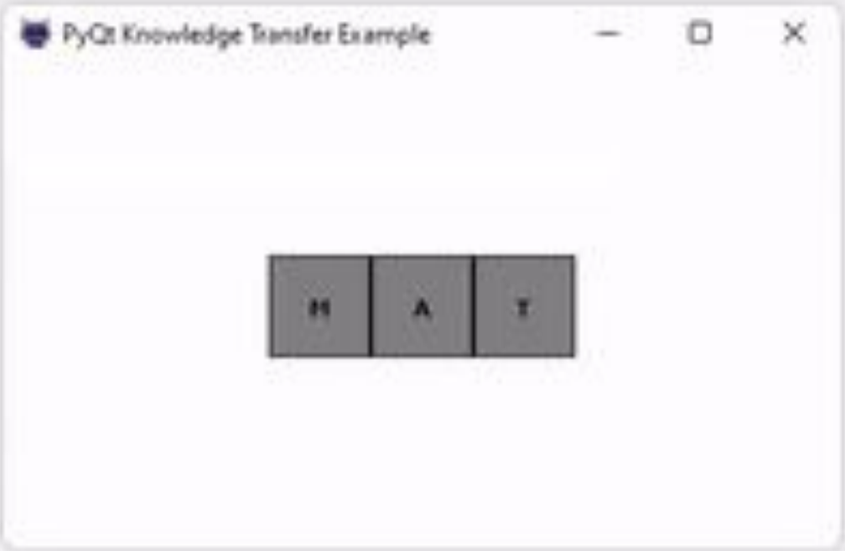
PyQt Knowledge Transfer

Victoria “Cat” Catlett

10 December 2024

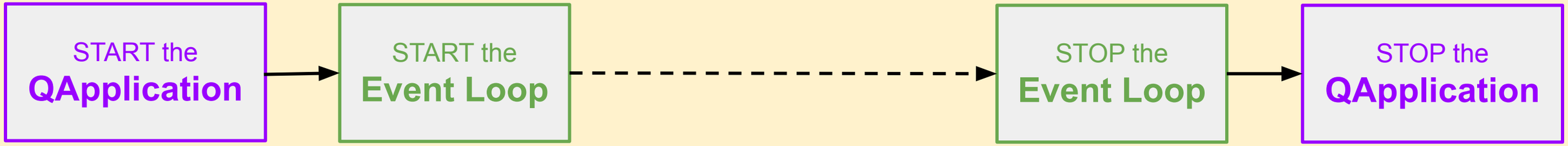
I want to walk you through building this



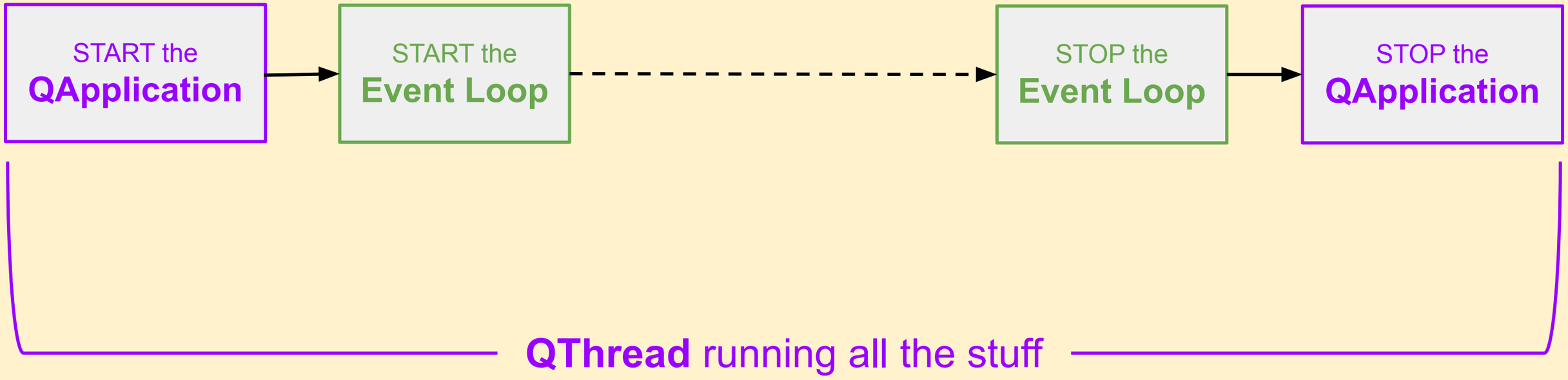
Explanation

Applied Example

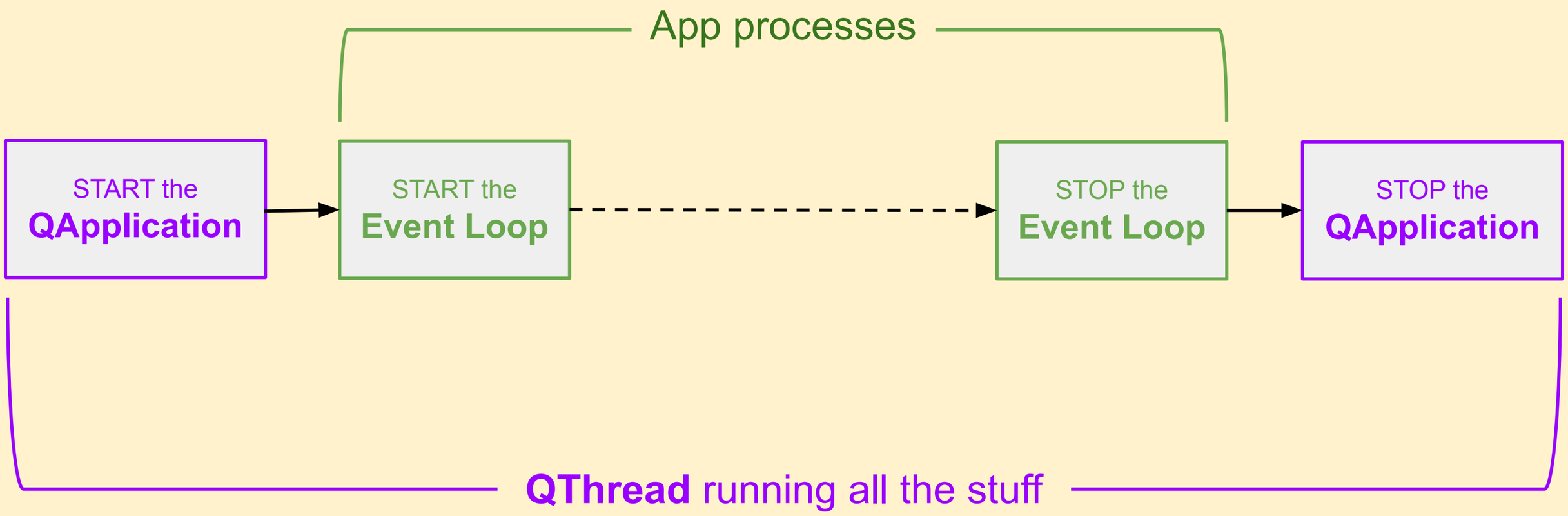
The base of any PyQt app is a **QApplication** and its **Event Loop**



The application runs in a single **QThread**



The app processes happen in the **Event Loop**



Here’s what this looks like in Python code

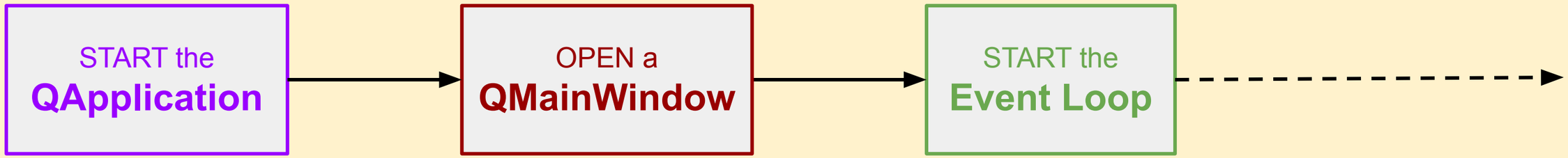
```python
import sys
from PyQt5.QtWidgets import QApplication

# Start the app
app = QApplication(sys.argv)

# Start the event loop
sys.exit(app.exec())
```

main.py
(eventually)

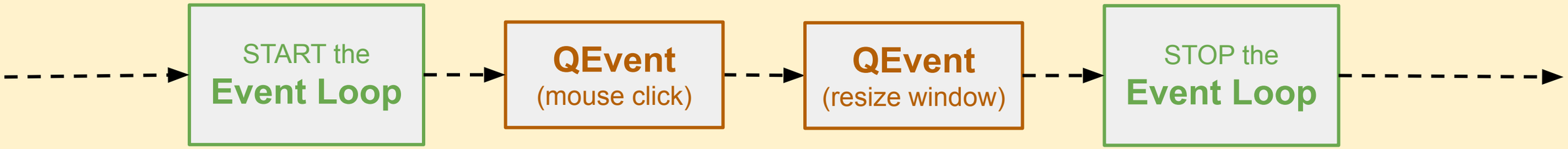
App windows are usually **QMainWindows**

(Note for the fancy developer: A “window” is actually just any visible widget that isn’t embedded in a parent)

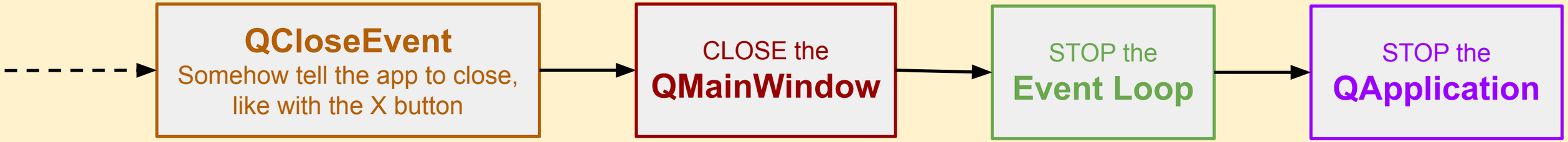
You typically open a primary **QMainWindow** before the **Event Loop** starts



QEvents are interactive things that happen to/in a window



The final event is usually a **QCloseEvent**

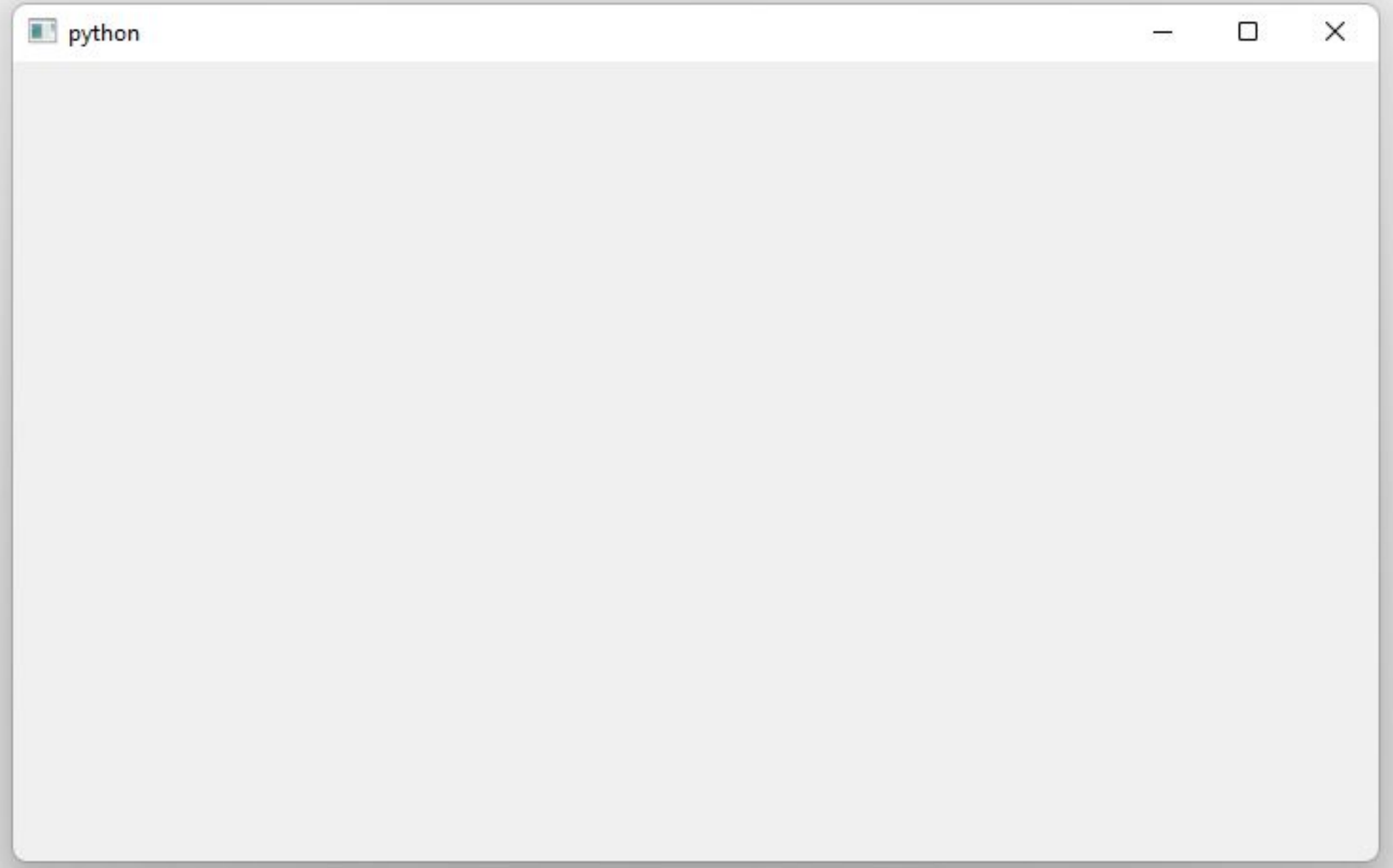


Let’s add a blank window to our app

```python
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow

# Start the app
app = QApplication(sys.argv)

# Define an instance of the QMainWindow class
window = QMainWindow()

# Tell the window to show itself
window.show()

# Start the event loop
sys.exit(app.exec())
```

main.py
(eventually)

Yay, a boring window. Let’s improve it.



