Formation

Emmanuel Obara

Apr 10, 2024

GETTING STARTED

CHAPTER

ONE

INSTALLATION

To use Formation studio, python 3.7 or higher is required. You can download and install python here.

Formation studio can be installed using pip:

```
pip install formation-studio
```

Note: Some linux distributions do not include pip and require you to install it separately. You can follow these instructions to do so.

If you are using multiple versions of python, pip can install Formation studio on a per version basis. For example, if you wanted to specify python 3.7:

```
pip3.7 install formation-studio
```

1.1 Installation on Linux

Formation studio uses tkinter that (depending on your distribution) may or may not be included by default. If you are using tkinter for the first time it is advised to install `tkinter` and `imagetk`.

For Debian based distributions (i.e. Ubuntu) you would use the following:

```
sudo apt-get install python3-tk, python3-pil.imagetk
```

Note: If your distribution is not Debian based you will need to subsitute the appropriate installation commands as per your distribution. Furthermore, Formation studio does **not** support `python 2`. Please ensure you install `python 3` packages only.

1.2 Launching Formation studio

Once installed you can launch Formation studio from the command line using the following command:

```
formation-studio
```

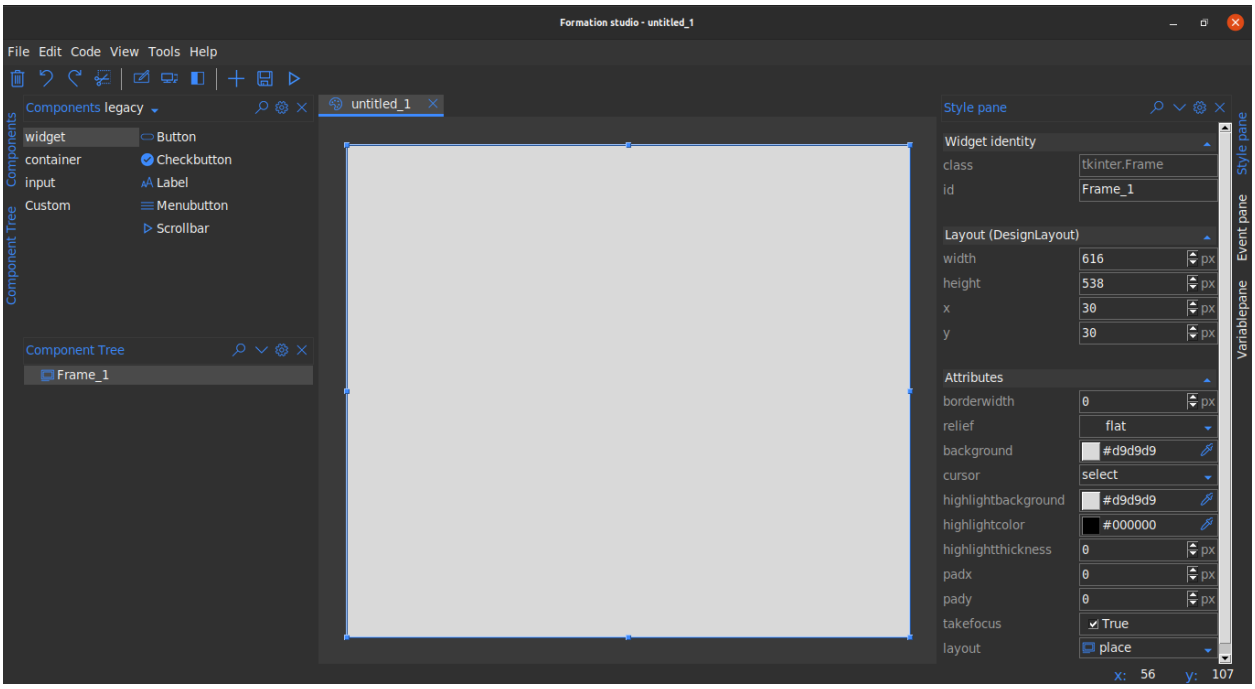
CHAPTER

TWO

QUICK START

Launch the studio from the commandline using the command

```
formation-studio
```

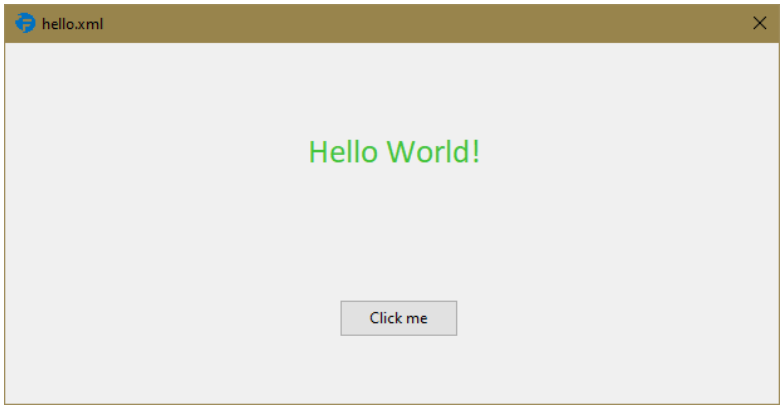


You can select widgets from the **Components** pane at the top and drag them onto the stage. Click to select widgets on the workspace and customize them on **Stylepane** to the right. You can view your widget hierarchies from the **Component tree** at the bottom left. To preview the the design, use the preview ("run button") on the toolbar. After you are satisfied with the design, save by heading to the menubar _File > Save. Below is a sample studio preview saved as `hello.xml`

The underlying xml uses namespaces and is as shown below:

```
<tkinter.Frame
    xmlns:attr="http://www.hoversetformationstudio.com/styles/"
    xmlns:layout="http://www.hoversetformationstudio.com/layouts/"
    name="Frame_1"
    attr:layout="place"
    layout:width="616"
    layout:height="287"
```

(continues on next page)



(continued from previous page)

```
    layout:x="33"
    layout:y="33">
    <tkinter.ttk.Label
        name="myLabel"
        attr:foreground="#44c33c"
        attr:font="{Calibri} 20 {}"
        attr:anchor="center" attr:text="Hello World!"
        layout:width="539"
        layout:height="89"
        layout:x="41"
        layout:y="41"/>
    <tkinter.ttk.Button
        name="myButton"
        attr:text="Click me"
        layout:width="95"
        layout:height="30"
        layout:x="266"
        layout:y="204"/>
</tkinter.Frame>
```

Note: Note: this xml file has been manually formated to make it more legible but the actual xml file is minimally formatted since it's not expected that the developer will need to modify the xml file manually

To load the design in your python code is as simple as:

```
# import the formation library which loads the design for you
from formation import AppBuilder

app = AppBuilder(path="hello.xml")

print(app.myLabel["text"]) # outputs text in the label 'Hello world!'
print(app.myButton["text"]) # outputs text in the button 'Click me'

app.mainloop()
```

Note: Note: Its advisable that you use widget names that are valid python identifiers to avoid possible issues while

use the dot syntax to access the widget from the builder object. Use the widgets exact name as specified in the design to avoid *AttributeError*

CHAPTER

THREE

COMPONENT PANE

The component pane allows you to access widgets you can put on your design. The component pane is divided into two major groups:

- Legacy (classic tkinter widgets)
- Native (ttk extension widgets)

The widgets are further divided into sub-groups to allow you to easily locate them based on their functions. These sub-groups are:

- Container (*widgets that can contain other widgets within them*)
- Widget (*widgets that have special functionality*)
- Input (*widgets that allow text and other values to be input*)

More groups may appear depending on what extensions you are using. The canvas tool for example may avail an additional `canvas` group with widgets that can be drawn on a canvas. Find out more on this in the *Canvas* section

---

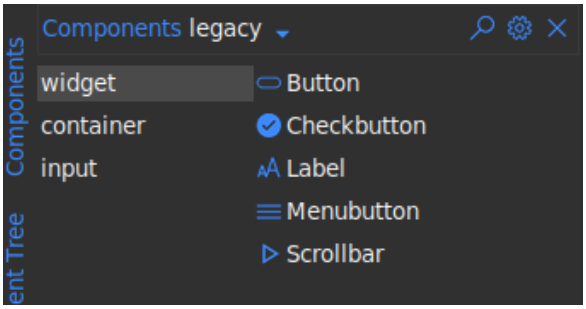
Note:

- You can switch to the group (legacy or native) you want to work with in the drop down at the top left of the component pane.
- To use a widget in your design file, just **drag** it to the design area
- You can use the search icon at the top of the pane to find a widget across all sub-groups with ease.
- You can mix widgets in Legacy and Native in the same design.

