Formation

Emmanuel Obara

Oct 17, 2022

GETTING STARTED

1 Installation 1

2 Quick Start 3

3 Component Pane 7

4 Component Tree 9

5 Custom Widgets 11

6 Custom Properties 17

7 Canvas 21

8 Simple Calculator 23

9 Formation 31

10 Hoverset 35

11 Studio 57

12 Indices and tables 63

Python Module Index 65

Index 67

CHAPTER ONE

INSTALLATION

To use Formation studio, python 3.6 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:

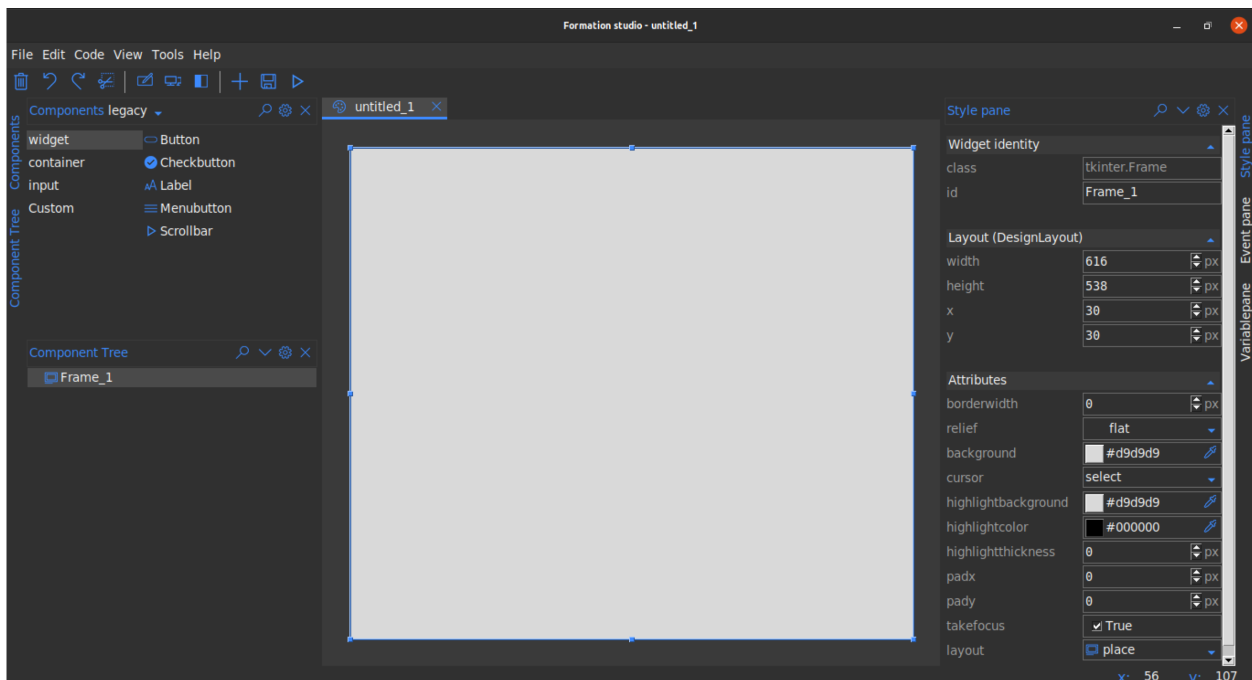
```
formation-studio
```

CHAPTER TWO

QUICK START

Launch the studio from the commandline using the command

```
formation-studio
```

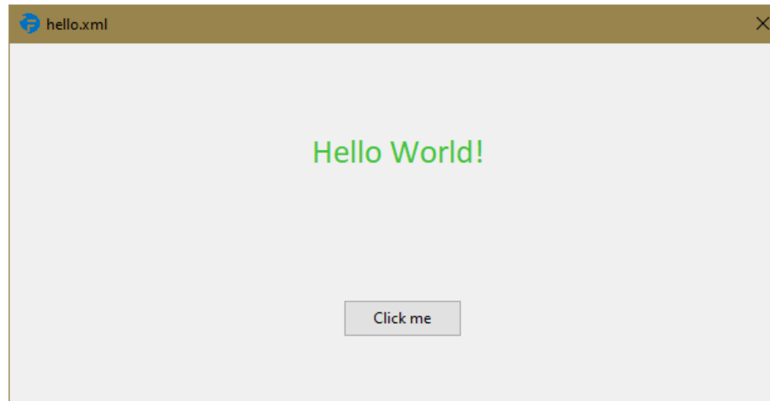


You can select widgets from the **Components** pane at the top and drag them onto the stage. Click to select widgets on the workspace and customize them on **Stylepane** to the right. You can view your widget hierarchies from the **Component tree** at the bottom left. To preview the the design, use the 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"
```

(continues on next page)



(continued from previous page)

```
    layout:height="287"
    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

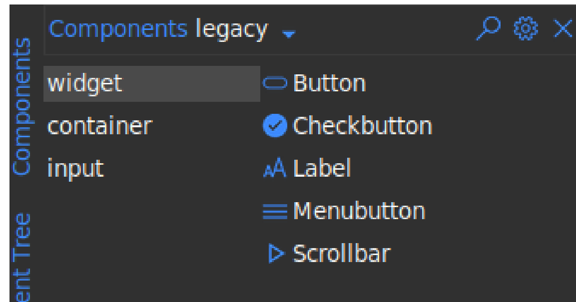
Note:

- 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:

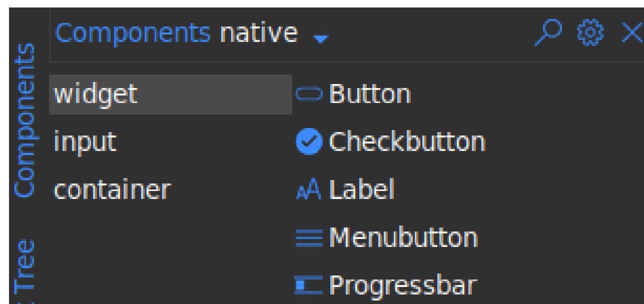
- **Listbox** (*displays a list of items*)
- **Canvas** (*allows flexible drawing of shapes, images and text*)
- **Text** (*text area allowing multiline text input*)
- **Message** (*label for longer text*)



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*)

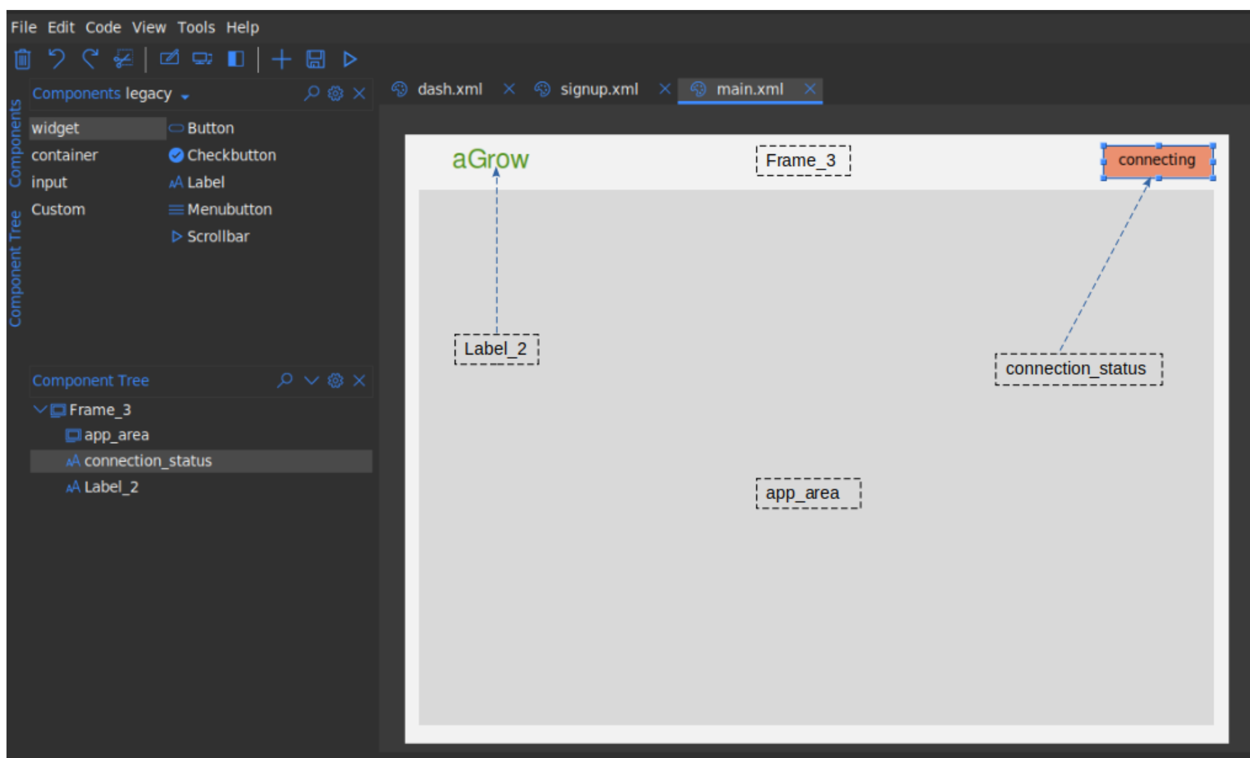


CHAPTER

FOUR

COMPONENT TREE

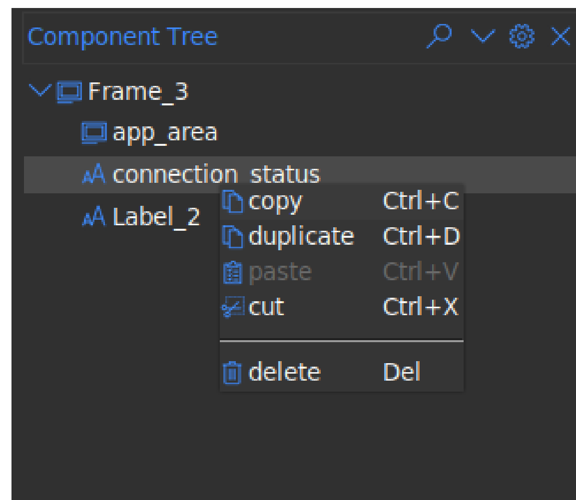
The component tree allows you to view widgets in your design in a hierarchical fashion. You can select widgets from the component tree by clicking on them. The text shown on the component tree corresponds to the widget id.



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_dimensions	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
```

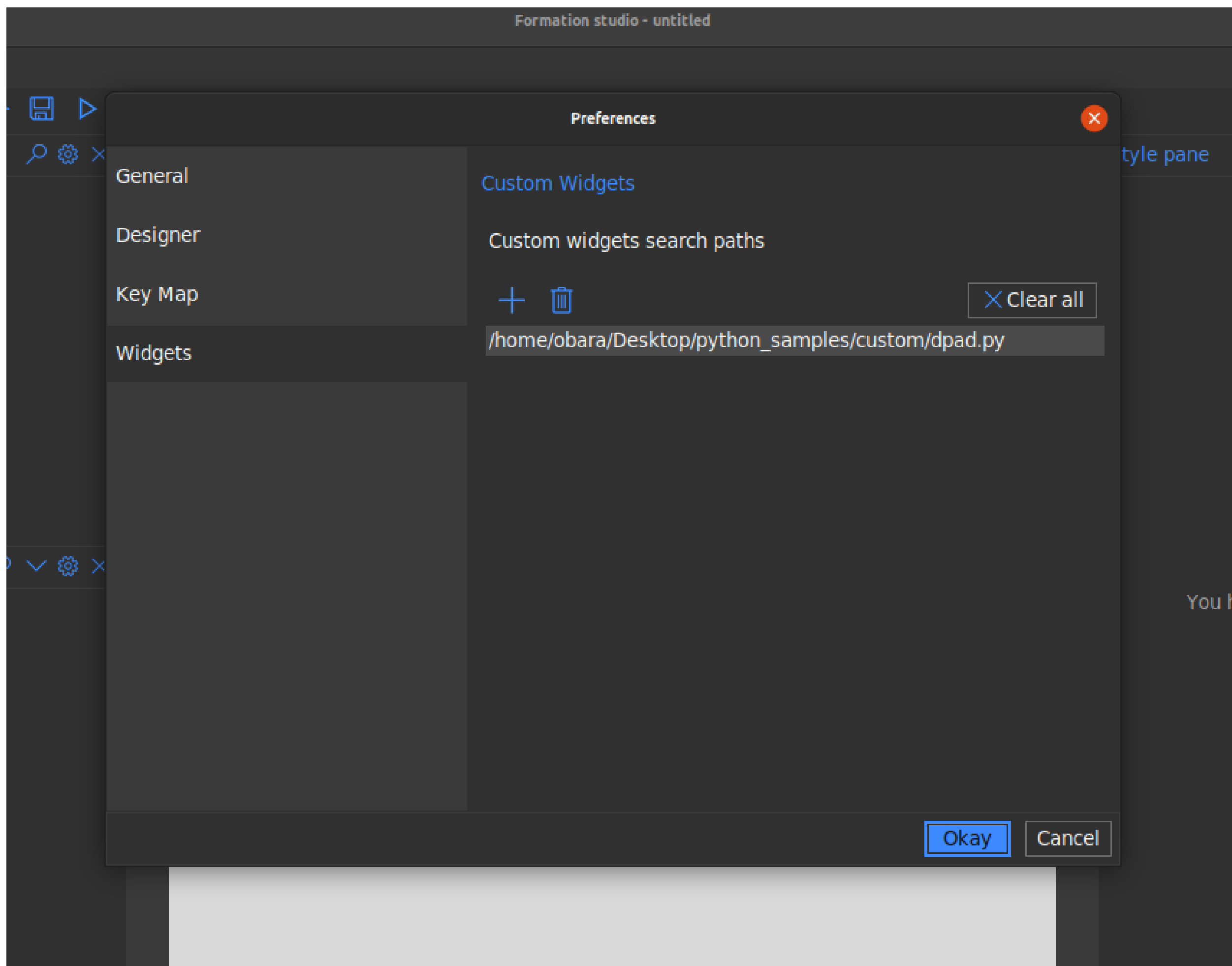
5.3 Connecting to the studio

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)
```