The Qt docs describe what you can do to/with a **QMainWindow**

```
def addToolBar (title)
def addToolBar (toolbar)
def addToolBarBreak ([area=Qt.TopToolBarArea])
def centralWidget ()
def corner (corner)
def dockOptions ()
def dockWidgetArea (dockwidget)
def documentMode ()
def iconSize ()
def insertToolBar (before, toolbar)
def insertToolBarBreak (before)
def isAnimated ()
def isDockNestingEnabled ()
def isSeparator (pos)
```

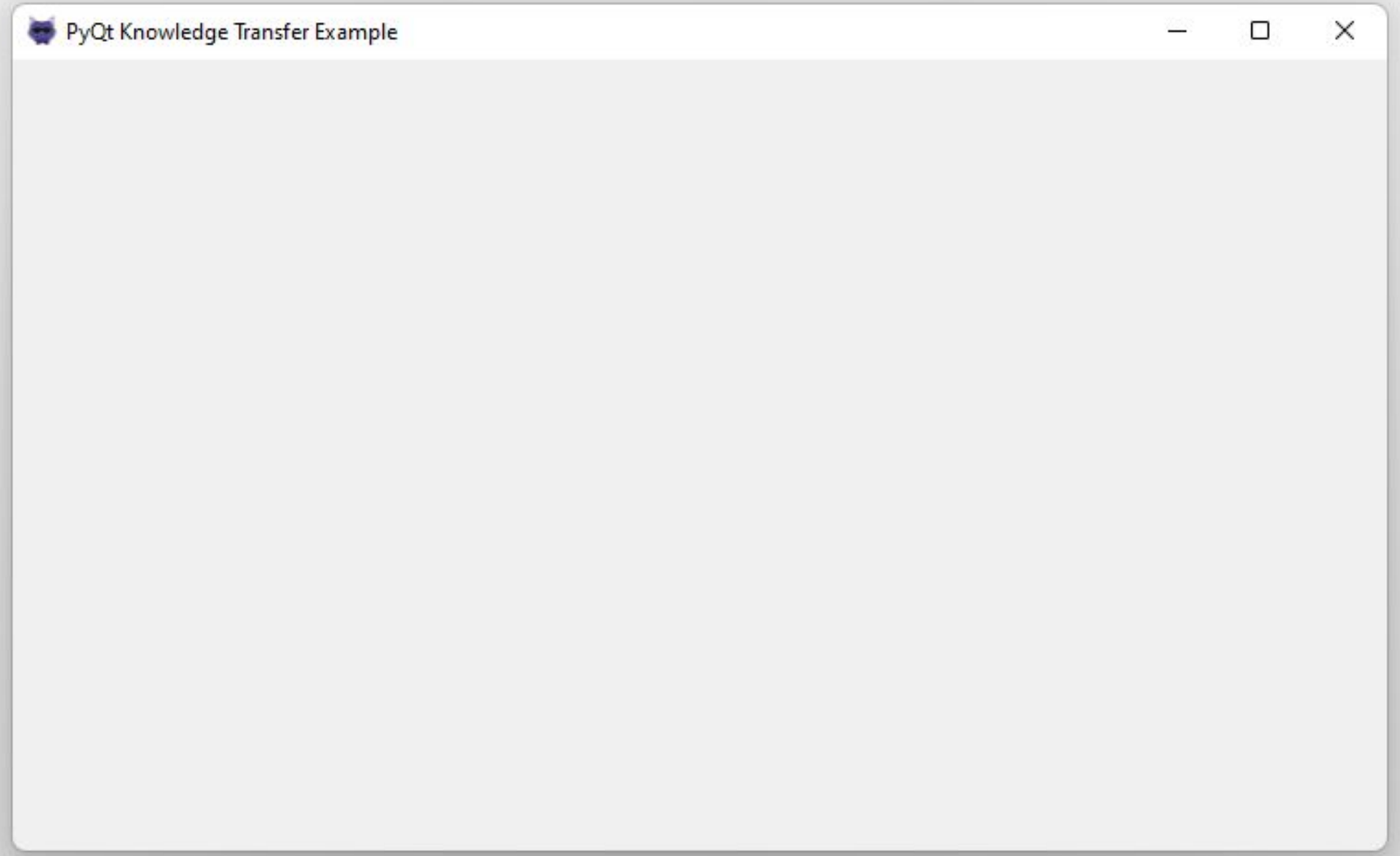
```
def menuBar ()
def menuWidget ()
def removeDockWidget (dockwidget)
def removeToolBar (toolbar)
def removeToolBarBreak (before)
def resizeDocks (docks, sizes, orientation)
def restoreDockWidget (dockwidget)
def restoreState (state[, version=0])
def saveState ([version=0])
def setCentralWidget (widget)
def setCorner (corner, area)
def setDockOptions (options)
def setDocumentMode (enabled)
def setIconSize (iconSize)
```

Let’s add a title and icon to the main window

```
# Define path to static files
STATIC_ROOT = Path(__file__).parent / "static"

# Start the app
app = QApplication(sys.argv)

# Define an instance of the QMainWindow class
window = QMainWindow()

# Set the window title
window.setWindowTitle("PyQt Knowledge Transfer Example")

# Add an icon
icon_path = STATIC_ROOT / "favicon.ico"
window.setWindowIcon(QIcon(str(icon_path)))
```

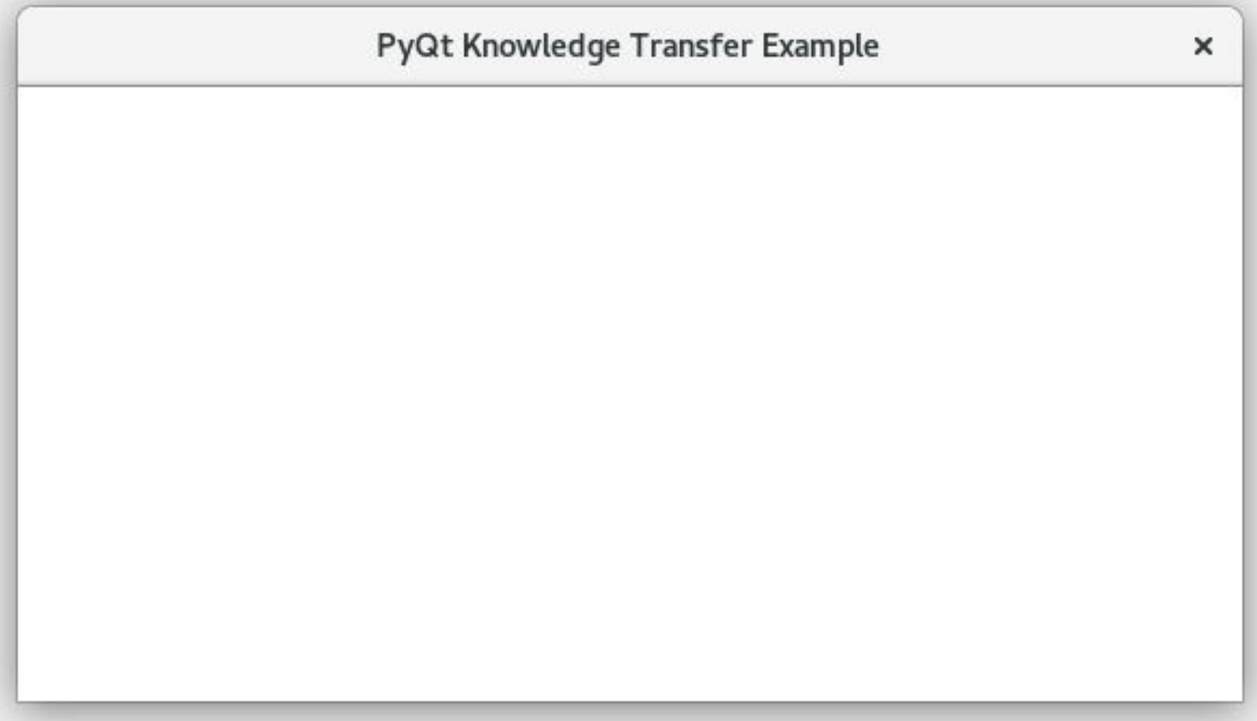
main.py
(eventually)

The window now has a title and an icon!

