Workshop - Building Applications Using Python and Tkinter

Wendy Ivins

Introduction

This workshop will introduce you to building applications using graphical user interfaces (GUIs). It gives a detailed step-by-step guide to building a teamwork questionnaire application, and will introduce you to a range of widgets (buttons, labels, frames, listboxes etc.) that can be used for building your own applications.

Make sure you have completed all of the exercises in this workbook by the end of week 5.

What is Tkinter?

Tkinter is the most commonly used toolkit for creating GUIs for Python programmes and is included when Python is installed. Tkinter is a Python interface to the Tcl/Tk GUI toolkit (***Tk inter***face). It uses the Tcl/Tk libraries to provide widgets that can be used to create a user interface. Tk uses native widgets for each platform and so generates the appropriate look and feel for GUIs on Windows, Unix and Macintosh platforms. If your application does not use platform-specific fonts then you will be able to use the same code on a different platform. Therefore Tkinter promotes *portable* code.

Tkinter adds object-oriented interfaces to Tk. In Tkinter the widget references are objects and these are driven using object *methods* and their *attributes*.

Create the Main Window

When teams work together to develop sophisticated applications they break down their work into separate modules. Teams can then reduce the overall time to completion by developing modules simultaneously.

We create each module in its own file. The main window for the Questionnaire will be developed as the first module. Start by creating a new file and write your code in the editing window.

I've developed my python code using the Enthought Canopy editor and my examples are run on Windows. You can use another editor such as Sublime Text if you prefer.

We start by importing the Tkinter module:

```
from Tkinter import *
```

We then set up the basic code for our application window to provide a frame for the Questionnaire application:

```
class Questionnaire(Frame):
    # GUI Setup

    def __init__(self, master):
        # Initialise Questionnaire Class

        Frame.__init__(self, master)
        self.grid()
```

The def __init__ statement will be used to call all the methods needed to create our application window. There is quite a lot of code to write so we will be breaking it down into a number of def statements and we will frequently run our code to check it is working.

We will start by seeing if we have created the application window. First we need to develop the Main part of the code where we call the Tk method, set the title of our application, create a new instance of the Questionnaire class, and start the window with the mainloop method:

```
# Main
root = Tk()
root.title("Teamwork Questionnaire")
app = Questionnaire(root)
root.mainloop()
```

Exercise 1
Type out the code so far in a new Python script file. Save your file as Questionnaire.py. You should create an appropriately named folder for each application and save all of the script files to this folder. Python will import modules from this folder and store data in this folder.

Run the module. If you have made a mistake then error messages will appear in the interactive window and you will need to correct your code – check the syntax and layout carefully.

You should get a window like the one shown on the right. It does not do much as it does not have any widgets. However, you can move the window, resize the window at the corners, use the resize buttons and close the window.

Screen Layout

Screen layout is determined by geometry managers, which control the size and position of widgets on the screen. One or more *slave* widgets (e.g. labels, buttons, checkbuttons etc) are placed within a *master* widget (a container such as a frame or canvas). The geometry manager controls the behaviour of these widgets as the window is resized, or more widgets are added.

Tkinter provides three geometry managers to control the size and position of widgets on the screen:

- The **Pack** Geometry Manager is the simplest manager to use. It packs child widgets into a parent by treating them as rectangular blocks placed in a frame.
- The **Grid** Geometry Manager lays widgets out in table-like layouts using a two-dimensional grid. It is the most flexible of the geometry managers to use and is easier than the Pack Manager to use when there are complex layouts.
- The **Placer** Geometry Manager allows precise positioning of widgets in window by providing co-ordinates.

Please note that you should never mix grid and pack in the same master window.

Placing Widgets using the Grid Geometry Manager

We will be using grid manager to create the interface for our Questionnaire application.

