

GUI programming

Lecture 18

GUI and Event-driven programming

With the Tk binding of Python, we will learn how to write programs with graphical user interface (GUI). We will see that the logic of the whole program need to be changed to event driven.

References:

- Manpages of Tk: <http://dev.scriptics.com/man/tcl8.3/Tk-Lib/contents.htm>
- An introduction to Tkinter: <http://www.pythonware.com/library/an-introduction-to-tkinter.htm>
- Tkinter Life Preserver: <http://www.python.org/doc/life-preserver/index.html>

PMOOP(0396A)

PMOOP(0396A)-18.1

The challenge of GUI programming

Any programming model supporting GUI programming must address the following issues:

- At any time, **the user is allowed to perform a large variety of different things**.

E.g., he may type a character, type a hot-key, click on a button, resize the window, obscure the window with another application, minimize it, etc.

- The program **need to adapt itself to different window size**.

At some times the user want the window to be large, to display larger edit area, to get better resolution, etc. At other times the user want the window to be small, so that he can see other windows at the same time.

Accordingly, most windowing systems use **event-driven model**. They use **widget systems** with window sizes controlled by a **geometry manager**.

PMOOP(0396A)-18.2

PMOOP(0396A)-18.3

Using Python Tkinter

To use Tkinter, the first thing to do is to import the Tkinter library:

```
from Tkinter import *
```

Then we should create the main window of the application:

```
mainwin = Tk()
```

This will give you a main window to play with. If you write it as a script (instead of interactively), you should execute the following before the script ends:

```
mainwin.mainloop()
```

This **keeps the script running**. Otherwise the script exits and the window has no time to get displayed on your screen.

PMOOP(0396A)-18.4

PMOOP(0396A)-18.5

The term “graphical user interface” should need no introduction.

Good things about GUI

- Intuitive to the user, if designed properly.
- Some tasks, e.g., putting the cursor at a particular location, is more efficient with GUI.
- Use multiple threads of activities intuitively.

Bad things about GUI

- Easily mis-designed, and mis-designed GUI is inefficient to use.
- Difficult to automate things done by a GUI. (e.g., difficult to write a program to push a button in another application.)

The windowing system of Python: Tkinter

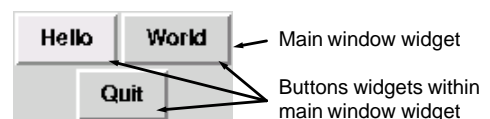
Python uses a windowing system called Tkinter. It is related with the windowing system in the following way.

Our script (in Python): real application
Tkinter (in Python): glue to tkinter
tkinter Python plugin (in C): translate to Tk calls
Tk widgets (in C and Tcl): the Widget implementation
Tklibrary (in C): glue to Xlib
Xlib (in C): the actual windowing system

This is a rather tall hierarchy involving 3 languages, so not very efficient. But so what... we don't need GUI to be really fast most of the time.

The widget system

Each major window component is called a **widget**, and is represented by a **Python object** (i.e., a variable of a python class).



Note that a **widget can be contained within another widget**—just like a *TExpression* can be contained within another *TExpression* in our assignment 2.

The word Widget means “windowing gadget”, which means “a useful building block (gadget) to make a windowing system”.

There are many different type of widgets, as we will see.

PMOOP(0396A)-18.4

PMOOP(0396A)-18.5

Making the widgets

To make a widget, you can **construct** it using `ClassName()`. E.g., use `Button()` to create a button.

You can specify a **parent** when constructing an object, e.g., `Button(parent)`. By default, the parent is the main window. There can also be other arguments, specifying the options of the widget.

E.g., to make all the widgets in the previous example:

```
from Tkinter import *
mainwin = Tk()
a = Button(mainwin, text="Hello")
b = Button(mainwin, text="World")
c = Button(mainwin, text="Quit")
# something missing here...
mainwin.mainloop()
```

But this won't show the window: we are still one step behind.

PMOOP(0396A)-18.6

Named arguments in Python function

You should notice something very funny in the program: we can say

```
a = Button(mainwin, text="Hello")
```

I.e., arguments need not be ordered like the function definition. Such **named arguments** are very useful when there are many arguments.

In Python, you can give default values to arguments just like C++:

```
def MakeComplex(real=0, imag=0):
    return complex(real, imag)
```

You can pass arguments to the function by either **position** or **name**:

```
MakeComplex(2, 5)      # Normal call
MakeComplex(2)         # Call MakeComplex(2, 0)
MakeComplex(imag=2)   # Call MakeComplex(0, 2)
```

PMOOP(0396A)-18.7

Geometry manager

We need specifying **where** we want the widget to appear, and **how much space** to allocated to it. But we can't say "I want it at coordinate (0, 0) with size (500, 100)", since the **user might resize the window**.