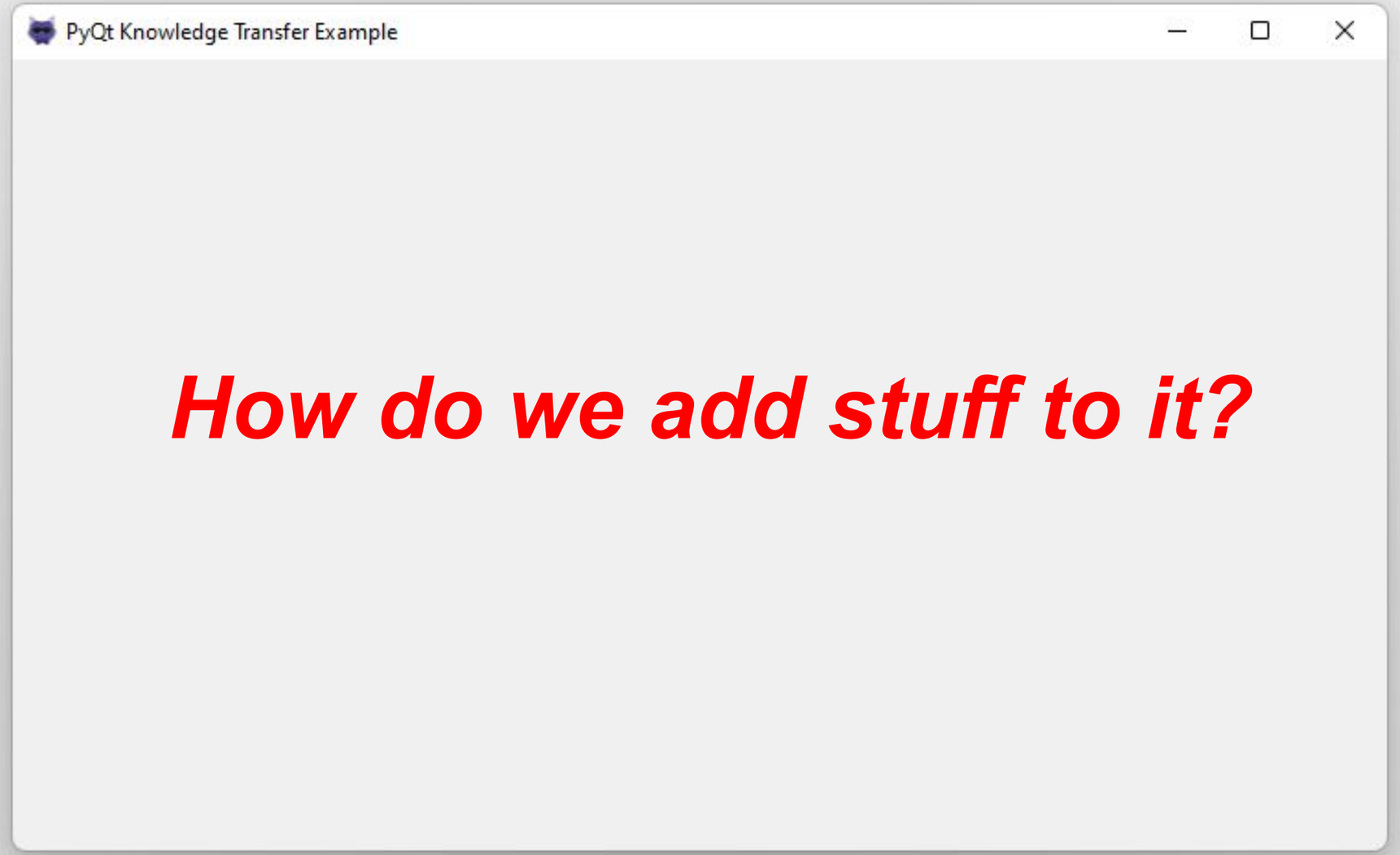


Note that things look different on the RHEL machines

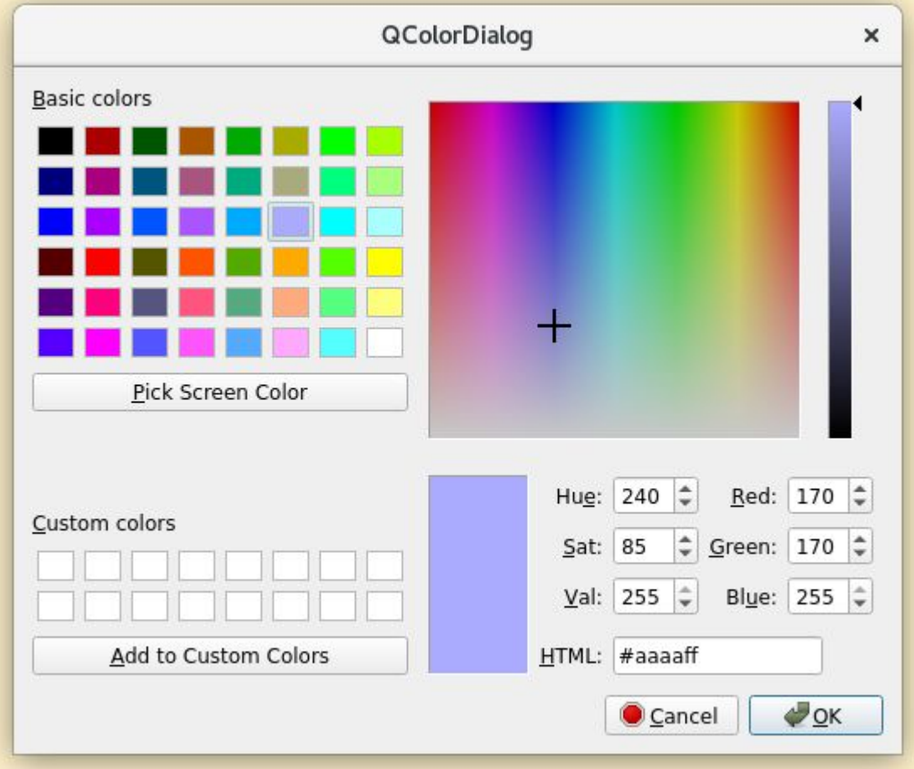
(Most of my screenshots are from Windows 11)

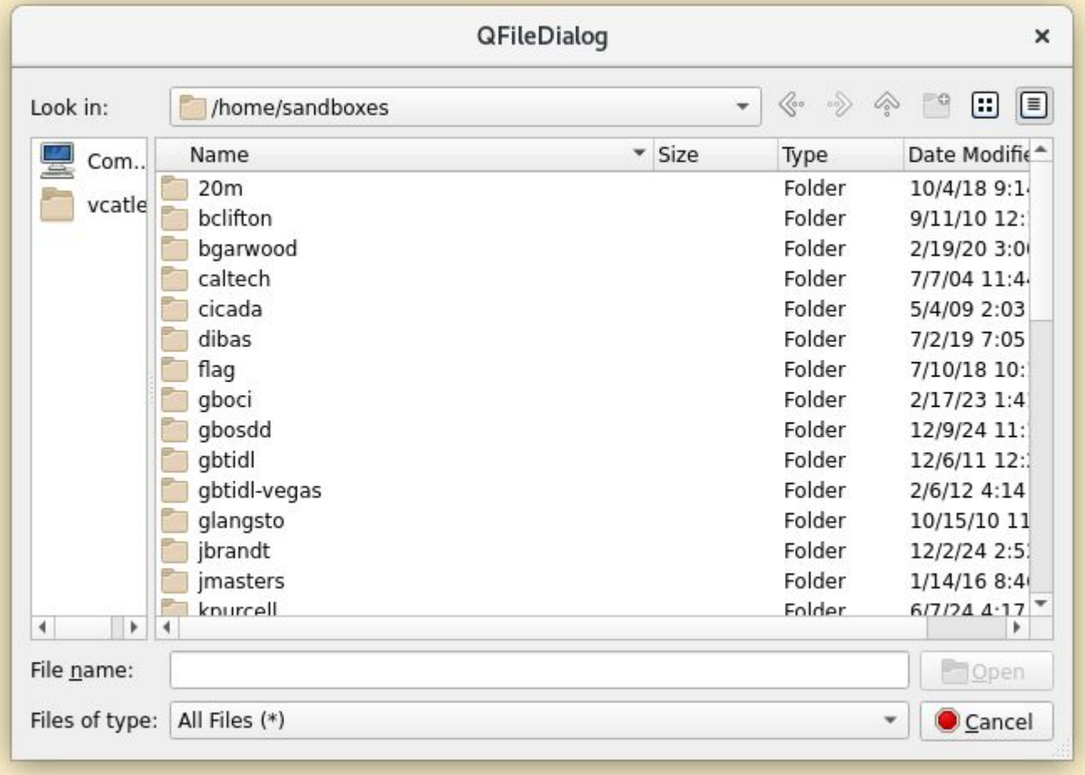


Now that we have a main window…

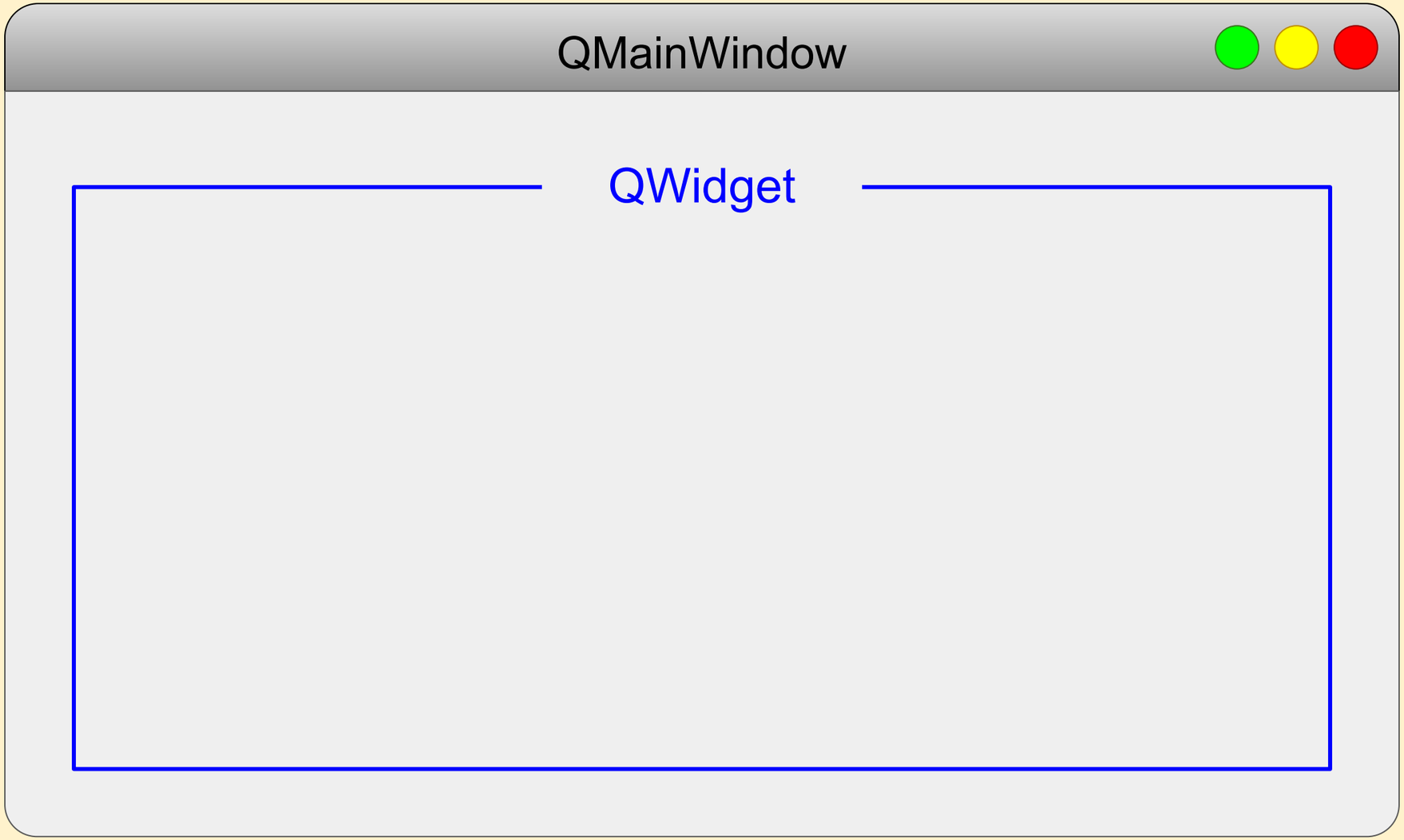


PyQt offers a few custom popup windows that inherit from **QMainWindow**





In general, a window must have a main **QWidget**

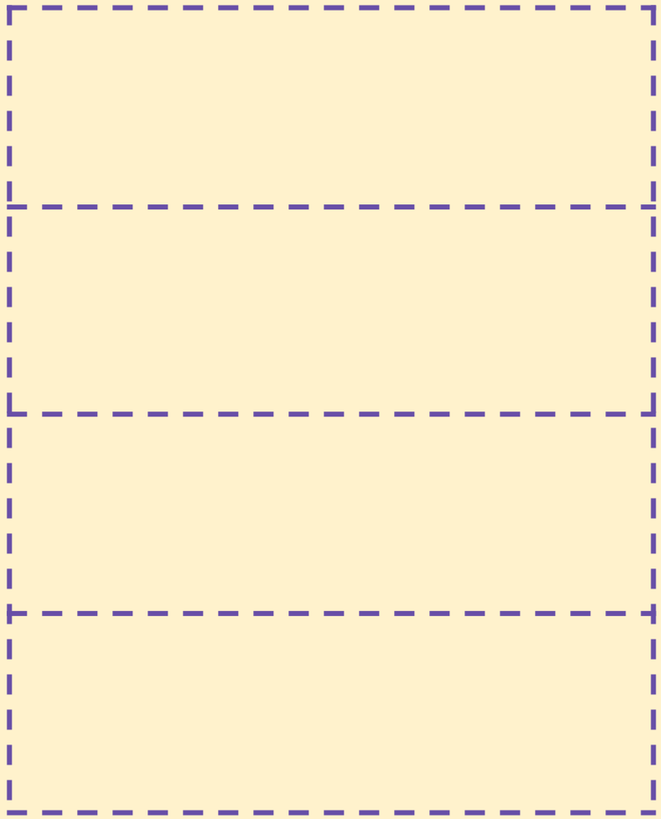


A **widget** is some sort of UI element.
PyQt has a lot of pre-made widgets, like these:

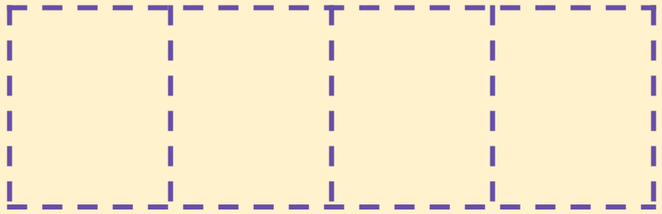
widget_showcase.py

A custom **widget** needs a **layout** to hold content (more widgets)