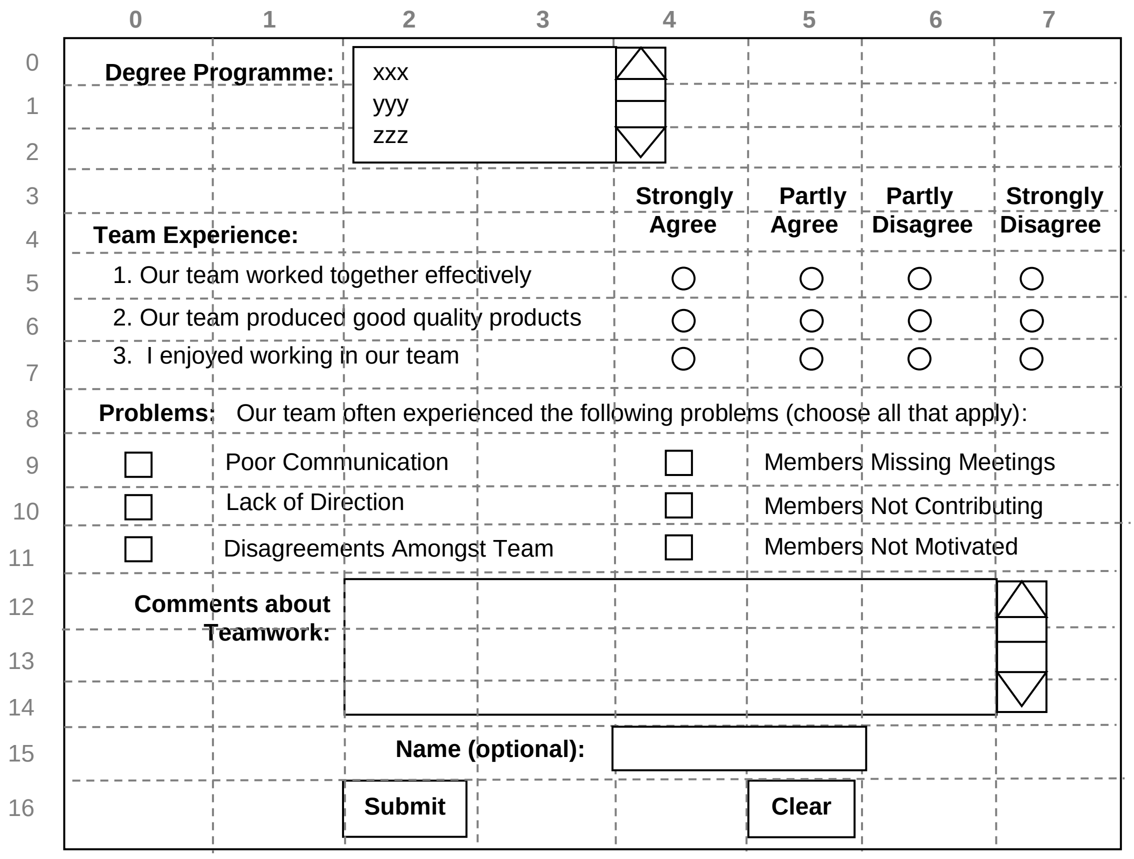
You don't have to specify the size of the grid as the manager automatically determines that from the widgets placed within it.

The `grid` method has options for setting the `row` and `column` for the required widget. We can set `rowspan` and `columnspan` if widgets span more than one cell. Widgets are centred in their cell but we can set the `sticky` option to change the alignment (N, S, E, W or combinations of these e.g. NE)

We change the look and feel of widgets by changing options such as text, color, size. All options have default values so you only need to change the ones where you don't like the default.

Example – Creating a Teamwork Questionnaire

If we draw out the required interface we can see that it requires a complex interface. The Grid manager is our best option so we will add a grid to our drawing to help with the layout. We will use the grid positions in our code to help improve the layout.



Incremental Development

It is good practice to develop your code a little at a time and run it frequently so you can spot if you have introduced a mistake and correct it.

This interface breaks down nicely into several sections:

- Creating the Degree Programme Selection
- Creating the Team Experience Questions
- Creating the Problems Selection
- Creating the Comments and Name
- Creating the Buttons

We will create each of these sections as separate def statements. They will follow the code in the def __init__ statement but will come before the main part of the code (before *#Main*).

We also need to make sure that we call our def statements in the def __init__ statement of our Questionnaire class to ensure they are added to our application window.

Creating the Degree Programme Selection

We are creating a scrolling listbox to allow selection from a range of degree programmes:
CS, CS with, SE, BIS and Joints

This requires a label, listbox and scrollbar.

```
    def createProgSelect(self):
        # Create widgets to select a degree programme from a list

          lblProg = Label(self, text='Degree Programme:', font=('MS', 8,'bold'))     1
          lblProg.grid(row=0, column=0, columnspan=2, sticky=NE)

          self.listProg = Listbox(self, height= 3)
          scroll = Scrollbar(self, command= self.listProg.yview)                      2
          self.listProg.configure(yscrollcommand=scroll.set)

          self.listProg.grid(row=0, column=2, columnspan=2, sticky=NE)
          scroll.grid(row=0, column=4, sticky=W)                                     3

          for item in ["CS", "CS with", "BIS", "SE", "Joints",""]:                   4
              self.listProg.insert(END, item)

         self.listProg.selection_set(END)                                            5
```

Label

Labels are used to place text or images on the screen. Default font is MS, size 8 font.

We have set up the text option in bold for the first label lblProg. We have placed it in the grid at row 0 to span two columns. We have aligned the label to the top (N) and right(E). 1

Listbox and Scrollbar

Listboxes are used to display a list of values that can be chosen by a user. The default is BROWSE which allows you to select a single item, but the selection will follow your mouse if you drag to a different line. .

