# Qt-Material

Yeison Cardona

Jun 17, 2025

# CONTENTS

Qt-Material is a modern stylesheet library for **PySide6** and **PyQt6**, inspired by Material Design.

It provides: - Dark and light themes - Custom accent colors and fonts - Runtime theme switching - Export to `.qss` + `.rcc` for use in C++ or standalone apps - Density scaling for accessibility

CHAPTER

ONE

# THEME PREVIEWS

Qt-Material includes a variety of built-in themes, both in dark and light modes.

**Dark themes:**

Fig. 1: dark

**Light themes:**

Fig. 2: light

CHAPTER

TWO

# INSTALLATION

```
pip install -U qt-material
```

Or from source:

```
git clone https://github.com/dunderlab/qt-material.git
cd qt-material
pip install .
```

CHAPTER

THREE

# DOCUMENTATION

Comprehensive usage guides, examples, and API reference are available online:

- Read the Docs – Official Documentation
- PyPI – Package Distribution Only

CHAPTER

FOUR

# BASIC USAGE

To apply a Material Design-inspired stylesheet to your Qt application, simply use `apply_stylesheet()` from the `qt_material` module.

## 4.1 Example using PySide6

```
import sys
from PySide6 import QtWidgets
from qt_material import apply_stylesheet

# Create application instance and main window
app = QtWidgets.QApplication(sys.argv)
window = QtWidgets.QMainWindow()

# Apply a built-in theme
apply_stylesheet(app, theme='dark_teal.xml')

# Show the window and execute
window.show()
app.exec()
```

## 4.2 Optional: Using PyQt5 or PyQt6

Just replace the import line:

```
from PyQt5 import QtWidgets
# or
from PyQt6 import QtWidgets
```

CHAPTER

# FIVE

# THEMES

To view all available built-in themes, use the `list_themes()` function:

```
from qt_material import list_themes

print(list_themes())
```

Example output:

```
['dark_amber.xml',
 'dark_blue.xml',
 'dark_cyan.xml',
 'dark_lightgreen.xml',
 'dark_pink.xml',
 'dark_purple.xml',
 'dark_red.xml',
 'dark_teal.xml',
 'dark_yellow.xml',
 'light_amber.xml',
 'light_blue.xml',
 'light_cyan.xml',
 'light_cyan_500.xml',
 'light_lightgreen.xml',
 'light_pink.xml',
 'light_purple.xml',
 'light_red.xml',
 'light_teal.xml',
 'light_yellow.xml']
```

To apply any of these themes, pass the filename to `apply_stylesheet()`:

```
apply_stylesheet(app, theme='light_purple.xml')
```

CHAPTER

SIX

# LIGHT THEMES

When using a light theme, it's recommended to enable `invert_secondary=True` to ensure text and contrast are properly rendered for bright backgrounds.

```
from qt_material import apply_stylesheet

apply_stylesheet(app, theme='light_red.xml', invert_secondary=True)
```

This helps maintain proper visibility and icon behavior in light mode.

CHAPTER

SEVEN

# ENVIRONMENT VARIABLES

The following environment variables are related to the current theme used. These variables are **for consultation purposes only**.

| Environment Variable | Description | Example |
|---|---|---|
| QTMATERIAL_PRIMARYCOLOR | Primary color | #2979ff |
| QTMA TERIAL_PRIMARYLIGHTCOLOR | A bright version of the primary color | #75a7ff |
| Q TMATERIAL_SECONDARYCOLOR | Secondary color | #f5f5f5 |
| QTMATE RIAL_SECONDARYLIGHTCOLOR | A bright version of the secondary color | #ffffff |
| QTMAT ERIAL_SECONDARYDARKCOLOR | A dark version of the secondary color | #e6e6e6 |
| QTM ATERIAL_PRIMARYTEXTCOLOR | Color for text over primary background | #000000 |
| QTMAT ERIAL_SECONDARYTEXTCOLOR | Color for text over secondary background | #000000 |
| QTMATERIAL_THEME | Name of theme used | light _blue.xml |

CHAPTER

EIGHT

# RUN EXAMPLES

A window with almost all widgets (see the previous screenshots) is available to test all themes and **create new ones**.

```
git clone https://github.com/UN-GCPDS/qt-material.git
cd qt-material
python setup.py install
cd examples/full_features
python main.py --pyside6
```

This will launch a live preview application where you can:

- Browse all available themes
- Switch stylesheets at runtime
- Customize fonts and colors
- Export your personalized theme

Fig. 1: theme

CHAPTER

NINE

# TROUBLESHOOTS

## 9.1 QMenu

`QMenu` may render differently depending on the Qt backend (PySide6, PyQt6), platform, or even the style engine (e.g. Fusion). This can affect spacing, height, and padding.

To improve appearance or fix spacing issues, you can manually define `QMenu` settings using the `extra` argument:

```
extra = {
    'QMenu': {
        'height': 50,
        'padding': '10px 20px 10px 20px'  # top, right, bottom, left
    }
}
```

This customization is independent of the `density_scale` setting and can be used to ensure consistent appearance across platforms.

