Tkintertoy Documentation

Release 1.3.0

Mike Callahan

May 13, 2020

Contents:

CHAPTER 1

Tkintertoy 1.3 Tutorial

Date May 13, 2020

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 a simple one like *Tkinter*. So I developed an easy to use GUI library that made it much simpler for their applications. This was posted on PIPY as *EzDialog*. Over the first months of 2019 I took some of the original ideas in EzDialog and developed Tkintertoy, which is even easier to use, but more powerful as well. Since that time, I have been teaching local and uploaded a series of narrated Powerpoint slide of those seminars using Tkintertoy in the development of easy applications. As result, I have fixed a few minor bugs, improved the documentation, and improved the operation of the library for version 1.3.

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.

While the early (by early I mean experience, not age) 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 of the widgets. Tkintertoy makes sure that all aspects of tk/ttk are exposed when the programmer needs them.

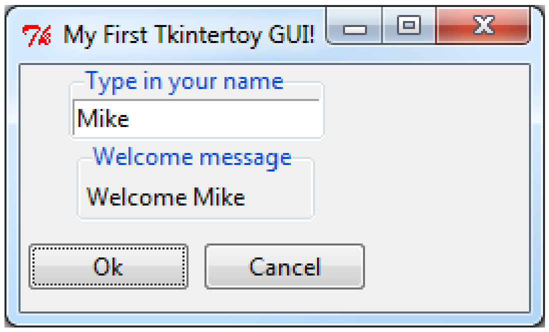
In the following example below, one can see how the ideas in Tkintertoy can be used to create simple but useful GUIs. GUI programming can be fun, which puts the “toy” in Tkintertoy.

1.2 A “Hello World” Example

Let’s look at a bare bones example of a complete GUI. This GUI will ask for the user’s name and use it in a welcome message:

```python
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.plot('name', row=0)
gui.plot('welcome', row=1)
gui.plot('commands', row=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 a 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 ‘’ or None.
4. Add an **ttentry** widget to `gui`. We are going to tag it with ‘name’ since that is what we are going to collect there. However, the tag can be any string. All Tkintertoy widgets must have a tag which acts as the key for the widget in the `content` dictionary. 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. The default width of the Entry widget is 20 characters, but this, like many other options can be overridden.
5. Add a **ttlabel** widget to `gui`. This tag will be ‘welcome’ since this is where the welcome message will appear. Labels are a good widget for one line information to appear that the user cannot edit.
6. Add a **ttbuttonbox** row. It defaults to two buttons, ‘Ok’ and ‘Cancel’. The default action is when the user clicks on ‘Ok’ the GUI processing loop is exited. However, if the user clicks on ‘Cancel’, the loop is exited and the `content` dictionary is emptied. Of course, the button labels and these actions can be easily modified by the programmer.

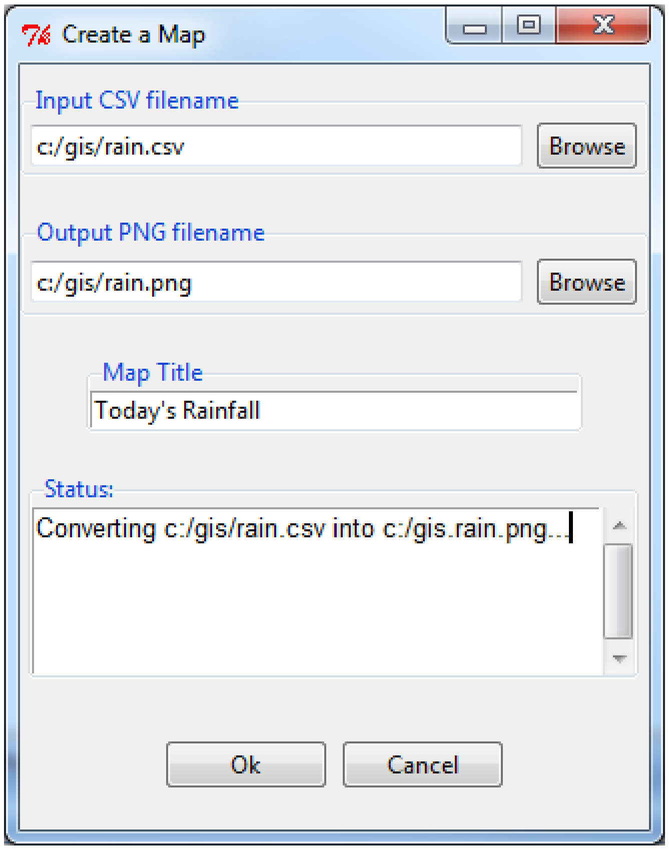
7. Place the 'name' widget at row 0 (first row) of `gui` centered. The `row=0` parameter could have been left off since it is the default. The `plot()` method is really a synonym for the tk `grid()` method. All arguments to `grid()` can be used in `plot()`. Plot was selected as a better word for a beginner. Until a widget is plotted, it will not appear. However, the `gui` window is automatically plotted.
8. Place the 'welcome' widget at row 1 (second row) of `gui` centered. There is a 3 pixel default vertical spacing between the Label widget and Entry widget.
9. Place the command bar at row 2 (third row) of `gui` centered with a vertical spacing of 10 pixels.
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 beginning programmer. 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.
12. 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. This line of code will not be reached until the user clicks on a button.
13. Since the user clicked on the 'Ok' button, collect the contents of the name widget and add it to the "Welcome" string in the welcome widget. This shows how easy it is to get and set the contents of a widget using the given methods. Also, 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.
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 program.
15. Break the infinite loop and exit the program. Notice the difference between the program loop set up by the `while` statement and the event processing loop set up by the `waitforUser()` method.

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.

1.3 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 not written in an object-oriented mode in order to help the typical GIS script or early Python script writer. Object-oriented mode 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 an *Open* filename widget, a *Save As* filename widget, and an *Entry* widget, and a *Text* widget as a status window.

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



c:/gis/rain.csv

c:/gis/rain.png

Today's Rainfall

Converting c:/gis/rain.csv into c:/gis.rain.png...

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