If you don't want your listbox to take up too much space then you can combine it with a scrollbar. The default height is 10 lines so we have set the listbox height to display 3 rows and set a scrollbar for the listbox that controls the y axis. We will want to be able to read and clear the value selected so we will use `self.` before the listbox name to make it available later in the program. 2

We have placed the listbox next to the label, in column 2 and set the scroll bar to after the listbox in column 4. The listbox and scrollbar have been aligned to be next to one another. 3

We have inserted the required values into the listbox, including an empty string at the end. 4

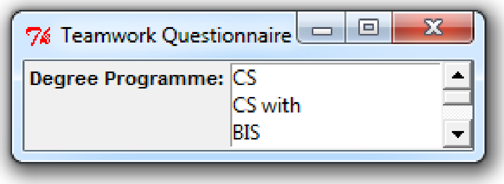
We have set the default selection to the empty string at the end so we can later check that the user has selected a Degree Programme 5

Finally we call our createProgSelect() method just after the `self.grid()` method in the initialisation setup for the Questionnaire Class:

```
        self.createProgSelect()
```

Exercise 2

Update your Questionnaire script file with the code needed to create the Degree Programme selection, then save and run the application. You should get a window similar to the one shown.



Creating the Team Experience Questions

First start by defining a method to set up the Team Experience Questions:

```
  def createTeamExpQuest(self):
      # Create widgets to ask Team Experience Questions
```

We want to set the Strongly Agree label over two rows, at row 3, column 5. We can split the text using a newline (\n) in the string:

```
     lblStrAgr = Label(self, text = 'Strongly \n Agree', font=('MS', 8,'bold'))
     lblStrAgr.grid(row=3, column= 4, rowspan=2)
```

Now add appropriate labels to set up the rest of this part of the form and ask the questions. *You should know enough about labels to do the rest of this yourself.*

Radiobutton

We will use radiobuttons to display the answers. Radiobuttons allow the user to select only one choice from a group of buttons. Each group of Radiobuttons are associated with single variable. Each button then represents a single value for that variable. We can normally set text for each radiobutton but this application will use the text from labels in row 3.

We will need three sets of four radiobuttons, one set for each question. We will pass back integer values (4 = Strongly Agree, 3 = Partly Agree, 2 = Partly Disagree, 1 = Strongly Disagree) to store in the variable for each question. We will use `self.` so we can read the variables and use them to reset the radiobuttons later in the program. Here is the code for the radiobuttons for the first question:

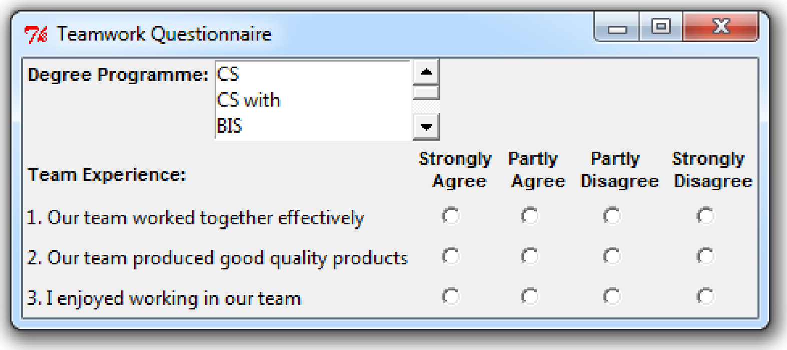
```
     self.varQ1 = IntVar()

     R1Q1 = Radiobutton(self, variable=self.varQ1, value=4)
     R1Q1.grid(row=5, column= 4)

     R2Q1 = Radiobutton(self, variable= self.varQ1, value=3)
     R2Q1.grid(row=5, column= 5)

     R3Q1 = Radiobutton(self, variable= self.varQ1, value=2)
     R3Q1.grid(row=5, column= 6)

     R4Q1 = Radiobutton(self, variable= self.varQ1, value=1)
     R4Q1.grid(row=5, column= 7)
```

To set up the code for the buttons in other questions we need to change all variables (e.g. `varQ1` to `varQ2`), change our buttons (e.g. `R1Q1` to `R1Q2)` and change rows (e.g. 5 to 6).

Don't forget to call the createTeamExpQuest() method in the initialisation setup for the Questionnaire Class.

Exercise 3

Update your Questionnaire script file with the code needed to create the Team Experience Questions, then save and run the application. You should get a window similar to the one shown.



Creating the Problems Selection

We will start with creating the labels to ask about the problems. We will use two so one part can be bold and another in normal font.

```
  def createProblems(self):
      #Create Widgets to show Problems experienced

      lblProb1 = Label(self, text='Problems:', font=('MS', 8,'bold'))
      lblProb1.grid(row=8, column = 0)

      lblProb2 = Label(self, text='Our team often experienced the ' +
                                  'following problems (choose all that apply):')
      lblProb2.grid(row=8, column = 1, columnspan=6, sticky=W)
```

Note that we have split the long text in the second label to make it more readable.

