GUIs with Flet

Programmazione Avanzata 2025-26

Graphics user interfaces

- Graphics user interfaces (GUIs)
  - Foster an enhanced, user-friendly interaction in applications
  - Provide a productivity boost
  - Support accessibility and inclusivity
  - Enable a visual representation of data

GUIs in Python

- Several alternatives
  - Tk
    - Default graphic library
    - Basic set of widgets for building GUIs
  - Flutter
    - Proposed by Google as a multi-platform UI software devel kit
  - Flet
    - Library to build Flutter apps in Python
    - Support the creation of web, mobile and desktop apps
    - Requires minimal frontend knowledge
    - https://flet.dev

Creating the first app with Flet

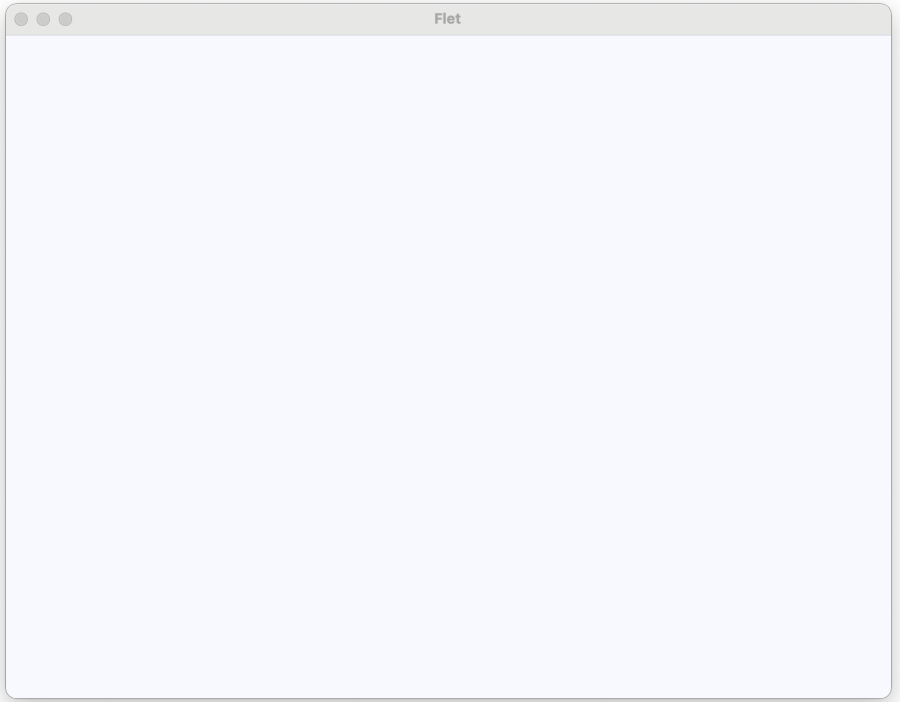
- After having installed and imported the **`flet`** package (e.g., v0.28.3), function **`app()`** needs to be invoked to create an app
- The **`app()`** function receives a **`target`** argument that specifies the function to be called for drawing the interface

```
import flet as ft

def main(page : ft.Page): # or simply main(page)
    # Here, controls are added and updated
    pass

ft.app(target = main)
```

Containers

- Flet apps are organized in containers, which host all all the graphics elements
- The top-most container is **`View`**
  - Default option is **`FLET_APP`**, though other constants can be used to create other types of apps (e.g., **`FLET_APP_WEB`**)

```
ft.app(target = main,
view = ft.AppView.FLET_APP)
```



