

GUI programming

Lecture 21

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/tcl8.3.2/-TkCmd/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)-21.1

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)-21.2

PMOOP(0396A)-21.3

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 is to keep the script running. Otherwise the script exits and the window has no time to get realized on your screen.

PMOOP(0396A)-21.4

PMOOP(0396A)-21.5

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

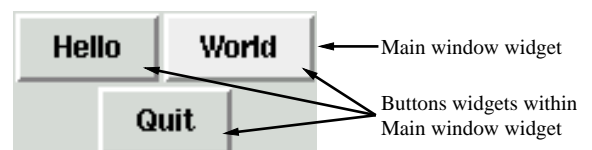
Python uses a windowing system called Tkinter. It is related with the windowing system in the following way.

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
Tk library (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.



Note that a **widget can be contained within another widget**: the Composite pattern is used in Tk.

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)-21.4

PMOOP(0396A)-21.5

Making the widgets

To make a widget, you use the function which name is the same as the widget's type. E.g., use `Button(parent)` to create a button. Here *parent* is the widget in which the new widget is created.

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

```

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)-21.6

Named arguments in Python function

You should notice something very interesting 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)-21.7

Geometry manager

To display a widget, you specify where you want the widget to appear, and how much space should be allocated to it.

But as we have seen, this cannot be done by just saying "I want this widget to be at screen coordinate (0, 0) with size (500, 100)", since the *user might resize the window*.

Instead, a **geometry manager** decides the exact position. We just configure 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()

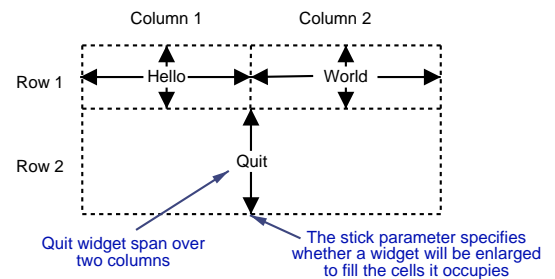
```

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

PMOOP(0396A)-21.8

The grid geometry manager

The grid geometry manager (master) consider its space as a grid of cells. Each widget in it (slave) spans through a few of these cells.



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

PMOOP(0396A)-21.9

Other grid 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

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

They are mapped to Python in the way we have just shown: *each option becomes a named argument to the grid member function*.

PMOOP(0396A)-21.10

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 want to expand! We need to setup some **grid master options** to allow this.

```

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

```

Now row 0 and column 0 can resize, while row 1 and column 1 can't.

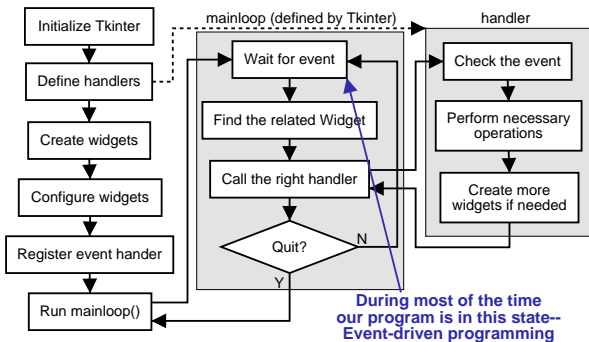
Note that the option is set in the *master*, not *slave*. Other options:

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

PMOOP(0396A)-21.11

Event-driven model

To do something useful, our program should react to users' activities. The programming model to allow this (**w2.py**):



PMOOP(0396A)-21.12

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)-21.13

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)-21.14

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)-21.15