Checkbutton

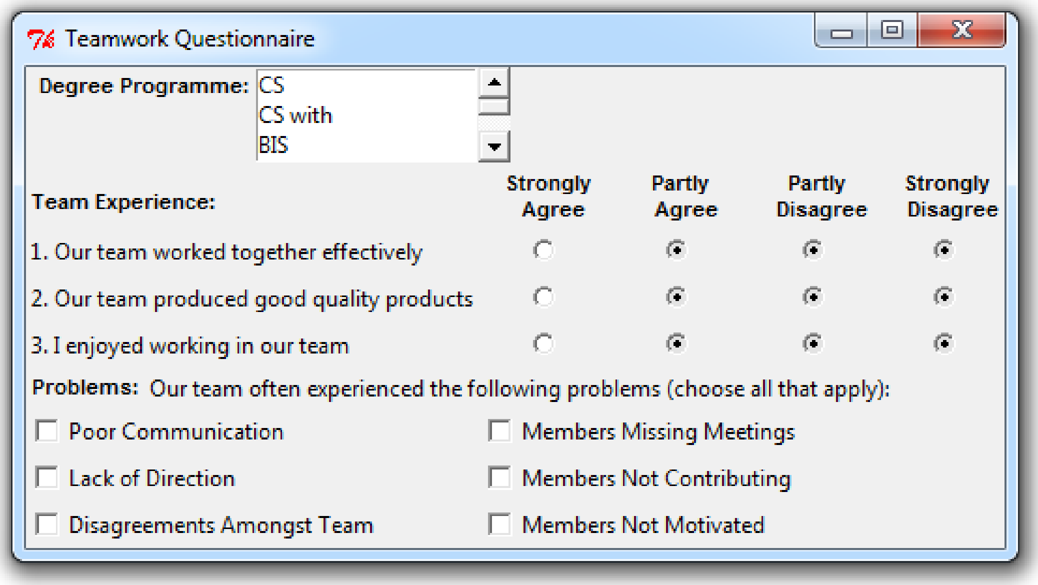
Checkbuttons allow a user to select none, some or all of a range of choices. We will make use of checkbuttons to indicate the problems experienced by a team. We will be reading the variables and using them to reset the checkboxes later in the program.

```
    self.varCB1 = IntVar()
    CB1 = Checkbutton(self, text=" Poor Communication", variable=self.varCB1)
    CB1.grid(row=9, column=0, columnspan=4, sticky=W)
```

Don't forget to include a call for the createProblems() method in the initialisation setup for the Questionnaire Class

Exercise 4

Update your Questionnaire script file with the code needed to create the Problems Selection, then save and run the application. You should get a window similar to the one shown.



Creating the Comments Box and Name

Define a createComments method to set up the Comments and Name entries, and add the appropriate label to appear in row 12.

Text

Text widgets can display or capture multiple lines of text. The default height is 24 (lines) and the default width is 80 (characters). We will use a text widget and scrollbar to capture the comments. We will resize the textbox to look more in proportion to the rest of the display. We will want to read the text later in the program.

```
self.txtComment = Text(self, height=3,width=40)

scroll = Scrollbar(self, command=self.txtComment.yview)
self.txtComment.configure(yscrollcommand=scroll.set)

self.txtComment.grid(row=12, column=2,columnspan=5, sticky=E)
scroll.grid(row=12, column=7, sticky=W)
```

We then need to add a label for the name on row 15.

Entry

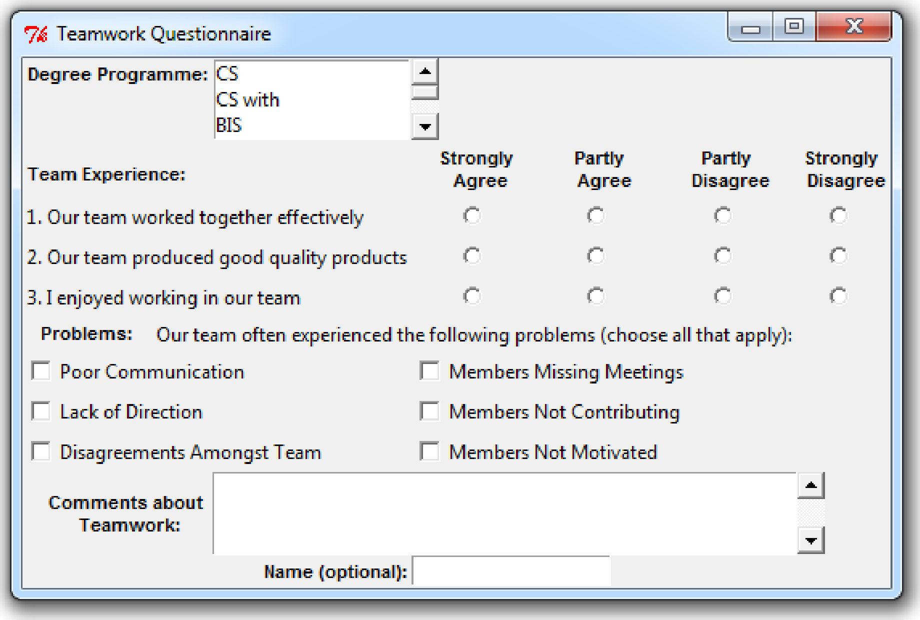
Entry widgets accept a single line of text from a user. The default width is 20 (characters). We will want to read the text later in the program.

```
self.entName = Entry(self)
self.entName.grid(row=15, column=2, columnspan=2, sticky=E)
```