Controls

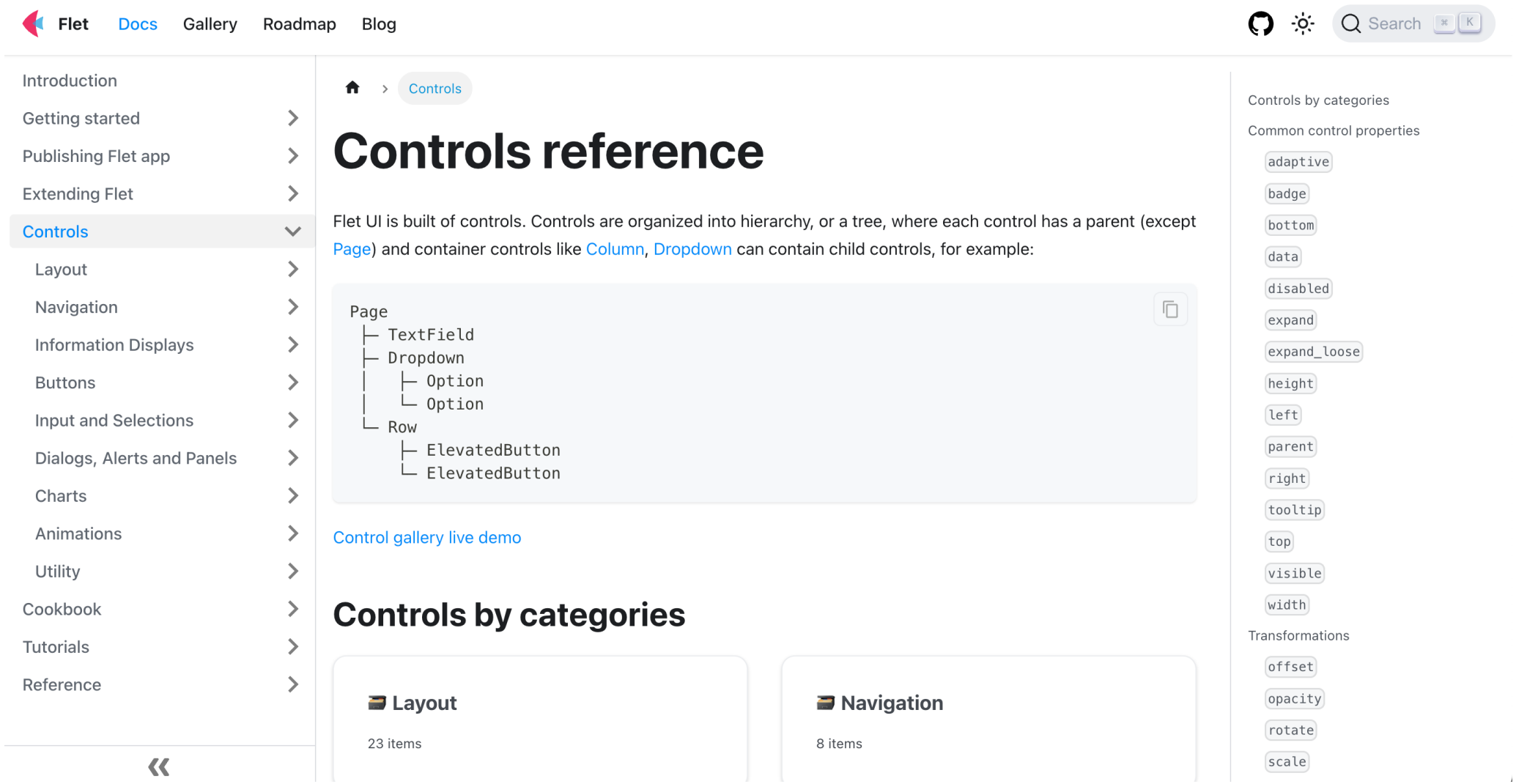
- Within **`View`**, a **`Page`** container is defined, where the GUI elements will be added
- GUI elements are called controls and are simply Python classes which need to be instantiated in a **`Page`** (acting as root control)
- Many types of controls available, each coming with many configurable parameters
  - https://flet.dev/docs/controls

Controls

- For instance, to add some text it is possible to use **`ft.Text()`**, setting initial value, color, size, etc.
- Information on the control and its parameters is available in Flet documentation

```
import flet as ft

def main(page):
    # Here, controls are added and updated
    myText = ft.Text(value = "Hello!", color = "blue")

ft.app(target = main, view = ft.AppView.FLET_APP)
```

Controls


Flet
Docs
Gallery
Roadmap
Blog
Search
Introduction
Getting started
Publishing Flet app
Extending Flet
Controls
Layout
Navigation
Information Displays
Buttons
Input and Selections
Dialogs, Alerts and Panels
Charts
Animations
Utility
Cookbook
Tutorials
Reference
Controls
Controls reference
Flet UI is built of controls. Controls are organized into hierarchy, or a tree, where each control has a parent (except Page) and container controls like Column, Dropdown can contain child controls, for example:
Page
├─ TextField
├─ Dropdown
│ ├─ Option
│ └─ Option
└─ Row
├─ ElevatedButton
└─ ElevatedButton
Control gallery live demo
Controls by categories
Layout
23 items
Navigation
8 items
Controls by categories
Common control properties
adaptive
badge
bottom
data
disabled
expand
expand_loose
height
left
parent
right
tooltip
top
visible
width
Transformations
offset
opacity
rotate
scale