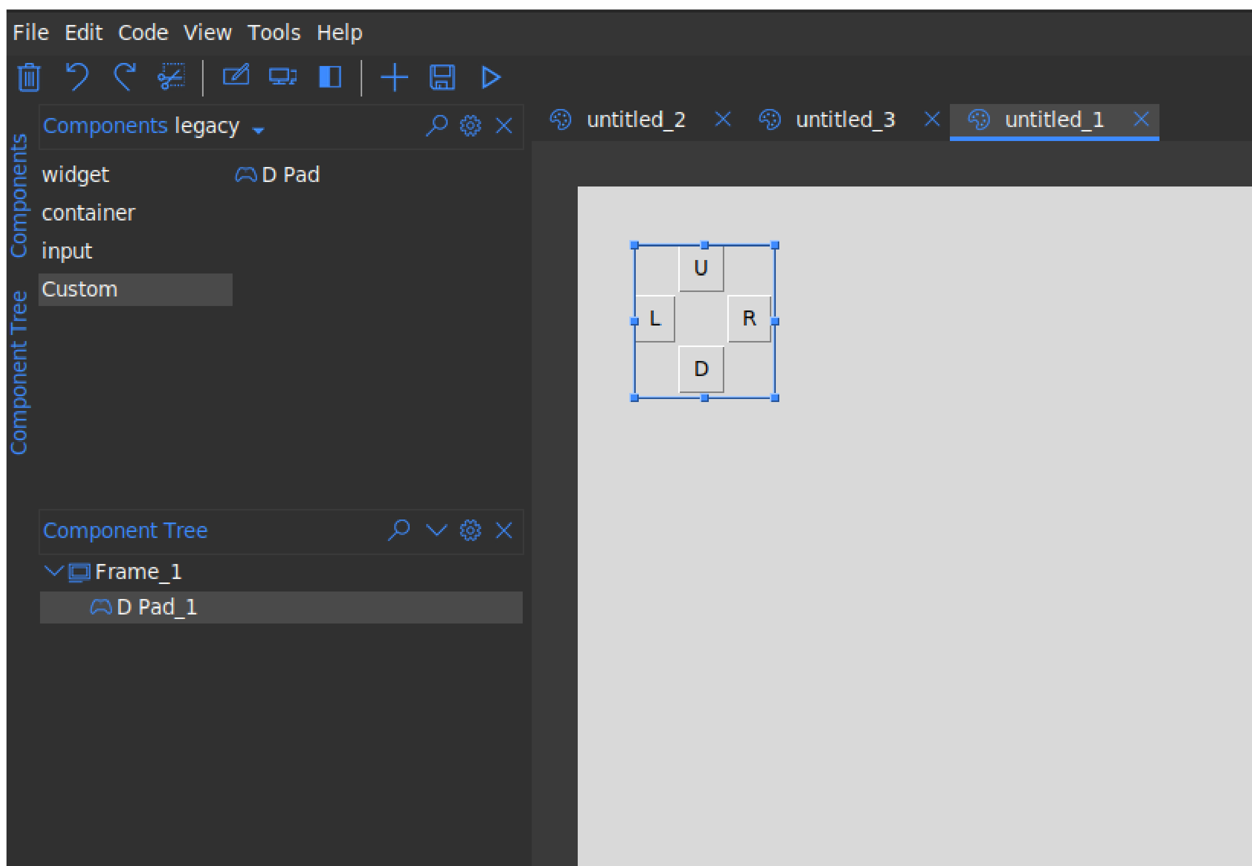
(continues on next page)


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)

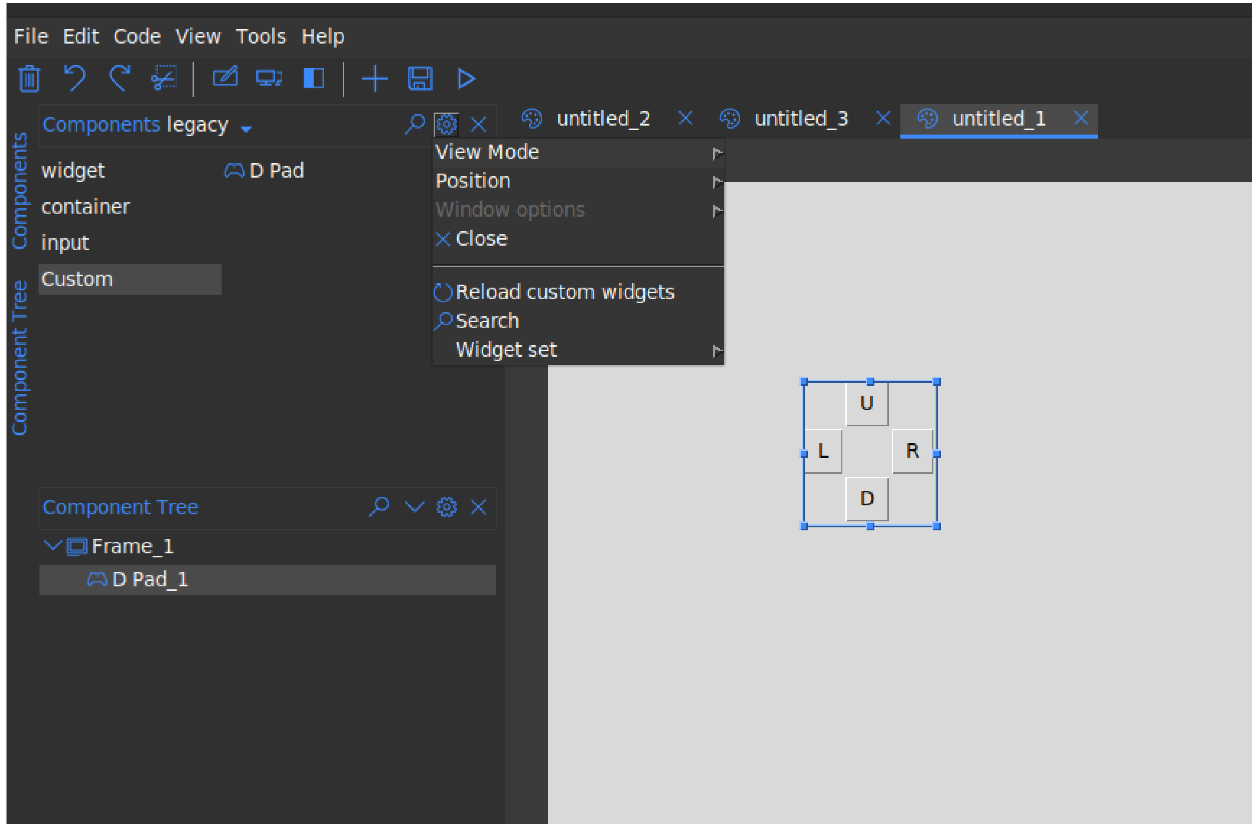
```
        self.left.grid(row=1, column=0)
        self.right.grid(row=1, column=2)
        self.down.grid(row=2, column=1)


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.

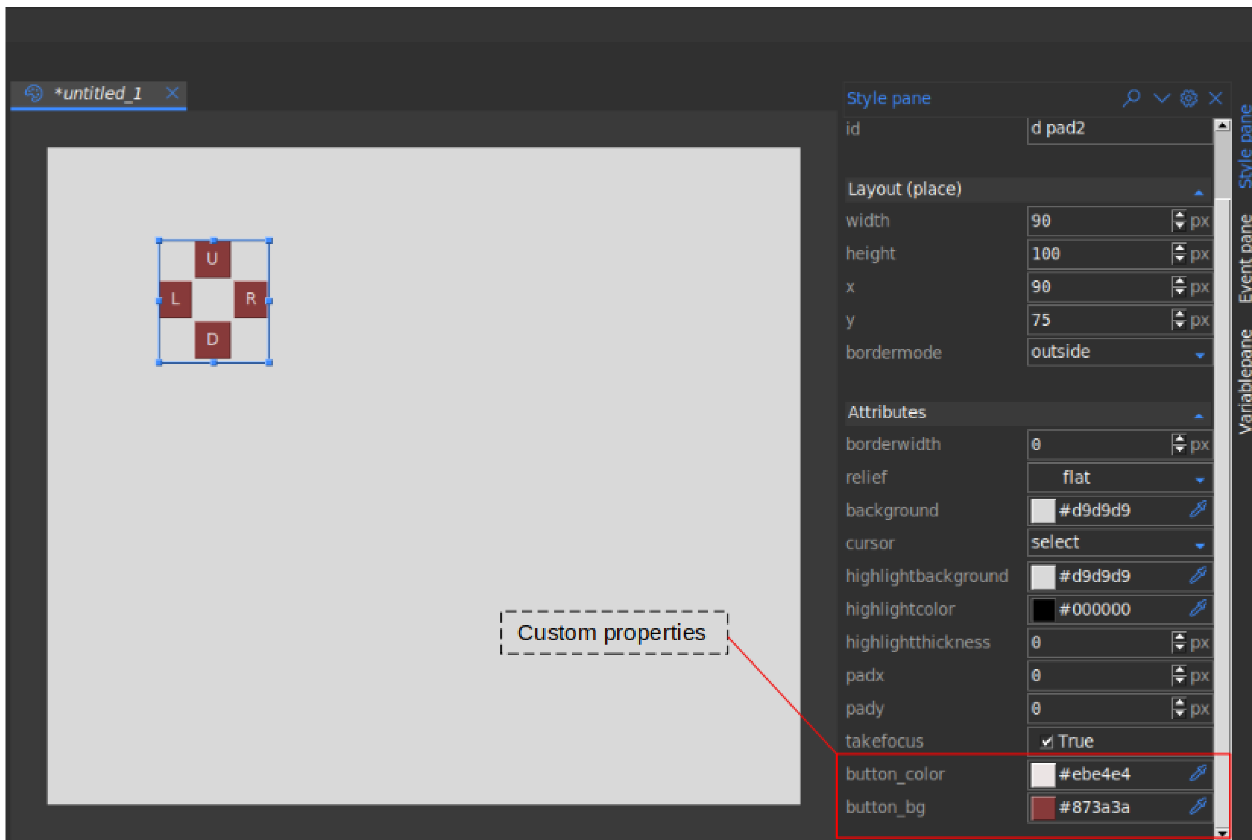
To specify the types our custom properties, we will modify the meta class as shown below:

```
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
height
100
x
90
y
75
bordermode
outside
Attributes
borderwidth
0
relief
flat
background
#d9d9d9
cursor
select
highlightbackground
#d9d9d9
highlightcolor
#000000
highlightthickness
0
padx
0
pady
0
takefocus
True
button_color
#ebe4e4
button_bg
#873a3a
Style pane
Event pane
Variablepane

CHAPTER
SEVEN

CANVAS

Canvas

CHAPTER

EIGHT

SIMPLE CALCULATOR

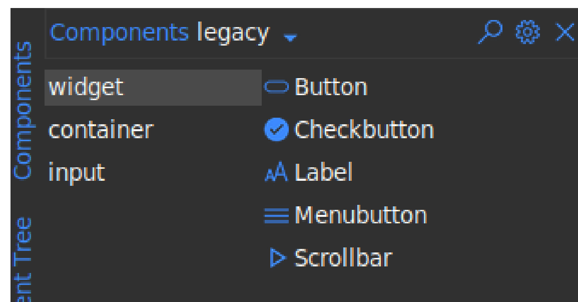
8.1 Setting up

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.