Include a call for the createComments() method in the initialisation setup for the Questionnaire Class

Exercise 5

Update your Questionnaire script file with the code needed to create the Comments, then save and run the application. You should get a window similar to the one shown.



Creating the Buttons

Define a createButtons method to set up the Select and Cancel buttons.

Button

Buttons react to mouse and keyboard events. We can bind a method to a button, which will be invoked when the button is activated. We will add the submit button which will invoke the method storeResponse when the Submit button is pressed.

```
    butSubmit = Button(self, text='Submit',font=('MS', 8,'bold'))
    butSubmit['command']=self.storeResponse        #Note: no () after the method
    butSubmit.grid(row=16, column=2, columnspan=2)
```

We can also add a Clear button, which will invoke the method clearResponse when the Clear button is pressed.

Clearing the Questionnaire

First define the clearResponse method.

```
  def clearResponse(self):
      #Clear the Questionnaire
```

We can clear the listbox and reset the selection to the empty string at the end of the list

```
        self.listProg.selection_clear(0,END)
        self.listProg.selection_set(END)
```

We can clear radiobuttons and checkboxes by setting their variables to zero.

```
        self.varQ1.set(0)
        self.varCB1.set(0)
```

Add the rest of the code to reset other radiobuttons and checkboxes.

We clear the entry and textboxes using the delete method. The delete method for entry widgets works on characters, but the delete method for text widgets uses indexes.

```
        self.entName.delete(0, END)
        self.txtComment.delete(1.0, END)
```

Storing the Response

First define the storeResponse method.

```
    def storeResponse(self):
        #Store the results of the Questionnaire
```

First we will check that a degree programme has been selected. We have set the default to be the last item in the list (the empty string).

```
        index = self.listProg.curselection()[0]
        strProg = str(self.listProg.get(index))
        strMsg=""
```

If there is no programme we will set up a message string.

```
        if strProg == "":
            strMsg = "You need to select a Degree Programme. "
```

We will now check that each of the Team Experience Questions has been answered. If any of the variables for the radio buttons are 0 then we will update our message string.

```
        if (self.varQ1.get()== 0) or (self.varQ2.get() == 0) or (self.varQ3.get() == 0):
            strMsg = strMsg + "You need to answer all Team Experience Questions"
```

Note: we don't have to show errors for Problem checkboxes, Comment and Name information as these can remain blank

If we have not found any problems (i.e. strMsg is still empty) then we will store the response in a Response class. The Response class will also be used to retrieve results in another module so we will store the Response class in a separate script file (Response.py). This should be stored in the same folder as Questionnaire.py

```
class Response:
    def __init__(self, respNo="", prog="", q1=0, q2=0,q3=0,
                 pr1=0, pr2=0, pr3=0, pr4=0, pr5=0, pr6=0,
                 comment="", name=""):
        self.respNo = respNo
        self.prog = prog
        self.q1 = q1
        self.q2 = q2
        self.q3 = q3
        self.pr1 = pr1
        self.pr2 = pr2
        self.pr3 = pr3
        self.pr4 = pr4
        self.pr5 = pr5
        self.pr6 = pr6
        self.comment = comment
        self.name = name
```

The Response Class provides default values in its interface. We can read and write values directly for instances of the class (class objects) using the interface to the class.

We need to import the Response module at the top of the Questionnaire module. This can go after our import of the Tkinter module.

```
from Response import Response
```

Pickle and Shelve

We can store the responses generated when running the questionnaire programme in files or databases. Pickle can directly format objects into strings and reconstruct the objects when they are retrieved from storage. This saves the need to write the extra code to create and parse the objects if these were stored in text files. Shelve uses pickle to translate the object to its pickled string and store it in a dbm database so it can be retrieved using a key string. Shelve also maps dictionary operations (e.g. len, in, sorted) on objects stored in the database, as long as the database is open.

We can now return to our storeResponse code. We will start by using shelve to open a database. We have named the database responsedb and it will be created in the same folder as the Questionnaire script file.

```
        if strMsg == "":

              import shelve
              db=shelve.open('responsedb')
```

We need to ensure that each Response object has a unique key. This will be achieved by developing a key based on Response number. This will be generated by counting the number of responses and adding 1 to the next key. This assumes that we won't delete individual records in the database. We will then create a new response object and write the values submitted into the new object and store it in responsedb.

```
              responseCount = len(db)
              Ans = Response(str(responseCount+1), strProg,
                              self.varQ1.get(), self.varQ2.get(), self.varQ3.get(),
                              self.varCB1.get(), self.varCB2.get(), self.varCB3.get(),
                              self.varCB4.get(),  self.varCB5.get(), self.varCB6.get(),
                              self.txtComment.get(1.0,END), self.entName.get())

              db[Ans.respNo] = Ans
              db.close
```

tkMessageBox

We can inform users using the tkMessageBox Dialog. This supports standard messages such as showwarning, askquestion, showinfo etc. First we need to import tkMessagebox module after the import statement for Tkinter at the top of the Program.

```
import tkMessageBox
```

We now go back to our storeResults method and call tkMessagebox to inform the user that their response is stored, then clear the form by calling the clearResponse method.

```
              tkMessageBox.showinfo("Questionnaire", "Questionnaire Submitted")
              self.clearResponse()
```

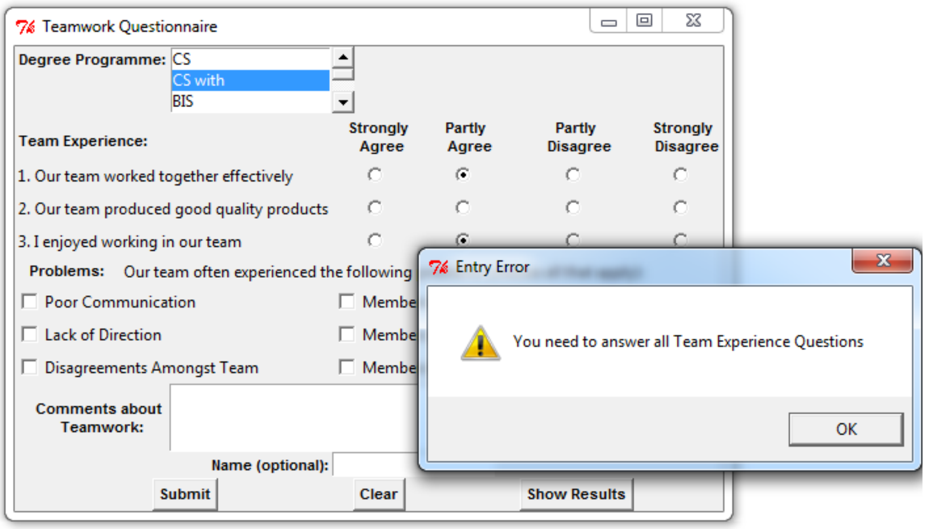
Finally we will display the error messages in a warning box (if strMsg was not empty) .

```
        else:
                tkMessageBox.showwarning("Entry Error", strMsg)
```

Exercise 6

Put in the code for setting up the buttons, clearing the response and storing the response in Questionnaire.py. Don't forget to create the Response class and store this in Response.py.

The screen shot on the right shows you what happens if you forget to select one of the team experience questions.



Teamwork Questionnaire

Degree Programme: CS / CS with / BIS

Team Experience:	Strongly Agree	Partly Agree	Partly Disagree	Strongly Disagree
1. Our team worked together effectively	○	●	○	○
2. Our team produced good quality products	○	○	○	○
3. I enjoyed working in our team	○	●	○	○

Problems: Our team often experienced the following

☐ Poor Communication ☐ Membe

☐ Lack of Direction ☐ Membe

☐ Disagreements Amongst Team ☐ Membe

Comments about Teamwork:

Name (optional):

Entry Error

You need to answer all Team Experience Questions

Displaying the Results

We will create an interface to show the results from the questionnaire.

	0	1	2	3	4	5	6	7	8	9
0	**Degree Programme:**				**All**	**CS**	**CS with**	**BIS**	**SE**	**Joints**
1	Number of Responses:				x	x	x	x	x	x
2	**Team Experience:**									
3	(Score: 4= Strongly Agree to 1=Strongly Disagree)									
4	1. Our team worked together effectively				x	x	x	x	x	x
5	2. Our team produced good quality products				x	x	x	x	x	x
6	3. I enjoyed working in our team				x	x	x	x	x	x
7	**Problems Experienced:**									
8	Poor Communication				x	x	x	x	x	x
9	Members Missing Meetings				x	x	x	x	x	x
10	Lack of Direction				x	x	x	x	x	x
11	Members Not Contributing				x	x	x	x	x	x
12	Disagreements Amongst Team				x	x	x	x	x	x
13	Members Not Motivated				x	x	x	x	x	x
14										

We could display the results using labels in a grid, which will need a lot of widgets. Instead we will use a textbox and format the text.