```python
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.plot('input', row=0, pady=10)
gui.plot('output', row=1, pady=10)
gui.plot('title', row=2, pady=10)
gui.plot('status', row=3, pady=10)
gui.plot('commands', row=4, pady=20)
gui.waitforUser()
if gui.content:
    message = 'Converting {} into {}...\n'.format(gui.get('input'), gui.get(
↪'output'))
    gui.set('status', message)
    gui.master.after(5000)
    # 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, with a 40 character wide **ttentry** widget, filtering only CSV files.
6. We want to limit our output to .png only.
7. Add a **ttsaveas** dialog widget, with a 40 character wide **ttentry** widget, filtering only 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, with a width of 40 characters, a height of 5 lines, which will be used for all status messages.
10. Add a **ttbuttonbox** with the default 'Ok' and 'Cancel' buttons.
11. Plot the input widget in the first row (row 0), vertically separating widgets by 10 pixels.
12. Plot the output widget in the second row, 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 in the third row, vertically separating widgets by 10 pixels.
14. Plot the status widget in the fourth row, vertically separating widgets by 10 pixels.
15. Plot the command widget in the fifth row, 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.
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.

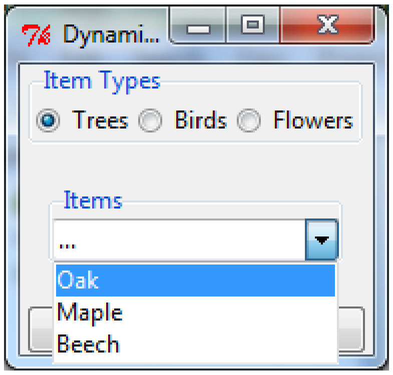
Notice, if the user clicks on the Cancel button, the program exits at step 17.

1.4 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 **ttradiobutton** row. Radiobuttons limit the user to one option out of a fixed set. Comboboxes combine a listbox with an entry widget. Thus the user can select one of several options or type in their own choice.

The trick have have the contents of a combobox be dependent on a radiobutton is to create a **ttcombobox** widget and then create a *callback* function which looks at the contents of the **ttradiobutton** row and then sets the item list attribute of the combo widget. Again, we will avoid an object-oriented approach in order not to confuse the early script writer. However, you will see later that an object-oriented approach will eliminate some strange looking 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. This complexity can be eliminated by writing in an object-oriented fashion, which will be covered in the following section.

Below is the code:

```
1  from tkintertoy import Window
2
3  def update(gui): # callback function
4      """ set the alist attribute by what is in the radio button box """
5      lookup = {'Trees':['Oak','Maple','Beech'],
6          'Birds':['Cardinal','Robin','Sparrow'],
7          'Flowers':['Rose','Petunia','Daylily']}
8      select = gui.get('category')
9      gui.set('items', lookup[select], allValues=True)
10
11 categories = ['Trees','Birds','Flowers']
12 gui = Window()
13 gui.setTitle('Dynamic Widget Demo')
14 gui.addRadio('category', 'Item Types', categories)
15 gui.addCombo('items', 'Items', None, postcommand=(lambda: update(gui)))
16 gui.addButton('command')
17 gui.set('category', 'Trees')
18 gui.set('items', '...')
19 gui.plot('category', row=0)
20 gui.plot('items', row=1, pady=20)
21 gui.plot('command', row=2)
22 gui.waitforUser()
23 if gui.content:
24     selected_cat = gui.get('category')
25     item = gui.get('items')
26     # more code would go here...
27     gui.cancel()
```

Below explains every line:

1. Import `Window` from tkintertoy.
2. Blank lines improve code readability.
3. Define the callback function. It will have a single parameter, the calling `Window`.
4. This is the function documentation string.
5. These next three lines define the lookup dictionary.
6. Same as above.

7. Same as above.
8. Get the category the user clicked on.
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, which is why the `allValues` option is used.
10. Blank lines improve code readability.
11. Create the three categories.
12. Create an instance of `Window` assigned to `gui`.
13. Set the title for `gui`.
14. Add a **ttradiobutton** box using the categories.
15. Add a **ttcombobox** widget which will update its items list whenever the user clicks on a **Radio** button. This is an example of using the `postcommand` option for the **ttcombobox** widget. 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.
16. Add a **ttbuttonbox** with the default 'Ok' and 'Cancel' buttons.
17. Initialize the category widget. This will be just as if the user clicked on Trees.
18. Initialize the items widget entry widget to just three dots. Notice the difference between this line an line 9.
19. Plot the category widget in the first row.
20. Plot the items widget in the second row.
21. Plot the command buttons in the third row.
22. 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.
23. Check to see if the user clicked on Ok by seeing if content is not empty.
24. Retrieve the value of the category widget using the get method.
25. Retrieve the value of the items widget that was selected or typed in.
26. This where the actual processing code would start.
27. Exit the program. Calling `cancel` is the same as clicking on the Cancel button.

1.5 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. 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 did not change.