CHAPTER

# TEN

# DOCUMENTATION OVERVIEW

## 10.1 Customization Guide

This notebook demonstrates how to customize Qt-Material themes, including color schemes, font styles, and advanced stylesheet modifications.

### 10.1.1 Theme Customization

To create a custom theme, edit the color values directly in an XML file with the following format:

[ ]:
```
<!--?xml version="1.0" encoding="UTF-8"?-->
<resources>
<color name="primaryColor">#00e5ff</color>
<color name="primaryLightColor">#6effff</color>
<color name="secondaryColor">#f5f5f5</color>
<color name="secondaryLightColor">#ffffff</color>
<color name="secondaryDarkColor">#e6e6e6</color>
<color name="primaryTextColor">#000000</color>
<color name="secondaryTextColor">#000000</color>
</resources>
```

Save it as `my_theme.xml` or similar and apply the style sheet from Python.

[ ]:
```
apply_stylesheet(app, theme='dark_teal.xml')
```

### 10.1.2 Custom QPushButton Styles and Fonts

You can customize the appearance of `QPushButton` widgets by using the `extra` argument when applying the stylesheet. This allows you to define custom colors and fonts.

[ ]:
```
extra = {
    # Button colors
    'danger': '#dc3545',
    'warning': '#ffc107',
    'success': '#17a2b8',
```

(continues on next page)

(continued from previous page)

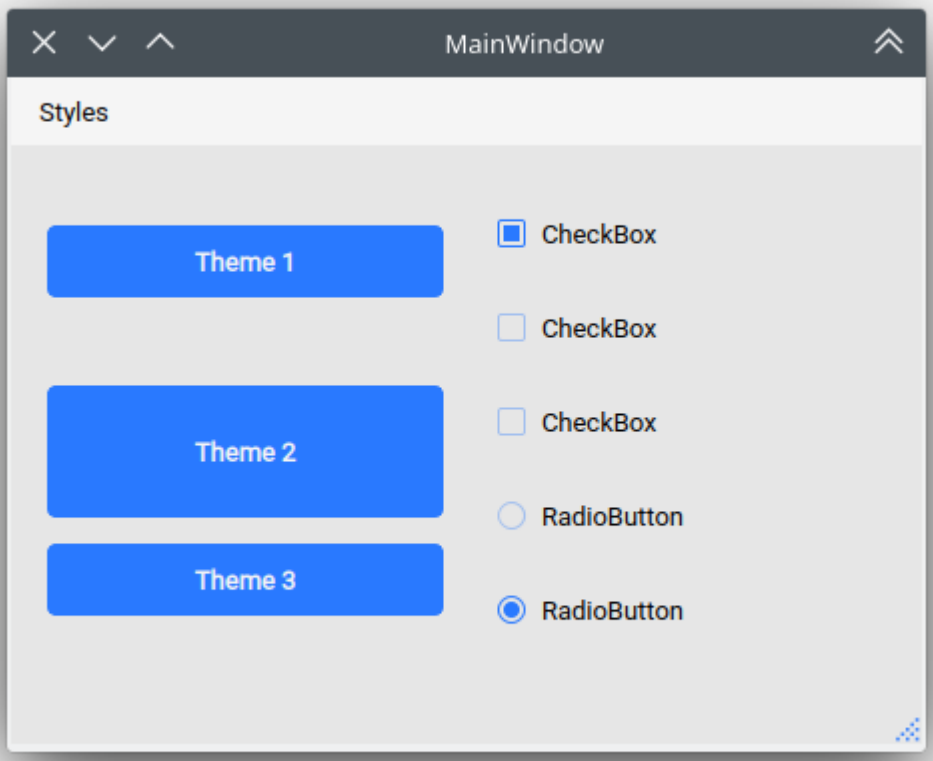
```
    # Font
    'font_family': 'Roboto',
}

apply_stylesheet(app, 'light_cyan.xml', invert_secondary=True, extra=extra)
```

To apply a specific style to a button, use the `setProperty()` method with the appropriate class name:

```
[ ]: pushButton_danger.setProperty('class', 'danger')
pushButton_warning.setProperty('class', 'warning')
pushButton_success.setProperty('class', 'success')
```

This will apply the color defined for each class in the stylesheet.



### 10.1.3 Custom stylesheets

You can override or extend the default Qt-Material styles using an external `.css` file.