We will create a new script file DisplayResults.py. We will use the Pack Geometry Manager because the interface is much simpler than before (however, don't use grid and pack in the same window)

```
from Tkinter import *
from Response import Response

class DisplayResults(Frame):
    # GUI Setup

    def __init__(self, master):
         # Initialise Questionnaire Class

         Frame.__init__(self, master)
        self.pack()

# Main
root = Tk()
root.title("Display Results")
app = DisplayResults(root)
root.mainloop()
```

Retrieving the Results

We will create the retreiveResponse method to retrieve the results.

```
def retrieveResponse(self):
```

First we need to set variables for all programmes and for each programme to: count the responses; sum the scores for each team experience question and to sum the occurrences that each problem was encountered. We will set all these variables to 0 (integer), except the team experience questions which will be set to 0.0 (float). Some examples are given below:

```
      countAll = 0
      sumQ1SE = 0.0
      sumP5Joints = 0
```

We will now open responsedb and start to retrieve the data.

```
      import shelve
       db=shelve.open('responsedb')
      respNo = len(db)
```

We retrieve responses from responsedb one at a time.

```
      for i in range (1, respNo):
             Ans = get.db(str(i))
```

Update the variables with the values from the current response object

```
            countAll +=1
            sumQ1All += Ans.q1
            sumQ2All += Ans.q2
            sumQ3All += Ans.q3
            sumP1All += Ans.pr1
            sumP2All += Ans.pr2
            sumP3All += Ans.pr3
            sumP4All += Ans.pr4
            sumP5All += Ans.pr5
            sumP6All += Ans.pr6
```

Check which programme corresponds to the current response record and update its variables

```
           if Ans.prog == "CS":
                countCS +=1
                sumQ1CS += Ans.q1
```

And so on for all other variables for CS. Repeat with a similar if statement for all other programmes. We then close the database (*note: this is not part of the for loop*).

```
      db.close
```

We now want to display the results in a text box. First we set up the textbox and configure tags for normal and bold font.

```
      self.txtDisplay = Text(self, height=14,width=85)
      self.txtDisplay.tag_configure('boldfont', font =('MS', 8, 'bold'))
      self.txtDisplay.tag_configure('normfont', font =('MS', 8))
```

If you want to run your code to display the window as you go along then use the pack method. However, this should remain as the last line of the retrieveResults method when you add further code.

```
      self.txtDisplay.pack()
```

We lay out the first line of the display making use of tab characters (\t) to help with layout.

```
       tabResults = ""
       tabResults += ("\t" + "\t" + "\t" + "\t" + "\t")
       self.txtDisplay.insert(END, "Degree Programme" + tabResults + "ALL" + "\t"
                            + "CS" + "\t" + "CS with" + "\t" + "BIS" + "\t"
                            + "SE" + "\t" + "Joints" +'\n', 'boldfont')
```

We then display the count for all responses and for each programme

```
       self.txtDisplay.insert(END, "Number of Responses:" + tabResults + str(countAll)
                       + "\t" + str(countCS) + "\t" + str(countCSwith) + "\t" +
                      str(countBIS)+ "\t"+ str(countSE) + "\t"+ str(countJoints)
                      +'\n', 'normfont')
```

Add lines to display the team experience heading and explanation of the scores.

We then want to show the average scores for the team experience questions. This is done by taking the sum of the values for each question and dividing it by the count of responses However if any of our count variables are zero we will get a divide by zero error. In this case we would just want to show our questions as zero.

```
        if countAll > 0:
             Q1All = sumQ1All/countAll
             Q2All = sumQ2All/countAll
             Q3All = sumQ3All/countAll
         else:
             Q1All = 0
             Q2All = 0
             Q3All = 0
```

We need to similar checks and setting of appropriate variables for each programme. We can then display our answers to one decimal place using string formatting (%.1f).

```
     self.txtDisplay.insert(END, "1. Our team worked together effectively" + tabResults
                            + "%.1f" % Q1All + "\t %.1f" % Q1CS + "\t %.1f" % Q1CSwith
                            + "\t %.1f" % Q1BIS+ "\t %.1f" % Q1SE + "\t %.1f" %
                            Q1Joints +'\n', 'normfont')
```

We need to repeat appropriately for the other two team experience questions. We can then add a line to show the Problem Experienced heading.

We then want to show the percentages for the problems experienced by teams. This is done by multiplying the sum of the values for each question by 100 and dividing by the count of responses. Once again we need to deal with any divide by zero errors if the counts are zero. We can add the appropriate to the previous if statements.

Where count > 0 we add appropriate code to each if part, for example:

```
          P1All = sumP1All*100/countAll
```

And for the else part, for example:

```
          P1All = 0
```

And so on for all problems for each programme. The result is displayed using string formatting (%d). We also need to print out the % character.

```
      self.txtDisplay.insert(END, "Poor Communication" + tabResults + "%d" % P1All +
                             "% \t" + "%d" % P1CS + "% \t" + "%d" % P1CSwith + "% \t"
                             + "%d" % P1BIS +  "% \t" + "%d" % P1SE + "% \t" + "%d" %
                             P1Joints + "% \n", 'normfont')
```

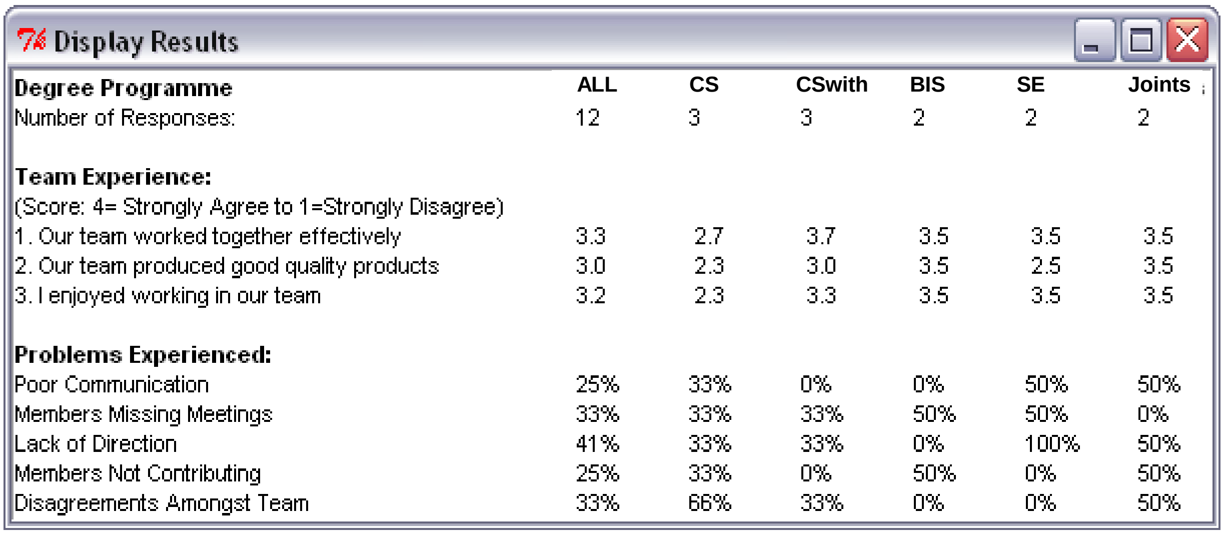
We need to repeat appropriately for the other five problems.

Finally we disable the textbox so the user cannot alter its contents just before the the pack method for txtDisplay.

```
      self.txtDisplay['state'] = DISABLED
```

Exercise 6

Put in the code for displaying the Results in DisplayResults.py. Run this and you should get the window shown on the next page:



Display Results

Degree Programme	ALL	CS	CSwith	BIS	SE	Joints
Number of Responses:	12	3	3	2	2	2
Team Experience:						
(Score: 4= Strongly Agree to 1=Strongly Disagree)						
1. Our team worked together effectively	3.3	2.7	3.7	3.5	3.5	3.5
2. Our team produced good quality products	3.0	2.3	3.0	3.5	2.5	3.5
3. I enjoyed working in our team	3.2	2.3	3.3	3.5	3.5	3.5
Problems Experienced:						
Poor Communication	25%	33%	0%	0%	50%	50%
Members Missing Meetings	33%	33%	33%	50%	50%	0%
Lack of Direction	41%	33%	33%	0%	100%	50%
Members Not Contributing	25%	33%	0%	50%	0%	50%
Disagreements Amongst Team	33%	66%	33%	0%	0%	50%

Opening the Results Window from the Questionnaire Window

Finally, we want to be able to open the results window from the Questionnaire Window. We need to import the DisplayResults module at the top of Questionnaire.py

```
from DisplayResults import *
```

We will need to add a new button to the CreateButtons method in Questionnaire.py. This should call the openResultsWindow method when the View Results button is pressed. You may want to reassign the column variables for the buttons to make them look better.

The code for the OpenResultsWindow method is given below:

```
  def openResultsWindow(self):

        t1 = Toplevel(root)
        DisplayResults(t1)
```

This uses the TopLevel widget to create a child window (Display Results) that acts independently from the root window (Questionnaire). However, if the root window is closed then it will also close the child window.

Exercise 7

Update your code in Questionnaire.py so it can displaying the Results window when the View Results button is pressed.

And Finally ….

There are a number of other widgets that you may find useful in developing interfaces, such as Canvas Widgets for displaying images and drawing simple shapes, Menu Widgets for providing Menu bars for applications. The widgets we have used have many other values and methods. There are also specialist widgets available (e.g. Python megawidgets *Pmw*) that you may also find useful.

There are many good online references for Tkinter. A good starting point is:
An Introduction to Tkinter By Fredrik Lundh, which can be found at:
http://effbot.org/tkinterbook/