Below is the new code:

```python
from tkintertoy import Window

class Gui(object):
    """ A simple gui class """

    def __init__(self):
```

(continues on next page)

(continued from previous page)

```
 7          """ create the GUI """
 8          categories = ['Trees','Birds','Flowers']
 9          self.gui = Window()
10          self.gui.setTitle('Dynamic Widget Demo')
11          self.gui.addRadio('category', 'Item Types', categories)
12          self.gui.addCombo('items', 'Items', None, postcommand=self.update)
13          self.gui.addButton('command')
14          self.gui.set('category', 'Trees')
15          self.gui.set('items', '...')
16          self.gui.plot('category', row=0)
17          self.gui.plot('items', row=1, pady=20)
18          self.gui.plot('command', row=2)
19
20      def update(self): # callback function
21          """ set the combobox values by what is in the radio button box """
22          lookup = {'Trees':['Oak','Maple','Beech'],
23              'Birds':['Cardinal','Robin','Sparrow'],
24              'Flowers':['Rose','Petunia','Daylily']}
25          select = self.gui.get('category')
26          self.gui.set('items', lookup[select], allValues=True)
27
28  app = Gui()
29  app.gui.waitforUser()
30  if app.gui.content:
31      selected_cat = app.gui.get('category')
32      item = app.gui.get('items')
33      # more code would go here...
34      app.gui.cancel()
```

And the line explanations:

1. Import `Window` from tkintertoy.
2. Blank lines improve code readability.
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 lines improve code readability.
6. Create an initialize method that will create the interface. All methods in the class will have access to `self.gui`.
7. This is the method documentation string.
8. Create the three categories.
9. Create an instance of `Window` assigned to `self.gui`. This means that all methods in the class will be able to access the `Window` through `self.gui`.
10. Set the title for `self.gui`.
11. Add a **ttradiobutton** box using the categories.
12. Add a **ttcombobox** widget which will update its items list whenever the user clicks on a **Radio** 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.
13. Add a **ttbuttonbox** with the default 'Ok' and 'Cancel' buttons.
14. Initialize the category widget.

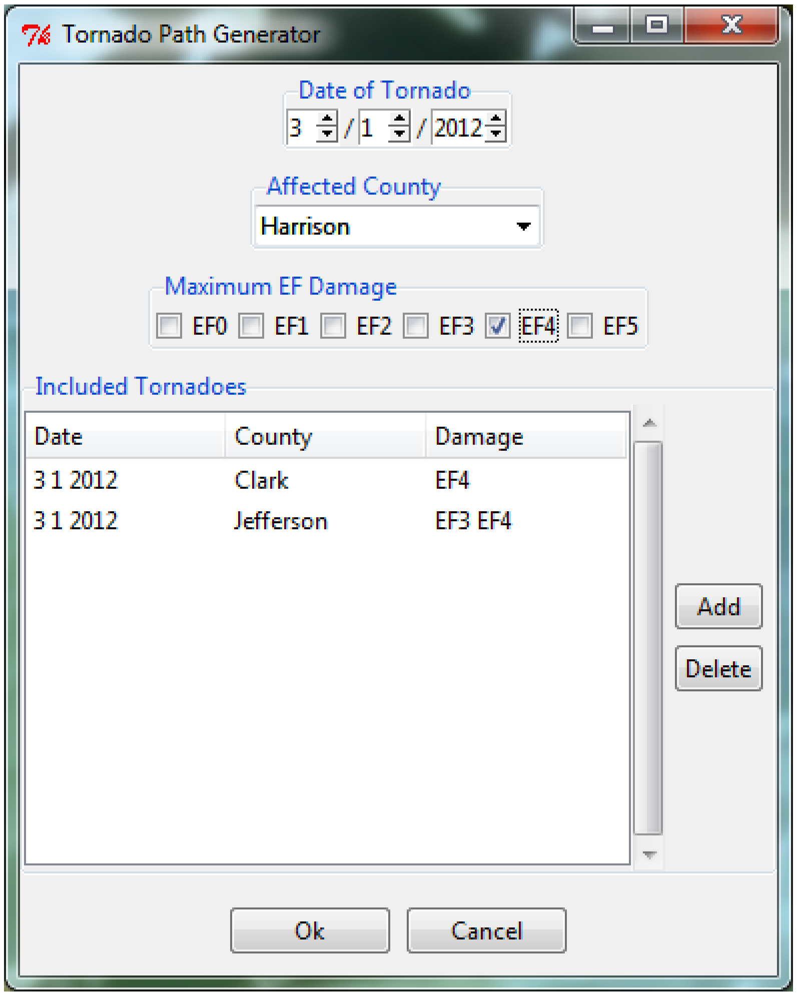
15. Initialize the items widget.
16. Plot the category widget in the first row.
17. Plot the items widget in the second row.
18. Plot the command buttons in the third row.
19. Blank lines improve code readability.
20. Create the callback method using the `self` parameter.
21. This is the method documentation string.
22. These next three lines define the lookup dictionary.
23. Same as above.
24. Same as above.
25. Get the category the user clicked on.
26. Using this category as a key, set all the items in the **ttcombobox** widget list to the list returned by the lookup dictionary, rather than the entry widget, which is why the `allValues` option is used.
27. Blank lines improve code readability.
28. 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.
29. Start the event processing loop and wait for the user to click on a button.
30. Check to see if the user clicked on Ok by seeing if content is not empty.
31. Retrieve the value of the category using the get method.
32. Retrieve the value of the entry part of the **ttcombobox**. Again, note the difference between this line and line 26.
33. Same as above.
34. This where the actual processing code would start.
35. Exit the program.

