# 2.3 A small lexicon of widgets - Part 3

There are two remaining widgets we want to tell you about – the first one is just a widget, while the second is, in fact, a **set of cooperating widgets**.

The `Entry` widget not only presents a line of text, but is also able to **edit** the text according to the user's actions. Using an `Entry` is necessary when you are going to ask the user for any textual information: name, password, email, etc. The widget implements all standard edit operations like inserting, removing, scrolling, selecting, copying and pasting, etc..

We'll show you only the basic possibilities of the widget, as it's full equipment is extremely complex. Fortunately, we don't need its entire flexibility when we just want to enter and validate a line of text.

Here are some of `Entry`'s properties:

| Entry property | Property meaning |
|---|---|
| `command` | although `Entry` is obviously a clickable widget, it doesn't allow you to bind a callback through the `command` property. You can observe and control all occurring changes instead by setting the **tracer function** for the observable variable which cooperates with `Entry` (we'll show you this – be patient!) |
| `show` | a string assigned to this property will be displayed instead of the **actual** characters entered into the input field; e.g., if you set `show='*'`, this will enable the widget to safely edit the user's password |
| `state` | the same as for `Button` |
| `textvariable` | an observable `StringVar` reflecting the current state of the input field |
| `width` | the input field's width (in characters) |

And now, some of `Entry`'s methods:

| Entry method | Method role |
|---|---|
| `get()` | returns the current input field's contents as a string |
| `set(s)` | sets the whole input field's contents with the s string |
| `delete(first, last=None)` | deletes a part of the input field's contents; `first` and `last` can be integers with values indexing the string; if the `last` argument is omitted, a single character is deleted; if `last` is specified as `END`, it points to the place after the last field's character |
| `insert(index, s)` | inserts the `s` string at the field position pointed to by `index` |

Our sample program in the editor shows you how to use an **observable variable** along with the **trace callback** (tracer) to force a user to enter **only digits** – all other characters will be silently **ignored**.

The tracer is invoked each time the input field is **modified**. The tracer remembers its **previous** state (using the last_s variable) and restores the field to this state if its current contents are invalid.

Note: we've had to use the `focus_set()` method, as the widget doesn't take the focus itself.

Run the code and test its behavior.

```
import tkinter as tk


def digits_only(*args):
    global last_string
    string = text.get()
    if string == '' or string.isdigit():  # Field's content is valid.
        last_string = string
    else:
        text.set(last_string)


last_string = ''
window = tk.Tk()
text = tk.StringVar()
entry = tk.Entry(window, textvariable=text)
text.set(last_string)
text.trace('w', digits_only)
entry.pack()
entry.focus_set()
window.mainloop()
```

Try to modify the code to allow the user to enter not more than five digits.

The last part of our story will tell you about menus.

```
mnu = Menu(master, option, ...)
```

Let's summarize the most important menu traits:

- a classic menu is actually a **horizontal bar** located at the top of the application window;
- the bar contains a number of horizontally deployed **options**, often referred to as **items** or **entries**;
- these options can have **hot-keys** (keyboard shortcuts enabling the user to quickly access selected operations without using a mouse; usually, hot-keys are triggered by pressing Alt-hotkey on the keyboard)
- **selecting a menu's option** (it doesn't matter whether through a hotkey or by a mouse click) causes one of **two effects**:
    - **it launches** a callback bound to the option;
    - **it unrolls** a new menu (actually a submenu)
- if you want to have such a menu within your Tkinter application, you have to:
    - **create** a top-level menu object;
    - **embed** it inside the window;
    - **bind** a number of required submenus (this is called a **cascade**) or connect a single callback.

We'll show you the whole process step-bystep. Track all our movements carefully.

Our steps are as follows:

- line 5: we define a simple callback which displays the About dialog;
- line 9: main window creation (nothing special at all)
- line 12: we create the main menu object...
- line 13: and embed it into the main window (note the way in which the config() method is used and which property we utilize to bind the menu)

- line 16: we create a submenu object (note the master window argument specification)
- line 17: we add the submenu to the main menu's first item (note the add_cascade() method invocation)
- line 20: we create another submenu object...
- line 21: ...and bind a callback to it (note the add_command() method invocation)

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


window = tk.Tk()

# main menu creation
main_menu = tk.Menu(window)
window.config(menu=main_menu)

# 1st main menu item: an empty (as far) submenu
sub_menu_file = tk.Menu(main_menu)
main_menu.add_cascade(label="File", menu=sub_menu_file)

# 2nd main menu item: a simple callback
sub_menu_help = tk.Menu(main_menu)
main_menu.add_command(label="About...", command=about_app)

window.mainloop()
```

Run the code and test it. Do you see that strange dashed line visible when you click the File main menu item?

Don't worry, this it's normal. We'll deal with it soon.

A menu like this has one important **disadvantage** – it's hard to use it without a mouse. Of course, you can use the *Alt* key to activate the menu and navigate through it using the cursor keys and Enter (you can test this), but we need something quicker and more convenient.

Look at the code below. We've made two changes – we've added two underlined property specifications to the `Menu()` invocations:

- at line 15: underline=0 (sets Alt-F as a hotkey)
- at line 18: underline=1 (sets Alt-B as a hotkey)

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


window = tk.Tk()

main_menu = tk.Menu(window)
window.config(menu=main_menu)
sub_menu_file = tk.Menu(main_menu)
# setting the hotkey to "Alt-F"
```

```
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
sub_menu_help = tk.Menu(main_menu)
# setting the hotkey to "Alt-B"
main_menu.add_command(label="About...", command=about_app, underline=1)

window.mainloop()
```

Check if it works – we hope it does. Note: you're obliged to ensure that all hotkeys are unique!

