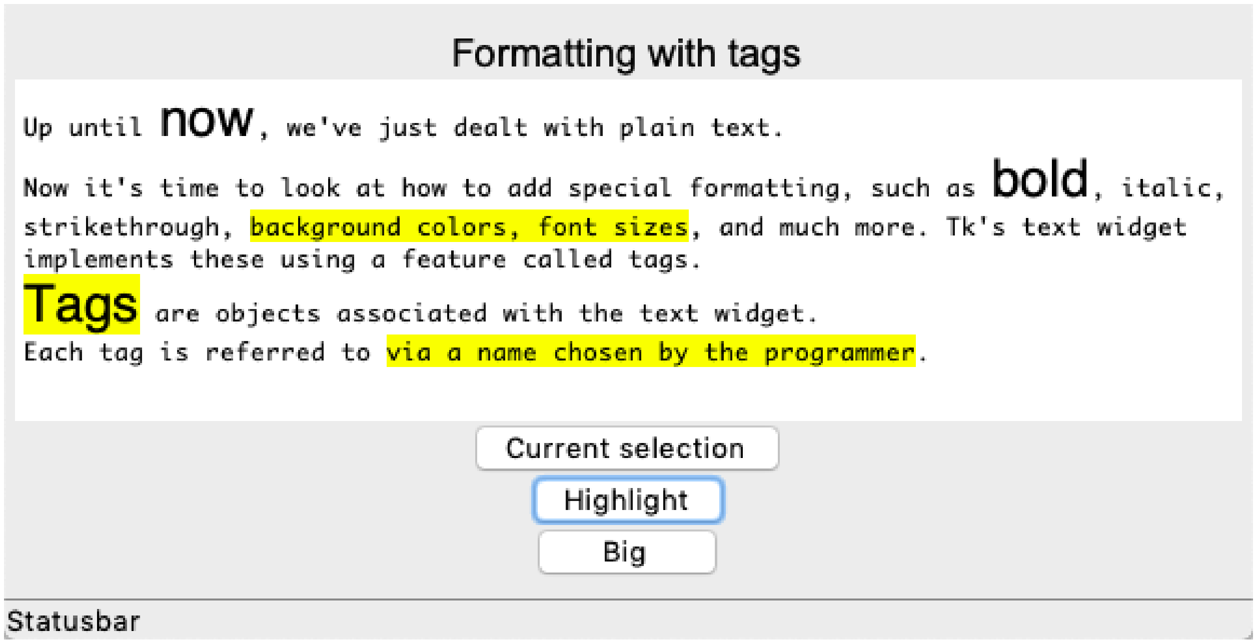
(continued from previous page)

```
        Label("Formatting with tags", font="Arial 18")

        App.text = Text(str, height=20, width=50)
        App.text.config(undo=True)

        App.text.tag_add('highlight', '5.0', '6.0')
        App.text.tag_add('highlight', '7.0', '7.18')
        App.text.tag_configure('highlight', background='yellow',
          font='helvetica 14 bold', relief='raised')

        App.text.tag_configure('big', font='helvetica 24 bold', foreground='red')
        App.text.tag_add('big', '8.0', '8.16')

        App.text.insert('end', 'new material ', ('big'))
        App.text.insert('end', 'is ready ', ('big', 'highlight'))
        App.text.insert('end', 'soon', ('highlight'))

        b = Button('Popup Menu')
        App.m = ContextMenu(b)
        Item('Item 1', 'print(1)')
        Item('Item 2', 'print(2)')
        Item('Item 3', 'print(3)')

        App.text.tag_bind('big', '<1>', App.m.popup)

Demo().run()
```

text4.py

13.5 Add custom formats

Custom formats can be added.


Formatting with tags

Up until now, we've just dealt with plain text.

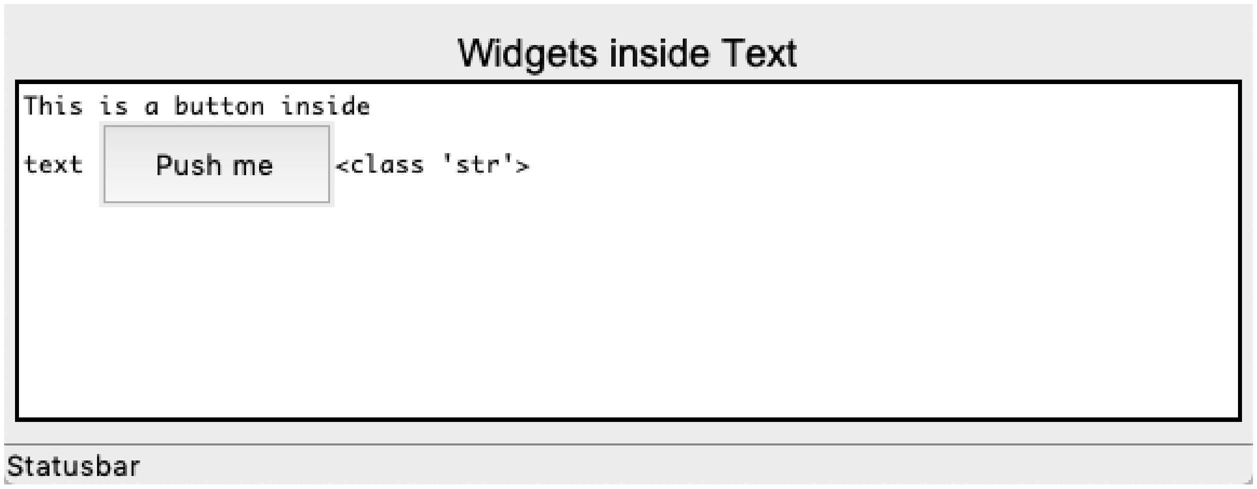
Now it's time to look at how to add special formatting, such as bold, italic, strikethrough, background colors, font sizes, and much more. Tk's text widget implements these using a feature called tags.
Tags are objects associated with the text widget.
Each tag is referred to via a name chosen by the programmer.

```
# Add custom styles
from tklib import *

str = """Up until now, we've just dealt with plain text.
Now it's time to look at how to add special formatting, such as bold, italic,
strikethrough, background colors, font sizes, and much more. Tk's text widget
implements these using a feature called tags.
Tags are objects associated with the text widget.
Each tag is referred to via a name chosen by the programmer.
"""

def highlight():
    sel = App.text.tag_ranges('sel')
    if len(sel) > 0:
        App.text.tag_add('highlight', *sel)

def big():
    sel = App.text.tag_ranges('sel')
    if len(sel) > 0:
        App.text.tag_add('big', *sel)

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Formatting with tags", font="Arial 18")

        App.text = Text(str, height=10)
        App.text.tag_configure('highlight', background='yellow')
        App.text.tag_configure('big', font='helvetica 24')

        Button('Selection range', 'print(App.text.tag_ranges("sel"))')
        Button('Highlight ranges', 'print(App.text.tag_ranges("highlight"))')
        Button('Big ranges', 'print(App.text.tag_ranges("big"))')
        Button('Mark names', 'print(App.text.mark_names())')

        Button('Highlight', highlight)
        Button('Big', big)

Demo().run()
```

text5.py

13.6 Place widgets inside text

Widgets can be placed inside text. They move with the text.



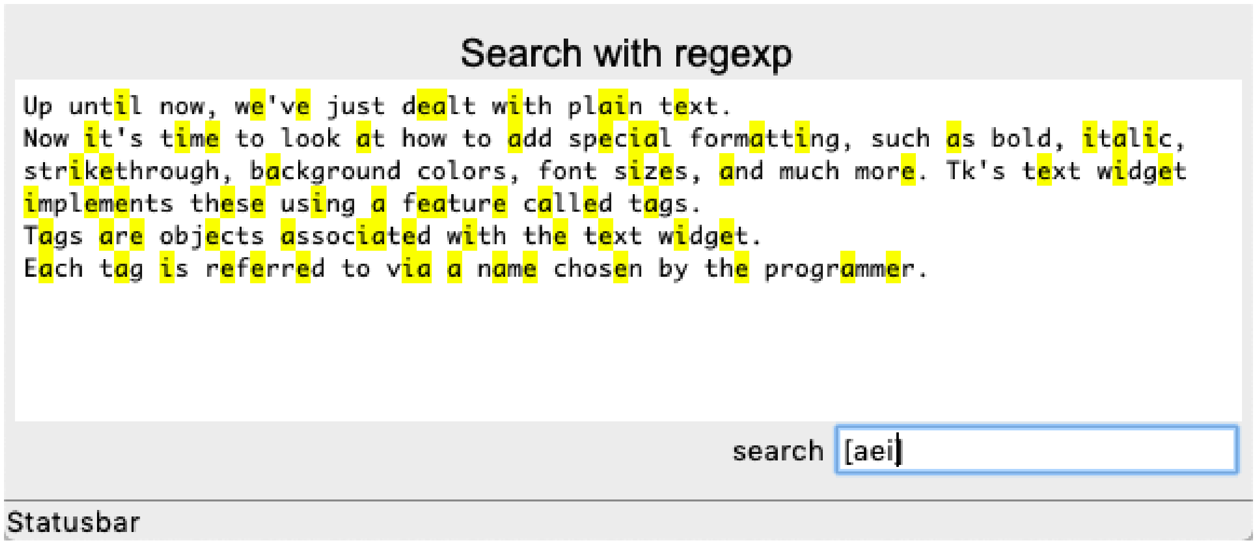
This is a button inside
text <class 'str'>

```
"""Widgets inside Text"""
from tklib import *

def hello():
    print('hello')

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Widgets inside Text", font="Arial 18")

        App.text = Text(str, height=10, width=50)
        b = ttk.Button(App.text, text='Push me', command=hello, padding=10)
        App.text.window_create('1.0', window=b)

Demo().run()
```

text6.py

13.7 Search inside text

Text can be searched. Regular expressions can be used. The exemple below shows the search for vowels which are highlighted in yellow.


Search with regexp

```
Up until now, we've just dealt with plain text.
Now it's time to look at how to add special formatting, such as bold, italic,
strikethrough, background colors, font sizes, and much more. Tk's text widget
implements these using a feature called tags.
Tags are objects associated with the text widget.
Each tag is referred to via a name chosen by the programmer.
```

search [aei]