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

1.6 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 `collector` widget, which acts as a dialog inside a dialog. Below is the screenshot:



3 / 1 / 2012

Harrison

EF0 [ ] EF1 [ ] EF2 [ ] EF3 [ ] EF4 [✓] EF5 [ ]

Date	County	Damage
3 1 2012	Clark	EF4
3 1 2012	Jefferson	EF3 EF4

You can see for the date we will use a **ttspinbox**, the county will be a **ttcombobox** widget'', the damage will use **ttcheckbutton** row, and all choices will be shown in the **ttcollector** widget. Here is the code:

```python
from tkintertoy import Window

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

    def __init__(self):
        """ 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],[4,1900,2100]]
        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)
```

(continues on next page)

(continued from previous page)

```
20         self.gui.addCollector('paths', 10, cols, ['tdate','county','level'],
↪'Included Tornadoes')
21         self.gui.addButton('command')
22         self.gui.plot('tdate', row=0, pady=5)
23         self.gui.plot('county', row=1, pady=5)
24         self.gui.plot('level', row=2, pady=5)
25         self.gui.plot('paths', row=3, pady=5)
26         self.gui.plot('command', row=4, pady=10)
27 
28 def main():
29     """ the driving function """
30     app = Gui()
31     app.gui.waitforUser()
32     if app.gui.content:
33         data = app.gui.get('paths', allValues=True)
34         #magic tornado path generation code
35 
36 main()
```

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

1. Import `Window` from tkintertoy.
2. Blank lines improve code readability.
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 lines improve code readability.
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 as above.
10. Create a list of damage levels.
11. Create the parameter list for the date spinner. The first digit is the width, 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 no 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**. It will be labeled tdate in order to not cause any confusion with a common date library.
17. Set the date to the default.
18. Add a county **ttcombobox**.
19. Add a damage level **ttcheckbutton** box.
20. Add a **ttcollector**.

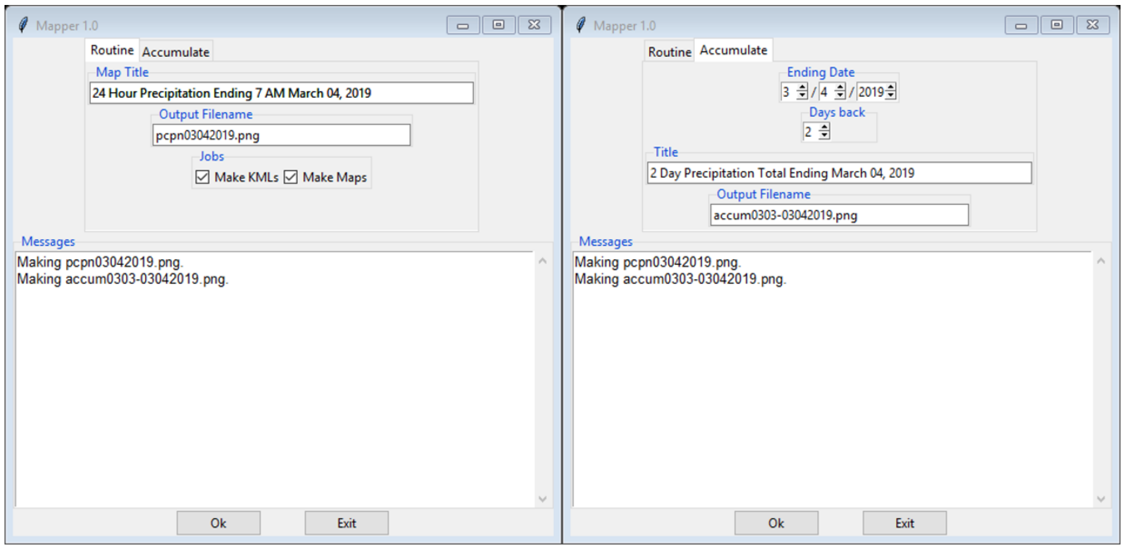
21. Add a command **ttbuttonbox**.
22. Plot the date widget in the first row, separating the widgets by 5 pixels.
23. Plot the county widget in the second row, separating the widgets by 5 pixels.
24. Plot the damage level widget in the third row, separating the widgets by 5 pixels.
25. Plot the path widget in the fourth row, separating the widgets by 5 pixels.
26. Plot the command widget in the fifth row, separating the widgets by 10 pixels.
27. Blank lines improve code readability.
28. Create a `main` function. This is the way most Python scripts work.
29. This is the function documentation.
30. Blank lines improve code readability.
31. Create an instance of the `Gui` class which will create the GUI.
32. Wait for the user to click a button.
33. Get all the lines in the collector as a list of dictionaries.
34. This is where the tornado path generation code would begin.
35. Blank lines improve code readability.
36. Run the driving function.

Note when you click on add, the current selections in tdate, counties, and level will be added into the **ttcollector** 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.

1.7 Using the Notebook Container

Tkintertoy includes containers which are `Windows` within `Windows` in order to organize widgets. A very useful one is the **ttnotebook**. This example shows a notebook that combines two different map making methods into a single GUI.

Below is a screenshot:



Map Title: 24 Hour Precipitation Ending 7 AM March 04, 2019
Output Filename: pcpn03042019.png
Jobs: ☑ Make KMLs ☑ Make Maps
Messages:
```
Making pcpn03042019.png.
Making accum0303-03042019.png.
```

Ending Date: 3 / 4 / 2019
Days back: 2
Title: 2 Day Precipitation Total Ending March 04, 2019
Output Filename: accum0303-03042019.png
Messages:
```
Making pcpn03042019.png.
Making accum0303-03042019.png.
```