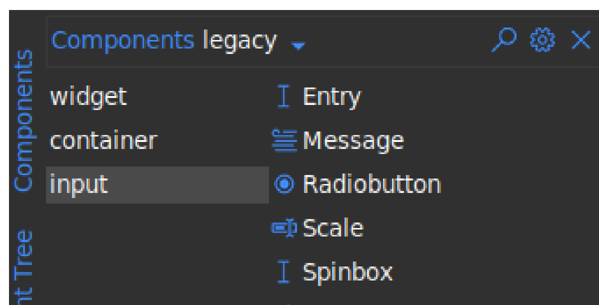
```
formation-studio
```

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.

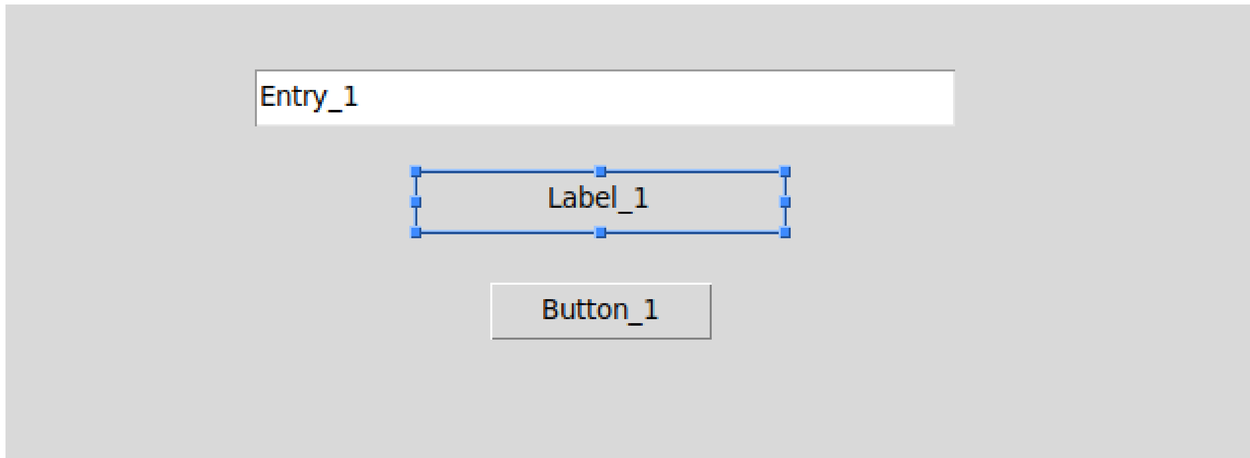


Now select `input` on the vertical tab to access a new set of widgets and drag one `Entry` to the design pad.

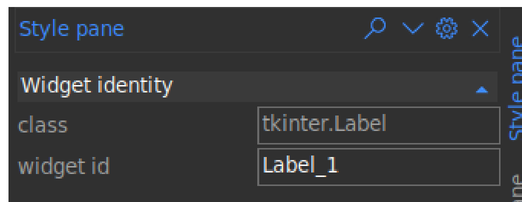


Arrange them as you please and you should have something as shown below.

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**

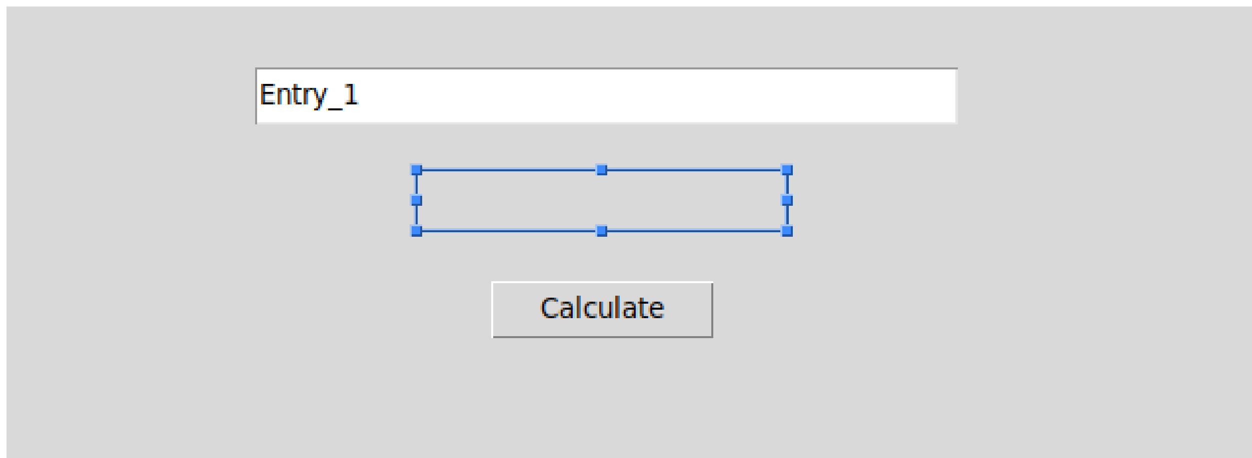
Double click the `Label` and the `Button` and change the texts as shown below. Leave the label blank since we will display the results of the calculations here.

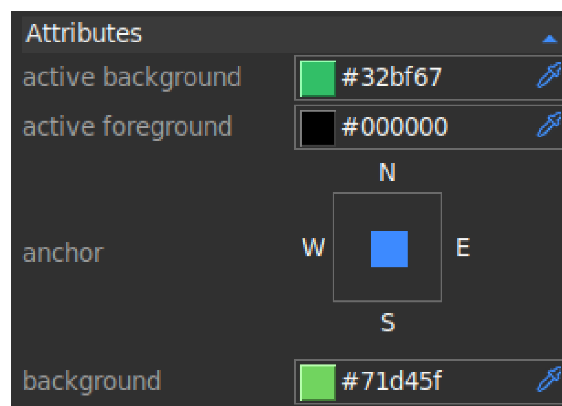
Alternatively, you can use the style pane to set the text attribute along with other style options you deem fit. The style pane as a whole is divided into 3 main parts

- **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

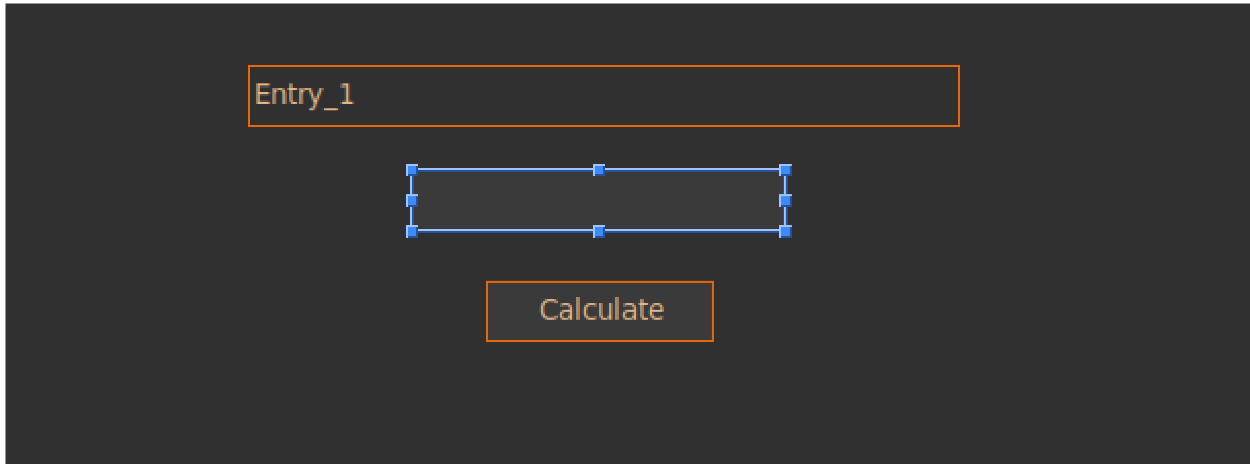
Play around with the styles to achieve your desired look. Try changing the colors and fonts. The design can look however you want.

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

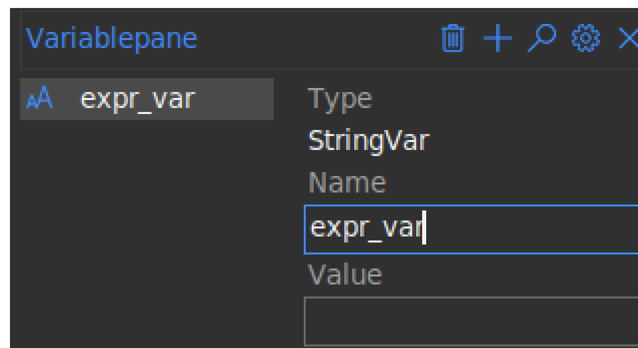

Attributes
active background
#32bf67
active foreground
#000000
anchor
N
W
E
S
background
#71d45f

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

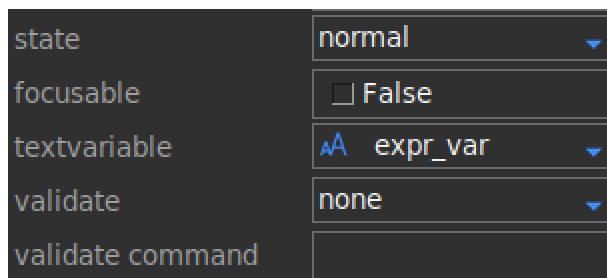


8.3 Connecting Variables

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.

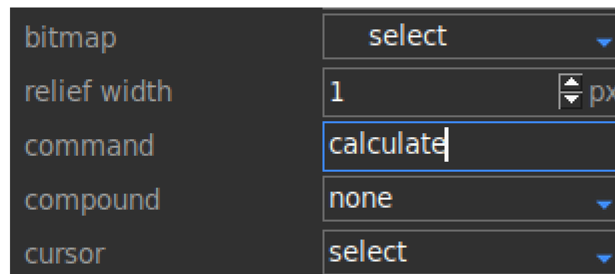
8.4 Connecting Commands

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.

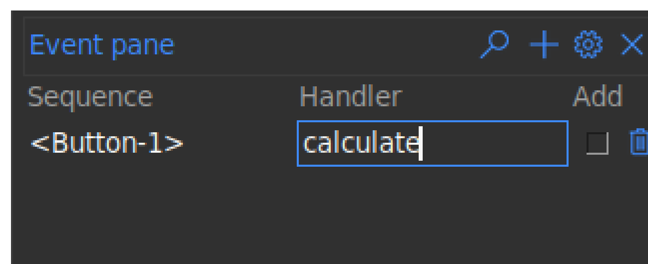
8.4.1 1. Using the command option

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

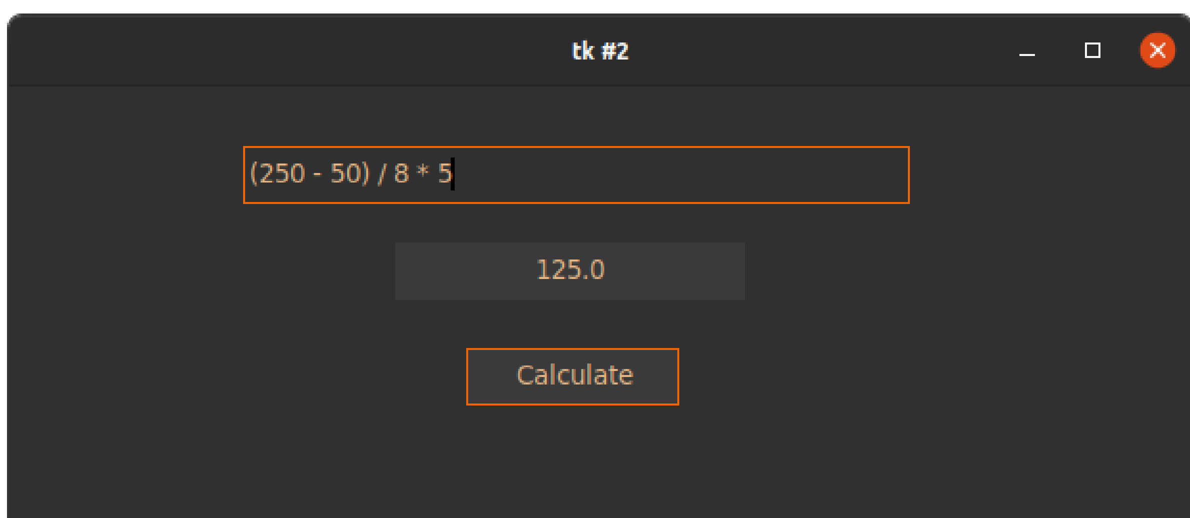
```
app.mainloop()
```

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

Contains classes that load formation design files and generate user interfaces

class formation.loader.**AppBuilder**(*app=None*, **args*, ***kwargs*)

