Introduction to GUI programming with `tkinter`

We have previously seen how to write text-only programs which have a *command-line interface*, or CLI. Now we will briefly look at creating a program with a *graphical user interface*, or GUI. In this chapter we will use `tkinter`, a module in the Python standard library which serves as an interface to Tk, a simple *toolkit*. There are many other toolkits available, but they often vary across platforms. If you learn the basics of `tkinter`, you should see many similarities should you try to use a different toolkit.

We will see how to make a simple GUI which handles user input and output. GUIs often use a form of OO programming which we call *event-driven*: the program responds to *events*, which are actions that a user takes.

Note

in some Linux distributions, like Ubuntu and Debian, the `tkinter` module is packaged separately to the rest of Python, and must be installed separately.

Event-driven programming

Anything that happens in a user interface is an *event*. We say that an event is *fired* whenever the user does something – for example, clicks on a button or types a keyboard shortcut. Some events could also be triggered by occurrences which are not controlled by the user – for example, a background task might complete, or a network connection might be established or lost.

Our application needs to monitor, or *listen* for, all the events that we find interesting, and respond to them in some way if they occur. To do this, we usually associate certain functions with particular events. We call a function which performs an action in response to an event an *event handler* – we *bind* handlers to events.

`tkinter` basics

`tkinter` provides us with a variety of common GUI elements which we can use to build our interface – such as buttons, menus and various kinds of entry fields and display areas. We call these elements *widgets*. We are going to construct a *tree* of widgets for our GUI – each widget will have a parent widget, all the way up to the *root window* of our application. For example, a button or a text field needs to be *inside* some kind of containing window.

The widget classes provide us with a lot of default functionality. They have methods for configuring the GUI's appearance – for example, arranging the elements according to some kind of *layout* – and for handling various kinds of user-driven events. Once we have constructed the backbone of our GUI, we will need to customise it by integrating it with our internal application class.

Our first GUI will be a window with a label and two buttons:

```
from tkinter import Tk, Label, Button

class MyFirstGUI:
    def __init__(self, master):
        self.master = master
        master.title("A simple GUI")

        self.label = Label(master, text="This is our first GUI!")
        self.label.pack()

        self.greet_button = Button(master, text="Greet", command=self.greet)
        self.greet_button.pack()

        self.close_button = Button(master, text="Close", command=master.quit)
        self.close_button.pack()

    def greet(self):
        print("Greetings!")

root = Tk()
my_gui = MyFirstGUI(root)
root.mainloop()
```

Try executing this code for yourself. You should be able to see a window with a title, a text label and two buttons – one which prints a message in the console, and one which closes the window. The window should have all the normal properties of any other window you encounter in your window manager – you are probably able to drag it around by the titlebar, resize it by dragging the frame, and maximise, minimise or close it using buttons on the titlebar.

Note

The *window manager* is the part of your operating system which handles windows. All the widgets inside a window, like buttons and other controls, may look different in every GUI toolkit, but the way that the window frames and title bars look and behave is determined by your window manager and should always stay the same.

We are using three widgets: `Tk` is the class which we use to create the *root* window – the main window of our application. Our application should only have one root, but it is possible for us to create other windows which are separate from the main window.

`Button` and `Label` should be self-explanatory. Each of them has a parent widget, which we pass in as the first parameter to the constructor – we have put the label and both buttons inside the main window, so they are the main window's children in the tree. We use the `pack` method on each widget to position it inside its parent – we will learn about different kinds of layout later.

All three of these widgets can display text (we could also make them display images). The label is a static element – it doesn't *do* anything by default; it just displays something. Buttons, however, are designed to cause something to happen when they are clicked. We have used the `command` keyword parameter when constructing each button to specify the function which should handle each button's click events – both of these functions are object methods.

We didn't have to write any code to make the buttons fire click events or to bind the methods to them explicitly. That functionality is already built into the button objects – we only had to provide the handlers. We also didn't have to write our own function for closing the window, because there is already one defined as a method on the window object. We did, however, write our own method for printing a message to the console.

There are many ways in which we could organise our application class. In this example, our class doesn't inherit from any `tkinter` objects – we use *composition* to associate our tree of widgets with our class. We could also use *inheritance* to extend one of the widgets in the tree with our custom functions.

`root.mainloop()` is a method on the main window which we execute when we want to run our application. This method will loop forever, waiting for events from the user, until the user exits the program – either by closing the window, or by terminating the program with

a keyboard interrupt in the console.