Here is the code. We will also demonstrate more dynamic widgets and introduce some simple error trapping:

```python
import datetime
from tkintertoy import Window

class Gui(object):
    """ the GUI for the script """
    def __init__(self, mapper):
        """ create the interface """
        self.mapper = mapper
        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.plot('title', row=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.plot('outfile', row=1)
        jobs = ['Make KMLs', 'Make Maps']
        self.routine.addCheck('jobs', 'Jobs', jobs)
        self.routine.set('jobs', jobs[:2])
        self.routine.plot('jobs', row=2)
        # accum pcpn page
        self.accum = pages[1]
        parms = [[2, 1, 12], [2, 1, 31], [4, 2000, 2100]]
        self.accum.addSpin('endDate', parms, '/', 'Ending Date',
            command=self.updateAccum)
        self.accum.set('endDate', [today.month, today.day, today.year])
        self.accum.plot('endDate', row=0)
        self.accum.addSpin('daysBack', [[2, 1, 45]], '', 'Days back',
            command=self.updateAccum)
        self.accum.set('daysBack', [2])
        self.accum.plot('daysBack', row=1)
        self.accum.addEntry('title', 'Title', width=60)
        self.accum.plot('title', row=2)
        self.accum.addEntry('outfile', 'Output Filename', width=40)
        self.accum.plot('outfile', row=3)
        self.updateAccum()
        # dialog
        self.dialog.addText('messages', 70, 15, 'Messages')
        self.dialog.plot('messages', row=1)
        self.dialog.addButton('commands', space=20)
        self.dialog.changeWidget('commands', 0, command=self.go)
        self.dialog.changeWidget('commands', 1, text='Exit')
        self.dialog.plot('commands', row=2)
        self.dialog.plot('notebook', row=0)
        self.dialog.set('notebook', 0)

```

(continues on next page)

(continued from previous page)

```
    def updateAccum(self):
        """ update widgets on accum page """
        end = self.accum.get('endDate')
        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
        mapper = self.mapper(self)                         # create a Mapper␣
↪instance using the Gui
                                                           # instance which is␣
↪self
        try:
            if run == 0:
                mapper.runRoutine()
            elif run == 1:
                mapper.runAccum()
        except:
            self.dialog.set('messages', self.dialog.catchExcept())

class Mapper(object):
    """ 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', 'Making {}.\n'.format(filename))
        # 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', 'Making {}.\n'.format(filename))
        # magic map making code goes here

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

```

(continues on next page)

(continued from previous page)

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

Here are the line explanations:

1. Import datetime for automatic date functions
2. Import `Window` from tkintertoy.
3. Blank lines improve code readability.
4. Create a class called `Gui`. This will contain all the code dealing with the interface.
5. This is a class documentation string.
6. Create an initialize method that will create the interface. All methods in the class will have access to `self`. We are also going to pass Mapper class (not an instance) which will contain all the non-interface code. In this case it will be stubs where real code would go. We will see how this works in line 77.
7. This is the method documentation string.
8. This lets all methods in this class access the Mapper instance.
9. Create an instance of `Window` that will be asignned to an attribute `dialog`. All methods in this class will have access.
10. Set the title of the window to Mapper 1.0.
11. This code section is for the notebook widget.
12. 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.
13. Add a **ttnotebook**. The notebook will return two `Windows` which will be used as a container for each notebook page.
14. This code section is for the `Routine` notebook page.
15. Assign the first page (page[0]) of the notebook, which is a `Window` to an attribute `routine`.
16. Get today's date.
17. Convert it to [date, month, year, month abr]; ex. [25, 12, 2018, 'Dec']
18. Add a title **ttentry** widget. This will be filled in dynamically.
19. Set the title using today's date.
20. Same as above.
21. Plot the title in the first row.
22. Add an output filename **ttentry** widget. This will also filled in dynamically.
23. Set the output filename using today's date.
24. Plot the output filename widget in the second row.
25. Create a list of two types of jobs: Make KMLs & Make Maps.
26. Add a jobs **ttchecks**.
27. Turn on both check boxes, by default.
28. Plot the jobs widget in the third row.
29. This code section is for the `Accumulate` notebook page.

30. Assign the second page (page[1]) of the notebook, which is a `Window` to an attribute `accum`.
31. Create the list for the parameters of a date spinner.
32. Add an ending date **ttspin** row, with the callback set to self.updateAccum().
33. Same as above.
34. Set the ending date to today.
35. Plot the ending date widget in the first row.
36. Add a single days back **ttspin** with the callback set to self.updateAccum() as well.
37. Same as above.
38. Set the default days back to 2.
39. Plot the days back widget in the second row.
40. Add a title **ttentry**. This will be filled in dynamically.
41. Plot the title widget in the third row.
42. Add an output filename **ttentry**. This will be filled in dynamically.
43. Plot the output filename widget in the fourth row.
44. Fill in the title using the default values in the above widgets.
45. This section of code is for the rest of the dialog window.
46. Add a messages **tttext**. This is where all messages to the user will appear.
47. Plot the messages widget in the second row of the dialog window. The notebook will be in the first row.
48. Add a command **ttbutton** row, the default are labeled Ok and Cancel.
49. 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.
50. Set the label of the second button to Exit using the same method as above but changing the *text* parameter.
51. Plot the command buttons in the third row.
52. Plot the notebook in the first row.
53. Set the default notebook page to `Routine`. This will be the page displayed when the application first starts.
54. Blank lines improve readability.
55. This method will update the widgets on the accumulate page expanding on dynamic widgets.
56. This is the method documentation string.
57. Get the ending date from the widget. It will come back as [month, day, year].
58. This will turn the list of ints into a datetime object.
59. Turn the object into a comma-separated string ‘date-int, month-int, year, month-abrev’ like ‘27,12,2018,Dec’.
60. Get the number of days back the user wanted.
61. 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.
62. Same as above.
63. Calculate the beginning date from the ending date and the days back.
64. Convert the datetime into a list of strings [‘date-int’,’month-int’] like [‘25’,‘12’].