3.1 Legacy

This consists of the classic tkinter widgets. They allow more style attributes to be set. They look the same on all systems and their default look may seem outdated but this is made up for by the multitude of style options at your disposal. Some widgets can only be found in this Legacy group for instance:

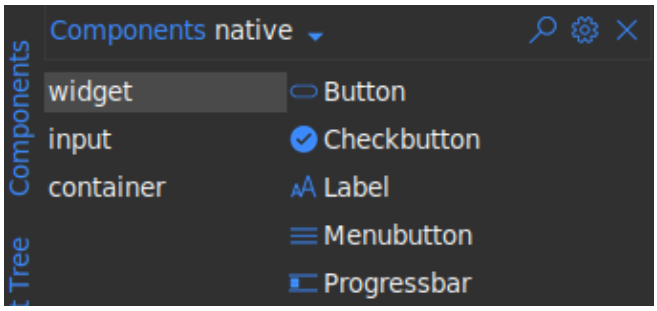
- **Listbox** (*displays a list of items*)
- **Canvas** (*allows flexible drawing of shapes, images and text*)
- **Text** (*text area allowing multiline text input*)
- **Message** (*label for longer text*)



3.2 Native

This consists of ttk extension widgets. These types of widgets are designed to be themed and hence don't allow you to modify several style options that were available in `legacy`. These widget will look different on different platforms since they try to look as native as possible to the respective platforms. Some widgets can only be found in this native group for instance:

- **Treeview** (*displays a items in tabular hierarchical structure*)
- **Sizegrip** (*a resizable frame*)
- **Combobox** (*Entry widget allowing selection of values from a list*)
- **Progressbar** (*display progress of a task*)
- **LabeledScale** (*A scale with a built-in label*)

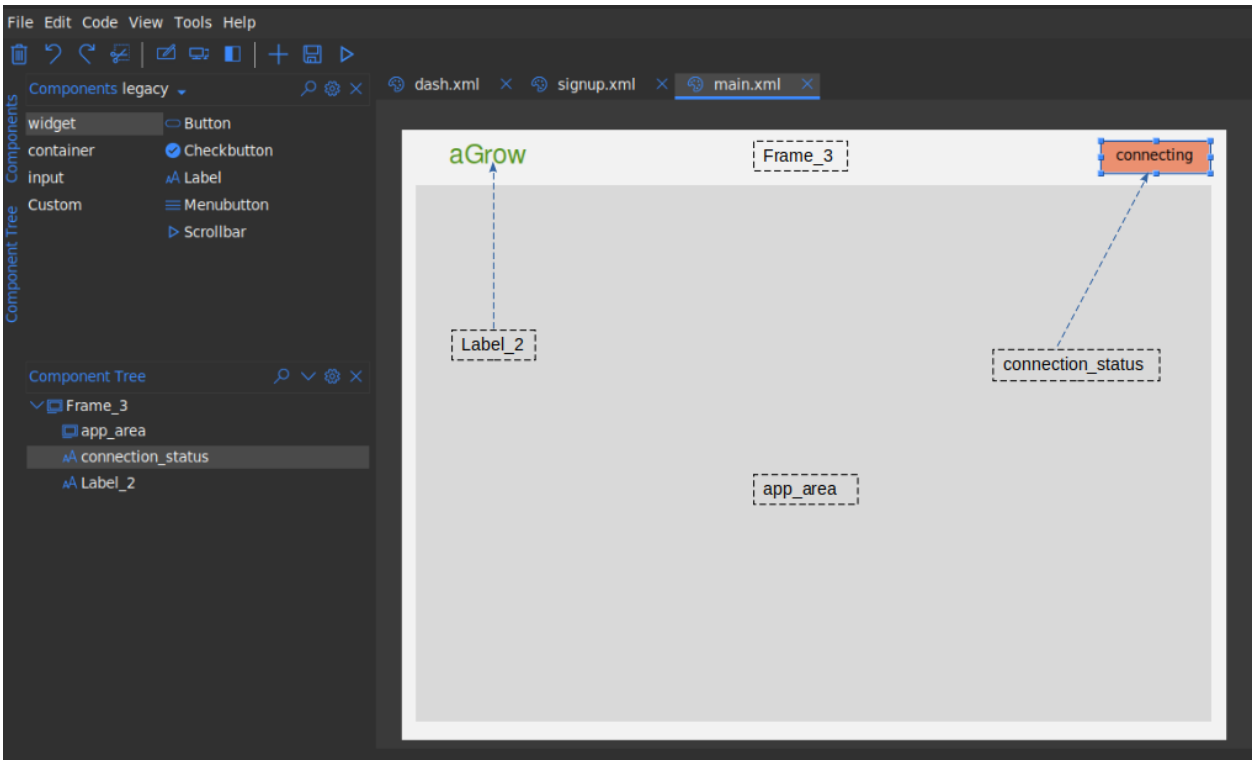


CHAPTER

FOUR

COMPONENT TREE

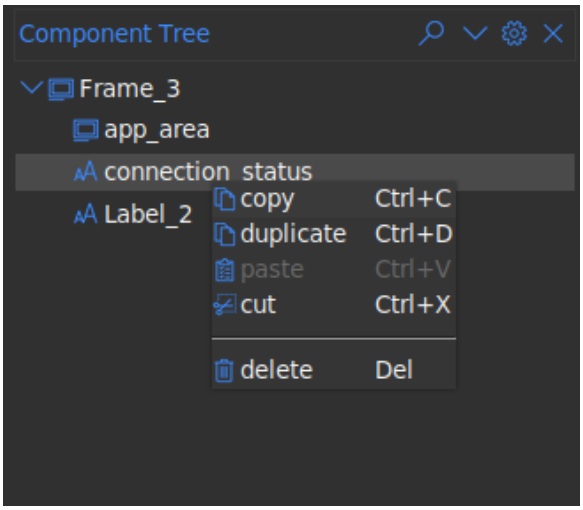
The component tree allows you to view widgets in your design in a hierarchical fashion. You can select widgets from the component tree by clicking on them. The text shown on the component tree corresponds to the widget id.



You can also access the widget context menu on the component tree as well. This menu is the same as what would be shown when you `right-click` on the widget in the design area

Note:

- You can use the search icon at the top of the pane to search through all widgets in the component tree
- You can use the collapse/expand icon at the top of the pane to collapse or expand all nodes in the component tree.


Component Tree
Frame_3
app_area
connection_status
Label_2
copy Ctrl+C
duplicate Ctrl+D
paste Ctrl+V
cut Ctrl+X
delete Del

CHAPTER

FIVE

CUSTOM WIDGETS

5.1 Introduction

Formation studio provides a way to use your own custom widgets apart from the builtin widget sets (tk and ttk). The widgets will be treated just as any other widget allowing you customize it using tools provided by the studio

5.2 Setting up

To make your custom widget usable in the studio, you will need to define a separate class with some metadata required by the studio. This class is what the studio will manipulate under the hood. Let us define a simple Custom widget that can be used as a d-pad like one would find on a game controller

```
from tkinter import Frame, Button

class DPad(Frame):

    def __init__(self, master, **kw):
        super(DPad, self).__init__(master, **kw)
        self.left = Button(self, text="L", padx=8, pady=5)
        self.right = Button(self, text="R", padx=8, pady=5)
        self.up = Button(self, text="U", padx=8, pady=5)
        self.down = Button(self, text="D", padx=8, pady=5)

        self.up.grid(row=0, column=1)
        self.left.grid(row=1, column=0)
        self.right.grid(row=1, column=2)
        self.down.grid(row=2, column=1)
```

This is a simple compound widget that really just displays the 4 buttons of a d-pad. To Add this to the studio we would need to define the class with the metadata. Below are some of the supported metadata.

