INTRO2CS

Tirgul 12 – GUI

Today:

- GUI - tkinter
  - Packing
  - Events
  - Canvas
  - OptionMenu

- Lambda and Nested Functions

GUI

Graphical User Interface

- A GUI is a graphical (rather than purely textual) user interface to a computer.

- In the past interfaces to computers were not graphical, they were text-and-keyboard oriented.

- Today almost all operating systems, applications and programs we use consist of a GUI.

GUI in Python

- tkinter is the standard GUI library for Python.
- The GUI consist of the main window and different widgets within it.

GUI in Python

- To initialize Tkinter, we have to create a Tk **root** widget. This is an ordinary window, with a title bar and other decoration provided by your window manager.

- You should only create one root widget for each program, and it must be created before any other widgets.

- The root widget contains all other widgets.

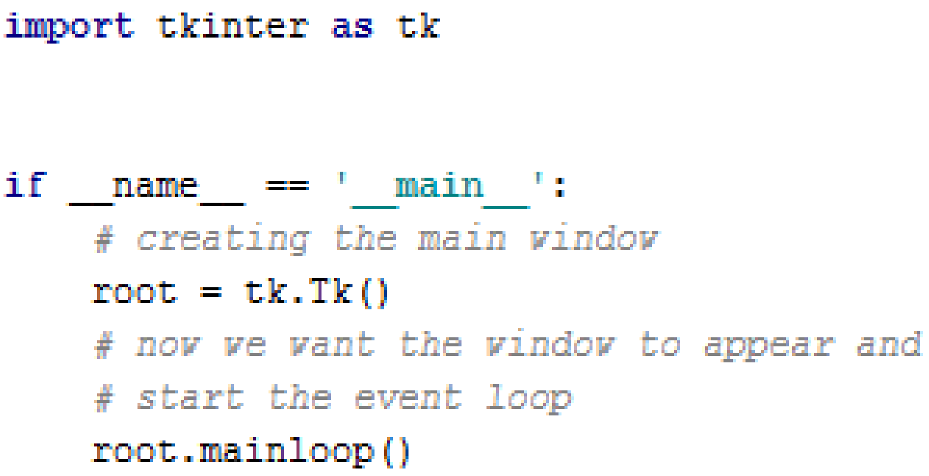
GUI in Python

- After creating the main window and its contained widgets, the window will not appear until we will enter the event loop by calling ***mainloop()*** method on the main window.

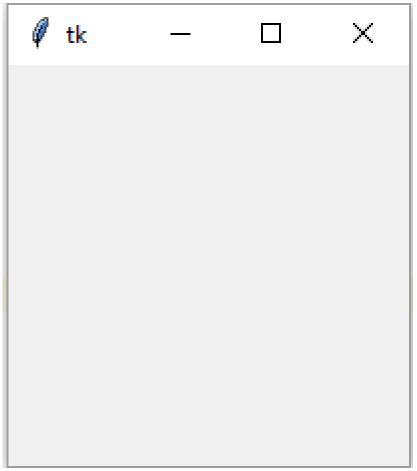
- The program will stay in the event loop until we close the window.

- It enables us to handle events from the user.

GUI in Python



```
import tkinter as tk


if __name__ == '__main__':
    # creating the main window
    root = tk.Tk()
    # now we want the window to appear and
    # start the event loop
    root.mainloop()
```



- The window won’t appear until we’ve entered the Tkinter event loop (mainloop).
- The program will stay in the event loop until we close the window.

Widgets

- ***Widget*** is an element of interaction in a GUI, such as button or scroll bar.
- Tkinter provides the following widgets:
  - Button
  - Canvas
  - entry
  - Frame
  - Label
  - Menu
  - text
  - And many more..

http://www.python-course.eu/python_tkinter.php

Adding Widgets

- All widgets are implemented in widgets classes, so each time we use a widget we create such object.

- The first parameter in the constructor of a widget is its parent widget.

Widget Geometry Managers:

- After adding a widget, we need to call a geometry manager in order to display it.

- There are three special methods that we can use for doing that: ***grid, pack*** and ***place***.

- Try not to use grid and pack on the same container.

Widget *pack()* Method

- ***pack()*** method of the widget object:
  widget.pack(pack_options)

- This organizes widgets in blocks before placing them in the parent widget.

- The ***pack*** method doesn’t really display the widget; it adds the widget to a list of widgets managed by the parent widget.

Widget *pack()* options:

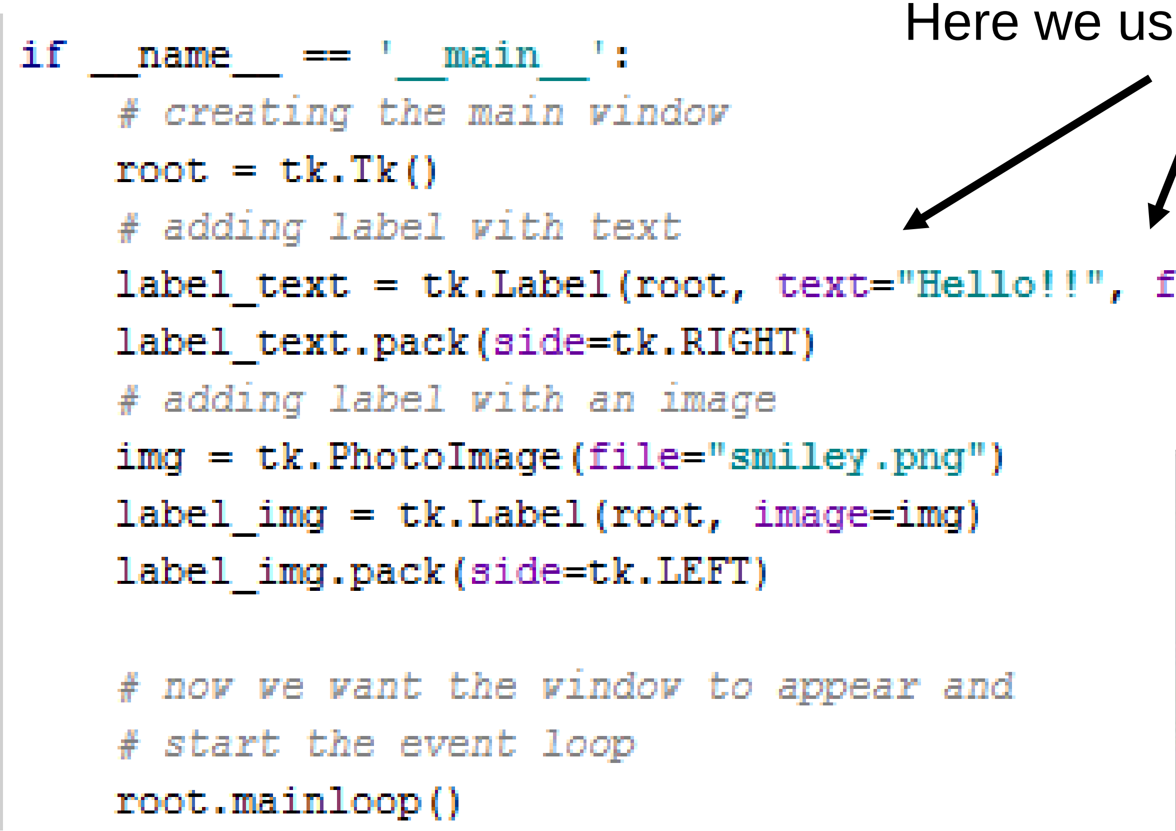
- **expand:** When set to true, widget expands to fill any space not otherwise used in widget's parent.
- **fill:** Determines whether widget fills any extra space allocated to it by the packer only horizontally, only vertically, both or none (default).
- **side:** Determines which side of the parent widget packs against: TOP (default), BOTTOM, LEFT, or RIGHT.