Subclass of *formation.loader.Builder* that allow opening of designs files without toplevel root widget. It automatically creates a toplevel window and adapts its size to fit the design perfectly. The underlying toplevel window can be accesses as _app. The private accessor underscore is to free as much as of the builder namespace to your user defined names and prevent possible issues

Parameters

- **app** – optional custom external toplevel to use, if unspecified a toplevel window is created for you
- **args** – Additional arguments to be passed to underlying toplevel window
- **kwargs** – Keyword arguments to be passed to underlying toplevel window. The arguments allowed are:
  - path: Path to the design file to be loaded
  - string: serialized design as string to be loaded
  - node: Node node to be loaded

  These arguments are mutually exclusive since design can be loaded from only one format at time.

```
1 # import the formation library which loads the design for you
2 from formation import AppBuilder
3
4 # hello.xml can be any design file created in formation studio
5 app = AppBuilder(path="hello.xml")
6
7 app.mainloop()
```

mainloop(*n: int = 0*)

Start the mainloop for the underlying toplevel window

Parameters **n** –

Returns

class formation.loader.**Builder**(*parent*, ***kwargs*)

Load design file into a GUI with all components accessible as attributes To access a widget use its name as set in the designer

Parameters

- **parent** – The parent window where the design widgets are to be loaded
- **kwargs** – Options used in loading * **path**: Path to file of supported format from which to load the design * **string**: String of supported format to be used in loading the design * **node**: an instance of `Node` from which to load the design directly * **format**: an instance of `BaseFormat` to be used in loading the string contents provided by the **string** option

---

Note: if the **string** option is used, not providing the **format** option will raise a :class:ValueError

---

connect_callbacks(*object_or_dict*)

Connect bindings and callbacks to user defined functions and methods. It connects commands added through the various command config options such `command`, `yscrollcommand`, `xscrollcommand` among others. Callbacks added through bindings are connected as well. Below are possible ways to connect global functions.

```
...

def on_click(event):
    print("button clicked")

def on_keypress(event):
    print("key pressed")

# method 1 ---------------------------

build = Builder(parent, path="my_design.xml")
build.connect_callbacks({
    "on_click": on_click,
    "on_keypress: on_keypress,
})

# method 2 ---------------------------

build = Builder(path="my_design.xml")
build.connect_callbacks(globals())

...
```

To connect to object methods instead the code can be as shown below

```
...

class App(tkinter.Tk):

    def __init__(self):
        self.build = Builder(self, path="my_design.xml")
        self.build.connect_callbacks(self)

```

(continues on next page)

(continued from previous page)

```
    def on_click(self, event):
        print("button clicked")

    def on_keypress(self, event):
        print("key pressed")

app = App()
app.mainloop()

...
```

Parameters `object_or_dict` – A dictionary containing function mappings or an object defining all the callback methods. The callback names have to exactly match what was entered in the studio.

`load_node`(*node: formation.formats._base.Node*)
Load the builder from a `Node` instance

Parameters `node` – `Node` to be loaded

Returns root widget

`load_path`(*path*)
Load design file

Parameters `path` – Path to design file to be loaded

Returns root widget

`load_string`(*content_string*, *format_*)
Load the builder from a string

Parameters

- `content_string` – string containing serialized design to be loaded
- `format` – the format class sub-classing `BaseFormat` to be used

Returns root widget

`property path`
Get absolute path to loaded design file if available

Returns path to currently loaded design file if builder was loaded from path else `None`

9.2 Formation utilities

Documentation for useful utilities provided by formation

`class formation.utils.CustomPropertyMixin`
Simplifies addition of custom properties to tkinter widgets. By using a simple definition of the properties in the `prop_info` class attribute, it allows you to access and modify custom properties as though they were built-in tkinter properties. To define a property, you need to specify the following attributes

- name : The name of the custom attribute. Should be the same as the key of the property in the `prop_info` dictionary
- default: The default value of the custom attribute

- setter: name of the method used to set the property as a string. the setter should accept one argument which is the value being set
- getter: name of the attribute used to store the property.

Note: It is upto you to store the custom attribute values and provide them on demand in the getter. Ensure the getter contains the most upto-date value.

```
class MyCustomWidget(CustomPropertyMixin, tkinter.Frame):

    prop_info = {
        "background_image": {
            "name": "background_image",
            "default": None,
            "setter": "set_background_img",
            "getter": "bg_img"
        },
        "title_text": {
            "name": "title_text",
            "default": "",
            "setter": "set_title",
            "getter": "title"
        },
    }

    def __init__(master=None, **kw):
        # do not pass kw directly to super call
        super().__init__(master)
        self.title = ""
        self.bg_img = None

        # should be the last thing
        self.configure(kw)

    def set_background_img(self, value):
        self.bg_img = value
        # extra logic to apply attribute

    def set_title(self, value):
        self.title = value
        # extra logic to apply attribute
```

The custom widget can then be used as shown below:

```
>>> widget = MyCustomWidget(parent)
>>> widget.config(title_text="my title", bg="red", background_image=img)
>>> print(widget["title_text"])
my_title
>>> print(widget.cget("title_text"))
my_title
>>> widget["title_text"] = "new title"
>>> print(widget["title_text"])
new_title
```

CHAPTER

TEN

HOVERSET

10.1 Widget Catalogue

This is the foundation of all GUI components used in hoverset based project. These widget classes are used in the Formation studio and have special features such as tkinter styles loaded from css files and available for use anywhere inside them. All gui manifestation should strictly use hoverset widget set for easy maintenance in the future.

class `hoverset.ui.widgets.`**`Application`**(**args*, ***kwargs*)

The main toplevel widget for hoverset widgets. All hoverset widgets must be children or descendants of an *`hoverset.ui.widgets.Application`* object.

`bind_all`(*sequence=None*, *func=None*, *add='+'*)

Total override of the tkinter bind_all method allowing events to be bounds to all the children of a widget and not the entire application. This is useful for compound widgets which need to behave as a single entity

Parameters

- **`sequence`** – Event sequence to be bound
- **`func`** – Callback function
- **`add`** – specifies whether func will be called additionally to the other bound function or whether it will replace the previous function entirely.

Returns identifier of the bound function allowing it to be unbound later

`load_styles`(*theme*)

Accepts a path to a cascading style sheet containing the styles used by the widgets. The style dependency is loaded here

Parameters **`theme`** – name of the theme to be loaded with or without the .css extension

Returns A `hoverset.ui.styles.StyleDelegator` object

property style

Get the currently loaded css :return: a `hoverset.ui.styles.StyleDelegator`

`unbind_all`(*sequence*, *func_id=None*)

Unbind sequence from immediate children

Parameters

- **`sequence`** – sequence to be unbound
- **`func_id`** – function id if any to unbind a specific callback

Returns

class `hoverset.ui.widgets.`**`Button`**(*master=None*, ***cnf*)

Completely custom Hoverset widget built on top of *`hoverset.ui.widgets.Frame`*

config(***cnf*)

Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

class hoverset.ui.widgets.**Canvas**(*master=None*, ***kwargs*)

Hoverset wrapper for `tkinter.Canvas`

class hoverset.ui.widgets.**Checkbutton**(*master=None*, ***cnf*)

Hoverset wrapper for `tkinter.ttk.Checkbutton`

get()

Get the selection state

Returns `True` if selected else `False`

set(*value*)

Set the boolean value directly

Parameters **value** – boolean value to be set

class hoverset.ui.widgets.**ComboBox**(*master=None*, ***cnf*)

config(***cnf*)

Configure resources of a widget.

The values for resources are specified as keyword arguments. To get an overview about the allowed keyword arguments call the method keys.

class hoverset.ui.widgets.**CompoundList**(*master=None*, ***cnf*)

ListBox widget allowing for more flexibility with custom items extending `CompoundList.BaseItem`. Here is an example:

```
from hoverset.ui.widgets import CompoundList, Application, Label

app = Application()

my_list = CompoundList(app)

class CustomItem(CompoundList.BaseItem):
    # Custom class to display two fields in a single item

    def render(self):
        occupation, name = self.value
        Label(self, text=f"Occupation: {occupation}").pack(side="top")
        Label(self, text=f"Name: {name}").pack(side="top")

my_list.set_item_class(CustomItem)
my_list.set_values([["Engineer", "John"], ["Professor", "Sir Isaac"]])
my_list.pack()

app.mainloop()
```

class **BaseItem**(*master*, *value*, *index*, *isolated=False*)

Base class for all custom list items

clone_to(*parent*)

Create a copy of the item for positioning in a new parent

Parameters **`parent`** – New intended parent

Returns the new item clone

`deselect()`
Marks item as deselected and applies the required styles and configuration to make it return to its normal state

`get()`
Get the value represented by the item

Returns Value represented by the item

`on_hover(*_)`
Applies styles and config required when item is hovered

`on_hover_ended(*_)`
Revert the item config when no longer under hover

`render()`
Create the custom section of a custom item. Override this method and add new widgets to the item. The default rendering is a label containing the value of the item

`select(*_)`
Marks item as selected and applies the required styles and configuration to make it appear selected such as the color

`select_self`(*event=None*, *_)
Set the item as selected in its parent list

Parameters **`event`** – event causing the selection. Default is `None`

`property value`
The value the item is supposed to display. Can be any object depending on what is set through *`CompoundList.set_values`*

Returns Value represented by item

`add_values`(*values*)
Append new values to the list

Parameters **`values`** – an iterable containing items to be added

Returns

`get()`
Get currently selected item(s)

Note: This does not return the underlying value but the rendered item currently selected which is a *`CompoundList.BaseItem`* object. To obtain the value use its `value` property or `get` method

---

Returns selected item if mode is not set to MULTI_MODE otherwise a list of all selected items. If no item is selected `None` is returned

`get_class()`
Get the item class currently in use

Returns current item class

`get_mode()`
Get currently set mode

`on_change`(*func*, **args*, ***kwargs*)
Set a callback function to be called on selection change

Parameters

- **func** – callback function
- **args** – extra positional arguments to be passed to callback in addition to the selected item
- **kwargs** – keyword arguments to be passed to callback function

select(*index*, *event=None*)
Select item at given index

Parameters

- **index** – index to be selected
- **event** – event generating the selection if any

set_item_class(*cls*)
Set the class used to render the list items in the case of custom items.

Parameters **cls** – A a subclass of *CompoundList.BaseItem*

set_mode(*mode*)
Set the mode of selection

Parameters **mode** – mode value which can be one of the following

- `CompoundList.SINGLE_MODE`: allows selection of one item at a time
- `CompoundList.MULTI_MODE`: allows selection of multiple items by holding down the control key
- `CompoundList.BROWSE_MODE`: allows selection of one item at a time. Selection will follow the currently hovered item

set_values(*values*)
Set the values to be displayed by the list box. Clears current values. to add new values use `add_values`

Parameters **values** – an iterable containing the item values to be displayed

class hoverset.ui.widgets.**ContextMenuMixin**
Adds context menu functionality to a widget

static add_context_menu(*menu*, *widget*)
Setup context menu for other widgets not extending this mixin

Parameters

- **menu** – menu to be set up as context
- **widget** – widget to context menu on

make_menu(*parent=None*, *style: hoverset.ui.styles.StyleDelegator = None*, *dynamic=True*, ***cnf*)
Create a dynamic menu object under a tkinter widget parent

Parameters

- **dynamic** – suppress dynamic behaviour, useful for toplevel menubar. Default is set to true
- **style** – hoverset StyleDelegator object to allow retrieval of necessary menu theme styles
- **templates** – a tuple that may contain the following

1. a tuples of the format (type, label, icon, command, additional_config) where type is either `command, cascade, radiobutton, checkbutton` and additional_config is a dict containing menu item configuration

2. a tuple of he format ('separator', additional_config) to declare a separator. The additional_config is optional
3. a `Hoverset.ui.menu.Manipulator` object.