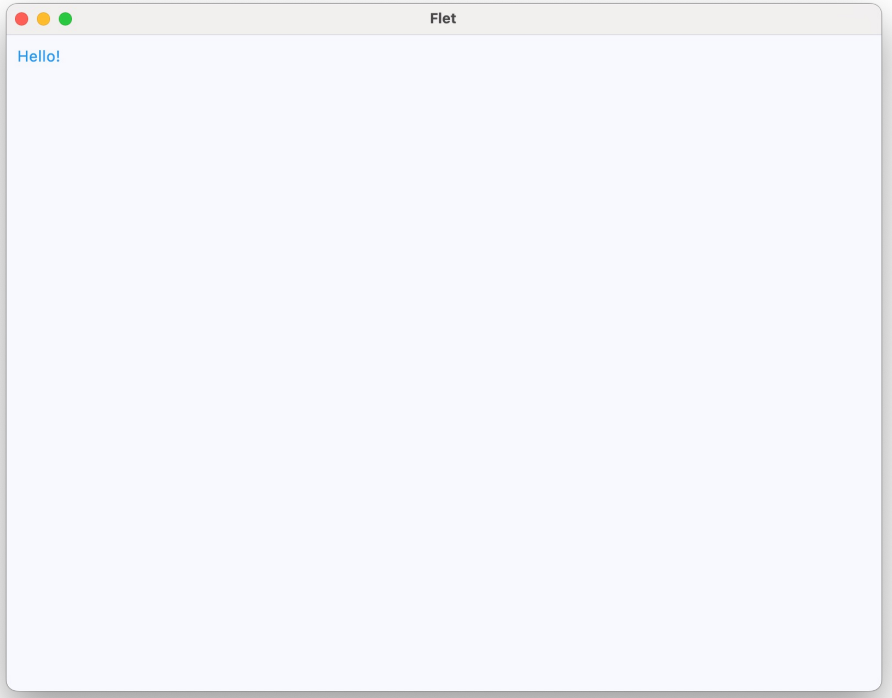
Adding controls

- To make a control visible in the page
  - It has to be added to the page (better, to the controls of that page)
  - Then, the page needs to be updated

```
import flet as ft

def main(page):
    # Here, controls are added and updated
    myText = ft.Text(value = "Hello!", color = "blue")
    page.controls.append(myText)
    page.update()

ft.app(target = main, view = ft.AppView.FLET_APP)
```

Updates

- Elements in a page are updated only when **`page.update()`** is called
- Content of controls can be updated anytime, not only when added to the page



Updates

- An example could be a counter that is incremented every one second

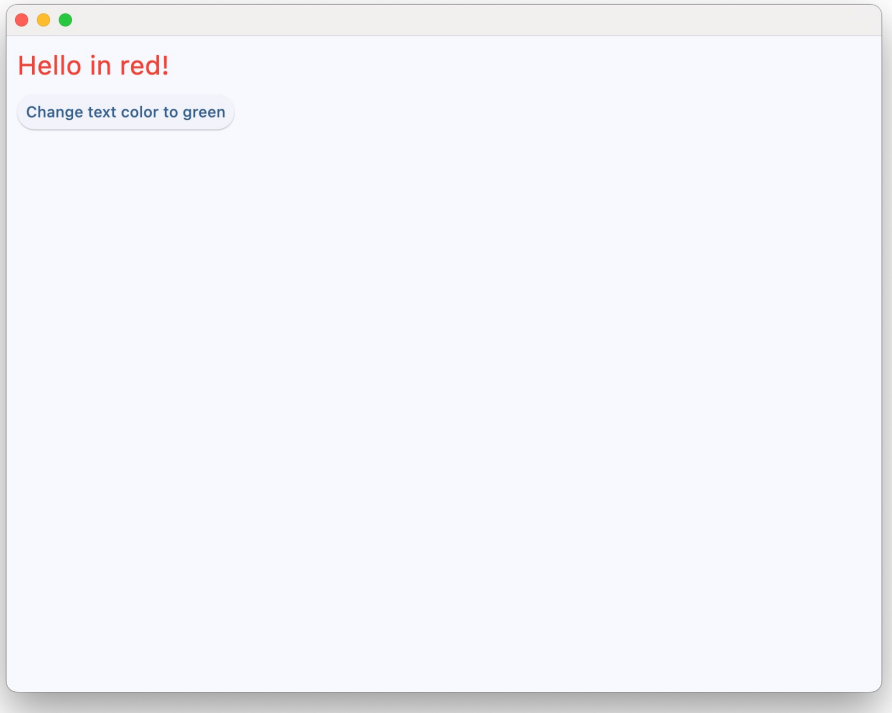
```
def main(page):
    myText =
        ft.Text(value="Counter:",
                size=24)
    page.controls.append(myText)
    page.update()
    for i in range(1,100):
        myText.value = "Counter: "
                       + str(i)
        myText.update()
        # from time import sleep
        sleep(1)
```

Handling events

- Some control items can trigger events
  - E.g, buttons when clicked
- To handle the event (i.e., respond to it by performing some operations), an event handler function, or simply event handler, needs to be specified when creating the control
  - Different event handlers, depending on the event: **`on_click`**, **`on_focus`**, **`on_hover`**, etc.