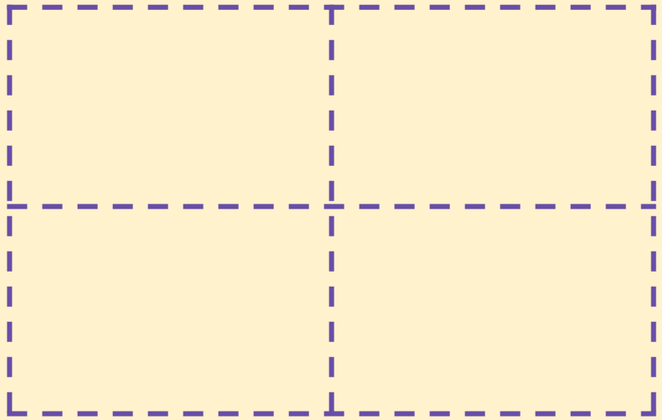
QVBoxLayout



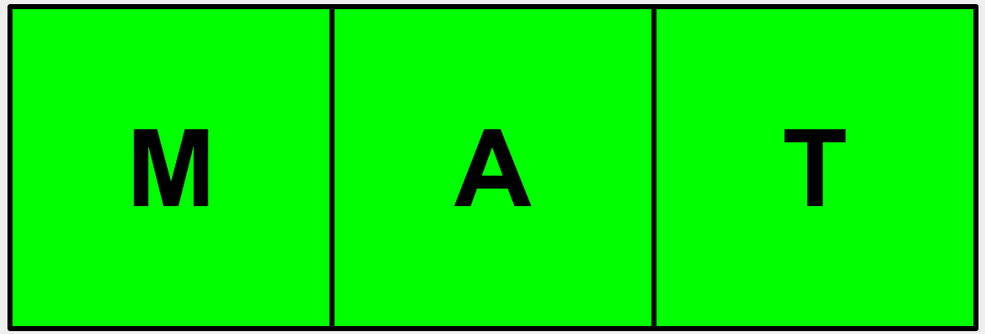
QHBoxLayout



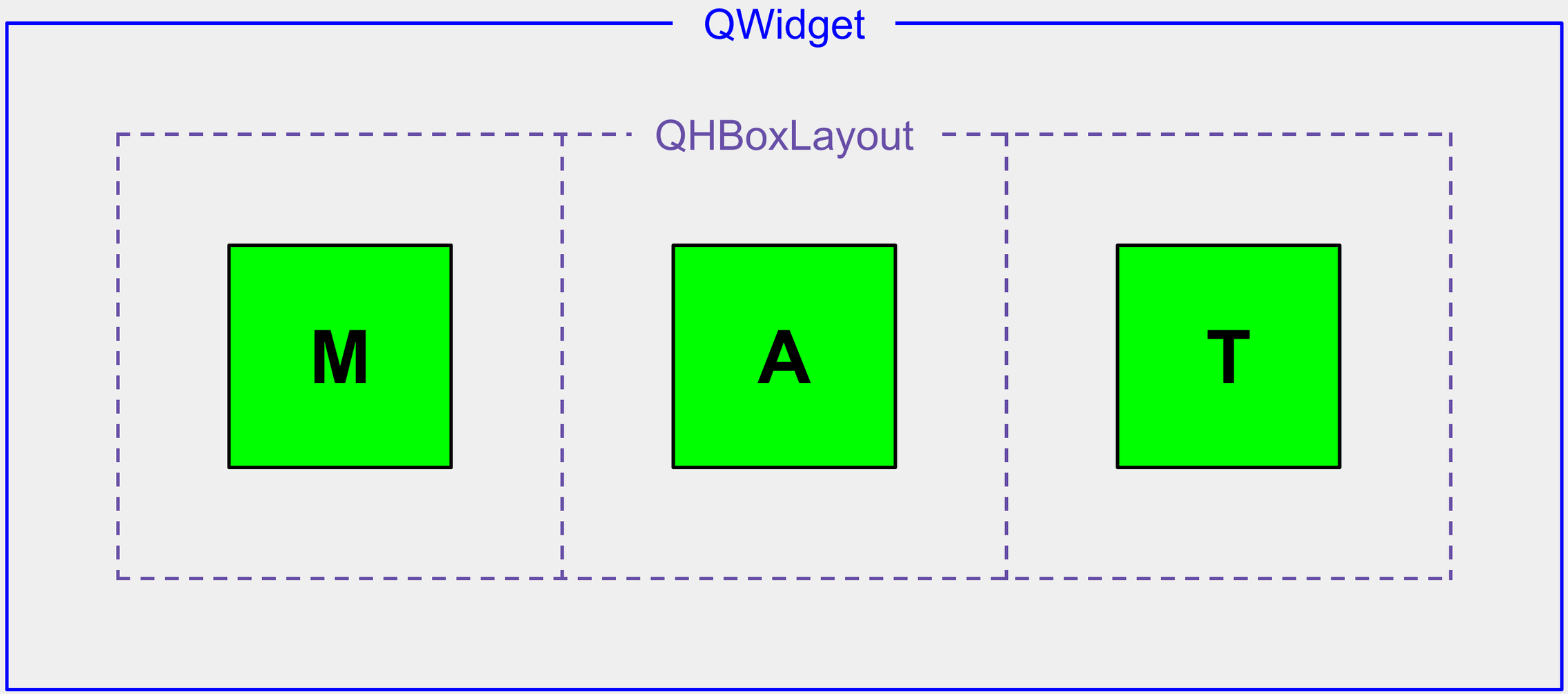
QGridLayout



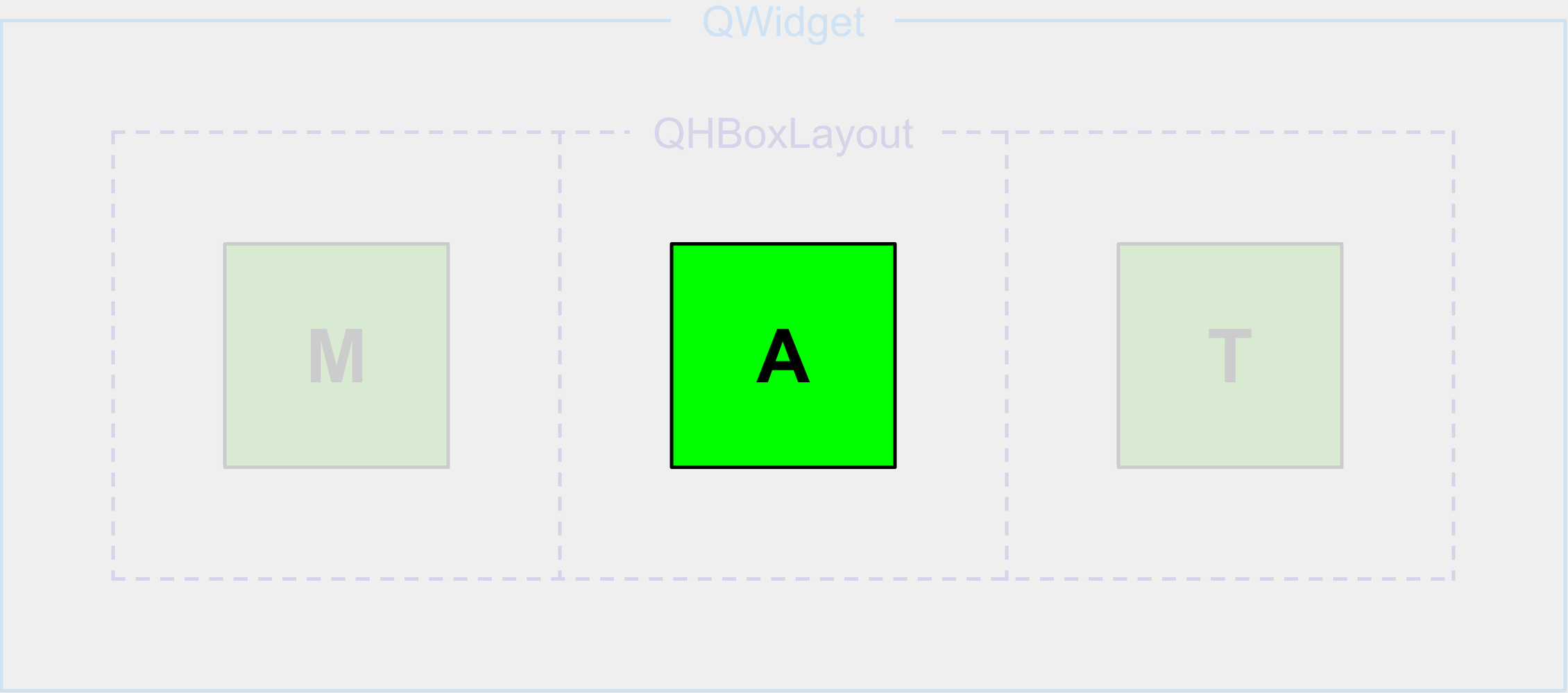
Let’s make a widget called **MATWidget**



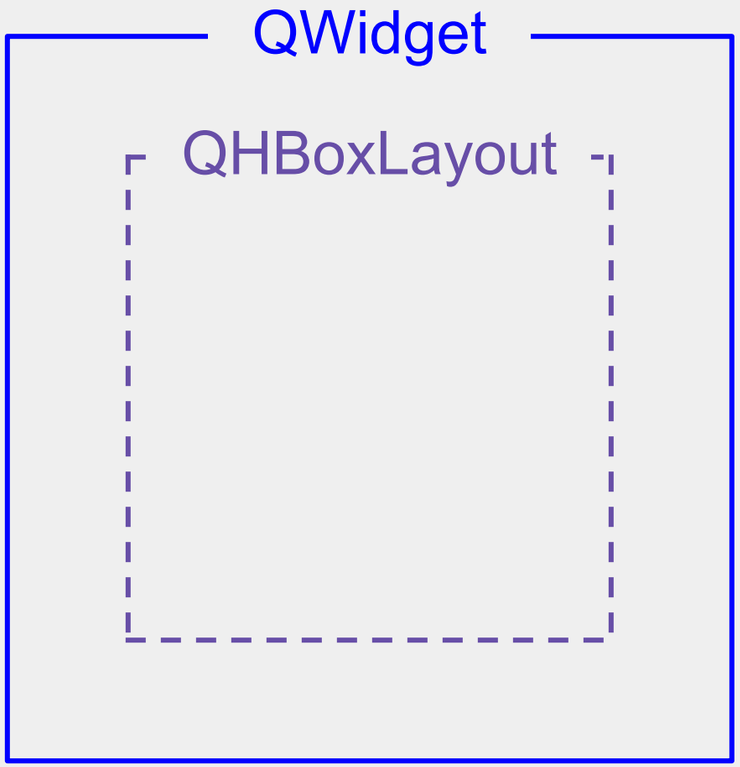
The overall structure of **MATWidget** would be…



