

# GUIs

David Croft

Coventry University

*david.croft@coventry.ac.uk*

December 15, 2015

# Overview

- 1 GUIs
  - Hello World!
- 2 Layout
  - Containers
- 3 Events
  - Event arguments
  - Loops
- 4 Recap

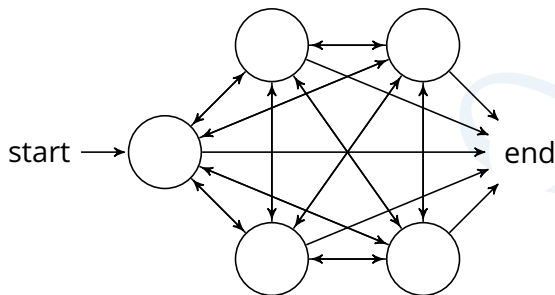
You're programs so far have followed a procedural pattern.

- Program is a series of steps.
- Moves through those steps in a predetermined pattern.
- Expects user input in a very specific order.



Going to look at event driven programming.

- Program reacts to events.
- Events have actions associated with them.
- Order and frequency of events is unpredictable.
- Does not have a predefined sequence of actions to perform.
- Does not have a predefined end.



What sort of applications would benefit from an event driven paradigm?

What sort of applications would benefit from an event driven paradigm?

- GUIs
- Control systems
- Embedded systems

## GUI events would include...

- Button presses
- Text entry
- Keyboard events
  - Pressing a key
  - Releasing a key
- Mouse events
  - Pressing a button
  - Releasing a button
  - Moving
  - Scrolling

### How to create a GUI.

- Wide range of different libraries available.
  - Depends on language and platform.
- Tkinter is the built-in Python default.



# Terminology

- Window
- Component/widget/element

```
import sys
from tkinter import *

def main():
    root = Tk()

    label = Label(root, text='Hello World!')
    label.pack()

    root.mainloop()

if __name__ == '__main__':
    sys.exit(main())
```

lec\_getting\_started.py

## Hello World!

```
import sys
from tkinter import *

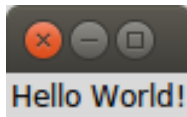
def main():
    root = Tk()

    label = Label(root, text='Hello World!')
    label.pack()

    root.mainloop()

if __name__ == '__main__':
    sys.exit(main())
```

lec\_getting\_started.py



GUI code should be structured as a class.

- Become clear later.

```
class Gui:
    def __init__(self, root):
        self.root = root

        self.label = Label(self.root, \
                           text='Hello World!')

        self.label.pack()

def main():
    root = Tk()
    gui = Gui(root)
    root.mainloop()
```

lec\_classes.py

So far we have seen how elements are added to window.

```
class Gui:
    def __init__(self, root):
        self.root = root

        for i in range(1,10):
            button = Button(self.root, text=i)
            button.pack()
```

lec\_layout.py

So far we have seen how elements are added to window.

```
class Gui:
    def __init__(self, root):
        self.root = root

        for i in range(1,10):
            button = Button(self.root, text=i)
            button.pack()
```

lec\_layout.py



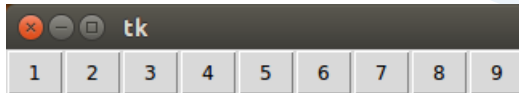
Can use the side parameter for `.pack()`.

- TOP (default).
- Also LEFT, RIGHT and BOTTOM.

```
class Gui:
    def __init__(self, root):
        self.root = root

    for i in range(1,10):
        button = Button(self.root, text=i)
        button.pack(side=LEFT)
```

lec\_layout2.py



## Use side to control layout?

```
class Gui:
    def __init__(self, root):
        self.root = root

        Button(self.root, text=1).pack(side=TOP)
        Button(self.root, text=2).pack(side=LEFT)
        Button(self.root, text=3).pack(side=LEFT)
        Button(self.root, text=4).pack(side=TOP)
        Button(self.root, text=5).pack(side=LEFT)
        Button(self.root, text=6).pack(side=LEFT)
        Button(self.root, text=7).pack(side=TOP)
        Button(self.root, text=8).pack(side=LEFT)
        Button(self.root, text=9).pack(side=LEFT)
```

lec\_layout3.py



## Use side to control layout?

```
class Gui:
    def __init__(self, root):
        self.root = root

        Button(self.root, text=1).pack(side=TOP)
        Button(self.root, text=2).pack(side=LEFT)
        Button(self.root, text=3).pack(side=LEFT)
        Button(self.root, text=4).pack(side=TOP)
        Button(self.root, text=5).pack(side=LEFT)
        Button(self.root, text=6).pack(side=LEFT)
        Button(self.root, text=7).pack(side=TOP)
        Button(self.root, text=8).pack(side=LEFT)
        Button(self.root, text=9).pack(side=LEFT)
```

lec\_layout3.py



# Containers

Need to learn about containers.

- Windows are containers.
  - Elements are 'contained' inside.
- Tkinter also has frames.
  - Special type of element.
  - Contains other elements.
- Group elements together using frames.
  - Can be visible/invisible.

```
class Gui:
    def __init__(self, root):
        self.root = root

        self.frame1 = Frame(self.root)
        self.frame1.pack()

        self.frame2 = Frame(self.root)
        self.frame2.pack()
```

```
Button(self.frame1, text=1).pack(side=LEFT)
Button(self.frame1, text=2).pack(side=LEFT)
Button(self.frame1, text=3).pack(side=LEFT)
```

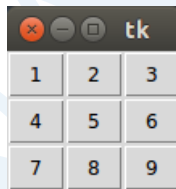
```
Button(self.frame3, text=7).pack(side=LEFT)
Button(self.frame3, text=8).pack(side=LEFT)
Button(self.frame3, text=9).pack(side=LEFT)
```

# Frames

```
class Gui:
    def __init__(self, root):
        self.root = root

        self.frame1 = Frame(self.root)
        self.frame1.pack()

        self.frame2 = Frame(self.root)
        self.frame2.pack()
```



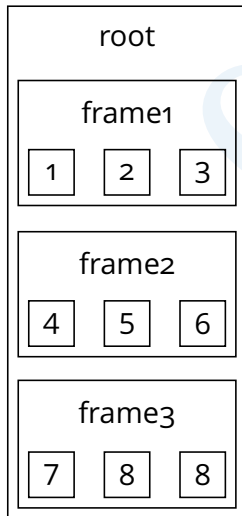
```
Button(self.frame1, text=1).pack(side=LEFT)
Button(self.frame1, text=2).pack(side=LEFT)
Button(self.frame1, text=3).pack(side=LEFT)
```

```
Button(self.frame3, text=7).pack(side=LEFT)
Button(self.frame3, text=8).pack(side=LEFT)
Button(self.frame3, text=9).pack(side=LEFT)
```

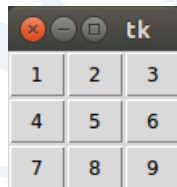
lec\_frames.py

So what's happening?

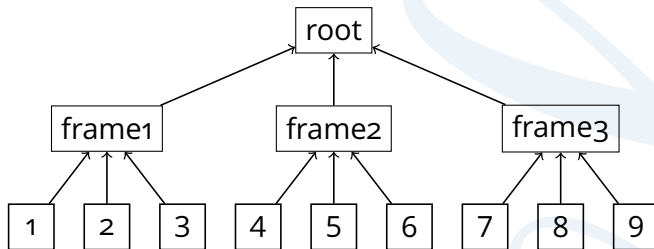
- Elements are nested in containers.
- Containers are nested in other containers.



## Nesting



# Hierarchical structure



How do we get our code to actually DO stuff?

- Using Python/Tkinter.
- Other languages/frameworks == different syntax.
  - Same concepts.
- Event handling.
  - Bind events to elements.

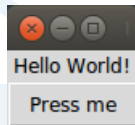
```
class Gui:
    def __init__(self, root):
        self.root = root

        self.label = Label(self.root, text='Hello World!')
        self.label.pack()

        self.button = Button(self.root, text='Press me')
        self.button.bind('<Button-1>', self.say_bye)
        self.button.pack()

    def say_bye(self, event):
        self.label.config(text='Bye!')
```

lec\_events.py





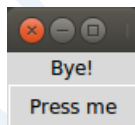
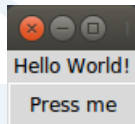
```
class Gui:
    def __init__(self, root):
        self.root = root

        self.label = Label(self.root, text='Hello World!')
        self.label.pack()

        self.button = Button(self.root, text='Press me')
        self.button.bind('<Button-1>', self.say_bye)
        self.button.pack()

    def say_bye(self, event):
        self.label.config(text='Bye!')
```

lec\_events.py



Callbacks are how we respond to events.

```
class Gui:
    def __init__(self, root):
        self.root = root

        self.label = Label(self.root, text='Hello World!')
        self.label.pack()

        self.button = Button(self.root, text='Press me')
        self.button.bind('<Button-1>', self.say_bye)
        self.button.pack()

    def say_bye(self, event):
        self.label.config(text='Bye!')
```

lec\_events.py

User → Event → Listener → Callback

User actions can trigger multiple events.

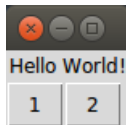
- I.e. clicking on button.
  - 1 Press LMB whilst pointer over button.
  - 2 Release LMB whilst pointer over button.
- Standard interaction code included in Tkinter.
  - Use `command` parameter.

```
class Gui:
    def __init__(self, root):
        self.root = root

        self.button = Button(self.root, text='Press me' , \
                             command=self.say_bye)
        self.button.pack()

    def say_bye(self):
        self.label.config(text='Bye!')
```

lec\_events2.py



```
class Gui:  
    def __init__(self, root):
```

```
        Button(self.root, text='1', \  
               command=self.pressed_1).pack(side=LEFT)  
        Button(self.root, text='2', \  
               command=self.pressed_2).pack(side=LEFT)
```

```
    def pressed_1(self):  
        self.label.config(text='Pressed 1')
```

```
    def pressed_2(self):  
        self.label.config(text='Pressed 2')
```

## Event arguments II

Much better to have one function.