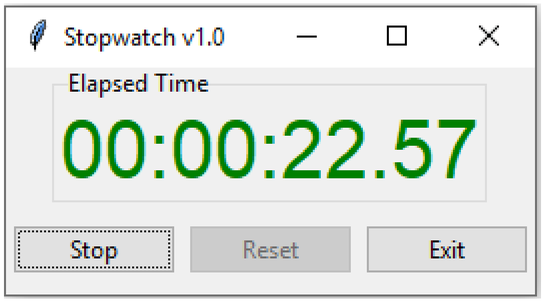
65. Set the title of the map file to something like ‘accum1225-12272018’. Again, this will be dynamically updated and can be overridden.
66. Same as above.
67. Blank lines improve code readability.
68. This method will execute the correct the map generation code.
69. This is the method documentation string.
70. Get the selected notebook tab page, either 0 for the routine page or 1 for the accumulation page.
71. 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 and why we passed the Mapper class as an argument. The Gui instance `self` is used as an argument to the Mapper initialization method. It looks funny but it works.
72. Blank lines improve code readability.
73. This code might fail so we place it in a try…except block.
74. If the current page is the routine page…
75. Run the routine map generation code.
76. If the current page is the accumulation page…
77. Run the accumulated map generation code.
78. Catch any exceptions.
79. Place all error messages into the messages widget.
80. Blank lines improve code readability.
81. 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.
82. Class documentation line.
83. Blank lines improve code readability.
84. 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!
85. Method documentation lines.
86. Same as above.
87. Make the Gui object an attribute of the instance so all methods have access.
88. Blank lines improve code readability.
89. This method contains the code for making the routine daily precipitation map.
90. Method documentation line.
91. Get the desired map title. This will be used in the magic map making code section.
92. Get the filename of the map.
93. Send a message to the user that the magic map making has begun.
94. This is well beyond the scope of this tutorial.
95. Blank lines improve code readability.

96. This method contains the code for making accumulated precipitation maps, that is, precipitation that fell over several days.
97. Method documentation line.
98. Get the desired map title. This will be used in the magic map making code section.
99. Get the filename of the map.
100. Send a message to the user that the magic map making has begun.
101. This is well beyond the scope of this tutorial.
102. Blank lines improve code readability.
103. Create the `main` function.
104. Create the GUI.
105. Run the GUI.
106. Blank lines improve code readability.
107. Standard Python. If you are executing this code from the command line, execute the main function. If importing, don't.
108. Same as above.

1.8 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.

Below is a screenshot:



Here is the code:

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

import time
from tkintertoy import Window

def sec2hmsc(secs):
    """ convert seconds to (hours, minutes, seconds, cseconds)
        secs:float -> (int, int, int, int) """
    hours, rem = divmod(secs, 3600)                   # extract hours
    minutes, rem = divmod(rem, 60)                    # extract minutes
```

(continues on next page)

(continued from previous page)

```
    seconds, cseconds = divmod(rem*100, 100)           # extract seconds,␣
↪cseconds
    return (int(hours), int(minutes), int(seconds), int(cseconds))

class Stopwatch(object):
    """ Encapsulate a simple stopwatch """

    def __init__(self):
        """ initialize the stopwatch """
        self.then = 0.0                                # starting time
        self.elapsed = 0.0                             # elapsed time during␣
↪stop
        self.running = False                           # running flag

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

    def check(self):
        """ check the elapsed time
            -> (int, int, int, int) """
        if self.running:
            now = time.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.__init__()                                # clear everything

class Gui(object):
    """ Gui for stopwatch """

    def __init__(self, stopwatch):
        """ init stopwatch gui
            stopwatch:Stopwatch -> """
        self.win = Window()                            # make window an␣
↪attribute
        self.stopw = stopwatch                         # make stopwatch an␣
↪attribute
        self.makeGui()                                 # create gui

    def makeGui(self):
        """ create the Gui """
        self.win.setTitle('Stopwatch v1.0')
        self.win.addStyle('r.TLabel', foreground='red',  # create the styles
            font=('Helvetica', '30'))
        self.win.addStyle('g.TLabel', foreground='green',
```

(continues on next page)

(continued from previous page)