It will contain three **MATBox** widgets

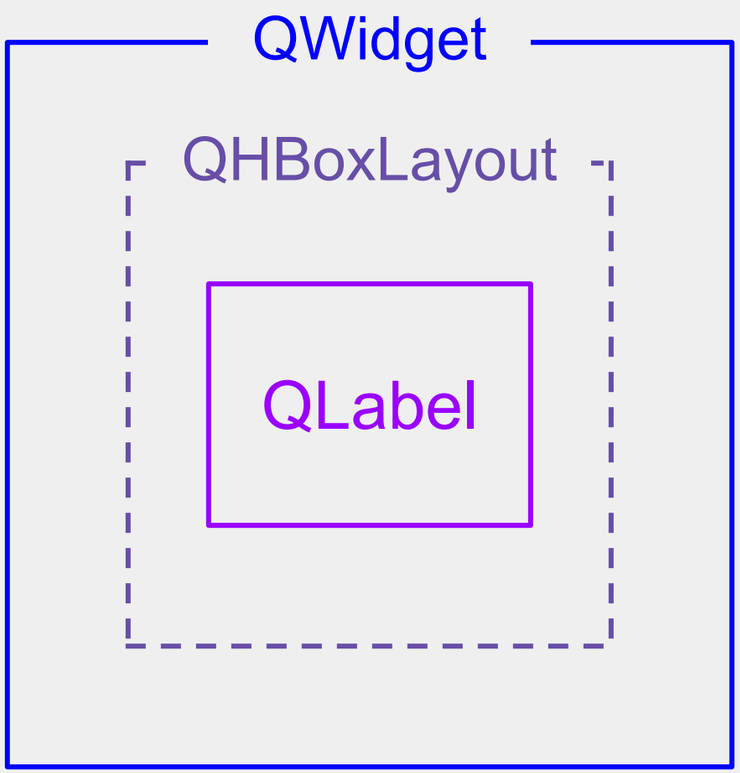


MATBox Architecture



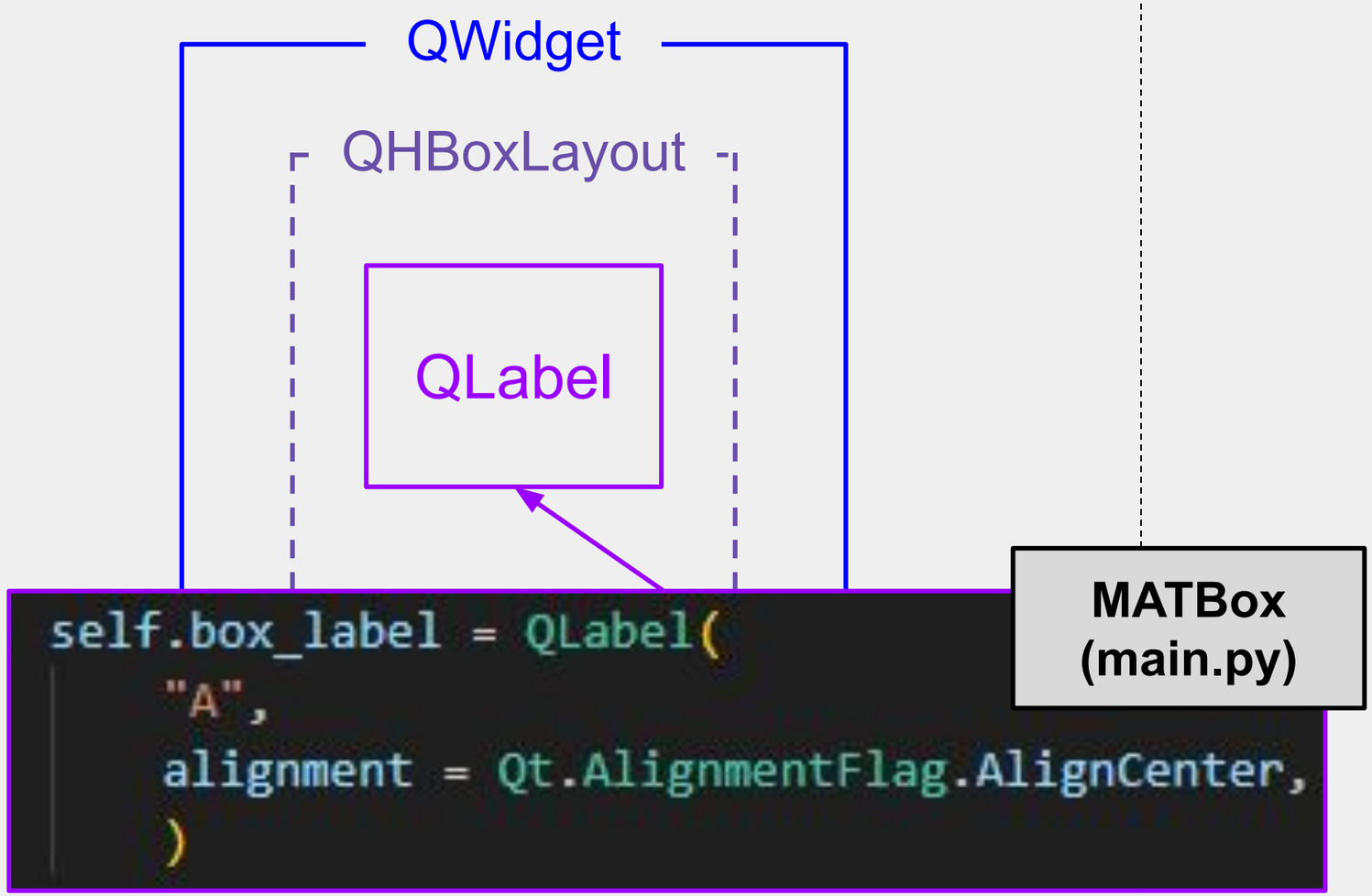
Result

MATBox Architecture



Result

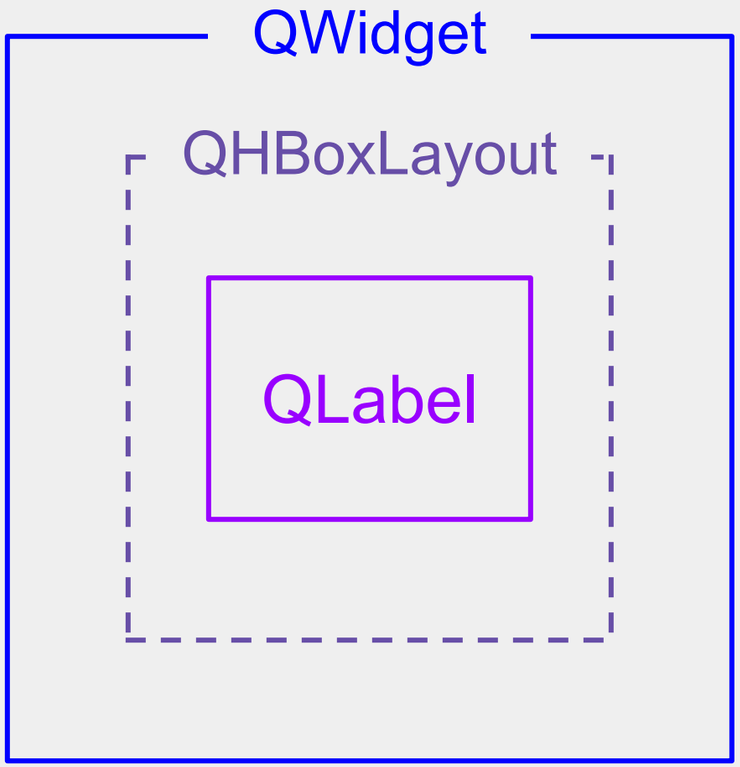
MATBox Architecture




```
self.box_label = QLabel(
    "A",
    alignment = Qt.AlignmentFlag.AlignCenter,
    )
```

Result

MATBox Architecture



Result

How can we change the color?

You can define UI styles with SCSS

(or CSS or QSS, if you hate yourself)

```scss
// Named colors
$c-black: rgb(0, 0, 0);
$c-gray: rgb(128,128,128);
$c-white: rgb(255, 255, 255);
$c-maroon: rgb(128,0,0);
$c-red: rgb(255, 0, 0);
$c-yellow: rgb(255, 255, 0);
$c-olive: rgb(128,128,0);
$c-green: rgb(0,128,0);
$c-lime: rgb(0, 255, 0);
$c-cyan: rgb(0,255,255);
$c-teal: rgb(0,128,128);
$c-navy: rgb(0,0,128);
$c-purple: rgb(128,0,128);
$c-pink: rgb(255,0,255);

// Alerts
$c-status-init: rgba($c-gray, 1);
$c-status-clear: rgba($c-lime, 1);
$c-status-warn: rgba($c-yellow, 1);
$c-status-assert: rgba($c-red, 1);
```

```scss
////////////////////////////////////////////////////////
// GLOBAL STYLES ///////////////////////////////////////
////////////////////////////////////////////////////////

* {
  padding: 0px;
  margin: 0px;
  border: 0px;
  border-style: none;
  border-image: none;
  outline: 0;
}

////////////////////////////////////////////////////////
// CUSTOM CLASSES //////////////////////////////////////
////////////////////////////////////////////////////////

.MainWindow {
  background-color: $c-white;
}
```

styles.scss

Here’s how we can apply an SCSS stylesheet

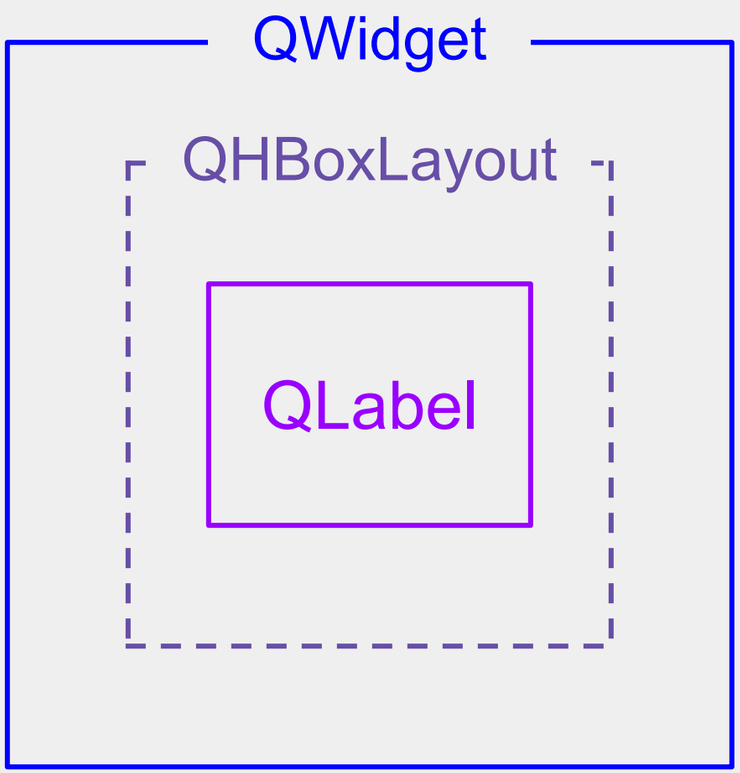
get_styles.py

```python
from pathlib import Path
import sass

def compile_styles():
    """Compile CSS styles from an SCSS file"""
    # Path to the style file
    styles_path = Path(__file__).parent / "static/scss/styles.scss"
    # String of CSS compiled from SCSS file
    css_content = sass.compile(filename=str(styles_path))
    return css_content
```

main.py

```python
# Start the app
app = QApplication(sys.argv)

# Set global stylesheet
style = compile_styles()
app.setStyleSheet(style)
```