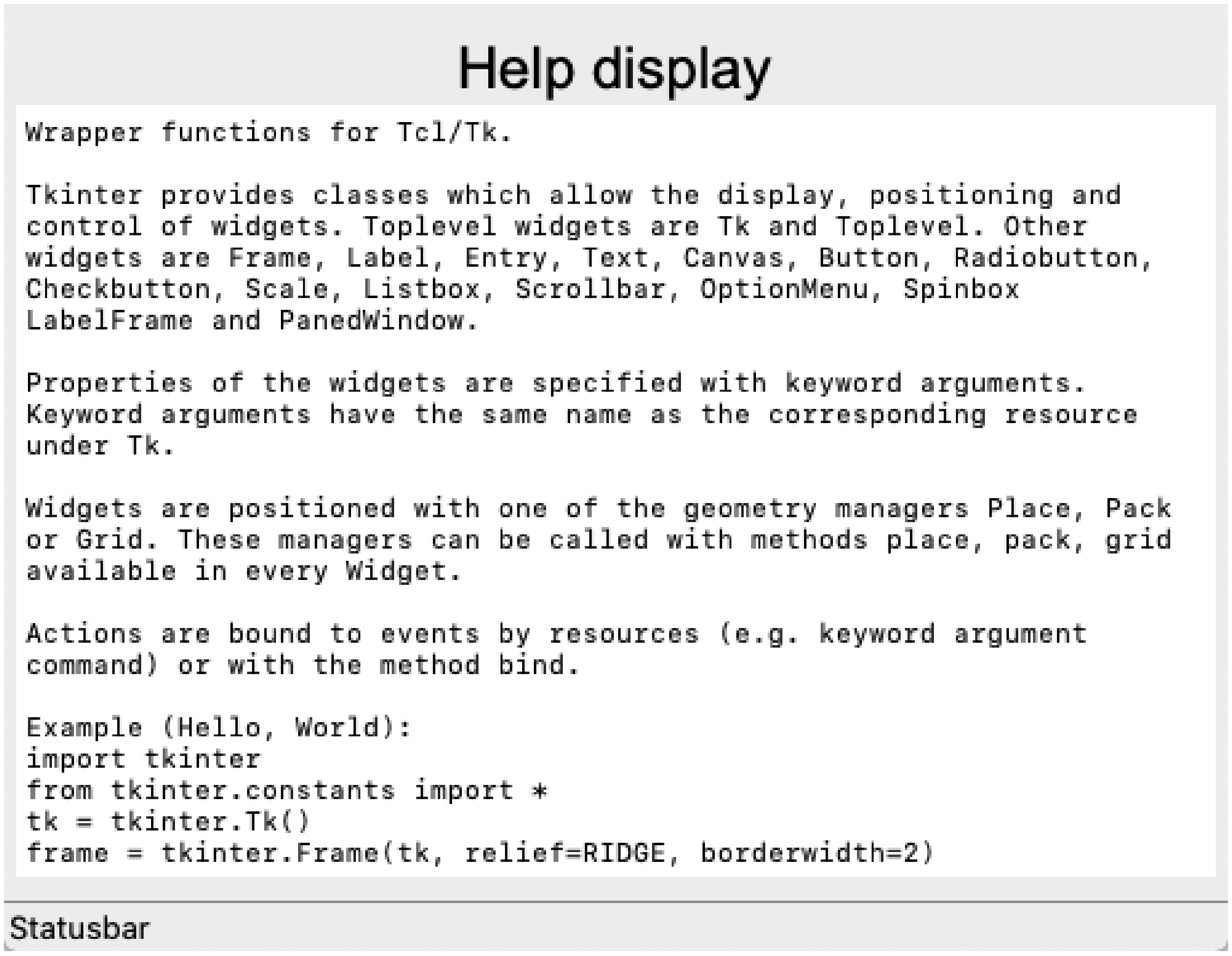
Statusbar

```
"""Search in text."""
from tklib import *

str = """Up until now, we've just dealt with plain text.
Now it's time to look at how to add special formatting, such as bold, italic,
strikethrough, background colors, font sizes, and much more. Tk's text widget
implements these using a feature called tags.
"""

def highlight():
    sel = App.text.tag_ranges('sel')
    if len(sel) > 0:
        App.text.tag_add('highlight', *sel)

def search(event=None):
    App.text.tag_remove("highlight", "1.0", "end")
    start = 1.0
    while True:
        pattern = App.re.val.get()
        pos = App.text.search(pattern, start, stopindex='end', regexp=True)
        if not pos:
            break
        print(pos)
        App.text.tag_add('highlight', pos, pos+'+1c')
        start = pos + '+1c'

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Search with regexp", font="Arial 18")

        App.text = Text(str, height=10, width=50)
        App.text.tag_configure('highlight', background='yellow')

        App.re = Entry('search', search)

Demo().run()
```

text7.py

13.8 Show a help text

Python being introspective it is easy to add help functionality.



```
Wrapper functions for Tcl/Tk.

Tkinter provides classes which allow the display, positioning and
control of widgets. Toplevel widgets are Tk and Toplevel. Other
widgets are Frame, Label, Entry, Text, Canvas, Button, Radiobutton,
Checkbutton, Scale, Listbox, Scrollbar, OptionMenu, Spinbox
LabelFrame and PanedWindow.

Properties of the widgets are specified with keyword arguments.
Keyword arguments have the same name as the corresponding resource
under Tk.

Widgets are positioned with one of the geometry managers Place, Pack
or Grid. These managers can be called with methods place, pack, grid
available in every Widget.

Actions are bound to events by resources (e.g. keyword argument
command) or with the method bind.

Example (Hello, World):
import tkinter
from tkinter.constants import *
tk = tkinter.Tk()
frame = tkinter.Frame(tk, relief=RIDGE, borderwidth=2)
```

```python
"""Help browser."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label("Help display", font="Arial 24")

        str = tk.__doc__
        App.text = Text(str, width=70)

Demo().run()
```

text8.py

CHAPTER 14

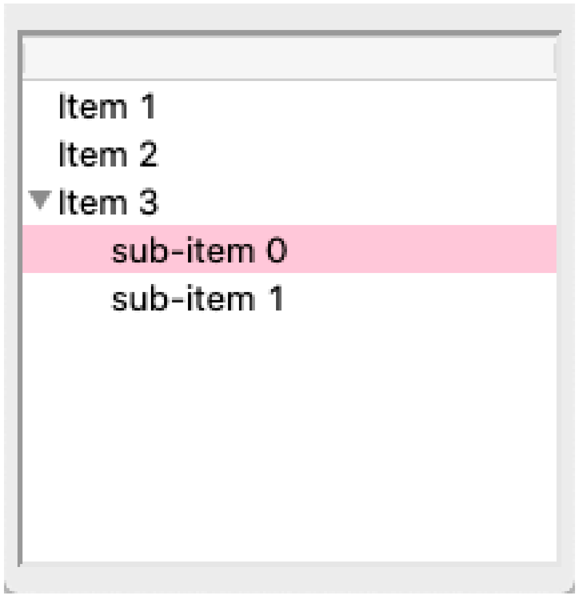
Treeview

A **treeview** widget can display a hierarchy of items. The items are organized in the form of a tree. The parent node is `''` and is not displayed. Within a node the items are indexed: 0 being the first item, `'end'` representing the position after the last item.

To insert an item to t treeview use the function:

```
tree.insert(node, index, name, text='Label')
```

The first example creates a treeview, adds an item at position 0, and another item at position `end`. The id of the third item is assigned to a local variable in order to use it as a node for creating to sub-items.



```
from tklib import *
app = App('Treeview')
```

(continues on next page)

(continued from previous page)

```
tree = ttk.Treeview(App.stack[-1])
tree.grid()
tree.insert('', 0, text='Item 1')
tree.insert('', 'end', text='Item 2')

id = tree.insert('', 5, text='Item 3')
tree.insert(id, 0, text='sub-item 0')
tree.insert(id, 1, text='sub-item 1')
app.run()
```

tree0.py

14.1 Multiple columns

The option `tree['columns']` adds more columns:

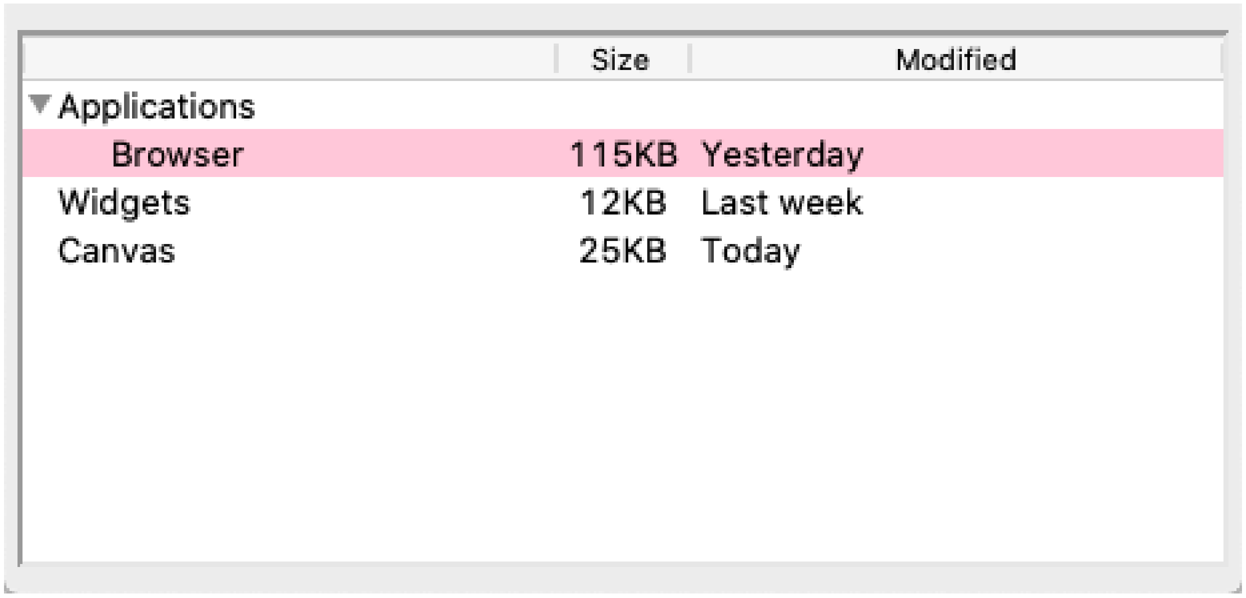
```
tree['columns'] = ('size', 'modified')
```

The width, alignment and label of a column can further be specified. To insert values to the new columns use the method `set`:

```
tree.set('widgets', 'size', '12KB')
tree.set('widgets', 'modified', 'Last week')
```

To insert all values at creation use the option `values`:

```
tree.insert('', 'end', text='Listbox', values=('15KB Yesterday'))
```



	Size	Modified
▼ Applications		
Browser	115KB	Yesterday
Widgets	12KB	Last week
Canvas	25KB	Today

```
from tklib import *
app = App('Treeview - multiple columns')

tree = ttk.Treeview(App.stack[-1])
```

(continues on next page)

(continued from previous page)

```
tree.grid()

tree.insert('', 'end', 'widgets', text='Widgets')
tree.insert('', 0, 'apps', text='Applications')

tree['columns'] = ('size', 'modified')
tree.column('size', width=50, anchor='center')
tree.heading('size', text='Size')
tree.heading('modified', text='Modified')

tree.set('widgets', 'size', '12KB')
tree.set('widgets', 'modified', 'Last week')

tree.insert('', 'end', text='Canvas', values=('25KB Today'))
tree.insert('apps', 'end', text='Browser', values=('115KB Yesterday'))

app.run()
```

tree1_col.py

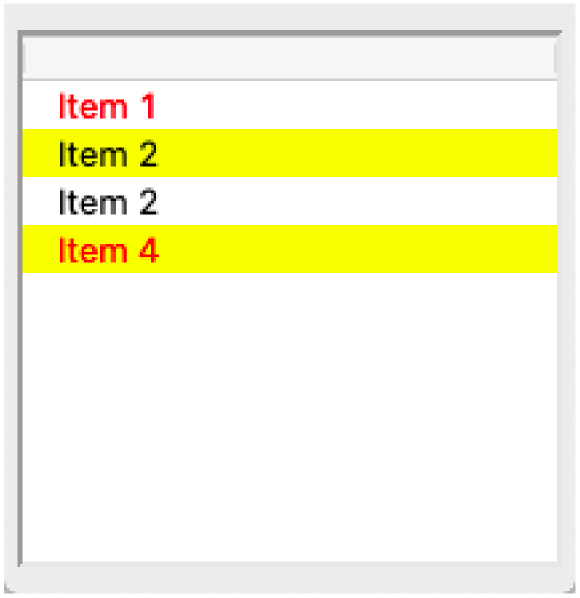
14.2 Configure style

Like the text and canvas widgets, the `Treeview` widget uses **tags** to modify the appearance of lines. Tags are simply a list of strings, such as:

```
tree.insert('', 'end', text='Item 4', tags=('fg', 'bg'))
```

Configure the tag with background and foreground color:

```
tree.tag_configure('bg', background='yellow')
tree.tag_configure('fg', foreground='red')
```



```
from tklib import *
app = App('Treeview - tags')

tree = ttk.Treeview(App.stack[-1])
tree.grid()

tree.insert('', 'end', text='Item 1', tags=('fg'))
tree.insert('', 'end', text='Item 2', tags=('bg'))
tree.insert('', 'end', text='Item 2')
tree.insert('', 'end', text='Item 4', tags=('fg', 'bg'))

tree.tag_configure('bg', background='yellow')
tree.tag_configure('fg', foreground='red')

app.run()
```

tree1_tag.py

14.3 Bind to events

Tags are also used to bind lines to events. This adds a callback to the button click:

```
tree.tag_bind('cb', '<1>', cb)
```

This adds 3 virtual events:

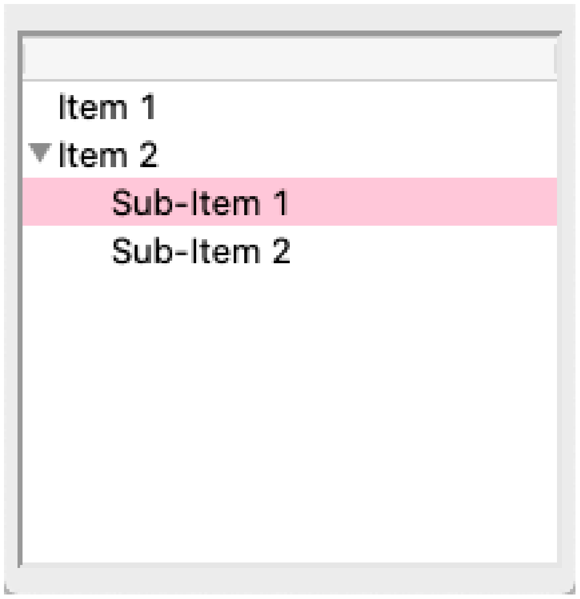
```
tree.tag_bind('cb', '<<TreeviewSelect>>', cb)
tree.tag_bind('cb', '<<TreeviewOpen>>', cb)
tree.tag_bind('cb', '<<TreeviewClose>>', cb)
```

In the callback function cb we print the event, the selection and the focus:

```
def cb(event):
    print(event, tree.selection(), tree.focus())
```

Something like this will be printed to the console:

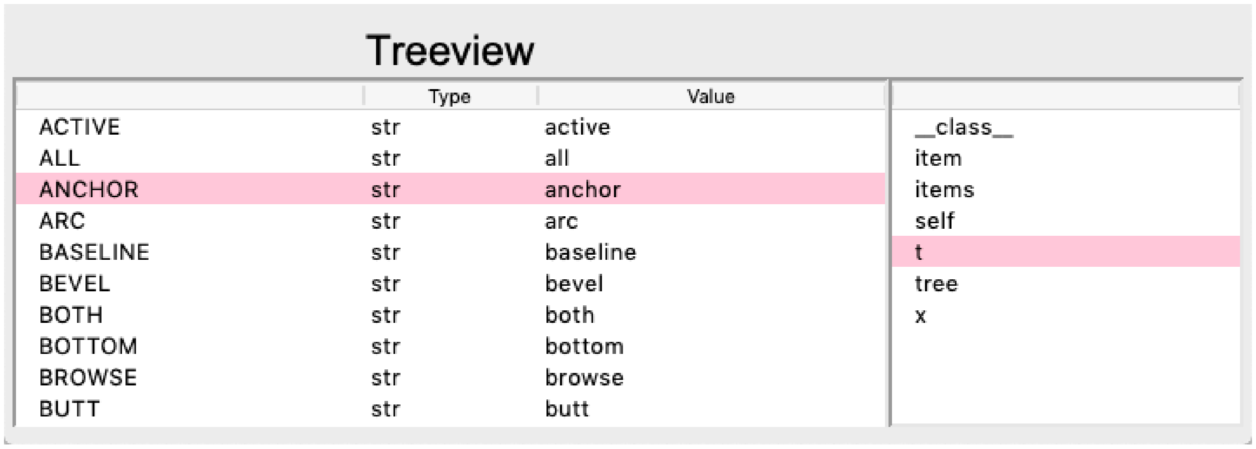
```
<ButtonPress event num=1 x=8 y=46> ('I002',) I002
<VirtualEvent event x=0 y=0> ('I002',) I002
<VirtualEvent event x=0 y=0> ('I002',) I002
```



```python
from tklib import *
app = App('Treeview - bind')

def cb(event):
    print(event, tree.selection(), tree.focus())

tree = ttk.Treeview(App.stack[-1])
tree.grid()

tree.insert('', 'end', text='Item 1', tags=('cb'))
id = tree.insert('', 'end', text='Item 2', tags=('cb'))
tree.insert(id, 'end', text='Sub-Item 1', tags=('cb'))
tree.insert(id, 'end', text='Sub-Item 2', tags=('cb'))

tree.tag_bind('cb', '<1>', cb)
tree.tag_bind('cb', '<<TreeviewSelect>>', cb)
tree.tag_bind('cb', '<<TreeviewOpen>>', cb)
tree.tag_bind('cb', '<<TreeviewClose>>', cb)

app.run()
```

tree1_bind.py

14.4 Two treeviews next to each other



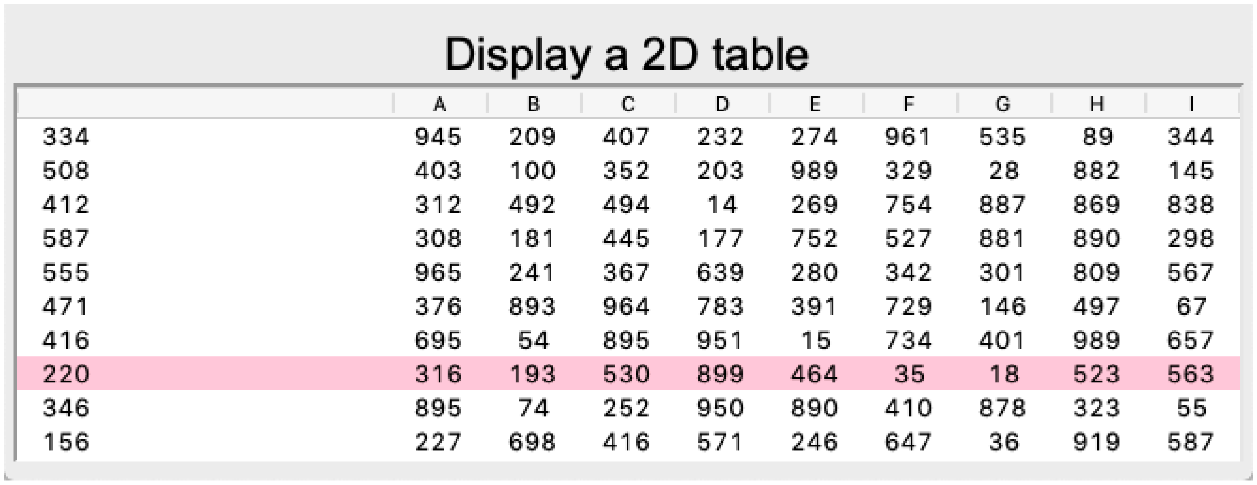
	Type	Value
ACTIVE	str	active
ALL	str	all
ANCHOR	str	anchor
ARC	str	arc
BASELINE	str	baseline
BEVEL	str	bevel
BOTH	str	both
BOTTOM	str	bottom
BROWSE	str	browse
BUTT	str	butt

__class__
item
items
self
t
tree
x

```
from tklib import *

app = App('Treeview')

items = dir(tk)
tree = Treeview()
tree['columns'] = ('type', 'value')
tree.column('type', width=100)
tree.heading('type', text='Type')
tree.heading('value', text='Value')

for item in items:
    x = eval('tk.'+item)
    t = type(x)
    print(t.__name__, x)
    tree.insert('', 'end', text=item, values=(t.__name__, x))

items = dir()
Treeview(items).grid(row=0, column=1)

app.run()
```

tree2.py

14.5 Display a 2D table



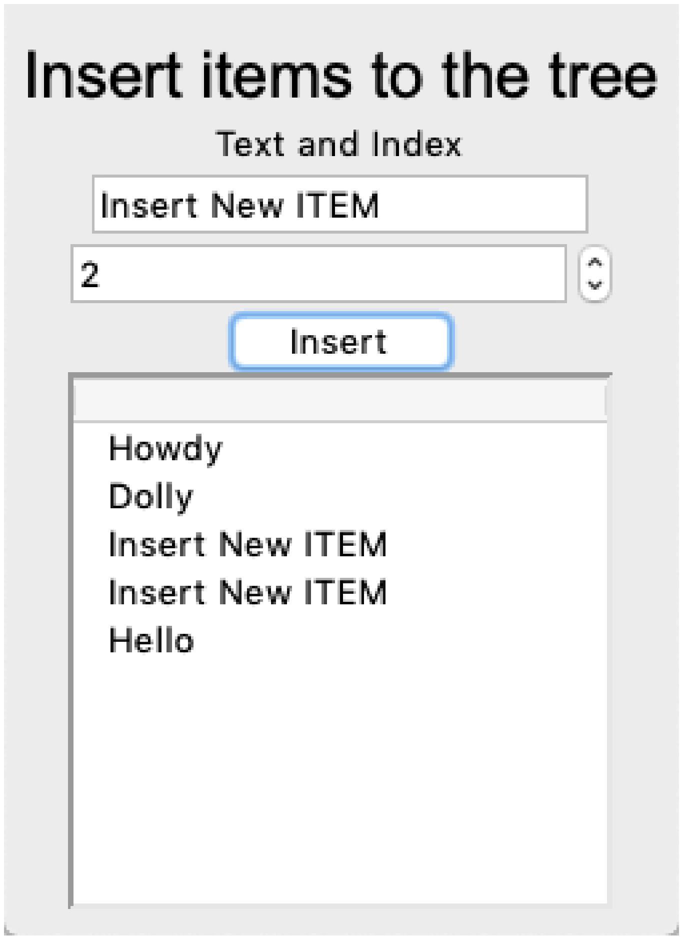
Display a 2D table

	A	B	C	D	E	F	G	H	I
334	945	209	407	232	274	961	535	89	344
508	403	100	352	203	989	329	28	882	145
412	312	492	494	14	269	754	887	869	838
587	308	181	445	177	752	527	881	890	298
555	965	241	367	639	280	342	301	809	567
471	376	893	964	783	391	729	146	497	67
416	695	54	895	951	15	734	401	989	657
220	316	193	530	899	464	35	18	523	563
346	895	74	252	950	890	410	878	323	55
156	227	698	416	571	246	647	36	919	587

```python
"""Display a 2D table."""

from tklib import *
import random

n, m = 40, 10
table = []
for i in range(n):
    line = []
    for j in range(m):
        line.append(random.randint(0, 999))
    table.append(line)


class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Display a 2D table', font='Arial 24')
        Label('Click on header to sort')
        Combobox('A;B;C')

        tree = Treeview()
        for i in range(n):
            tree.insert('', 'end', text=table[i][0], values=table[i][1:])

        tree['columns'] = list(range(m-1))
        headings=list('ABCDEFGHI')
        for j in range(m-1):
            tree.column(j, width=50, anchor='e')
            tree.heading(j, text=headings[j])

if __name__ == '__main__':
    Demo().run()
```

tree4.py

14.6 Insert items into a treeview



```
"""Insert items to the tree."""

from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Insert items to the tree', font='Arial 24')

        App.text = Entry('Text', 'print(self.val.get())')
        App.index = Spinbox('Index', 'print(self.val.get())', to=100)

        Button('Insert', 'App.tree.insert("", App.index.val.get(), text=App.text.
↪get())')
        Button('Insert to Folder', 'App.tree.insert("folder", App.index.val.get(),␣
↪text=App.text.get())')

        App.tree = Treeview()

        L = 'Hello;Dolly;How;Are;You'.split(';')
        for item in L:
            App.tree.insert('', 'end', text=item)
```

