Tkintertoy Documentation

Release 1.6.0

Mike Callahan

Aug 07, 2023

Contents:

CHAPTER 1

Tkintertoy 1.6 Tutorial

Date Aug 07, 2023

Author **Mike Callahan**

1.1 Introduction

Tkintertoy grew out of a GIS Python (mapping) class I taught at a local college. My students knew GIS but when it came time to put the workflows into a standalone application, they were stumped with the complexity of programming a GUI, even with *Tkinter*. So I developed an easy to use GUI library based on *Tkinter* that made it much simpler to code applications. After several trials, the result was *Tkintertoy* which is easy to use, but also can be create more complex GUIs. I have been teaching a Python class in a local vocational technical college using *Tkintertoy* with great success.

With this version, I have fixed a few minor bugs, improved the documentation, improved the operation of the library, and cleaned up the code for version 1.6. Support for Python 2 was removed since the library is no longer tested using Python 2.

Tkintertoy creates `Windows` which contain widgets. Almost every *tk* or *ttk* widget is supported and a few combined widgets are included. Most widgets are contained in a `Frame` which can act as a prompt to the user. The widgets are referenced by string *tags* which are used to access the widget, its contents, and its containing Frame. All this information is in the `content` dictionary of the Window. The fact that the programmer does not need to keep track of every widget makes interfaces much simpler to write, one only needs to pass the window. Since the widgets are multipart, I call them **ttWidgets**.

Tkintertoy makes it easy to create groups of widgets like radio buttons, check boxes, and control buttons. These groups are referenced by a single tag but individual widgets can be accessed through an index number. While the novice programmer does not need to be concerned with details of creating and assigning a tk/ttk widget, the more advanced programmer can access all the tk/ttk options and methods of the widgets. Tkintertoy makes sure that all aspects of tk/ttk are exposed when the programmer needs them. *Tkintertoy* is light-weight wrapper of *Tkinter* and can be used a gentle introduction to the complete library.

1.2 The Zen of Tkintertoy

1. It must be very simple to use. Not much more complicated than `input` or `print`.
2. It must produce well-balanced and clean, if simple, interfaces.
3. It must be very light-weight and easy to install. Everything is basically in one file, tt.py.
4. It must be based on Tkinter. Tkinter is still the default Gui library for Python. After working in Tkintertoy, the student can easily move into more complex Tkinter.
5. The source code should be easy to follow.

In the following examples below, one can see how the ideas in Tkintertoy can be used to create simple but useful GUIs. GUI programming can be fun, which puts the "toy" in *Tkintertoy*.

1.3 A "Hello World" Example

Let's look at a bare bones example of a complete GUI using imparative style. Imparative code are sometimes called scripts since their structure is simple. More complex code are ususally called applications.

This GUI will ask for the user's name and use it in a welcome message. This example uses these widgets: **ttEntry**, **ttLabel**, and **ttButtonbox**.

In relating this application to a command-line application, the entry replaces the `input` function, the label replaces the `print` function, and the buttonbox replaces the Enter key. Below is the code followed by an explanation of every line:

```
from tkintertoy import Window
gui = Window()
gui.setTitle('My First Tkintertoy GUI!')
gui.addEntry('name', 'Type in your name')
gui.addLabel('welcome', 'Welcome message')
gui.addButton('commands')
gui.plotxy('name', 0, 0)
gui.plotxy('welcome', 0, 1)
gui.plotxy('commands', 0, 2, pady=10)
while True:
    gui.waitforUser()
    if gui.content:
        gui.set('welcome', 'Welcome ' + gui.get('name'))
    else:
        break
```

Here is a screen shot of the resulting GUI:

Here is an explanation of what each line does:

1. Import the `Window` code which is the foundation of Tkintertoy.
2. Create an instance of a `Window` object assigned to `gui`. This will initialize Tk, create a Toplevel window, create an application Frame, and create a `content` dictionary which will hold all the widgets.
3. Change the title of `gui` to "My First Tkintertoy GUI!". If you don't do this, the title of the `Window` will default to "Tk". If you want no title, make the argument '' (a null string) or None.
4. Add an **ttEntry** widget to `gui`. This will be the combination of a ttk.Entry in a ttk.LabelFrame. We are going to tag it with 'name' since that is what we going to collect there. However, the tag can be any string. All Tkintertoy widgets must have a unique tag which acts as the key for the widget in the `content` dictionary. However, most of the time the programmer does not access the `content` dictionary directly, *Tkintertoy* provides methods for this. The title of the Frame surrounding the Entry widget will be 'Type in your name'. Entry frame titles are a great place to put instructions to your user. If you don't want a title, just leave off this argument. *Tkintertoy* will use a plain ttk/tk.Frame instead. The default width of the Entry widget is 20 characters, but this, like many other options can be changed.
5. Add a **ttLabel** widget to `gui`. This will be the combination of a ttk.Label in a ttk.LabelFrame. This tag will be 'welcome' since this where the welcome message will appear. Labels are a good widget for one line information to appear that the user cannot edit. The explanation to the user of the type of information displayed in the **ttLabel** is displayed in the LabelFrame, just like in the **ttEntry**
6. Add a **ttButtonbox** row with a tag of 'commands'. It defaults to two ttk.Buttons, labeled 'Ok' and 'Cancel' contained in a unlabeled ttk.Frame. Each button is connected to a function or method, called a "callback" which will execute when the user clicks on that button. The default callback for the 'Ok' button is the `breakout` method which exits the GUI processing loop but keeps displaying the window. This will be explained below. The 'Cancel' button callback is the `cancel` method which exits the loop, removes the window, and empties the `content` dictionary. Of course, the button labels and these actions can be easily modified by the programmer, but by providing a default pair of buttons and callbacks, even a novice programmer can create a working GUI application quickly. No callback programming is necessary.
7. Place the 'name' ttwidget at column 0 (first column), row 0 (first row) of `gui` centered. The second argument is the column (x dimension counting from zero) and the third argument is the row (y dimension). Both these value default to 0 but it is a good idea to always include them. The `plotxy` method is basically the tk `grid` method with the column and row keywords arguments specified. All other keyword arguments to `grid` can be used in `plotxy`. Plot was selected as a better word for a novice. However, `grid` will also work. Until a widget is plotted, it will not appear. However, the `gui` window is automatically plotted. Actually, you are plotting the ttk.LabelFrame, the ttk.Entry widget is automatically plotting in the Frame filling up the entire frame using *sticky='nswe'*.
8. Place the 'welcome' widget at column 0, row 1 (second row) of `gui` centered. There is a 3 pixel default vertical spacing between widget rows.
9. Place the 'command' widget at column 0, row 2 of `gui` centered with a vertical spacing of 10 pixels with *pady=10*.
10. Begin an infinite loop.
11. Wait for the user to press click on a button. The `waitforUser` method is a synonym for the tk `mainloop` method. Again, the name was changed to help a novice programmer. However, `mainloop` will also work. This method starts the event processing loop and is the heart of all GUIs. It handles all key presses and mouse clicks. Nothing will happen until this method is running. This loop will continue until the user clicks on the either the 'Ok' or 'Cancel' button. Clicking on close window system widget will have the same action as clicking on the 'Cancel' button. This action is built-in to all *Tkintertoy* windows.
12. To get to this line of code, the user clicked on a button. Test to see if the `content` dictionary contains anything. If it does, the user clicked on the 'Ok' button. Otherwise, the user clicked on the 'Cancel' button.

13. To get to this line of code, the user clicked on the 'Ok' button. Collect the contents of 'name' and add it to the "Welcome" string in 'welcome'. This shows how easy it is to get and set the contents of a widget using the given methods. To get the value of a widget call the `get` method. To change the value of any widget call the `set` method. The type of widget does not matter, `get` and `set` work for all widgets. Since all widgets are contained in the `content` directory of `gui`, the programmer does not need to keep track of individual widgets, only their containing frames or windows. Again, the usually programmer does not access `content` directly, they should use `get` and `set` methods.
14. This line of code is reached only if the user clicked on 'Cancel' which emptied the `content` directory. In this case, the user is finished with the application.
15. Break the infinite loop and exit the program. Notice the difference between the infinite application loop set up by the `while` statement and the event processing loop set up by the `waitforUser` method. Also, note that when the user clicked on 'Cancel', the tkintertoy code exited, but the Python code that called tkintertoy was still running. This is why you must break out of infinite loop.

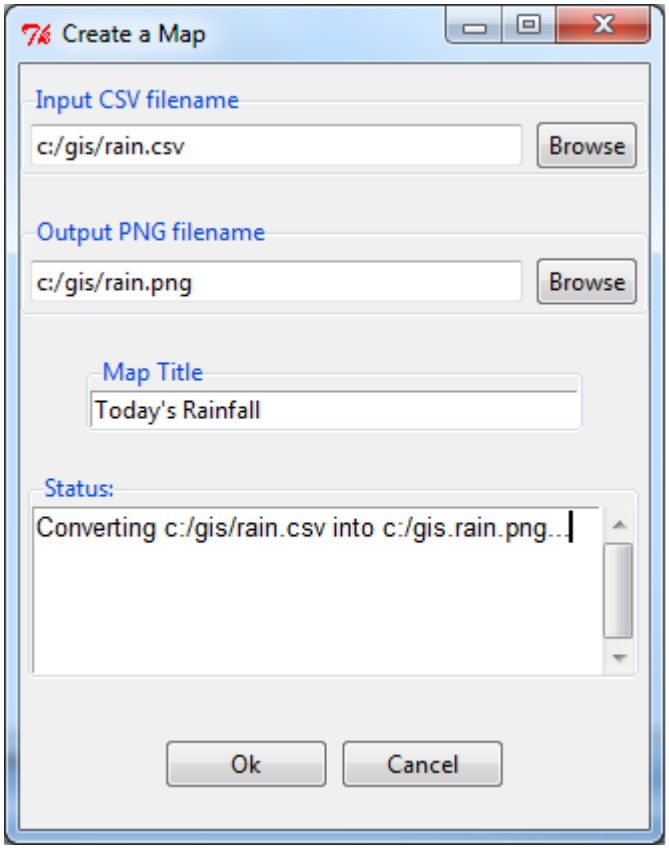
So you can see, with 15 lines of code, Tkintertoy gives you a complete GUI driven application, which will run on any platform Tkinter runs on with little concern of the particular host. Most *Tkintertoy* code is cross platform.

1.4 Simple Map Creation Dialog

Below is the code to create a simple dialog window which might be useful for a GIS tool which creates a map. This example was also written in imparative style in order to help the typical GIS or novice Python script writer. Procedure and object-oriented style coding will be demonstrated later.

We will need the filename of the input CSV file, the output PNG map image, and the title for the map. We will use the following widgets: **ttOpen**, **ttSaveAs**, **ttEntry**, and **ttText** as a status window.

We want the layout for the dialog to look like this:



Here is the code (we will not worry not the code that actually creates the map!):

```
from tkintertoy import Window
gui = Window()
gui.setTitle('Create a Map')
csv = [('CSV files', ('*.csv'))]
gui.addOpen('input', 'Input CSV filename', width=40, filetypes=csv)
png = [('PNG files', ('*.png'))]
gui.addSaveAs('output', 'Output PNG filename', width=40, filetypes=png)
gui.addEntry('title', 'Map Title', width=40)
gui.addText('status', width=40, height=5, prompt='Status:')
gui.addButton('commands')
gui.plotxy('input', 0, 0, pady=10)
gui.plotxy('output', 0, 1, pady=10)
gui.plotxy('title', 0, 2, pady=10)
gui.plotxy('status', 0, 3, pady=10)
gui.plotxy('commands', 0, 4, pady=20)
gui.waitforUser()
if gui.content:
    message = f"Converting {gui.get('input')} into {gui.get('output')}...\n"
    gui.set('status', message)
    gui.master.after(5000)  # pause 5 seconds
    # magic map making code goes here...
    gui.cancel()
```

Each line of code is explained below:

1. Import the `Window` object from tkintertoy.
2. Create an instance of a `Window` and label it `gui`.

3. Set the title `gui` to "Create a Map".
4. We want to limit the input files to *.csv only*. This list will be used in the method in the next line. Notice, you can filter multiple types.
5. Add an **ttOpen** dialog widget. This is a combination of a ttk.Entry widget, a 'Browse' ttk. Button, and a ttk.LabelFrame. If the user clicks on the 'Browse' button, they will see a directory limited to CSV files. To allow the user to see the entire path, we changed the width of the entry to 40 characters.
6. We want to limit our output to .png only.
7. Add a **ttSaveAs** dialog widget. This is a combination of a ttk.Entry widget, a 'Browse' ttk. Button, and a ttk.LabelFrame. If the user clicks on the 'Browse' button, they will see a directory limited to PNG files. If the file already exists, an overwrite confirmation dialog will pop up.
8. Add an **ttEntry** widget that is 40 characters wide to collect the map title.
9. Add a **ttText** widget, which is a combination of a ttk.Text widget, a vertical ttk.Scrollbar, and a ttk.LabelFrame. It will have a width of 40 characters, a height of 5 lines, and will be used for all status messages. The **ttText** widget is extremelly useful for many different purposes.
10. Add a **ttButtonbox** with the default 'Ok' and 'Cancel' buttons.
11. Plot the 'input' widget at column 0, row 0, vertically separating widgets by 10 pixels.
12. Plot the 'output' widget at column 0, row 1, vertically separating widgets by 10 pixels. Notice this will cause a 20 pixel separation between the input and output widgets.
13. Plot the 'title' widget at column 0, row 2, vertically separating widgets by 10 pixels.
14. Plot the 'status' widget at column 0, row 3, vertically separating widgets by 10 pixels.
15. Plot the 'commands' widget at column 0, row 4, vertically separating widgets by 20 pixels. This will be 30 pixels from the status widget.
16. Enter the event processing loop and exit when the user clicks on a button. This script will execute once so there is no need for an infinte loop.
17. If the user clicked on the OK button do the following:
18. Create the status message.
19. Display the status message.
20. Pretend we are making a map but in reality just pause for 5 seconds so the user can see the status message.
21. This is where the actual map making code would begin.
22. Exit the program.

Notice, if the user clicks on the Cancel button, the program exits at line 17.

1.5 Selection Widgets

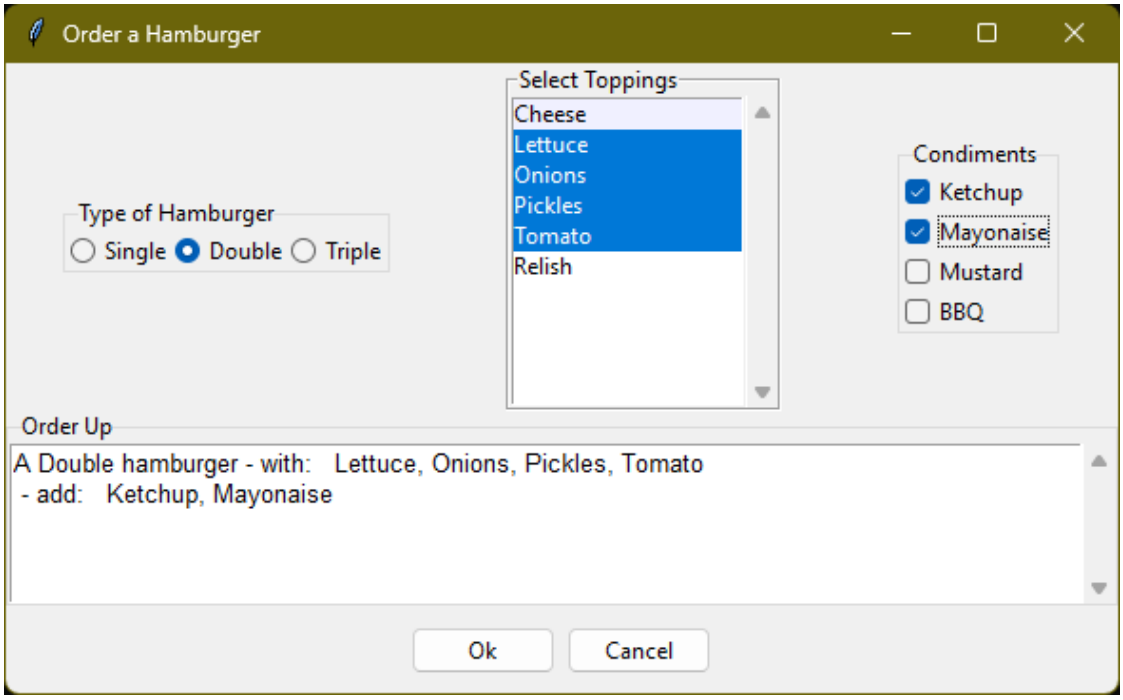
Many times you want to limit the user to a fixed set of options. This next example demonstrates widgets that are useful for this task. We will create a hamburger ordering application which will use three type of selection widgets: **ttRadiobox**, **ttCheckbox**, and **ttListbox**. We will stay with imparative style programming.

Radiobox widgets are great for showing the user an list of dependent options. Only one option in the group can be selected at a time. The name "radiobutton" comes from old-fashioned car radio tuner buttons, when you pushed one to change a station, the previous one selected poped-up.

Checkboxes allow the user to select many independent options at a time. Listboxes can be programmed to do both.

We will use a radiobox to select whether the user want a single, double, or a triple burger. We will use a listbox to indicate which toppings the user wants, and a checkbox to indicate the desired condiments.