Metadata	default	Description
display_name	name of the meta class	The name used to refer to the widget within the studio
impl	the meta's super class	The custom widget class
is_container	False	whether the custom widget allows other widgets to be placed within it.
icon	play	Text identifier to on of the built in icons to be used as image identifier for the widget
initial_dimension	Automatic	The initial dimensions of the widget when first placed on the design pad provided as a (width, height) tuple

Each of the above metadata is optional and the default will be used if not its not provided. To mark the metadata class for use by the studio we need to use the *WidgetMeta* class provided by the studio. It is a python metaclass and is responsible for the all the magic that goes on under the hood. Below is a sample metadata class for a our D-pad widget

```
from studio import WidgetMeta

class DPadMeta(DPad, metaclass=WidgetMeta):
    display_name = 'D Pad'
    # impl is not necessary and can be inferred from the inheritance list
    impl = DPad
    icon = "gaming"
    is_container = False
    initial_dimensions = 90, 100
```

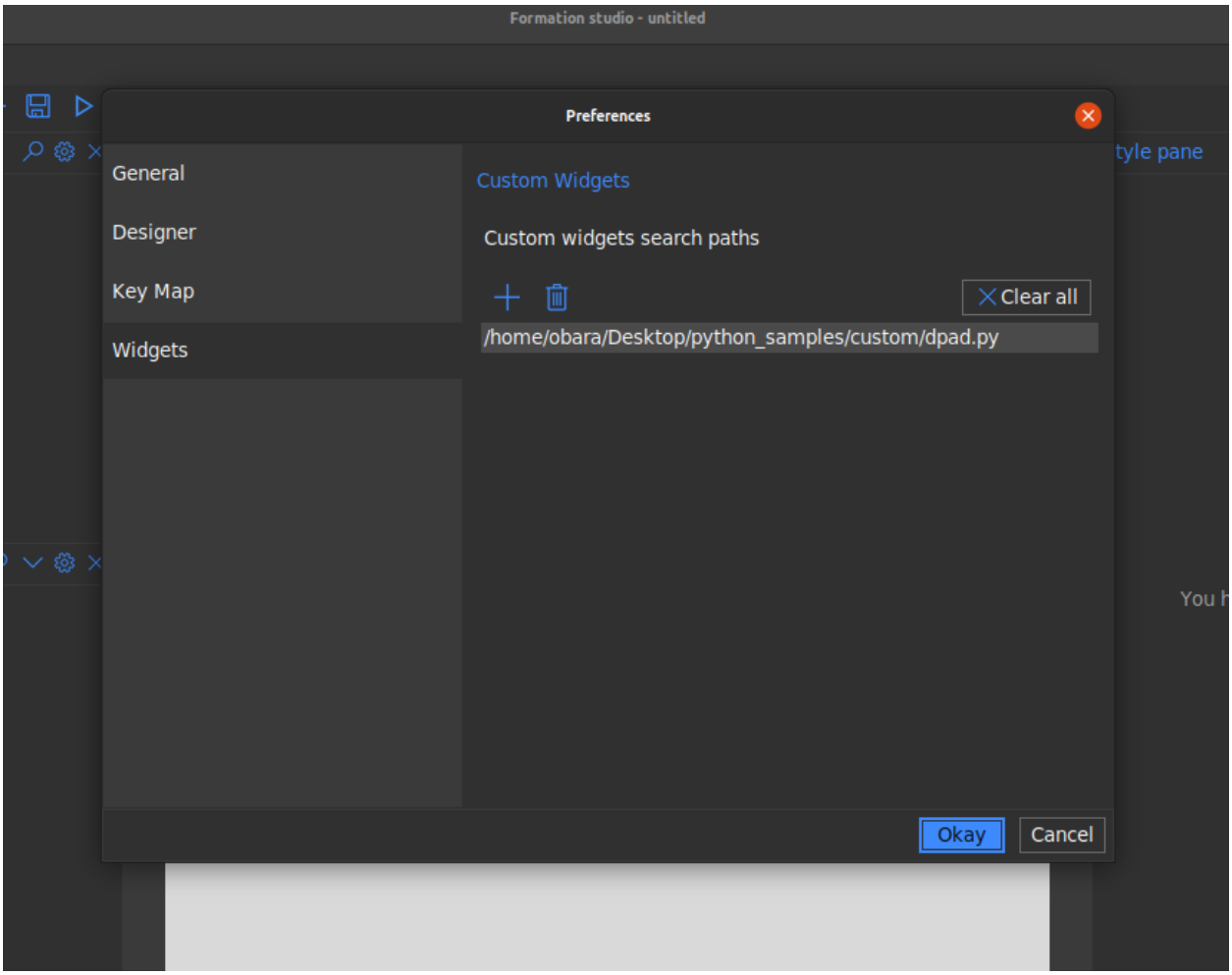
5.3 Connecting to the studio

We need to configure the path to the file containing our metadata class in the studio. We head `Settings > Widgets` . Click on the + icon and select the path to the file with the metadata class. Click `Okay` to save the changes.

The `dpad.py` file used here contains both the implementation and the metadata but that is not necessary. Only the metadata class is required to be in the file.

```
from tkinter import Frame, Button
from studio import WidgetMeta

class DPad(Frame):

    def __init__(self, master, **kw):
        super(DPad, self).__init__(master, **kw)
        self.left = Button(self, text="L", padx=8, pady=5)
        self.right = Button(self, text="R", padx=8, pady=5)
        self.up = Button(self, text="U", padx=8, pady=5)
        self.down = Button(self, text="D", padx=8, pady=5)

        self.up.grid(row=0, column=1)
```

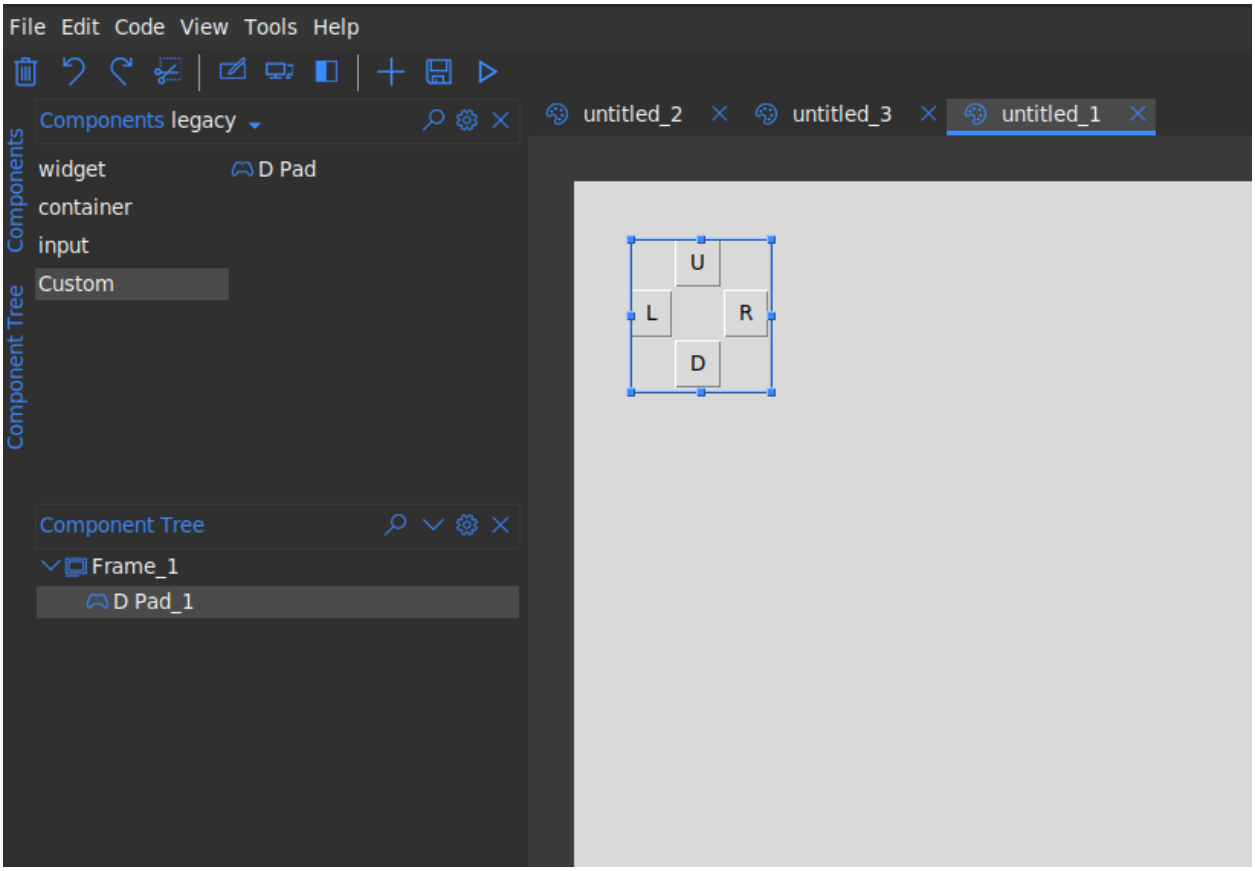
(continues on next page)