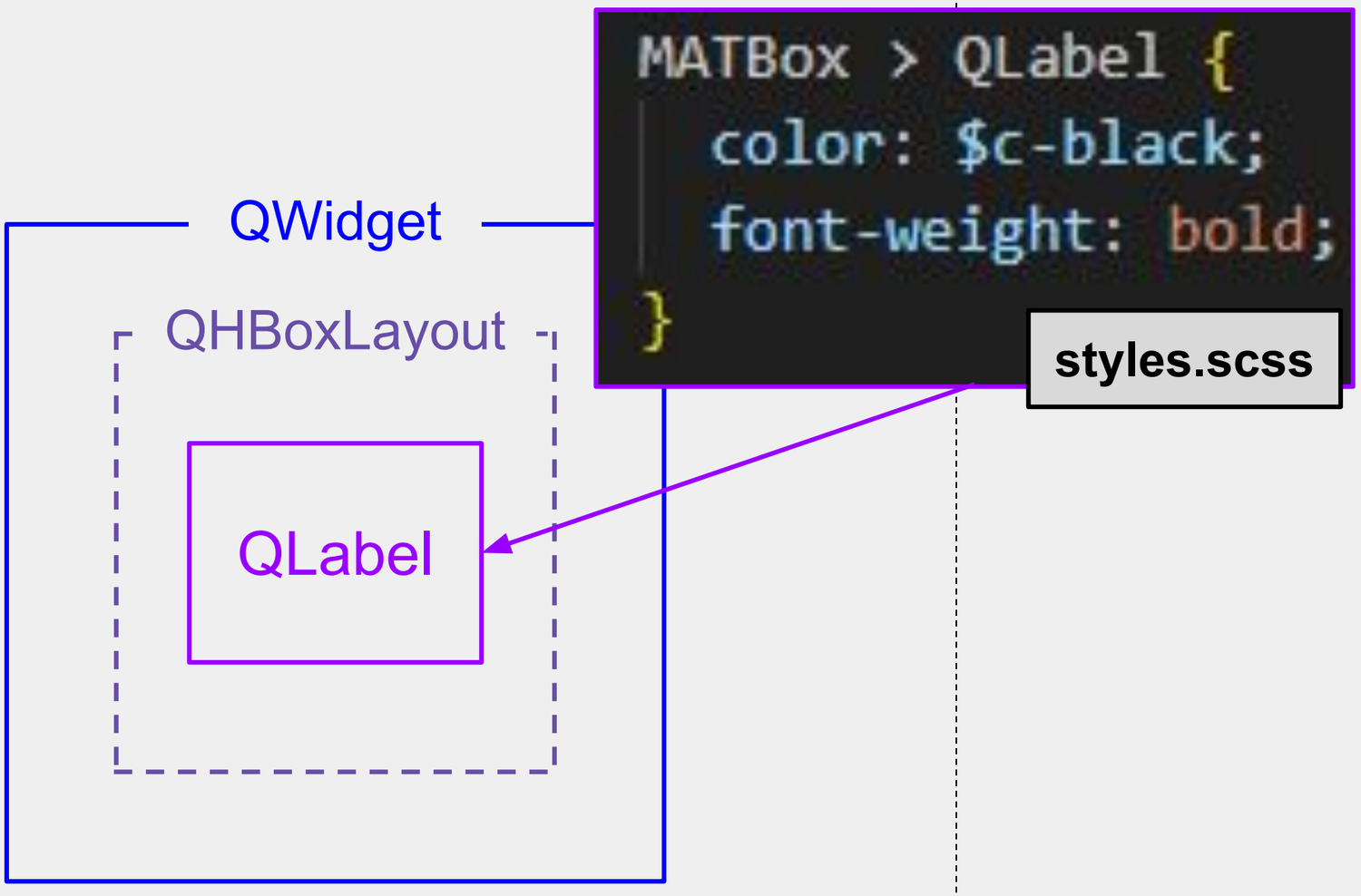
MATBox Architecture



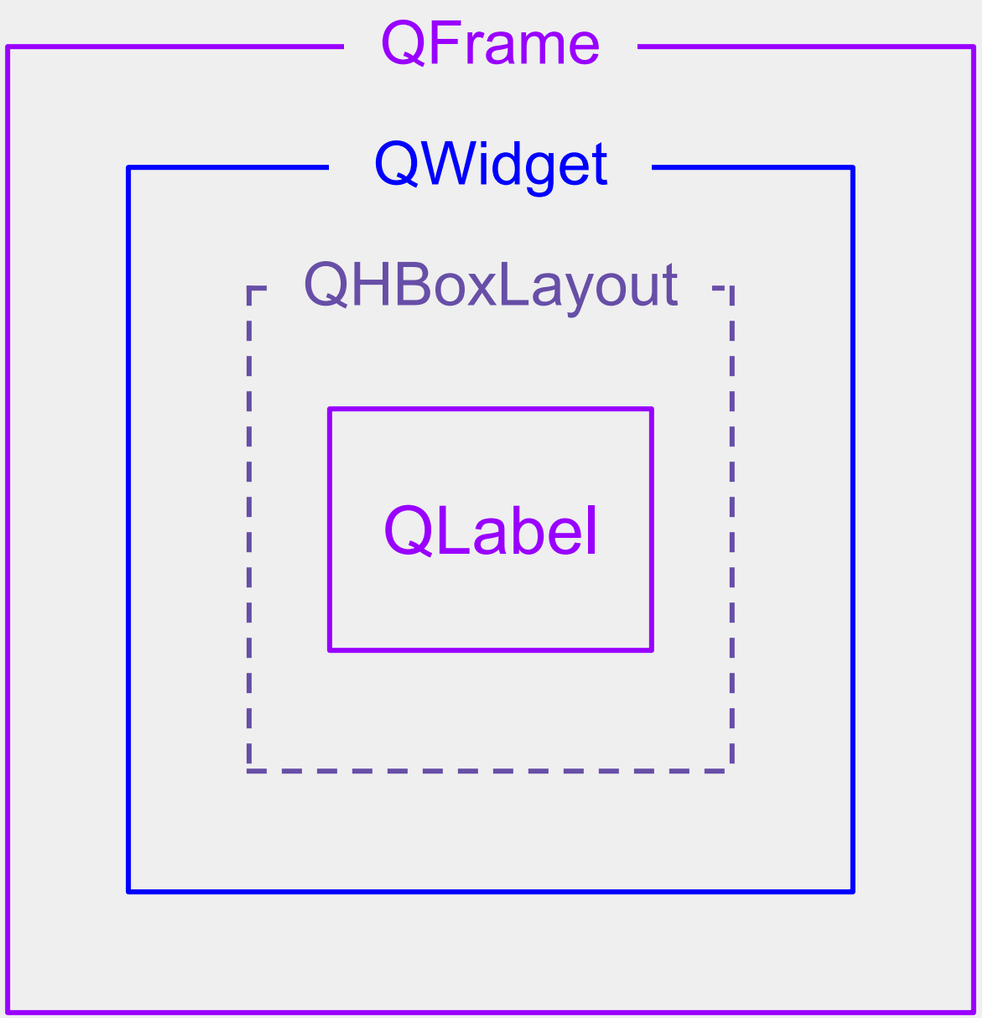
Result

MATBox Architecture

Result



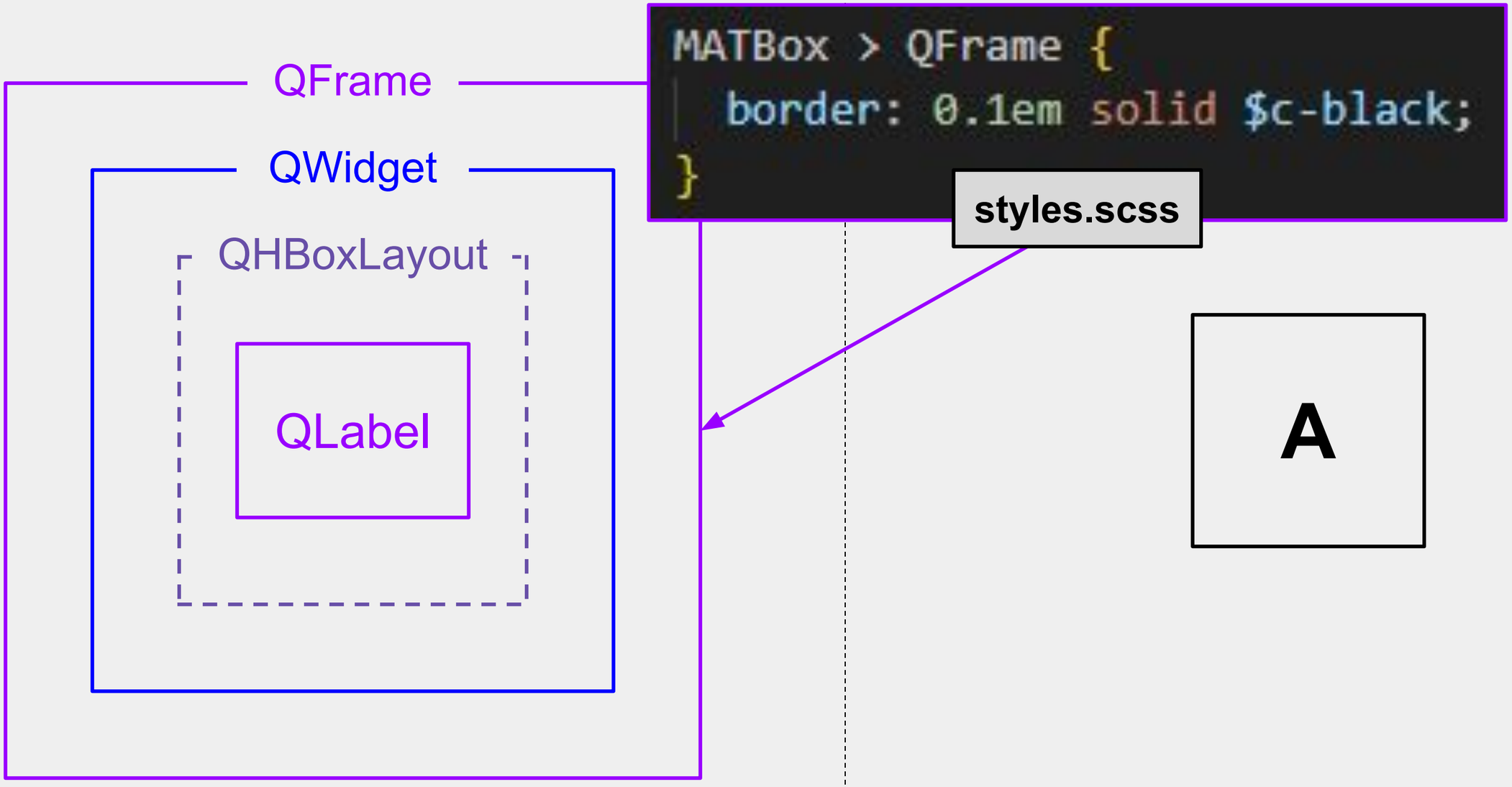
MATBox Architecture



Result

MATBox Architecture

Result



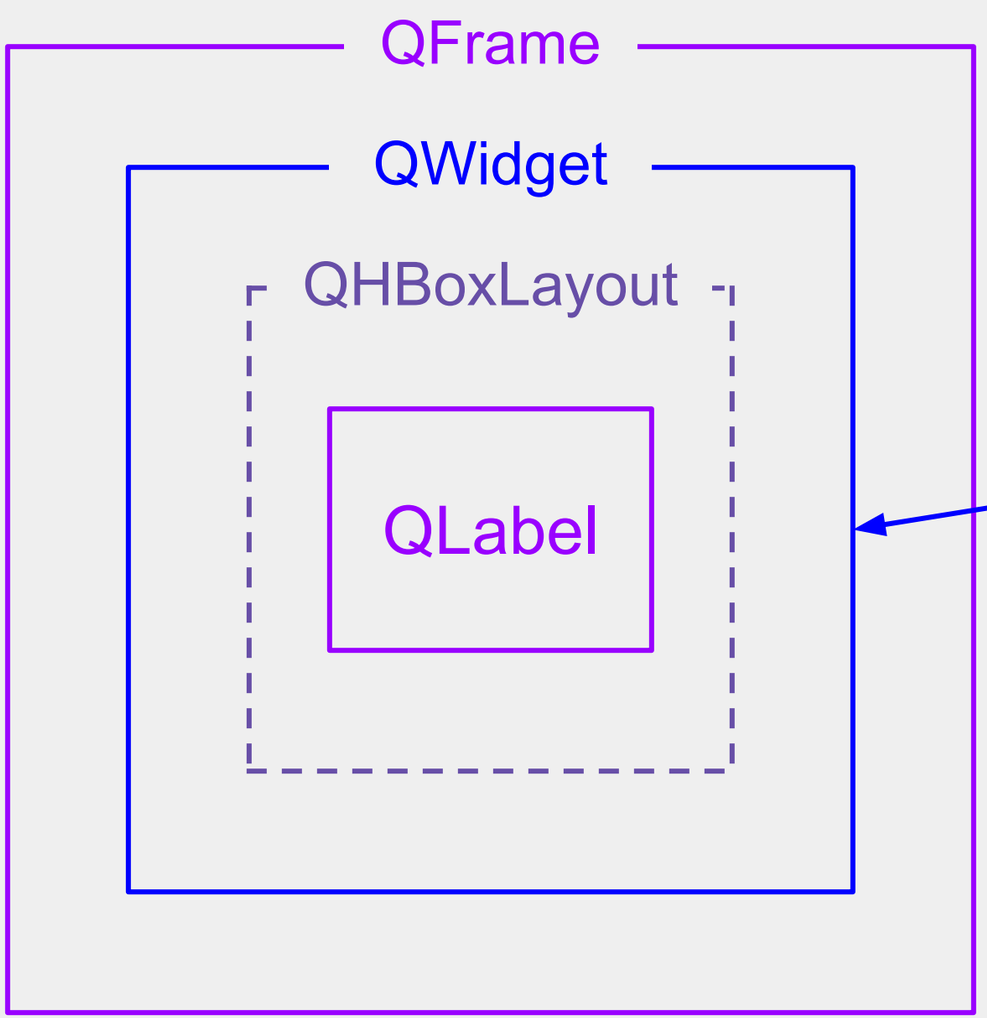
I made a **FakeDevice** class to connect to this demo:

```
Device "My fake device" has started


M: warn
A: clear
T: clear


M: assert
A: clear
T: clear
```

simulations.py (output)

It generates random MAT statuses every second

MATBox Architecture

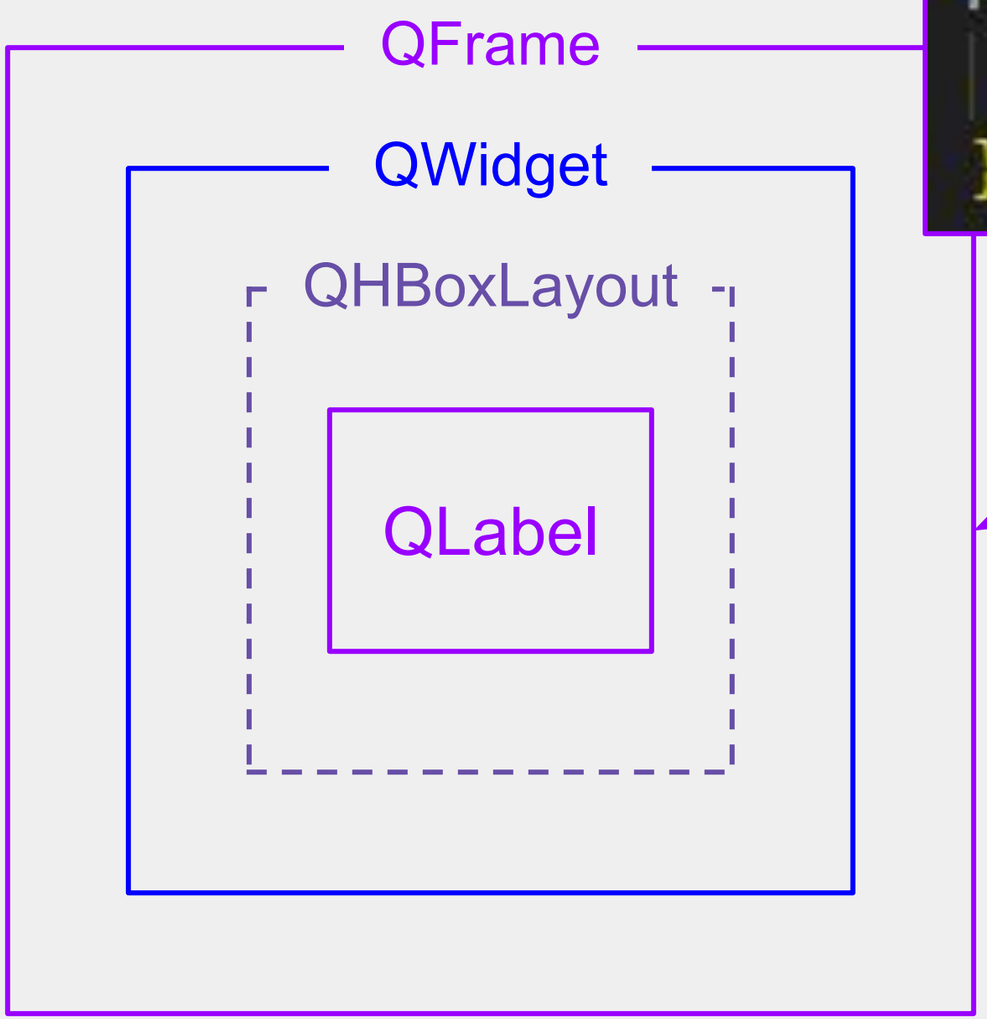


Result

MATBox (main.py)

```python
@property
def status(self):
    """Get the status"""
    return self._status

@status.setter
def status(self, value):
    """Set the status"""
    allowed_values = ["init", "clear", "warn", "assert"]
    if value not in allowed_values:
        raise ValueError(
            f"Status must be one of {allowed_values}"
            )
    self._status = value
    self.setProperty("status", self.status)
```