Below is a screenshot of the application:



Here is the code:

```
from tkintertoy import Window
app = Window()
app.setTitle('Order a Hamburger')
burgerType = ['Single', 'Double', 'Triple']
app.addRadio('type', 'Type of Hamburger', burgerType)
toppings = ['Cheese', 'Lettuce', 'Onions', 'Pickles', 'Tomato', 'Relish']
app.addList('toppings', 'Select Toppings', toppings, selectmode='multiple')
condiments = ['Ketchup', 'Mayonaise', 'Mustard', 'BBQ']
app.addCheck('condiments', 'Condiments', condiments, orient='vertical')
app.addText('order', 'Order Up', height=5)
app.addButton('commands')
app.plotxy('type', 0, 0)
app.plotxy('toppings', 1, 0)
app.plotxy('condiments', 2, 0)
app.plotxy('order', 0, 1, columnspan=3)
app.plotxy('commands', 0, 2, columnspan=3, pady=10)

while True:
    app.waitforUser()
    if app.content:
        btype = app.get('type')
        toppings = app.get('toppings')
        condiments = app.get('condiments')
        app.set('order', f'A {btype} hamburger', allValues=True)
        if toppings:
            app.set('order', ' - with: ')
            tops = ', '.join(toppings)
```

(continues on next page)

(continued from previous page)

```
            app.set('order', f'  {tops}\n')
        else:
            app.set('order', ' - plain\n')
        if condiments:
            app.set('order', ' - add: ')
            conds = ', '.join(condiments)
            app.set('order', f'  {conds}\n')
        app.reset('type')
        app.reset('toppings')
        app.reset('condiments')
    else:
        break

```

1. Import the `Window` object from tkintertoy.
2. Create an instance of a `Window` and label it `app`.
3. Set the title `app` to "Order a Hamburger".
4. Create a list of burger types.
5. Add a **ttRadiobox** which is a list of three ttk.Radiobuttons labeled with the type of burgers. These will be referenced with a single tag, 'type'. If we want to reference a single Radiobutton, we will use an index; [0], [1], or [2].
6. Create a list of burger toppings.
7. Add a **ttListbox** which is a tk.Listbox with a vertical tk.Scrollbar. The elements are the items in the list of toppings. Notice that *selectmode='multiple'* so the user will be able to select multiple toppings without pressing the control or shift keys. This is a good example of when a listbox is useful for multiple options. While it does take up screen space, it makes it easy to select many multiple options but restricts the user to a fixed set of options.
8. Create a list of condiments.
9. Create a **ttCheckbox** which is a list of three ttk.Checkbuttons labeled with the condiments. The orientation will be vertical. This is another widget where the user can select multiple options. It is best used with a small number of options.
10. Add a **ttText** with a height of 5. This is where the order will appear. Note that the width of the text widget determines the width of the entire application.
11. Add a **ttButtonbox** with the default 'Ok' and 'Cancel' buttons.
12. Plot the 'type' widget at column 0, row 0.
13. Plot the 'toppings' widget at column 1, row 0.
14. Plot the 'condiments' widget at column 2, row 0.
15. Plot the 'order' widget at column 0, row 1, strectched across three columns with *colunmspan=3*.
16. Plot the 'commands' widget at column 0, row 2, also stretched across three columns.
17. Blank line
18. Begin a infinite loop.
19. Enter the event processing loop and exit when the user clicks on a button.
20. If the user clicked on the OK button do the following:
21. Get the burger type.

22. Get the selected toppings list.
23. Get the selected condiments list.
24. Start the order message. The *allValue=True* clears the text widget of any previous orders.
25. If the user selected any toppings...
26. Add the toppings phrase in the 'orders' widget.
27. Create a string containing the selected toppings separated by a comma.
28. Add it to the 'orders' widget.
29. If the user selected no toppings...
30. Mark the burger as plain.
31. If the user selected any condiments...
32. Add the condiments phrase.
33. Create a string containing the selected condiments separated by a comma.
34. Add it to the order.
35. Reset the 'type' widget.
36. Reset the 'toppings' widget.
37. Reset the 'condiments' widget and loop back to 19.
38. If the user clicked on the 'Cancel' button...
39. Break the infinate loop. The *Tkintertoy* application was automatically canceled.

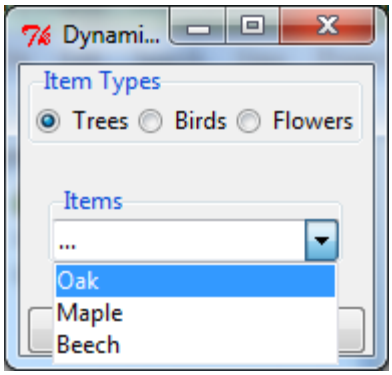
This is a example showed some of the selection widgets that are available in *Tkintertoy*. The best one to use is up to the programmer's discretion. As you can see, this code is getting too long for imparative style. We will use procedure style in the next example.

1.6 Dynamic Widgets

A very useful technique is to create a widget which is dependent on the contents of another widget. The code below shows a **ttCombobox** which is dependent on a **ttRadiobox** row.

The trick to have the contents of a combobox be dependent on a radiobox, is to create a combo widget and then create a callback function which looks at the contents of the radiobox and then sets the item list attribute of the combo widget. This time we will use procedure style code which is a more advanced style but still accessable to the novice programmer. We will also do a better job in adding comments to the code.

Here is the screenshot:



The callback function will have to know the widget that called it which is included when the Window is passes as an argument, which will lead to some strange looking code. This complexity can be eliminated by writing in an object-oriented fashion, which will be covered in the next example.

Below is the code:

```
from tkintertoy import Window

def update(gui): # callback function
    """ set the alist attribute by what is in the radio button box """
    lookup = {'Trees':['Oak','Maple','Beech'],
        'Birds':['Cardinal','Robin','Sparrow'],
        'Flowers':['Rose','Petunia','Daylily']}
    select = gui.get('category')
    gui.set('items', lookup[select], allValues=True)
    gui.set('items', '...')

def main():
    """ main driving function """
    categories = ['Trees','Birds','Flowers']
    gui = Window()
    gui.setTitle('Dynamic Widget Demo')
    gui.addRadio('category', 'Item Types', categories)
    gui.addCombo('items', 'Items', None, postcommand=(lambda : update(gui)))
    gui.addButton('command')
    gui.set('items', '...')
    gui.plotxy('category', 0, 0)
    gui.plotxy('items', 0, 1, pady=20)
    gui.plotxy('command', 0, 2)
    gui.waitforUser()
    if gui.content:
        selected = gui.get('category')
        item = gui.get('items')
        # more code would go here...
        gui.cancel()

main()
```

Below explains every line:

1. Import `Window` from tkintertoy.
2. Blank line.
3. Define the callback function, `update`. It will have a single parameter, the calling `Window`.
4. This is the function documentation string. It is a great idea to have a documentation string for every function

and method. Since we are using the triple quote our comment can exceed a single line.

5. These next three lines define the lookup dictionary.
6. Same
7. Same
8. Get the category the user clicked on. This shows an advantage of *Tkintertoy's* content directory. All widgets are included in the window. The programmer does not have to pass individual widgets.
9. Using this category as a key, set all the values in the **ttCombobox** widget list to the list returned. by the lookup dictionary, rather than the entry widget. This is why *allValues=True*.
10. Change the entry value of 'items' to '...' which is why *allValues=False*. This will overwrite any selection the user had made. The allValues option has different effects depending on the widget type.
11. Blank line.
12. Create the main function, `main`. It will have no parameters. Most Python applications have a main driving function.
13. The documentation line for `main`
14. Create the three categories.
15. Create an instance of `Window` assigned to `gui`.
16. Set the title for `gui`.
17. Add a **ttRadiobox** box using the categories.
18. Add a **ttCombobox** widget. This is a combination of a ttk.Combobox contained in a ttk.LabelFrame. This widget will update its items list whenever the user clicks on a radiobox button. This is an example of using the *postcommand* option for the combobox. Normally, *postcommand* would be assigned to a single method or function name. However, we need to include `gui` as an parameter. This is why `lambda` is there. Do not fear `lambda`. Just think of it as a special `def` command that defines a function in place.
19. Add a **ttButtonbox** with the default 'Ok' and 'Cancel' buttons.
20. Initialize the items widget entry widget to just three dots. This lets the user know there are selections available in the pulldown.
21. Plot the category widget at column 0, row 0.
22. Plot the items widget at column 0, row 1.
23. Plot the command buttons at column 0, row 2.
24. Start the event processing loop and wait for the user to click on a button. Notice that as the user clicks on a category button, the list in the items combobox changes and the event loop keeps running. We do not need an infinite loop.
25. If the user clicked on 'Ok' by seeing if content is not empty.
26. Retrieve the value of the category widget using the get method.
27. Retrieve the value of the items widget that was selected or typed in.
28. This where the actual processing code would start.
29. Exit the program. Calling `cancel` is the same as clicking on the Cancel button.
30. Blank line.
31. Call `main`. Even though we defined `main` above, Python will not execute the function until we call it.

1.7 Object-Oriented Dynamic Widgets

While I told you to not fear lambda, if you write code in an object-oriented mode, you don't have to be concerned about lambda. One can write complex guis in **Tkintertoy** without object-oriented style, which might be better for novice programmers, but most guis should be oject-oriented once the programmer is ready. While, the details of writing object-oriented code is far beyond the scope of this tutorial, we will look at the previous example in an object-oriented mode using composition. You will see, it is not really complicated at all, just a little different. The GUI design did not change.

Below is the new code:

```
from tkintertoy import Window

def update(gui): # callback function
    """ set the alist attribute by what is in the radio button box """
    lookup = {'Trees':['Oak','Maple','Beech'],
        'Birds':['Cardinal','Robin','Sparrow'],
        'Flowers':['Rose','Petunia','Daylily']}
    select = gui.get('category')
    gui.set('items', lookup[select], allValues=True)
    gui.set('items', '...')

def main():
    """ main driving function """
    categories = ['Trees','Birds','Flowers']
    gui = Window()
    gui.setTitle('Dynamic Widget Demo')
    gui.addRadio('category', 'Item Types', categories)
    gui.addCombo('items', 'Items', None, postcommand=(lambda : update(gui)))
    gui.addButton('command')
    gui.set('items', '...')
    gui.plotxy('category', 0, 0)
    gui.plotxy('items', 0, 1, pady=20)
    gui.plotxy('command', 0, 2)
    gui.waitforUser()
    if gui.content:
        selected = gui.get('category')
        item = gui.get('items')
        # more code would go here...
        gui.cancel()

main()
```

And the line explanations:

1. Import `Window` from tkintertoy.
2. Blank line.
3. Create a class called `Gui`. This will contain all the code dealing with the interface. We are not inheriting from a parent class in this example. We will see how to do this in another example below.
4. This is a class documentation string. It is a great idea to document all classes, too.
5. Blank line.
6. Create an initialize method that will create the interface, called `__init__`. This strange name is required. Methods names that begin and end with double underscore are special in Python.
7. This is the method documentation string.

8. Create the three categories.
9. Create an instance of `Window` assigned to `self.gui`. The self means gui is an attribute of the instance and all methods in the class will have access to `self.gui`.
10. Set the title for `self.gui`.
11. Add a **ttRadiobox** using the categories.
12. Add a **ttCombobox** widget which will update its items list whenever the user clicks on a radiobox button. Notice that the *postcommand* option now simply points to the callback method without `lambda` since ALL methods can access `self.gui`. This is the major advantage to object-oriented code. It reduces argument passing.
13. Add a **ttButtonbox** with the default 'Ok' and 'Cancel' buttons.
14. Initialize the items widget.
15. Plot the category widget at column 0, row 0.
16. Plot the items widget at column 0, row 1.
17. Plot the command buttons at column 0, row 2.
18. Blank line.
19. Create the callback method using the `self` parameter.
20. This is the method documentation string.
21. These next three lines define the lookup dictionary.
22. Same
23. Same
24. Get the category the user clicked on.
25. Using this category as a key, set all the items in the combobox widget list to the list returned by the lookup dictionary, rather than the entry widget, which is why *allValues=True*.
26. Clear the items widget.
27. Blank line.
28. Create the main driving function.
29. Main documentation string.
30. Create an instance of the `Gui` class labeled `app`. Notice that `app.gui` will refer to the `Window` created in the `__init__` method and `app.gui.content` will have the contents of the window.
31. Start the event processing loop and wait for the user to click on a button.
32. If the user clicked on Ok...
33. Retrieve the value of the category.
34. Retrieve the value of the entry part of the combobox.
35. This where the actual processing code would start.
36. Blank line.
37. Call main.

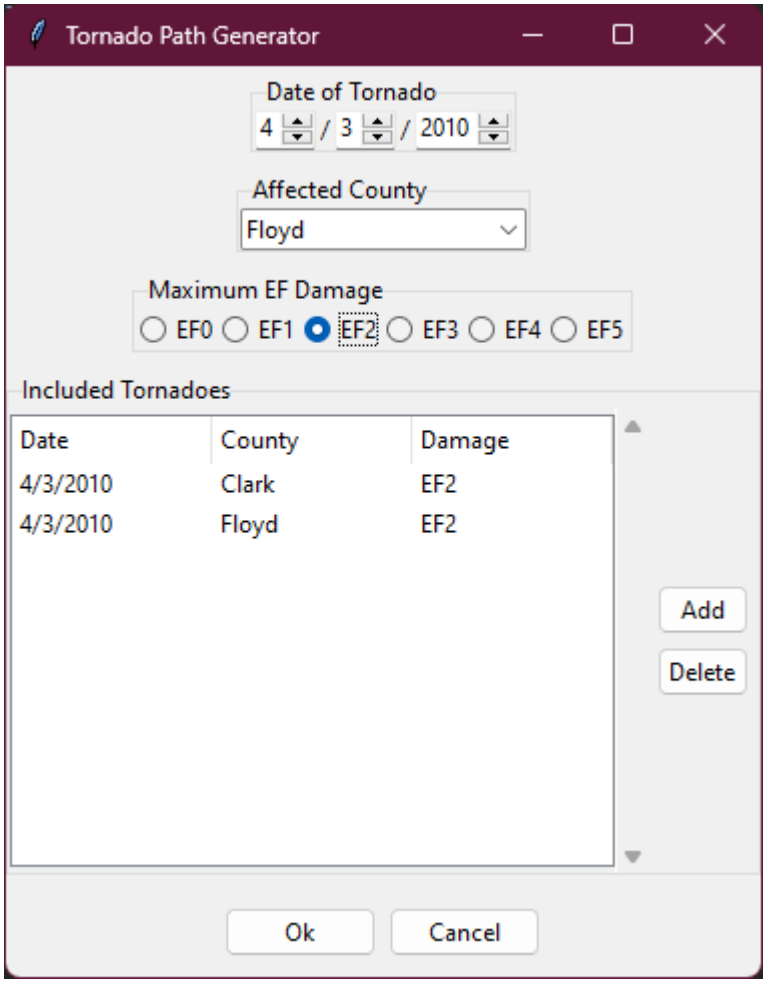
Notice if the user clicks on 'Cancel' there is no more code to execute.

There are very good reasons for learning this style of programming. It should be used for all except the simplest GUIs. You will quickly get use to typing "self." All future examples in this tutorial will use object-oriented style of coding.

1.8 Using the Collector Widget

This next example is the interface to a tornado path generator. Assume that we have a database that has tornado paths stored by date, counties that the tornado moved through, and the maximum damaged caused by the tornado (called the Enhanced Fajita or EF scale).

This will demonstrate the use of the **ttCollector** widget, which is a combination of a ttk.Treeview, and two ttk.Buttons. It acts as a dialog inside a dialog. Below is the screenshot:



You can see for the date we will use a **ttSpinbox**. A ttSpinbox is a group of tk/ttk.spinboxes that are limited to integers, separated by a string, and contained in a tk/ttk.Frame. This is a excellent widget for dates, times, social security numbers, etc. The `get` method will return s string with the values of each box, with the separtor in between. The `set` method also requires the separtor in the string.

The county will be a **ttCombobox** widget, the damage will use **ttCheckbox** and all choices will be shown in the **ttCollector** widget. Here is the code:

```
from tkintertoy import Window

class Gui(object):
    """ The Tornado Path Plotting GUI """

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
        """ create the GUI """
        counties = ['Clark','Crawford','Dubois','Floyd','Harrison','Jefferson
↪',
            'Orange','Perry','Scott','Washigton']
        damage = ['EF0','EF1','EF2','EF3','EF4','EF5']
        dateParms = [[2,1,12],[2,1,12],[5,1900,2100]]
        initDate = '1/1/1980'
        cols = [['Date', 100],['County', 100],['Damage', 100]]
        self.gui = Window()
        self.gui.setTitle('Tornado Path Generator')
        self.gui.addSpin('tdate', dateParms, '/', 'Date of Tornado')
        self.gui.set('tdate', initDate)
        self.gui.addCombo('county', 'Affected County', counties)
        self.gui.addRadio('level', 'Maximum EF Damage', damage)
        self.gui.addCollector('paths', cols, ['tdate','county','level'],
↪'Included Tornadoes',
            height=10)
        self.gui.addButton('command')
        self.gui.plotxy('tdate', 0, 0, pady=5)
        self.gui.plotxy('county', 0, 1, pady=5)
        self.gui.plotxy('level', 0, 2, pady=5)
        self.gui.plotxy('paths', 0, 3, pady=5)
        self.gui.plotxy('command', 0, 4, pady=10)

def main():
    """ the driving function """
    app = Gui()
    app.gui.waitforUser()
    if app.gui.content:
        data = app.gui.get('paths', allValues=True)
        print(data)
        # magic tornado path generation code
        app.gui.cancel()

main()
```

Here are the line explanations, notice the first steps are very similar to the previous example:

1. Import `Window` from tkintertoy.
2. Blank line.
3. Create a class called `Gui`. This will contain all the code dealing with the interface.
4. This is a class documentation string.
5. Blank line.
6. Create an initialize method that will create the interface. All methods in the class will have access to `self`.
7. This is the method documentation string.
8. Create a list of county names.
9. Same
10. Create a list of damage levels.
11. Create the parameter list for the date spinner. The first digit is the width in characters, the second is the lower limit, the third is the upper limit.
12. The initial date will be 1/1/1980.