Formation studio - untitled
Preferences
General
Designer
Key Map
Widgets
Custom Widgets
Custom widgets search paths
Clear all
/home/obara/Desktop/python_samples/custom/dpad.py
Okay
Cancel

(continued from previous page)

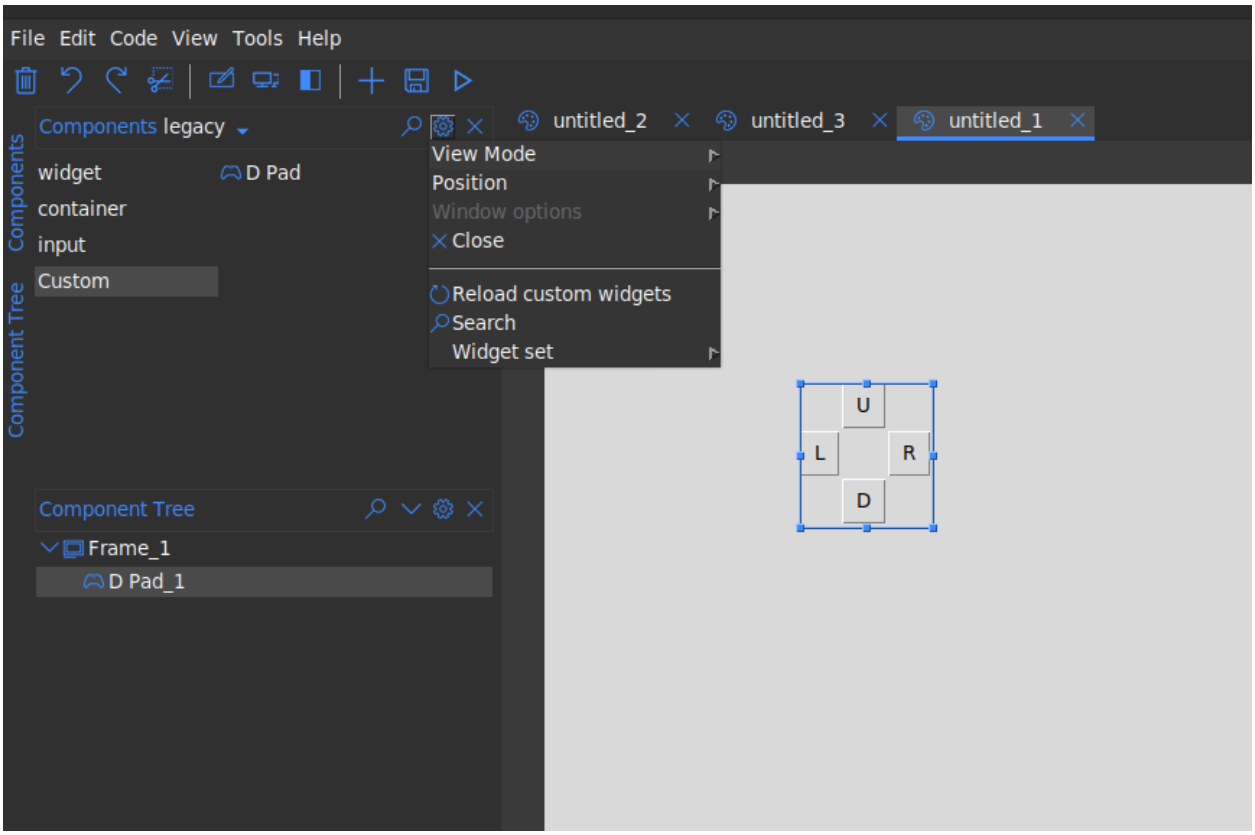
```
        self.left.grid(row=1, column=0)
        self.right.grid(row=1, column=2)
        self.down.grid(row=2, column=1)


class DPadMeta(DPad, metaclass=WidgetMeta):
    display_name = 'D Pad'
    impl = DPad
    icon = "gaming"
    is_container = False
    initial_dimensions = 90, 100
```

Our new custom widget should now be available in the components pane under the `custom` group.



5.4 Reloading Changes

If any changes are made to metadata class file, you can reload the changes without having to restart the studio. Just head to the component pane settings and select `Reload custom widgets`.



Note: Widgets already added to the design pad will not be affected. They will continue to use the old definitions until the design is reloaded the next time. It is advisable that you remove them before you save the design file to avoid nasty issues when reloading them next time.

CHAPTER

SIX

CUSTOM PROPERTIES

Formation studio allows you to specify custom properties for your *Custom Widgets*. Owing to the adaptive nature of how the studio handles properties adding custom properties is relatively easy. The studio relies on proper implementation of the `configure`, `keys`, `cget`, `__getitem__` and `__setitem__` methods of a widget for custom properties to be correctly detected. Luckily, formation provides utilities to achieve this through the `CustomPropertyMixin` Let us add a custom properties *button_color* and *button_bg* to our *Dpad* widget. We will modify our class to use the `CustomPropertyMixin` as shown below

```
from tkinter import Frame, Button
from formation.utils import CustomPropertyMixin

class DPad(CustomPropertyMixin, Frame):

    prop_info = {
        "button_color": {
            "name": "button_color",
            "default": None,
            "setter": "set_btn_fg",
            "getter": "_btn_fg"
        },
        "button_bg": {
            "name": "button_bg",
            "default": None,
            "setter": "set_btn_bg",
            "getter": "_btn_bg"
        }
    }

    def __init__(self, master, **kw):
        super(DPad, self).__init__(master, **kw)
        self.left = Button(self, text="L", padx=8, pady=5)
        self.right = Button(self, text="R", padx=8, pady=5)
        self.up = Button(self, text="U", padx=8, pady=5)
        self.down = Button(self, text="D", padx=8, pady=5)

        self.up.grid(row=0, column=1)
        self.left.grid(row=1, column=0)
        self.right.grid(row=1, column=2)
        self.down.grid(row=2, column=1)

        self._btns = [self.left, self.right, self.up, self.down]
```

(continues on next page)

(continued from previous page)

```
        self._btn_bg = self.left["bg"]
        self._btn_fg = self.left["fg"]

    def set_btn_bg(self, val):
        for i in self._btns:
            i["bg"] = val
        self._btn_bg = val

    def set_btn_fg(self, val):
        for i in self._btns:
            i["fg"] = val
        self._btn_fg = val
```

Note: The `CustomPropertyMixin` is not necessary if configure, keys and the other methods are already implemented to accommodate you custom properties. It is however advisable to use the mixin as it has been thoroughly tested and is less prone to issues.

Our widget is ready for use. We still need to inform the studio on how our properties should be handled and what type of values they contain. The studio supports the following property types:

- **choice** : Allows selection from a set of values using a dropdown widget. Options are specified as a tuple using the `options` key.
- **boolean** : Selection of true or false using a checkbutton
- **relief** : Allows selection from the available relief types in tkinter
- **cursor** : Allows selection from available cursor types in tkinter
- **bitmap** : Allows selection from the built-in bitmaps
- **color** : Provides a colorpicker dialog to select colors
- **text** : Allows entry of arbitrary text
- **textarea** : Similar to text but allows entry of longer texts.
- **number** : Entry of integers
- **float** : Entry of floating point numbers
- **duration** : Allow entry of durations. You can specify the `units` options which can be one of ('ns', 'ms', 'sec', 'min', 'hrs').
- **font** : Selection from available system fonts. It also includes a font picker that can pick fonts from anywhere within the studio.
- **dimension** : Entry of dimension. Currently only supports pixels
- **anchor** : Allows easy setting of anchor and sticky values by providing realtime preview of anchor/sticky behaviour on a dummy widget. Setting the `multiple` option allows the application of multiple anchors simultaneously
- **image** : Allows user to pick an image from their local machine
- **variable** : Allows user to select from variables created by the `Variable pane`
- **stringvariable**: A variation of the variable type that only allows selection of `tk.StringVar`

Note: It is currently not possible to implement your own types but we hope to make allow custom types in future.