Events and updates

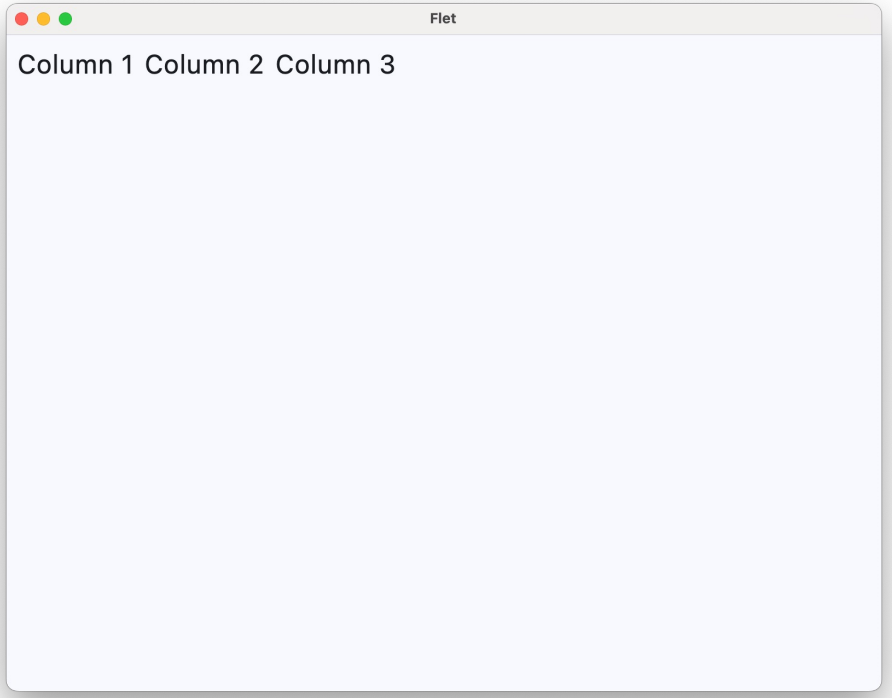
```
def main(page):

    myText = ft.Text(value="Hello in red!",
                     color="red",size=24)

    # Event handler function for button pressed
    def handleChangeColor(e):
        print("Changing color!")
        myText.value = "Hello in green!"
        myText.color = "green"
        myText.update()

    btnIncrease =
          ft.ElevatedButton(text="Change text
          color to green", on_click=handleChange)

    page.controls.append(myText)
    page.controls.append(btnIncrease)
    page.update()
```