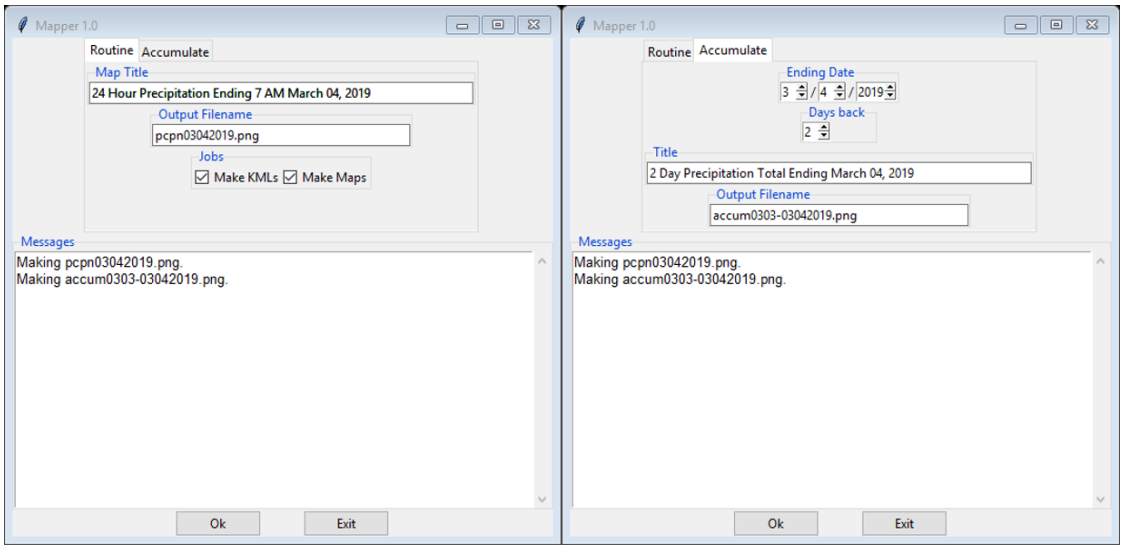
13. Set up the column headers for the **ttCollector** widget. The first value is the the header string, the second is the width of the column in pixels.
14. Create an instance of `Window` labeled `self.gui`. Again, the `self` means that every method in the class will have access. Notice, there are no other methods in this class so making gui an attribute of self is unnecessary. However, it does no harm, other programmers expect it, and future methods can be added easily.
15. Set the title of `self.gui` to "Tornado Path Generator".
16. Add a date **ttSpinbox**. This is a combination of 3 ttk.Spinboxes seperated by a slash (/) contained in a ttk.LabelFrame. It will be labeled 'tdate' in order to not cause any confusion with a common date library.
17. Set the 'tdate' to the default. Notice to set and value of a spinbox you use a string with seperators.
18. Add a county **ttCombobox**.
19. Add a damage level **ttCheckbox**.
20. Add a **ttCollector**. The collector has a tag, the column header list from line 13, a list of the widget tags it needs to collect, and the propmt. It also includes two buttons, 'Add' and 'Delete'. Clicking on 'Add' will collect the values in the widgets and add them in a line in the treeview. Clicking on 'Delete' will delete the currently selected line in the treeview.
21. Same.
22. Add a **ttButtonbox** with the default 'Ok' and 'Cancel' buttons.
23. Plot the 'tdate' widget at column 0, row, 0, separating the widgets by 5 pixels.
24. Plot the 'county' widget at column 0, row 1, separating the widgets by 5 pixels.
25. Plot the 'damage' level widget at column 0, row 2, separating the widgets by 5 pixels.
26. Plot the 'path' widget at column 0, row 3, separating the widgets by 5 pixels.
27. Plot the 'command' widget at column 0, row 4, separating the widgets by 10 pixels.
28. Blank line.
29. Create a `main` function.
30. This is the function documentation.
31. Create an instance of the `Gui` class which will create the GUI.
32. Start the event processing loop
33. If the user clicked on 'Ok'...
34. Get all the lines in the collector as a list of lists.
35. This is where the tornado path generation code would begin but we are just going to print the data in a pop-up information window. The example gives [['4/3/2010', 'Clark', 'EF2'], ['4/3/2010', 'Floyd', 'EF2']].
36. Call the driving function.

When you click on 'Add', the current selections in 'tdate', 'counties', and 'level' will be added into the collector widget in a row. If you select a row and click on 'Delete', it will be removed. Thus the collector acts as a GUI inside of a GUI, being fed by other widgets. If this was a real application, we would generate a tornado path map of the EF-2 tornadoes that moved through Clark and Floyd counties on April 4, 2010.

1.9 Using the Notebook Container

Tkintertoy includes containers which are `Windows` within `Windows` in order to organize widgets. A very useful one is the **ttNotebook** which is a ttk.Notebook. This example shows a notebook that combines two different map making methods into a single GUI. This will use the following widgets: **ttEntry**, **ttCheckbox**, **ttText**, **ttSpinbox**, and **ttButtonbox**. The style of code will stay with composition.

Below is a screenshot:



Here is the code. We will also demonstrate to the set and get the contents of more widgets and introduce some simple error trapping:

```
import datetime
from tkintertoy import Window

class Gui:
    """ the GUI for the script """
    def __init__(self):
        """ create the interface """
        self.dialog = Window()
        self.dialog.setTitle('Mapper 1.0')
        # notebook
        tabs = ['Routine', 'Accumulate']
        pages = self.dialog.addNotebook('notebook', tabs)
        # routine page
        self.routine = pages[0]
        today = datetime.date.today()
        self.dt = today.strftime('%d,%m,%Y,%B').split(',')
        self.routine.addEntry('title', 'Map Title', width=60)
        self.routine.set('title', '24 Hour Precipitation Ending 7 AM {0[3]}
↪{0[0]}, {0[2]}'.format(
            self.dt))
        self.routine.plotxy('title', 0, 0)
        self.routine.addEntry('outfile', 'Output Filename', width=40)
        self.routine.set('outfile', 'pcpn{0[1]}{0[0]}{0[2]}.png'.format(self.
↪dt))
        self.routine.plotxy('outfile', 0, 1)
        jobs = ['Make KMLs', 'Make Maps']
```

(continues on next page)

(continued from previous page)

```
        self.routine.addCheck('jobs', 'Jobs', jobs)
        self.routine.set('jobs', jobs)
        self.routine.plotxy('jobs', 0, 2)
        # accum pcpn page
        self.accum = pages[1]
        parms = [[3, 1, 12], [3, 1, 31], [5, 2000, 2100]]
        self.accum.addSpin('endDate', parms, '/', 'Ending Date',
            command=self.updateAccum)
        self.accum.set('endDate', f'{today.month}/{today.day}/{today.year}')
        self.accum.plotxy('endDate', 0, 0)
        self.accum.addSpin('daysBack', [[2, 1, 45]], '', 'Days back',
            command=self.updateAccum)
        self.accum.set('daysBack', '2')
        self.accum.plotxy('daysBack', 0, 1)
        self.accum.addEntry('title', 'Title', width=60)
        self.accum.plotxy('title', 0, 2)
        self.accum.addEntry('outfile', 'Output Filename', width=40)
        self.accum.plotxy('outfile', 0, 3)
        self.updateAccum()
        # dialog
        self.dialog.addText('messages', 'Messages', width=70, height=15)
        self.dialog.plotxy('messages', 0, 1)
        self.dialog.addButton('commands', space=20)
        self.dialog.setWidget('commands', 0, command=self.go)
        self.dialog.setWidget('commands', 1, text='Exit')
        self.dialog.plotxy('commands', 0, 2)
        self.dialog.plotxy('notebook', 0, 0)
        self.dialog.set('notebook', 'Routine')

    def updateAccum(self):
        """ update widgets on accum page """
        end = [int(i) for i in self.accum.get('endDate').split('/')]
        endDate = datetime.date(end[2], end[0], end[1])
        endDateFmt = endDate.strftime('%d,%m,%Y,%B').split(',')
        daysBack = self.accum.get('daysBack')[0]
        self.accum.set('title', '{0} Day Precipitation Total Ending {1[3]}
↪{1[0]}, {1[2]}'.format(
            int(daysBack), endDateFmt))
        begDate = endDate - datetime.timedelta(int(self.accum.get('daysBack
↪')[0]) - 1)
        begDateFmt = begDate.strftime('%d,%m').split(',')
        self.accum.set('outfile', 'accum{0[1]}{0[0]}-{1[1]}{1[0]}{1[2]}.png'.
↪format(
            begDateFmt, endDateFmt))

    def go(self):
        """ get current selected page and make map """
        run = self.dialog.get('notebook')               # get selected tab␣
↪number
        mapper = Mapper(self)                       # create a Mapper␣
↪instance using the Gui
                                                    # instance which is␣
↪self
        try:
            if run == 'Routine':
                mapper.runRoutine()
            elif run == 'Accumulate':
```

(continues on next page)

(continued from previous page)

```
                mapper.runAccum()
        except:
            self.dialog.set('messages', self.dialog.catchExcept())

class Mapper:
    """ contain all GIS methods """

    def __init__(self, gui):
        """ create Mapper instance
            gui: Gui object """
        self.gui = gui

    def runRoutine(self):
        """ make the routine precipitation maps """
        title = self.gui.routine.get('title')
        filename = self.gui.routine.get('outfile')
        self.gui.dialog.set('messages', f'Making {filename}.\n')
        # magic map making code goes here

    def runAccum(self):
        """ make the accumulate precipitation map """
        title = self.gui.accum.get('title')
        filename = self.gui.accum.get('outfile')
        self.gui.dialog.set('messages', f'Making {filename}.\n')
        # magic map making code goes here

def main():
    gui = Gui() # create a Gui instance and pass Mapper class to it
    gui.dialog.waitforUser()

if __name__ == '__main__':
    main()
```

Here are the line explanations:

1. Import datetime for automatic date functions
2. Import `Window` from tkintertoy.
3. Blank line.
4. Create a class called `Gui`. This will contain the code dealing with the interface.
5. Class documentation string.
6. Create an initialize method that will create the interface. All methods in the class will have access to `self`.
7. This is the method documentation string.
8. Create an instance of `Window` that will be asignned to an attribute `dialog`. All methods in this class will have access.
9. Set the title of the window to Mapper 1.0.
10. This code section is for the notebook widget.
11. Create a list which contains the names of the tabs in the notebook: 'Routine' & 'Accumulate'. 'Routine' will make a map of one day's rainfall, 'Accumulate' will add up several days worth of rain.
12. Add a **ttNotebook**. The notebook will return two `Windows` in a list which will be used as a container for each notebook page.

13. This code section is for the 'Routine' notebook page.
14. Assign the first page (page[0]) of the notebook, which is a `Window` to an attribute `routine`.
15. Get today's date.
16. Convert it to [date, month, year, month abr]; ex. [24, 6, 2023, 'Jun']
17. Add a title **ttEntry** widget. This will be filled in dynamically and be the title of the map.
18. Set the title using today's date.
19. Same.
20. Plot the title at column 0, row 0.
21. Add an output filename **ttEntry** widget. This will also filled in dynamically.
22. Set the output filename using today's date.
23. Plot the output filename widget at column 0, row 1.
24. Create a list of two types of jobs: Make KMLs & Make Maps.
25. Add a jobs **ttCheckbox**.
26. Turn on both check boxes, by default.
27. Plot the jobs widget at column 0, row 2.
28. This code section is for the 'Accumulate' notebook page.
29. Assign the second page (page[1]) of the notebook, which is a `Window` to an attribute `accum`.
30. Create the list for the parameters of a date spinner.
31. Add an ending date **ttSpinbox**, with the callback set to self.updateAccum().
32. Same.
33. Set the ending date to today.
34. Plot the ending date widget at column 0, row 0.
35. Add a single days back **ttSpinbox** with the callback set to self.updateAccum() as well.
36. Same.
37. Set the default days back to 2.
38. Plot the days back widget at column 0, row 1.
39. Add a title **ttEntry**. This will be filled in dynamically.
40. Plot the title widget at column 0, row 2.
41. Add an output filename **ttEntry**. This will be filled in dynamically.
42. Plot the output filename widget at column 0, row 3.
43. Fill in the title using the default values in the above widgets.
44. This section of code is for the rest of the dialog window.
45. Add a messages **ttText**. This is where all messages to the user will appear.
46. Plot the messages widget at column 0, row 1 of the dialog window. The notebook will be at column 0, row 0.
47. Add a command **ttButtonbox**, the default are labeled Ok and Cancel.

48. Set the callback for the first button to the `go` method. We are changing the *command* parameter. This shows how easy it is to get to the more complex parts of Tk/ttk from tkintertoy. The `setWidget` allows the programmer to change any of the tk/ttk options after the widget is created.
49. Set the label of the second button to `Exit` using the same method as above but changing the *text* parameter. This shows how options of buttons can be dynamic.
50. Plot the command buttons at column 0, row 2.
51. Plot the notebook at column 0, row 0.
52. Set the default notebook page to 'Routine'. This will be the page displayed when the application first starts. Note that `set` and `get` use the notebook tab names.
53. Blank line.
54. This method will update the widgets on the 'Accumulate' tab.
55. This is the method documentation string.
56. Get the ending date from the widget. This is an example of a use of a list comprehension. The `get` method will return a date string. The `split` method will return a list of str, and the list comprehension convert the values to ints. The result will be [month, day, year].
57. This will turn the list of ints into a datetime object.
58. Turn the object into a comma-separated string 'date-int, month-int, year, month-abrev' like '24,6,2023,Jun'.
59. Get the number of days back the user wanted.
60. Set the title of the map in the title widget. As the user changes the dates and days back, this title will dynamically change. The user can edit this one last time before they click on 'Ok'.
61. Calculate the beginning date from the ending date and the days back.
62. Convert the datetime into a list of strings ['date-int','month-int'] like ['22','6'].
63. Same.
64. Set the title of the map file to something like 'accum06022-06242023'. Again, this will be dynamically updated and can be overridden. Notice that one method is updating two widgets.
65. Same.
66. Blank line.
67. This method will execute the correct the map generation code.
68. This is the method documentation string.
69. Get the selected notebook tab name.
70. Create an instance of a Mapper object. However, we have a chicken/egg type problem. Mapper must know about the Gui instance in order to send messages to the user. That is why the Mapper instance must be created after the Gui instance. However, the Gui instance must also know about the Mapper instance in order to execute the map making code. That is why the Mapper instance is created inside of this method. The Gui instance `self` is used as an argument to the Mapper initialization method. It looks funny but it works.
71. Blank line.
72. This code might fail so we place it in a try... except block.
73. If the current tab is 'Routine'...
74. Run the routine map generation code.
75. If the current tab is 'Accumulate'...

76. Run the accumulated map generation code.
77. Catch any exceptions.
78. Place all error messages into the messages widget. Any error messages will pop-up in a window.
79. Blank line.
80. Create a `Mapper` class which contains all the map generation code. This will be a stud here since map generation code is well beyond the scope of this tutorial.
81. Class documentation line.
82. Blank line.
83. Create an initialize method that will contain all the map making methods. For this example, this will be mainly stubs since actual GIS code is well beyond the scope of this tutorial.
84. Method documentation lines.
85. Same.
86. Make the Gui object an attribute of the instance so all methods have access.
87. Blank line.
88. This method contains the code for making the routine daily precipitation map.
89. Method documentation line.
90. Get the desired map title. This will be used in the magic map making code section.
91. Get the filename of the map.
92. Send a message to the user that the magic map making has begun.
93. This is well beyond the scope of this tutorial.
94. Blank line.
95. This method contains the code for making accumulated precipitation maps, that is, precipitation that fell over several days.
96. Method documentation line.
97. Get the desired map title. This will be used in the magic map making code section.
98. Get the filename of the map.
99. Send a message to the user that the magic map making has begun.
100. This is well beyond the scope of this tutorial.
101. Blank line.
102. The `main` function.
103. Create the GUI.
104. Run the GUI.
105. Blank line.
106. Standard Python. If you are executing this code from the command line, execute the main function. If importing, don't.

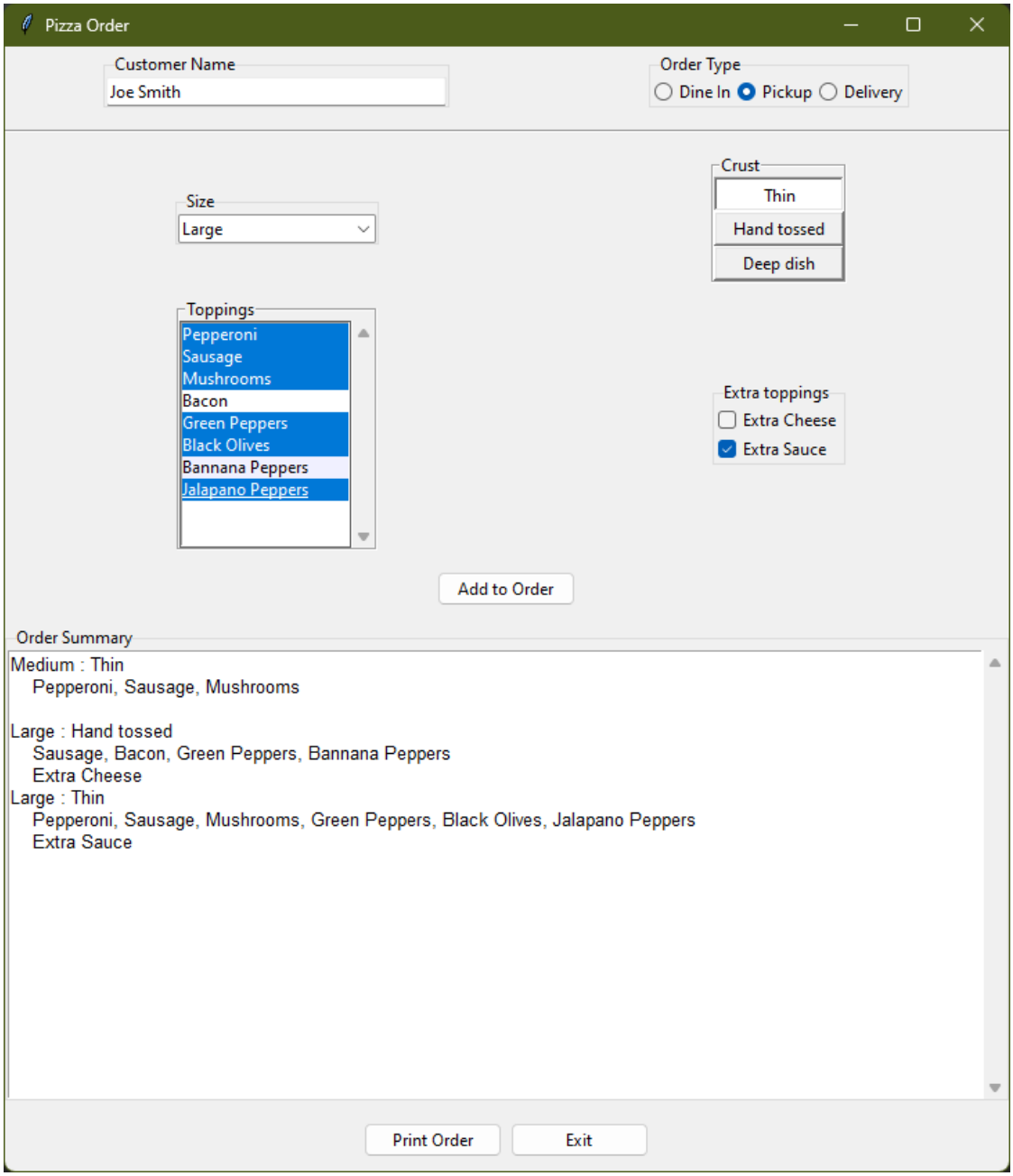
1.10 Object-Oriented Style Using Inheritance

This example gets away from map maiking and is a demonstation of writting in an object-oriented style using inheritance. This is the style most textbooks will use when explaining GUI creation. Inheritance means that the application window will inherit all the features of a **Tkintertoy** `Window`. So instead of refering to the tkintertoy window in the class as self.gui you would use just self. Think of composition as the application *has* a Window and inheritance as the application *is* a Window.

The example below is a pizza ordering system. It demostates several ttwidgets: **ttEntry**, **ttRadiobox**, **ttCombobox**, **ttLine**, two **ttCheckboxes** with the indicator off and on, **ttListbox**, **ttText**, and several **ttButtons**.