Instead, each widget (**master**) uses a **geometry manager** to decide the exact position of its children (**slave**). The children uses "GM slave options" to modify the behaviour of the geometry manager. E.g., (**w1.py**):

```
a.grid(row=0, column=0, sticky=N+E+S+W)
b.grid(row=0, column=1, sticky=N+E+S+W)
c.grid(row=1, column=0, columnspan=2, sticky=N+S)
mainwin.mainloop()
```

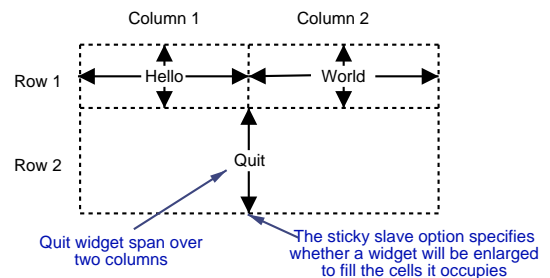
This tells that the geometry manager **grid** should be used, and specify the grid coordinates of each widget.

There is also a **pack** and a **place** geometry manager, achieving other effects.

PMOOP(0396A)-18.8

The grid geometry manager

The grid geometry manager consider the **space of master as a grid of cells**. Each slave spans through a few of these cells.



A slave option "**sticky**" determines whether it will expand to fill up the space it occupies. (Example: tries to add E+W option to Quit.)

PMOOP(0396A)-18.9

Other grid slave options

There are more grid slave options to control how a slave should be drawn. Use the manpage of Tk, "man grid", to read the complete documentation. Briefly:

- **column, row**: the place of the top-left corner
- **columnspan, rowspan**: the number of cells allocated to it.
- **padx, pady**: the amount of empty space to insert outside the border
- **ipadx, ipady**: same as padx and pady, but inside border
- **sticky**: the direction that the slave should expand
- **in**: a widget in which the slave is put into. By default this is the parent widget specified when the widget is created.

These are listed in the "configure" part of the manual page.

PMOOP(0396A)-18.10

Dealing with resize

What will happen if we **resize** the window?

While the slaves want to expand, none of the rows or columns of the master expands! We need to set some **grid master options** to allow this. (**w2.py**)

```
mainwin.columnconfigure(0, weight=1)
mainwin.rowconfigure(0, weight=1)
```

So row 0 and column 0 expands, while row 1 and column 1 won't. If more than one row/column expand, **weights decides which expands more**.

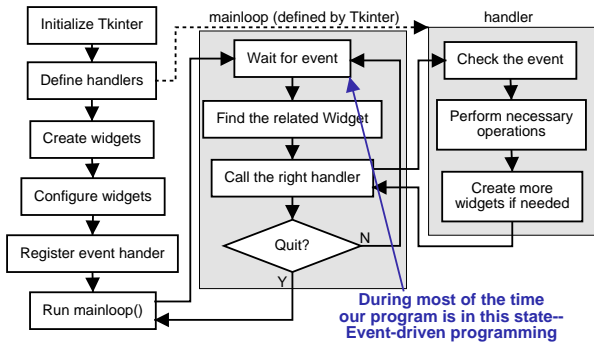
Note that the option is set in the *master*, not *slave*, because we are not configuring individual slave. Other options:

- **pad**: the amount of space to add outside the column or row.
- **minsize**: the minimum amount of space to allocate.

PMOOP(0396A)-18.11

Event-driven model

To do something useful, our program must **react to users' actions**. The programming model to allow this:



PMOOP(0396A)-18.12

Specifying functions to execute

We can specify a function (handler) to run on button pressed (**w3.py**):

```
...
a['command']=greet_button_handler
b['command']=greet_button_handler
c['command']=mainwin.quit
mainwin.mainloop()
```

The button widget provides an option that specifies a command to execute when the button is pressed. It is set using the **command** widget option.

To find the list of widget options, read the Tk manpage "button". When reading Tk manpages, note that Tk options becomes elements of the object.

The option can also be set when creating the object, like this:

```
a = Button(mainwin, text="hello", command=greet_button_handler)
```

PMOOP(0396A)-18.13

More on events

Note that the handler takes no argument. If we want to know the triggering widget, there is a possible way: a function that returns a function (**w4.py**).

```
def make_greet_handler(widget):
    def handler(mywidget=widget):
        print mywidget['text']
    return handler
...
a['command']=make_greet_handler(a)
b['command']=make_greet_handler(b)
```

Here mywidget is the argument of the newly defined handler, and widget is the *default argument*. When the event loop call the handler, no argument is passed, so the default argument applies.

N.B.: In Python 2.1, the nested function can use the variables of the enclosing function without such default-argument tricks.