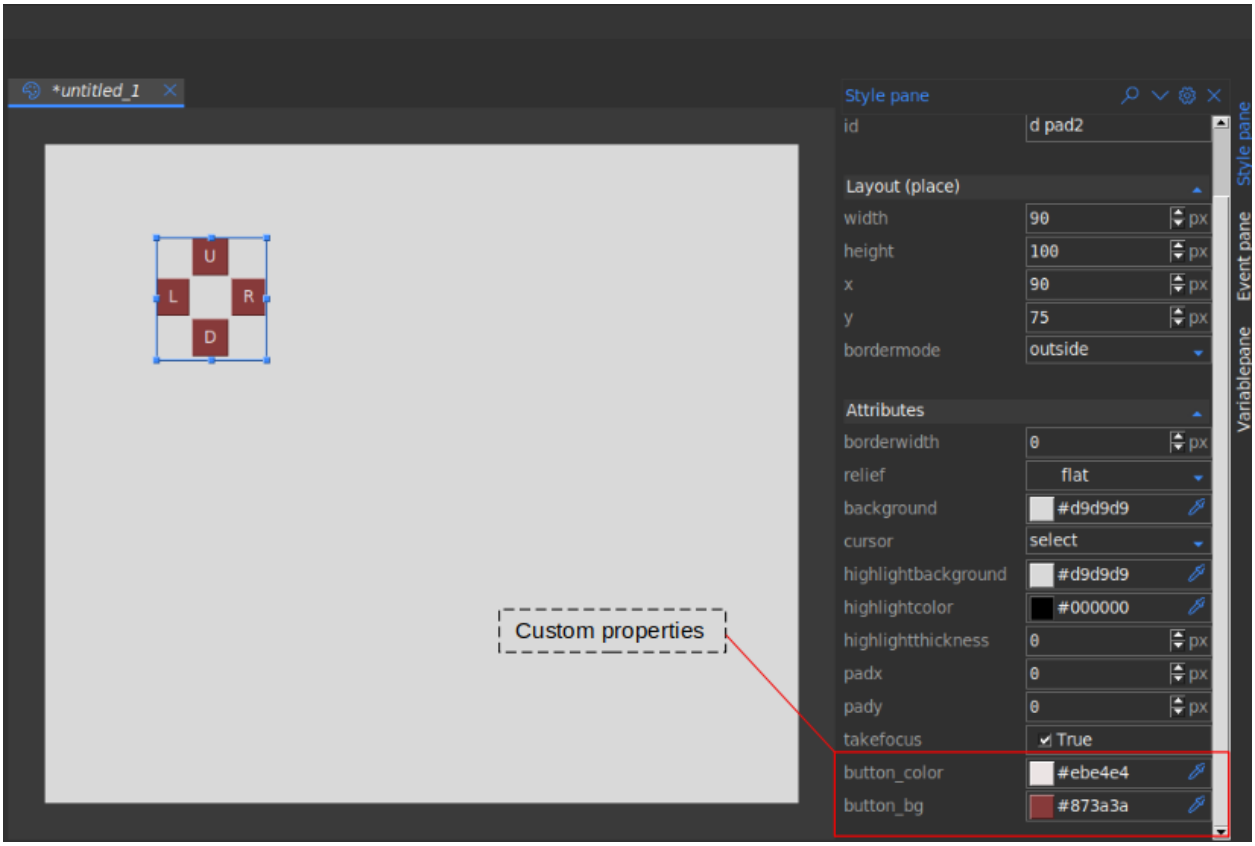
To specify the types our custom properties, we will modify the meta class as shown below:

```
class DPadMeta(DPad, metaclass=WidgetMeta):
    display_name = 'D Pad'
    impl = DPad
    icon = "gaming"
    is_container = False
    initial_dimensions = 90, 100

    DEF_OVERRIDES = {
        "button_color": {
            "display_name": "button color",
            "type": "color",
            # you can specify additional options supported by type here
            "name": "button_color"
        },
        "button_bg": {
            "display_name": "button bg",
            "type": "color",
            "name": "button_bg"
        }
    }
```

`DEF_OVERRIDES` is a special attribute checked at runtime by the studio to make decisions on what properties to display and how. You can also override behaviour of default properties by specifying alternative definitions here.

Note: The key and the `name` should always match to avoid issues.

Assuming your widget is properly setup as explained in *Custom Widgets*, if you open the studio and use your custom widget, the custom properties will appear in the `attributes` section on the `stylepane` as shown below


*untitled_1
U
L
R
D
Custom properties
Style pane
id
d pad2
Layout (place)
width
90
px
height
100
px
x
90
px
y
75
px
bordermode
outside
Attributes
borderwidth
0
px
relief
flat
background
#d9d9d9
cursor
select
highlightbackground
#d9d9d9
highlightcolor
#000000
highlightthickness
0
px
padx
0
px
pady
0
px
takefocus
True
button_color
#ebe4e4
button_bg
#873a3a
Style pane
Event pane
Variablepane

CHAPTER

SEVEN

CANVAS

Canvas

CHAPTER

EIGHT

SIMPLE CALCULATOR

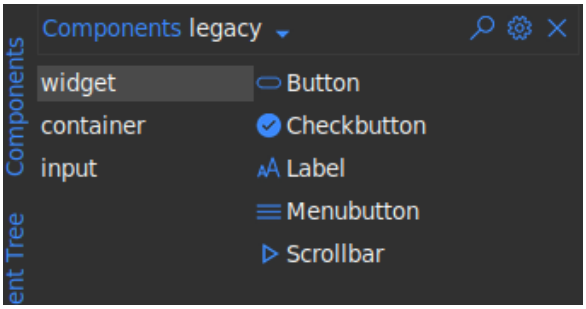
8.1 Setting up

We are going to be building a simple calculator to familiarise ourselves with the basic features of formation studio. Assuming you already have formation studio installed on your machine (if not, see *Installation* instructions) fire up the studio in the terminal as shown below.

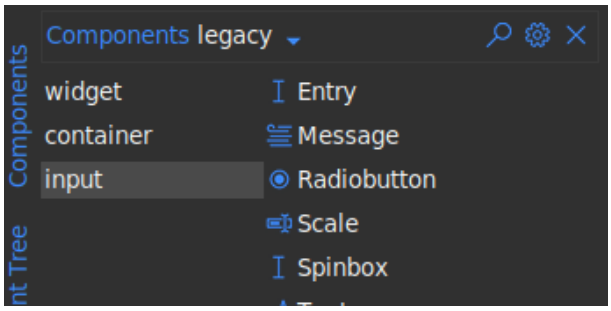
```
formation-studio
```

8.2 Creating the design

A blank design will open up assuming you are using default settings. On the **components** pane on the top left, select `legacy` on the drop down menu to use classic tkinter widgets and not themed `ttk` widgets. This will allow us to customize more attributes. On the vertical tab on the left, select `widget` and drag one `Button` and one `Label` to the design pad.