Widget classes

There are many different widget classes built into `tkinter` – they should be familiar to you from other GUIs:

- A `Frame` is a container widget which is placed inside a window, which can have its own border and background – it is used to group related widgets together in an application's layout.
- `Toplevel` is a container widget which is displayed as a separate window.
- `Canvas` is a widget for drawing graphics. In advanced usage, it can also be used to create custom widgets – because we can draw anything we like inside it, and make it interactive.
- `Text` displays formatted text, which can be editable and can have embedded images.
- A `Button` usually maps directly onto a user action – when the user clicks on a button, something should happen.
- A `Label` is a simple widget which displays a short piece of text or an image, but usually isn't interactive.
- A `Message` is similar to a `Label`, but is designed for longer bodies of text which need to be wrapped.
- A `Scrollbar` allows the user to scroll through content which is too large to be visible all at once.
- `Checkbutton`, `Radiobutton`, `Listbox`, `Entry` and `Scale` are different kinds of input widgets – they allow the user to enter information into the program.
- `Menu` and `Menubutton` are used to create pull-down menus.

Layout options

The GUI in the previous example has a relatively simple layout: we arranged the three widgets in a single column inside the window. To do this, we used the `pack` method, which is one of the three different *geometry managers* available in `tkinter`. We have to use one of the available geometry managers to specify a position for each of our widgets, otherwise the widget will not appear in our window.

By default, `pack` arranges widgets vertically inside their parent container, from the top down, but we can change the alignment to the bottom, left or right by using the optional `side` parameter. We can mix different alignments in the same container, but this may not work very well for complex layouts. It should work reasonably well in our simple case, however:

```
from tkinter import LEFT, RIGHT

# (...)

self.label.pack()
self.greet_button.pack(side=LEFT)
self.close_button.pack(side=RIGHT)
```

We can create quite complicated layouts with `pack` by grouping widgets together in frames and aligning the groups to our liking – but we can avoid a lot of this complexity by using the `grid` method instead. It allows us to position widgets in a more flexible way, using a *grid layout*. This is the geometry manager recommended for complex interfaces:

```
from tkinter import W

# (...)

self.label.grid(columnspan=2, sticky=W)
self.greet_button.grid(row=1)
self.close_button.grid(row=1, column=1)
```

We place each widget in a cell inside a table by specifying a row and a column – the default row is the first available empty row, and the default column is `0`.

If a widget is smaller than its cell, we can customise how it is aligned using the `sticky` parameter – the possible values are the cardinal directions (`N`, `S`, `E` and `W`), which we can combine through addition. By default, the widget is centered both vertically and horizontally, but we can make it *stick* to a particular side by including it in the `sticky` parameter. For example, `sticky=W` will cause the widget to be left-aligned horizontally, and `sticky=W+E` will cause it to be stretched to fill the whole cell horizontally. We can also specify corners using `NE`, `SW`, etc..

To make a widget span multiple columns or rows, we can use the `columnspan` and `rowspan` options – in the example above, we have made the label span two columns so that it takes up the same space horizontally as both of the buttons underneath it.

Note

Never use both `pack` and `grid` inside the same window. The algorithms which they use to calculate widget positions are not compatible with each other, and your program will hang forever as `tkinter` tries unsuccessfully to create a widget layout which satisfies both of them.

The third geometry manager is `place`, which allows us to provide explicit sizes and positions for widgets. It is seldom a good idea to use this method for ordinary GUIs – it's far too inflexible and time consuming to specify an absolute position for every element. There are some specialised cases, however, in which it can come in useful.

Custom events

So far we have only bound event handlers to events which are defined in `tkinter` by default – the `Button` class already knows about button clicks, since clicking is an expected part of normal button behaviour. We are not restricted to these particular events, however – we can make widgets listen for other events and bind handlers to them, using the `bind` method which we can find on every widget class.

Events are uniquely identified by a sequence name in string format – the format is described by a mini-language which is not specific to Python. Here are a few examples of common events:

- `"<Button-1>"`, `"<Button-2>"` and `"<Button-3>"` are events which signal that a particular mouse button has been pressed while the mouse cursor is positioned over the widget in question. *Button 1* is the left mouse button, *Button 3* is the right, and *Button 2* the middle button – but remember that not all mice have a middle button.
- `"<ButtonRelease-1>"` indicates that the left button has been released.
- `"<B1-Motion>"` indicates that the mouse was moved while the left button was pressed (we can use *B2* or *B3* for the other buttons).
- `"<Enter>"` and `"<Leave>"` tell us that the mouse curson has entered or left the widget.
- `"<Key>"` means that any key on the keyboard was pressed. We can also listen for

specific key presses, for example `"<Return>"` (the *enter* key), or combinations like `"<Shift-Up>"` (*shift-up-arrow*). Key presses of most printable characters are expressed as the bare characters, without brackets – for example, the letter `a` is just `"a"`.

- `"<Configure>"` means that the widget has changed size.

We can now extend our simple example to make the label interactive – let us make the label text cycle through a sequence of messages whenever it is clicked:

```
from tkinter import Tk, Label, Button, StringVar

class MyFirstGUI:
    LABEL_TEXT = [
        "This is our first GUI!",
        "Actually, this is our second GUI.",
        "We made it more interesting...",
        "...by making this label interactive.",
        "Go on, click on it again.",
    ]
    def __init__(self, master):
        self.master = master
        master.title("A simple GUI")

        self.label_index = 0
        self.label_text = StringVar()
        self.label_text.set(self.LABEL_TEXT[self.label_index])
        self.label = Label(master, textvariable=self.label_text)
        self.label.bind("<Button-1>", self.cycle_label_text)
        self.label.pack()

        self.greet_button = Button(master, text="Greet", command=self.greet)
        self.greet_button.pack()

        self.close_button = Button(master, text="Close", command=master.quit)
        self.close_button.pack()

    def greet(self):
        print("Greetings!")

    def cycle_label_text(self, event):
        self.label_index += 1
        self.label_index %= len(self.LABEL_TEXT) # wrap around
        self.label_text.set(self.LABEL_TEXT[self.label_index])

root = Tk()
my_gui = MyFirstGUI(root)
root.mainloop()
```

Updating a label's text is a little convoluted – we can't simply update the text using a normal Python string. Instead, we have to provide the label with a special `tkinter` string variable object, and set a new value on the object whenever we want the text in the label to change.

We have defined a handler which cycles to the next text string in the sequence, and used the `bind` method of the label to bind our new handler to left clicks on the label. It is important to note that this handler takes an additional parameter – an event object, which contains some information about the event. We could use the same handler for many different events (for example, a few similar events which happen on different widgets), and use this parameter to distinguish between them. Since in this case we are only using our handler for one kind of event, we will simply ignore the event parameter.

Putting it all together

Now we can use all this information to create a simple calculator. We will allow the user to enter a number in a text field, and either add it to or subtract it from a running total, which we will display. We will also allow the user to reset the total:

```
from tkinter import Tk, Label, Button, Entry, IntVar, END, W, E

class Calculator:

    def __init__(self, master):
        self.master = master
        master.title("Calculator")

        self.total = 0
        self.entered_number = 0

        self.total_label_text = IntVar()
        self.total_label_text.set(self.total)
        self.total_label = Label(master, textvariable=self.total_label_text)

        self.label = Label(master, text="Total:")

        vcmd = master.register(self.validate) # we have to wrap the command
        self.entry = Entry(master, validate="key", validatecommand=(vcmd, '%P'))

        self.add_button = Button(master, text="+", command=lambda: self.update("add"))
        self.subtract_button = Button(master, text="-", command=lambda:
self.update("subtract"))
        self.reset_button = Button(master, text="Reset", command=lambda:
self.update("reset"))
```

```
        # LAYOUT

        self.label.grid(row=0, column=0, sticky=W)
        self.total_label.grid(row=0, column=1, columnspan=2, sticky=E)

        self.entry.grid(row=1, column=0, columnspan=3, sticky=W+E)

        self.add_button.grid(row=2, column=0)
        self.subtract_button.grid(row=2, column=1)
        self.reset_button.grid(row=2, column=2, sticky=W+E)

    def validate(self, new_text):
        if not new_text: # the field is being cleared
            self.entered_number = 0
            return True

        try:
            self.entered_number = int(new_text)
            return True
        except ValueError:
            return False

    def update(self, method):
        if method == "add":
            self.total += self.entered_number
        elif method == "subtract":
            self.total -= self.entered_number
        else: # reset
            self.total = 0

        self.total_label_text.set(self.total)
        self.entry.delete(0, END)

root = Tk()
my_gui = Calculator(root)
root.mainloop()
```

We have defined two methods on our class: the first is used to validate the contents of the entry field, and the second is used to update our total.

Validating text entry

Our `validate` method checks that the contents of the entry field are a valid integer: whenever the user types something inside the field, the contents will only change if the new value is a valid number. We have also added a special exception for when the value is nothing, so that the field can be cleared (by the user, or by us). Whenever the value of the

field changes, we store the integer value of the contents in `self.entered_number`. We have to perform the conversion at this point anyway to see if it's a valid integer – if we store the value now, we won't have to do the conversion again when it's time to update the total.

How do we connect this validation function up to our entry field? We use the `validatecommand` parameter. The function we use for this command must return `True` if the entry's value is allowed to change and `False` otherwise, and it *must* be wrapped using a widget's `register` method (we have used this method on the window object).

We can also optionally specify arguments which must be passed to the function – to do this, we pass in a tuple containing the function and a series of strings which contain special codes. When the function is called, these codes will be replaced by different pieces of information about the change which is being made to the entry value. In our example, we only care about one piece of information: what the new value is going to be. The code string for this is `'%P'`, so we add it into the tuple.

