122COM: GUIs

David Croft

January 20, 2017

Abstract

This week we are going to be looking at Graphical User Interfaces (GUIs), in terms of programming paradigms and practical implementations. This week will be taught in Python only.

1 Introduction

All of our programs so far have interacted using a Command Line Interface (CLI). CLIs have many advantages over GUIs but are generally less intuitive.

As with most programming languages, Python has a huge range of different GUI creation libraries. Tkinter is the default module included in the Python standard library. Other modules have more advanced features but the general GUI programming concepts are the same across them all.

The source code files for this lab are available at:

```
https://github.com/dscroft/122COM_guis.git
```

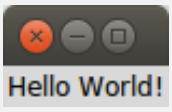
```
git@github.com:dscroft/122COM_guis.git
```

1.1 Getting started

As is traditional we're going to start with a "Hello World!" example.

Pre-lab work:

1. Run pre_getting_started.py
2. When run a window similar to the one below should appear.



Having run the code let's look at what each line is doing.

Creates the top level window, this is the window with the title bar and minimise/maximise/close buttons.

```
root = Tk()
```

Creates a text label, the first argument tells the Label that its parent is the root window. The second argument is optional and controls the text that will appear in the label.

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

Packs the label into the root window. To put it another way the label is placed inside the root window. The order in which elements are packed affects their position on screen. It is, therefore, important that elements are packed in the correct order.

```
label.pack()
```

Runs the tkinter event loop. This is what actually draws our window and label to the screen. In more sophisticated GUIs this would also be watching out for user events. Your program will remain in this loop until you press the window close button.

```
root.mainloop()
```

1.2 Classes

Unless you are writing a very small program you should be wrapping your GUI code in a class. This will make it easier to keep track of all the GUI objects and associated functions. pre_classes.py shows how to restructure our previous program as a class.

All GUI classes will need an `__init__` function in which the various GUI elements are created, configured and packed. Make sure that GUI elements are created as class members and not local variables (use `self`).

1.2.1 Event-driven programming

The key feature of GUIs is that they respond to events. Events in this case being user actions in the GUI. Every time that the user clicks, types or drags generates an events, by listening to these events we can get our code to react to the user actions.

So far we've created GUIs that don't do anything. If we want to make our GUIs able to react to user commands then we first need to learn about buttons and more importantly need to learn about callbacks.

Pre-lab work:

3. Run pre_events.py.
4. Try pressing the button.

Having run the code let's look at what each line is doing.

Creates a button and sets its text to be 'Press me'. Of more interest is the `command` parameter, this is controls which function should be run when the button is pressed. Passing a function to be called later like this is known as a callback. When the button is pressed it will "call back" this function and execute it.

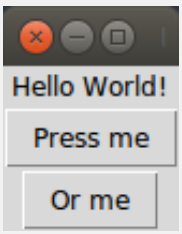
```
self.button = Button(self.root,
                     text='Press me',
                     command=self.say_bye)
```

This is the function that gets called when we press the button. All it does is to change the text of the label in our GUI. The `.config()` function is used to change the parameters of GUI objects.

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

Pre-lab work:

5. Add a second button to the program that can be used to change the label back to saying 'Hello World!'.
6. The result should look something like the GUI shown below but it doesn't matter if it is not exactly the same.



2 Layout

So far our GUI elements have appeared stacked one on top of another. We've been able to control what order they are in by controlling when we pack them but that's it. In order to have more control over the layout of our elements we need to learn about Frame elements and the parameters available for the `pack()` function.

2.0.1 `side`

The pad function has several parameters that alter it's behaviour. The first of these is `side`, which takes the values TOP (default), BOTTOM, LEFT and RIGHT.

Lab work:

7. Run lab_side.py.
8. See how the `side` parameter affects the layout.

2.0.2 `expand` and `fill`

The other two important parameters are `expand` and `fill`. These two work together but can be confusing. By default each GUI object takes up the minimum amount of space necessary, in the case of Buttons or Labels this is the amount of space needed to fit the label text, in the case of Frames and Windows this is the space needed to fit all the elements contained within it.

`fill` means that the element will take up all the space that it has been given, it takes the values BOTH, X and Y which controls in which directions it will do this.

`expand` means that the element will try and take any free space, it takes the value True or False (default).

Lab work:

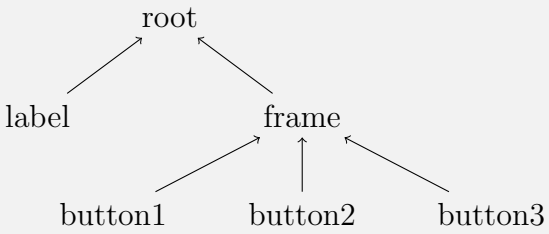
9. Run lab_expand_fill.py.
10. See how resizing the window affects the GUI layout.
11. Experiment with the `side`, `fill` and `expand` parameters until you are confident you understand what each of them does.

2.1 Frames

Layout is important as it controls how our GUI will react to window resizing. One of the most useful tools in controlling your GUI layouts are frames. Frames are containers for other elements, they can be used both decoratively and invisibly to group elements together.

Lab work:

12. Run lab_frames.py.
    - Notice that the buttons are surrounded by a border, this is the frame.
13. See how the lab_frames.py code relates to the layout hierarchy shown below.



14. Expand lab_frames.py so that it resembles the following.

3 Callback arguments

We've previously seen how to use callbacks to get our code to react to user generated events[1].

If we have a large number of different buttons each performing a different action then we can bind each of those buttons to a different callback function. If we have a large number of buttons that do very similar actions then it is wasteful to create lots of separate callback functions. What we need is to be able to pass additional information to the callback functions so then can react differently depending on which button was pressed.

Lab work:

15. Modify your code to include the `button_press` function shown in lab_callback_args.py.
16. Try running your code.
    - What happens?

[1] Button presses.

The problem with the code in lab_callback_args.py is that Python not telling the button to callback `button_press()` with the various different arguments. Instead Python is calling `button_press()` taking the results and telling the button to callback those results instead.

This means that `button_press()` gets called once each time we create a button (which we don't want) but not when we press the buttons (which we do want).

3.1 `lambda` functions

To solve this problem we need to use `lambda` functions. There are two ways to create functions in Python, `def` and `lambda`. `def` is used when we want to create a named function that we can call from elsewhere in our code (e.g. `def main()`, `def button_press()` etc). `lambda` is used when we want to create a small function that will only be used in this one location.

In this case we're going to create a `lambda` function that calls `button_press()` with an argument. We attach the `lambda` function to the button as a callback and when the button is pressed the `lambda` function will call `button_press()` on behalf of the button.

Button → `lambda` → `button_press()`

The `lambda` syntax is very simple. First we tell Python that this is a `lambda` function. Next we create any variables that we are going to need and set their values. Finally we have our contents of our `lambda` function.

```
lambda: print("Hello!")
lambda a=1, b=2: a+b
lambda a=42: range(0,a,2)
lambda a=1: self.button_press(a)}
```

The whole line would therefore look something like this:

```
self.button1 = Button(self.frame, text='1',
                      command=lambda a=1: self.button_press(a))
```

Lab work:

17. Finish your calculator program so that you can actually use it to add numbers together.

Extended work:

18. Create a full functional calculator (i.e. /, -, and * buttons).
    - Can you get it working with decimals?
19. Pack is not the only Tkinker layout manager. Investigate the grid manager and see if you can re-create the calculator program using it.
20. Create a simple text editor.
    - Save and load from files.
    - You will need to investigate some of the other Tkinter objects.
21. Create a Connect Four program.
    - `https://en.wikipedia.org/wiki/Connect_Four`
22. Create a Conway's Game of Life program.
    - Will be seriously impressed if anyone does this.
    - You'll need to figure out how to create and place objects using iteration. Do **NOT** try and create several hundred boxes by hand.
    - It may be helpful to write a CLI version first and then adapt it.