- **`parent`** – The parent of the menu. You will never need to set this attribute directly as it only exists for the purposes of recursion
- **`cnf`** – configuration for created menu

Returns dynamic menu

static popup(*event*, *menu*)
Show context menu at event location

Parameters
- **`event`** – event whose location is to be used
- **`menu`** – menu to be displayed

set_up_context(*templates*, ***cnf*)
Set up a context menu using the template which is a tuple containing items in the format `(type, label, icon, command, additional_configuration={})`

Parameters
- **`templates`** – menu template
- **`cnf`** – config for menu

class hoverset.ui.widgets.**DragWindow**(*master*, ***cnf*)

class hoverset.ui.widgets.**DrawOver**(*master*, ***cnf*)

destroy()
Destroy this and all descendants widgets.

class hoverset.ui.widgets.**EditableMixin**
This mixin implements all methods applicable to all widgets that allow entry of data using the keyboard. All widgets that have such functionality should ensure they extend this mixin.

clear_validator()
Remove any existing validators set by set_validator

disabled(*flag*)
Change the state of an editable widget, whether disable or enabled

Parameters **`flag`** – set to `True` to disable and `False` to enable

get()
Overrides default get method which often gives an outdated value and instead returns latest value straight from the control variable

Returns current value of editable widget, type depends on the control variable type

on_change(*callback*, **args*, ***kwargs*)
Set the callback when data in the input widget is changed either explicitly or implicitly.

Parameters **`callback`** –

Returns

on_entry(*callback*, **args*, ***kwargs*)
Set the callback when data in the input widget is changed explicitly i.e when the user actually types values into the input widget.

Parameters **callback** –

Returns

set_validator(*validator*, **args*, ***kwargs*) → None
Allows addition of realtime validation of data entered by the user to input widgets. This validation is carried out at the lowest level before the user interface even displays the value in the widget allowing invalid data to be blocked before it is ever displayed.

Parameters **validator** – The validation method that accepts one argument which is the string to be validated. Such functions can be found or added at hoverset.util.validators

Returns None

class hoverset.ui.widgets.**Entry**(*master=None*, ***cnf*)

class hoverset.ui.widgets.**EventMask**
Event mask values to be used to test events occurring with these states set. For instance, to check whether control button was down the following check can be performed

```
def on_event(event):
    if event.state & EventMask.CONTROL:
        print("Control button pressed")
```

Event Mask	Event status
EventMask.SHIFT	Shift key down
EventMask.CAPS_LOCK	Caps lock key down
EventMask.CONTROL	Control key down
EventMask.L_ALT	Left Alt key down
EventMask.NUM_LOCK	Num lock key down
EventMask.MOUSE_BUTTON_1	Right mouse button down
EventMask.MOUSE_BUTTON_2	Mouse wheel down
EventMask.MOUSE_BUTTON_3	Left mouse button down

class hoverset.ui.widgets.**EventWrap**(*x_root*, *y_root*, *x*, *y*)
Imitate a tkinter event object for use when handling synthetic events

x
Alias for field number 2

x_root
Alias for field number 0

y
Alias for field number 3

y_root
Alias for field number 1

class hoverset.ui.widgets.**FontStyle**(*root=None*, *font=None*, *name=None*, *exists=False*, ***options*)
Hoverset equivalent of `tkinter.font.Font` with additional functionality

static **families**(*root=None*, *displayof=None*)
Get font families as tuple

`static names(root=None)`
Get names of defined fonts (as a tuple)

`static nametofont(name)`
Given the name of a tk named font, returns a Font representation.

`class hoverset.ui.widgets.Frame(master=None, **kwargs)`

`class hoverset.ui.widgets.ImageCacheMixin`
Performs automatic handling of images in tkinter widgets that use images by overriding configure methods and adding references to the images to shield them from garbage collection. It also handles animated widgets

`class hoverset.ui.widgets.Label(master=None, **kwargs)`

`class hoverset.ui.widgets.LabelFrame(master=None, **kwargs)`
Hoverset wrapper for `tkinter.LabelFrame`

`class hoverset.ui.widgets.MenuButton(master=None, **kwargs)`
Hoverset wrapper for `tkinter.Menubutton`

`class hoverset.ui.widgets.Message(master=None, **kwargs)`

`class hoverset.ui.widgets.PanedWindow(master=None, **cnf)`

`class hoverset.ui.widgets.Popup(master, pos=None, **cnf)`

`destroy()`
Destroy this and all descendants widgets.

`class hoverset.ui.widgets.PositionMixin`
Automatic positioning of popup windows, it positions windows such that the are visible from any point of the screen by providing a post method

`get_pos(widget, **kwargs)`
Get the position of a popup window anchored around a widget

Parameters

- `widget` – A tk widget to be used as an anchor point
- `kwargs` –

-side: a string value "nw", "ne", "sw", "se", "auto" representing where the dialog is to be position relative the anchor widget

-padding: an integer indicating how much space to allow between the popup and the anchor widget

-width: prospected width of the popup which can be used even before the popup is initialized by tkinter. If not provided its obtained from the popup hence the popup must have been initialized by tkinter

-height: prospected height of the popup. Same rules on `width` apply here

Returns None

`post(widget, **kwargs)`
Display a popup window anchored around a widget

Parameters

- `widget` – A tk widget to be used as an anchor point
- `kwargs` –

-side: a string value "nw", "ne", "sw", "se", "auto" representing where the dialog is to be position relative the anchor widget

-padding: an integer indicating how much space to allow between the popup and the anchor widget

-width: prospected width of the popup which can be used even before the popup is initialized by tkinter. If not provided its obtained from the popup hence the popup must have been initialized by tkinter

-height: prospected height of the popup. Same rules on `width` apply here

Returns None

`class hoverset.ui.widgets.`**`ProgressBar`**`(`*master*, ***kw*`)`

Custom progress bar for use by hoverset applications

`color``(`*color*`)`

Set the progress bar color

Parameters **`color`** – A named color or hex defined color

Raises `ValueError` if color is not a valid tk color

`get``()`

Fetch the current progress of the bar.

Returns a floating point from 0 to 1 representing current progress. If mode is set to indeterminate None is returned

`interval``(`*milliseconds*`)`

Controls the speed of the indeterminate mode of the progressbar

Parameters **`milliseconds`** – the update time in milliseconds, the smaller the faster

`mode``(`*value*`)`

Set the mode of the progressbar to determinate or indeterminate

Parameters **`value`** – constant value either ProgressBar.DETERMINATE or ProgressBar.INDETERMINATE

`set``(`*value: float*`)`

Sets the progress to a fraction value

Parameters **`value`** – A floating point value between 0 and 1 inclusive which determines the progress

`class hoverset.ui.widgets.`**`RadioButton`**`(`*master*, ***cnf*`)`

Hoverset wrapper for `tkinter.ttk.Radiobutton`

`class hoverset.ui.widgets.`**`RadioButtonGroup`**`(`*master=None*, *choices=()*, *label=''*, ***cnf*`)`

Group of *`RadioButton`* objects used to obtain a single value out of multiple options.

```
button_group = RadioButtonGroup(
    parent,
    choices=(
        ("yellow", "This is the yellow option"),
        ("red", "This is the red option"),
    ),
    label="Select an option"
)
button_group.add_choice(("blue", "This is the blue option"))
```

(continues on next page)

(continued from previous page)

```
button_group.pack(side="top")
button_group.set("red")  # selects the red option
button_group.get()  # returns red
```

add_choice(*choice*)
Add a choices to be appended at the end of the radio group as a `(value, label)` pair

Parameters **choice** – a `(value label)` pair

config_all(***cnf*)
Use this method to correctly configure all radio buttons in the group

Parameters **cnf** – config options

Returns None

get()
Get the value of the currently selected option

Returns value of the current option

set(*value*, *silent=False*)
Set the selected option

Parameters

- **value** – value to be set
- **silent** – set to `True` to trigger change event

set_choices(*choices*)
Add the choices to be displayed in the radio group as `(value, label)` pairs

Parameters **choices** – a tuple of `(value label)` pairs

set_label(*label*)
Set the group label

Parameters **label** – string to be set as label

class hoverset.ui.widgets.Scale(*master=None*, *variable=None*, ***cnf*)
Hoverset wrapper for `tkinter.ttk.Scale`

config_all(***kwargs*)
Configure all options including ttk themed options automatically

Parameters **kwargs** – config options

Returns None

get(*x=None*, *y=None*)
Gets the current value directly from the underlying variable if either x or y is not provided.

Parameters

- **y** – return value at x if provided
- **x** – return value at y if provided

Returns current scale value

set(*value*)
Set scale value. Overrides default behaviour and sets the value directly to the underlying variable

Parameters **value** – Value to be set

class hoverset.ui.widgets.**Screen**(*window: tkinter.Tk*)

What can comfortably be considered a tkinter fashion window for the root window (Tk) This allows calculations for centering the window possible with reference to the whole screen

class hoverset.ui.widgets.**ScrollableInterface**

Interface that allows widgets to be managed by the _MouseWheelDispatcherMixin which handles mousewheel events which may be tricky to handle at the widget level.

class hoverset.ui.widgets.**ScrolledFrame**(*master=None*, ***cnf*)

config_all(***cnf*)

A way to config all the children of a widget. Especially useful for compound widgets where styles need to be applied uniformly or following a custom approach to all contained child widgets. Override this method to customize its behaviour to suit your widget. It defaults to the normal config

Parameters

- **cnf** – `dict` containing configuration
- **kwargs** – configurations as keyword arguments

Returns None

set_scrollbars(*flag*)

Parameters **flag** – set to tkinter.X to enable horizontal scrollbar, tkinter.Y to enable vertical scrollbar, tkinter.BOTH to enable both scrollbars and None to disable all scrollbars. The default is tkinter.Y for the vertical scrollbar.

Returns None

class hoverset.ui.widgets.**SpinBox**(*master=None*, ***cnf*)

get()

Overrides default get method which often gives an outdated value and instead returns latest value straight from the control variable

Returns current value of editable widget, type depends on the control variable type

class hoverset.ui.widgets.**Spinner**(*master=None*, ***_*)

Combobox widget allowing easy customization of choice items

config_all(***cnf*)

A way to config all the children of a widget. Especially useful for compound widgets where styles need to be applied uniformly or following a custom approach to all contained child widgets. Override this method to customize its behaviour to suit your widget. It defaults to the normal config

Parameters

- **cnf** – `dict` containing configuration
- **kwargs** – configurations as keyword arguments

Returns None

disabled(*flag*)

Set the state of a widget to disabled .Override this method for compound widgets to obtain the expected behaviour as the state is by default only applied to the containing/parent widget.

Parameters **flag** – True or False

Returns

`class hoverset.ui.widgets.TabView(master, **cnf)`

`class Tab(master, **cnf)`

`config_tab(**cnf)`

`on_deselect()`

`on_drag(event)`
Called when widget is dragged. This is called on each motion event so it's best to keep computation in this function at a minimum

`on_drag_end(event)`
Called when widget is dropped and dragging ends

`on_drag_start(*args)`
Called whe the widget is first dragged

`on_select()`

`render_drag(window)`
Override this method to create and position widgets on the drag shadow window (The object displayed as the widget is dragged around). Create your custom widget hierarchy and position it in window.

Parameters **window** – The drag window provided by the drag manager that should be used as the widget master

Returns None

`class hoverset.ui.widgets.Text(master, **cnf)`

`class hoverset.ui.widgets.ToggleButton(master=None, **cnf)`
Hoverset button allowing toggling

`class hoverset.ui.widgets.ToolWindow(master, **cnf)`

`show()`
The window is initialized as invisible to allow you to set it up first. Call this method to make it visible

`class hoverset.ui.widgets.Tree`
Tree Abstraction for management of tree_views and similar tree-like widgets in hoverset. To use this class, subclass it and implement the `Tree.get_body()` method to specify the container widget.

Note: Remember to call `Tree.initialize_tree()` in your constructor before performing any operations on the tree.

`class Node(tree, **config)`

`BLANK = None`

`COLLAPSED_ICON = None`

`EXPANDED_ICON = None`

`add(node)`

`add_as_node(**options)`
Adds a node to the tree view.

Parameters **options** – Options used in creating the node like name, icon e.t.c. depending on the Node

Returns The created Node

`bind_all`(*sequence=None*, *func=None*, *add=None*)
Total override of the tkinter bind_all method allowing events to be bounds to all the children of a widget and not the entire application. This is useful for compound widgets which need to behave as a single entity

Parameters

- **sequence** – Event sequence to be bound
- **func** – Callback function
- **add** – specifies whether func will be called additionally to the other bound function or whether it will replace the previous function entirely.

Returns identifier of the bound function allowing it to be unbound later

`clear`()

`collapse`()

`collapse_all`()

property depth

`deselect`(*_)

`expand`()

`expand_all`()

`index`()

`insert`(*index=None*, **nodes*)
Insert all child nodes passed into parent node starting from the given index

Parameters

- **index** – int representing the index from which to insert
- **nodes** – Child nodes to be inserted

`insert_after`(**nodes*)
Insert the nodes immediately after this node in the same parent

Parameters nodes – List of nodes to be inserted

`insert_before`(**nodes*)
Insert the nodes immediately before this node in the same parent

Parameters nodes – List of nodes to be inserted

`is_descendant`(*node*)

property name

`remove`(*node=None*)
Remove the node from node's child nodes. If node is not provided the the node removes itself from its parent

Parameters node – Node to be removed (optional)

Returns None

`search`(*query*)

`select`(*event=None*, *silently=False*)

`toggle`(*_)
Toggle between the expanded and collapsed state

`toggle_select`(*event*)

class Strip(*master=None*, ***config*)
An interface for event binding to tree view items

`deselect()`

`select()`

`add(node)`
Add an already created node to the tree view. Use add_as_node instead to avoid tkinter parent issues.