Now we'll extend the `File` submenu and add a `Quit` option to this.

The option's action will be implemented by a simple callback giving the user the chance to change their mind.

We'll use the `add_command()` method to achieve this goal.

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


def are_you_sure():
    if messagebox.askyesno("", "Are you sure you want to quit the App?"):
        window.destroy()


window = tk.Tk()

main_menu = tk.Menu(window)
window.config(menu=main_menu)
sub_menu_file = tk.Menu(main_menu)
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
# add the QUIT action to the submenu
sub_menu_file.add_command(label="Quit", underline=0, command=are_you_sure)
sub_menu_help = tk.Menu(main_menu)
main_menu.add_command(label="About...", command=about_app, underline=1)

window.mainloop()
```

This is what we get:

Check line #21 – this will tell you everything. Run the code.

The strange dashed line appearing at the top of the File submenu is called the **tear-off**, an archaic detail used by very old GUIs. We don't need it. We don't even want to know how it worked in the past.

Let's get rid of it in a very simple way.

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")
```

```
def are_you_sure():
    if messagebox.askyesno("", "Are you sure you want to quit the App?"):
        window.destroy()


window = tk.Tk()

main_menu = tk.Menu(window)
window.config(menu=main_menu)
# we don't want the tear-off here
sub_menu_file = tk.Menu(main_menu, tearoff=0)
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
sub_menu_file.add_command(label="Quit", underline=0, command=are_you_sure)
sub_menu_help = tk.Menu(main_menu)
main_menu.add_command(label="About...", command=about_app, underline=1)

window.mainloop()
```

Analyze line #19, and run the code to see the difference.

Of course, you can add as many submenu items as you want. We've added one now – can you see it?

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


def are_you_sure():
    if messagebox.askyesno("", "Are you sure you want to quit the App?"):
        window.destroy()


def open_file():
    messagebox.showinfo("Open doc", "We'll open a file here...")


window = tk.Tk()

main_menu = tk.Menu(window)
window.config(menu=main_menu)
sub_menu_file = tk.Menu(main_menu, tearoff=0)
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
# a new submenu item is here!
sub_menu_file.add_command(label="Open...", underline=0, command=open_file)
sub_menu_file.add_command(label="Quit", underline=0, command=are_you_sure)
sub_menu_help = tk.Menu(main_menu)
main_menu.add_command(label="About...", command=about_app, underline=1)

window.mainloop()
```

Yes, it's Open, and we added it to line #25.

If you want the submenu to display its contents in a more readable way, you can add a separator to it. This is done by a method named... `add_separator()`, of course.

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


def are_you_sure():
    if messagebox.askyesno("", "Are you sure you want to quit the App?"):
        window.destroy()


def open_file():
    messagebox.showinfo("Open doc", "We'll open a file here...")


window = tk.Tk()

main_menu = tk.Menu(window)
window.config(menu=main_menu)
sub_menu_file = tk.Menu(main_menu, tearoff=0)
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
sub_menu_file.add_command(label="Open...", underline=0, command=open_file)
# separator is here!
sub_menu_file.add_separator()
sub_menu_file.add_command(label="Quit", underline=0, command=are_you_sure)
sub_menu_help = tk.Menu(main_menu)
main_menu.add_command(label="About...", command=about_app, underline=1)

window.mainloop()
```

If any of the submenu's items should unroll another cascade, you can add it using the well-know `add_cascade()` method, just like in lines #23 and #27:

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


def are_you_sure():
    if messagebox.askyesno("", "Are you sure you want to quit the App?"):
        window.destroy()


def open_file():
    messagebox.showinfo("Open doc", "We'll open a file here...")


window = tk.Tk()
```

```
main_menu = tk.Menu(window)
window.config(menu=main_menu)
sub_menu_file = tk.Menu(main_menu, tearoff=0)
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
sub_menu_file.add_command(label="Open...", underline=0, command=open_file)

sub_sub_menu_file = tk.Menu(sub_menu_file, tearoff=0)
sub_menu_file.add_cascade(label="Open recent file...", underline=5,
menu=sub_sub_menu_file)

sub_menu_file.add_separator()
sub_menu_file.add_command(label="Quit", underline=0, command=are_you_sure)
sub_menu_help = tk.Menu(main_menu)
main_menu.add_command(label="About...", command=about_app, underline=1)