Another optional parameter which is passed to `Entry` is `validate`, which specifies when validation should occur. the default value is `'none'` (a string value, not Python's `None`!), which means that no validation should be done. We have selected `'key'`, which will cause the entry to be validated whenever the user types something inside it – but it will also be triggered when we clear the entry from inside our `update` method.

Updating the total

We have written a single handler for updating the total, because what we have to do in all three cases is very similar. However, the way that we update the value depends on which button was pressed – that's why our handler needs a parameter. This presents us with a problem – unfortunately, `tkinter` has no option for specifying parameters to be passed to button commands (or *callbacks*). We can solve the problem by wrapping the handler in three different functions, each of which calls the handler with a different parameter when it is called. We have used lambda functions to create these wrappers because they are so simple.

Inside the handler, we first update our running total using the integer value of the entry field (which is calculated and stored inside the `validate` method – note that we initialise it to zero in the `__init__` method, so it's safe for the user to press the buttons without typing anything). We know how to update the total because of the parameter which is passed into the handler.

Once we have updated the total, we need to update the text displayed by the label to show the new total – we do this by setting the new value on the `IntVar` linked to the label as its text variable. This works just like the `StringVar` in the previous example, except that an `IntVar` is used with integer values, not strings.

Finally, once we have used the number the user entered, we clear it from the entry field using the entry widget's delete method by deleting all the characters from the first index (zero) to the end (`END` is a constant defined by `tkinter`). We should also clear our internal value for the last number to be entered – fortunately, our deletion triggers the validation method, which already resets this number to zero if the entry is cleared.

Exercise 1

1. Explain why we needed to use lambdas to wrap the function calls in the last example. Rewrite the button definitions to replace the lambdas with functions which have been written out in full.
2. Create a GUI for the guessing game from exercise 3 in the previous chapter.

Answers to exercises

Answer to exercise 1

1. The lambdas are necessary because we need to pass *functions* into the button constructors, which the button objects will be able to call later. If we used the bare function calls, we would be calling the functions and passing their return values (in this case, `None`) into the constructors. Here is an example of how we can rewrite this code fragment with full function definitions:

```
def update_add():
    self.update("add")

def update_subtract():
    self.update("subtract")

def update_reset():
    self.update("reset")

self.add_button = Button(master, text="+", command=update_add)
self.subtract_button = Button(master, text="-", command=update_subtract)
self.reset_button = Button(master, text="Reset", command=update_reset)
```

2. Here is an example program:

```
import random
from tkinter import Tk, Label, Button, Entry, StringVar, DISABLED, NORMAL, END, W, E

class GuessingGame:
    def __init__(self, master):
        self.master = master
        master.title("Guessing Game")

        self.secret_number = random.randint(1, 100)
        self.guess = None
        self.num_guesses = 0

        self.message = "Guess a number from 1 to 100"
        self.label_text = StringVar()
        self.label_text.set(self.message)
        self.label = Label(master, textvariable=self.label_text)

        vcmd = master.register(self.validate) # we have to wrap the command
        self.entry = Entry(master, validate="key", validatecommand=(vcmd, '%P'))

        self.guess_button = Button(master, text="Guess", command=self.guess_number)
        self.reset_button = Button(master, text="Play again", command=self.reset,
state=DISABLED)

        self.label.grid(row=0, column=0, columnspan=2, sticky=W+E)
        self.entry.grid(row=1, column=0, columnspan=2, sticky=W+E)
        self.guess_button.grid(row=2, column=0)
        self.reset_button.grid(row=2, column=1)

    def validate(self, new_text):
        if not new_text: # the field is being cleared
            self.guess = None
            return True

        try:
            guess = int(new_text)
            if 1 <= guess <= 100:
                self.guess = guess
                return True
            else:
                return False
        except ValueError:
            return False

    def guess_number(self):
        self.num_guesses += 1

        if self.guess is None:
            self.message = "Guess a number from 1 to 100"
```

```
        elif self.guess == self.secret_number:
            suffix = '' if self.num_guesses == 1 else 'es'
            self.message = "Congratulations! You guessed the number after %d guess%s."
% (self.num_guesses, suffix)
            self.guess_button.configure(state=DISABLED)
            self.reset_button.configure(state=NORMAL)

        elif self.guess < self.secret_number:
            self.message = "Too low! Guess again!"
        else:
            self.message = "Too high! Guess again!"

        self.label_text.set(self.message)

    def reset(self):
        self.entry.delete(0, END)
        self.secret_number = random.randint(1, 100)
        self.guess = 0
        self.num_guesses = 0

        self.message = "Guess a number from 1 to 100"
        self.label_text.set(self.message)

        self.guess_button.configure(state=NORMAL)
        self.reset_button.configure(state=DISABLED)

root = Tk()
my_gui = GuessingGame(root)
root.mainloop()
```