Parameters `node` – The child Node to be added to the Node

`add_as_node(**options)` → *hoverset.ui.widgets.Tree.Node*
Adds a base node to the Tree. The node will belong to a subclass' Node definition if any.

Parameters `options` – Options used in creating the node like name, icon e.t.c.

Returns The created Node

`allow_multi_select(flag)`
Allow or disallow multiple widgets to be selected

Parameters `flag` – Set to True to allow multiple items to be selected by the tree view and false to disable selection of multiple items.

`clear_selection()`
Deselect all currently selected nodes

`collapse_all()`
Collapse all nodes and sub-nodes so that their sub-node are not displayed

`expand_all()`
Expand all nodes and sub-nodes so that their sub-nodes are displayed

`get()`
Get the currently selected node if multi select is set to False and a list of all selected items if multi select is set to True. Returns None if no item is selected.

Returns Selected widget or None if no widget is selected

`abstract get_body()`
Return the tkinter container like a Frame where the nodes are packed. This method must be implemented and must return a valid container.

`initialize_tree()`
Initialize tree properties. Make sure its called before any operations are performed on the tree.

`select(n, silently=False)`
Select a node :param n and deselect all other selected nodes

Parameters

- `silently` – Flag set to true to prevent firing on change event and vice versa. Default is false
- `n` – Node to be selected

`selected_count()` → int
Return the total number of items currently selected usually 1 if multi-select is disabled.

Returns total number of items selected

`class hoverset.ui.widgets.TreeView(master=None, **config)`
Custom tree view implementation that is way more flexible for hoverset applications. Can be easily modified and works well with hoverset themes.

`get_body()`
Return the tkinter container like a Frame where the nodes are packed. This method must be implemented and must return a valid container.

`class hoverset.ui.widgets.Widget`
Base class for all hoverset widgets implementing all common methods.

`absolute_bounds()`
Get the position of the widget on the screen

Returns a tuple containing the bounding box of the widget (x1, y2, x2, y2)

`accept_context(context)`
This method is called when a drag drop operation is completed to allow the dropped object to be handled

Parameters `context` – Object being dropped at the widget

`property allow_drag`
Determines whether widgets can be dragged in-case of a drag drop event

`classmethod ancestor_first(start_from, class_: type, ignore=None)`
Gets the first widget belonging to *class_* starting from *start_from*. This widget may be the top widget or it's parents and grandparents deep down the hierarchy. Useful when you want to access a widget's first ancestor of a given type down the stacking order

Parameters

- `start_from` – widget whose ancestor is to be determined
- `class` – the class of the widget we are interested in
- `ignore` – widget to be ignored if any

Returns the first widget belonging to *class_*, if no widget is found None is returned

`bind_all(sequence=None, func=None, add=None)`
Total override of the tkinter bind_all method allowing events to be bounds to all the children of a widget and not the entire application. This is useful for compound widgets which need to behave as a single entity

Parameters

- `sequence` – Event sequence to be bound
- `func` – Callback function
- `add` – specifies whether func will be called additionally to the other bound function or whether it will replace the previous function entirely.

Returns identifier of the bound function allowing it to be unbound later

`clone(parent)`
Generates a clone of the current widget for the given parent. Override this method in a custom widget to provide a custom implementation for cloning

Parameters `parent` – A tk widget which will be clones parent

Returns A clone of the current widget

`static clone_to(parent, widget)`
Clone a tkinter widget to a different parent. Tkinter widget parents cannot be changed directly. This method performs recursive cloning of widget hierarchies. For cloning of custom tkinter widgets it is advisable to use clone method instead to specify your clone procedure

Parameters

- `parent` – The new parent for cloned widget

- **widget** – The widget to be cloned

Returns cloned widget

config_all(*cnf=None*, ***kwargs*)
A way to config all the children of a widget. Especially useful for compound widgets where styles need to be applied uniformly or following a custom approach to all contained child widgets. Override this method to customize its behaviour to suit your widget. It defaults to the normal config

Parameters

- **cnf** – `dict` containing configuration
- **kwargs** – configurations as keyword arguments

Returns None

static **containing**(*x*, *y*, *widget*)
A safer alternative for tk winfo_containing that does extra checks just in case the widget at the target position is not recognized or managed by our tk instance

Parameters

- **x** – x coordinate
- **y** – y coordinate
- **widget** – widget to be checked.

Returns name of widget at position

static **copy_config**(*from_*, *to*)
Copy styles and configuration from one widget to another

Parameters

- **from** – Widget whose configuration is to be copied
- **to** – Widget receiving the copied configurations

Returns None

disabled(*flag: bool*) → None
Set the state of a widget to disabled .Override this method for compound widgets to obtain the expected behaviour as the state is by default only applied to the containing/parent widget.

Parameters **flag** – True or False

Returns

drag_start_pos(*event*)
Override and return the preferred drag start position as a tuple (x, y). Default is the current widget position

classmethod **event_first**(*event*, *widget*, *class_: type*, *ignore=None*)
Gets the first widget belonging to *class_* at the event position. This widget may be the top widget or it's parents and grandparents deep down the hierarchy. Useful when you want to ignore widgets and cascade the event to a specific lower level widget.

Parameters

- **event** – a tk event object containing the position data
- **widget** – any widget preferably the toplevel widget
- **class** – the class of the widget we are interested in
- **ignore** – widget to be ignored if any

Returns the first widget belonging to *class_*, if no widget is found None is returned

`static event_in(event, widget)`
Check whether event has occurred within a widget

Parameters

- **event** – event object containing position data
- **widget** – the widget to be checked

Returns True if event occurred in within widget else False

`property height: int`
Wrapper property of the tk Misc class w.winfo_height() method for quick access to widget height property in pixels :return: int

`on_drag(event)`
Called when widget is dragged. This is called on each motion event so it's best to keep computation in this function at a minimum

`on_drag_end(event)`
Called when widget is dropped and dragging ends

`on_drag_start(*args)`
Called whe the widget is first dragged

`render_drag(window)`
Override this method to create and position widgets on the drag shadow window (The object displayed as the widget is dragged around). Create your custom widget hierarchy and position it in window.

Parameters **window** – The drag window provided by the drag manager that should be used as the widget master

Returns None

`render_tooltip(window)`
Create a custom tooltip body by overriding this method. The default rendering displays a simple Label with the tooltip text

Parameters **window** – The tooltip window instance to be used as parent for the custom elements

Returns None

`setup(_=None)`
It performs the necessary dependency injection and event bindings and set up.

`tooltip(text, delay=1500)`
Set the tooltip text for a widget

Parameters

- **text** – Tooltip text to be displayed
- **delay** – Amount of time in milliseconds it takes for the tooltip to appear

Returns None

`property width: int`
Wrapper property of the tk Misc class w.winfo_width() method for quick access to widget width property in pixels

Returns width of widget

`exception hoverset.ui.widgets.WidgetError`
Extra errors thrown by hoverset widgets

class hoverset.ui.widgets.**Window**(*master=None*, *content=None*, **args*, ***kwargs*)

class hoverset.ui.widgets.**WindowMixin**

hoverset.ui.widgets.**chain**(*func*)

Decorator function that allows class methods to be chained by implicitly returning the object. Any method decorated with this function returns its object.

Parameters **func** –

Returns

hoverset.ui.widgets.**clean_styles**(*widget*, *styles*) → dict

Ensures safety while passing styles to tkinter objects. Normally tkinter objects raise errors for declaring styles that are not allowed for a given widget. This function takes in the styles dictionary and removes invalid styles for the particular widget returning the cleaned styles dictionary. As a bonus, duplicate definitions are overwritten.

Parameters

- **widget** –
- **styles** –

Returns dict cleaned_styles

hoverset.ui.widgets.**set_ttk_style**(*widget*, *cnf=None*, ***styles*) → None

Allows you set styles for ttk widgets just like conventional tkinter widgets. It bypasses the need to work with ttk styles. It is important however to note that unsupported styles will be silently ignored!

Parameters

- **widget** – A ttk widget
- **styles** – keyword arguments representing conventional tkinter style
- **cnf** – Dictionary of styles to applied

Returns None

hoverset.ui.widgets.**suppress_change**(*func*)

Wraps a method to prevent it from emitting an on change event. Works with the hoverset architecture where objects contain a _on_change attribute for the callback fired on change

hoverset.ui.widgets.**system_fonts**()

A list of all font on the current system

Returns list of font names

10.2 Dialog Windows

Common dialogs customised for hoverset platform

class hoverset.ui.dialogs.**MessageDialog**(*master*, *render_routine=None*, ***kw*)

Main class for creation of hoverset themed alert dialogs. It has the various initialization methods for the common dialogs. The supported forms/types are shown below:

- OKAY_CANCEL *MessageDialog.ask_okay_cancel()*
- YES_NO *MessageDialog.ask_question()*
- RETRY_CANCEL *MessageDialog.ask_retry_cancel()*
- SHOW_PROGRESS *MessageDialog.show_progress()*

- SHOW_WARNING *MessageDialog.show_warning()*
- SHOW_ERROR *MessageDialog.show_error()*
- SHOW_INFO *MessageDialog.show_info()*

There are three icons used in the dialogs:

- `MessageDialog.ICON_ERROR`
- `MessageDialog.ICON_INFO`
- `MessageDialog.ICON_WARNING`

Some dialogs return values while others just notify. Below illustrates using the various dialogs

```
# assuming we have a hoverset Application object app
# program will wait for value to be obtained

val = MessageDialog.ask_retry_cancel(
    title="ask_okay",
    message="This is an ask-okay-cancel message",
    parent=app
)

# do whatever you need with the value, in this case let's just print the response

if val == True:
    print("retry")
elif val == False:
    print("cancel")
elif val is None:
    print("no value selected")

# show dialogs don't need to return values
MessageDialog.show_error(
    title="Error",
    message="This is an error message",
    parent=app
)

val = MessageDialog.builder(
    # define the buttons
    {"text": "Continue", "value": "continue", "focus": True},
    {"text": "Pause", "value": "pause"},
    {"text": "Cancel", "value": False},
    wait=True,
    title="Builder",
    message="We just built this dialog from scratch",
    parent=app,
    icon="flame"
)
print(val)
# prints 'continue', 'pause' or False depending on the value of button clicked
```

classmethod ask(*form*, ***kw*)

General method for creation common dialogs. You do not have to use this method directly since there are specialized methods as shown below.

instead of	use this
MessageDialog.ask(MessageDialog.OKAY_CANCEL, ...)	MessageDialog.ask_okay_cancel(...)
MessageDialog.ask(MessageDialog.RETRY_CANCEL, ...)	MessageDialog.ask_retry_cancel(...)
MessageDialog.ask(MessageDialog.SHOW_INFO, ...)	MessageDialog.show_info(...)
MessageDialog.ask(MessageDialog.SHOW_ERROR, ...)	MessageDialog.show_error(...)
MessageDialog.ask(MessageDialog.SHOW_PROGRESS, ...)	MessageDialog.show_progress(...)
MessageDialog.ask(MessageDialog.SHOW_WARNING, ...)	MessageDialog.show_warning(...)
MessageDialog.ask(MessageDialog.YES_NO, ...)	MessageDialog.ask_question(...)

Parameters

- **form** – The type of dialog to be created as defined in *forms* above
- **kw** – The keywords arguments included. These are the common arguments:
  - **parent**: (Required) A hoverset based toplevel widget such as `hoverset.ui.widgets.Application`
  - **title**: Title to be used for the dialog window
  - **message**: Message to be displayed in the alert dialog
  - **icon**: Icon to be displayed in the dialog. Should be one of these *icons*.

> **Warning:** The `parent` argument is mandatory and should always be provided. Absence of the parent argument will result in missing theme style definitions and cause errors.

Returns A value or `None` depending on the dialog type. See specialized method for mor details

classmethod ask_okay_cancel(***kw*)

Show a dialog windows with two buttons: `okay` and `cancel`

Parameters **kw** – Keyword arguments as defined in *common_args*.

Returns Returns `True` if "okay" is selected, `False` if "cancel" is selected or `None` if no choice is selected

classmethod ask_question(***kw*)

Show a dialog window with two button: `yes` and `no`

Parameters **kw** – Keyword arguments as defined in *common_args*.