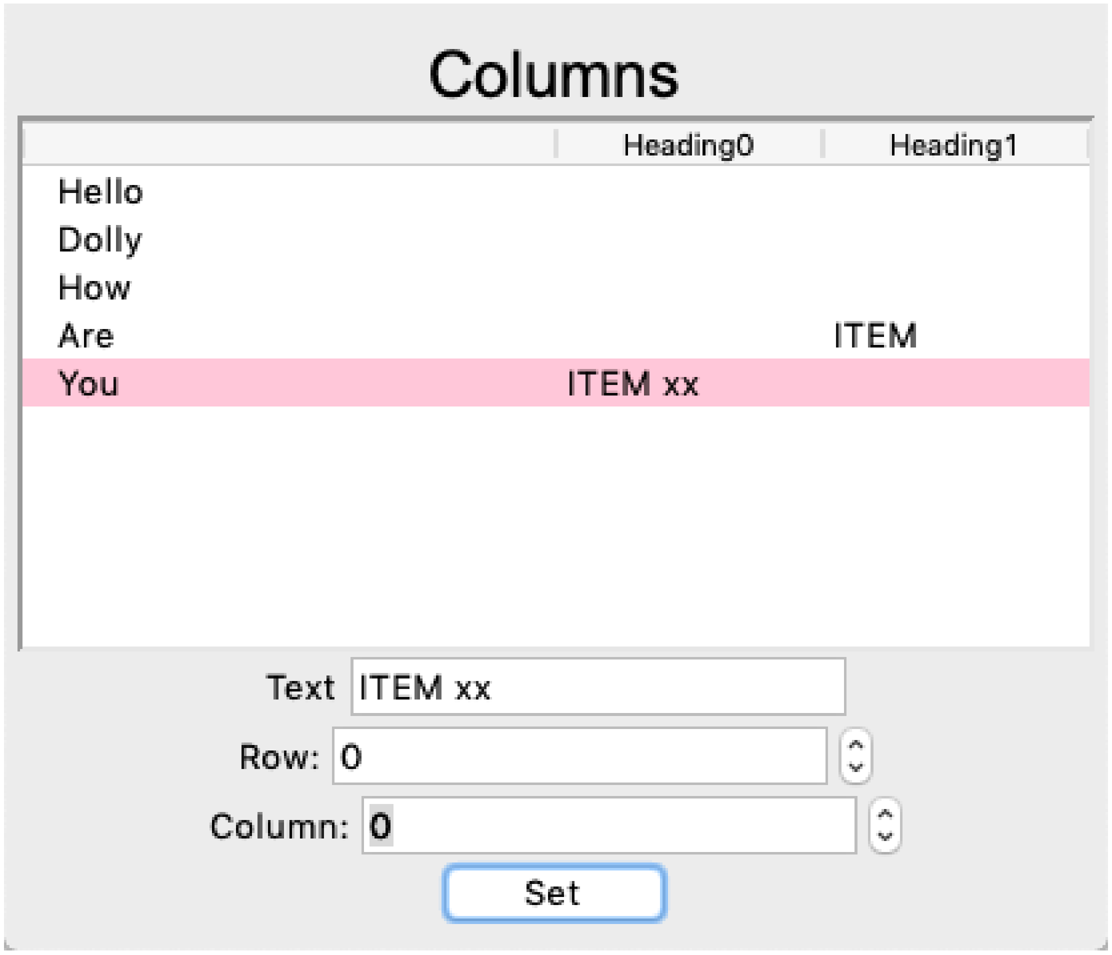
(continues on next page)

(continued from previous page)

```
        App.tree.insert('', 0, 'folder', text='Folder')

if __name__ == '__main__':
    Demo().run()
```

tree5.py

14.7 Add items to specific column



```
"""Insert multiple columns to the tree."""

from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Columns', font='Arial 24')

        App.tree = Treeview()
        # App.tree['columns'] = ('size', 'date', 'type')
        App.tree['columns'] = range(3)
```

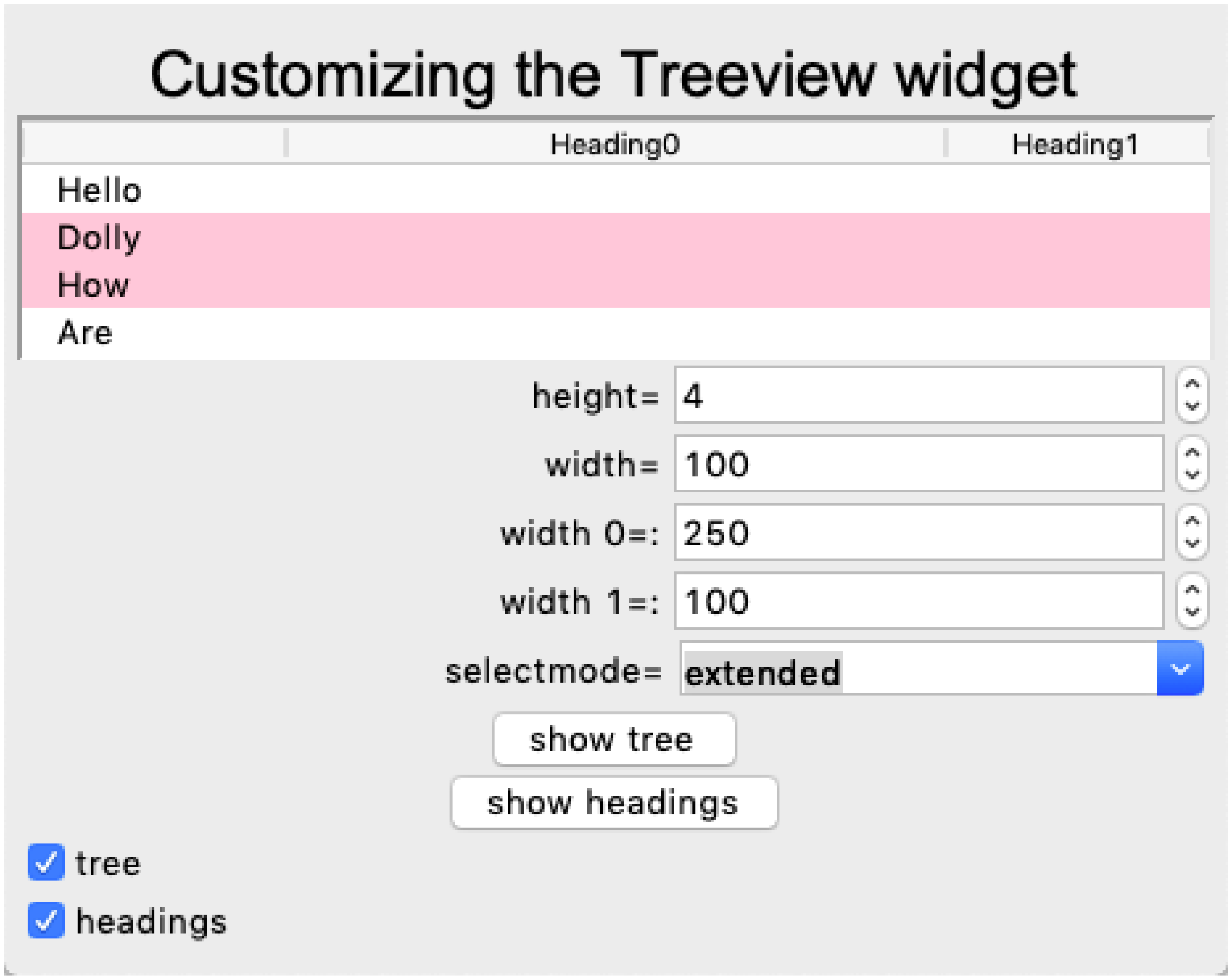
(continues on next page)

(continued from previous page)

```
        App.text = Entry('Text', 'print(self.val.get())')
        App.row = Spinbox('Row:', 'print(self.val.get())', to=100)
        App.col = Spinbox('Column:',  'print(self.val.get())', to=100)
        Button('Set', 'App.tree.set(App.tree.focus(), App.col.val.get(), App.text.
↪get())')

        for i in range(2):
            App.tree.column(i, width=100, anchor='w')
            App.tree.heading(i, text='Heading' + str(i))

        L = 'Hello;Dolly;How;Are;You'.split(';')
        for item in L:
            App.tree.insert('', 'end', text=item)

if __name__ == '__main__':
    Demo().run()
```

tree6.py

14.8 Customize the treeview widget



```
"""Custumize the Treeview widget."""

from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()
        Label('Customizing the Treeview widget', font='Arial 24')

        App.tree = Treeview()
        App.tree['columns'] = range(3)

        Spinbox('height=', 'App.tree["height"]=self.var.get()', to=20)
        Spinbox('width=', 'App.tree.column("#0", width=self.var.get())', inc=50,␣
↪to=500)
        Spinbox('width 0=:', 'App.tree.column(0, width=self.var.get())', inc=50,␣
↪to=500)
        Spinbox('width 1=:', 'App.tree.column(1, width=self.var.get())', inc=50,␣
↪to=500)
        Combobox('selectmode=', 'browse;extended;none', 'App.tree["selectmode"]=self.
↪var.get()')
```

(continues on next page)

(continued from previous page)

```
        Button('show tree', "App.tree['show']='tree'")
        Button('show headings', "App.tree['show']='headings'")
        Checkbutton('tree;headings')

        for i in range(2):
            App.tree.column(i, width=100, anchor='w')
            App.tree.heading(i, text='Heading' + str(i))

        L = 'Hello;Dolly;How;Are;You'.split(';')
        for item in L:
            App.tree.insert('', 'end', text=item)

if __name__ == '__main__':
    Demo().run()
```

tree7.py

CHAPTER 15

Windows

In this section we present how to create new windows and how to open dialog windows.

To create a new window we define a new `Window` class which instantiates the `Toplevel` class:

```
class Window():
    """Create a new window."""
    def __init__(self, title='Window'):
        top = tk.Toplevel(App.root)
        top.title(title)
        frame = ttk.Frame(top, width=300, height=200, padding=(5, 10))
        frame.grid()
        App.stack.append(frame)
```

First we create a toplevel window and add a title to it:

```
top = tk.Toplevel(App.root)
top.title(title)
```

Then we add a themed frame widget in order to get the theme's background color, add the geometry manager (grid) and place the frame on the widget stack, so new widgets are added to the new window:

```
frame = ttk.Frame(top, width=300, height=200, padding=(5, 10))
frame.pack()
App.stack.append(frame)
```

This is the constructor method.

```
    def __init__(self, title='Window', top=None):
        if top == None:
            top = tk.Toplevel(App.root)
        top.title(title)
        top.columnconfigure(0, weight=1)
        top.rowconfigure(0, weight=1)
        top.bind('<Command-i>', self.inspector)
```

(continues on next page)

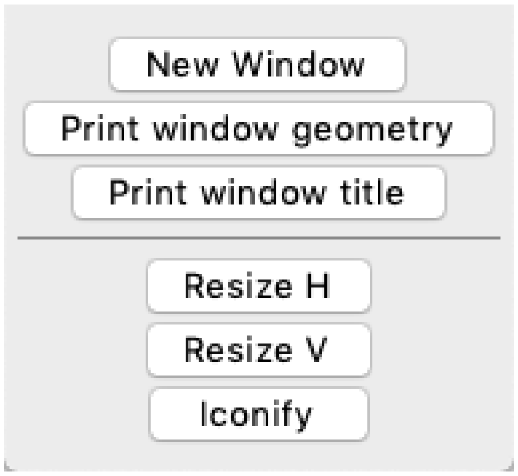
(continued from previous page)

```
        top.bind('<Command-p>', self.save_img)
        self.top = top

        frame = ttk.Frame(top, width=300, height=200, padding=(5, 10))
        frame.grid(sticky='nswe')

        App.stack.append(frame)
        App.win = top
        App.menus = [tk.Menu(App.win)]
        App.win['menu'] = App.menus[0]
```

15.1 Create new windows

The following example adds two more windows besides the root window. Each one can be closed individually, but only if the root window is closed, the application ends:

```
Window('Text')
Text(height=10, width=40)

Window('Canvas')
Canvas()
```



```
"""Open multiple windows."""
from tklib import *

class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        App.root.title('Windows and dialogs')

        Button('New Window', 'Window()')
        Button('Print window geometry', 'print(App.root.geometry())')
        Button('Print window title', 'print(App.root.title())')
        Separator()

        Button('Resize H', 'App.root.resizable(True, False)')
        Button('Resize V', 'App.root.resizable(False, True)')
```

(continues on next page)

(continued from previous page)

```
        Button('Iconify', 'App.root.iconify()')

        Window('Text')
        Text(height=10, width=40)

        Window('Canvas')
        Canvas()

Demo().run()
```

`window1.py`

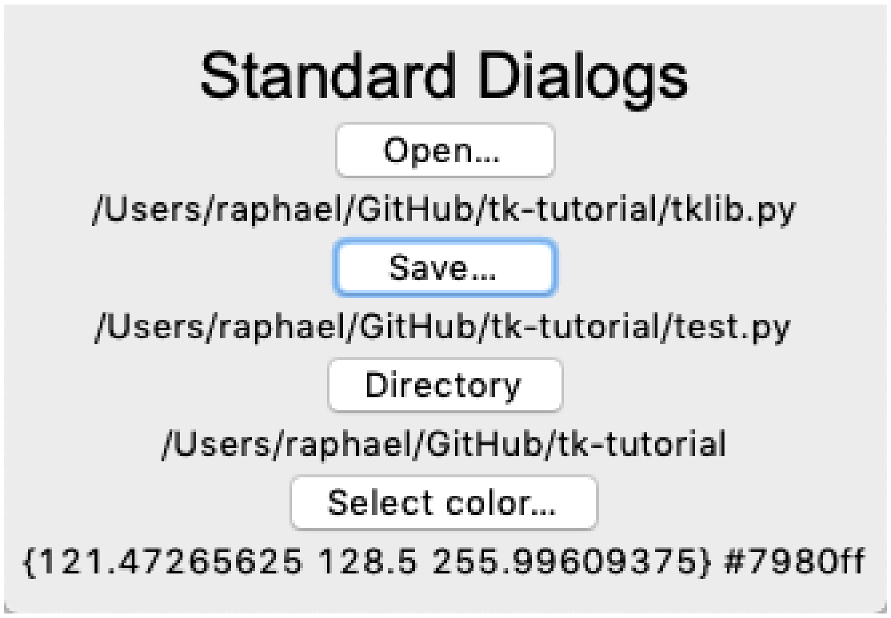
The buttons have the following functions:

- **New Window** creates a new window
- **Print window geometry** prints the window placement (191x175+47+51)
- **Print window title** prints the window title (Window and dialogs)
- **Resize H** allows for only horizontal resizing
- **Resize V** allows for only vertical resizing
- **Iconify** iconifies the window

15.2 Standard dialogs

Tk provides multiple standard dialogs for

- asking the open file name
- asking the save file name
- asking a directory name
- finding a color



Standard Dialogs

/Users/raphael/GitHub/tk-tutorial/tklib.py

/Users/raphael/GitHub/tk-tutorial/test.py

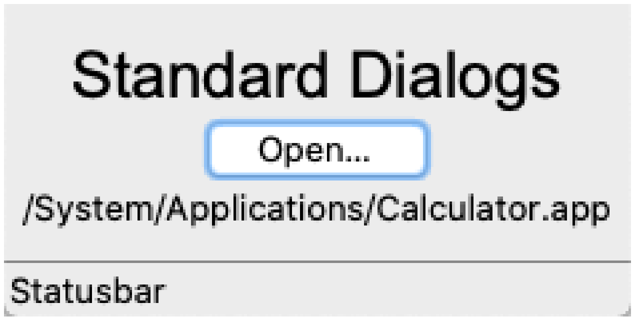
/Users/raphael/GitHub/tk-tutorial

{121.47265625 128.5 255.99609375} #7980ff

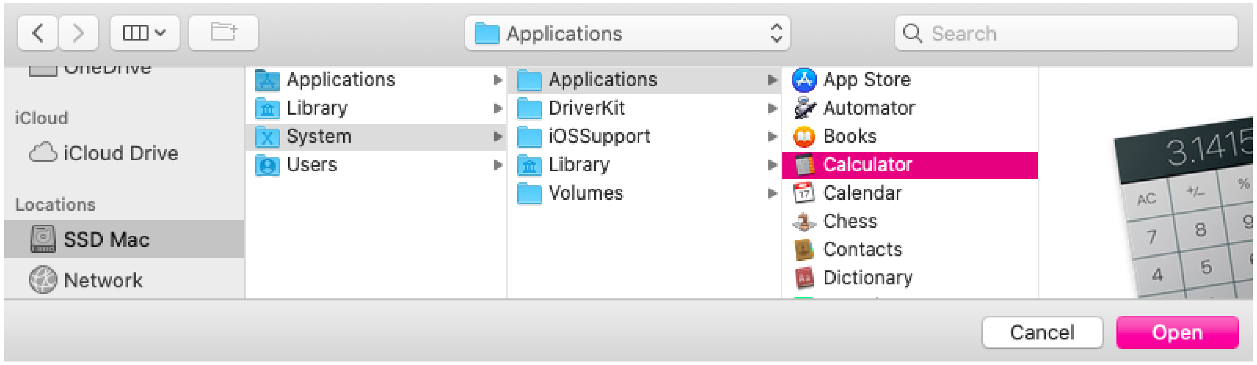
```python
"""Standard dialogs."""
from tklib import *
from tkinter import filedialog, colorchooser

class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        App.root.title('Windows and dialogs')
        Label('Standard Dialogs', font='Arial 24')

        Button('Open...', 'App.open["text"] = tk.filedialog.askopenfilename()')
        App.open = Label('File')

        Button('Save...', 'App.save["text"] = tk.filedialog.asksaveasfilename()')
        App.save = Label('File')

        Button('Directory...', 'App.dir["text"] = tk.filedialog.askdirectory()')
        App.dir = Label('Directory')

        Button('Select color...', 'App.col["text"] = tk.colorchooser.askcolor()')
        App.col = Label('Color')

Demo().run()
```

window2.py

15.3 Open dialog



Pressing the **Open…** button opens a standard *open file* dialog and returns a path or an empty string (if cancelled).



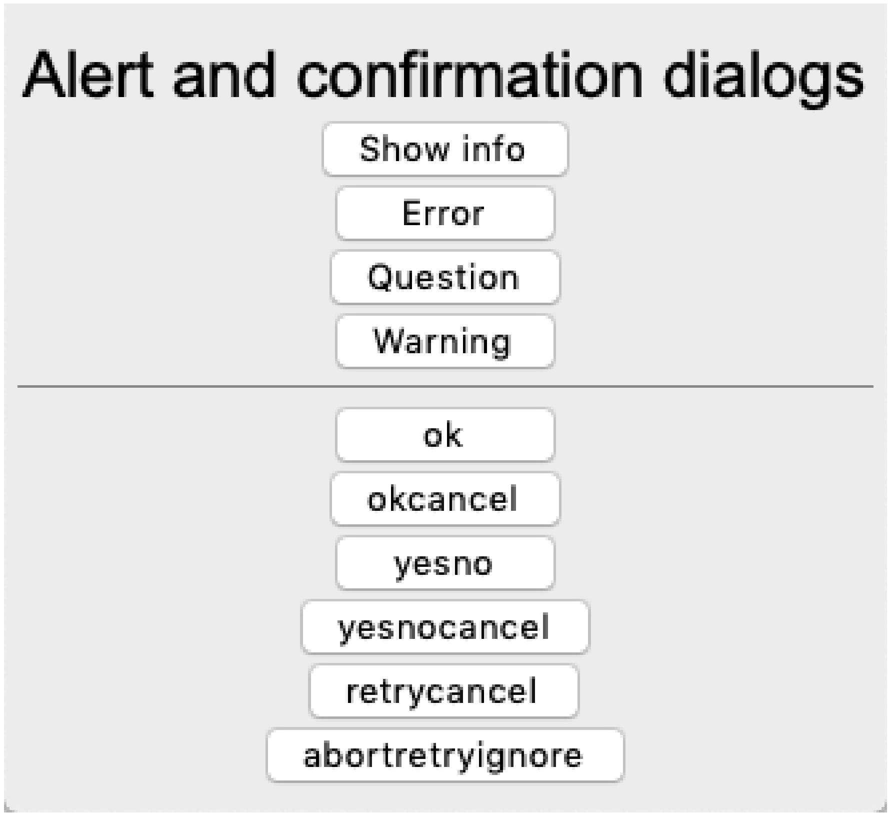
```
"""Standard dialogs."""
from tklib import *

app = App('Standard dialogs')
Label('Standard Dialogs', font='Arial 24')
Button('Open...', 'App.open["text"] = filedialog.askopenfilename()')
App.open = Label('File')

app.run()
```

dialog1.py

15.4 Alert and confirmation dialogs

Tk provides multiple standard dialogs for showing

- info
- error
- question
- warning



```
"""Alert and confirmation dialogs."""
from tklib import *
from tkinter import filedialog, messagebox
```

(continues on next page)

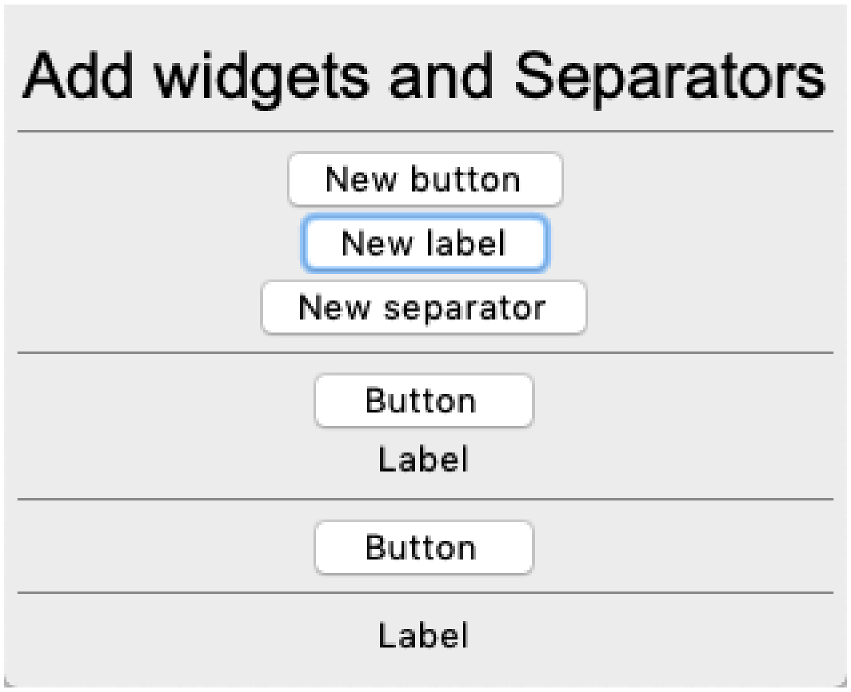
(continued from previous page)

```
class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Alert and confirmation dialogs', font='Arial 24')

        Button('Show info', 'tk.messagebox.showinfo(message="Hello world")')
        Button('Error', 'tk.messagebox.showinfo(message="Error", icon="error")')
        Button('Question', 'tk.messagebox.showinfo(message="Question", icon="question
↪")')
        Button('Warning', 'tk.messagebox.showinfo(message="Warning", icon="warning")')
        Separator()

        types = ('ok', 'okcancel', 'yesno', 'yesnocancel', 'retrycancel',
↪'abortretryignore')
        for t in types:
            Button(t, 'tk.messagebox.showinfo(message="{}", type="{}")'.format(t, t))

Demo().run()
```

window3.py

15.5 Add widgets and separators

The following demo program shows how to insert buttons, labels and separators.



```
"""Separators."""
from tklib import *

class Demo(App):
    def __init__(self, **kwargs):
```

(continues on next page)

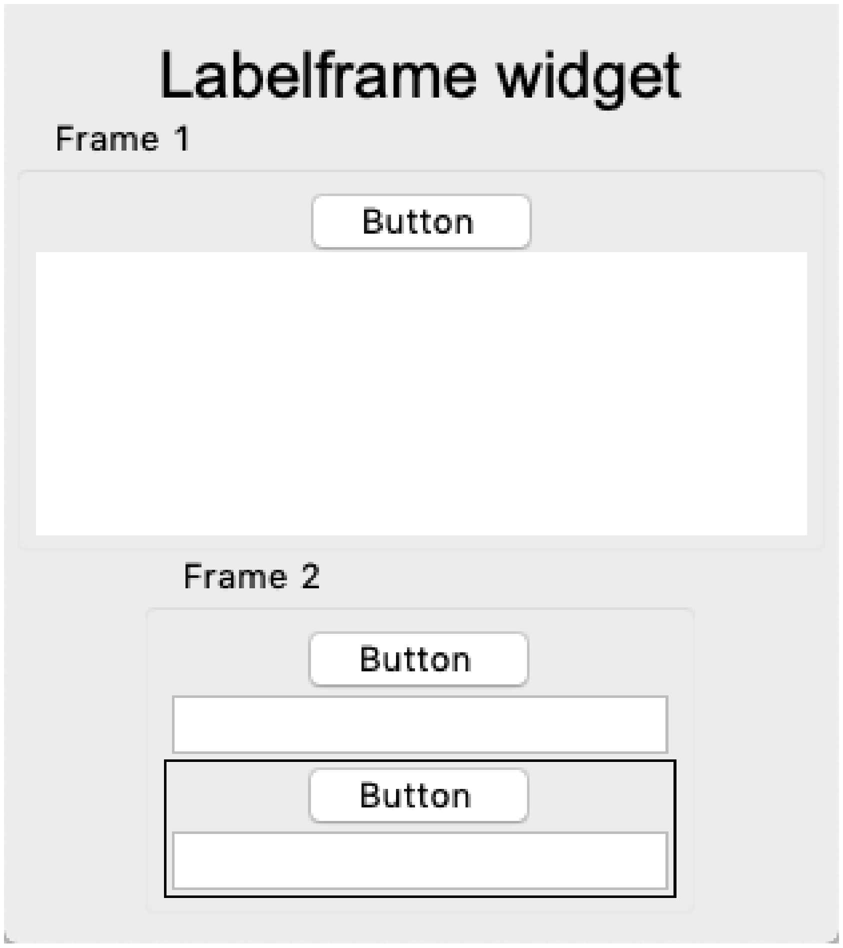
(continued from previous page)

```
        super().__init__(**kwargs)
        Label('Add widgets and Separators', font='Arial 24')
        Separator()

        Button('New button', 'Button()')
        Button('New label', 'Label()')
        Button('New separator', 'Separator()')
        Separator()

Demo().run()
```

window4.py

15.6 The Labelframe widget

The **Labelframe** widget is a frame used for grouping widgets, but has a label attached to it



```
"""Add Labelframes."""
from tklib import *

class Demo(App):
```