window.mainloop()
```

Adding an item (or items) to the newly added cascade requires some steps you already know – look, we've engaged the `for` loop to simulate the presence of eight recently opened files (lines #28 through #30):

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


def are_you_sure():
    if messagebox.askyesno("", "Are you sure you want to quit the App?"):
        window.destroy()


def open_file():
    messagebox.showinfo("Open doc", "We'll open a file here...")


window = tk.Tk()

main_menu = tk.Menu(window)
window.config(menu=main_menu)
sub_menu_file = tk.Menu(main_menu, tearoff=0)
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
sub_menu_file.add_command(label="Open...", underline=0, command=open_file)
sub_sub_menu_file = tk.Menu(sub_menu_file, tearoff=0)
sub_menu_file.add_cascade(label="Open recent file...", underline=5,
menu=sub_sub_menu_file)

for i in range(8):
    number = str(i + 1)
    sub_sub_menu_file.add_command(label=number + ". file.txt", underline=0)

sub_menu_file.add_separator()
sub_menu_file.add_command(label="Quit", underline=0, command=are_you_sure)
sub_menu_help = tk.Menu(main_menu)
```

```
main_menu.add_command(label="About...", command=about_app, underline=1)

window.mainloop()
```

If you want some of the (sub)menu items to be accessible through a dedicated **hot-key**, you have to perform two steps:

- **set** the item's property named `accelerator` with a string naming the hot-key (note: this has no other effect than just displaying the right-aligned string inside the item – **no callback is set at the moment**)
- make a **global binding** linking the hot-key with a proper callback.

It's invoked in line #21. Check how the separator presents itself inside the submenu.

```
import tkinter as tk
from tkinter import messagebox


def about_app():
    messagebox.showinfo("App", "The application\nthat does nothing")


def are_you_sure(event=None):
    if messagebox.askyesno("", "Are you sure you want to quit the App?"):
        window.destroy()


def open_file():
    messagebox.showinfo("Open doc", "We'll open a file here...")


window = tk.Tk()

main_menu = tk.Menu(window)
window.config(menu=main_menu)
sub_menu_file = tk.Menu(main_menu, tearoff=0)
main_menu.add_cascade(label="File", menu=sub_menu_file, underline=0)
sub_menu_file.add_command(label="Open...", underline=0, command=open_file)
sub_sub_menu_file = tk.Menu(sub_menu_file, tearoff=0)
sub_menu_file.add_cascade(label="Open recent file...", underline=5,
menu=sub_sub_menu_file)

for i in range(8):
    number = str(i + 1)
    sub_sub_menu_file.add_command(label=number + ". file.txt", underline=0)

sub_menu_file.add_separator()
sub_menu_file.add_command(label="Quit", accelerator="Ctrl-Q",
                          underline=0, command=are_you_sure)
sub_menu_help = tk.Menu(main_menu)
main_menu.add_command(label="About...", command=about_app, underline=1)

window.bind_all("<Control-q>", are_you_sure)
window.mainloop()
```

Look – we've applied these steps to our code making Ctrl-Q a shortcut designed to close the application – it happens in lines #9 (we modified the AreYouSure() function header), #33 (we added the accelerator property)

and #38 (global binding to <Control-q>).

Note: you cannot modify any of the (sub)menu item by using the standard `config()` method invocation, because from `tkinter`'s point of view, **the item is not a widget** – it's only a very specific widget **component**.

If you want to manipulate a menu's item, you should use a dedicated method named `entryconfigure()`. The method accepts two parameters:

```
item.entryconfigure(i, prop=value)
```

- the first is an integer **index** of the modified item (entry)
- the second is a keyworded argument pointing to the **modified property**.

The snippet shows you how it works – it plays with the second entry's state property – run the code, and observe its behavior:

Let's gather together some of the menu's properties and methods:

| Property | Property role |
|---|---|
| `postcommand` | a **callback** invoked every time a menu's item is activated |
| `tearoff` | set to zero **removes** the tear-off decoration from the top of the cascade |
| `state` | when set to `DISABLED`, the menu item is grayed and inaccessible; setting it to `ACTIVE` restores its normal functionality |
| `accelerator` | a **string** describing a hot-key bound to the menu's item |

| Method | Method role |
|---|---|
| `add_cascade(prop=val, ...)` | adds a **cascade** to the menu's item |
| `add_command(prop=val, ...)` | assigns an **action** to the menu's item |
| `add_separator()` | adds an **separator** line to the menu |
| `entryconfigure(i, prop=val,...)` | modifies the *i*-th menu item's property named `prop` |

Now we're going to show some ways of shaping the main window's appearance.

```
import tkinter as tk


def on_off():
    global accessible
    if accessible == tk.DISABLED:
        accessible = tk.ACTIVE
    else:
        accessible = tk.DISABLED
    sub_menu.entryconfigure(1, state=accessible)


accessible = tk.DISABLED
window = tk.Tk()
```

```
menu = tk.Menu(window)
window.config(menu=menu)
sub_menu = tk.Menu(menu, tearoff=0)
menu.add_cascade(label="Menu", menu=sub_menu)
sub_menu.add_command(label="On/Off", command=on_off)
sub_menu.add_command(label="Switch", state=tk.DISABLED)
window.mainloop()
```