Returns Returns `True` if "yes" is selected, `False` if "no" is selected or `None` if no choice is selected

classmethod ask_retry_cancel(***kw*)

Show a dialog window with two buttons: `retry` and `cancel`

Parameters **kw** – Keyword arguments as defined in *common_args*.

Returns Returns `True` if "retry" is selected, `False` if "cancel" is selected or `None` if no choice is selected

`classmethod builder(*buttons, **kw)`

Create custom dialogs with custom buttons, icons and return values. An example of the use of a builder has been provided at the beginning of this page.

Parameters

- **buttons** – A tuple containing a dictionary defining the custom buttons with the following keys
  - **text**: Text to be displayed in the button
  - **value**: Value returned when button is clicked
  - **focus**: Whether to focus on button when dialog is displayed. Only a single button should have `focus` set to `True`
- **kw** – config options for the builder:
  - **parent**: A hoverset based toplevel widget such as `hoverset.ui.widgets.Application`
  - **title**: Title to be used for the dialog window
  - **message**: Message to be displayed in the alert dialog
  - **icon**: Icon to be displayed in the dialog. Should be one of these *icons*.
  - **wait**: Set to `True` to suspend the program and wait a value to be returned or `False` to just continue program execution without waiting for a value. Useful when you just need to display a message

Returns A custom value depending on the custom button clicked. If `wait = False` no value is returned. If no value is selected by the user `None` is returned

`classmethod show_error(**kw)`

Show an error message with an error icon.

Parameters **kw** – Keyword arguments as defined in *common_args*.

Returns `None`

`classmethod show_info(**kw)`

Show an info message with an info icon.

Parameters **kw** – Keyword arguments as defined in *common_args*.

Returns `None`

`classmethod show_progress(**kw)`

Parameters **kw** – Config options for the progress dialog

- **parent**: A hoverset based toplevel widget such as `hoverset.ui.widgets.Application`
- **title**: Title to be used for the dialog window
- **message**: Message to be displayed in the alert dialog
- **icon**: Icon to be displayed in the dialog. Should be one of these *icons*.
- **mode**: One of the two modes
  - `MessageDialog.DETERMINATE`

- MessageDialog.INDETERMINATE
- **color**: Color to be used for the progressbar
- **interval**: The update interval in MessageDialog.INDETERMINATE in milliseconds. The smaller the interval the faster the animation.

Returns The dialog window. The underlying progressbar can then be accessed through the property MessageDialog.progress. The progressbar is a *hoverset.ui.widgets.ProgressBar* object and can be updated as required

classmethod **show_warning**(***kw*)
Show an warning message with a warning icon.

Parameters **kw** – Keyword arguments as defined in *common_args*.

Returns None

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

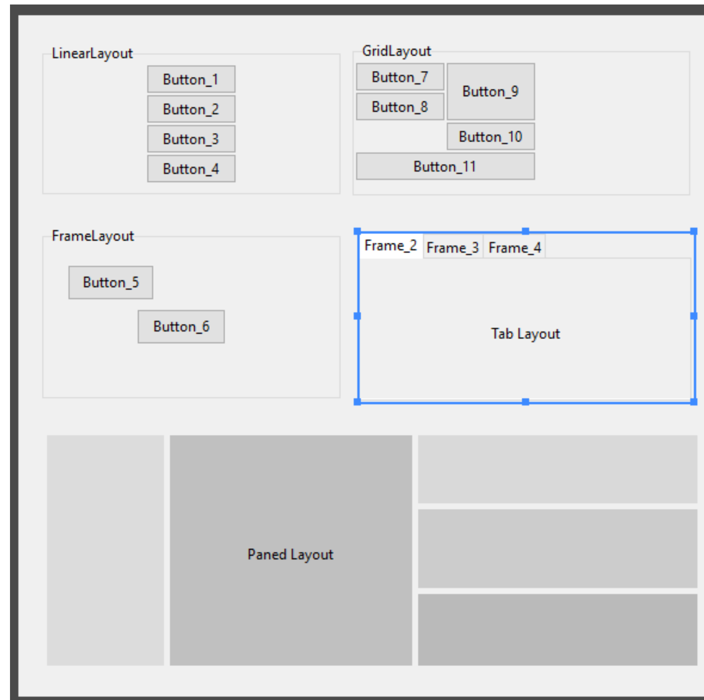
11.1.1 Introduction

- **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

Helper geometry functionality. The following terms will be used:

1. `bound`: refers to a tuple with coordinates of top left and bottom right corners of a screen section i.e. `(x1, y1, x2, y2)`. It may be shortened to `bd`
2. `dimension`: used to refer to top left corner and width/height of a screen section i.e. `(x, y, width, height)`
3. `position` is used to refer to a single point on screen i.e. `(x, y)`.

The following modifiers may be used together with the terms described above:

- **absolute**: Relative to the entire screen
- **relative**: Relative to a widget or toplevel window

`studio.ui.geometry.`**`absolute_bounds`**`(`*widget*`)`
Get bounds of widget relative to the whole screen

Parameters **`widget`** – widget whose bounds are to be determined

Returns absolute bound tuple `(x1, y1, x2, y2)`

`studio.ui.geometry.`**`absolute_position`**`(`*widget*`)`
Get dimension of widget relative to the whole screen

Parameters **`widget`** – widget whose bounds are to be determined

Returns absolute dimension tuple `(x, y, width, height)`

`studio.ui.geometry.`**`bounds`**`(`*widget*`)`
Get bounding box of widget relative to its parent

Parameters **`widget`** – widget whose bounds are to be determined

Returns bound tuple `(x1, y1, x2, y2)` representing position of widget within its parent

`studio.ui.geometry.`**`center`**`(`*bound*`)`
Get the coordinates of the center of a bound

Parameters **`bound`** – a bound tuple

Returns integer coordinates of center as a tuple `(x, y)`

`studio.ui.geometry.`**`compute_overlap`**`(`*bound1*, *bound2*`)`
Get the bound of the overlapping section between two bounds

Parameters

- **`bound1`** – bound where overlap is to be checked
- **`bound2`** – bound where overlap is to be checked

Returns bound tuple of overlap between `bound1` and `bound2` if there is no overlap `None` is returned

`studio.ui.geometry.`**`dimension_to_bounds`**`(`*x*, *y*, *width*, *height*`)`
Convert a positioned rectangular section into a bounding box coordinates indicating the top left and bottom right.

Parameters

- **`x`** – x position or rectangular section
- **`y`** – y position or rectangular section
- **`width`** – width of section
- **`height`** – height of section

Returns a tuple representing the top left and bottom right corners (x1, y1, x2, y2)

`studio.ui.geometry.dimensions(bound)`

Get the width and height of a bound

Parameters **bound** – a bound tuple

Returns a tuple containing the width and height of the `bound` i.e `(width, height)`

`studio.ui.geometry.is_within(bound1, bound2) → bool`

Checks whether bound2 is within bound1 i.e bound1 completely encloses bound2

Parameters

- **bound1** – A tuple, The enclosing bound
- **bound2** – A tuple, The enclosed bound

Returns `True` if `bound1` encloses `bound2` else `False`

`studio.ui.geometry.make_event_relative(event, relative_to)`

Change the event object's `x` and `y` attributes such that they are relative to `relative_to` parameter

Parameters

- **event** – tk event to be modified
- **relative_to** – The widget to which the event is to be made relative to

Returns None

`studio.ui.geometry.parse_geometry(geometry, default=None)`

Parse a tk geometry string and return a dict with the width, height, x any y extracted from it. The geometry string is usually in the form `<width>x<height>(-/+)<x>(-/+)<y>` for instance `200x200-10+40`. The dimensions part could be missing like in `+60+67` or the position part like in `200x200` or even both. If any of these parts is missing but the string is valid, the dictionary value will be set to value of `default`. If an invalid geometry string is provided `None` will be returned

Parameters

- **geometry** – Geometry string of the form `<width>x<height>(-/+)<x>(-/+)<y>`
- **default** – Default value for when a section of the geometry string is missing

Returns a dictionary `{"width": "", "height": "", "x": "", "y": ""}` if geometry string is valid otherwise `None`. The values returned are strings so you may need to cast them to number yourself

`studio.ui.geometry.relative_bounds(bd, widget)`

Convert bounds `bd` to be relative to the relative bound of `widget`

Parameters

- **bd** – bounds to be converted
- **widget** – Widget to which bounds are to be relative to

Returns relative bound tuple

`studio.ui.geometry.resolve_bounds(bd, widget)`

Convert bounds `bd` to be relative to the absolute bound of `widget`

Parameters

- **bd** – bounds to be converted
- **widget** – Widget to which bounds are to be relative to

Returns resolved bound tuple

`studio.ui.geometry.`**`resolve_position`**(*position*, *widget*)

Convert an absolute position such that it is relative to a `widget`

Parameters

- **`position`** – absolute position `(x, y)` to be resolved
- **`widget`** – widget to which position is to be made relative

Returns position `(x, y)` resolved to be relative to `widget`

`studio.ui.geometry.`**`upscale_bounds`**(*bound*, *widget*)

Convert bounds to be relative to a higher level `widget` i.e scale up

Parameters

- **`bound`** – relative bounds to be up-scaled
- **`widget`** – A higher level widget to which bounds are to be relative

Returns

CHAPTER TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

formation.loader, 31
formation.utils, 33

h

hoverset.ui.dialogs, 51
hoverset.ui.menu, 55
hoverset.ui.widgets, 35

s

studio.ui.geometry, 60

INDEX

A

absolute_bounds() (hoverset.ui.widgets.Widget method), 48
absolute_bounds() (in module studio.ui.geometry), 60
absolute_position() (in module studio.ui.geometry), 60
accept_context() (hoverset.ui.widgets.Widget method), 48
add() (hoverset.ui.widgets.Tree method), 47
add() (hoverset.ui.widgets.Tree.Node method), 45
add_as_node() (hoverset.ui.widgets.Tree method), 47
add_as_node() (hoverset.ui.widgets.Tree.Node method), 45
add_choice() (hoverset.ui.widgets.RadioButtonGroup method), 43
add_context_menu() (hoverset.ui.widgets.ContextMenuMixin static method), 38
add_values() (hoverset.ui.widgets.CompoundList method), 37
allow_drag (hoverset.ui.widgets.Widget property), 48
allow_multi_select() (hoverset.ui.widgets.Tree method), 47
ancestor_first() (hoverset.ui.widgets.Widget class method), 48
AppBuilder (class in formation.loader), 31
Application (class in hoverset.ui.widgets), 35
ask() (hoverset.ui.dialogs.MessageDialog class method), 52
ask_okay_cancel() (hoverset.ui.dialogs.MessageDialog class method), 53
ask_question() (hoverset.ui.dialogs.MessageDialog class method), 53
ask_retry_cancel() (hoverset.ui.dialogs.MessageDialog class method), 53

B

bind_all() (hoverset.ui.widgets.Application method), 35
bind_all() (hoverset.ui.widgets.Tree.Node method), 46
bind_all() (hoverset.ui.widgets.Widget method), 48
BLANK (hoverset.ui.widgets.Tree.Node attribute), 45
bounds() (in module studio.ui.geometry), 60
Builder (class in formation.loader), 31
builder() (hoverset.ui.dialogs.MessageDialog class method), 54
Button (class in hoverset.ui.widgets), 35

C

Canvas (class in hoverset.ui.widgets), 36
center() (in module studio.ui.geometry), 60
chain() (in module hoverset.ui.widgets), 51
Checkbutton (class in hoverset.ui.widgets), 36
clean_styles() (in module hoverset.ui.widgets), 51
clear() (hoverset.ui.widgets.Tree.Node method), 46
clear_selection() (hoverset.ui.widgets.Tree method), 47
clear_validator() (hoverset.ui.widgets.EditableMixin method), 39
clone() (hoverset.ui.widgets.Widget method), 48
clone_to() (hoverset.ui.widgets.CompoundList.BaseItem method), 36
clone_to() (hoverset.ui.widgets.Widget static method), 48
collapse() (hoverset.ui.widgets.Tree.Node method), 46
collapse_all() (hoverset.ui.widgets.Tree method), 47
collapse_all() (hoverset.ui.widgets.Tree.Node method), 46
COLLAPSED_ICON (hoverset.ui.widgets.Tree.Node attribute), 45
color() (hoverset.ui.widgets.ProgressBar method), 42
ComboBox (class in hoverset.ui.widgets), 36
CompoundList (class in hoverset.ui.widgets), 36
CompoundList.BaseItem (class in hoverset.ui.widgets), 36
compute_overlap() (in module studio.ui.geometry), 60
config() (hoverset.ui.widgets.Button method), 35
config() (hoverset.ui.widgets.ComboBox method), 36
config_all() (hoverset.ui.widgets.RadioButtonGroup method), 43
config_all() (hoverset.ui.widgets.Scale method), 43

config_all() (*hoverset.ui.widgets.ScrolledFrame method*), 44
config_all() (*hoverset.ui.widgets.Spinner method*), 44
config_all() (*hoverset.ui.widgets.Widget method*), 49
config_tab() (*hoverset.ui.widgets.TabView.Tab method*), 45
connect_callbacks() (*formation.loader.Builder method*), 32
containing() (*hoverset.ui.widgets.Widget static method*), 49
ContextMenuMixin (*class in hoverset.ui.widgets*), 38
copy_config() (*hoverset.ui.widgets.Widget static method*), 49
CustomPropertyMixin (*class in formation.utils*), 33