(continues on next page)

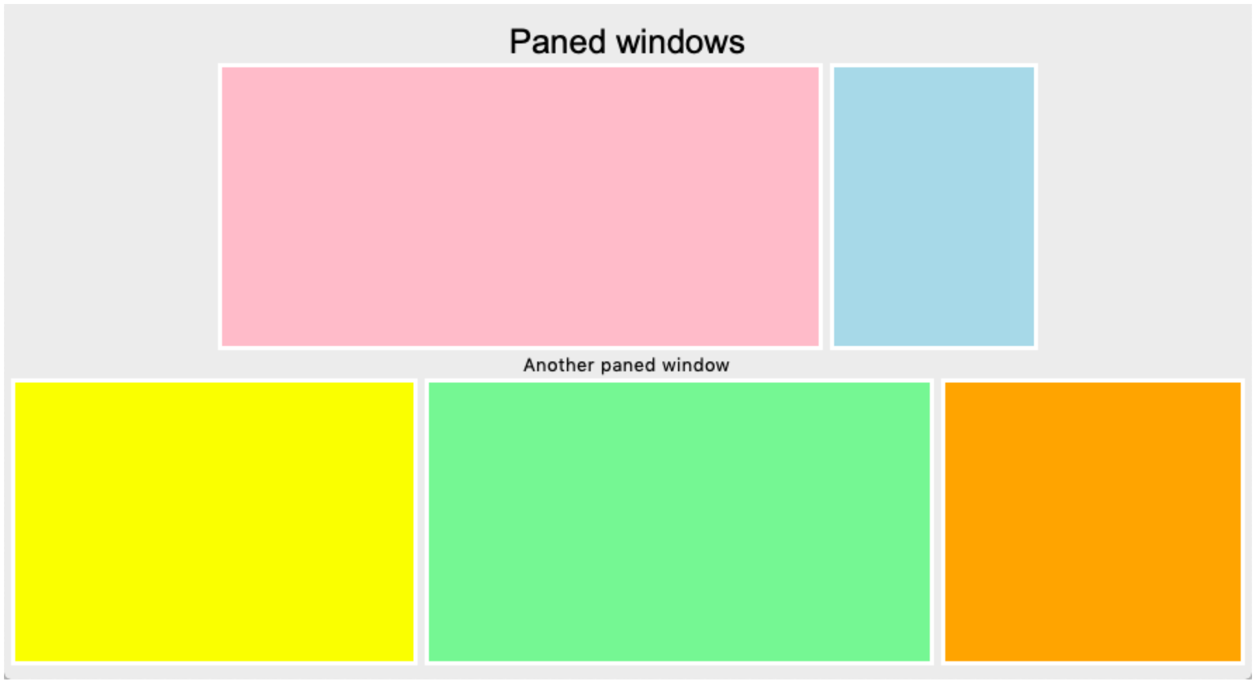
(continued from previous page)

```
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Labelframe widget', font='Arial 24')

        Labelframe(text='Frame 1', padding=5)
        Button()
        Canvas(height=100)

        App.stack.pop()

        Labelframe(text='Frame 2', padding=5)
        Button()
        Entry()

        Frame()
        Button()
        Entry()

Demo().run()
```

window5.py

15.7 Paned windows

The **Panedwindow** widget creates a slider and allows to change the width or hight between two or more widgets.



```
"""Paned windows."""
from tklib import *

class Demo(App):
```

(continues on next page)

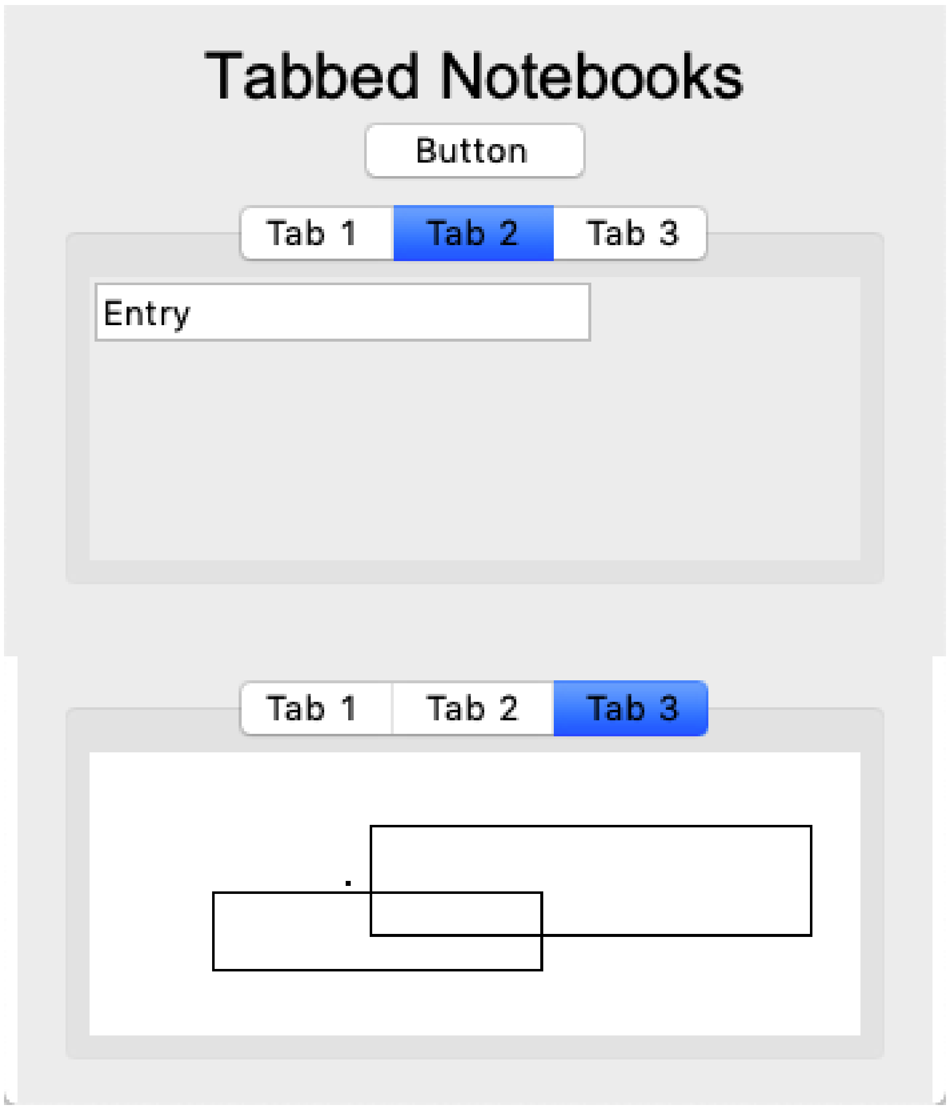
(continued from previous page)

```
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Paned windows', font='Arial 24')

        p = Panedwindow(orient='horizontal')
        print(p)
        f1 = Canvas(background='pink')
        f2 = Canvas(background='lightblue')
        p.add(f1)
        p.add(f2)

        App.stack.pop()
        Label('Another paned window')

        p2 = Panedwindow(orient='horizontal')
        print(p2)
        p2.add(Canvas(background='yellow'))
        p2.add(Canvas(background='lightgreen'))
        p2.add(Canvas(background='orange'))

Demo().run()
```

`window6.py`

15.8 Tabbed notebooks

The **Notebook** widget creates a section with tabbed frames.



```python
"""Tabbed notebooks."""
from tklib import *

class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Tabbed Notebooks', font='Arial 24')
        Button()

        Notebook()
        Frame(nb='Tab 1')
        Button()
        Frame(nb='Tab 2')
        Entry()
        Frame(nb='Tab 3')
        Canvas(height=100)
        App.stack.pop()

        Notebook()
```

(continues on next page)

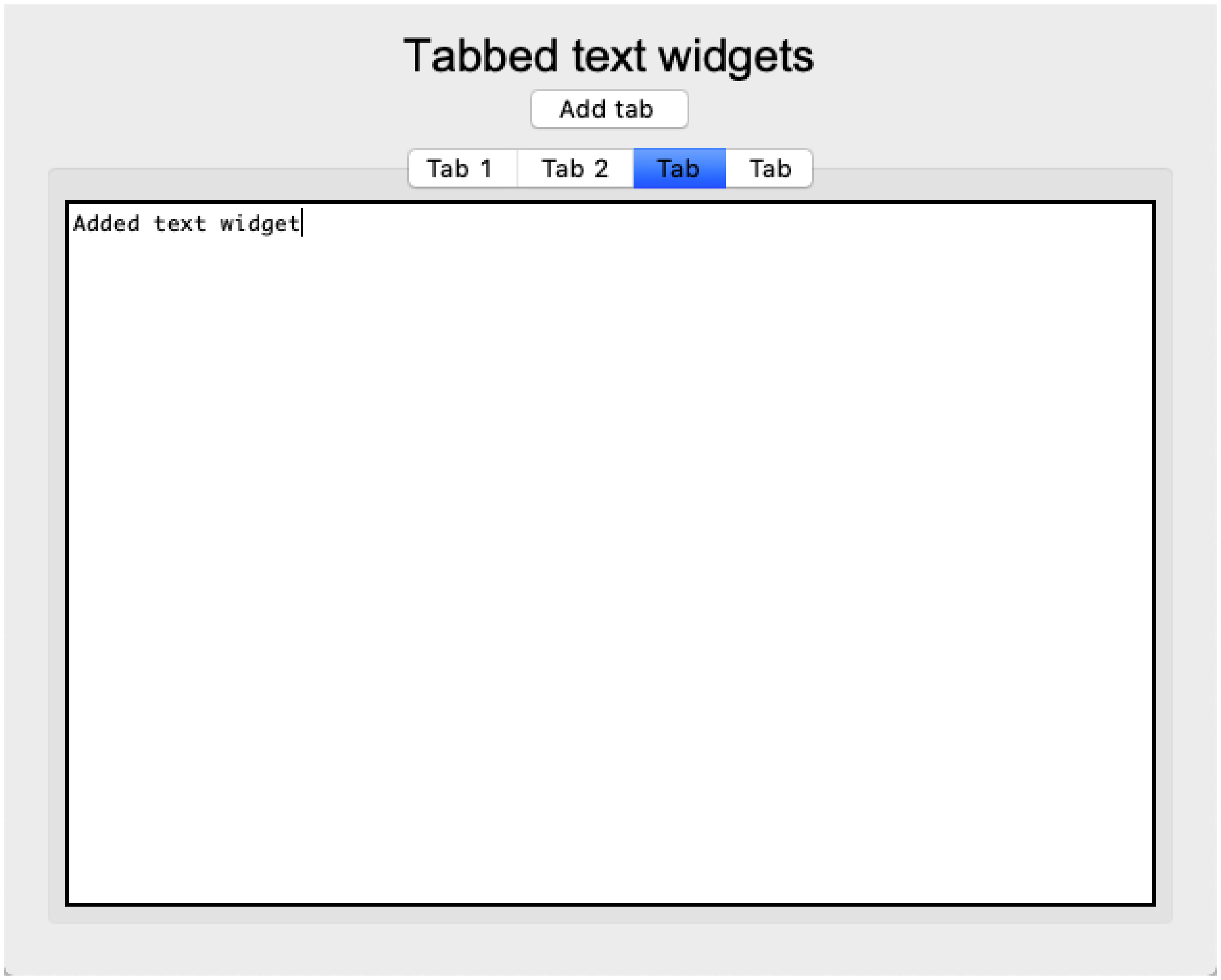
(continued from previous page)

```
        Frame(nb='Tab 1')
        Button()
        Button()
        Frame(nb='Tab 2')
        Entry()
        Frame(nb='Tab 3')
        Canvas(height=100)

Demo().run()
```

window7.py

15.9 Add more tabs

In the following example we add more tabs by clicking on a button.