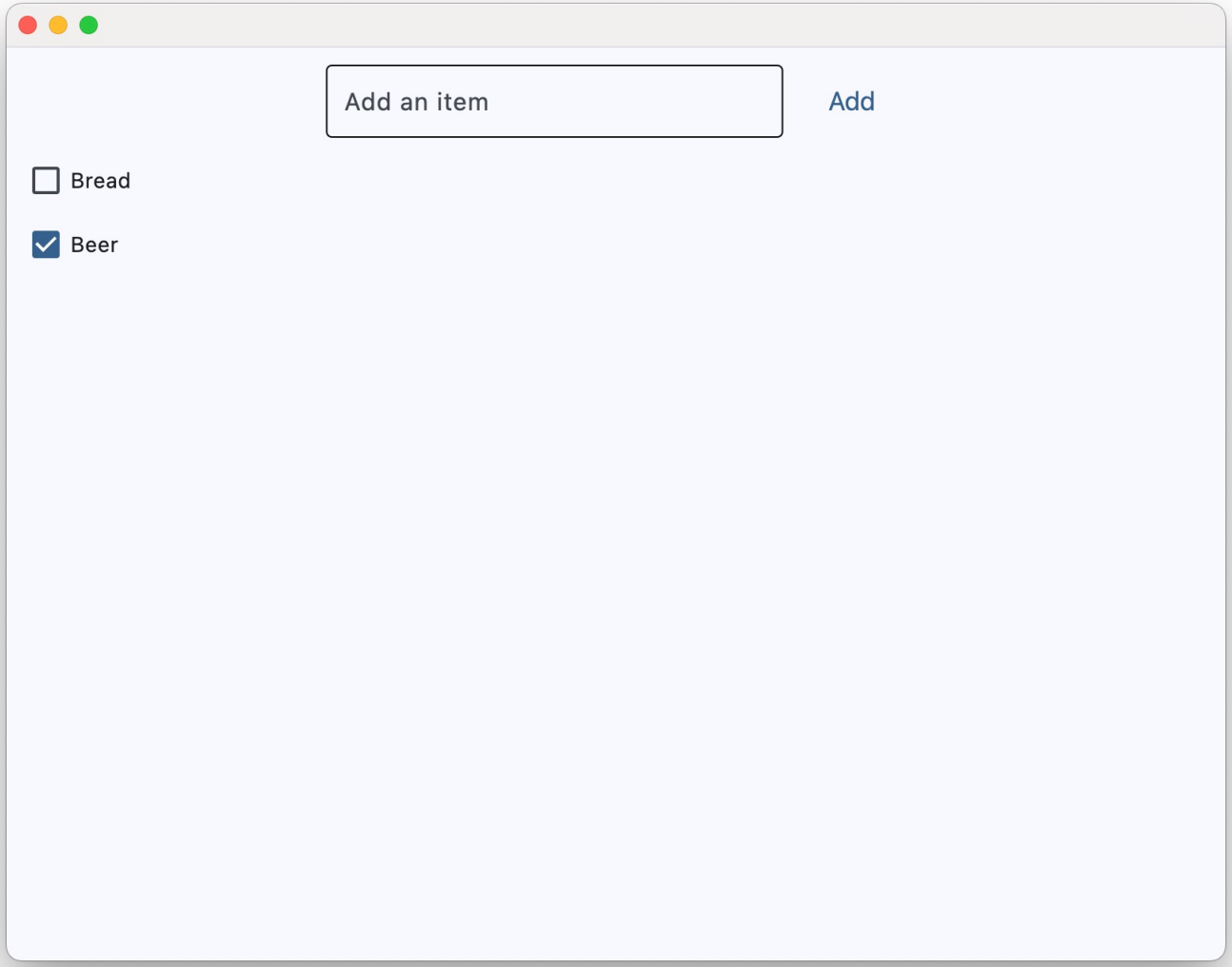
Organizing controls

- To give some structure to a page, containers can be nested
- For instance, it is possible to organize controls in rows using **`ft.Row()`**

```
def main(page):
    myText1=ft.Text(value="Col1")
    myText2=ft.Text(value="Col2")
    myText3=ft.Text(value="Col3")

    row = ft.Row([myText1,
                  myText2,
                  myText3])
    page.controls.append(row)
    page.update()
```