- Function takes argument.
- Reuse of each button.

```
class Gui:
    def __init__(self, root):

        Button(self.root, text='1', \
               command=self.pressed_button(1)).pack(side=LEFT)
        Button(self.root, text='2', \
               command=self.pressed_button(2)).pack(side=LEFT)

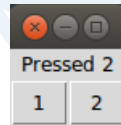
    def pressed_button(self, number):
        self.label.config(text='Pressed %d' % number)
```

lec\_event\_args2.py

## Event arguments II

Much better to have one function.

- Function takes argument.
- Reuse of each button.
- Doesn't work.
  - Calls function immediately.



```
class Gui:
    def __init__(self, root):

        Button(self.root, text='1', \
               command=self.pressed_button(1)).pack(side=LEFT)
        Button(self.root, text='2', \
               command=self.pressed_button(2)).pack(side=LEFT)

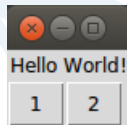
    def pressed_button(self, number):
        self.label.config(text='Pressed %d' % number)
```

lec\_event\_args2.py

## Event arguments III

`lambda` functions.

- Only calls function when button is pressed.



```
class Gui:
    def __init__(self, root):

        Button(self.root, text='1', \
               command=lambda: self.pressed_button(1)).pack(side=LEFT)
        Button(self.root, text='2', \
               command=lambda: self.pressed_button(2)).pack(side=LEFT)

    def pressed_button(self, number):
        self.label.config(text='Pressed %d' % number)
```

lec\_event\_args3.py

Already seen we can use create elements in loops.

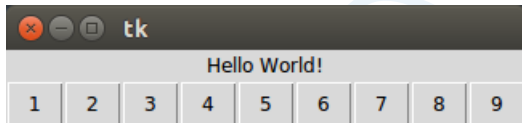
- Create lots of elements easily.
- How can we combine this with callback arguments?



## Loop arguments

lambda function in loop.

- What happens when any button is pressed?



```
class Gui:
    def __init__(self, root):

        for i in range(1,10):
            b = Button(self.root, text=i, \
                        command=lambda: self.pressed_button(i))
            b.pack(side=LEFT)

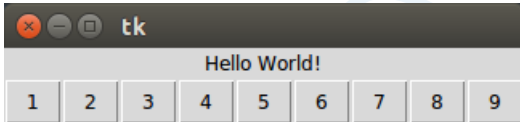
        def pressed_button(self, number):
            self.label.config(text='Pressed %d' % number)
```

lec\_loop\_args.py

lambda function in loop.

- What happens when any button is pressed?

- DEMO.



```
class Gui:
    def __init__(self, root):

        for i in range(1,10):
            b = Button(self.root, text=i, \
                       command=lambda: self.pressed_button(i))
            b.pack(side=LEFT)

        def pressed_button(self, number):
            self.label.config(text='Pressed %d' % number)
```

lec\_loop\_args.py

## Loop arguments II

```
for i in range(1,10):  
    b = Button(self.root, text=i, \  
               command=lambda: self.pressed_button(i))  
    b.pack(side=LEFT)
```

lec\_loop\_args.py

- Each button will call a lambda function when pressed.
- The lambda function will call `self.pressed_button(i)`.
- `pressed_button()` will change the label using the value of `i`.

## Loop arguments II

```
for i in range(1,10):  
    b = Button(self.root, text=i, \  
               command=lambda: self.pressed_button(i))  
    b.pack(side=LEFT)
```

lec\_loop\_args.py

- Each button will call a lambda function when pressed.
- The lambda function will call `self.pressed_button(i)`.
- `pressed_button()` will change the label using the value of `i`.
  - What is the value of `i`?

```
for i in range(1,10):  
    b = Button(self.root, text=i, \  
               command=lambda: self.pressed_button(i))  
    b.pack(side=LEFT)
```

lec\_loop\_args.py

- Each button will call a lambda function when pressed.
- The lambda function will call `self.pressed_button(i)`.
- `pressed_button()` will change the label using the value of `i`.
  - What is the value of `i`?
- It's whatever it was at the end of the loop, i.e. 9.
  - No matter what button we press, `i` is always 9.

## Loop arguments III

lambda arguments.

- The `lambda` function for each button copies the value of `i` right then.
- Uses that value when it runs in the future.

```
class Gui:
    def __init__(self, root):

        for i in range(1,10):
            b = Button(self.root, text=i, \
                        command=lambda n=i: self.pressed_button(n))
            b.pack(side=LEFT)

        def pressed_button(self, number):
            self.label.config(text='Pressed %d' % number)
```

lec\_loop\_args2.py

- GUIs are an example of event driven programming.
- GUI elements are arranged in containers.
- Containers can hold other containers.
- User actions generate events.
- Callbacks are functions that are run in response to events.

# The End