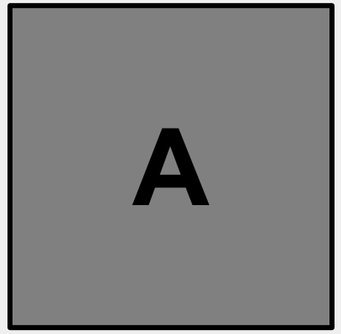
MATBox Architecture



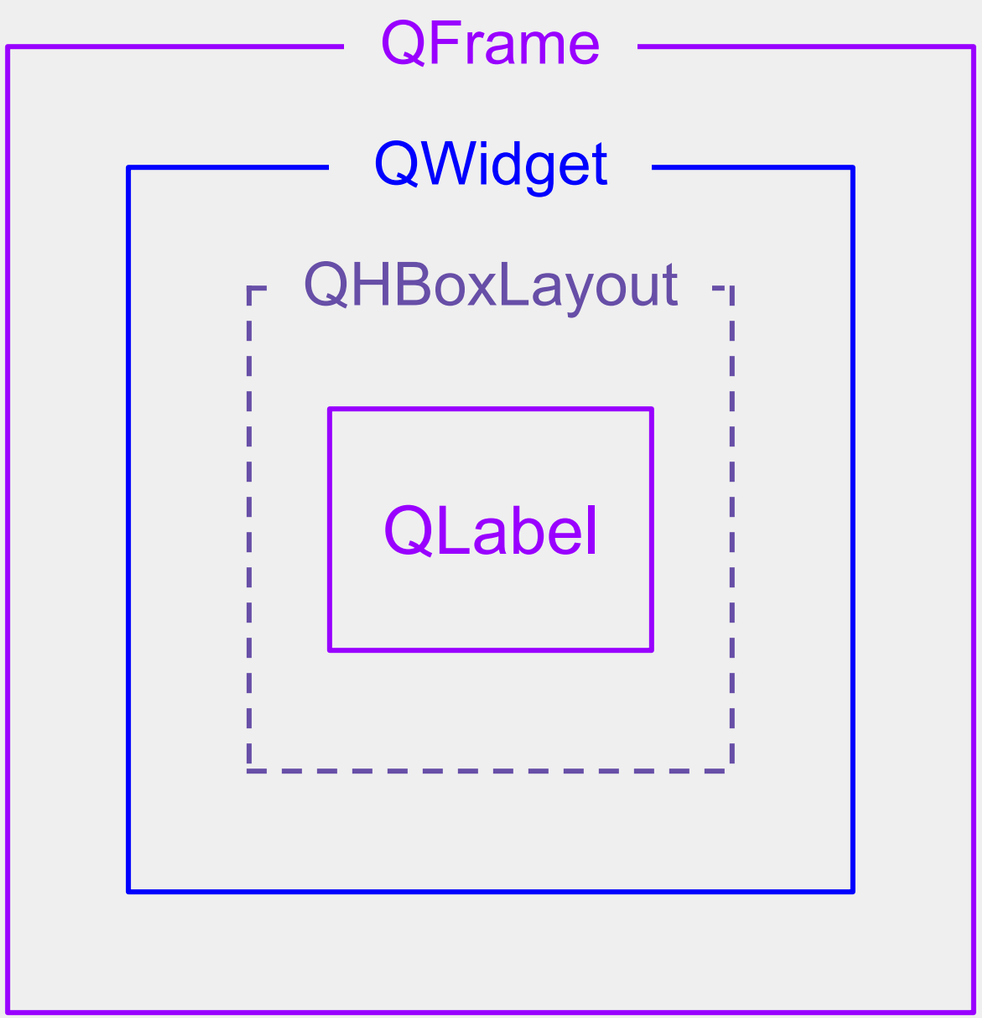
Result

```
MATBox[status="init"] > QFrame {
  background-color: $c-status-init;
}
```

styles.scss

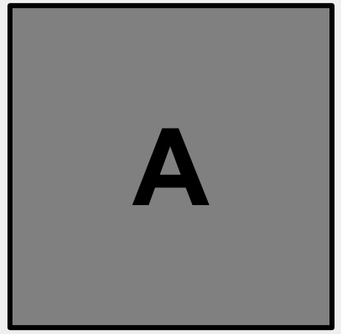


MATBox Architecture

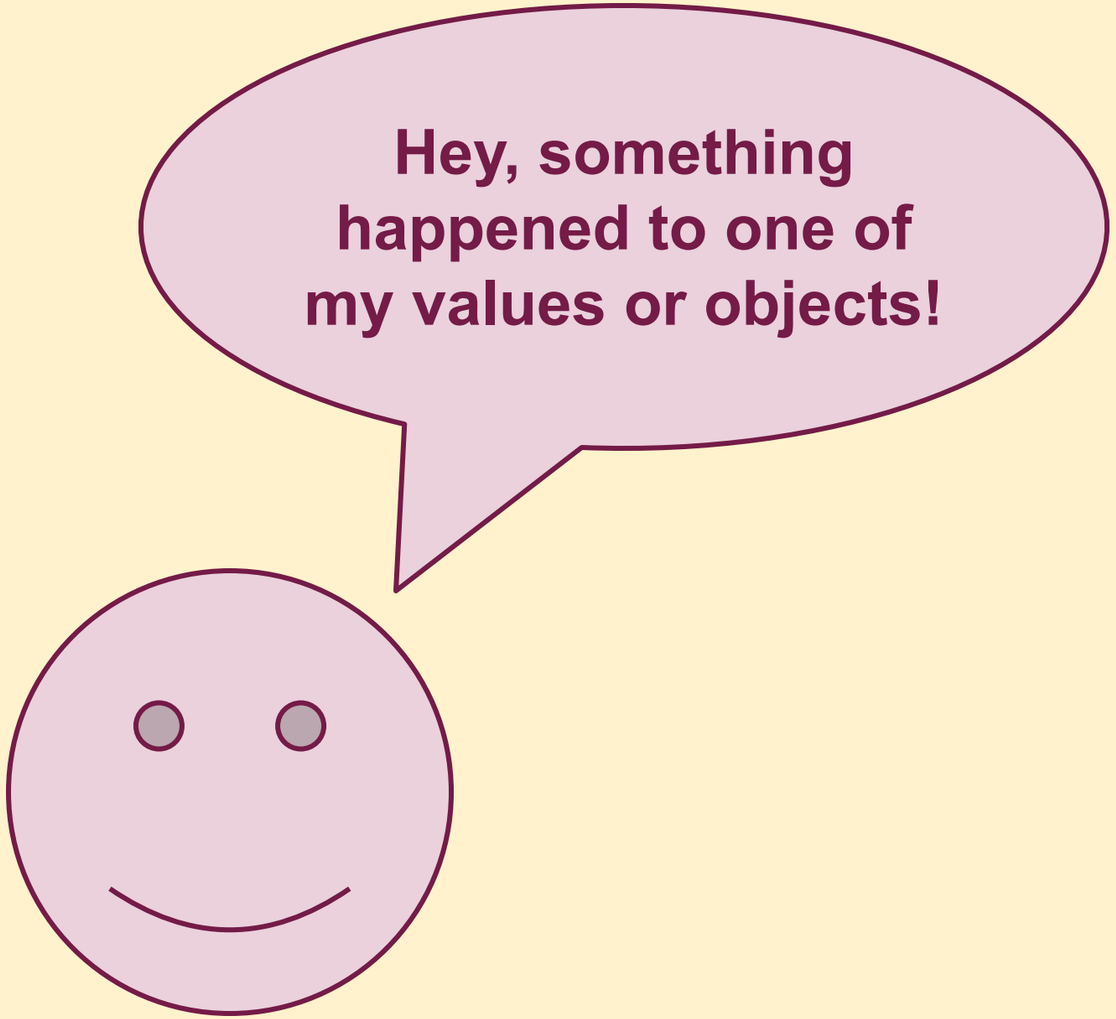


Result

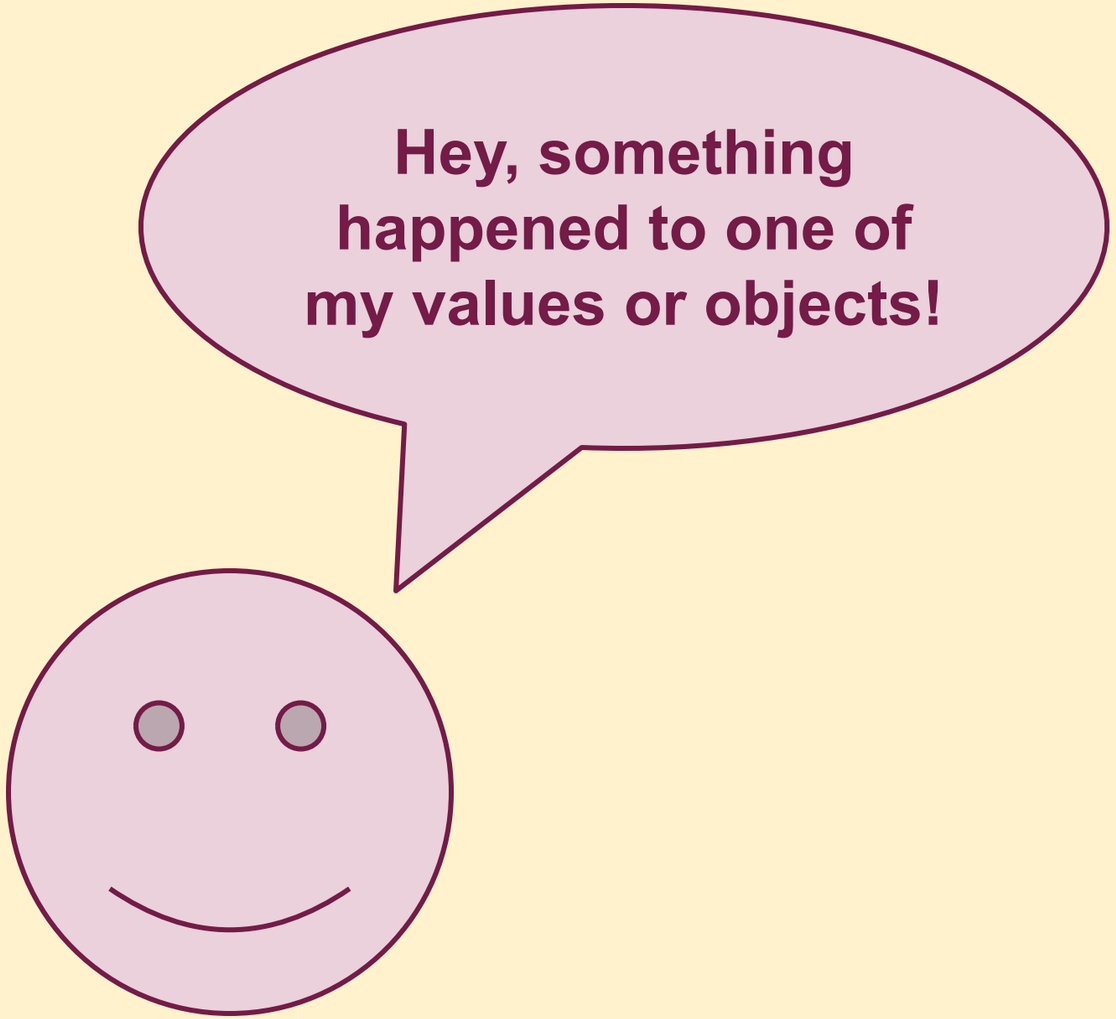
How can we DYNAMICALLY change the color based on "status"?