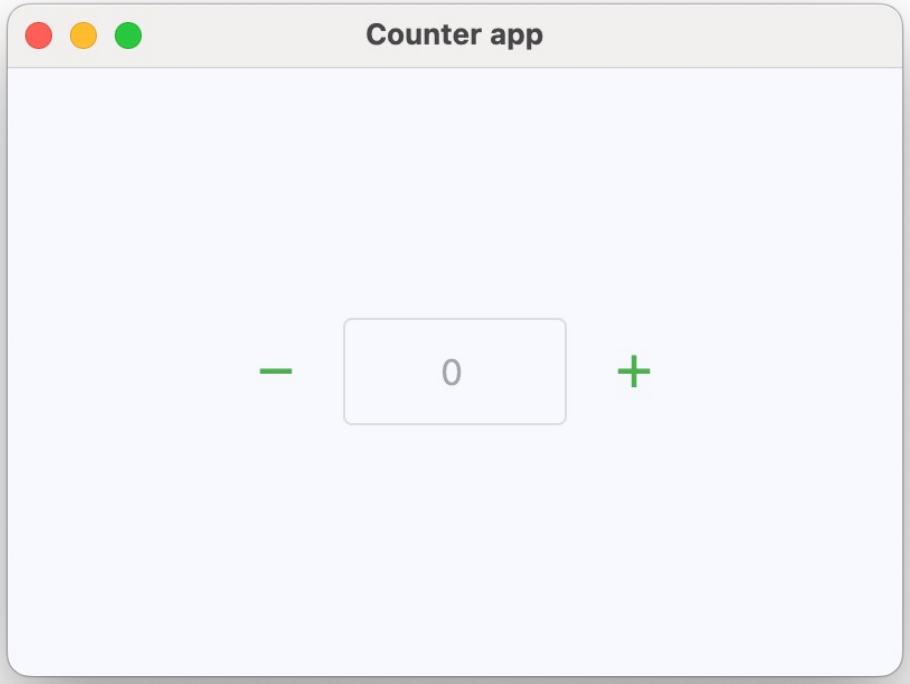
Example: todo list app



Example: todo list app

```
def main(page: ft.Page):

    def addCheckbox(e):
        strToAdd = txtIn.value
        txtIn.value = ""
        if strToAdd == "":
            return
        page.add(ft.Checkbox(label=strToAdd))

    txtIn = ft.TextField(label="Add an item")
    btnAdd = ft.CupertinoButton(text="Add",
                                on_click=addCheckbox)
    row1 = ft.Row([txtIn,btnAdd],
                  alignment=ft.MainAxisAlignment.CENTER)
    page.add(row1)
```

Example: counter app



Example: counter app

```
def main(page: ft.Page):
    page.window.width = 400
    page.window.height = 300
    page.vertical_alignment = ft.MainAxisAlignment.CENTER
    page.window_resizable = True
    page.title = "Counter app"

    def handleAdd(e):
        currentVal = txtOut.value
        txtOut.value = currentVal + 1
        txtOut.update()

    def handleRemove(e):
        currentVal = txtOut.value
        txtOut.value = currentVal - 1
        txtOut.update()
```

Example: counter app

```
#https://gallery.flet.dev/icons-browser/

btnMinus = ft.IconButton(icon = ft.Icons.REMOVE,
                         icon_color="green",
                         icon_size= 24, on_click= handleRemove)
btnAdd = ft.IconButton(icon = ft.Icons.ADD,
                         icon_color="green",
                         icon_size= 24, on_click= handleAdd)

txtOut = ft.TextField(width=100,disabled=True,
                      value=0, border_color="green",
                      text_align=ft.TextAlign.CENTER)

row1 = ft.Row([btnMinus, txtOut, btnAdd],
              alignment=ft.MainAxisAlignment.CENTER)
page.add(row1)
```

Execution flow with GUIs

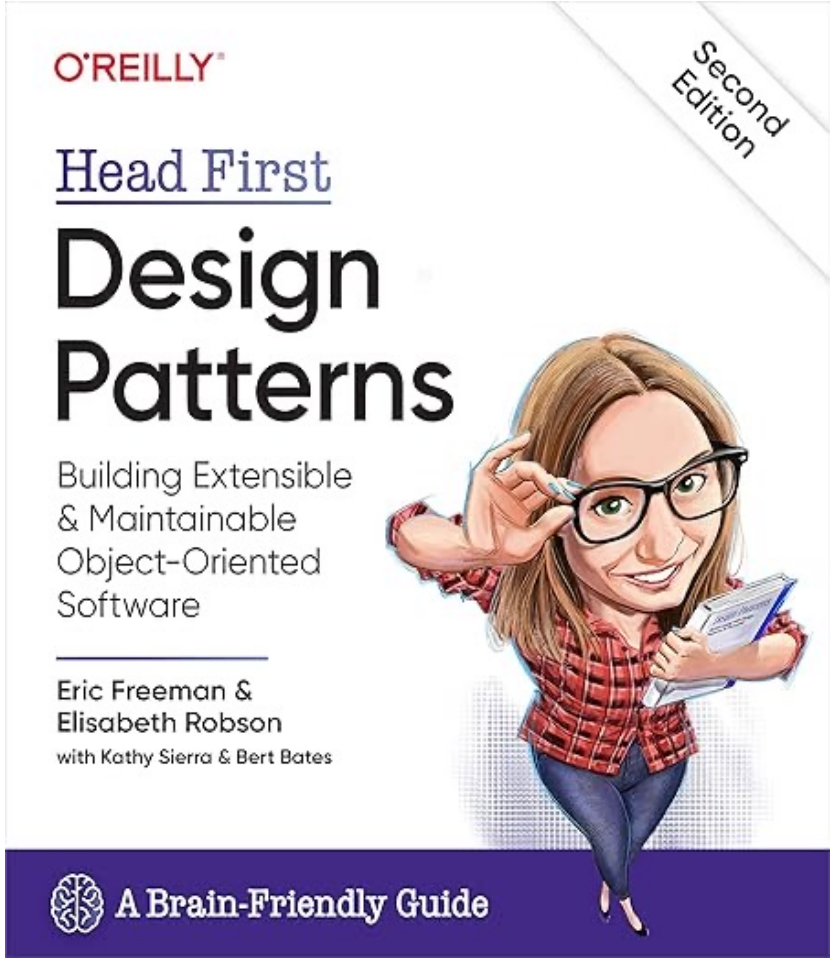
- Interactive, graphics applications exhibit complex interaction patterns
  - Operations are performed in response to external events (e.g., mouse clicks)
- User's actions are mainly asynchronous
  - Hence, there is no predefined order of execution in applications with a GUI
  - In the programs seen so far, method **`main()`** had the only goal of instantiating the graphics elements