Added text widget

```
"""Tabbed Text widgets."""
from tklib import *
```

(continues on next page)

(continued from previous page)

```
class Demo(App):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        Label('Tabbed text widgets', font='Arial 24')

        Button('Add tab', 'Frame(nb="Tab");Text()')
        Notebook()
        Frame(nb='Tab 1')
        Text()
        Frame(nb='Tab 2')
        Text()

Demo().run()
```

`window8.py`

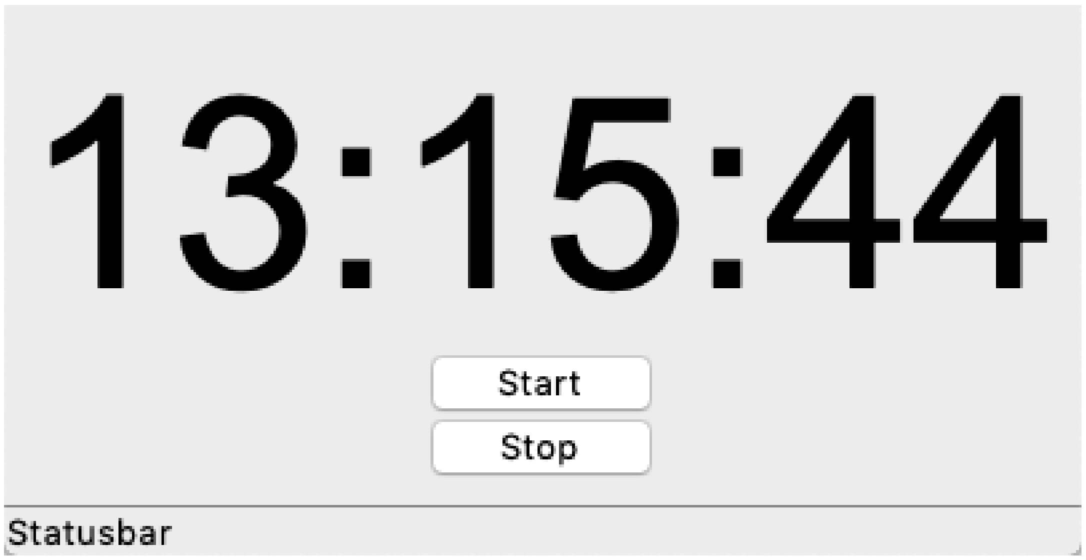
CHAPTER 16

Time

In this section we are going to deal with time. The goal is to build four simple applications

- clock
- timer
- alarm
- chronometer

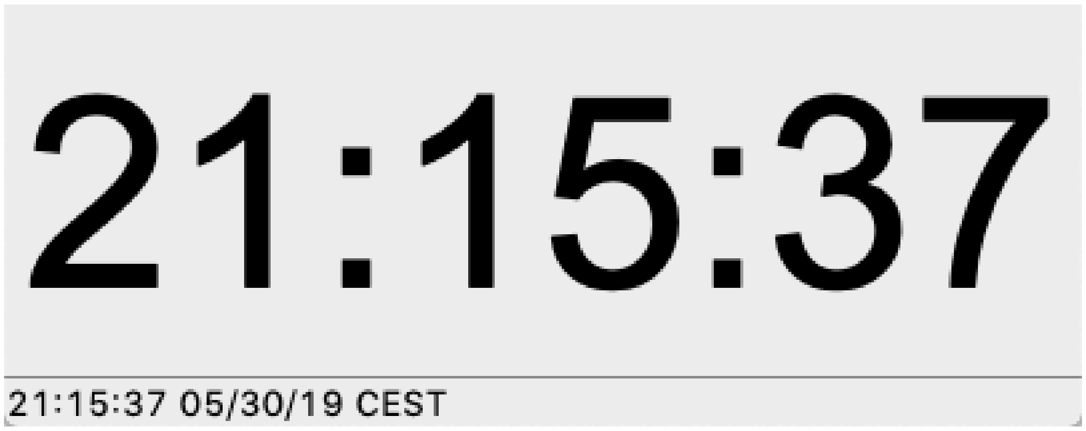
16.1 Clock



```
"""Display a clock ."""
from tklib import *
import time

class Clock(Window):
    def __init__(self):
        super(Clock, self).__init__('Clock')
        self.lb = Label('hh:mm:ss', font='Arial 100')
        top = self.lb.winfo_toplevel()
        top.resizable(width=False, height=False)
        self.cb()

    def cb(self, event=None):
        """Display the time every second."""
        t = time.strftime('%X')
        self.lb['text'] = t

        d = time.strftime('%X %x %Z')
        self.status['text'] = d
        self.lb.after(1000, self.cb)

class Demo(App):
    def __init__(self):
        super().__init__()
        Button('New Clock', Clock)
        Clock()

if __name__ == '__main__':
    Demo().run()
```

time1.py

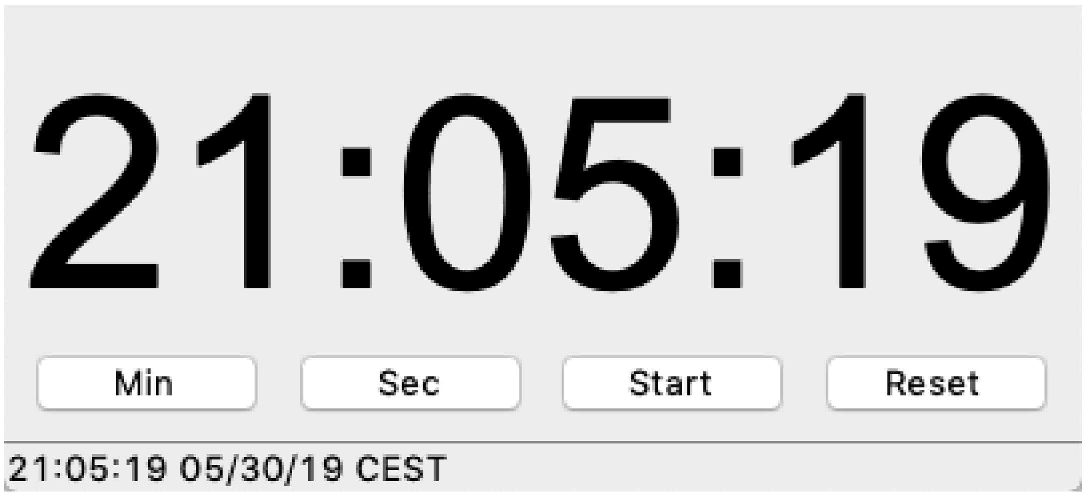
16.2 Timer



```
21:15:37
```

```
21:15:37 05/30/19 CEST
```

Or a different layout.



21:05:19 05/30/19 CEST

```python
"""Display a timer ."""
from tklib import *
import time

class Clock(Window):
    def __init__(self):
        super(Clock, self).__init__('Clock')
        self.lb = Label('hh:mm:ss', font='Arial 100')
        top = self.lb.winfo_toplevel()
        top.resizable(width=False, height=False)
        self.cb()

    def cb(self, event=None):
        """Display the time every second."""
        t = time.strftime('%X')
        self.lb['text'] = t

        d = time.strftime('%X %x %Z')
        self.status['text'] = d
        self.lb.after(1000, self.cb)

class Timer(Window):
    def __init__(self):
        super(Timer, self).__init__('Timer')
        self.lb = Label('hh:mm:ss', font='Arial 100')
        self.lb.grid(columnspan=4)
        Button('Min').grid(row=1, column=0)
        Button('Sec').grid(row=1, column=1)
        Button('Start').grid(row=1, column=2)
        Button('Reset').grid(row=1, column=3)
        self.cb()

    def cb(self, event=None):
        """Display the time every second."""
        t = time.strftime('%X')
        self.lb['text'] = t

        d = time.strftime('%X %x %Z')
        self.status['text'] = d
        self.lb.after(1000, self.cb)
```

(continues on next page)

(continued from previous page)

```python

class Demo(App):
    def __init__(self):
        super().__init__()
        Button('New Clock', Clock)
        Button('New Timer', Timer)

if __name__ == '__main__':
    Demo().run()
```

time2.py

CHAPTER 17

Applications

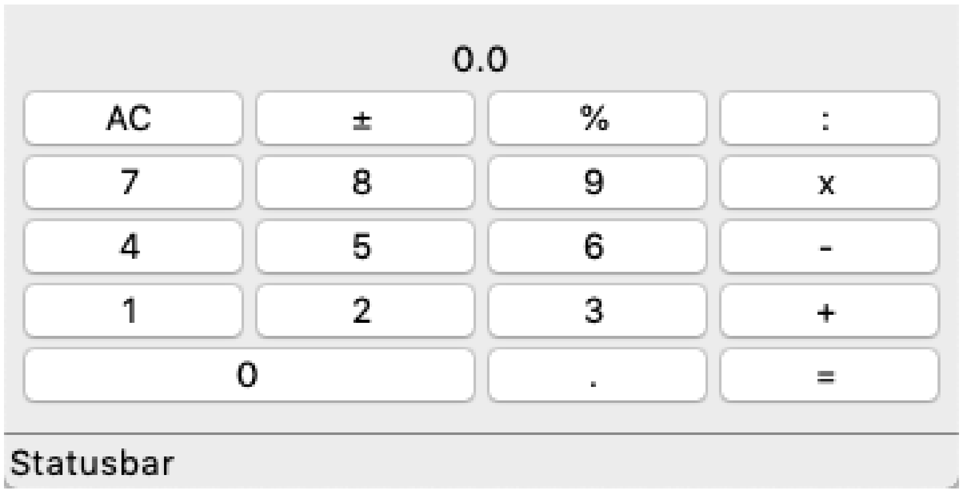
In this section we are showing some practical applications.

17.1 Calculator

The first example is a pocket calculator.

First, we just do the layout for the buttons:

```
App.lb = Label('0.0')
App.lb.grid(columnspan=4)

Button('AC', ).grid(row=1)
Button('±').grid(row=1, column=1)
Button('%').grid(row=1, column=2)
Button(':').grid(row=1, column=3)
```