PMOOP(0396A)-18.14

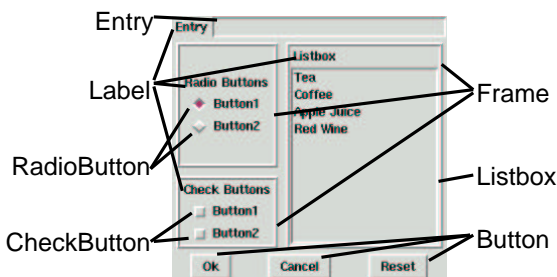
Some common widget options

The common widget options are described in the Tk "options" manpage. Some more important ones:

- **text, bitmap, image:** the thing to show, e.g., in a label or button.
- **foreground, background:** the foreground and background color.
- **activeforeground, activebackground:** Same, but used when widget is "active", e.g. when the mouse is over it.
- **borderwidth:** The width of the widget border.
- **jump:** whether a scrollbar or dial will update continuous or jump on dragging.
- **orient:** the orientation of scrollbar, etc.
- **font:** the font to use for text. Read manpage of font to see how fonts are specified.

PMOOP(0396A)-18.15

Available widgets (eg1.py)

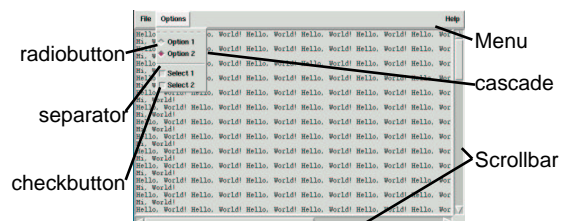


Try to figure out how the program associate a variable to the radio button.

Note also how frames are used to put a grid within another grid.

PMOOP(0396A)-18.16

More available widgets (eg2.py)



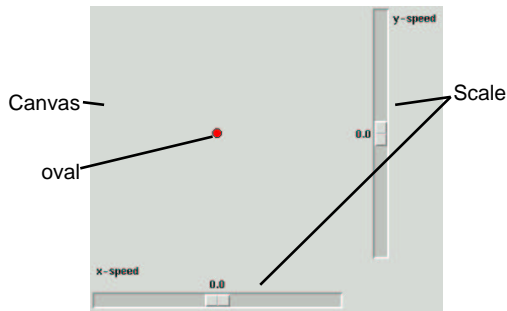
The text widget and the scrollbars are connected.

When a Scrollbar is dragged, the handler specified in the command option of Scrollbar notifies the text widget.

When the text changes, the handlers specified in xscrollcommand and yscrollcommand options of text notifies the scrollbar.

PMOOP(0396A)-18.17

Even more available widgets (eg3.py)



This also demonstrates how to make an event occur some time after the current time (using `mainwin.after(time_in_minisecond, handler)`).

PMOOP(0396A)-18.18

More about events

- So far we have seen one type of events: those **triggered by a button** when it is pressed.
- Such commands are available for buttons, check buttons, radio buttons, scroll bars, scales and menu entries.
- But even **when the widget does not provide a command**, we might want to know whether something happens on it.
- We can **bind an 'event'** to a widget.

```
def process_entry(event): # Event handler takes one argument, the event
    print myentry.get()
```

```
myentry = Entry()
# Bind to <Return> event, i.e., when somebody press Return
myentry.bind('<Return>', process_entry)
```

PMOOP(0396A)-18.19

Available events

A large number of events are available ('man n bind' to see all). Some more important ones:

- <Button-1>, <ButtonRelease-1>, <DoubleClick-1>
Mouse button 1 is pressed, released or double-clicked.
- <Enter>, <Leave> The pointer enters/leaves a widget.
- <Configure> The widget size is changed.
- <Key>-a The key 'a' is pressed.
- <Shift-Up> Shift-up is pressed.

The event is a map contains the following keys:

- widget The triggering widget.
- x, y The pointer position when the event is triggered.
- width, height Width and height of new window size.
- num, char The button number or key that is pressed.

PMOOP(0396A)-18.20

What's next for Tkinter

This quick overview omits many features of Tkinter. E.g.,

- Two other **geometry managers** are available. The **pack** manager gives one side of its area to a widget each time. The **place** manager allows widgets to be put to absolute coordinates.
- Some **standard dialogs**: `tkMessageBox`, `tkColorChooser`, `FileDialog`.
- You can create two types of non-widget images: **bitmap** and **photoimage**, which respectively stores a black/white and color image.
- We can set an **idle command**, which is executed whenever there is nothing to do in the mainloop (i.e., all events are handled).
- A **Drag-and-drop** protocol is defined. Look at `Tkdnd.py`.
- Even more widgets are available if you install the **"Pmw"** package. More image operations are available with **PIL** (Python Imaging Library).

PMOOP(0396A)-18.21