Programs with GUIs: design questions

- How to organize the program for interactive graphics applications?
- Where to store data?
- How to decouple application logic from interface details?
- How to keep in sync the inner data with the visibile interface?

Design patterns

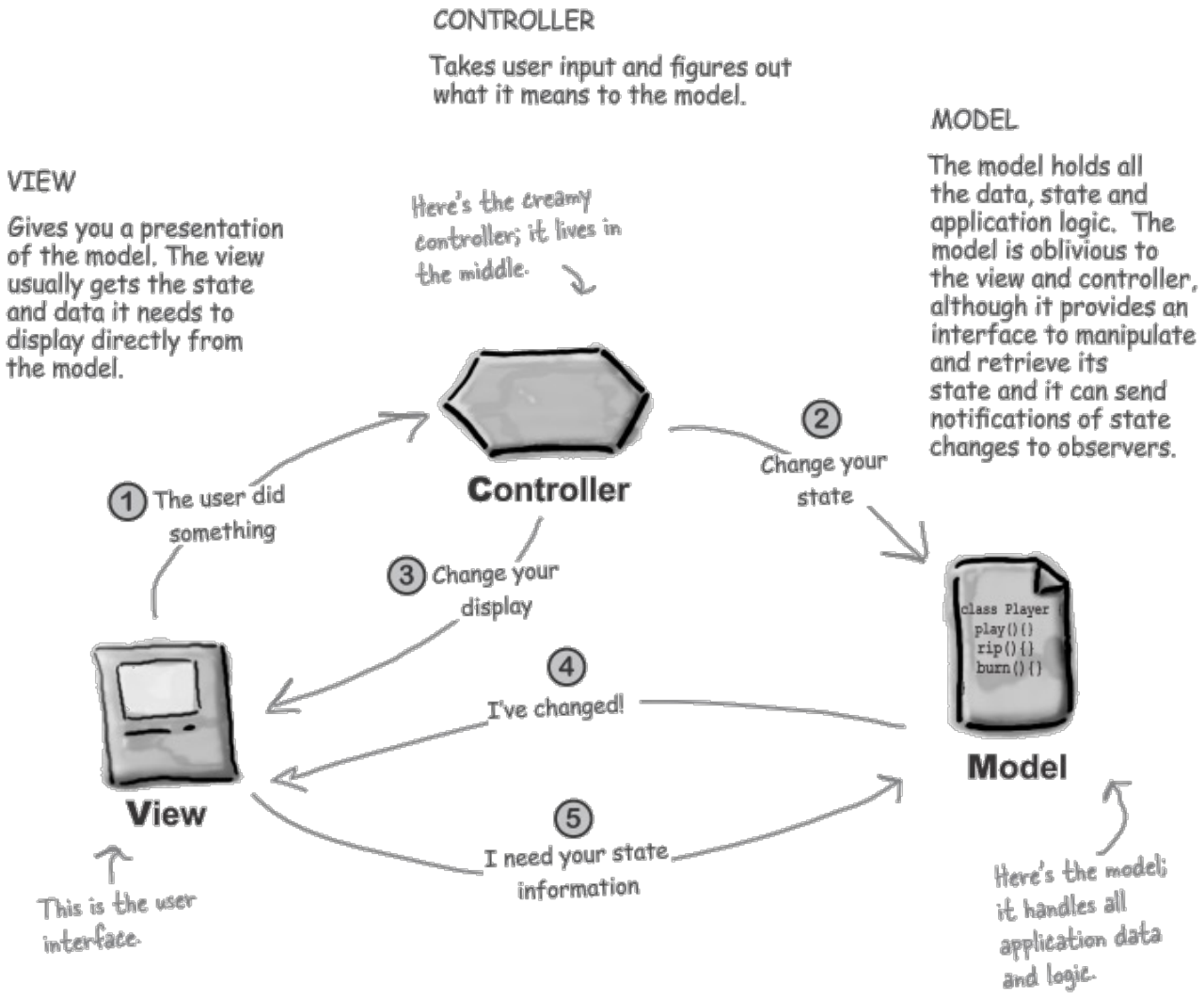
- How to build systems with good OO design qualities
  - Reusable, extensible, maintainable
- Patterns: proven solutions to recurrent problems
  - Design problems
  - Programming problems
- Adopt and combine the OO constructs
  - Interface, inheritance, abstract classes, information hiding, polymorphism, objects, …
- Help dealing with changes in software
  - Some part of a system is free to vary, independently from the rest