```
 64             font=('Helvetica', '30'))
 65         self.win.addLabel('elapsed', 'Elapsed Time', style='r.TLabel')
 66         buttons = [('Start', self.startstop), ('Reset', self.reset),
 67             ('Exit', self.win.cancel())]                # label and assign buttons
 68         self.win.addButton('buttons', cmd=buttons)      # create buttons
 69         self.win.plot('elapsed', row=0)
 70         self.win.plot('buttons', row=1, pady=10)
 71         self.update()                                   # update display
 72 
 73     def startstop(self):
 74         """ start or stop the stopwatch """
 75         if self.stopw.running:
 76             self.stopw.stop()
 77             self.win.changeWidget('buttons', 0, text='Start')   # relabel button
 78             self.win.changeWidget('elapsed', style='r.TLabel')   # color display
 79             self.win.changeState('buttons', 1, ['!disabled'])   # enable Reset
 80         else:
 81             self.stopw.start()
 82             self.win.changeWidget('buttons', 0, text='Stop')  # relabel button
 83             self.win.changeWidget('elapsed', style='g.TLabel')   # color display
 84             self.win.changeState('buttons', 1, ['disabled'])  # disable Reset
 85 
 86     def reset(self):
 87         """ reset stopwatch """
 88         self.stopw.reset()                              # reset it
 89 
 90     def update(self):
 91         """ update display """
 92         etime = self.stopw.check()                      # get elapsed time
 93         template = '{:02}:{:02}:{:02}.{:02}'            # 2 digits leading zero
 94         stime = template.format(*etime)                 # format as hh:mm:ss.cc
 95         self.win.set('elapsed', stime)                  # update display
 96         self.win.master.after(10, self.update)          # call again after .01 sec
 97 
 98 def main():
 99     """ the main function """
100     stopw = Stopwatch()                                  # create a stopwatch instance
101     gui = Gui(stopw)                                     # create gui and go
102 
103 if __name__ == '__main__':
104     main()
105 
```

Here are the line explanations:

1. File documentation.
2. Blank lines improve code readability.

3. We will need the time function from the time module
4. Import `Window` from tkintertoy.
5. Blank lines improve code readability.
6. Define a function, `sec2hmsc` which will change decimal seconds into (hours, minutes, seconds, centiseconds).
7. Function documentation string.
8. Same as above.
9. Split decimal seconds into whole hours with a remainder.
10. Split the remainder into whole minutes with a remainder.
11. Split the remainder into whole seconds and centiseconds.
12. Return the time values as a tuple.
13. Blank lines improve code readability.
14. Define the `Stopwatch` class which will encapsulate a stopwatch.
15. This is the class documentation string.
16. Blank lines improve code readability.
17. Create the `__init__` method. This will initialize the stopwatch.
18. This is the method documentation string.
19. Create an attribute which will hold the beginning time.
20. Create an attribute which will hold the time elapsed while stopped.
21. Create an attribute which will hold the running flag.
22. Blank lines improve code readability.
23. Create the `start` method. This will start the stopwatch.
24. This is the method documentation string.
25. Get the current time and save it in the `then` attribute.
26. Check to see if the `elapsed` attribute is non-zero.
27. If so, the stopwatch has been stopped and `then` needs to be adjusted.
28. Set the `running` attribute to True.
29. Blank lines improve code readability.
30. Create the `check` method. This method will return the elapsed time as a tuple.
31. This is the method documentation string.
32. Same as above.
33. Check to see if the stopwatch is running.
34. If so, get the current time.
35. Adjust `elapsed` with the current time.
36. In any case, call convert the decimal seconds to a time tuple
37. Return the time tuple.
38. Blank lines improve code readability.

39. Create the `stop` method. This will stop the stopwatch.
40. This is the method documentation string.
41. Update the elapsed time.
42. Set `running` to False.
43. This is the method documentation string.
44. Create the `reset` method. This resets the stopwatch.
45. This is the method documentation string.
46. This method is the same as the `__init__` so just call it.
47. Blank lines improve code readability.
48. Create the `Gui` class. This class will contain the gui for the stopwatch.
49. This is the class documentation string.
50. Blank lines improve code readability.
51. Create the `__init__` method which will initialize the gui.
52. This is the method documentation string.
53. Same as above
54. Create an instance of a **Tkintertoy** window and save it as the `win` attribute.
55. Save the inputted Stopwatch as the `stopw` attribute.
56. Create the gui.
57. Blank lines improve code readability.
58. Create the `makeGui` method which will create the gui and begin a display loop.
59. This is the method documentation string.
60. Set the title of the window.
61. Create a **ttstyle** which has large red characters. This is how we will color our **ttlabel** in the stopped state. Due to operating system styles, **ttlabels** seem to be the safest widgets to experiment with styles. Certain parameters might be ignored by other widgets like **ttentry**. Notice that the style must be created for each type of widget. Since this style is for **ttlabels**, the tag must end with `.TLabel`.
62. Same as above.
63. Create a **ttstyle** which has large green characters. The is how we will color our **ttlabel** in the running state.
64. Same as above.
65. Create a **ttlabel** which will hold the elapsed time of the stopwatch.
66. Create a list of button labels and commands, `buttons`, for the buttons. Note the commands are Gui methods.
67. Same as above.
68. Create a row of **ttbuttons** which will be initialized using the labels and commands in `buttons`.
69. Plot the **ttlabel**
70. PLot the **ttbutton** row.
71. Update the gui. You will see that calling update will start an event processing loop without the use of `waitfoUser`.

72. Blank lines improve code readability.
73. 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.
74. This is the method documentation string.
75. Check to see if the stopwatch is running.
76. If so, stop it.
77. Retext the first button as Start. It was Stop.
78. Change the color to red.
79. 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.
80. Else, the stopwatch was stopped.
81. Start the stopwatch.
82. Retext the first button as Stop. It was Start.
83. Change the color to green.
84. Disable the Reset button.
85. Blank lines improve code readability.
86. 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.
87. This is the method documentation string.
88. Reset the stopwatch.
89. Blank lines improve code readability.
90. Create the `update` method which shows the elapsed time in the **ttlabel**.
91. This is the method documentation string.
92. Get the elapsed time and a time tuple, (hours, minutes, seconds, centiseconds).
93. 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'.
94. Using the template, convert the time tuple into a string.
95. Update the **ttlabel** with the time string.
96. 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 create an event processing loop but it only executes every 0.01 second which makes sure that the stopwatch is displaying the correct elapsed time.
97. Blank lines improve code readability.
98. Create the `main` function.
99. This is the function documentation.
100. Create a stopwatch.
101. Create and run the gui. Note, that assigning the gui is unnecessary.

102. Blank lines improve code readability.
103. Standard Python. If you are executing this code from the command line, execute the main function. If importing, don't.
104. Same as above.

1.9 Conclusion

It is hoped that with Tkintertoy, a Python instructor can quickly lead a young 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 early 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 including some useful combined widgets. I call these ttwidgets because they are not really complex enough to be called “megawidgets”. Most tk/ttk widgets are placed in a frame which can act as a label 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 Tk 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.

Parameters

- **master** (*tk.Toplevel*) – Toplevel 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)

__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(ttk/tk.buttons)

addCanvas(*tag*, *width*, *height*, *prompt=''*, ***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
- **width** (*int*) – Width of window (pixels)

- **height** (*int*) – Height of window (pixels)
- **prompt** (*str*) – Text of frame label

Keyword Arguments

- **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 [ttk/tk.checkbuttons]

addChooseDir (*tag*, *prompt=''*, *width=20*, ***tkparms*)
Create a ttchoosedir box 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 Tk documentation. Get/set uses str. Normally, this would be use in a dialog. For a menu command use popChooseDir.

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(ttk/tk.entry, ttk/tk.button)

`addCollector`(*tag*, *height*, *columns*, *widgets*, *prompt=''*, ***tkparms*)
Create a ttcollector 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
- **`height`** (*`int`*) – Height of 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

- **`padding`** (*`int`*) – Spaces around values
- **`style`** (*`ttk.Style`*) – Style used for ttk.Treeview

Returns [ttk.treeview, ttk.button, ttk.button]

`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 listboxes for a large number of options. Due to problems with textvariable in nested frames with ttk, they are not used. Get/set uses 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. Due to problems with textvariables in nested frames with ttk, they are not used. 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. Due to problems with textvariable in nested frames with ttk, the textvariable is not used. 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*, *height*, *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
- **height** (*int*) – Height of widget
- **columns** (*list*) – (Column headers, width (pixels))
- **prompt** (*str*) – Text of frame label

Keyword Arguments

- **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.

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.

Listboxes 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. This implementation avoids the listvariable. 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 menu

Menus are complex so read the Tk 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 Tk Documentation)

Keyword Arguments **`Varies`** (*`dict`*) – see Tk documentation

Returns tk.menu

`addMenuButton` (*tag*, *usetk=False*, ***tkparms*)

Add a menu button

A menubuuton 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 Tk documentation carefully.

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

Keyword Arguments **`Varies`** (*`dict`*) – see Tk 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. Due to problems with textvariables in nested ttk windows, they are not used. 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 selected the active window by clicking on the tab. Assignment allows the program to display a page number (counting from 0), and return the currently selected page number. There is no frame. Get/set uses int. Ther is no tk version.

Parameters

- **tag** (*str*) – Reference to container
- **tabs** (*list*) – (Tab Titles)

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(tt.windows)

addOpen (*tag*, *prompt=”*, *width=20*, ***tkparms*)

Create a ttopen box 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 Tk documentation. Get/set uses str. Normally, this widget would be in a dialog. For a menu command use popOpen.

Parameters

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

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(ttk/tk.entry, 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. 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 [tt.windows]

addProgress (*tag*, *length*, *prompt=''*, *orient='horizontal'*, ***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 Tk documentation. Get/set uses int. There is no tk version.

Parameters

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

- **length** (*int*) – Length of widget (pixels)
- **prompt** (*str*) – Text of frame label
- **orient** (*str*) – 'horizontal' or 'vertical'

Keyword Arguments

- **maximum** (*int*) – Maximum value [100]
- **mode** (*str*) – ['determinate'] or 'indeterminate'
- **style** (*str*) – Style to use for ttk.Progressbar

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 [ttk/tk.radiobuttons]

addSaveAs (*tag*, *prompt=''*, *width=20*, ***tkparms*)

Create an ttsaveas box 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 Tk documentation. Get/set uses str. Normally, this widget would be in a dialog. For a menu command, use popOpen.

Parameters

- **tag** (*str*) – Reference to widget
- **prompt** (*str*) – Text of frame label
- **width** (*int*) – Width of 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(ttk/tk.entry, ttk/tk.button)

`addScale` (*tag*, *parms*, *prompt=”*, *width=20*, *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.

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(ttk/tk.scale, ttk/tk.entry)

`addScrollbar` (*tag*, *widgetTag*, *orient=’horizontal’*, *usetk=False*, ***tkparms*)

Add a scrollbar 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 sizegrip 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 list of int.

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(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 Tk documentation.

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

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

addText (*tag*, *width*, *height*, *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 Tk 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
- **width** (*int*) – Width of window (chars)
- **height** (*int*) – Height of window (chars)
- **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'

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

changeState(*tag*, *index=None*, **states*)
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' or 'normal'. Index parameter allows you to change an individual element in multipart widget. See Tk documentation.

Parameters

- **tag** (*str*) – Reference to widget
- **index** (*int*) – Index to element in multipart widget
- **states** (*list*) – States of widget, usually ['disabled'] or ['!disabled']

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

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

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 The ttk/tk widget

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

Place a frame and widget in a cell of a window using the row and column. Plot was selected as an easier name for beginners than grid. Other tkparms are extremely useful here and should be understood. Look at the Tk 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

- **rowconfigure** (*int*) – rate widget expands in vertical if resized
- **columnconfigure** (*int*) – rate widget expands in horizontal
- **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)

popChooseDir (***tkparms*)

Popup a choose directory dialog

This pops up a standard choose directory dialog. Normally, one would use addChooseDir.

Keyword Arguments

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

Returns str

popColorChooser (***tkparms*)

Popup a color chooser dialog.

This pops up a standard color chooser dialog.

Keyword Arguments color (str): Initial color title (str): Title of pop-up window ['Color']

Returns (red, green blue)

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*

popOpen (***tkparms*)

Popup a open dialog

This pops up a standard open dialog.

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 str

`popSaveAs`(***tkparms*)

Popup a save as dialog

This pops up a standard save as dialog.

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 str

`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.

`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

`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

`ver = 1.3`

`waitforUser`()

Alias for mainloop, better label for beginners.

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

CHAPTER 3

Indices and tables

- genindex
- search

Python Module Index

Index