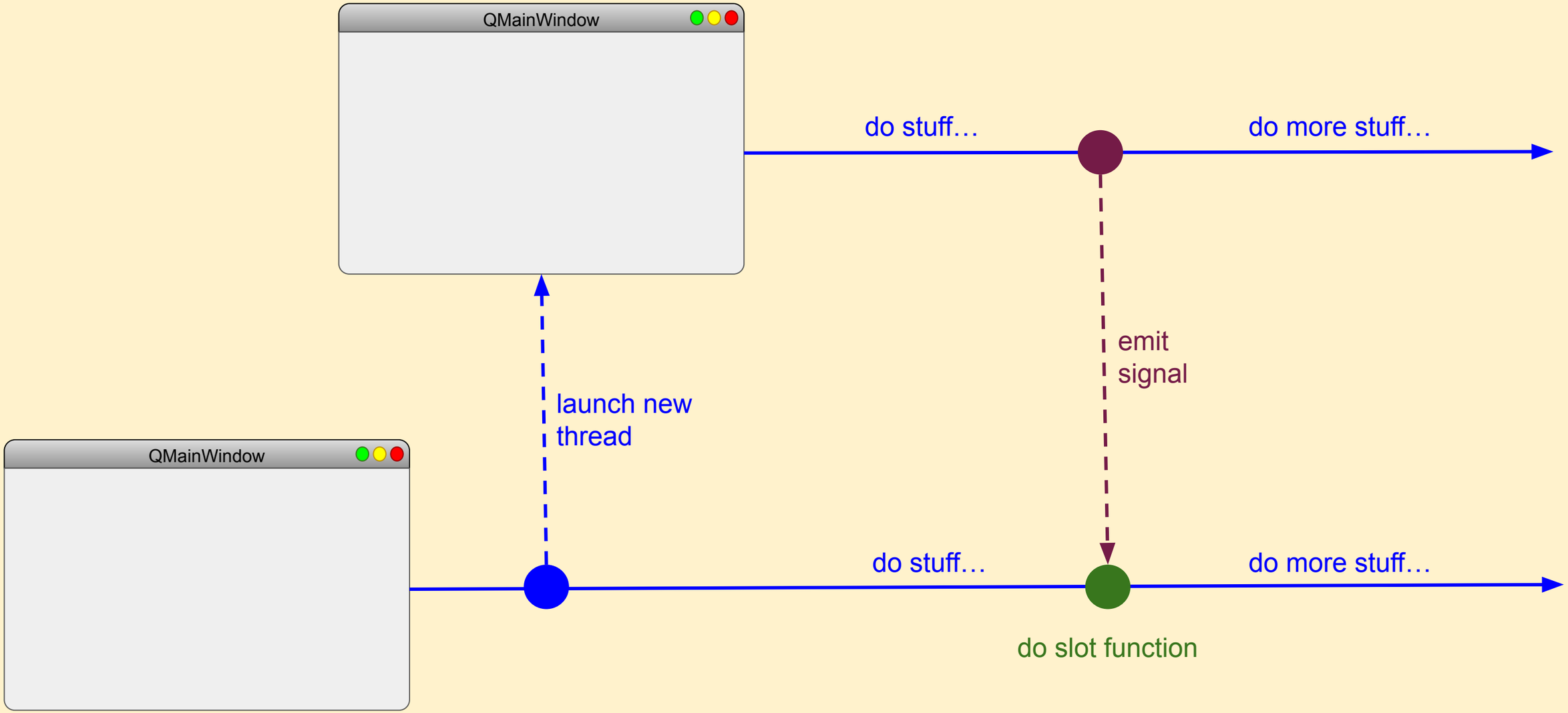
We're going to need **signals**



We're going to need **signals** and **slots**



Signals and **slots** are helpful when communicating information between **QThreads**



FakeDevice status *changes* are connected to **signals**

```python
class FakeDevice(QWidget):
    """Mimic simple device behavior for the MAT example"""
    # Define signals for this class
    managerChanged = pyqtSignal(str)
    accessorChanged = pyqtSignal(str)
    transporterChanged = pyqtSignal(str)
```

simulations.py

```python
@manager_status.setter
def manager_status(self, value):
    """Set the manager status"""
    if self.manager_status != value:
        self.managerChanged.emit(value)
    self._manager_status = value
```

Each **MATBox** has a **slot** to its appropriate **signal**

MATWidget (main.py)

```python
def _init_status_updates(self):
    """Connect value changes to actions"""
    self.device.managerChanged.connect(self.m_widget.get_status)
    self.device.accessorChanged.connect(self.a_widget.get_status)
    self.device.transporterChanged.connect(self.t_widget.get_status)
```

MATBox (main.py)

```python
def get_status(self):
    """Get the current device status"""
    if self.btype.lower() == "m":
        self.status = self._device.manager_status
    elif self.btype.lower() == "a":
        self.status = self._device.accessor_status
    elif self.btype.lower() == "t":
        self.status = self._device.transporter_status
    else:
        raise ValueError("btype must be 'm', 'a', or 't'")
    self.apply_styles()
```

Ideally, this alone could trigger a style change…

```
.MATBox[status="init"] > QFrame {
  background-color: $c-status-init;
}

.MATBox[status="clear"] > QFrame {
  background-color: $c-status-clear;
}

.MATBox[status="warn"] > QFrame {
  background-color: $c-status-warn;
}

.MATBox[status="assert"] > QFrame {
  background-color: $c-status-assert;
}
```

styles.scss

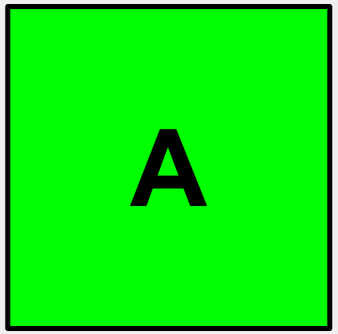
…*but* PyQt is weird about refreshing inherited styles, so I just use a clunky workaround

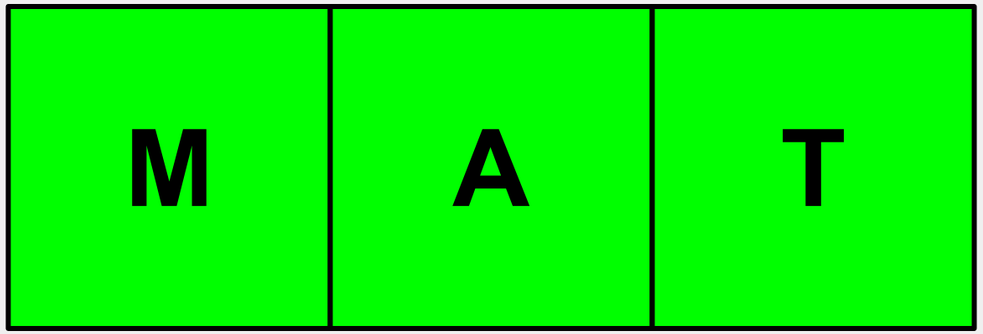
MATBox (main.py)

```python
def apply_styles(self):
    """Change the widget background color based on status"""
    if self.status == "init":
        self.setStyleSheet(f"background-color: {C_STATUS_INIT}")
    elif self.status == "clear":
        self.setStyleSheet(f"background-color: {C_STATUS_CLEAR}")
    elif self.status == "warn":
        self.setStyleSheet(f"background-color: {C_STATUS_WARN}")
    elif self.status == "assert":
        self.setStyleSheet(f"background-color: {C_STATUS_ASSERT}")
    else:
        raise ValueError(f"Unknown status value: {self.status}")
```

The colors now update properly!
Looks like we have some sizing issues, though…

The **MATBox** and **MATWidget** should have fixed aspect ratios





$$\frac{\text{width}}{\text{height}} = 1$$

$$\frac{\text{width}}{\text{height}} = 3$$

Let’s have our widgets inherit from a custom **AspectWidget** that does what we want

```python
class MATBox(AspectWidget):
    """...
    def __init__(self, device: FakeDevice, min_height_px=50, btype="M"):
        # Initialize MATBox with the properties of AspectWidget
        super().__init__(ratio=1)
```

main.py

```python
class MATWidget(AspectWidget):
    """...
    def __init__(self, min_height_px=50, padding=0):
        # Initialize MATWidget with the properties of AspectWidget
        super().__init__(ratio=3)
```

This **AspectWidget** extends the abilities of **QWidget**

```python
from PyQt5.QtWidgets import QWidget, QSizePolicy

class AspectWidget(QWidget):
    """A widget that maintains its aspect ratio."""
    def __init__(self, *args, ratio=1, **kwargs):
        super().__init__(*args, **kwargs)
        self.ratio = ratio
        self.adjusted_to_size = (-1, -1)
        self.setSizePolicy(QSizePolicy(QSizePolicy.Ignored, QSizePolicy.Ignored))

    def resizeEvent(self, event):
        size = event.size()
        if size == self.adjusted_to_size:
            # Avoid infinite recursion
            return
        self.adjusted_to_size = size

        full_width = size.width()
        full_height = size.height()
        width = min(full_width, full_height * self.ratio)
        height = min(full_height, full_width / self.ratio)

        h_margin = round((full_width - width) / 2)
        v_margin = round((full_height - height) / 2)

        self.setContentsMargins(h_margin, v_margin, h_margin, v_margin)
```

widgets.py

The squares are keeping the right shape now, but we don’t want them to grow with the window

We can set additional constraints on the sizing of the **MATWidget**

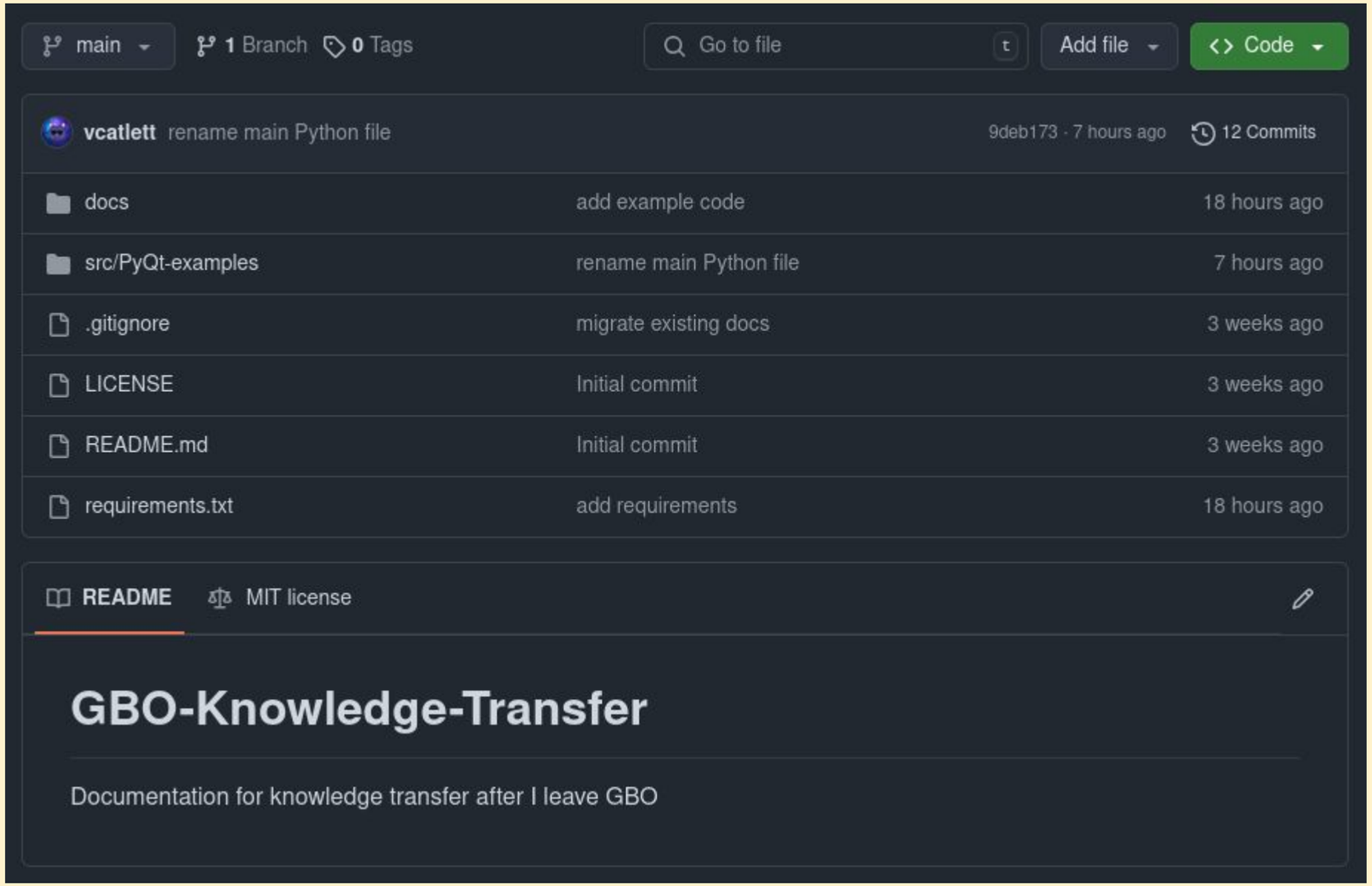
MATWidget (main.py)

```
def _init_styles(self):
    """Initialize the styles of the widget"""
    self.setMinimumSize(QSize(self.min_width_px, self.min_height_px))
    self.setSizePolicy(QSizePolicy.MinimumExpanding, QSizePolicy.Fixed)
```

All done!

This code is a part of my Knowledge Transfer repository

(coming VERY soon, but it’s still private)



This repo also contains my Knowledge Transfer docs

(see my demo on Friday!)

Search Ctrl + K

- Getting Started
- SDD Setup Instructions
- Python Packages
  - Sphinx
    - Sphinx Extensions
    - Sphinx Autobuilds
    - Figures and Diagrams
    - Docstrings
  - PyQt
- Glossary

Sphinx Extensions

These docs use an assortment of `sphinx` extensions. You can see them all in `docs/source/conf.py`. Here is a summary:

Sphinx Extensions

Name	Description
`sphinx-book-theme`	Primary HTML theming of the pages
`hoverxref`	Provides tooltips upon hovering
`myst_nb`	Text renderer that includes iPython notebook rendering
`numpydoc`	Auto-generate documentation on code with NumPy-style docstrings
`sphinx-copybutton`	Provides a “copy” button in code blocks
`sphinx-inline-tabs`	Capability to create tabbed items within a page
`sphinxcontrib-mermaid`	Include Mermaid diagrams in RST files
`sphinxemoji`	Easy inclusion of emojis in text 🥳

Previous Sphinx

Next Sphinx Autobuilds

that’s it

got any questions?