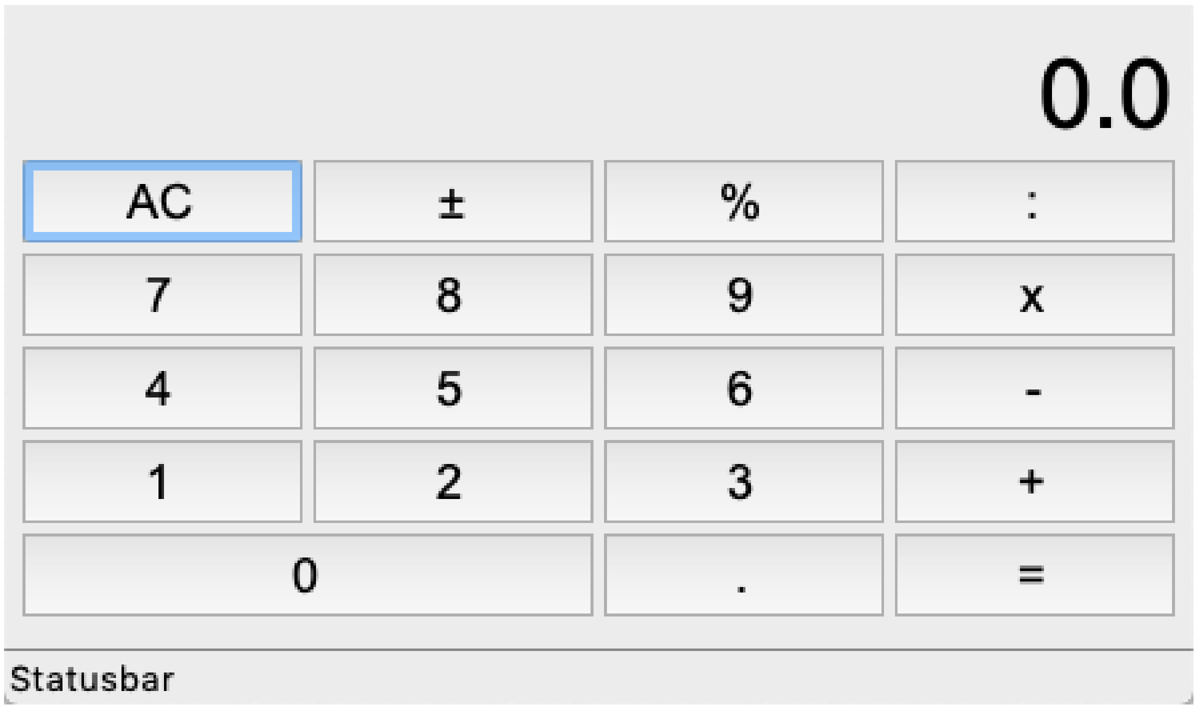


```
"""Create calculator buttons."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()

        App.lb = Label('0.0')
        App.lb.grid(columnspan=4)

        Button('AC', ).grid(row=1)
        Button('±').grid(row=1, column=1)
        Button('%').grid(row=1, column=2)
        Button(':').grid(row=1, column=3)

        Button('7').grid(row=2, column=0)
        Button('8').grid(row=2, column=1)
        Button('9').grid(row=2, column=2)
        Button('x').grid(row=2, column=3)

        Button('4').grid(row=3, column=0)
        Button('5').grid(row=3, column=1)
        Button('6').grid(row=3, column=2)
        Button('-').grid(row=3, column=3)

        Button('1').grid(row=4, column=0)
        Button('2').grid(row=4, column=1)
        Button('3').grid(row=4, column=2)
        Button('+').grid(row=4, column=3)

        Button('0').grid(row=5, columnspan=2, sticky='we')
        Button('.').grid(row=5, column=2)
        Button('=').grid(row=5, column=3)

if __name__ == '__main__':
    Demo().run()
```

calc1.py

Then we are going to add callback functions to the keys:

```
Button('7', 'App.lb["text"]  = float(App.lb["text"])*10 + 7').grid(row=2, column=0)
Button('8', 'App.lb["text"]  = float(App.lb["text"])*10 + 8').grid(row=2, column=1)
Button('9', 'App.lb["text"]  = float(App.lb["text"])*10 + 9').grid(row=2, column=2)
Button('x').grid(row=2, column=3)
```



```
"""Create calculator buttons."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()

        s = ttk.Style()
        s.configure('TButton', font='Arial 18', padding=5)

        App.lb = Label('0', font='Arial 36')
        App.lb.grid(columnspan=4, sticky='e')

        Button('C', 'App.lb["text"]  = 0.0').grid(row=1)
        Button('±').grid(row=1, column=1)
        Button('%').grid(row=1, column=2)
        Button(':').grid(row=1, column=3)

        Button('7', 'App.lb["text"]  = float(App.lb["text"])*10 + 7').grid(row=2,␣
↪column=0)
        Button('8', 'App.lb["text"]  = float(App.lb["text"])*10 + 8').grid(row=2,␣
↪column=1)
        Button('9', 'App.lb["text"]  = float(App.lb["text"])*10 + 9').grid(row=2,␣
↪column=2)
        Button('x').grid(row=2, column=3)

        Button('4', 'App.lb["text"]  = float(App.lb["text"])*10 + 4').grid(row=3,␣
↪column=0)
        Button('5', 'App.lb["text"]  = float(App.lb["text"])*10 + 5').grid(row=3,␣
↪column=1)
        Button('6', 'App.lb["text"]  = float(App.lb["text"])*10 + 6').grid(row=3,␣
↪column=2)
        Button('-').grid(row=3, column=3)
```

(continues on next page)

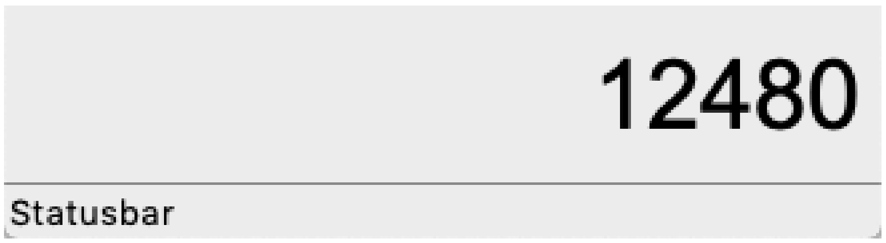
(continued from previous page)

```
        Button('1', 'App.lb["text"]  = float(App.lb["text"])*10 + 1').grid(row=4,␣
↪column=0)
        Button('2', 'App.lb["text"]  = float(App.lb["text"])*10 + 2').grid(row=4,␣
↪column=1)
        Button('3', 'App.lb["text"]  = float(App.lb["text"])*10 + 3').grid(row=4,␣
↪column=2)
        Button('+').grid(row=4, column=3)

        Button('0', 'App.lb["text"]  = float(App.lb["text"])*10 + 0').grid(row=5,␣
↪columnspan=2, sticky='we')
        Button('.').grid(row=5, column=2)
        Button('=').grid(row=5, column=3)

if __name__ == '__main__':
    Demo().run()
```

calc2.py

It turns out that it is simpler to place the calculator logic into a separate function, based on a single character:

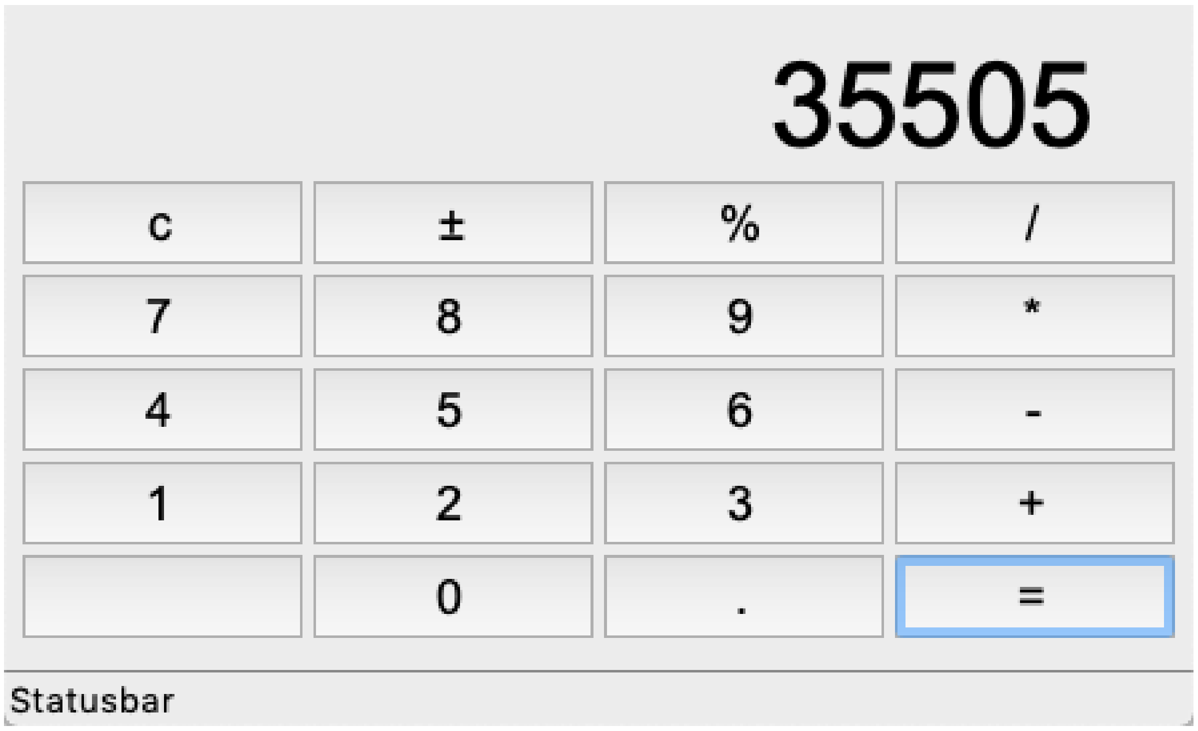
```
def calculate(self, c):
    """Calculator for 4 basic operations."""
    if c in '0123456789.±+-/*%c=':
        if c in '0123456789':
            if self.pos:
                self.val += int(c) * 10**self.pos
                self.pos -= 1
            else:
                self.val *= 10
                self.val += int(c)
        if c == '.':
            self.pos = -1
        if c == '%':
            self.val *= 0.01
        elif c in '+-/*':
            self.val2 = self.val
            self.val = 0
            self.pos = 0
            self.op = c
        elif c == '=':
            e = str(self.val2) + self.op + str(self.val)
            self.val = eval(e)
        if c == 'c':
            self.val = 0
            self.pos = 0

        self.lb['text'] = self.val
```



```python
"""Create calculator buttons."""
from tklib import *

class Demo(App):
    def __init__(self):
        super().__init__()

        s = ttk.Style()
        s.configure('TButton', font='Arial 18', padding=5)

        self.lb = Label('0', font='Arial 36', width=15, anchor='e')

        self.val = 0
        self.pos = 0
        self.val2 = 0

        App.root.bind('<Key>', self.cb)

    def cb(self, event=None):
        """React to key press events."""
        c = event.char
        if c != '':
            self.calculate(c)

    def calculate(self, c):
        """Calculator for 4 basic operations."""
        if c in '0123456789.±+-/*%c=':
            if c in '0123456789':
                if self.pos:
                    self.val += int(c) * 10**self.pos
                    self.pos -= 1
                else:
                    self.val *= 10
                    self.val += int(c)
            if c == '.':
                self.pos = -1
            if c == '%':
                self.val *= 0.01
            elif c in '+-/*':
                self.val2 = self.val
                self.val = 0
                self.pos = 0
                self.op = c
            elif c == '=':
                e = str(self.val2) + self.op + str(self.val)
                self.val = eval(e)
            if c == 'c':
                self.val = 0
                self.pos = 0

            self.lb['text'] = self.val

if __name__ == '__main__':
    Demo().run()
```

calc3.py

And finally we put everything together. New Calculator instances can be created via a button press or via a menu.



```
"""Create calculator buttons."""
from tklib import *

class Calculator():
    def __init__(self):
        self.win = Window('Calculator')
        s = ttk.Style()
        s.configure('TButton', font='Arial 18', padding=5)

        self.lb = Label('0', font='Arial 44', width=15, anchor='e')
        self.lb.grid(columnspan=4)

        buttons = 'c±%/789*456-123+ 0.='
        for i, b in enumerate(buttons):
            Button(b, lambda c=b: self.calculate(c)).grid(row=i//4+1, column=i%4)
        self.val = 0
        self.pos = 0
        self.val2 = 0
        self.op = ''

        self.win.top.bind('<Key>', self.cb)

    def cb(self, event=None):
        """React to key press events."""
        c = event.char
        if c != '':
            self.calculate(c)

    def calculate(self, c):
        """Calculator for 4 basic operations."""
        if c in '0123456789.±+-/*%c=':
            if c in '0123456789':
```

(continues on next page)

(continued from previous page)

```
                if self.pos:
                    self.val += int(c) * 10**self.pos
                    self.pos -= 1
                else:
                    self.val *= 10
                    self.val += int(c)
            elif c == '.':
                self.pos = -1
            elif c == '%':
                self.val *= 0.01
            elif c == '±':
                self.val *= -1
            elif c in '+-/*':
                self.val2 = self.val
                self.val = 0
                self.pos = 0
                self.op = c
            elif c == '=':
                e = str(self.val2) + self.op + str(self.val)
                self.val = eval(e)
            if c == 'c':
                self.val = 0
                self.pos = 0

            self.lb['text'] = self.val


class Demo(App):
    def __init__(self):
        super().__init__()
        Button('New calculator', Calculator)
        Menu('App')
        Item('Calculator', Calculator)

if __name__ == '__main__':
    Demo().run()
```

calc4.py

CHAPTER 18

Indices and tables

- genindex
- modindex
- search