This application works as follows. The user first fills in the customer's name in the entry and how they are going to get their pizzas in a radio button group with the indicator on. Next, for every pizza, the user selects a size using a combo and crest type using a radio group with the indicator off. Next, they click on the the toppings the customer asked for using a scrolling list. Now, the user add extra cheese or extra sauce of both using a check group. Once the order for the pizza is complete, the user clicks on the `Add to Order` button. This sends the pizza order to the text box and clears the pizza option widgets, making ready to enter the next pizza. When all the pizzas are entered. The user clicks on `Print Order`, which here just prints the user's name, their delivery method, and their pizzas on the terminal. In real life this information would go to another system.

Below is a screenshot:



Here is the code. We will also demonstrate to the set and get the contents of more widgets and introduce some simple error trapping:

```
from tkintertoy import Window

class PizzaGui(Window):
    """ Create a pizza ordering GUI """

    def __init__(self):
        """ Create an instance of PizzaGui """
        super().__init__()

    def makeGui(self):
```

(continues on next page)

(continued from previous page)

```
        """ Make the GUI """
        self.setTitle('Pizza Order')
        toppings = ('Pepperoni','Sausage','Mushrooms','Bacon','Green Peppers
↪',
                    'Black Olives', 'Bannana Peppers', 'Jalapano Peppers')
        crusts = ('Thin', 'Hand tossed', 'Deep dish')
        orderType = ('Dine In', 'Pickup', 'Delivery')
        extras = ('Extra Cheese', 'Extra Sauce')
        sizes = ('Personal','Small','Medium','Large','Extra Large')
        command = [('Print Order', self.printOrder),('Exit',self.cancel)]
        self.addEntry('name','Customer Name', width=40)
        self.addRadio('type','Order Type', orderType)
        self.addLine('line')
        self.addCombo('size', 'Size', sizes)
        self.addRadio('crust', 'Crust', crusts, usetk=True,␣
↪indicatoron=False,
             width=12, orient='vertical')
        self.addList('toppings', 'Toppings', toppings, selectmode='multiple')
        self.addCheck('extras', 'Extra toppings', extras, orient='vertical')
        self.addButton('addpizza', '', [('Add to Order', self.addOrder)],
            width=15)
        self.addText('summary', 'Order Summary', width=100, height=20)
        self.addButton('command','', command, width=15)
        self.plotxy('name', 0, 0, pady=5)
        self.plotxy('type', 1, 0, pady=5)
        self.plotxy('line', 0, 1, columnspan=2, pady=10, sticky='we')
        self.plotxy('size', 0, 2, pady=5)
        self.plotxy('crust', 1, 2, pady=5)
        self.plotxy('toppings', 0, 3, pady=5)
        self.plotxy('extras', 1, 3, pady=5)
        self.plotxy('addpizza', 0, 4, columnspan=2, pady=10)
        self.plotxy('summary', 0, 5, columnspan=2, pady=5)
        self.plotxy('command', 0, 6, columnspan=2, pady=10)
        self.set('size', 'Medium')

    def addOrder(self):
        """ Collect the widgets and add a pizza to the order """
        order = self.get('size') + ' : ' + self.get('crust')+'\n'
        toppings = ', '.join(self.get('toppings'))
        order += '    ' + toppings+'\n'
        extras = ', '.join(self.get('extras'))
        order += '    ' + extras + '\n'
        self.set('summary', order)
        self.clearPizza()

    def printOrder(self):
        """ Print the order to the console """
        summary = self.get('name') + ' : ' + self.get('type') + '\n'
        order = self.get('summary')
        self.popMessage(order, 'showinfo', 'Order')
        self.clearPizza()
        self.set('name','')
        self.reset('type')
        self.set('summary', '', allValues=True)

    def clearPizza(self):
        """ Clear a pizza """
```

(continues on next page)

(continued from previous page)

```
66          self.set('size', 'Medium')
67          self.reset('crust')
68          self.reset('toppings')
69          self.reset('extras')
70
71  def main():
72      """ The driving function """
73      app = PizzaGui()
74      app.makeGui()
75      app.waitforUser()
76
77  if __name__ == '__main__':
78      main()
79
80
```

Here are the line explanations:

1. Import Window from tkintertoy.
2. Blank line.
3. Create a class `PizzaGui` that inherits from `Window`. You can think of `PizzaGui` as a child of `Window`.
4. Class documentation.
5. Blank line.
6. Create an instance of `PizzaGui`.
7. Method documentation.
8. Initial an instance of `Window` and assign it to `self`. This is how to call the initialzation code of the parent class. This will make the instance of `PizzaGui` an instance of `Window`.
9. Blank line.
10. This method will contain all the code to create the GUI.
11. Method documetation.
12. Set the title of the window.
13. Create a toppings tuple. This could have been a list as well.
14. Same.
15. Create a crust-type tuple.
16. Create an order-type tuple.
17. Create a extra tuple.
18. Create a size tuple.
19. Create a command list for the command buttons.
20. Add an entry for the customer name.
21. Add a radiobox for the order type.
22. Add a **ttLine**. This is a horizontal ttk.Separtor which will strectch across the entire window. It has no frame.
23. Add a **ttCombobox** for the size selection.

24. Add a **ttRadiobox** for the crust type. The orientation will be vertical. We want the entire box to light up when selected so we are setting the *indicatoron=False*, which is a tk feature, so *usetk=True*.
25. Same.
26. Add the ttListbox for toppings. We also want this to be vertical and we want to be able to select multiple toppings without pressing the Control or Shift keys. This shows how a listbox can be used instead of a checkbox.
27. Add the ttCheckbox for extra cheese and/or sauce.
28. Add a single command button, 'addpizza', that adds the pizza to the order.
29. Same.
30. Add a ttText widget to show the order.
31. Add the two command buttons defined in line 19.
32. Plot the 'name' entry at column 0, row 0, with a five pixel spacing.
33. Plot the order 'type' radiobox at column 1, row 0, with a five pixel spacing.
34. Plot the line at column 0, row 1 strectched across all of the row with a 10 pixel spacing. If we did not use the sticky='we' option, the line would be a single point!
35. Plot the 'size' combobox at column 0, row 2, with a 5 pixel spacing.
36. Plot the 'crust' radiobox at column 1, row 2, with a 5 pixel spacing.
37. Plot the 'toppings' listbox at column 0, row 3, with a 5 pixel spacing.
38. Plot the 'extras' radiobox at column 1, row 3, with a 5 pixel spacing.
39. Plot the 'addpizza' button at column 0, row 4, spread across both columns, with a 10 pixel spacing.
40. Plot the 'summary' text widget at column 0, row 5, spread across both columns, with a 5 pixel spacing.
41. Plot the 'command' buttons at column 0, row 6, spread across both columns, with a 10 pixel spacing.
42. Set the 'size' combobox to 'Medium'.
43. Blank line.
44. This method adds a pizza to the order.
45. Method documentation
46. Get the 'size' and the 'crust' selections and create an order str.
47. Collect all the 'toppings' selection create a new str.
48. Add the 'toppings' str to the order str.
49. Collect the 'extras' selection and create a new str.
50. Add the 'extras' selection to the order str.
51. Add the 'order' str to the 'order' text widget.
52. Call the `clearPizza` method.
53. Blank line.
54. This method would send an order to another display or computer. Here we are just printing the order to the console.
55. Method documentation.
56. Create a summary str with the customer 'name' and the order 'type'.

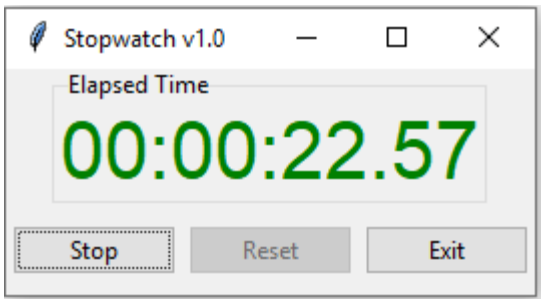
57. Get the contents of the 'summary' text widget.
58. Show the summary in a pop-up window. Normally this would go to a different display or computer.
59. Call the clearPizza method.
60. Clear the 'name' entry.
61. Clear the selections in the order 'type' radiobox.
62. Clear the 'summary' text widget.
63. Blank line.
64. This method will clear a pizza off of the widgets.
65. Method documentation
66. Set the 'size' combobox to 'Medium'
67. Clear the selection in the 'crust' radiobox.
68. Clear the selections in the 'toppings' listbox.
69. Clear the selections in the 'extras' checkbox.
70. Blank line.
71. The main function.
72. Function documentation.
73. Create an instance of *PizzaGui*.
74. Create the GUI.
75. Start the event loop.
76. Blank line.
77. Run `main` if not importing.

In this example, we see that the choice of which widget to use and how they appear is completely up to the programmer. Novice programmers are encouraged to try out different options to see which widgets meet their needs.

1.11 Dynamically Changing Widgets

The next example is a simple implementation of a digital stopwatch that demonstrates how to change a widget dynamically. **Tkintertoy** uses both tk and ttk widgets. The appearance of ttk widgets are changed using the concept of **ttStyles** which will be shown. In addition, this example will show how to change a widget state from enabled to disabled. This example will also show how to separate the implementation and the gui code into two separate classes. Lastly, this code will demonstrate how a complete application based on Tkintertoy could be written. We will stay with inheritance style coding.

Below is a screenshot:



Here is the code:

```
# stopwatch.py - A single stopwatch - Mike Callahan - 1/7/2020

from time import time
from tkintertoy import Window

def sec2hmsc(secs):
    """ convert seconds to (hours, minutes, seconds, cseconds) """
    hours, rem = divmod(secs, 3600)                    # extract hours
    minutes, rem = divmod(rem, 60)                     # extract minutes
    seconds, cseconds = divmod(rem*100, 100)           # extract seconds,
↪cseconds
    return (int(hours), int(minutes), int(seconds), int(cseconds))

class Stopwatch:
    """ Encapsulate a simple stopwatch """

    def __init__(self):
        """ initialize the stopwatch """
        self.reset()                                     # clear everything

    def start(self):
        """ start the stopwatch """
        self.then = time()                             # record starting time
        if self.elapsed > 0:
            self.then -= self.elapsed
        self.running = True                                # raise flag

    def check(self):
        """ check the elapsed time """
        if self.running:
            now = time()                               # get current time
            self.elapsed = now - self.then                 # update elapsed
        elptup = sec2hmsc(self.elapsed)
        return elptup

    def stop(self):
        """ stop the stopwatch """
        self.check()                                       # update elapsed
        self.running = False                               # lower flag

    def reset(self):
        """ reset the stopwatch """
        self.then = 0.0  # starting time
        self.elapsed = 0.0  # elapsed time during stop
        self.running = False  # running flag
```

(continues on next page)

(continued from previous page)

```
45
46  class Gui(Window):
47      """ Gui for stopwatch """
48
49      def __init__(self, stopwatch):
50          """ init stopwatch gui """
51          super().__init__()                          # create a window
52          self.stopw = stopwatch                      # make stopwatch an␣
    ↪attribute
53
54      def makeGui(self):
55          """ create the Gui """
56          self.setTitle('Stopwatch v1.0')
57          self.addStyle('r.TLabel', foreground='red',  # create the styles
58              font=('Helvetica', '30'))
59          self.addStyle('g.TLabel', foreground='green',
60              font=('Helvetica', '30'))
61          self.addLabel('elapsed', 'Elapsed Time', style='r.TLabel')
62          buttons = [('Start', self.startstop), ('Reset', self.reset),
63              ('Exit', self.cancel)]                  # label and assign buttons
64          self.addButton('buttons', cmd=buttons)      # create buttons
65          self.plotxy('elapsed', 0, 0)
66          self.plotxy('buttons', 0, 1, pady=10)
67          self.update()                               # update display
68
69      def startstop(self):
70          """ start or stop the stopwatch """
71          if self.stopw.running:
72              self.stopw.stop()
73              self.setWidget('buttons', 0, text='Start')  # relabel button
74              self.setWidget('elapsed', style='r.TLabel')  # color display
75              self.setState('buttons', ['!disabled'], 1)  # enable Reset
76          else:
77              self.stopw.start()
78              self.setWidget('buttons', 0, text='Stop')  # relabel button
79              self.setWidget('elapsed', style='g.TLabel')  # color display
80              self.setState('buttons', ['disabled'], 1)  # disable Reset
81
82      def reset(self):
83          """ reset stopwatch """
84          self.stopw.reset()                          # reset it
85
86      def update(self):
87          """ update display """
88          etime = self.stopw.check()                  # get elapsed time
89          template = '{:02}:{:02}:{:02}.{:02}'        # 2 digits leading␣
    ↪zero
90          stime = template.format(*etime)             # format as hh:mm:ss.
    ↪cc
91          self.set('elapsed', stime)                  # update display
92          self.master.after(10, self.update)          # call again after .01 sec
93
94  def main():
95      """ the main function """
96      stopw = Stopwatch()                              # create a stopwatch␣
    ↪instance
97      gui = Gui(stopw)                                 # create a window
```

(continues on next page)

(continued from previous page)

```
98      gui.makeGui()                                   # run the gui
99      gui.waitforUser()
100
101 if __name__ == '__main__':
102     main()
103
```

Here are the line explanations:

1. File documentation. While this is a first example, all files should have a some documentation on first lines.
2. Blank line.
3. We will need the time function from the time module.
4. Import `Window` from tkintertoy.
5. Blank line.
6. Define a function, `sec2hmsc` which will change floating seconds into (hours, minutes, seconds, centiseconds). Notice how type hints work. While the Python interpeter will take no action, other tools might find a use for them.
7. Function documentation string.
8. Split decimal seconds into whole hours with a remainder. This is an example of tuple unpacking.
9. Split the remainder into whole minutes with a remainder.
10. Split the remainder into whole seconds and centiseconds.
11. Return the time values as a tuple.
12. Blank line.
13. Define the `Stopwatch` class which will encapsulate a stopwatch. Since there is no suitable object to inherit from, we will use compositon.
14. Class documentation string.
15. Blank line.
16. Create the `__init__` method. This will initialize the stopwatch by calling `reset`.
17. Method documentation string.
18. Call `reset`. Since this will be the first time this method was called it will create an attributes which will hold the beginning time, the time elapsed while stopped, and the running flag.
19. Blank line.
20. Create the `start` method. This will start the stopwatch.
21. Method documentation string.
22. Get the current time and save it in the `then` attribute.
23. If the `elapsed` attribute is non-zero...
24. The stopwatch has been stopped and `then` needs to be adjusted.
25. Set the `running` attribute to True.
26. Blank line.
27. Create the `check` method. This method will return the elapsed time as a tuple.

28. Method documentation string.
29. If the stopwatch is running...
30. Get the current time.
31. Adjust `elapsed` with the current time.
32. In any case, call convert the decimal seconds to a time tuple
33. Return the time tuple.
34. Blank line.
35. Create the `stop` method. This will stop the stopwatch.
36. This is the method documentation string.
37. Update the elapsed time by calling `check`..
38. Set `running` to False.
39. Blank line.
40. Create the `reset` method. This resets the stopwatch.
41. Method documentation string.
42. Reset all the attributes to the initial state.
43. Same.
44. Same.
45. Blank line.
46. Create the `Gui` class. This class will contain the gui for the stopwatch. We will use inheritance.
47. This is the class documentation string.
48. Blank line.
49. Create the `__init__` method which will initialize the gui.
50. Mehod documentation string.
51. Create an instance of a `Window` which will be `self`.
52. Save the inputted Stopwatch as the `stopw` attribute.
53. Blank line.
54. Create the `makeGui` method which will create the gui and begin a display loop.
55. Method documentation string.
56. Set the title of the window.
57. Create a **ttStyle** which has large red characters. This is how we will color our **ttLabel** in the stopped state. We don't want the user to input anything so a label is the correct choice of widget. Notice that the style must be created for each type of widget. Since this style is for labels, the tag must end with `.TLabel`.
58. Same.
59. Create a **ttStyle** which has large green characters. The is how we will color our label in the running state.
60. Same.
61. Create a **ttlabel** which will hold the elapsed time of the stopwatch.
62. Create a list of button labels and commands, `buttons`, for the buttons. Note the commands are Gui methods.

63. Same.
64. Create a row of **ttButtons** which will be initialized using the labels and commands in `buttons`.
65. Plot the 'elapsed' at column 0, row 0.
66. Plot the 'buttons' at column 0, row 1, with a 10 pixel spacing.
67. Update the gui.
68. Blank line.
69. Create the `startstop` method. Since the user will start and stop the stopwatch using the same button, this method will have do handle both tasks.
70. This is the method documentation string.
71. If the stopwatch is running...
72. Stop it.
73. Retext the first button as 'Start'. It was 'Stop'. This is the method to use to change a widget dynamically.
74. Change the 'elapsed' color to red.
75. Enable the 'Reset' button. 'Reset' should only be used while the stopwatch is stopped. The ! means "not" so we are setting the state of the second button to "not disabled" which enables it.
76. Else, the stopwatch was stopped...
77. Start the stopwatch.
78. Retext the first button as 'Stop'. It was 'Start'.
79. Change the 'elapsed' color to green.
80. Disable the 'Reset' button.
81. Blank line.
82. Create the `reset` method, which will reset the stopwatch. Since this is connected to the 'Reset' button and this button is disabled unless the stopwatch is stopped, this method can only be executed while the stopwatch is stopped.
83. Method documentation string.
84. Reset the stopwatch.
85. Blank line.
86. Create the `update` method which shows the elapsed time in 'elapsed'.
87. Method documentation string.
88. Get the elapsed time as a time tuple, (hours, minutes, seconds, centiseconds).
89. Create a template for the `format` string method that will convert each time element as a two digit number with leading leading zero separated by colons. If the time tuple was (0, 12, 6, 13) this template convert it to '00:12:06:13'.
90. Using the template, convert the time tuple into a string.
91. Update 'elapsed' with the time string.
92. After 0.01 seconds, call `update` again. This allows the stopwatch to update its display every hundredth of a second. Every **Tkintertoy** window has a **master** attribute which has many useful methods you can call. This line interrupts the event processing loop every 0.01 second which makes sure that the stopwatch is displaying the correct elapsed time.

93. Blank line.
94. Create the `main` function.
95. Function documentation.
96. Create a stopwatch.
97. Create the gui instance.
98. Make the gui.
99. Start the event processing loop.
100. Run `main` if not importing.

1.12 Conclusion

It is hoped that with **Tkintertoy** and the included documentation, a Python instructor can quickly lead a novice Python programmer out of the boring world of command-line interfaces and join the fun world of GUI programming. To see all the widgets that **Tkintertoy** supports, run ttgallery.py. As always, looking at the code can be very instructive.

As a result of the classes I have been teaching, I have created a series of narrated slideshows on YouTube as *Programming on Purpose with Python* which features how to use **Tkintertoy** to develop complete applications. Just search for *Mike Callahan* and *programming*.

CHAPTER 2

tkintertoy module

class tt.**Window**(*master=None*, *extra=False*, ***tkparms*)

Bases: object

An easy GUI creator intended for novice Python programmers, built upon Tkinter.