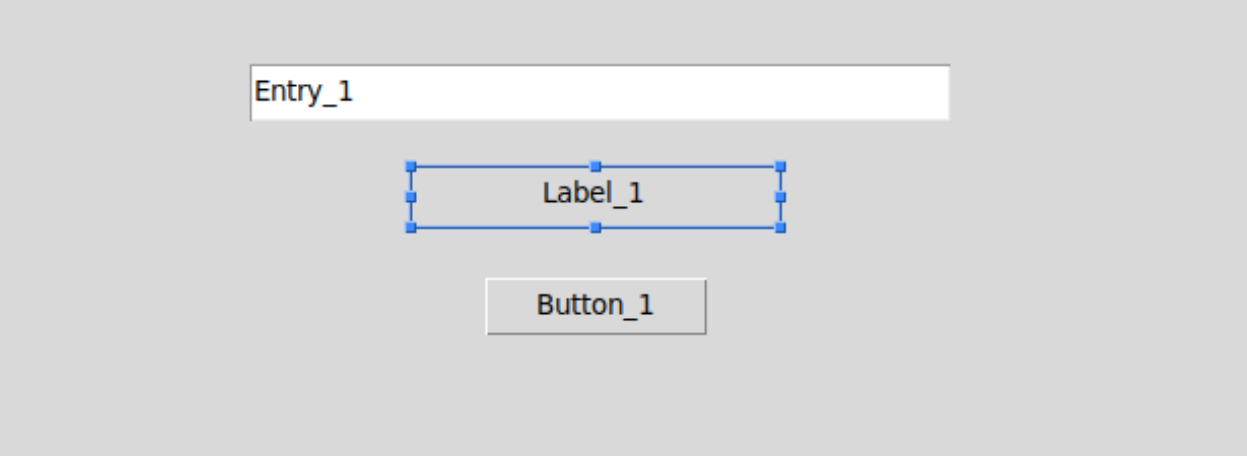


Now select `input` on the vertical tab to access a new set of widgets and drag one `Entry` to the design pad.

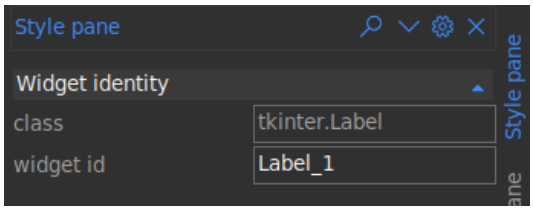


Arrange them as you please and you should have something as shown below.

Note: To **move** a widget around in the editor you will need to hold the *shift* key down when dragging. Alternatively you can move the cursor to the edges of the widget after selecting it and drag when the "hand" cursor appears. To **resize** a widget, drag the small squares at the edges and corners



We now set the `widget id` of these widgets. This is the most important part since this is the same id you will use to access the widget in your program. To set the `widget id` use the style pane on the right. The option will always be at the top in the `widget identity` section



For the purpose of this tutorial, set the `widget id` for the widgets added above as follows

- For the *Entry* widget set `widget id` to **expr**
- For the *Label* widget set `widget id` to **result**
- For the *Button* widget set `widget id` to **calculate**

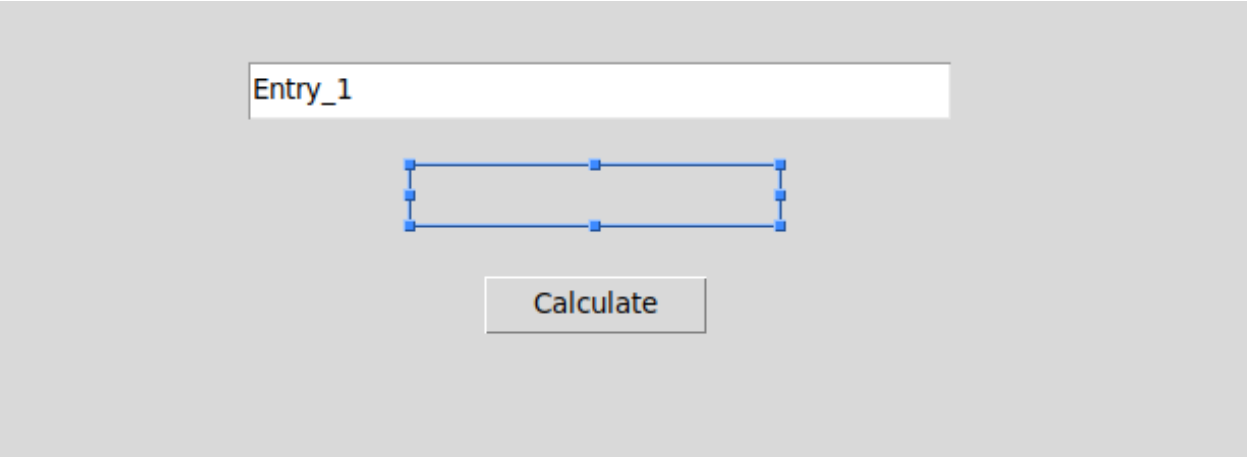
Double click the `Label` and the `Button` and change the texts as shown below. Leave the label blank since we will display the results of the calculations here.

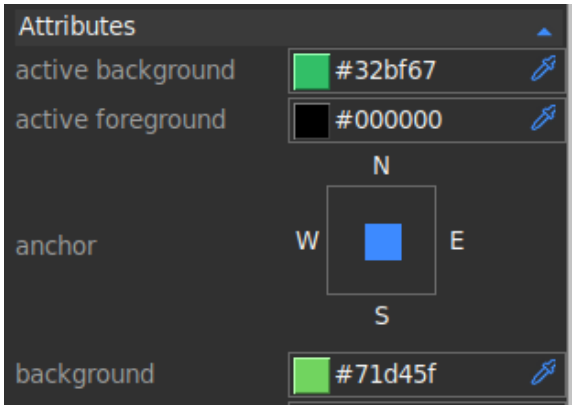
Alternatively, you can use the style pane to set the text attribute along with other style options you deem fit. The style pane as a whole is divided into 3 main parts

- **Widget identity** : contains the class and the id of the widget
- **Layout** : contains options that control the positioning the widget within its parent.
- **Attributes** : contains options that control the style and other aspects of the widget

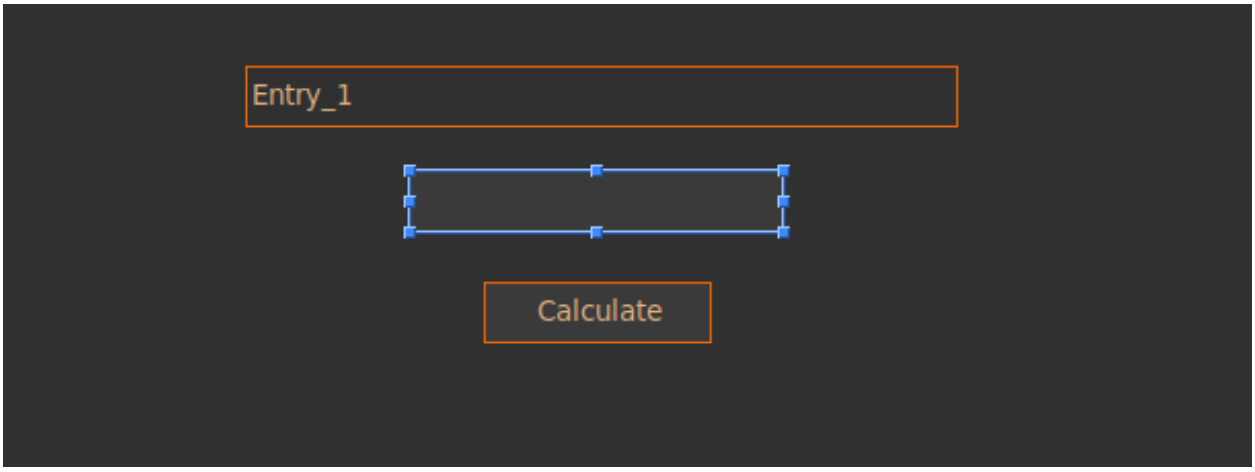
Play around with the styles to achieve your desired look. Try changing the colors and fonts. The design can look however you want.

Note: When selecting color, you can use the dropper on the right to select color from anywhere on your screen. The colored box on the left can be used to bring up the color picker to allow you more fine-grained control over the color.


Entry_1
Calculate

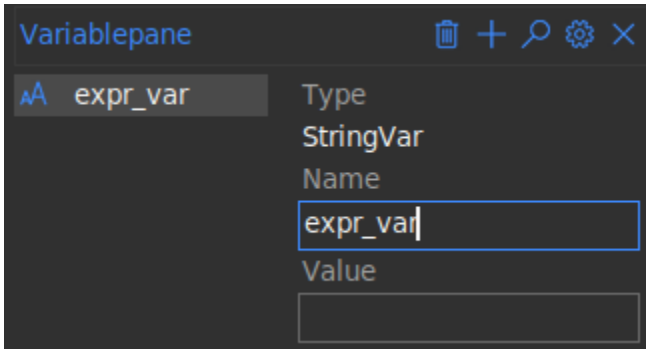

Attributes
active background
#32bf67
active foreground
#000000
N
anchor
W
E
S
background
#71d45f

You can also type your desired color name directly on the color entry box.