D

depth (*hoverset.ui.widgets.Tree.Node property*), 46
deselect() (*hoverset.ui.widgets.CompoundList.BaseItem method*), 37
deselect() (*hoverset.ui.widgets.Tree.Node method*), 46
deselect() (*hoverset.ui.widgets.Tree.Strip method*), 46
destroy() (*hoverset.ui.widgets.DrawOver method*), 39
destroy() (*hoverset.ui.widgets.Popup method*), 41
dimension_to_bounds() (*in module studio.ui.geometry*), 60
dimensions() (*in module studio.ui.geometry*), 61
disabled() (*hoverset.ui.widgets.EditableMixin method*), 39
disabled() (*hoverset.ui.widgets.Spinner method*), 44
disabled() (*hoverset.ui.widgets.Widget method*), 49
drag_start_pos() (*hoverset.ui.widgets.Widget method*), 49
DragWindow (*class in hoverset.ui.widgets*), 39
DrawOver (*class in hoverset.ui.widgets*), 39
dynamic_menu() (*in module hoverset.ui.menu*), 55

E

EditableMixin (*class in hoverset.ui.widgets*), 39
EnableIf (*class in hoverset.ui.menu*), 55
Entry (*class in hoverset.ui.widgets*), 40
event_first() (*hoverset.ui.widgets.Widget class method*), 49
event_in() (*hoverset.ui.widgets.Widget static method*), 50
EventMask (*class in hoverset.ui.widgets*), 40
EventWrap (*class in hoverset.ui.widgets*), 40
expand() (*hoverset.ui.widgets.Tree.Node method*), 46
expand_all() (*hoverset.ui.widgets.Tree method*), 47
expand_all() (*hoverset.ui.widgets.Tree.Node method*), 46
EXPANDED_ICON (*hoverset.ui.widgets.Tree.Node attribute*), 45

F

families() (*hoverset.ui.widgets.FontStyle static method*), 40
FontStyle (*class in hoverset.ui.widgets*), 40
formation.loader
    module, 31
formation.utils
    module, 33
Frame (*class in hoverset.ui.widgets*), 41

G

get() (*hoverset.ui.widgets.Checkbutton method*), 36
get() (*hoverset.ui.widgets.CompoundList method*), 37
get() (*hoverset.ui.widgets.CompoundList.BaseItem method*), 37
get() (*hoverset.ui.widgets.EditableMixin method*), 39
get() (*hoverset.ui.widgets.ProgressBar method*), 42
get() (*hoverset.ui.widgets.RadioButtonGroup method*), 43
get() (*hoverset.ui.widgets.Scale method*), 43
get() (*hoverset.ui.widgets.SpinBox method*), 44
get() (*hoverset.ui.widgets.Tree method*), 47
get_body() (*hoverset.ui.widgets.Tree method*), 47
get_body() (*hoverset.ui.widgets.TreeView method*), 47
get_class() (*hoverset.ui.widgets.CompoundList method*), 37
get_mode() (*hoverset.ui.widgets.CompoundList method*), 37
get_pos() (*hoverset.ui.widgets.PositionMixin method*), 41

H

height (*hoverset.ui.widgets.Widget property*), 50
hoverset.ui.dialogs
    module, 51
hoverset.ui.menu
    module, 55
hoverset.ui.widgets
    module, 35

I

ImageCacheMixin (*class in hoverset.ui.widgets*), 41
index() (*hoverset.ui.widgets.Tree.Node method*), 46
initialize_tree() (*hoverset.ui.widgets.Tree method*), 47
insert() (*hoverset.ui.widgets.Tree.Node method*), 46
insert_after() (*hoverset.ui.widgets.Tree.Node method*), 46
insert_before() (*hoverset.ui.widgets.Tree.Node method*), 46
interval() (*hoverset.ui.widgets.ProgressBar method*), 42
is_descendant() (*hoverset.ui.widgets.Tree.Node method*), 46

is_within() (*in module studio.ui.geometry*), 61

L

Label (*class in hoverset.ui.widgets*), 41
LabelFrame (*class in hoverset.ui.widgets*), 41
load_node() (*formation.loader.Builder method*), 33
load_path() (*formation.loader.Builder method*), 33
load_string() (*formation.loader.Builder method*), 33
load_styles() (*hoverset.ui.widgets.Application method*), 35
LoadLater (*class in hoverset.ui.menu*), 55

M

mainloop() (*formation.loader.AppBuilder method*), 31
make_event_relative() (*in module studio.ui.geometry*), 61
make_menu() (*hoverset.ui.widgets.ContextMenuMixin method*), 38
manipulated() (*hoverset.ui.menu.EnableIf method*), 55
manipulated() (*hoverset.ui.menu.LoadLater method*), 55
manipulated() (*hoverset.ui.menu.Manipulator method*), 55
manipulated() (*hoverset.ui.menu.ShowIf method*), 55
Manipulator (*class in hoverset.ui.menu*), 55
MenuButton (*class in hoverset.ui.widgets*), 41
Message (*class in hoverset.ui.widgets*), 41
MessageDialog (*class in hoverset.ui.dialogs*), 51
mode() (*hoverset.ui.widgets.ProgressBar method*), 42
module
- formation.loader, 31
- formation.utils, 33
- hoverset.ui.dialogs, 51
- hoverset.ui.menu, 55
- hoverset.ui.widgets, 35
- studio.ui.geometry, 60

N

name (*hoverset.ui.widgets.Tree.Node property*), 46
names() (*hoverset.ui.widgets.FontStyle static method*), 40
nametofont() (*hoverset.ui.widgets.FontStyle static method*), 41

O

on_change() (*hoverset.ui.widgets.CompoundList method*), 37
on_change() (*hoverset.ui.widgets.EditableMixin method*), 39
on_deselect() (*hoverset.ui.widgets.TabView.Tab method*), 45
on_drag() (*hoverset.ui.widgets.TabView.Tab method*), 45
on_drag() (*hoverset.ui.widgets.Widget method*), 50
on_drag_end() (*hoverset.ui.widgets.TabView.Tab method*), 45
on_drag_end() (*hoverset.ui.widgets.Widget method*), 50
on_drag_start() (*hoverset.ui.widgets.TabView.Tab method*), 45
on_drag_start() (*hoverset.ui.widgets.Widget method*), 50
on_entry() (*hoverset.ui.widgets.EditableMixin method*), 39
on_hover() (*hoverset.ui.widgets.CompoundList.BaseItem method*), 37
on_hover_ended() (*hoverset.ui.widgets.CompoundList.BaseItem method*), 37
on_select() (*hoverset.ui.widgets.TabView.Tab method*), 45

P

PanedWindow (*class in hoverset.ui.widgets*), 41
parse_geometry() (*in module studio.ui.geometry*), 61
path (*formation.loader.Builder property*), 33
Popup (*class in hoverset.ui.widgets*), 41
popup() (*hoverset.ui.widgets.ContextMenuMixin static method*), 39
PositionMixin (*class in hoverset.ui.widgets*), 41
post() (*hoverset.ui.widgets.PositionMixin method*), 41
ProgressBar (*class in hoverset.ui.widgets*), 42

R

RadioButton (*class in hoverset.ui.widgets*), 42
RadioButtonGroup (*class in hoverset.ui.widgets*), 42
relative_bounds() (*in module studio.ui.geometry*), 61
remove() (*hoverset.ui.widgets.Tree.Node method*), 46
render() (*hoverset.ui.widgets.CompoundList.BaseItem method*), 37
render_drag() (*hoverset.ui.widgets.TabView.Tab method*), 45
render_drag() (*hoverset.ui.widgets.Widget method*), 50
render_tooltip() (*hoverset.ui.widgets.Widget method*), 50
resolve_bounds() (*in module studio.ui.geometry*), 61
resolve_position() (*in module studio.ui.geometry*), 62

S

Scale (*class in hoverset.ui.widgets*), 43
Screen (*class in hoverset.ui.widgets*), 43
ScrollableInterface (*class in hoverset.ui.widgets*), 44
ScrolledFrame (*class in hoverset.ui.widgets*), 44
search() (*hoverset.ui.widgets.Tree.Node method*), 46
select() (*hoverset.ui.widgets.CompoundList method*), 38
select() (*hoverset.ui.widgets.CompoundList.BaseItem method*), 37

select() (*hoverset.ui.widgets.Tree method*), 47
select() (*hoverset.ui.widgets.Tree.Node method*), 46
select() (*hoverset.ui.widgets.Tree.Strip method*), 47
select_self() (*hoverset.ui.widgets.CompoundList.BaseItem method*), 37
selected_count() (*hoverset.ui.widgets.Tree method*), 47
set() (*hoverset.ui.widgets.Checkbutton method*), 36
set() (*hoverset.ui.widgets.ProgressBar method*), 42
set() (*hoverset.ui.widgets.RadioButtonGroup method*), 43
set() (*hoverset.ui.widgets.Scale method*), 43
set_choices() (*hoverset.ui.widgets.RadioButtonGroup method*), 43
set_item_class() (*hoverset.ui.widgets.CompoundList method*), 38
set_label() (*hoverset.ui.widgets.RadioButtonGroup method*), 43
set_mode() (*hoverset.ui.widgets.CompoundList method*), 38
set_scrollbars() (*hoverset.ui.widgets.ScrolledFrame method*), 44
set_ttk_style() (*in module hoverset.ui.widgets*), 51
set_up_context() (*hoverset.ui.widgets.ContextMenuMixin method*), 39
set_validator() (*hoverset.ui.widgets.EditableMixin method*), 40
set_values() (*hoverset.ui.widgets.CompoundList method*), 38
setup() (*hoverset.ui.widgets.Widget method*), 50
show() (*hoverset.ui.widgets.ToolWindow method*), 45
show_error() (*hoverset.ui.dialogs.MessageDialog class method*), 54
show_info() (*hoverset.ui.dialogs.MessageDialog class method*), 54
show_progress() (*hoverset.ui.dialogs.MessageDialog class method*), 54
show_warning() (*hoverset.ui.dialogs.MessageDialog class method*), 55
ShowIf (*class in hoverset.ui.menu*), 55
SpinBox (*class in hoverset.ui.widgets*), 44
Spinner (*class in hoverset.ui.widgets*), 44
studio.ui.geometry
    module, 60
style (*hoverset.ui.widgets.Application property*), 35
suppress_change() (*in module hoverset.ui.widgets*), 51
system_fonts() (*in module hoverset.ui.widgets*), 51

T

TabView (*class in hoverset.ui.widgets*), 44
TabView.Tab (*class in hoverset.ui.widgets*), 45
Text (*class in hoverset.ui.widgets*), 45
toggle() (*hoverset.ui.widgets.Tree.Node method*), 46
toggle_select() (*hoverset.ui.widgets.Tree.Node method*), 46
ToggleButton (*class in hoverset.ui.widgets*), 45
tooltip() (*hoverset.ui.widgets.Widget method*), 50
ToolWindow (*class in hoverset.ui.widgets*), 45
Tree (*class in hoverset.ui.widgets*), 45
Tree.Node (*class in hoverset.ui.widgets*), 45
Tree.Strip (*class in hoverset.ui.widgets*), 46
TreeView (*class in hoverset.ui.widgets*), 47

U

unbind_all() (*hoverset.ui.widgets.Application method*), 35
upscale_bounds() (*in module studio.ui.geometry*), 62

V

value (*hoverset.ui.widgets.CompoundList.BaseItem property*), 37

W

Widget (*class in hoverset.ui.widgets*), 48
WidgetError, 50
width (*hoverset.ui.widgets.Widget property*), 50
Window (*class in hoverset.ui.widgets*), 50
WindowMixin (*class in hoverset.ui.widgets*), 51

X

x (*hoverset.ui.widgets.EventWrap attribute*), 40
x_root (*hoverset.ui.widgets.EventWrap attribute*), 40

Y

y (*hoverset.ui.widgets.EventWrap attribute*), 40
y_root (*hoverset.ui.widgets.EventWrap attribute*), 40