This will create a Tk window with a contents dictionary. The programmer adds "ttWidgets" to the window using the add* methods where the programmer assigns a string tag to a widget. Almost all ttk and most tk widgets are included, with some useful combined widgets. Most tk/ttk widgets are placed in a frame which can act as a prompt of the ttWidget to the user. The programmer places the ttWidgets using the plot method which is a synonym for the tkinter grid geometry manager. Contents of the widget are assigned and retrieved by using the tags to the set and get methods. This greatly simplifies working with GUIs. Also, all ttWidgets are bundled into the window object so individual ttWidgets do not need to be passed to other routines, which simplifies interfaces. However, more experienced Python programmers can access the tk/ttk widget and frames directly and take advantage of the full power of Tk and ttk.

In the below methods, not all the possible keyword arguments are listed, only the most common ones were selected. The Tkinter documentation lists all for every widget. However, tk control variables should NOT be used since they might interfere on how the set and get methods work. Default values are shown in brackets [].

In some themes, certain parameters (like background) will not work in ttk widgets. For this reason, all ttk widgets have an option to use the older tk widget by setting the usetk argument to True.

Due to problems with textvariable in nested frames with ttk, the textvariable option is not used in any of the below methods.

After creating a Window object, the master attribute will either be a Tk Frame or a Toplevel window.

Here is a summary of the methods: add* - add a new ttWidget to a window get* - get the contents or an part of the ttWidget set* - change the contents or an attribute of the widget pop* - pop-up a dialog window

Parameters

- **master** (*tk.Toplevel or tk.Frame*) – The containing window
- **extra** (*bool*) – True if this is an extra window apart from the main

Keyword Arguments

- **borderwidth** (*int*) – Width of border (pixels)
- **height** (*int*) – Height of frame (pixels)
- **padding** (*int*) – Spaces between frame and widgets (pixels)
- **relief** (*str*) – ['flat'],'raised','sunken','groove', or 'ridge'
- **style** (*ttk.Style*) – Style used for ttk.Frame or ttk.LabelFrame
- **width** (*int*) – Width of frame (pixels)

Included in the installation is a copy of John Shipman's "Tkinter 8.5 reference: a GUI for Python" from New Mexico Tech, which is the best printed version known to the author. Unfortunately, Dr. Shipman has passed away and it is getting harder to find. When the code references Tkinter documentation it is referring to Dr. Shipman's work.

VERSION = '1.60'

__contains__(*tag*)
Checks if widget tag is in window.

Called using the in operator.

Returns True if 'tag' is in window

__len__()
Return number of widgets in window.

Called using the builtin len() function.

Returns Number of widgets in window

__repr__()
Display content dictionary structure, useful for debugging.

Called using the builtin repr() function.

Returns String of self.master, self.content

addButton(*tag*, *prompt=''*, *cmd=[]*, *space=3*, *orient='horizontal'*, *usetk=False*, ***tkparms*)
Create a ttButtonbox, defaults to Ok - Cancel.

This widget is where one would place most of the command buttons for a GUI, usually at the bottom of the window. Clicking on a button will execute a method usually called a callback. Two basic ones are included; Ok and Cancel. The keyword arguments will apply to EVERY button.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **cmd** (*list*) – (label:str, callback:function) for each button
- **space** (*int*) – space (pixels) between buttons
- **orient** (*str*) – ['horizontal'] or 'vertical'
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **compound** (*str*) – Display both image and text, see ttk docs
- **image** (*tk.PhotoImage*) – GIF/PNG image to display
- **style** (*ttk.Style*) – Style to use for checkboxes

- **width** (*int*) – Width of label (chars)

Returns list of ttk/tk.Buttons

addCanvas(*tag*, *prompt=''*, *scrollbars=False*, ***tkparms*)
Create a ttCanvas window.

The tk.Canvas is another extremely powerful widget that displays graphics. Again, read the Tkinter documentation to discover all the features of this widget.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **scrollbars** (*bool*) – True if scrollbars are added

Keyword Arguments

- **width** (*int*) – Width of window (pixels)
- **height** (*int*) – Height of window (pixels)
- **background** (*str*) – Background color
- **closeenough** (*float*) – Mouse threshold
- **confine** (*bool*) – Canvas cannot be scrolled ourside scrolling region
- **cursor** (*str*) – Mouse cursor
- **scrollregion** (*list of int*) – w, n, e, s bondaries of scrolling region

Returns tk.Canvas

addCheck(*tag*, *prompt=''*, *alist=[]*, *orient='horizontal'*, *usetk=False*, ***tkparms*)
Create a ttCheckbutton box.

Checkboxes are used to collect options from the user, similar to a listbox. Checkboxes might be better for short titled options because they don't take up as much screen space. The keyword arguments will apply to EVERY checkbutton. Get/set uses list of str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **alist** (*list*) – (str1, str2, ...)
- **orient** (*str*) – ['horizontal'] or 'vertical'
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **command** (*callback*) – Function to execute when boxes are toggled
- **compound** (*str*) – Display both image and text, see ttk docs
- **image** (*tk.PhotoImage*) – GIF image to display
- **style** (*ttk.Style*) – Style to use for checkboxes
- **width** (*int*) – Width of max checkbox label (chars), negative sets minimum

Returns list of ttk/tk.Checkbuttons

addChooseDir(*tag*, *prompt=''*, *width=20*, ***tkparms*)
Create a ttChoosedirbox which is a directory entry with a browse button.

This has all the widgets needed to select a directory. When the user clicks on the Browse button, a standard Choose Directory dialog box pops up. There are many tkparms that are useful for limiting choices, see the Tkinter documentation. Get/set uses str. Normally, this would be use in a dialog. For a menu command use popChooseDir. Width is a necessary option since tkparms is for the askopenfilename widget.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **width** (*int*) – Width of entry widget

Keyword Arguments

- **initialdir** (*str*) – Initial directory (space, ' ' remembers last directory)
- **title** (*str*) – Pop-up window's title

Returns list of ttk/tk.Entry and ttk/tk.Button

addCollector(*tag*, *columns*, *widgets*, *prompt=''*, ***tkparms*)
Create a ttCollectorbox which is based on a treeview that collects contents of other widgets.

This collection of widgets allows the programmer to collect the contents of other widgets into a row. The user can add or delete rows as they wish using the included buttons. Get/set uses list of str. There is no tk version.

Parameters

- **tag** (*str*) – Reference to widget
- **columns** (*list*) – (Column headers, width (pixels))
- **widgets** (*list*) – (Tags) for simple or (window, tag) for embedded widgets
- **prompt** (*str*) – Text of frame label

Keyword Arguments

- **height** (*int*) – Height of widget
- **padding** (*int*) – Spaces around values
- **style** (*ttk.Style*) – Style used for ttk.Treeview

Returns list of ttk.Treeview and two ttk.Buttons

addCombo(*tag*, *prompt=''*, *values=None*, ***tkparms*)
Create a ttCombobox.

Comboboxes combine features of Entry and Listbox into a single widget. The user can select one option out of the list or even type in their own. It is better than lists for a large number of options. Get/set uses str or list of str. There is no tk version.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **values** (*list*) – (str1, str2, ...)

Keyword Arguments

- **height** (*int*) – Maximum number of rows in dropdown [10]
- **justify** (*str*) – Justification of text ([‘left’], ‘right’, ‘center’)
- **postcommand** (*callback*) – Function to call when user clicks on downarrow
- **style** (*ttk.Style*) – Style to use for widget
- **width** (*int*) – Width of label (chars) [20]

Returns ttk.Combobox

addEntry(*tag*, *prompt=”*, *usetk=False*, ***tkparms*)
Create an ttEntry.

Entries are the widget to get string input from the user. Get/set uses str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **justify** (*str*) – Justification of text (‘left’ [def], ‘right’, ‘center’)
- **show** (*str*) – Char to display instead of actual text
- **style** (*ttk.Style*) – Style to use for widget
- **width** (*int*) – Width of label [20] (chars)

Returns ttk/tk.Entry

addFrame(*tag*, *prompt=”*, *usetk=False*, ***tkparms*)
Create a labeled or unlabeled frame container.

This allows the programmer to group widgets into a new window. The window can have either a title or a relief style, but not both.

Parameters

- **tag** (*str*) – Reference to container
- **prompt** (*str*) – Text of frame label
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **boarderwidth** (*int*) – width of border (for relief styles only)
- **height** (*int*) – Height of frame (pixels)
- **padding** (*int*) – Spaces between frame and widgets (pixels)
- **relief** (*str*) – ‘flat’,’raised’,’sunken’,’groove’, or ‘ridge’
- **style** (*int*) – Style used for ttk.Frame or ttk.LabelFrame
- **width** (*int*) – Width of frame (pixels)

Returns tt.Window

addLabel(*tag*, *prompt=''*, *effects=''*, *usetk=False*, ***tkpamrs*)
Create a ttLabel.

Labels are used to display simple messages to the user. An effects parameter is included for the simplest font types but this will override the font keyword argument. Get/set uses str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **effects** (*str*) – 'bold' and/or 'italic'
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **anchor** (*str*) – Position in widget; ['c'], 'w', 'e')
- **background** (*str*) – Background color
- **compound** (*str*) – Display both image and text, see ttk docs
- **font** (*tkfont.Font*) – Font for label
- **foreground** (*str*) – Text color
- **image** (*tk.PhotoImage*) – GIF image to display
- **justify** (*str*) – Justification of text; ['left'], 'right', 'center'
- **padding** (*list*) – Spacing (left, top, right, bottom) around widget (pixels)
- **text** (*str*) – The text inside the widget
- **width** (*int*) – Width of label (chars)
- **wraplength** (*int*) – Character position to word wrap

Returns ttk/tk.Label

addLedger(*tag*, *columns*, *prompt=''*, ***tkparms*)
Create a ttLedger which is based on a treeview that displays a simple list with column headers.

This widget allows a nice display of data in columns. It is a simplified version of the Collector widget. Due to a bug in ttk, sideways scrolling does not work correctly. If you need sideways scrolling use the Text widget. Get/set uses list of str. There is no tk version.

Parameters

- **tag** (*str*) – Reference to widget
- **columns** (*list*) – (Column headers, width (pixels))
- **prompt** (*str*) – Text of frame label

Keyword Arguments

- **height** (*int*) – Height of widget
- **padding** (*int*) – Spaces around values
- **selectmode** (*str*) – ['browse'] or 'extended'
- **(ttk** (*style*) – Style): Style used for ttk.Treeview

Returns ttk.Treeview

addLine (*tag*, ***tkparms*)
Create a horizontal or vertical ttLine across the entire frame.

Lines are useful for visually separating areas of widgets. They have no frame. There is no tk version. Be sure to use the sticky keyword when plotting or it will be a single dot.

Parameters **tag** (*str*) – Reference to widget

Keyword Arguments

- **orient** (*str*) – ['horizontal'] or 'vertical'
- **style** (*ttk.Style*) – Style to use for line

Returns ttk.Separator

addList (*tag*, *prompt=''*, *alist=[]*, ***tkparms*)
Create a ttListbox.

Lists allow the user to select a series of options in a vertical list. It is best for long titled options but does take up some screen space. Since this is a Tk widget, there is no style keyword argument. Get/set uses list of str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **alist** (*list*) – (str1, str2, ...)

Keyword Arguments

- **background** (*str*) – Background color
- **font** (*tkfont.Font*) – Font for label
- **foreground** (*str*) – Text color
- **height** (*int*) – Height of listbox (chars) [10]
- **selectmode** (*str*) – ['browse'], 'single', 'multiple', or 'extended'
- **width** (*int*) – Width of label (chars) [20]

Returns tk.listbox

addMenu (*tag*, *parent*, *items=None*, ***tkparms*)
Add a tt.Menu

Menus are complex so read the Tkinter documentation carefully.

Parameters

- **tag** (*str*) – Reference to menu
- **parent** (*ttk.Menubutton or tk.Frame*) – What menu is attached to
- **items** (*list*) – ('cascade' or 'checkbutton' or 'command' or 'radiobutton' or 'separator', coptions) (see Tkinter Documentation)

Keyword Arguments **Varies** (*dict*) – see Tkinter documentation

Returns tk.Menu

addMenuButton (*tag*, *usetk=False*, ***tkparms*)
Add a ttMenubutton

A menubutton always stays on the screen and is what the user clicks on. A menu is attached to the menubutton. Menus are complex so read the Tkinter documentation carefully.

Parameters **tag** (*str*) – Reference to menubutton

Keyword Arguments **Varies** (*dict*) – see Tkinter documentation

Returns ttk/tk.Menubutton

addMessage(*tag*, *prompt*, ***tkparms*)

Create a ttMessage which is like multiline label.

Messages are used to display multiline messages to the user. This is a tk widget so the list of options is extensive. This widget's behavior is a little strange so you might prefer the Text or Label widgets. Get/set uses str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label

Keyword Arguments

- **aspect** (*int*) – Ratio of width to height
- **background** (*str*) – Background color
- **borderwidth** (*int*) – Width of border (pixels)
- **font** (*tkfont.Font*) – Font for label
- **foreground** (*str*) – Text color
- **justify** (*str*) – Justification of text; ['left'], 'right', 'center'
- **padx** (*int*) – Horizontal spaces to place around widget (pixels)
- **pady** (*int*) – Vertical spaces to place around widget (pixels)
- **relief** (*str*) – 'flat','raised','sunken','groove', or 'ridge'
- **text** (*str*) – The text inside the widget
- **width** (*int*) – Width of message (pixels)

Returns tk.Message

addNotebook(*tag*, *tabs*, ***tkparms*)

Create a tabbed notebook container.

This allows the programmer to group similar pages into a series of new windows. The user selects the active window by clicking on the tab. Assignment allows the program to display a page tab, and return the currently selected page. There is no containing frame. Get/set uses int. There is no tk version.

Parameters

- **tag** (*str*) – Reference to container
- **tabs** (*list*) – (Tab Titles), each page must be unique

Keyword Arguments

- **height** (*int*) – Height of frame (pixels)
- **padding** (*int*) – Spaces between frame and widgets (pixels)
- **style** (*int*) – Style used for ttk.Frame or ttk.LabelFrame

- **width** (*int*) – Width of frame (pixels)

Returns list of tt.Windows

addOpen(*tag*, *prompt=''*, *width=20*, ***tkparms*)
Create a ttOpenbox which is a file entry and a browse button.

This has all the widgets needed to open a file. When the user clicks on the Browse button, a standard Open dialog box pops up. There are many tkparms that are useful for limiting choices, see the Tkinter documentation. Get/set uses str. Normally, this widget would be in a dialog. For a menu command use popOpen. Width is a necessary option since tkparms is for the askopenfilename widget. If the programmer as an icon, they can replace the 'Browse' text.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **width** (*int*) – Width of the entry widget

Keyword Arguments

- **defaultextension** (*str*) – extention added to filename (must strat with .)
- **filetypes** (*list*) – entrys in file listing ((label1, pattern1), (...))
- **initialdir** (*str*) – Initial directory (space, ' ' remembers last directory)
- **initialfile** (*str*) – Default filename
- **title** (*str*) – Pop-up window's title

Returns list of ttk/tk.Entry and ttk/tk.Button

addOption(*tag*, *prompt=''*, *alist=[]*)
Create an ttOptionmenu.

Option menus allow the user to select one fixed option, similar to Radiobutton. However, option menu returns a tk.Menu and is more difficult to manipulate. There are no keyword arguments in tk.OptionMenu. Get/set uses str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **alist** (*list*) – (str1, str2, ...)

Returns tk.OptionMenu

addPanes(*tag*, *titles*, *usetk=False*, ***tkparms*)
Create a multipaned window with user adjustable columns.

This is like a notebook but all the windows are visible and the widths are adjustable. There is no frame.

Parameters

- **tag** (*str*) – Reference to container
- **titles** (*list*) – (titles) of all embedded windows
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **height** (*int*) – Height of frame (pixels)

- **orient** (*str*) – ['horizontal'] or 'vertical'
- **padding** (*int*) – Spaces between frame and widgets (pixels)
- **style** (*int*) – Style used for ttk.Frame or ttk.LabelFrame
- **width** (*int*) – Width of frame (pixels)

Returns list of tt.Windows

addProgress(*tag*, *prompt=''*, ***tkparms*)
Create a ttProgressbar.

This indicates to the user how an action is progressing. The included method supports a determinate mode where the programmer tells the user exactly how far they have progressed. Ttk also supports a indeterminate mode where a rectangle bounces back a forth. See the Tkinter documentation. Get/set uses int. There is no tk version.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label

Keyword Arguments

- **maximum** (*int*) – Maximum value [100]
- **mode** (*str*) – ['determinate'] or 'indeterminate'
- **style** (*str*) – Style to use for ttk.Progressbar
- **length** (*int*) – Length of widget (pixels)
- **orient** (*str*) – 'horizontal' or 'vertical'

Returns ttk.Progressbar

addRadio(*tag*, *prompt=''*, *alist=[]*, *orient='horizontal'*, *usetk=False*, ***tkparms*)
Create a ttRadiobutton box.

Radiobuttons allow the user to select only one option. If they change options, the previous option is unselected. This was the way old car radios worked hence its name. They are better for short titled options. The keyword arguments will apply to EVERY radiobutton. Get/set uses str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*int*) – Text of frame label
- **alist** (*list*) – (str1, str2, ...)
- **orient** (*str*) – 'horizontal' or 'vertical'
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **command** (*callback*) – Function to execute when boxes are toggled
- **compound** (*str*) – Display both image and text, see ttk docs
- **image** (*tk.PhotoImage*) – GIF image to display
- **style** (*ttk.Style*) – Style to use for checkboxes
- **width** (*int*) – Width of max label (chars), negative sets minimun

Returns list of ttk/tk.Radiobuttons

addSaveAs (*tag*, *prompt=''*, *width=20*, ***tkparms*)
Create an ttSaveasbox which is a file entry with a browse button.

This has all the widgets needed to save a file. When the user clicks on the Browse button, a standard SaveAs dialog box pops up. If the user selects an existing file, it will pop up a overwrite confirmation box. There are many tkparms that are useful for limiting choices, see the Tkinter documentation. Get/set uses str. Normally, this widget would be in a dialog. For a menu command, use popSaveAs. Width is a necessary option since tkparms is for the asksaveasfilename widget.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **width** (*int*) – Width of the entry widget

Keyword Arguments

- **defaultextension** (*str*) – extention added to filename (must strat with .)
- **filetypes** (*list*) – entrys in file listing ((label1, pattern1), (...))
- **initialdir** (*str*) – Initial directory (space, ' ' remembers last directory)
- **initialfile** (*str*) – Default filename
- **title** (*str*) – Pop-up window's title

Returns list of ttk/tk.Entry and ttk/tk.Button

addScale (*tag*, *parms*, *prompt=''*, *width=4*, *usetk=False*, ***tkparms*)
Create a ttScale which is an integer scale with entry box.

Scale allows the user to enter an integer value using a sliding scale. The user can also type in a value directly in the entry box. Get/set uses int. The tk widget has many more options

Parameters

- **tag** (*str*) – Reference to widget
- **parms** (*list*) – Limits of scale (from, to)
- **prompt** (*str*) – Text of frame label
- **width** (*int*) – Width of entry widget (chars)
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **command** (*callback*) – Function to call when scale changes
- **length** (*int*) – Length of scale (pixels) [100]
- **orient** (*str*) – 'horizontal' or 'vertical'
- **style** (*str*) – Style to use for ttk.Scale

Returns list of ttk/tk.Scale and ttk/tk.Entry

addScrollbar (*tag*, *widgetTag*, *orient='horizontal'*, *usetk=False*, ***tkparms*)
Add a ttScrollbar to a widget.

This is usually this is done automatically. There is no frame. In order to plot the programmer must get the widget frame and use the correct sticky option. It was included for completeness.

Parameters

- **tag** (*str*) – Reference to widget
- **widgetTag** (*str*) – Tag of connected widget
- **orient** (*str*) – [‘horizontal’] or ‘vertical’

Keyword Arguments **style** (*ttk.Style*) – Style used for ttk.Scrollbar

Returns ttk/tk.Scrollbar

addSizegrip(*tag*, ***tkparms*)
Add a ttSizegrip widget to the window.

This places a sizegrip in the bottom right corner of the window. It is not needed since most platforms add this automatically. The programmer must use the configurerow and configurecolumn options when plotting widgets for this to work correctly. There is no frame. It was included for completeness. There is no tk version.

Parameters **tag** (*str*) – Reference to widget

Keyword Arguments **style** (*ttk.Style*) – Style used for ttk.Sizegrip, mainly background

Returns ttk.Sizegrip

addSpin(*tag*, *parms*, *between=' '*, *prompt=''*, *usetk=False*, ***tkparms*)
Create a ttSpinbox.

Spinboxes allow the user to enter a series of integers. It is best used for items like dates, time, etc. The keyword arguments will apply to EVERY spinbox. Since this is a Tk widget, there is no style keyword argument. Get/set uses str.

Parameters

- **tag** (*str*) – Reference to widget
- **parms** (*list*) – Parmeters for each spinbox ((width, from, to),...)
- **between** (*str*) – Label between each box
- **prompt** (*str*) – Text of frame label
- **usetk** (*bool*) – Use tk instead of ttk

Keyword Arguments

- **command** (*callback*) – Function to call when arrows are clicked
- **style** (*ttk.Style*) – Style to use for widget
- **justify** (*str*) – Text justification; [‘left’], ‘right’, ‘center’
- **wrap** (*bool*) – Arrow clicks wrap around

Returns list of ttk/tk.Spinboxes

addStyle(*tag*, ***tkparms*)
Add a ttk.Style to be used for other widgets.

This is the method for changing the appearance of ttk widgets. Styles are strictly defined strings so look at the Tkinter documentation.

Parameter: tag (str): Reference to style, must follow ttk naming

Keyword Arguments **with widget, see Tkinter documentation** (*Varies*) –

addText(*tag*, *prompt=''*, ***tkparms*)
Create a ttText window.

The tk.Text widget is an extremely powerful widget that can do many things, other than just displaying text. It is almost a mini editor. The default method allow the programmer to add and delete text. Be sure to read the Tkinter documentation to discover all the features of this widget. Since this is a Tk widget, there is no style keyword argument. Get/set uses str.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label

Keyword Arguments

- **background** (*str*) – Background color
- **font** (*tkfont.Font*) – Text font
- **foreground** (*str*) – Text color
- **wrap** (*str*) – Wordwrap method; [‘char’], ‘word’, or ‘none’
- **width** (*int*) – Width of window (chars)
- **height** (*int*) – Height of window (chars)

Returns tk.Text

breakout()
Exit the mainloop but don’t destroy the master.

This stops the event loop, but window remains displayed.

cancel()
Clear contents and exit mainloop.

This stops the event loop, removes the window, and deletes the widget structure.

catchExcept()
Catch the exception messages.

Use this in a try/except block to catch any errors:

Returns The exception message

Return type str

close()
Close the window.

This stops the event loop and removes the window. However, the window structure can still be referenced, and the window can be redisplayed.

focus(*tag*)
Switch focus to the desired widget.

This is useful to select the desired widget at the beginning so the user does not have to click.

Parameters **tag** (*str*) – Reference to widget

get(*tag*, *allValues=False*)
Get the contents of the ttwidget. With more complex widgets the programmer can choose to get all the values rather than user selected values.

Parameters

- **tag** (*str*) – Reference to widget, created in add*
- **allValues** (*bool*) – if true return all the values

Returns Contents of ttWidget

getFrame(*tag*)
Get the ttk frame if present.

Get the ttk.Frame or ttk.LabelFrame of the widget so the programmer can use more advanced methods.

Parameters **tag** (*str*) – Reference to widget

Returns ttk/tk.Frame or ttk/tk.LabelFrame

getType(*tag*)
Get the type of widget.

Get the type of widget as a string. All widgets have a type.

Parameters **tag** (*str*) – Reference to widget

Returns Type of widget as str

getWidget(*tag*)
Get the tk/ttk widget if present.

Get the underlying tk or ttk widget so the programmer can use more advanced methods.

Parameter: tag (str): - Reference to widget

Returns ttk/tk.Widget

grid(*tag=None*, ***tkparms*)
Same as plot, some instructors prefer grid which is standard tk.

Parameters **tag** (*str*) – Reference to widget

Keyword Arguments

- **row** (*int*) – the row number counting from 0
- **column** (*int*) – the column number
- **rowspan** (*int*) – the number of rows to span
- **columnspan** (*int*) – the number of columns to span
- **sticky** (*str*) – the directions to fill the cell for the widget
- **padx** (*int*) – horizontal space between widget cells (pixels)
- **pady** (*int*) – vertical space between widget cells (pixels)
- **ipadx** (*int*) – horizontal space within cell (pixels)
- **ipady** (*int*) – vertical space within cell (pixels)

mainloop()
Some instructors prefer mainloop

plot(*tag=None*, ***tkparms*)
Plot the ttWidget.

Deprecated, use plotxy. Place a frame and widget in a cell of a window using the row and column. Plot was selected as an easier name for novices than grid. Tkparms are extremely useful here and should be understood. Look at the Tkinter documentation.

Parameters **tag** (*str*) – Reference to widget

Keyword Arguments

- **row** (*int*) – the row number counting from 0
- **column** (*int*) – the column number
- **rowspan** (*int*) – the number of rows to span
- **columnspan** (*int*) – the number of columns to span
- **sticky** (*str*) – the directions to fill the cell for the widget
- **padx** (*int*) – horizontal space between widget cells (pixels)
- **pady** (*int*) – vertical space between widget cells (pixels)
- **ipadx** (*int*) – horizontal space within cell (pixels)
- **ipady** (*int*) – vertical space within cell (pixels)

plotxy (*tag=None*, *column=0*, *row=0*, ***tkparms*)
Plot the ttWidget at column x, row y.

Place a frame and widget in a cell of a window using the column (x) and row (y). Plot was selected as an easy name for novice programmers since they are plotting widgets in a xy grid. Tkparms are extremely useful here and should be understood. Look at the Tkinter documentation.

Parameters

- **tag** (*str*) – Reference to widget
- **column** (*int*) – the column number counting from 0
- **row** (*int*) – the row number

Keyword Arguments

- **rowspan** (*int*) – the number of rows to span
- **columnspan** (*int*) – the number of columns to span
- **sticky** (*str*) – the directions to fill the cell for the widget
- **padx** (*int*) – horizontal space between widget cells (pixels)
- **pady** (*int*) – vertical space between widget cells (pixels)
- **ipadx** (*int*) – horizontal space within cell (pixels)
- **ipady** (*int*) – vertical space within cell (pixels)

popDialog (*dtype='askopenfilename'*, ***tkparms*)
Popup a standard dialog.

This pops up a standard tk dialog.

Parameters **dtype** (*str*) – 'askopenfilename' or 'asksaveasfilename' or 'askdirectory' or 'askcolor'message (str): Message in box

Keyword Arguments

- **defaultextension** (*str*) – extention added to filename (must strat with .)
- **filetypes** (*list*) – entrys in file listing ((label1, pattern1), (...))
- **initialdir** (*str*) – Initial directory (space, ' ' remembers last directory)
- **initialfile** (*str*) – Default filename

- **title** (*str*) – Pop-up window's title
- **color** (*str*) – Initial color (for askcolor)

Returns str or (red, green, blue) for askcolor

popMessage(*message*, *mtype='showinfo'*, *title='Information'*, ***tkparms*)
Popup a tk message window.

Parameters

- **message** (*str*) – Message in box
- **mtype** (*str*) – 'showinfo' or 'showwarning' or 'showerror' or 'askyesno' or 'askokcancel' or 'askretrycancel'
- **title** (*str*) – Title of window

Keyword Arguments

- **default** (*str*) – 'OK' or 'Cancel' or 'Yes' or 'No' or 'Retry'
- **icon** (*str*) – 'error' or 'info' or 'question' or 'warning'

Returns 'ok' for show*, bool for ask*

refresh()
Alias for update_idletasks, better label for beginners.

This refreshes the appearance of all widgets. Usually this is called automatically after a widget contents are changed.

reset(*tag*)
Reset the selections in a widget

This clears any selections in a widget. This was created mainly for listboxes but is useful for all selection widgets.

Parameter: tag (str): - Reference to widget

set(*tag*, *value*, *allValues=False*)
Set the contents of the widget. The programmer has the option to replace all the values or just add new values.

Parameters

- **tag** (*str*) –
  - Reference to widget
- **value** (*object*) –
  - Value to set
- **allValues** (*bool*) – if True, replace all values

setState(*tag*, *states*, *index=None*)
Set or clear ttk or tk widget states

Change the underlying ttk or tk widget states. For ttk widgets the states are 'active', 'alternate', background', 'disabled', 'focus', 'invalid', 'pressed', 'readonly', and 'selected'. Preceding a state with '!' clears it. For tk widgets use 'disabled', 'normal', or 'readonly'. Index parameter allows you to change an individual element in multipart widget. See Tkinter documentation.

Parameters

- **tag** (*str*) – Reference to widget

- **states** (*list*) – States of widget, usually 'disabled' or '!disabled' for ttk
- **index** (*int*) – Index to element in multipart widget

setTitle(*prompt*)
Set the title for a window

This allows the programmer to set the title of the window. If this method is not used, the title will be Tk. This only works with top level windows.

Parameters **prompt** (*str*) – The title of the window

setWidget(*tag*, *index=None*, ***tkparms*)
Change a tk/ttk widget attribute

Change the underlying tk or ttk widget appearance using tkparms. Index parameter allows you to change an individual element in multipart widget. See Tkinter documentation.

Parameters

- **tag** (*str*) – Reference to widget
- **index** (*int*) – Index to element in multipart widget

Keyword Arguments

- **justify** (*str*) – Justification of text ('left' [def], 'right', 'center')
- **show** (*str*) – Char to display instead of actual text
- **style** (*ttk.Style*) – Style to use for widget
- **text** (*str*) – Text inside widget

waitforUser()
Alias for mainloop, better label for beginners.

This starts the event loop so the user can interact with window.

CHAPTER 3

Tkintertoy Gallery

Date Aug 07, 2023

Author **Mike Callahan**

3.1 Introduction

In order to demostrate the capabilities of *Tkintertoy*, I wrote an sampler-type application that demonstrates how to use most of the widgets in the library, *ttgallery*. This application is a simple modify and collect program where and user interacts with the widgets and sees their selections in a text widget. It also shows two independent windows, one that uses ttk widgets, the other uses older tk widgets.

3.2 A Gallery of ttWidgets

Below is the code followed by an explanation of every line:

```
#-------------------------------------------------------------------------------
# Name:         ttgallery.py
# Purpose:      Demostrate use of tkintertoy widgets
#
# Author:       mike.callahan
#
# Created:      7/5/2023
# Copyright:    (c) mike.callahan 2019 - 2023
# License:      MIT
#-------------------------------------------------------------------------------

from tkintertoy import Window

```

(continues on next page)

(continued from previous page)

```
class Gui:

    def __init__(self):
        """ Create the windows  """
        self.gui = Window()
        self.gui2 = Window(extra=True)
        self.gui.setTitle('Tkintertoy Gallery')
        self.gui2.setTitle('Tk Only Window')
        self.makeGui()

    def makeGui(self):
        """ Create the main (ttk) window """
        # a simple menu
        mymenu = self.gui.addMenu('ttmainmenu', self.gui.master) # create a main menu
        fmenul = [['command', {'label':'Open...', 'command':self.popOpen}], # create a file menu
            ['command', {'label':'Save As...', 'command':self.popSaveAs}],
            ['command', {'label':'Choose Directory...', 'command':self.popChooseDir}],
            ['command', {'label':'Exit', 'command':self.gui.cancel}]]
        mmenul = [['command', {'label':'About', 'command':self.popAbout}], # create a misc menu
            ['command', {'label':'ChooseColor', 'command':self.popColor}]]
        fmenuc = self.gui.addMenu('ttfmenu', mymenu, fmenul)                # create sub menus
        mmenuc = self.gui.addMenu('ttmmenu', mymenu, mmenul)
        mymenu.add('cascade', label='File', menu=fmenuc) # add them to the main menu
        mymenu.add('cascade', label='Misc', menu=mmenuc)
        self.gui.master['menu'] = mymenu                            # connect the main menu to the window
        # Notebook
        tabs = ['Simple','Dialog','Multi','Other']                  # label the tabs
        self.pages = self.gui.addNotebook('ttnotebook', tabs)     # create the notebook
        # Text Box
        self.gui.addText('ttext', 'Text Box', width=60, height=10) # create text area
        self.gui.plotxy('ttext', 0, 1)
        # Progress Bar
        self.gui.addProgress('ttprogress', 'Progress Bar', length=200) # create progrees bar
        self.gui.plotxy('ttprogress', 0, 2)
        # Command Buttons
        cmd = [['Collect',self.collect],['Exit', self.gui.cancel]] # create two buttons
        self.gui.addButton('ttbutton', '', cmd)
        self.gui.plotxy('ttbutton', 0, 3)
        # Notebook Pages
        self.makeSimple()
        self.makeDialog()
        self.makeMulti()
        self.makeOther()
        self.gui.plotxy('ttnotebook', 0, 0)
        self.gui.set('ttnotebook', 'Simple')                        # select first page
```

(continues on next page)

(continued from previous page)

```
        self.makeGui2()

    def makeSimple(self):
        """ Create the page with the most common widgets """
        self.simplePage = self.pages[0]
        # Label
        self.simplePage.addLabel('ttlabel', '', 'bold',          # create a label with an
            text='This is a BOLD label')                         # initial text
        self.simplePage.plotxy('ttlabel', 0, 0)
        # Line
        self.simplePage.addLine('ttline')                        # create a horizontal line
        self.simplePage.plotxy('ttline', 0, 1, sticky='we')      # stretch it horizontally
        # Entry
        self.simplePage.addStyle('g.TEntry', foreground='green') # create a green entry
        self.simplePage.addEntry('ttentry', 'Entry', style='g.TEntry')
        self.simplePage.set('ttentry', 'Green Text')             # add the text
        self.simplePage.plotxy('ttentry', 0, 3)
        # Combobox
        acombo = ['ComboOption1','ComboOption2','ComboOption3']
        self.simplePage.addCombo('ttcombo', 'Combo Box', acombo) # create combobox
        self.simplePage.plotxy('ttcombo', 0, 5)
        # Checkboxes
        achecks = ['CheckOption1','CheckOption2','CheckOption3']
        self.simplePage.addCheck('ttchecks', 'Check Box', achecks) # create 3 checkboxes
        self.simplePage.set('ttchecks','checkOption1')           # preselect first checkbox
        self.simplePage.plotxy('ttchecks', 0, 6)
        self.simplePage.setState('ttchecks', ['disabled'], index=1) # disable CheckOption2
        # Radio Buttons
        aradio = ['RadioOption1','RadioOption2','RadioOption3']
        self.simplePage.addRadio('ttradio', 'RadioButton Box', aradio) # create 3 radiobuttons
        self.simplePage.plotxy('ttradio', 0, 7)
        # Scale
        self.simplePage.addScale('ttscale', [1,10], 'Scale', width=2, length=200) # create a scale
        self.simplePage.plotxy('ttscale', 0, 8)
        # Spinners
        adate = [[2,1,12],[2,1,31],[4,2000,2099]]
        self.simplePage.addSpin('ttspin', adate, '/', 'Date Box') # create a date entry box
        self.simplePage.set('ttspin', '4/21/2023')               # set the initial date
        self.simplePage.plotxy('ttspin', 0, 9)

    def makeDialog(self):
        """ Create the dialog widget page """
        self.dialogPage = self.pages[1]
```

(continues on next page)

(continued from previous page)

```
        # Open
        self.dialogPage.addOpen('ttopen', 'Open', width=40)      # open dialog
        self.dialogPage.plotxy('ttopen', 0, 0)
        # SaveAs
        self.dialogPage.addSaveAs('ttsaveas', 'Save As', width=40) # save as dialog
        self.dialogPage.plotxy('ttsaveas', 0, 1)
        # ChooseDir
        self.dialogPage.addChooseDir('ttchoosedir', 'Choose Dir', width=40) # choose dir dialog
        self.dialogPage.plotxy('ttchoosedir', 0, 2)

    def makeMulti(self):
        """ Create the multi use widget page """
        self.multiPage = self.pages[2]
        # Listbox
        alist = ['ListOption1','ListOption2','ListOption3']
        self.multiPage.addList('ttlist', 'List', alist, height=4,
            selectmode='multiple') # create list
        self.multiPage.plotxy('ttlist', 0, 0)
        # Ledger
        cols = [['column1',100],['column2',80],['column3',80]]
        self.multiPage.addLedger('ttledger', cols, 'Ledger', height=4) # create ledger
        self.multiPage.set('ttledger', [['item0-0','item1-0','item2-0']])
        self.multiPage.set('ttledger', [['item0-1','item1-1','item2-1']])
        self.multiPage.set('ttledger', [['item0-2','item1-2','item2-2']])
        self.multiPage.plotxy('ttledger', 0, 1)
        # Collector
        self.subwin = self.multiPage.addFrame('ttframe', '', relief='groove')
        # -Combobox
        acombo = ['ComboOption2-1','ComboOption2-2','ComboOption2-3']
        self.subwin.addCombo('ttcombo2', 'Combo Box 2', acombo)
        self.subwin.plotxy('ttcombo2', 0, 0)
        # -Radio Button
        aradio = ['Radio2-1','Radio2-2','Radio2-3']
        self.subwin.addRadio('ttradio2', 'RadioButton Box 2', aradio)
        self.subwin.plotxy('ttradio2', 0, 1)
        # -Collector
        cols = [['Combo',110],['Radio', 90]]
        self.subwin.addCollector('ttcollector', cols, ['ttcombo2','ttradio2'],
            'Collector', height=4)
        self.subwin.plotxy('ttcollector', 0, 2)
        self.multiPage.plotxy('ttframe', 0, 2)

    def makeOther(self):
        """ Create page with the leftover widgets """
        self.otherPage = self.pages[3]
        # Canvas
        canvas = self.otherPage.addCanvas('ttcanvas', 'Canvas', width=300,
            height=100) # create canvas
        canvas.create_oval(10, 10, 290, 90, fill='green')
        self.otherPage.plotxy('ttcanvas', 0, 0)
        # Multipane
        paneTitles = ['Pane 1','Pane 2','Pane 3']
```

(continues on next page)

(continued from previous page)

```
        panes = self.otherPage.addPanes('ttpane', paneTitles, orient='horizontal')
        for i in range(3):
            # -Label
            tag = 'ttlabel' + str(i)
            panes[i].addLabel(tag)
            panes[i].set(tag, f'Inner label {i+1}')
            panes[i].plotxy(tag)
        self.otherPage.plotxy('ttpane', 0, 1)

    def popOpen(self):
        """ Open dialog """
        self.gui.set('ttext', self.gui.popDialog('askopenfilename',
            title='Open a File')+'\n')

    def popSaveAs(self):
        """ Save As dialog """
        self.gui.set('ttext', self.gui.popDialog('asksaveasfilename',
            title='Save a File')+'\n')

    def popChooseDir(self):
        """ Choose Directory dialog """
        self.gui.set('ttext', self.gui.popDialog('askdirectory',
            title='Select a Directory')+'\n')

    def popColor(self):
        """ Choose Color dialog """
        self.gui.set('ttext', str(self.gui.popDialog('askcolor',
            title='Select a Color'))+'\n')

    def popAbout(self):
        """ Pop Up an About window """
        self.gui.popMessage('Tkintertoy Gallery\nMost of the widgets in Tkintertoy.')

    def makeGui2(self):
        """ Fill a second independent window using tk widgets only """
        # Label
        self.gui2.addLabel('ttlabel2',usetk=True, text='These are Tk widgets.',
             effects='bold')
        # Entry
        self.gui2.addEntry('ttentry2','Type something here', usetk=True,
             foreground='blue', background='yellow')
        # Checkboxes
        achecks = ['CheckOption1','CheckOption2','CheckOption3']
        self.gui2.addCheck('ttchecks2', 'Check Box', achecks, usetk=True) # create 3 checkboxes
        self.gui2.set('ttchecks2','CheckOption3')          # preselect first checkbox
        # Radio Buttons
        aradio = ['RadioOption1','RadioOption2','RadioOption3']
        self.gui2.addRadio('ttradio3', 'RadioButton Box', aradio, usetk=True) # create 3 radiobuttons
        self.gui2.set('ttradio3', 'RadioOption2')
        # Message
        self.gui2.addMessage('ttmessage', 'Message', justify='center') # create a message
```

(continues on next page)

(continued from previous page)

```
        self.gui2.set('ttmessage', 'Useful for multi-line messages,\n'
            'like this one.')                                          # add␣
↪the text
        # Option
        alist = ['Option1','Option2','Option3']
        self.gui2.addOption('ttoption', 'Option List', alist) # create an␣
↪option list
        self.gui2.set('ttoption', 'Option1')
        # Scale
        self.gui2.addScale('ttscale2', [1,10], 'Scale', width=2, usetk=True,
            orient='horizontal', length=200)                                ␣
↪         # create a scale
        # Spinners
        adate = [[2,1,12],[2,1,31],[4,2000,2099]]
        self.gui2.addSpin('ttspin2', adate, '/', 'Date Box', usetk=True) #␣
↪create a date entry box
        self.gui2.set('ttspin2', '3/15/2001')                # set the␣
↪initial date
        # Buttons
        cmd = [['Collect',self.collect2],['Close', self.gui2.close]] #␣
↪create two buttons
        self.gui2.addButton('ttbutton2', '', cmd, usetk=True)
        # Plot widgets
        self.gui2.plotxy('ttlabel2', 0, 0, padx=30)
        self.gui2.plotxy('ttentry2', 0, 1)
        self.gui2.plotxy('ttchecks2', 0, 2)
        self.gui2.plotxy('ttradio3', 0, 3)
        self.gui2.plotxy('ttmessage', 0, 4)
        self.gui2.plotxy('ttoption', 0, 5)
        self.gui2.plotxy('ttscale2', 0, 6)
        self.gui2.plotxy('ttspin2', 0, 7)
        self.gui2.plotxy('ttbutton2', 0, 8, pady=10)

    def collect(self):
        """ Show contents of all widgets on the main (ttk) page """
        result = '\nMain Window\n  Simple Page:\n    '
        result += self.simplePage.get('ttlabel') + '\n    '
        result += self.simplePage.get('ttentry') + '\n    '
        result += self.simplePage.get('ttcombo') + '\n    '
        result += str(self.simplePage.get('ttchecks')) + '\n    '
        result += self.simplePage.get('ttradio') + '\n    '
        result += str(self.simplePage.get('ttscale')) + '\n    '
        result += self.simplePage.get('ttspin') + '\n    '
        self.gui.set('ttprogress', 33)
        self.gui.set('ttext', result)
        self.gui.master.after(1000) # wait one sec
        result = '  Dialog Page:\n    '
        result += self.dialogPage.get('ttopen') + '\n    '
        result += self.dialogPage.get('ttsaveas') + '\n    '
        result += self.dialogPage.get('ttchoosedir') + '\n    '
        self.gui.set('ttprogress', 66)
        self.gui.set('ttext', result)
        self.gui.master.after(1000) # wait one sec
        result = '  Multi Page:\n    '
        result += str(self.multiPage.get('ttlist')) + '\n    '
        result += str(self.multiPage.get('ttledger')) + '\n    '
        result += str(self.subwin.get('ttcollector', allValues=True)) + '\n␣
↪  '
```

(continues on next page)

(continued from previous page)

```
256         result += f"{self.gui.get('ttnotebook')} page selected\n"
257         result += '\n\n'
258         self.gui.set('ttprogress', 100)
259         self.gui.set('ttext', result)
260         self.gui.master.after(1000) # wait one sec
261         self.gui.set('ttprogress', 0)
262
263     def collect2(self):
264         """ Collect the infomation from the second window and place in ttext
    ↪"""
265         result = '\nSecond Window:\n    '
266         result += self.gui2.get('ttlabel2')+'\n    '
267         result += self.gui2.get('ttentry2')+'\n    '
268         result += str(self.gui2.get('ttchecks2'))+'\n    '
269         result += self.gui2.get('ttradio3')+'\n    '
270         result += self.gui2.get('ttmessage') + '\n    '
271         result += self.gui2.get('ttoption') + '\n    '
272         result += str(self.gui2.get('ttscale2'))+'\n    '
273         result += self.gui2.get('ttspin2')+'\n\n'
274         self.gui.set('ttext', result)
275
276 def main():
277     """ main driving function """
278     app = Gui()
279     try:
280         app.gui.waitforUser()
281     except:                                                    # trap all␣
    ↪Exceptions
282         errorMessage = app.gui.catchExcept()
283         app.gui.popMessage(errorMessage, 'showwarning', 'Error')
284         app.gui.cancel()
285
286 if __name__ == '__main__':
287     main()
```

Here is an explanation of what each line does:

1. Documentation of application.
2. Same.
3. Same.
4. Same.
5. Same.
6. Same.
7. Same.
8. Same
9. Same.
10. Same.
11. Blank line.
12. Import the `Window` code which is the foundation of Tkintertoy.
13. Blank line.

14. Create the `Gui` class. We will use composition style so we are not inheriting from any other class. `self` will be the application.
15. Blank line.
16. **__init__**. This method creates the windows, sets the titles, then calls the `makegui` method.
17. Method documentation.
18. Create a **Window** and assign it as an attribute, `gui`.
19. Create a second independent **Window** and assign it as an attribute, `gui2`.
20. Set the title of `gui`.
21. Set the title of `gui2`.
22. Call `makeGui` which will fill the windows with widgets.
23. Blank line.
24. **makeGui**. This method creates and places all the widgets in the main (ttk) window and then calls `makeGui2`.
25. Method documentation.
26. This is the **ttMenu** creation section. Menus are good for placing command options in a pulldown structure. These can be quite complex so this is a simple example. Read the Tkinter documentation for more information.
27. Create a **ttMenu** as the main menu, `mymenu` attached to the `master` attribute to the main window, `gui`. This shows in general how to add a *Tkintertoy* widget to a window. The first argument is a unique tag for the widget, 'ttmainmenu'. You will use this tag to work with the widget. In this application all tags start with 'tt' but tags can be any string.
28. Create a file menu list, `fmenul`, the first option is 'Open...' which is connected to the `popOpen` method...
29. The second option is 'Save AS...' which is attached to the `popSaveAs` method...
30. The third option is 'Choose Directory' which is connected to the `popChooseDir` method...
31. The fourth option is 'Exit' whic is attached to the `cancel` method of `gui`. This method is included with all *Tkintertoy* windows.
32. Create a misc menu list, `mmenul`, the first option is 'About' which is attached to the `popAbout` method...
33. The second option is 'ChooseColor' which is attached to the `popColor` method.
34. Create the file menu, `fmenuc`, attached to the main menu using `fmenul`.
35. Create the misc menu, `mmenuc`, attached to the main menu using `mmenul`.
36. Add `fmenuc` as a cascade (pulldown) under the 'File' label of `mymenu`.
37. Add `mmenuc` as a cascade under the 'Misc' label of `mymenu`.
38. Add `mymenu` to the `menu` option of the master attribute of `gui`. This will make the 'File' and 'Misc' labels appear at the top of the window.
39. This is the **ttNotebook** creation section. Notebooks are a collection of windows, called pages, stacked on top of each other accessed by a tab at the top of the window. It is a good way to save on screen space and hide groups of widgets. Notebooks are a ttk only widget which have no frame.
40. Create a list of tabs, `tabs`.
41. Create a **ttNotebook** using `tabs` with a tag of 'ttnotebook'. Store the list of pages in the `pages` attribute. Each tab will create its own page.

42. This is the **ttText** section. The text widget includes vertical scrollbars and is an extremely powerful widget with lots of uses. You can think of it as the replacement for the `print` function in command-line scripts. Text is a tk only widget. Read the Tkinter documentation for more information.
43. Add a **ttText** widget 60 characters wide by 10 characters high to `gui`. The first argument is tag, 'ttext'. The second argument is the text for the widget frame. Most *Tkintertoy* widgets have frames (menus and notebooks do not have frames) in which you can change the appearance. Frames are a great place for user prompts. The other arguments are keyword arguments which define the widget. In most cases, we do not need to save the widget in a variable, the tag does this for us.
44. Plot it at column 0, row 1. The notebook will be at 0, 0. This shows how to place a **ttWidget** in a window. The first argument is the widget tag, the second argument in the column or x position, and the third argument is the row or y position. Following this are keyword arguments that modify the placement of the widgets. Widgets will not appear until they are plotted. Note, in *Tkintertoy*, widget creation and widget placement are two different method calls. You can plot the widgets immediately after creation like this method, or you can collect all the `plotxy` calls at the end of the method as you will see in a later method.
45. This is the **ttProgressbar** creation section. Progress bars show the user what percentage of time is left elapsed during a long operation. Progress bars are a ttk only widget. We will see how to update a progress bar in the data collection method.
46. Create a **ttProgressbar** that is 200 pixels wide with a tag of 'ttprogress'.
47. Plot it at column 0, row 2.
48. This is the **ttButtonbox** creation section. Buttonboxes are groups of buttons connected to commands. These are the widgets that make actions happen when user click on them.
49. Create a button list, `cmd`, which has two labels ('Collect' and 'Exit') and the linked methods (`collect` and `cancel`).
50. Create a **ttButtonbox** using `cmd`, with a tag of 'ttbutton'.
51. Plot it at column 0, row 3.
52. This is the **ttNotebook** pages creator section. Each page has its own creation method.
53. Create the first page, 'Simple'.
54. Create the second page, 'Dialog'.
55. Create the third page, 'Multi'.
56. Create the fourth page, 'Other'.
57. Plot the notebook at column 0, row 0. Note, we filled the notebook pages before we plotted the notebook.
58. Set the displayed tab to 'Simple'.
59. Create the second window. We will fill this window with **ttWidgets** that set the keyword option *usetk=True* so you can see the difference between tk and ttk widgets. In some cases, working with ttk widgets is more complex and the visble difference may not be worth the hassle. A good example of this is the **ttEntry** widget.
60. Blank line.
61. **makeSimple**. This is the method that fills the first notebook page, 'Simple'. This page will contain the most commonly used widgets that are easy to implement.
62. Method documentation.
63. Create an attribute to store the first page window, `simplePage`.
64. This is the **ttLabel** secton. Labels are a good place to put data or images that don't change.

65. Add a **ttLabel** on the first page with bold text, with a tag of 'ttlabel'. Note, if you use the *text* keyword argument, you can specify the contents at creation, you don't have to use the `set` method. It does make the method call a bit long, however.
66. Same.
67. Plot it at column 0, row 0. Notice that the columns and rows of `simplePage` are different from `gui`.
68. This is the **ttLine** section. Lines are vertical or horizontal which seperate groups of widgets. This is a ttk only widget which has no frame.
69. Add a horizontal **ttLine** to the page, with a tag of ''ttline'.
70. Plot it at column 0, row 1, stetching across the page. If we did not use the *sticky='we'* keyword argument, it would have plotted a single point!
71. This is the **ttEntry** section. The entry widget allows the user to type in a response. You can think of it as a replacement from the `input` function in command-line scripts.
72. Add a **ttStyle** for a **ttEntry** with green text, with a tag of 'g.TEntry'. The tag must end with '.TEntry' since this is a style for an entry widget. To change he appearance of a ttk.Entry, you must use a style. With tk.Entrys this is not neccessary as you will see in the tk window. However, this style can be used for multiple entries.
73. Add a **ttEntry** using the 'g.TEntry' style, with a tag of 'ttentry'. Note, the difference between the tag of the entry and the tag for the style.
74. Set the entry contents to 'Green Text'. This string will appear as green because of the style argument.
75. Plot it at column 0, row 3
76. This is the **ttCombobox** section. Comboboxes are a combination of a entry and a list. They are good for giving the user a fixed set of options but allowing them to create their own.
77. Create a combobox option list, `acombo`.
78. Add a **ttCombobox** using `acombo`, with tag a of 'ttcombo'.
79. Plot it at column 0, row 5.
80. This is the **ttCheckbox** section. Checkboxes are a good way of letting the user select multiple independent options.
81. Create a list of checkbox options, `achecks`.
82. Add a **ttCheckbox** using `achecks`, with a tag of 'ttchecks'.
83. Set the selected option to 'CheckOption1'. Note that multiple options can be selected at a time.
84. Plot it at column 0, row 6.
85. Disable the second option ('CheckOption2') from being selected. This demonstrates how to change the state of a widget. To enable, you would set the state to ['!disabled'].
86. This is the **Radiobox** section. Radioboxes are a good way of letting the user select a single option from a group of dependent options.
87. Create a list of options, `aradio`.
88. Add a **ttRadiobox** using `aradio` with a tag of 'ttradio'. Note, only a single option can be selected at a time.
89. Plot it at column 0, row 7.
90. This is the **ttScale** section. Scales are a good widget for single integer entry if the range is small.
91. Add a horizontal **ttScale** that goes between 1 and 10, that has an entry width of 2 characters, a length of 200 pixels, with a tag of 'ttscale'.

92. Plot it at column 0, row 8.
93. This is the **ttSpinbox** section. Spinboxes are a great way to enter a group of related integers in a particular format like dates, times, ss numbers, etc.
94. Create a date list for month, date, and year, `adate`. The first option is the width, the second the minimum value, and the third the maximum value.
95. Add a **ttSpinbox** for dates that runs from 1/1/2000 to 12/31/2099, with a tag of 'ttspin'.
96. Set the date to 4/21/2023. Note, the `set` method requires a string with the separators.
97. Plot it at column 0, row 9.
98. Blank line.
99. **makeDialog**. Create the method that fills the 'Dialog' page. These widgets use the built-in tk dialog widgets.
100. Method documentation.
101. Create an attribute to store the second page window, `dialogPage`.
102. This is the **ttOpen** dialog section. This is how the user can select A file to open.
103. Add a **ttOpen** with an entry width of 40 characters with a tag of 'ttopen'.
104. Plot it on the 'Dialog' page at column 0, row 0.
105. This is the **ttSaveAs** dialog section. This is how the user can select a file to save their work. If the filename already exists, a confirming overwrite dialog pops up.
106. Add a **ttSaveAs** with an entry width of 40 characters with a tag of 'ttsaveas'.
107. Plot it at column 0, row 1.
108. This is the **ttChooseDir** dialog section. This allows the user to select a working directory.
109. Add a **ttChooseDir** with an entry width of 40 characters with a tag of 'ttchoosedir'.
110. Plot it at column 0, row 2.
111. Blank line.
112. **makeMulti**. This is the method that fills the 'Multi' page. This page will contain more complex widgets.
113. Method documentation.
114. Create an attribute to store the third page window, `multiPage`.
115. This is the **ttListbox** section. While an older tk only widget, listboxes are still very useful. They can be configured to allow a single or multiple option section.
116. Create a list of listbox options, `alist`.
117. Add a **ttlistbox** that uses `alist`, that is 4 characters high, with a tag of 'ttlist'. Listboxes default to single selection like a radiobox so we are changing this using *selectmode='multiple'*.
118. Plot it on the 'Multi' page at column 0, row 0.
119. This is the **ttLedger** section. Ledger is a new widget based on a a ttk.Treeview. It is good for displaying multicolumn data. it includes a vertical scrollbar. Horizontal scrolling in treeview does not work so if you need horizontal scrolling use a text widget.
120. Create a list of lists, `cols`, that contain the column header and width in pixels.
121. Add a **ttLedger**, using `cols`, with height of 4 characters and a tag of 'ttledger'.
122. Add a line of data to the Ledger.

123. Same.
124. Same.
125. Plot it at column 0, row 1.
126. This the **ttCollector** section. This is a new complex widget combining multiple widgets and a ledger with 2 command buttons, 'Add' and 'Delete'. In this example, we will combine a combobox and a radiobox box. It acts like a dialog inside of a dialog.
127. We are going to add a **ttFrame** with a tag of 'ttframe', and place all the widgets connected to the collection inside. It will be referenced by an attribute `subframe`.
128. This is the **ttCombobox** section for the collector.
129. Create a list of combobox options, `acombo`.
130. Add a **ttCombobox** using `acombo` with a tag of 'ttcombo2'. Note, While we reused `acombo` for a different list of options, the tag 'ttcombo2' is unique. We are doing this to eliminate any confusion in the code when we collect the widgets. However, we could have used the same tag since each window keeps its own dictionary of tags.
131. Plot it at column 0, row 0 in `subframe`.
132. This is the **ttRadiobox** section for the collector.
133. Create a list of radiobox options, `aradio`.
134. Create a **ttRadioBox** using `aradio` with a tag of 'ttradio2'.
135. Plot it at column 0, row 1.
136. This is the **ttCollector** section. This will connect the above widgets to the collector.
137. Create a list of lists, `cols`, that has the column headers and the width in pixels.
138. Create the **ttCollector** using `cols` and the list of connected widgets tags, that is 4 characters high, with a tag of 'ttcollector'. Note, the connected widgets must be created before the collector is created.
139. Same.
140. Plot the collector at column 0, row 2 of `subwin`.
141. Plot `subwin` (which has a tag 'ttframe') at column 0, row 2 of `multiPage`. Note how the arguments of `plotxy` are dependent on the current container you are working with and when plotting frames you use the tag.
142. Blank line.
143. **makeOther**. This method fills the 'Other' page. This page will contain widgets that are not in the first three pages.
144. Method documentation.
145. Create an attribute to store the fourth page window, `otherPage`.
146. The is the **ttCanvas** section. Canvas is a powerful tk widget that allows you to create drawings. It has extensive methods which are listed in the Tkinter documentaton. In this example, we are going to draw a simple green oval.
147. Add a **ttCanvas** that is 300 pixels wide and 100 pixels high, with a tag of 'ttcanvas' and save it under `canvas`. Almost all `addWidget` calls return the ttk or tk widget but most of the time, we don't need it becasue we reference the widget through the tag. In this case, we are going to store the canvas widget in a local varaible, `canvas`, since we are going to call a method of the widget. We are using a local variable since we are not going the access this widget outside this method. We could have also accessed the canvas widget using `getWidget('ttcanvas')`.

148. Same.
149. Create a green oval at position (10,10) that is 290 pixels wide and 90 pixels high by calling the `create_oval` method of `canvas`.
150. Plot this canvas at column 0, row 0 on `otherPage`.
151. This is the **ttMultipane** section. Multipanes are multiple windows placed overlapping each other that can be re-sized.
152. Create a list of pane titles, `paneTitles`.
153. Add a **ttMultipane** using `paneTitles` with a tag of 'ttpane". The default orientation is vertical so this is why we are using the *orient='horizontal'* keyword argument. Note, the method will return a list of 3 windows, which we will store in `panes`.
154. Set up a loop running from 0 to 2...
155. This is the **ttlabel** section of the multipane. We want to place a single label in each pane.
156. Create a dynamic tag that looks like 'ttlabeln', where n is 0-2.
157. Add a label with the above tag in the correct window.
158. Set the contents of the label like this: 'Inner label n' where n is 1-3.
159. Plot the label in the column 0, row 0 of the correct window.
160. Plot the multipane in column 0, row 1, of `otherPage`.
161. Blank line.
162. **popOpen**. This method pops-up an open dialog. Note,the next 4 methods all call the same method. Only the arguments are different. These are the methods that the menu options are connected to.
163. Method documentation.
164. Pop-up an open dialog. Display the user's entry 'ttext'.
165. Same.
166. Blank line.
167. **popSaveAs**. This method pops-up a save as dialog.
168. Method documentation.
169. Pop-up a save as dialog. Display the user's entry in 'ttext'.
170. Same.
171. Blank line.
172. **popChooseDir**. This method pops-up a choose directory dialog.
173. Method documentation.
174. Pop-up a choose directory dialog. Display the user's entry in 'ttext'.
175. Same.
176. Blank line.
177. **popColor**. This method pops-up a choose color dialog.
178. Method documentation.
179. Pop-up a choose color dialog. Display the user's entry in 'ttext'.
180. Same.

181. Blank line.
182. **popAbout**. This method pops-up an about window. This is where you put information about your application.
183. Method documentation.
184. Pop-up a message window. Note, you don't use a tag or store anything
185. Blank line.
186. **makeGui2**. This method fills in the second window with tk versions of **ttWidgets**. This way you can see the difference between the two type of widgets
187. Method documentation.
188. This is the **ttLabel** section.
189. Add a **ttLabel** to `gui2` with the keyword argument *usetk=True* and a tag of 'ttlabel2. This will use tk widgets instead of ttk widgets. You will see this argument repeated for every widget in `gui2`. The number of keyword arguments is greater with tk widgets since some of those options were sent to the style method in the ttk version. Read the Tkinter documentation for more information. Note, tk widgets are in the front of the documentation and not all tk widgets have ttk versions.
190. Same.
191. This is the **ttEntry** section.
192. Add a **ttEntry** to `gui2` with of 'ttentry2'. Note, you can specify the foreground and background colors as keyword arguments so styles are not required to change default colors.
193. Same.
194. This is the **ttCheckbox** section.
195. Create a list of checkbox options, `achecks`.
196. Add a group of checkboxes using `achecks` with a tag of 'ttchecks2'.
197. Preselect the third option.
198. This is the **ttRadiobox** section.
199. Create a list of radiobox options, `aradio`.
200. Add a **ttRadiobox** to `gui2` with a tag of 'ttradio3'.
201. Preselect the second option.
202. This is the **ttMessage** section. This is a tk only widget good for displaying multiple lines of text.
203. Add a **ttMessage** widget center justified with a tag of 'ttmessage'.
204. Set the message content.
205. Same.
206. This is the option list section. This is an older tk only widget, similar to a combox without the entry widget.
207. Create a list of options, `alist`.
208. Add a **ttOptionlist** using `alist` with a tag of 'ttoption'.
209. Set the selected option to 'Option1'. Note, like a radiobox, only a single option can be selected at a time.
210. This is the **ttScale** section.
211. Add a horizontal **ttScale** that goes between 1 and 10, that has an entry width of 2 characters and a length of 200 pixels and a tag of 'ttscale2'.

212. Same.
213. This is the **ttSpinbox** section.
214. Create a date list for month, date, and year, `adate`. The first value is the width in characters, the second is the minimum value, and the third is the maximum value.
215. Add a **ttSpinbox** for dates that runs from 1/1/2000 to 12/31/2099 with a tag of 'ttspin2'.
216. Set the date to 3/15/2021
217. This is the **ttButtonbox** creation section.
218. Create a button list, `cmd`, which has two labels ('Collect' and 'Close') and the linked methods (`collect2` and `close`). Unlike `cancel`, `close` will close the window but the apllication will contune to run.
219. Create a **ttButtonbox** using `cmd` with a tag of 'ttbutton2'.
220. This is the widget plotting section. In `makeGui` we plotted the widgets as soon as we created them. Here we are going the plot all the widgets at the end of the method. Some programmers like this technique because they can experiment with the placement of widgets easier.
221. Plot 'ttlabel2' at column 0, row 0.
222. Plot 'ttentry2' at column 0, row 1.
223. PLot 'ttchecks2' at column 0, row 2.
224. PLot 'ttradio3' at column 0, row 3.
225. PLot 'ttmessage' at column 0, row 4.
226. Plot 'ttoption' at column 0, row 5.
227. Plot 'ttscale2' at column 0, row 6.
228. Plot 'ttspin2' at column 0, row 7.
229. Plot 'ttbutton2' at column 0, row 8, with a 10 pixel vertical spacing.
230. Blank line.
231. **collect**. This method collects all the contents of the `gui` window. To get the contents of any widget, you call the `get` method on the window with the tag as the argument. You don't have to worry about the type of widget, `get` handles this automatically.
232. Method documentation.
233. Build a string that will contain the widget contents, `result`. The header will indication that these are widgets from `simplePage`.
234. Get the contents of 'ttlabel' and add to `result`.
235. Get the contents of 'ttentry' and add to `result`.
236. Get the contents of 'ttcombo' and add to `result`.
237. Get the contents of 'ttchecks' and add to `result`. Note, since checkboxes can have multiple values, `get` returns a list, so we must convert it to a string.
238. Get the contents of 'ttradio' and add to `result`.
239. Get the contents of 'ttscale' and add to `result`. Note, since `get` returns a int we must convert it to a string.
240. Get the contents of 'ttspin' and add to `result`.

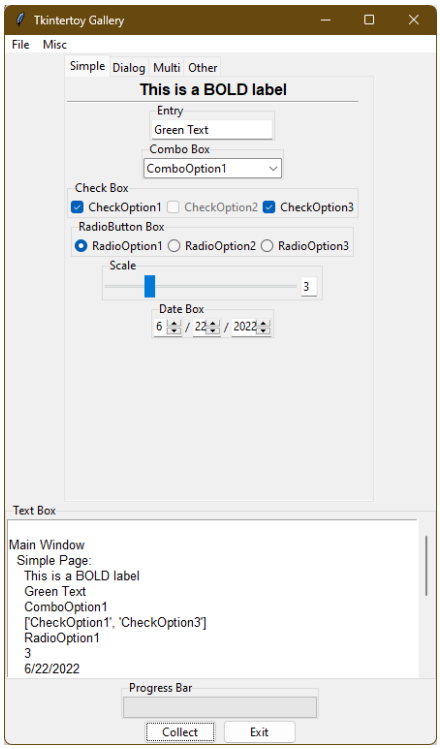
241. We have collected about a third of the widgets so lets move the ‘’ttprogress’ to the 33% position. To change the contents of any widget you use the `set` method on the window with the tag as the first argument and the value as the second argument. Again, you don’t have to worry about the type of widget, `set` handles this automatically.
242. Update ‘ttext’ with `result`.
243. Wait one second so the user can see the ‘ttprogress’ change. The `after` method of the master attribute has a number of very important uses. Read the Tkinter documentation for more information.
244. Create a new `result` for `dialogPage`.
245. Get the contents of ‘ttopen’ and add to `result`.
246. Get the contents of ‘ttsaveas’ and add to `result`.
247. Get the contents of ‘ttchoosedir’ and add to `result`.
248. We have collected about two-thirds of the widgets so lets move the ‘’ttprogress’ to the 66% position.
249. Update ‘ttext’ with `result`.
250. Wait one second so the user can see the ‘ttprogress’ change.
251. Create a new `result` for `multiPage`.
252. Get the contents of ‘ttlist’ and add to `result`. Note, since listboxes can have multiple values, `get` returns a list, so we must convert it to a string.
253. Get the contents of ‘ttledger’ and add to `result`. Note, since ledgers can have multiple values, `get` returns a list, so we must convert it to a string.
254. Get the contents of ‘ttcollector’ and add to `result`. Collector can be a single or multi value widget. We want a multi-value so the keyword argument is *allValues=True*, Note, since `get` returns a list, so we must convert it to a string.
255. Get the displayed page from ‘ttnotebook’ and add to `result`.
256. Complete `result`.
257. We have collected all of the widgets so lets move the ‘’ttprogress’ to the 100% position.
258. Update ‘ttext’ with `result`.
259. Wait one second so the user can see the ‘ttprogress’ change.
260. Result ‘ttprogess’ back to 0%.
261. Blank line.
262. **collect2**. This method collects all the contents of the `gui2` window.
263. Method documentation.
264. Build a string that will contain the widget contents, `result`. The header will indication that these are widgets from `gui2`.
265. Get the contents of ‘ttlabel2’ and add to `result`.
266. Get the contents of ‘ttentry2’ and add to `result`.
267. Get the contents of ‘ttchecks2’ and add to `result`. Note, since checkboxes can have multiple values, `get` returns a list, so we must convert it to a string.
268. Get the contents of ‘ttradio3’ and add to `result`.
269. Get the contents of ‘ttmessage’ and add to `result`.
270. Get the contents of ‘ttoption’ and add to `result`.

271. Get the contents of 'ttscale2' and add to `result`. Note, since `get` returns a int we must convert it to a string.
272. Get the contents of 'ttspin2' and add to `result`.
273. 249. Update 'ttext' with `result`.
274. Blank line.
275. **main**. Common Python. This is the main driving functon.
276. Function documentation.
277. Create an instance of Gui, `app`. Note, that this will build all the windows.
278. Begin a try block. This part of the application could crash and we want to capture any error messages.
279. Start the application loop and wait for the user to press a command button. This will continue to run until the user clicks on 'Exit'.
280. If an error occurs...
281. Catch the error message in `errorMessage`. The `catchExcept` method is included in all *Tkintertoy* windows.
282. Pop-up an message box containing `errorMessage`.
283. After the user click on 'Ok' in the message box, exit the program.
284. Blank line.
285. Standard Python. If you are not importing, excute `main`.
286. Same.

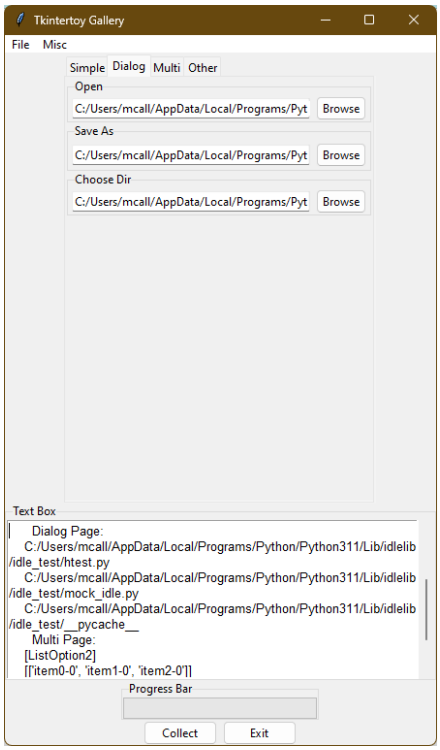
By looking at this code, the novice programmer should be able to use most of the *Tkintertoy* widgets for their own application. Be sure to also see the code examples in the tutorial for more information.

3.3 A Collection of Screenshots

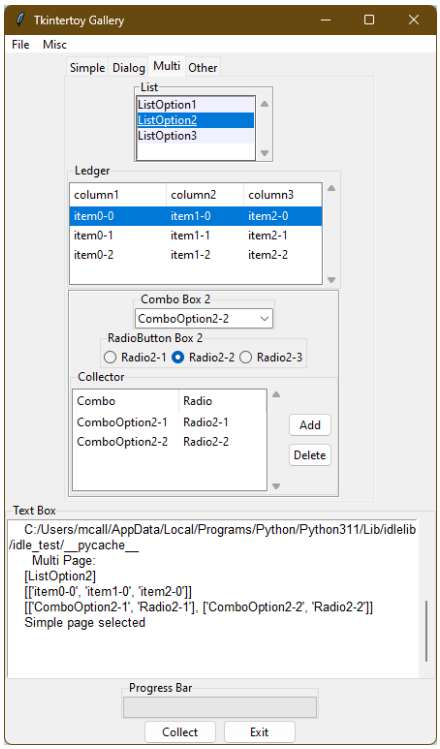
Here are screen shots of the resulting GUI, the Simple page:



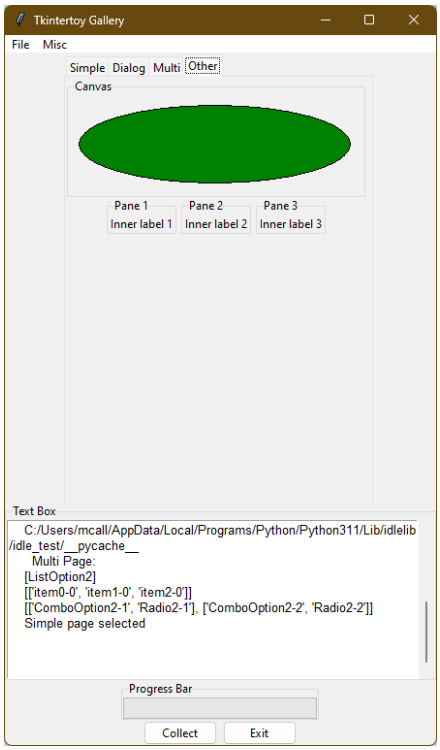
The Dialog page:



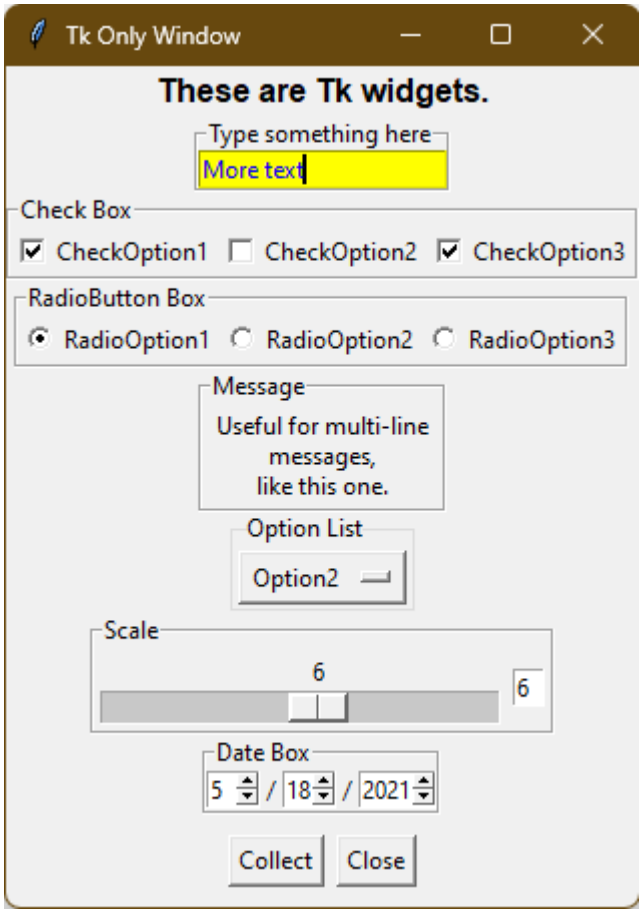
The Multi page:



The Other page:



The second (tk) window:


Tk Only Window
These are Tk widgets.
Type something here
More text
Check Box
CheckOption1
CheckOption2
CheckOption3
RadioButton Box
RadioOption1
RadioOption2
RadioOption3
Message
Useful for multi-line
messages,
like this one.
Option List
Option2
Scale
6
6
Date Box
5 / 18 / 2021
Collect
Close

CHAPTER 4

Indices and tables

- genindex
- search

Python Module Index

t

Index

Symbols

A

B

C

F

G

M

P

R

S

T

V

W