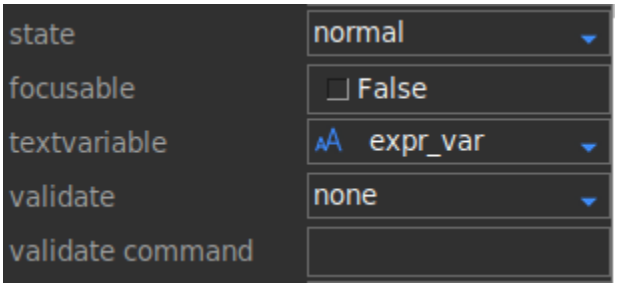


8.3 Connecting Variables

To access values from our `Entry` we will need to connect a variable to it. We can then access the value contained in the entry through the variable. To add a variable, on the right edge of the studio, select `Variablepane`. A new tool pane will open. Click on the "plus" icon at the top right and select `String` on the drop down menu. A new `String` variable will be created as shown below. Set the name to `expr_var`. Once again, this is an important value and will be used to access the variable in your program.



Now select our `Entry` widget in the design pad and search through the `attributes` section of the stylepane for the `textvariable` option. We have only created one variable named `expr_var` so select that.



Note: Once a variable has been created in the **VariablePane** it can be connected to multiple widgets through the `variable` and `textvariable` options allowing you to control the value in multiple widgets with just one variable.

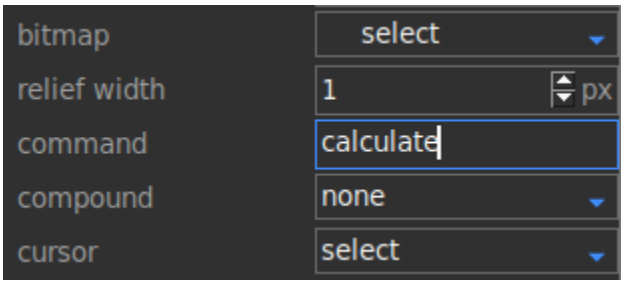
8.4 Connecting Commands

There are two ways of connecting commands in formation studio. We'll start with the easiest one

Note: Pick only one of the methods below (preferably the first one) since they basically do the same thing in different ways. The second method is more advanced and can be used to achieve more complex bindings.

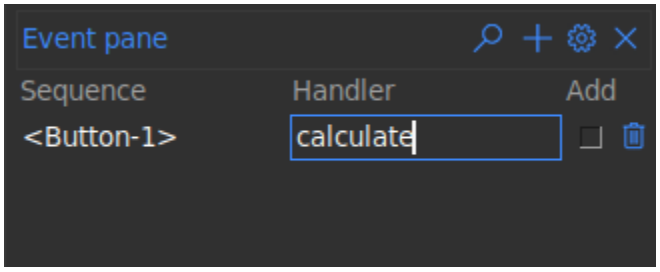
8.4.1 1. Using the command option

This is the easiest method. It is however limited and can only bind click events to buttons and other widgets with the `command` option. To bind a command, simply enter the name of the function to be called when the **calculate** `Button` is clicked. This is the same name we will use when defining our callback function so for the sake of the tutorial lets call it `calculate`



8.4.2 2. Using event bindings

This method can be used to bind all sorts of events since it uses tkinter's `bind` method under the hood. The binding is done pretty much the same way. To bind first select the **calculate** `Button` then open the `Eventpane` from the right edge of the studio. Click the "plus" icon at the top right of the pane to add a new event binding. Then fill out the `Sequence` and `Handler` as shown below.



8.5 Wrapping up the design

Save the design file as `calculator.xml` by doing any of the following

- Go to main menu `File > Save`
- Press `Ctrl+S`
- Click on the "Floppy disk" icon in the tool bar

8.6 Writing the code

In the same folder where `calculator.xml` is saved, create a python file named `calculator.py`. To load our design file we will need to import formation loaders and load `calculator.xml` as shown below. We will use `AppBuilder` which will create a toplevel window for us. If you wanted to only load a section and code the rest of the app yourself you would use `Builder` instead.

```
from formation import AppBuilder

app = AppBuilder(path="calculator.xml")
```

Now let's define our `calculate` function which we are to link to the app. This function will be called when the **calculate** `Button` is clicked

```
def calculate(event=None):
    # event parameter needs to be there because using the bind method passes an event␣
↪object
    # access the expr_var we created earlier to determine the current expression entered
    expr = app.expr_var.get()

    # evaluate the expression
    try:
        result = eval(expr)
    except Exception:
        # if the expression entered was malformed and could not be evaluated
        # we will display an error message instead
        result = "Invalid expression"

    # display the result
    app.result.config(text=result)
```

We will now connect the `calculate` function to our app

```
app.connect_callbacks({"calculate": calculate})
```

Alternatively, since the function is in the global scope, you can connect it directly using python's `globals()` function

```
app.connect_callbacks(globals())
```

Now everything is done we can fire app our app's `mainloop` to get the app running

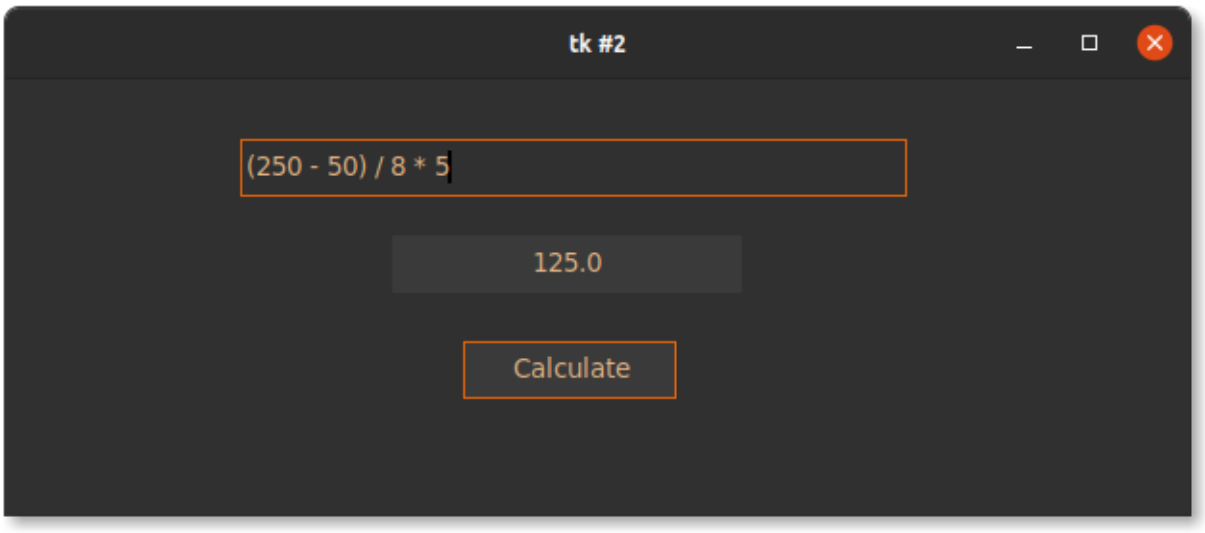
```
app.mainloop()
```

8.6.1 Wrapping it up

The complete code to run our app which will be located at `calculate.py` will be

```
from formation import AppBuilder

app = AppBuilder(path="calculator.xml")


def calculate(event=None):
    # event parameter needs to be there because using the bind method passes an event object
    # access the expr_var we created earlier to determine the current expression entered
    expr = app.expr_var.get()

    # evaluate the expression
    try:
        result = eval(expr)
    except Exception:
        # if the expression entered was malformed and could not be evaluated
        # we will display an error message instead
        result = "Invalid expression"

    # display the result
    app.result.config(text=result)


app.connect_callbacks(globals())

app.mainloop()
```

You can now run `calculator.py` and it should display your beautiful working app. Type a simple mathematical expression in the entry box and click "calculate" and it should display the computed result



8.7 Conclusion

This was a simple example to get you started. You can learn to build more complex applications using the vast number of widgets available with just about the same ease as building the one in this tutorial. You can find the detailed API reference for formation loaders used above in the *Loading a design* section.