Adding Widgets - Label

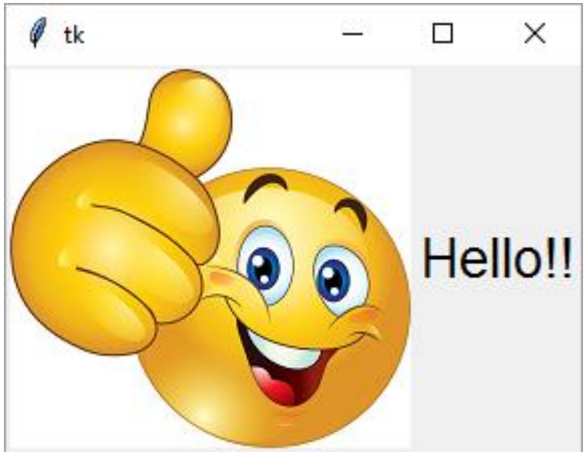
- The label is a widget that the user just views but not interact with.
- A Label is a widget which is used to display text or an image.
- We can set the text in the *text* argument in the Label constructor.
- More options: http://effbot.org/tkinterbook/label.htm

Adding Label Widget

Here we used *text* and *font* options



```
if __name__ == '__main__':
    # creating the main window
    root = tk.Tk()
    # adding label with text
    label_text = tk.Label(root, text="Hello!!", font=("Helvetica", 20))
    label_text.pack(side=tk.RIGHT)
    # adding label with an image
    img = tk.PhotoImage(file="smiley.png")
    label_img = tk.Label(root, image=img)
    label_img.pack(side=tk.LEFT)

    # now we want the window to appear and
    # start the event loop
    root.mainloop()
```



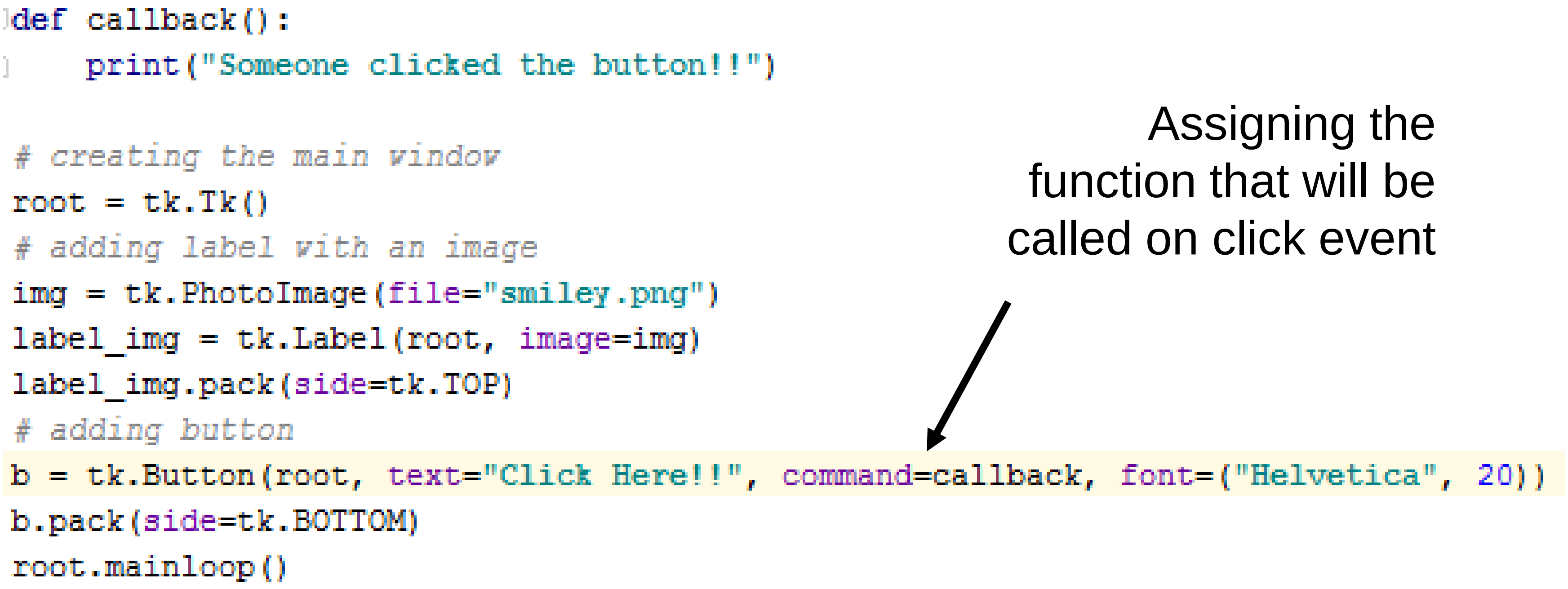
The Button Widget

- Buttons can contain text or images.

- Buttons can be associated with a function or a method that will be called after button click event.

- The association is by assigning command argument in the Button constructor to the function we wand to call.

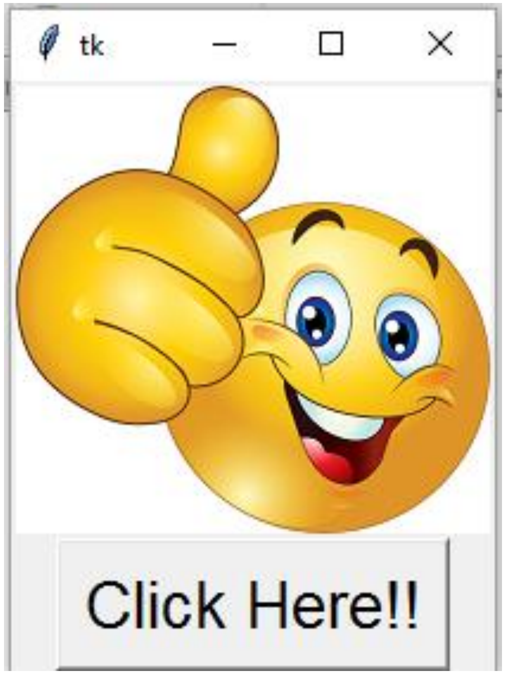
The Button Widget



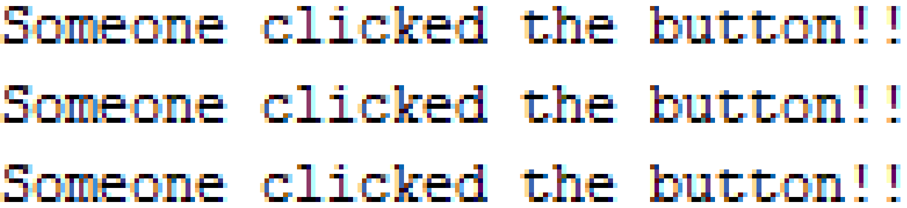
```
def callback():
    print("Someone clicked the button!!")

# creating the main window
root = tk.Tk()
# adding label with an image
img = tk.PhotoImage(file="smiley.png")
label_img = tk.Label(root, image=img)
label_img.pack(side=tk.TOP)
# adding button
b = tk.Button(root, text="Click Here!!", command=callback, font=("Helvetica", 20))
b.pack(side=tk.BOTTOM)
root.mainloop()
```

Assigning the function that will be called on click event

The Button Widget



Clicking three times on the button:



```
Someone clicked the button!!
Someone clicked the button!!
Someone clicked the button!!
```

The Canvas Widget

- The Canvas widget provides structured graphics facilities.
- It can be used to draw graphs and plots, create graphics editors, and implement various kinds of custom widgets.
- To display things on the canvas, you create one or more *canvas items* using ***create*** methods.
- The create method returns the item id.

Canvas Items and their Create Methods

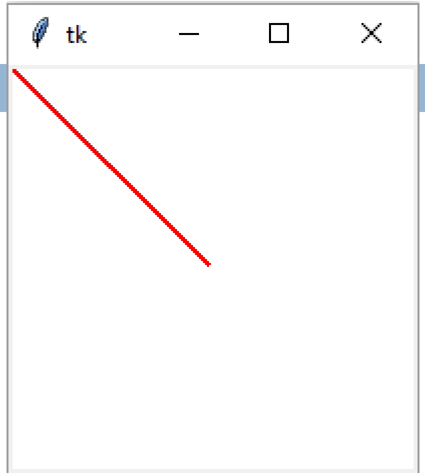
□ arc	□ create_arc
□ image	□ create_image
□ line	□ create_line
□ oval	□ create_oval
□ polygon	□ create_polygon
□ rectangle	□ create_rectangle
□ text	□ create_text
□ window	□ create_window

Canvas items

- The window coordinates of the canvas start at the upper left corner (this is (0,0)).

- The create methods get the coordinates as arguments, separated by commas.

Canvas Widget



```
# creating the main window
root = tk.Tk()
# adding canvas
w = tk.Canvas(root, width=200, height=200, bg='white')
w.pack()
# we will add a line starting at point (0,0)
# and ending at point (100, 100)
line = w.create_line(0, 0, 100, 100, fill='red', width=2)
root.mainloop()
```

we set the color and the
width of the line

Events and Bindings

- When we run the mainloop(), we are actually in events loop.
- Events can come from various sources, including key presses and mouse operations by the user, and redraw events from the window manager.
- For each widget, we can **bind** Python functions and methods to events:

  widget.bind(event, handler)

Events and Bindings

- If an event matching the *event* description occurs in the widget, the given *handler* is called with an object describing the event.

- The event is an object, and the handler is a callback method that we can implement.

Some Events Formats

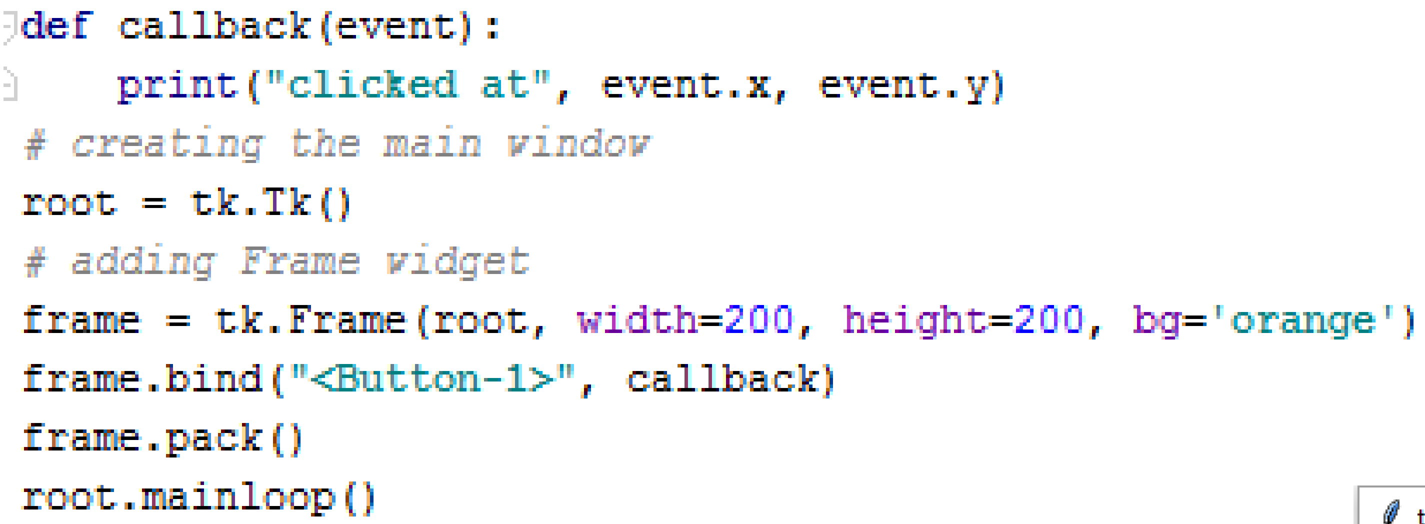
- <Button-1> , <Button-2>, <Button-3>
  - A mouse button is pressed over the widget. Button 1 is the leftmost button, button 2 is the middle button (where available), and button 3 the rightmost button.
  - The current position of the mouse pointer is provided in the **x** and **y** members of the event object passed to the callback.

Some Events Formats

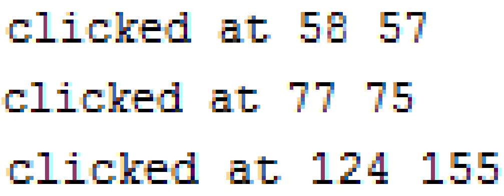
- <B1-Motion>
  - The mouse is moved, with mouse button 1 being held down (use B2 for the middle button, B3 for the right button).
  - The current position of the mouse pointer is provided in the **x** and **y** members of the event object passed implicitly to the callback.
- There are more events (mouse and keyboard):

http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm

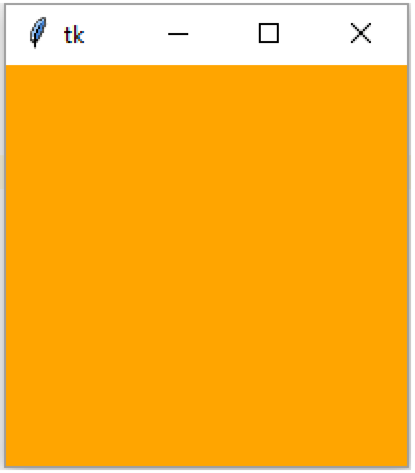
Events and Bindings - example



```
def callback(event):
    print("clicked at", event.x, event.y)
# creating the main window
root = tk.Tk()
# adding Frame widget
frame = tk.Frame(root, width=200, height=200, bg='orange')
frame.bind("<Button-1>", callback)
frame.pack()
root.mainloop()
```



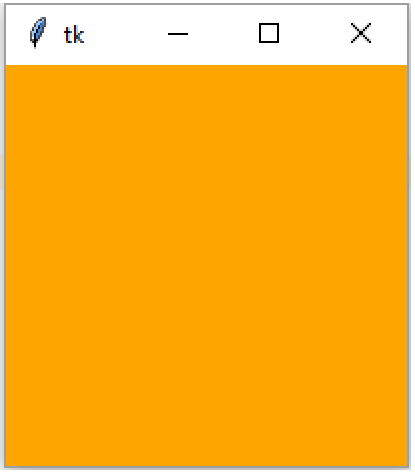
```
clicked at 58 57
clicked at 77 75
clicked at 124 155
```

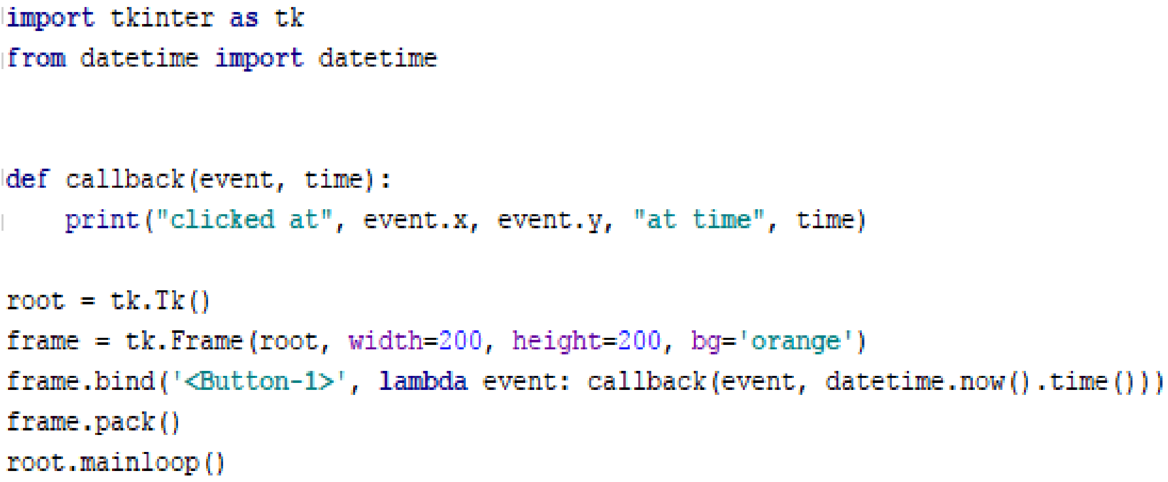


Read about Frame widget:
http://effbot.org/tkinterbook/frame.htm

Events and Bindings - example

- if we want to pass additional arguments to the callback function we can use lambda:





```
import tkinter as tk
from datetime import datetime


def callback(event, time):
    print("clicked at", event.x, event.y, "at time", time)

root = tk.Tk()
frame = tk.Frame(root, width=200, height=200, bg='orange')
frame.bind('<Button-1>', lambda event: callback(event, datetime.now().time()))
frame.pack()
root.mainloop()
```

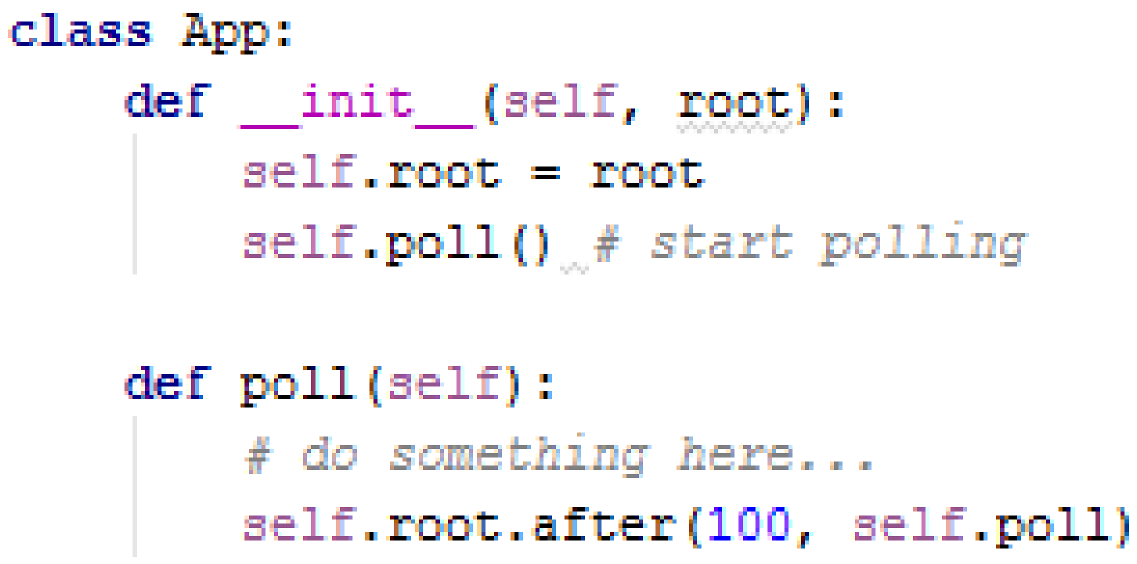
- clicked at 125 77 at time 12:17:20.207278
- clicked at 57 163 at time 12:17:21.814654
- clicked at 37 47 at time 12:17:24.811944

The *after* Method

after(delay_ms, callback=None, *args)

- ❑ This is a widget method.
- ❑ It registers a callback function that will be called after a given number of milliseconds.
- ❑ Since for running the GUI we are in a loop (the *mainloop()*), we can use after to call a certain method over and over again.

Using ***after*** example:



```
class App:
    def __init__(self, root):
        self.root = root
        self.poll() # start polling

    def poll(self):
        # do something here...
        self.root.after(100, self.poll)
```

Protocols Handlers

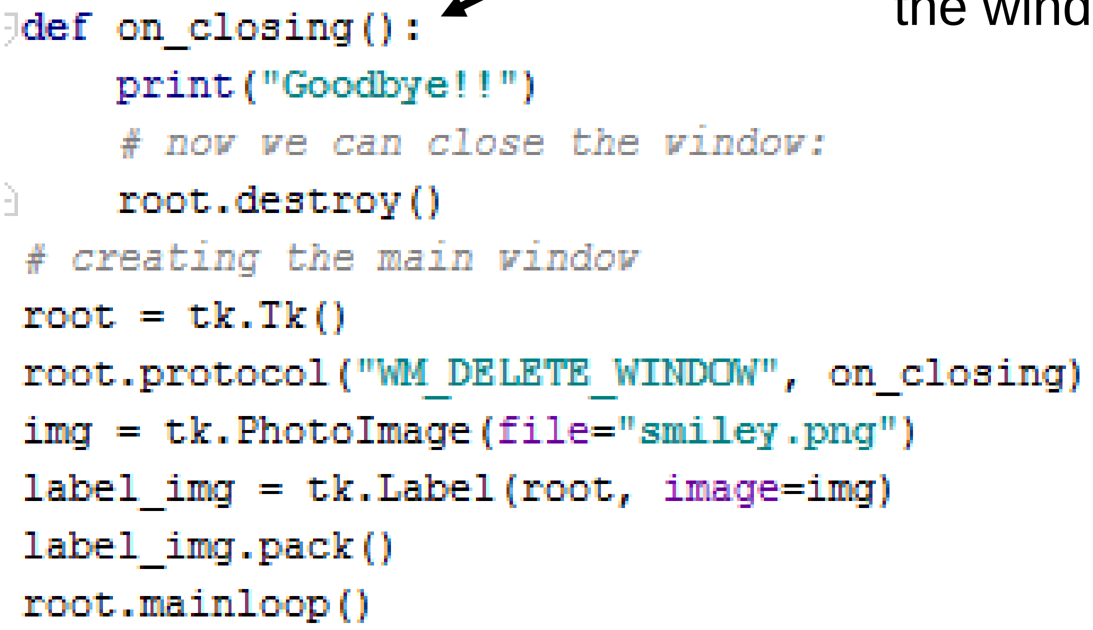
- The protocols refer to interaction between the application and the window manager.
- One example is closing a window using the window manager (the built-in x button on the upper right).
- Say we want to do something when the user closes the window
- We can use the protocol: **WM_DELETE_WINDOW**

Protocols Handlers

- The protocol method, much like bind, receives the protocol name and the handler (the method) that should be called upon it

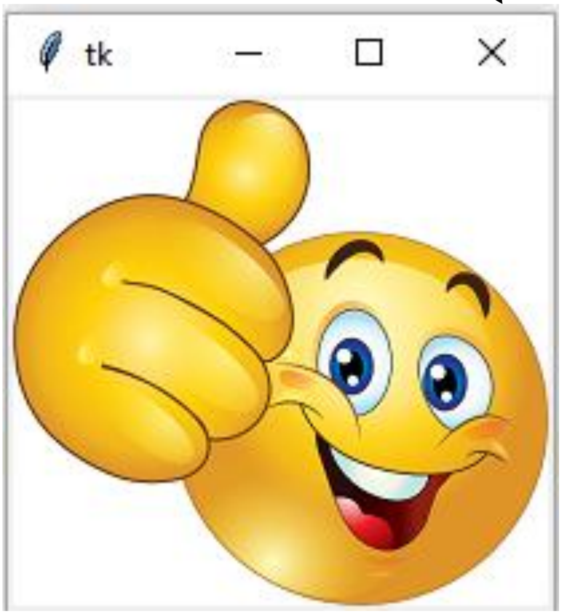
Protocols Handlers

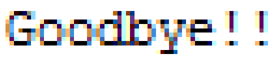
We want to call this method upon closing the window



```
def on_closing():
    print("Goodbye!!")
    # now we can close the window:
    root.destroy()
# creating the main window
root = tk.Tk()
root.protocol("WM_DELETE_WINDOW", on_closing)
img = tk.PhotoImage(file="smiley.png")
label_img = tk.Label(root, image=img)
label_img.pack()
root.mainloop()
```

Lets close the window



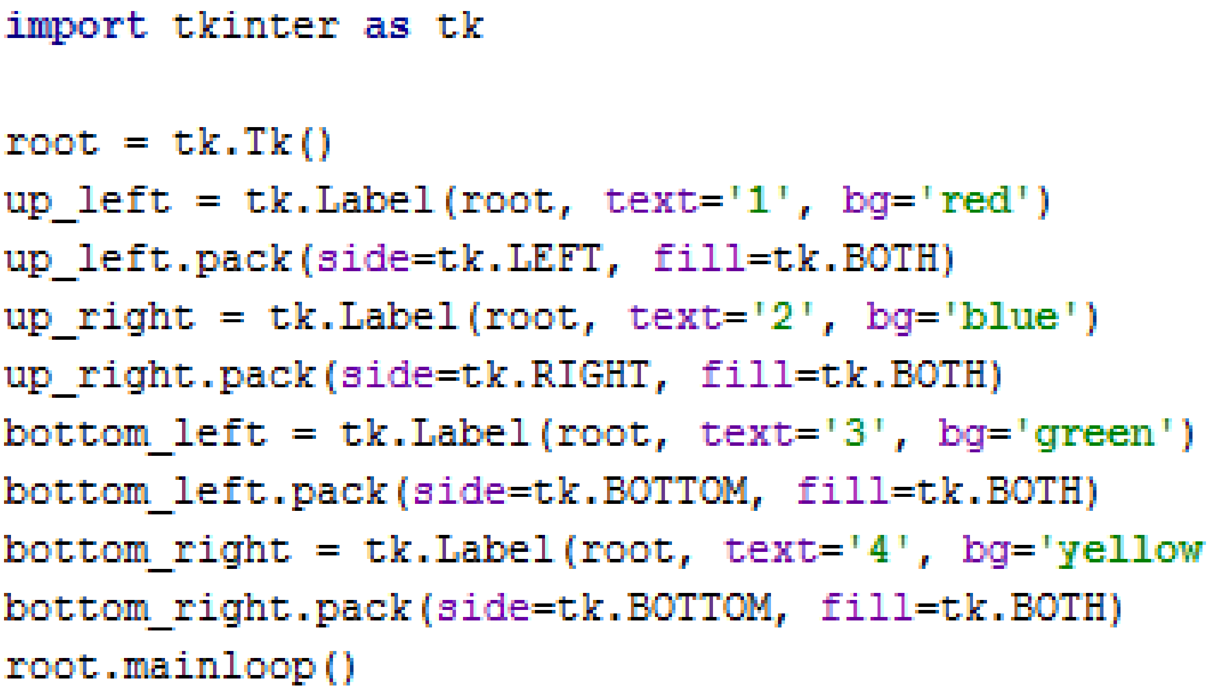


```
Goodbye!!
```

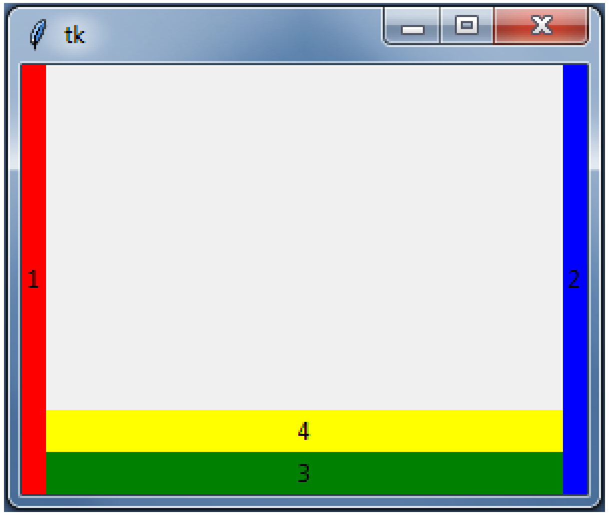
GUI in Python - Recap

- In Python, we interact with the GUI using widgets.
- Widgets are placed inside *other* widgets.
- The base widget is the **root**, initialized by Tk().
- The GUI starts “running” when we call root.mainloop().

Packing

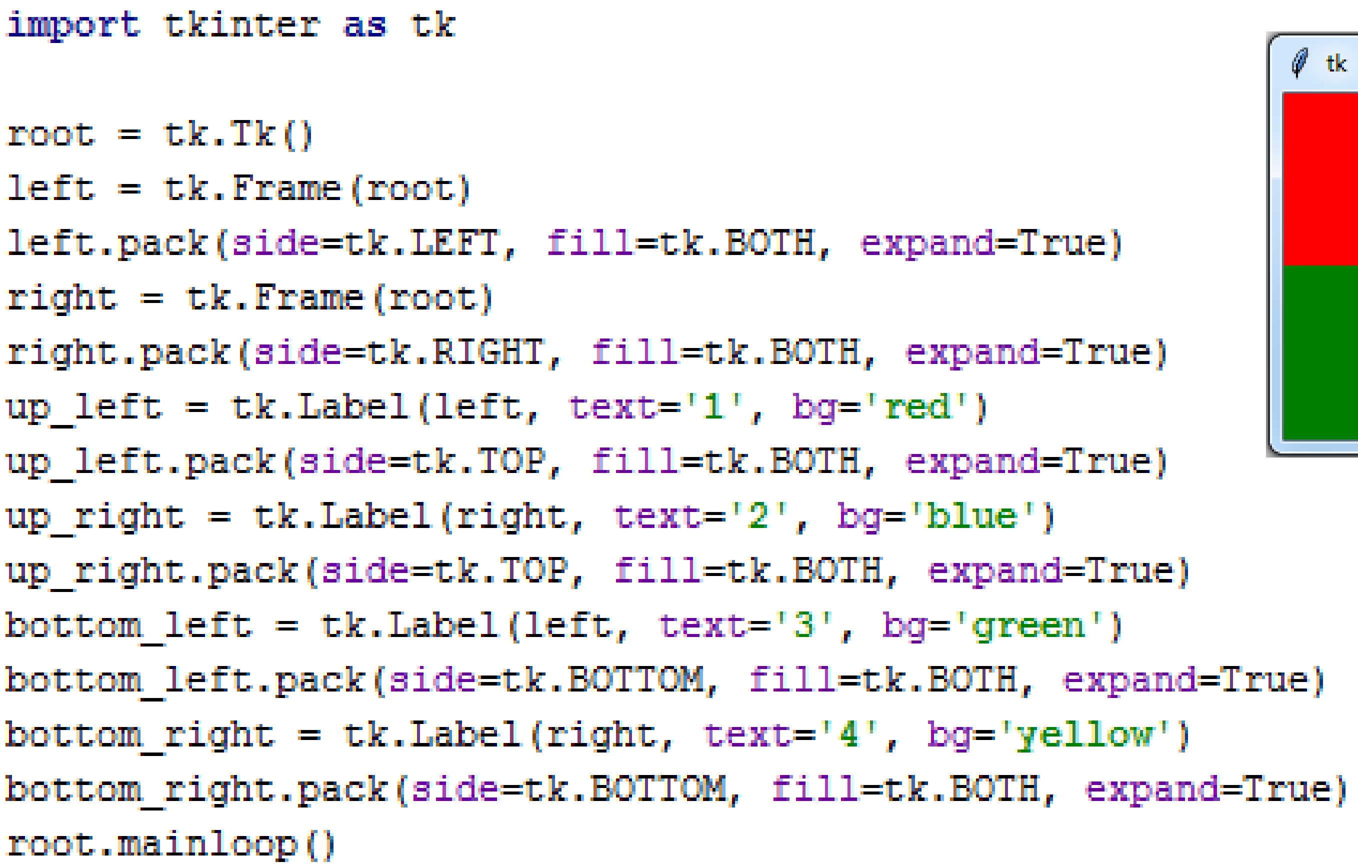


```
import tkinter as tk

root = tk.Tk()
up_left = tk.Label(root, text='1', bg='red')
up_left.pack(side=tk.LEFT, fill=tk.BOTH)
up_right = tk.Label(root, text='2', bg='blue')
up_right.pack(side=tk.RIGHT, fill=tk.BOTH)
bottom_left = tk.Label(root, text='3', bg='green')
bottom_left.pack(side=tk.BOTTOM, fill=tk.BOTH)
bottom_right = tk.Label(root, text='4', bg='yellow')
bottom_right.pack(side=tk.BOTTOM, fill=tk.BOTH)
root.mainloop()
```

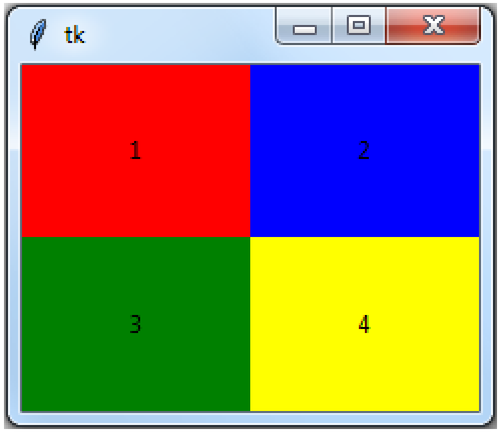


- packing is a simple way to put widgets in geometric order
- packing only assigns by top, bottom, left, right

Packing (cont.)

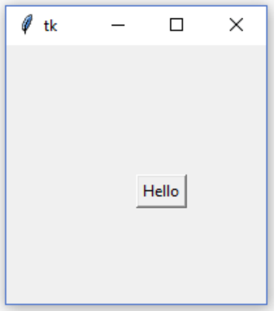


```
import tkinter as tk

root = tk.Tk()
left = tk.Frame(root)
left.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
right = tk.Frame(root)
right.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)
up_left = tk.Label(left, text='1', bg='red')
up_left.pack(side=tk.TOP, fill=tk.BOTH, expand=True)
up_right = tk.Label(right, text='2', bg='blue')
up_right.pack(side=tk.TOP, fill=tk.BOTH, expand=True)
bottom_left = tk.Label(left, text='3', bg='green')
bottom_left.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)
bottom_right = tk.Label(right, text='4', bg='yellow')
bottom_right.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=True)
root.mainloop()
```



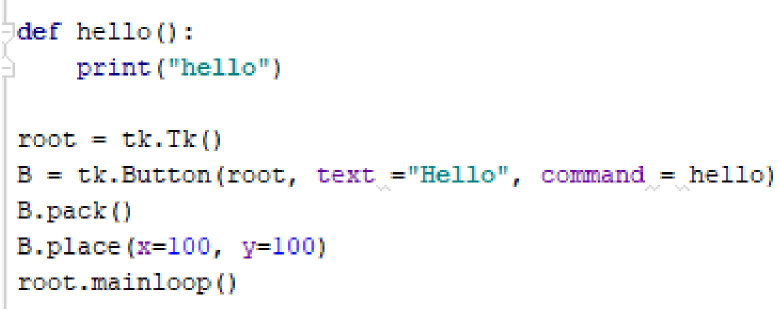
- use frames in order to sub-divide a screen

Place



- placing widgets in a specific position in the parent widget:
  - height, width − Height and width in pixels.
  - relheight, relwidth − Height and width as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
  - relx, rely − Horizontal and vertical offset as a float between 0.0 and 1.0, as a fraction of the height and width of the parent widget.
  - x, y − Horizontal and vertical offset in pixels.
  - And more..

- Fixed window size:
  - root.resizable(0,0)



```
def hello():
    print("hello")

root = tk.Tk()
B = tk.Button(root, text ="Hello", command = hello)
B.pack()
B.place(x=100, y=100)
root.mainloop()
```

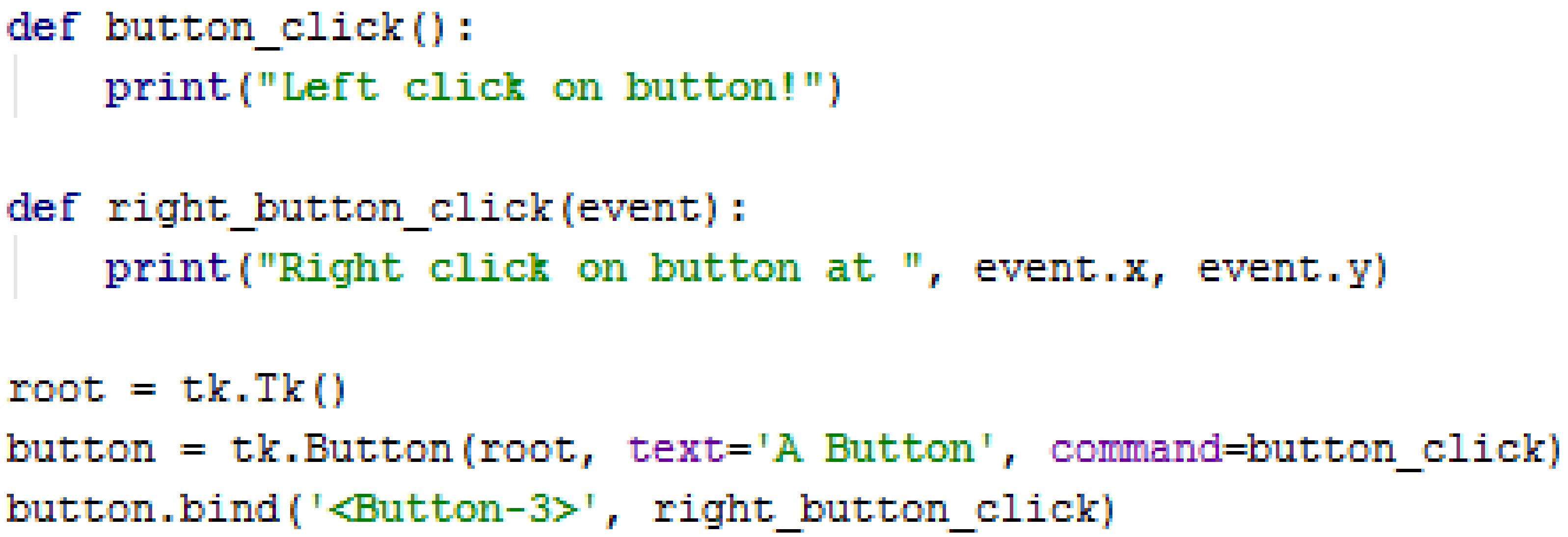
Widget *grid()* Method

- ***grid()*** method of the widget object: *widget.grid(row, column).*

- simply pour all the widgets into a single container widget, and use the grid manager to get them all where you want them.

Events Recap

- Some widgets (such as Button) can have a command associated with them
- General interactions with a widget (such as mouse over, mouse clicks, etc) can have a handler function *bound* to them

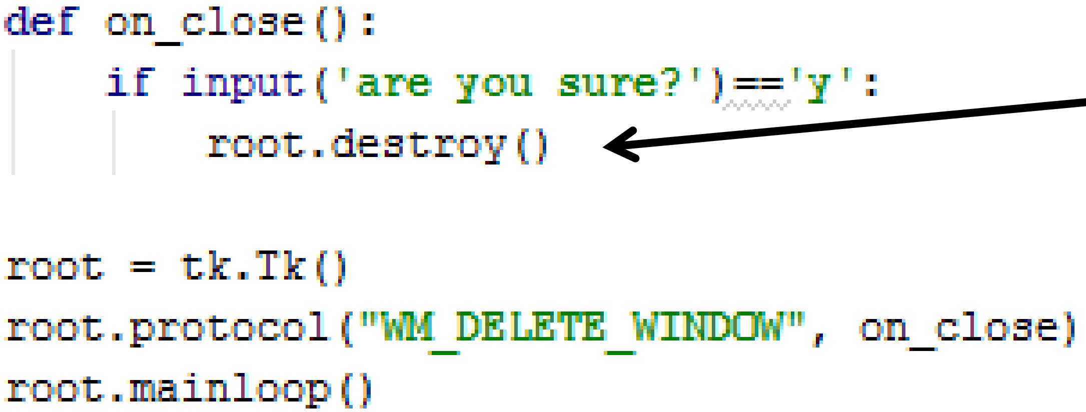


```
def button_click():
    print("Left click on button!")

def right_button_click(event):
    print("Right click on button at ", event.x, event.y)

root = tk.Tk()
button = tk.Button(root, text='A Button', command=button_click)
button.bind('<Button-3>', right_button_click)
```

- The handler function is automatically passed an *event* object

Events (cont.)

- We can also define what happens when we interact with the GUI *window*
- This is done similarly to *bind*, using the *protocol* method
- A useful protocol is **WM_DELETE_WINDOW**, for when the window ‘x’ button is pressed



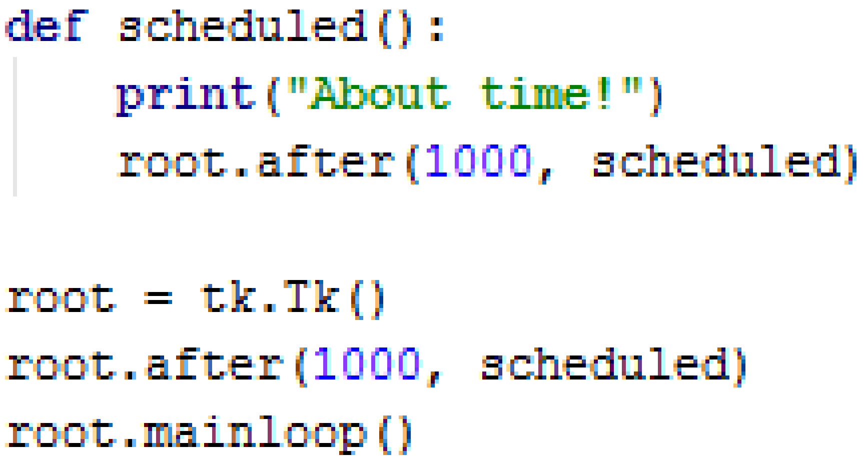
```
def on_close():
    if input('are you sure?')=='y':
        root.destroy()

root = tk.Tk()
root.protocol("WM_DELETE_WINDOW", on_close)
root.mainloop()
```

We need to explicitly destroy the window!

Events (cont.)

- Another way to generate an event is to schedule it using *after.*

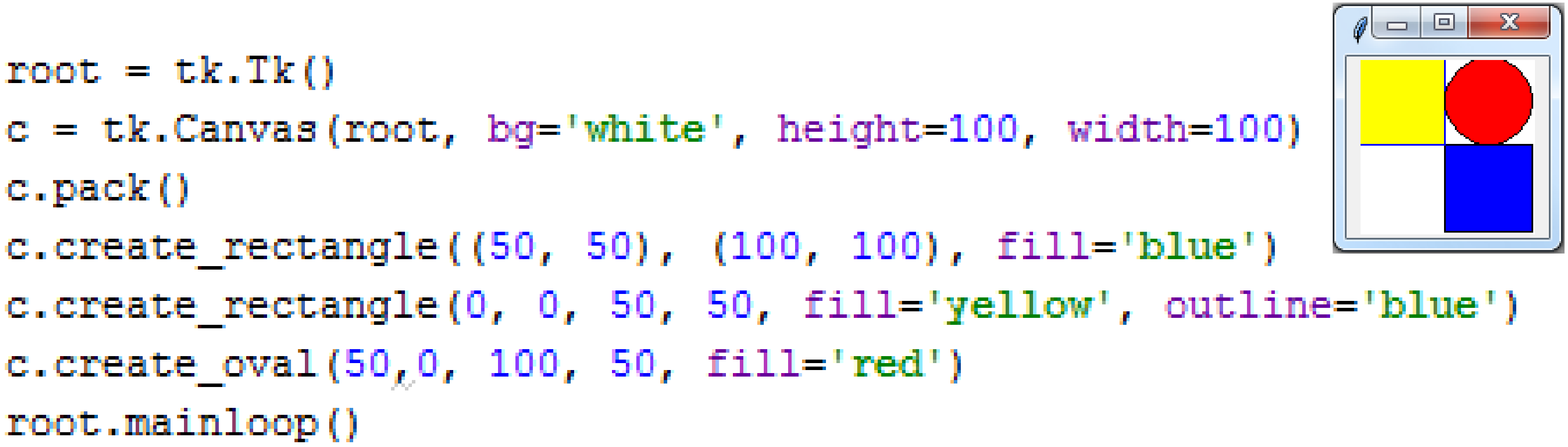
- Using *after* can also be used to generate sub-loops of mainloop!



```
def scheduled():
    print("About time!")
    root.after(1000, scheduled)

root = tk.Tk()
root.after(1000, scheduled)
root.mainloop()
```

The Canvas Widget

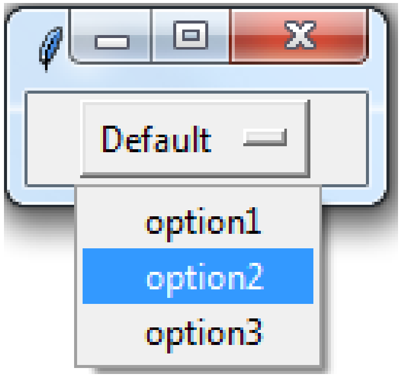
- The Canvas widget provides an area on which things can be displayed (“drawn”)
- Drawing is done via different *create* methods
- The create method returns an item id



```
root = tk.Tk()
c = tk.Canvas(root, bg='white', height=100, width=100)
c.pack()
c.create_rectangle((50, 50), (100, 100), fill='blue')
c.create_rectangle(0, 0, 50, 50, fill='yellow', outline='blue')
c.create_oval(50,0, 100, 50, fill='red')
root.mainloop()
```

The OptionMenu Widget

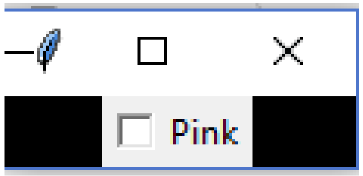
- Allows a selection from multiple options.
- Takes a *VariableClass* object and options.
- A VariableClass state can be queried by it’s get() method

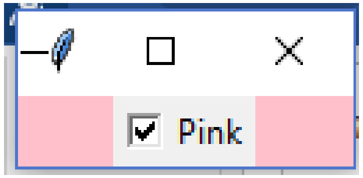


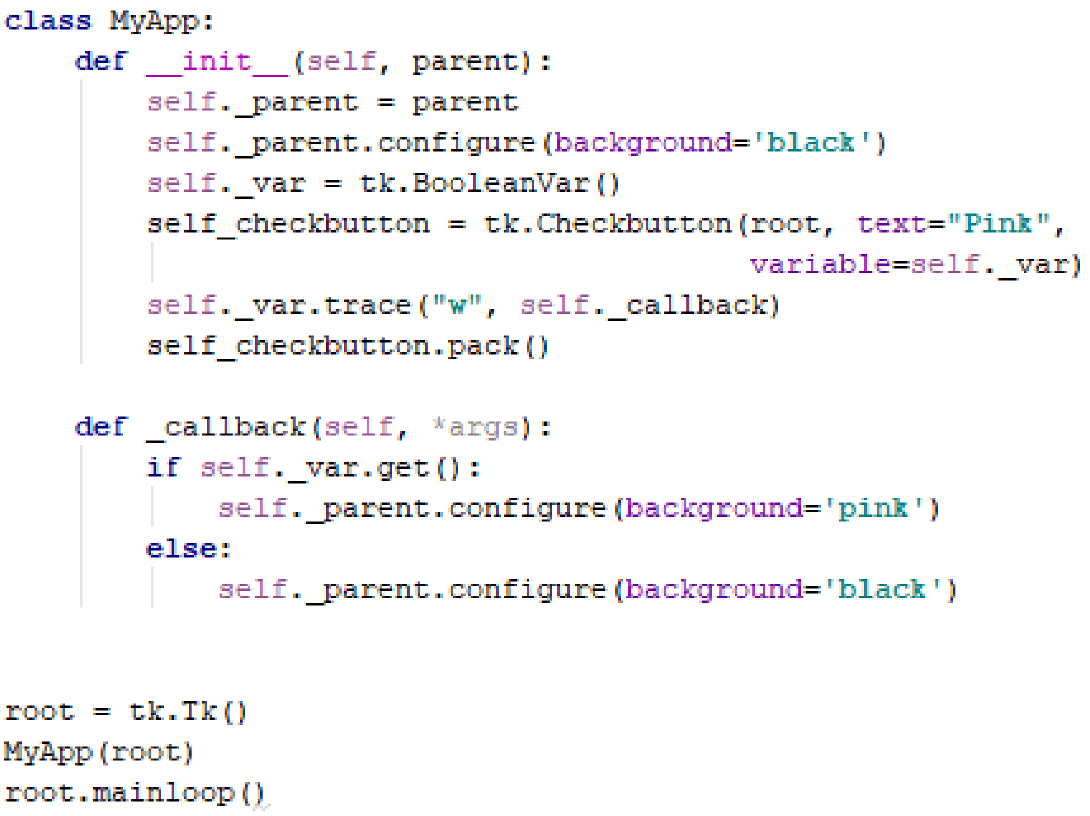
```
root = tk.Tk()
choice = tk.StringVar()
choice.set("Default")
options = tk.OptionMenu(root, choice, 'option1', 'option2', 'option3')
options.pack()
root.mainloop()
```

Variable Class

- Variable Class is a wrapper for variables
- Gives us a way to track changes to Python variables







```
class MyApp:
    def __init__(self, parent):
        self._parent = parent
        self._parent.configure(background='black')
        self._var = tk.BooleanVar()
        self_checkbutton = tk.Checkbutton(root, text="Pink",
                                          variable=self._var)
        self._var.trace("w", self._callback)
        self_checkbutton.pack()

    def _callback(self, *args):
        if self._var.get():
            self._parent.configure(background='pink')
        else:
            self._parent.configure(background='black')


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

Usfull Links

- ❑Events, Binding, and Protocol Handlers

http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm

- ❑Packing

http://effbot.org/tkinterbook/pack.htm

- ❑Tkinter widgets

http://effbot.org/tkinterbook/tkinter-index.htm#class-reference

- ❑A couple of tutorials

http://www.python-course.eu/python_tkinter.php

http://zetcode.com/gui/tkinter/

Lambda and Nested functions

Creating functions in Python

- Usually in Python we create functions by declaring them with **`def function_name`**.
- There is another way of creating functions in Python – without declaring the name of the function and using the special word **`lambda`**.
- Hence, such functions are called **lambda** functions or **anonymous** functions. Their syntax is:

```
lambda arguments: single line expression
```

Lambda Notation

```
>>> f1= lambda : print('hi')
>>> f1()
hi
>>> f1= lambda x: 2*x + 1
>>> f1(10)
21
>>> f2 = lambda x,y: x if x>y else y
>>> f2(3, 5)
5
>>> f3= lambda x,y,z: x in z and y in z
>>> f3('a','d',['b','a','c'])
False
>>> f3('a','c',['b','a','c'])
True
```

Lambda Notation Limitations

- Note: only **one** expression in the lambda body; Its value is always returned.

- The lambda expression must fit on one line!

Lambda example

- Recursive lambda:

```
is_divided_by_2 =
    lambda x: not x if x<2 else is_divided_by_2(x-2)
```

What does it do?

Lambda example

- Recursive lambda:

```
is_divided_by_2 =
    lambda x: not x if x<2 else is_divided_by_2(x-2)
```

- Equivalent to:

```
def is_divided_by_2(x):
   if x<2:
       return not x
    else:
      return is_divided_by_2(x-2)
# Since 1 is True and 0 is False, if x<2 (assuming non-negative
integer):
       if x is 1 (True) we want to return False
       if x is 0 (False) we want to return True
 So we can return "not x"
```

Nested functions

- Function inside function.

```
def f(n):
# Outer function

    def numbers_till_n():
    # inner function
        return [i for i in range(1, n+1)]

    return sum(numbers_till_n())

print(f(5))
```

Nested functions

- Function inside function.

```
def f(n):
# Outer function

    def numbers_till_n():# return [1, 2, 3, 4, 5]
    # inner function
        return [i for i in range(1, n+1)]

    return sum(numbers_till_n()) # return  15

print(f(5))
```

Nested functions

- Function inside function.

```
def f(n):
# Outer function

          def numbers_till_n():
          # inner function
                    return [i for i in range(1, n+1)]

          return sum(numbers_till_n())

print(f(5))
```

The *inner* function knows the parameters of the *outer* functions

(it knows ***n*** without getting it as a parameter)

- Let’s do this a little bit differently …

Nested functions

- Remember that functions are just like any other object, they can be the output of a function.

```
def f(n):
# Outer function

    def numbers_till_n():
    # inner function
        return [i for i in range(1, n+1)]

    return numbers_till_n

numbers_generator = f(5)
```

- print(type(numbers_generator)) → ?
  print(numbers_generator()) → ?
  print(sum(numbers_generator())) → ?

Nested functions

- f returns functions!

```python
def f(n):
# Outer function

  def numbers_till_n():
  # inner function
    return [i for i in range(1, n+1)]

  return numbers_till_n # This returns a function

numbers_generator = f(5)
```

- print(type(numbers_generator)) → <class 'function'>
  print(numbers_generator()) → [1, 2, 3, 4, 5]
  print(sum(numbers_generator())) → 15

Nested functions with parameters

- How will ***two*** outputs of `f` behave like?

```
ng1 = f(4)
ng2 = f(5)
print(sum(ng1))→?
print(sum(ng2))→?
```

```
def f(n):
# Outer function

    def numbers_till_n():
    # inner function
        return [i for i in range(1, n+1)]

    return numbers_till_n
```

Nested functions with parameters

- `f` is a generator of functions.
- Each returned function **remembers** its own *n*.

```
ng1 = f(4)
ng2 = f(5)
print(sum(ng1))→ 10
print(sum(ng2))→ 15
```

```
def f(n):
# Outer function

    def numbers_till_n():
    # inner function
        return [i for i in range(1, n+1)]

    return numbers_till_n
```

n is “**closed**” in the definition of **`numbers_till_n`**