Pattern MVC

- According to the principles of Model-View-Controller (MVC) pattern, when building GUIs one must consider two main aspects:
  - Layout (View): how to place the graphical elements to achieve a give visual aspect
  - Events (Controller): which behavior associate to elements' events
  - Application logic and data (Model) must remain, as far as possible, separate from user interface

Pattern MVC


CONTROLLER
Takes user input and figures out what it means to the model.
MODEL
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.
VIEW
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.
Here's the creamy controller; it lives in the middle.
Controller
1 The user did something
2 Change your state
3 Change your display
4 I've changed!
5 I need your state information
View
Model
class Player
play()()
rip()()
burn()()
This is the user interface.
Here's the model; it handles all application data and logic.

Pattern MVC

1. **You're the user — you interact with the view.**
   The view is your window to the model. When you do something to the view (like click the Play button) then the view tells the controller what you did. It's the controller's job to handle that.

2. **The controller asks the model to change its state.**
   The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.

3. **The controller may also ask the view to change.**
   When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

4. **The model notifies the view when its state has changed.**
   When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

5. **The view asks the model for state.**
   The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

Mapping MVP principles to Python

- View: presenting the UI
  - Class acting as Flet's `View`, which instantiates the controls
  - Local variable to dialogue with the controller
- Controller: reacting to user actions
  - Hosts the event handlers
  - Local variables to handle the view's and status and access data and application logic in the model
- Model: handling the data
  - Class(es) including data and operations onto them, i.e., methods (as done before introducing the GUIs)
  - Persistent data stored, e.g., in databases

Example: counter app with MVC

- Main aspects to consider
  - How many classes are needed, and which ones?
  - Which class should have access to which?
  - Where are instances of those classes created?

File main.py

```
import flet as ft

from model import Model
from view import View
from controller import Controller

def main(page: ft.Page):
    m = Model()
    print(str(m.currentVal))
    v = View(page)
    c = Controller(v, m)
    v.setController(c)
    v.initInterface()

ft.app(target=main)
```

File model.py

```
class Model():
    def __init__(self):
        self._currentVal = 0

    @property
    def currentVal(self):
        return self._currentVal

    @currentVal.setter
    def currentVal(self, value):
        self._currentVal = value
```

File controller.py

```
import flet as ft
from model import Model

class Controller():
    def __init__(self, view : ft.View, model : Model):
        self._view = view
        self._model = model

    def handleAdd(self, e):
        self._model.currentVal += 1
        self._view.setTxtOutValue(self._model.currentVal)

    def handleRemove(self, e):
        self._model.currentVal -= 1
        self._view.setTxtOutValue(self._model.currentVal)
```

File view.py

```
import flet as ft
from mvc.controller import Controller

class View():
    def __init__(self, page : ft.Page):
        self._page = page
        self._controller = None

    def setController(self, controller):
        self._controller = controller

    def initInterface(self):
        self._page.window.width = 400
        self._page.window.height = 300
        self._page.vertical_alignment=
                                    ft.MainAxisAlignment.CENTER
        self._page.window_resizable = True
        self._page.title = "Counter app"
```

File view.py

```
        btnMinus = ft.IconButton(icon=ft.Icons.REMOVE,
                                 icon_color="green",
                                 icon_size= 24, on_click=
                                 self._controller.handleRemove)
        btnAdd = ft.IconButton(icon = ft.Icons.ADD,
                               icon_color="green",
                               icon_size= 24, on_click=
                               self._controller.handleAdd)

        # Instance var., accessible from other methods
        self._txtOut = ft.TextField(width=100,disabled=True,
                                    value=0,
                                    border_color="green",
                                    text_align=ft.TextAlign.CENTER)

        row1 = ft.Row([btnMinus, self._txtOut, btnAdd],
                      alignment=ft.MainAxisAlignment.CENTER)
        self._page.add(row1)
```

File view.py

```
def setTxtOutValue(self, value):
     self._txtOut.value = value
     self._txtOut.update()
```