CHAPTER

NINE

FORMATION

9.1 Loading a design

9.2 Formation utilities

Documentation for useful utilities provided by formation

CHAPTER

TEN

HOVERSET

10.1 Widget Catalogue

10.2 Dialog Windows

10.3 Working with menus

class hoverset.ui.menu.**EnableIf**(*predicate*, **templates*)

Built in manipulator that displays only a set of menu items if a condition is met at runtime. If condition is nt met, the menu items are displayed but are disabled

manipulated()

Generate templates to be rendered when menu is displayed

Returns
manipulated templates menu

class hoverset.ui.menu.**LoadLater**(*loader*)

A built in manipulator that generates templates at runtime using a loader function passed in its constructor

manipulated()

Generate templates to be rendered when menu is displayed

Returns
manipulated templates menu

class hoverset.ui.menu.**Manipulator**(**templates*)

Enables simplification of creation of dynamic menus allowing menu items to be easily manipulated at runtime

manipulated()

Generate templates to be rendered when menu is displayed

Returns
manipulated templates menu

class hoverset.ui.menu.**ShowIf**(*predicate*, **templates*)

Builtin manipulator that displays a set of menu items only if a certain condition is met at runtime

manipulated()

Generate templates to be rendered when menu is displayed

Returns
manipulated templates menu

hoverset.ui.menu.**dynamic_menu**(*func*)

Generate a dynamic menu from a class method

Parameters

func – An instance method taking one positional argument menu. This method wil be called every time the menu needs to be posted. Note that the menu will always be cleared before the method is called

Returns

the wrapped method returns a dynamic menu

CHAPTER

ELEVEN

STUDIO

11.1 Studio Architecture

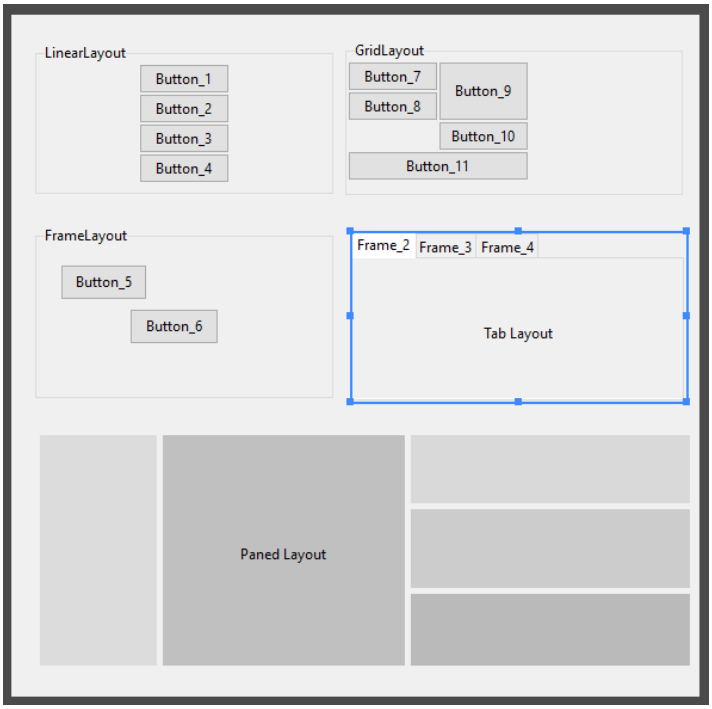
11.1.1 Introduction

- **Formation** is a design tool intended to make the work of tkinter UI designers easy by providing an intuitive drag drop interface. It allows designers to employ all layout managers (Pack, Place and Grid) to flexibly achieve various design goals. For ease of implementation the designer itself is written in what we like to call contemporary tkinter provided by the hoverset library. You can view the widget catalogue at `hoverset.ui.widgets`. The supported widgets have been organised into families of widgets referred to here as **widget sets** and include:
  - Tkinter default (Legacy)
  - Tkinter ttk extension (Native implementations of tkinter widgets)
  - Extension set (Incomplete)
  - Hoverset widget set (Incomplete)

11.1.2 Features Description

A feature is a complete tool window providing means to manipulate the design file. Below are core features implemented in the formation studio:

- **Drag drop designer**: Formation provides an easy to use drag drop designer. The designer can be expanded to full screen display to allow focus on design. The designer allows widgets to be moved from parent to parent as needed to simplify the design process. The designer supports manipulation using the following layout strategies:
  - pack (formerly LinearLayout)
  - grid (formerly GridLayout)
  - place (formerly FrameLayout)
  - TabLayout (for py:class:*tkinter.ttk.Notebook*)
  - PaneLayout (for PanedWindows)
- **Component library** The component library allows designers to search through the supported widget sets and add them to the designer. They can filter the components based on their widget sets.
- **Component Tree** Display the Widget hierarchy and select widgets that may be (due to design) difficult to access directly from the design pane. Access the context menu of the widget from the component tree which is basically just an extension/handle of the widget.



- **Style pane** Access the style and layout attributes of selected widgets. The layout attributes automatically switch to match the layout manager currently handling the widget. Easily manage a wide range of properties using intuitive editors such as:
  - Color: Modify color in RGB, HSL and HSV and hex notation as well as pick colors from anywhere on your computer screen even outside formation itself
  - Anchor: Intuitively set anchors as well as sticky attributes
  - Text: Write out text values with ease
  - Choice: Get all options valid for a given property
  - Dimensions: Handles all tkinter dimension notations
  - Boolean: Toggle between boolean attributes with a click
- **Menu editor** Create and edit menus using easy to use drag and drop gestures. Access all attributes applicable to the various types of menu items and preview the modified menu with the click of a button.
- **Variable pane** Create tkinter control variables, access and assign them to widgets in the designer. Modify the values of the variables on the fly from the manager window. Any control variables added from the manager immediately reflect in the style pane allowing the designer to assign them to as many widgets as they desire. Control variables provide an elegant way to set values to connected widgets which rely on the same value.

11.1.3 Structure

- `studio.feature` : Contains implementation of the various key components of the designer such as:
  - `studio.feature.component_tree`
  - `studio.feature.design`
  - `studio.feature.components`
  - `studio.feature.stylepane`
  - `studio.feature.variablepane`

  These components all implement `studio.feature._base.BaseFeature` which abstracts all Feature behaviour and manipulation which can then be built upon if special behaviour is needed. It contains methods that are to be overridden so as to handle events broadcast by the main application such as change in widget selection or deletion of a widget among others.
- `studio.lib` : Contains implementation of widget sets, complete definitions of their properties, behaviour. It also has implementation for the various layouts used by the designer. Definitions and implementation of menus and properties that can be applied to the menu components can also be found here. The files under this folder are:
  - `studio.lib.layouts`: layout implementation
  - `studio.lib.legacy`: classic tkinter widget definition
  - `studio.lib.native`: ttk themed widget extension widgets
  - `studio.lib.properties`: definition for all widget properties modifiable by the style pane.
  - `studio.lib.pseudo`: Base classes for widgets used in the studio designer with added functionality to allow for easy manipulation. Definition for container widgets can also be found here
  - `studio.lib.menu`: Utilities and definitions for handling menus in the studio
  - `studio.lib.variables`: Classes for managing tk variables in the studio
- `studio.parsers` : Contains implementation for classes that handle conversion from various designated file formats to design view and vice versa. Currently on only xml defined in `studio.parsers.xml` format is supported but if any other formats are to be added this would be the package location
- `studio.ui`: Contain implementation of widgets and user interface components used in the studio. The included are:
  - `studio.ui.editors`: The ui elements used to modify various widget properties as explained in the style pane feature
  - `studio.ui.geometry`: Access, analyse and manipulate position and sizes of widgets used by various studio routines
  - `studio.ui.highlight`: Transient widgets used to guide designers to which widgets currently have focus. Also contains implementations for resizing and moving widgets in the designer
  - `studio.ui.tree`: Implementation of base class for the tree view widgets used in the studio which allows easy manipulation using drag drop gestures
  - `studio.ui.widgets`: Assortment of special widgets used in the studio
  - `studio.ui.about`: The about window for the studio
- `studio.main`: Contains the entry point of studio user interface. Implementation for general functionality and the coordination of feature windows can be found inside the `studio.main.StudioApplication` class

11.2 Geometry functions

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX