Started on	Monday, 13 July 2020, 8:47 PM
State	Finished
Completed on	Monday, 13 July 2020, 11:30 PM
Time taken	2 hours 43 mins
Marks	3.00/3.00
Grade	**10.00** out of 10.00 (**100**%)

Information

Aims, background, resources

The aim of this lab is to familiarise you with Graphical User Interface (GUI) programming with Python/TkInter. Note that many of the basic concepts of GUI programming are similar across programming platforms.

At the end of this lab you should understand the basics of the following:

1. widget creation with Python/TkInter;
2. geometry management with the packer and gridder;
3. event binding.

Background

Hopefully you are already familiar with GUI programming, and hopefully you have built several GUIs with Python/TkInter, but don't worry if you haven't. The first two labs of COSC368 will give you a quick introduction to key GUI programming concepts. Some of this material was initially covered in COSC121.

COSC368 is primarily concerned with the foundations of Human-Computer Interaction, and with user interface design. These concerns are substantially independent from actual user interface programming. However, the final outcome of good HCI is systems that are easy to learn, fast to use, and subjectively satisfying, and these systems must be programmed. As you will learn later in the course, paper and pencil prototypes are pivotal in early design, but as designs mature, the primary medium for design exploration becomes executable simulations and prototypes. Therefore, it's important that you are able to program user interfaces.

TkInter is a Python package that is automatically included with the Python release, and it is therefore the 'standard' GUI package for Python (although, several others are available). You need to know that TkInter is just a Python interface to the Tk GUI toolkit, which was originally written as an extension to the scripting language Tcl. Consequently, you may find that some of the documentation for Tk is written in Tcl syntax (not Python). It shouldn't take you long to learn how to translate from one syntax to the other.

Online Resources

There are many good online sources for Python/TkInter. Start with:

- http://www.tkdocs.com/
- http://infohost.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf
- https://docs.python.org/3.4/library/tkinter.html

GUI Hello World

Type the following into a Python program and run it (e.g., using WingIDE 101). Once running, type something into the Entry widget.

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
data = StringVar()
data.set("Data to display")
label = Label(window, textvariable=data)
label.grid(row=0, column=0)
entry = Entry(window, textvariable=data)
entry.grid(row=1, column=0)
window.mainloop()
```

Lines 1 and 2 make everything available from python packages tkinter and from ttk. Ttk provides a set of 'themed' widgets that update the original tkinter widgets, allowing styles to be applied to widgets.

The invocation to Tk() on line 3 creates a root window and initialises Tk's capabilities.

Line 4 creates a StringVar, which is a special Tk class supporting mutable strings; and line 5 sets the value of the string.

Line 6 creates a Label widget that is a child of the root window. The label is instructed to display text in the variable data.

Line 7 uses a grid for geometry management, which assigns the widget into the smallest space required to properly display the widget on the first (zeroth) row and column.

Line 8 creates an Entry widget that can be used to enter text. The widget is instructed to display and control the content of the variable data.

Line 9 uses a grid for geometry management again, packing the widget into its parent (the window).

Line 10 initiates the main loop, which is an infinite loop that continually awaits user input on the GUI and allows user events on widgets (such as typing in the Entry widget) to be processed.

When running the program, you should notice that editing the text in the Entry widget also changes the text in the label above it. This is because editing text in the Entry widget also changes the StringVar, which automatically

label above it. This is because editing text in the Entry widget also changes the StringVar, which automatically propagates those changes to other controls that use it.

Question **1**

Complete

Mark 1.00 out of 1.00

Hello World v2

Add the following lines to the program above (above the call to window.mainloop()).

```
clear = Button(window, text="Clear", command=lambda: clear_data(data))
clear.grid(row=2, column=0)
quit = Button(window, text="Quit", command=window.destroy)
quit.grid(row=3, column=0)
```

The program won't quite work yet, but when it does, clicking the "Quit" button will terminate the program by destroying the root window (which also kills the main loop). Clicking on the "Clear" button is supposed to make the Label and Entry widgets blank; the click will call a function called clear_data, passing in a reference to the StringVar object. *You need to write the function*.

Submit your entire program here.

```
from tkinter import *
from tkinter.ttk import *
class Lab1Window(Tk):
    """ testing """
    def __init__(self):
        self.window = Tk()
        self.data = StringVar()
        self.data.set("Data to display")

        self.lblDisplay = Label(self.window, textvariable=self.data)
        self.lblDisplay.grid(row=0, column=0)
        self.txtDisplay = Entry(self.window, textvariable=self.data)
        self.txtDisplay.grid(row=1, column=0)

        self.btnClear = Button(self.window, text="Clear",
                               command=lambda: self.clear_data())
        self.btnClear.grid(row=2, column=0)
        self.btnQuit = Button(self.window, text="Quit", command=self.window.destroy)
        self.btnQuit.grid(row=3, column=0)

    def clear_data(self):
        self.data.set("")

    def mainloop(self):
        self.window.mainloop()

def helloWorld():
    window = Lab1Window()
    window.mainloop()

helloWorld()
```

Comment:

Information

Widget Set and Widget Options

The programs above demonstrate several Tk widgets: a 'toplevel' window (implicitly created by the call to Tk), and Entry, Button, and Label widgets. There are many more, and they are fairly standard across GUI languages: Frame (contains/group other widgets), Checkbutton, Radiobutton, Listbox, Menu, Menubutton, Message, Scale, Scrollbar, Text, Canvas, etc.

These are all implemented as TkInter classes (see an incomplete list at http://www.tkdocs.com/tutorial/widgets.html), with most classes offering many methods. Obviously, there's more here than we can cover in two labs. Luckily, once you've learned how to work with one type of widget, much of that learning transfers to other widgets too.

The appearance of widgets can be configured through styles.

Try adding the following two lines after the "clear.grid" statement:

```
s = Style()
s.configure('TButton', font='helvetica 24', foreground='green')
```

The first statement makes s refer to a Style object from ttk. The second statement configures the style object so that all 'Tbutton' objects (all ttk buttons) have a large helvetica font with a green foreground.

Working with Styles is more advanced than our main aims for now, but you can look at details here: http://www.tkdocs.com/tutorial/styles.html

Information

Geometry Management

Geometry managers are used to control where widgets are placed within their parent widget (which might be a top-level window or a container widget such as a Frame). Widgets will not appear unless placed using a geometry manager. Each parent widget will allow a single geometry manager to be used, so mixing geometry managers in one parent widget is not possible.

The two most commonly used geometry managers with TkInter are the packer and the gridder.

Packer

Try the following:

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
side_labels = ["bottom1", "bottom2", "top1", "top2", "left1", "right1"]
for theside in side_labels:
    button = Button(window, text=theside)
    button.pack(side=theside[0:-1])
window.mainloop()
```

This uses the packer for geometry management; attaching widgets to the bottom, top, left, or right of the *remaining* available space, and assigning a 'parcel' of space for the widget on *x* and *y* dimensions. The first button grabs a parcel at the bottom of the space; the next grabs the bottom of what's left; the next gets the top, and so on. As you might expect, there are many configuration options, such as allowing widgets to expand to fill all of the *x, y,* or *both* coordinates within its parcel. The internal and external padding of the widget can also be configured. Details of the packer algorithm are available here: http://effbot.org/tkinterbook/pack.htm

Gridder

Try the following:

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
for label_num in range(6):
    button = Button(window, text="Button"+str(label_num))
    button.grid(row=label_num // 3, column=label_num % 3)
window.mainloop()
```

This uses the grid geometry manager, which places items in (you guessed it) grid coordinates (documentation at http://www.tkdocs.com/tutorial/concepts.html#geometry and http://www.tkdocs.com/tutorial/grid.html). Again, there are lots of options, such as allowing items to 'stick' to grid edges when the grid is resized, and allowing items to span multiple columns, as demonstrated by the following program:

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
for label_num in range(6):
    button = Button(window, text="Button" + str(label_num))
    button.grid(row=label_num // 2, column=label_num % 3)
    if label_num==1:
        button.grid(columnspan=2, sticky="ew")
    elif label_num==3:
        button.grid(rowspan=2, sticky="ns")
window.columnconfigure(1, weight=1)
window.rowconfigure(1, weight=1)
window.rowconfigure(2, weight=1)
window.mainloop()
```

Try resizing the window and observe how each of the buttons changes their size. The sticky flag causes the given edges of the button to expand with the size of the grid; the columnconfigure() and rowconfigure() functions adjust which rows and columns of the grid shrink and expand as the window resizes.

Frames for layout

You can use Frame widgets to contain other widgets. Each Frame has its own geometry manager. This gives you a widget hierarchy, with the root window containing a set of Frames, and each Frame may have its own child Frames as well as other widgets.

For example, try the following, and resize the window when it appears:

```
from tkinter import *
from tkinter.ttk import *
window = Tk()
frame_left = Frame(window, borderwidth=4, relief=RIDGE)
frame_left.pack(side="left", fill="y", padx=5, pady=5)
frame_right = Frame(window)
frame_right.pack(side="right")

button1 = Button(frame_left, text="Button 1")
button1.pack(side="top")
button2 = Button(frame_left, text="Button 2")
button2.pack(side="bottom")

for label_num in range(4):
    button = Button(frame_right, text="Button" + str(label_num + 3))
    button.grid(row=label_num // 2, column=label_num % 2)
window.mainloop()
```

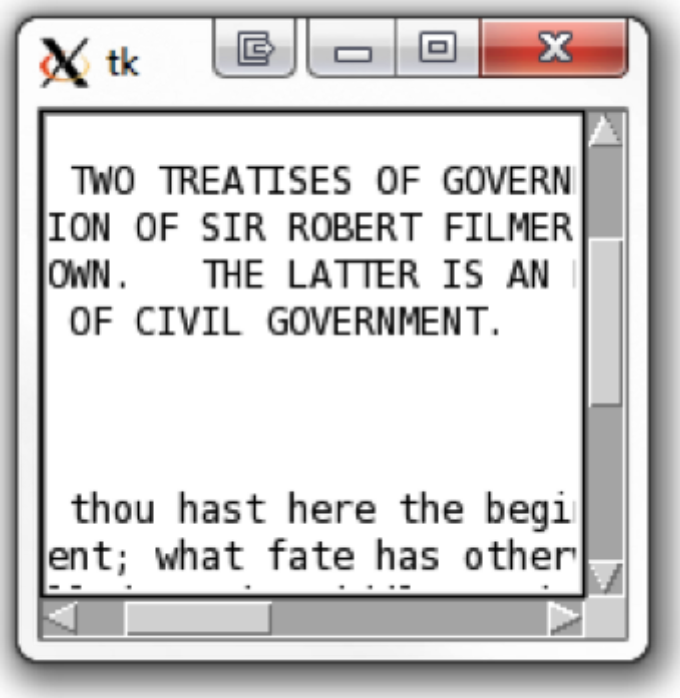
Question **2**

Complete

Mark 1.00 out of 1.00

2D Scrolling Window

Create a window containing a Text widget (24 characters wide and 10 characters high) with horizontal and vertical scrollbars. The scrollbars must control the viewport into the text window (obviously, you'll first need some text in the Text widget). When you're done, it should look something like the following:




```
from tkinter import *
from tkinter.ttk import *
class TestWindow(Tk):
    def __init__(self, title):
        self.window = Tk()
        self.window.title(title)

    def mainloop(self):
        self.window.mainloop()
class Window2e(TestWindow):
    def __init__(self, title):
        super().__init__(title)
        data = """A Bézier curve (/ˈbɛz.i.eɪ/ BEH-zee-ay)[1] is a parametric curve
used in computer graphics and related fields.[2] The curve, which is related
to the Bernstein polynomial, is named after Pierre Bézier, who used it in the
1960s for designing curves for the bodywork of Renault cars.[3] Other uses
include the design of computer fonts and animation.[3] Bézier curves can be
combined to form a Bézier spline, or generalized to higher dimensions to form
Bézier surfaces.[3] The Bézier triangle is a special case of the latter.

In vector graphics, Bézier curves are used to model smooth curves that can be scaled
indefinitely. "Paths", as they are commonly referred to in image manipulation
programs,[note 1] are combinations of linked Bézier curves. Paths are not bound by the
limits of rasterized images and are intuitive to modify.

Bézier curves are also used in the time domain, particularly in animation, user
interface[note 2] design and smoothing cursor trajectory in eye gaze controlled
interfaces.[4] For example, a Bézier curve can be used to specify the velocity over
time of an object such as an icon moving from A to B, rather than simply moving at a
fixed number of pixels per step. When animators or interface designers talk about the
```

"physics" or "feel" of an operation, they may be referring to the particular Bézier curve used to control the velocity over time of the move in question.

This also applies to robotics where the motion of a welding arm, for example, should be smooth to avoid unnecessary wear."""

```
        scrollVertical = Scrollbar(self.window)
        scrollVertical.pack(side = RIGHT, fill = Y )
        scrollHorizontal = Scrollbar(self.window, orient='horizontal')
        scrollHorizontal.pack(side = BOTTOM, fill = X )

        txtScroll = Text(self.window, width=24, height=10, wrap=NONE,
            yscrollcommand=scrollVertical.set, xscrollcommand=scrollHorizontal.set)
        txtScroll.insert(1.0, data)
        txtScroll.pack(side = LEFT, fill = BOTH)

        scrollVertical.config( command = txtScroll.yview )
        scrollHorizontal.config( command = txtScroll.xview )

def ques2e():
    window = Window2e("Scrollbar")
    window.mainloop()
ques2e()
```

Comment:

Information

Event Binding

We bind functions to particular events on specific widgets. Some widgets (such as Buttons) have shortcut configuration options that ease "binding" simple function calls to simple user events (e.g., a click). Otherwise, the mechanisms for binding functions to user events are fairly similar across widget classes.

Try the following:

```
from tkinter import *
from tkinter.ttk import *
def add_one():
    value.set(value.get()+1)

def wow(event):
    label2.config(text="WWWWOOOOWWWW")

window = Tk()
value = IntVar(0)
label = Label(window, textvariable=value)
label.pack()
label2 = Label(window)
label2.pack()
button = Button(window, text="Add one", command=add_one)
button.bind("<Shift-Double-Button-1>", wow)
button.pack()
window.mainloop()
```

Line 13 of this program uses the command option to identify a simple zero-parameter function that is called when the "Add one" button is clicked with the left mouse button.

Line 14, in contrast, uses the bind method of the Button class (this method is defined for most widget classes) to bind a call to the function "wow" when the user completes the action specified in the event descriptor on the widget (a shift-double-click of the first (left) mouse button). When functions are bound to events using the bind method, an additional parameter describing the event is automatically passed to the named function. Consequently, the "wow" method has a parameter to receive the event.

Event descriptors and binding are summarised at: http://www.tkdocs.com/tutorial/concepts.html#events and http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm

Parameter passing with event binding

Given that the bind method (or the command option for Buttons) doesn't allow parameters to be stipulated, how can the programmer control what parameters are passed to callback functions?

This is achieved using 'lambda functions'. Lambda functions are single-expression nameless functions. Try the following:

```
from tkinter import *
from tkinter.ttk import *
def change(the_value, n):
    the_value.set(the_value.get()+n)

window = Tk()
value = IntVar(0)
label = Label(window, textvariable=value)
label.pack()
button = Button(window, text="Left +1, Right -1")
button.bind("<Button-1>", lambda event: change(value, 1))
button.bind("<Button-3>", lambda event: change(value, -1))
button.pack()
window.mainloop()
```

Lines 10 and 11 create nameless functions that receive one parameter (named event), which is supplied by tkinter. The body of the function is a call to function change, with two parameters.

Note that lambda functions can have any number of parameters, each of which can have a default value. This will be important in next week's lab when you complete the event binding for the exercise below.

Question **3**

Complete

Mark 1.00 out of 1.00

Keyboard GUI

In the coming labs you will use a small keyboard GUI to conduct some experiments investigating human performance.

GUI Layout

Produce a GUI that looks like the interface shown below. Note that the content of the keyboard should be defined by a single list, something like the following Python statement:

```
board = ['qwertyuiop', 'asdfghjkl', 'zxcvbnm']
```

Event Binding

Append characters to the Label when the letter keys are clicked

There are several ways to wire up the buttons of the keyboard to append their characters to the label, but the cleanest is to use a lambda function with a default parameter.

Let's assume that you have a function called append, that receives a parameter containing a string for the keyboard character that has been clicked (for example, when the user clicks on the 'h' key, you want to invoke append('h')). Let's also assume that you create all of your keyboard buttons inside of a for loop that builds a button for each key (rather than individually creating 26 keys), where each character is stored in the loop variable ch.

The following statement will not work as expected:

```
            b = Button(board, text=ch, command=lambda: append(ch))
```

This is because the name ch is not evaluated until the button is clicked, at which point the for loop will have since completed and ch will be whatever it was in the last iteration of the for loop. In order to capture the value of ch correctly, you will need need something like the following:

```
            b = Button(board, text=ch, command=lambda x=ch: append(x))
```

This statement creates a button labelled with the content of variable ch. The binding is achieved using the button's command option. The value for the command option needs to be a reference to a zero argument function that will be invoked when the button is pressed. The lambda function returns a reference to a function that *can* receive one argument, but can also be invoked with zero arguments, in which case it uses the default value, which is the content of ch. The value of ch is immediately evaluated (rather than evaluated when the function is called); consequently, the bound function is a call to append with the default parameter value stipulated at the time that the button is created. Phew!

Clear the text in the Label when 'Clear' is clicked

Wire up the 'Clear' Button so that it clears any text entered into the Label.

Submit your completed keyboard program here.

```
from tkinter import *
from tkinter.ttk import *
class TestWindow(Tk):
    def __init__(self, title):
        root = Tk()
        root.title(title)
        self.window = Frame(root)
        self.window.pack(padx=10, pady=10)

    def mainloop(self):
        self.window.mainloop()

class Window3c(TestWindow):
```

```
def __init__(self, title):
    super().__init__(title)
    board = ['qwertyuiop', 'asdfghjkl', 'zxcvbnm']
    self.data = StringVar()
    self.data.set("")

    self.lblType = Label(self.window, textvariable=self.data)
    self.lblType.grid(row=0, column=0, sticky="W")
    self.btnClear = Button(self.window, text="Clear", command=self.clear)
    self.btnClear.grid(row=0, column=1, sticky="E")

    frame = Frame(self.window, borderwidth=4, relief=RIDGE)
    frame.grid(row=1, column=0, columnspan=2)
    for i in range(len(board)):
        row = board[i]
        for j in range(len(row)):
            ch = row[j]
            btn = Button(frame, text=ch, width=2, command=lambda x=ch: self.append(x))
            btn.grid(row=i, column=i+j*2, columnspan=2)

    def clear(self):
        self.data.set("")

    def append(self, ch):
        self.data.set(self.data.get()+ch)
def ques3c():
    window = Window3c("Keyboard")
    window.mainloop()
ques3c()
```

Comment:

◄ Māori Usability Issues with Computer Interfaces

Jump to...

Lab 2: GUI Programming with Python/TkInter (II) ►