DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 242 NOTES
Spring 2019-2020: Anthony Zoko

Week of April, 21st, 2020

Announcements

- 2 weeks until the Midterm!

Graphical user interfaces

Graphical user interfaces (GUIs) give a better overview of what an application can do and make it easier to carry out application tasks.

Tkinter

The Tkinter module (Tk interface at http://docs.python.org/py3k/library/tkinter.html) is the standard Python interface to the Tk GUI toolkit for Tcl.

What is that? The following is taken from the http://wiki.tcl.tk/ website.

What is Tcl?

Tcl is a simple-to-learn yet very powerful language. Its syntax is described in just a dozen rules, but it has all the features needed to rapidly create useful programs in almost any field of application, on a wide variety of international platforms.

What is Tk?

Tk is a graphical toolkit for Tcl. It allows you to develop graphical applications that run on Windows, Linux, MacOS X and many other platforms. Tk can be used from many languages including C, Ruby, Perl, Python, and Lua.

Tcl/Tk stands for the scripting language Tcl and the graphical extension Tk. Tk is a library of basic elements and tools for building graphical user interfaces within Tcl. It is open source and is available for different operating systems. Tk provides basic widgets, top-level windows, and geometric managers to build GUI applications. First designed for Tcl, it is now used by other script language developers.

Both Tk and Tkinter are available on most Unix platforms, as well as on Windows and Mac systems.

To use Tkinter, all you need to do is import the **Tkinter** module:

```
import tkinter
```

And then use tkinter in front of each class/method you use.

Or alternatively:

```
from tkinter import ???
```

where ??? is whatever particular classes you want to use.

An overview

There are several important objects/methods we will be using to **create our first GUIs**.

These include the following:

- **Tk**: A Tk() object is a GUI widget that represents the GUI window
- **Label**: The Label() method can be used to display text or images in a window.
- **pack**: The pack() method places widgets inside of other widgets.
- **PhotoImage**: The PhotoImage class is used to transform a GIF image into an object that can be recognized by tkinter.
- **Scale**: Is used to create a sliding scale that represents numbers in a range.

In addition to creating windows, we will be creating functions that allow **our programs to respond to events** generated by the user.

The event loop is best described using the following pseudocode:

```
def mainloop():
      while the main window has not been closed:
            if an event has occurred:
                  run the associated event-handler function
```

Again, let's consider a high-level view of the type of windows and event handlers we will create:

- A GUI button, created using the **Button**() method, generates events that correspond to clicks on the button.
- The **Entry**() widget creates a text box into which information can be typed.
- The **Text** widget allows the user to enter multiple lines of text.

Remember that the documentation for Tkinter is at:
http://docs.python.org/library/tkinter.html

A first example

See the file **hello_world.py** for a first GUI example.

Let’s understand this program:

- `from tkinter import Label, Tk`: This statement imports the Tk class and the Label class from the Tkinter module
- `Tk()`: This creates a main window of the program, the one that starts when the program starts. In this program, we call this window root.
- `Widget = Label(master = root, text = “Hello GUI world!”)`: This statement creates a label widget inside the root window. A label widget is used to display text inside a window.
- `widget.pack()`: This statement arranges the widget inside its window (See http://docs.python.org/library/tkinter.html#the-packer)
- `root.mainloop()`: Start the event loop

Note that in this example we did not define the event-handler functions, so the default event-handler functions (i.e. do nothing) are used instead.

Exercise: Create a hello world example that displays an image. See **hello_image.py** for the solution.

Adding buttons and callback handlers

The standard way to interact with a GUI is through a button.

You can find an example at **reply.py**.

Instead of creating a label widget, we are creating an instance of a **button widget** inside the root window with this statement:

```
 widget = Button(root, text="Press",
command=reply)
```

The text option specifies the **callback handler**, i.e. the function to be run when the button is pressed.

The **event** is the pressing of the button. In this case, the event handler is the user-defined function reply.

reply() is the callback handler function defined as follows:

```
def reply():
    showinfo(title='popup', message='Button
pressed!')
```

showinfo() is a function in the tkinter.messagebox module that is used to show a message in a new window.

Exercise: Add another button and another event handler to the example. See the solution in the file **extendedReply.py**.

The packer

The **packer** is one of Tk's geometry-management mechanisms.

Geometry managers are used to specify the relative positioning of the widgets within their container – their mutual **master**.

The packer takes qualitative relationship specification – **above**, **to the left of**, **filling**, etc. – and works everything out to determine the exact placement coordinates for you.

The size of any **master widget** is determined by the size of the **slave widgets** inside.

The arrangement is **dynamically adjusted** to accommodate incremental changes to the configuration, once it is packed.

Note that widgets do not appear until they have had their geometry specified with a **geometry manager**.

A widget will appear only after it has had, for example, the packer's **pack**() method applied to it.

The **pack**() method can be called with **keyword-option/value pairs** that control where the widget is to appear within its container, and how it is to behave when the main application window is resized.

These options can be found in Table 9.2 on page 314 of the textbook.

Binding events to callback handlers

Let's look at another example to understand the behavior of callback handlers and how they can be used.

Consider the example in the file **clicks.py**.

The call to the **Frame** constructor creates a 100*100 pixel Frame widget inside the root window:

```
frame = Frame(width=100, height=100)
```

A Frame widget can contain other widgets, although it doesn't in this case.

Next we **bind** the "left button click" **events to the callback handler** callback with the following:

```
frame.bind("<Button-1>", handler)
```

An object of type **event** is created for each event, and it is passed to the callback handler function.

We define the callback handler as follows:

```
def handler(event):
    print('you clicked at ({}, {})'.format(event.x,
event.y))
```

The callback handler has an argument of type **Event**.

Event type objects are containers for the properties of an event.

`event.x` and `event.y` will contain the coordinate of the mouse click that created the event.

For more information, run `from tkinter import Frame` and then `help(Frame)` and `from tkinter import Frame` and then `help(Event)`.

Exercise: Add a callback handler and the appropriate binding method call to also register and output right button clicks. See the solution in **clicksExercise.py**.

Hint: Table 9.6 on page 324 lists event pattern modifiers, types, and details which is helpful for this exercise.

See **slider.py** for how to bind an event to a slider and respond to its changes.

Another example

The next example is a **keylogger**, which is an application that looks like a text editor but records and prints every keystroke the user types in the Text widget.

See the code in **key_symbol.py**.

The following statement gives the **main window a title**:

```
root.title("Keysym Logger")
```

keysum and **keysum_num** are functions of the Event class which provide information about the key symbol and key symbol numbers for the Event that generated the object.

The following statement **creates a widget to enter text**, with options for the width, height, and highlightthickness specified:

```
text=Text(root, width=20, height=5,
highlightthickness=2)
```

This statement **places the widget into the container**, indicating that it should be expanded to fill the container:

```
text.pack(expand=1, fill="both")
```

Passing arguments using lambda functions

Up until this point our callback handlers have not had parameters.

Passing parameters to callbacks is a bit tricky and can involve something called a lambda expression.

A **lambda expression** is a Python expression (not a statement) that evaluates to a function object.

Like def, a lambda expression creates a function to be called later, but returns it instead of assigning it to a name.

See the example below:

```
>>> f = lambda x, y, z: x + y + z
>>> f(2, 3, 4)
9
```

Note the following:

- A lambda is an expression, not a statement
    - A lambda expression can appear where a def cannot
    - A lambda expression allows a function definition to be embedded within the code that uses it
- lambda bodies are a single expression, not a block of statements

See a first example in **personal.py**.

Unfortunately this isn't an optimal example, because it's possible to write this example without using a lambda expression.

All we need to do is define a function cmd() that calls reply() with a parameter. See the example rewritten in **personal2.py**.

We can see the full power of lambda expressions if we **migrate the lambda expression** that calls reply() with a parameter into the call to the Button constructor.

See the **personal3.py** example for the details.

We can also get rid of lambdas altogether by making the GUI object-oriented.

Exercise: Create an OO version of the personal example.

Using grid() instead of pack()

The **grid**() method allows for more control when packing widgets into the window.

The grid() method organizes the master widget into rows and columns.

Table 9.3 on page 316 gives **some options for the grid**() method.

Note: The methods pack() and grid() **should not be used together**.

See the first example in **reply2.py**.

To consider a less artificial example, let's consider **a basic calculator**.

In the calculator example, we will also see the insert() and delete() **methods of the Entry widget**. Recall that Table 9.4 on page 321 gives some Entry methods including insert() and delete().

The **insert**() method takes two parameters: index and text. It inserts text into the widget before the index specified.

The **delete**() method takes two parameters: from and to. It deletes the substring in the widget between the indices from and to – 1.

See the code for the first version of the calculator in **basic_calc.py**.

It has two buttons:

1. The first button evaluates the expression typed into the Entry widget and replaces the expression with the result.
2. The second button clears the information from the Entry widget.

Exercises:

1. Modify the GUI so that it doesn't crash when given a string that isn't an arithmetic expression.
2. Modify the basic calculator to be object-oriented