Example `custom.css` content:

[ ]:
```
QPushButton {{
  color: {QTMATERIAL_SECONDARYCOLOR};
  text-transform: none;
  background-color: {QTMATERIAL_PRIMARYCOLOR};
}}

.big_button {{
  height: 64px;
}}
```

#### Apply the stylesheet:

[ ]:
```
apply_stylesheet(app, theme='light_blue.xml', css_file='custom.css')
```

#### Runtime update:

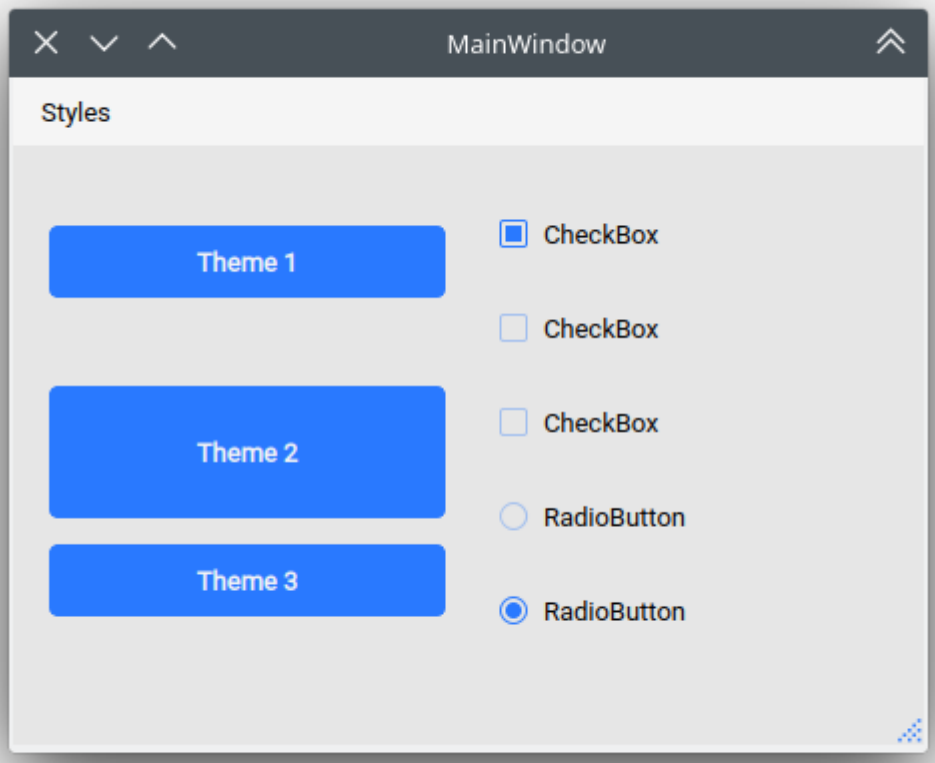
You can also apply a custom stylesheet dynamically at runtime:

[ ]:
```
stylesheet = app.styleSheet()
with open('custom.css') as file:
    app.setStyleSheet(stylesheet + file.read().format(**os.environ))
```

To apply a class-based style:

[ ]:
```
self.main.pushButton.setProperty('class', 'big_button')
```

This allows you to modularly apply different layout or visual rules without modifying the core theme.




### 10.1.4 Remove theme from single widget

In some cases, you might want to exclude a specific widget from the global theme. You can reset the style for that widget using:

```
widget.setStyleSheet('')
```

This removes all inherited styling, making the widget fallback to the system or Qt default appearance. Useful for custom painting or embedded platform-native controls.

### 10.1.5 Density scale

The `extra` argument also includes an option to control the **density scale**, which affects the spacing and sizing of UI elements. By default, the scale is set to `0`.

You can reduce or increase density using negative or positive values:

```
[ ]: extra = {
         'density_scale': '-2',  # Lower density (more compact layout)
     }
```

(continues on next page)

(continued from previous page)

```
apply_stylesheet(app, 'default', invert_secondary=False, extra=extra)
```

This setting is helpful for adapting your UI to different screen sizes or accessibility preferences.

## 10.2 Runtime Features

This notebook demonstrates how to change themes dynamically, integrate theme selection menus, and use the dock interface to create and export new themes at runtime.

### 10.2.1 Change theme in runtime

To enable dynamic theme switching at runtime, inherit from the `QtStyleTools` class alongside your main window. This gives access to the `apply_stylesheet()` method, which can be used to update the UI on the fly.

[ ]:
```
from qt_material import QtStyleTools, apply_stylesheet
from PySide6.QtWidgets import QMainWindow
from PySide6.QtUiTools import QUiLoader

class RuntimeStylesheets(QMainWindow, QtStyleTools):
    def __init__(self):
        super().__init__()
        self.main = QUiLoader().load('main_window.ui', self)

        # Apply a theme at startup
        self.apply_stylesheet(self.main, 'dark_teal.xml')

        # Other theme options
        # self.apply_stylesheet(self.main, 'light_red.xml')
        # self.apply_stylesheet(self.main, 'light_blue.xml')
```

This approach allows your application to switch themes without restarting the interface.

### 10.2.2 Integrate stylesheets in a menu

You can add a built-in menu to your application that lists and applies all available themes dynamically.

To do this, use the `add_menu_theme()` method provided by `QtStyleTools`. This method attaches theme options to an existing `QMenu` object in your UI.

[ ]:
```
from qt_material import QtStyleTools
from PySide6.QtWidgets import QMainWindow
from PySide6.QtUiTools import QUiLoader

class RuntimeStylesheets(QMainWindow, QtStyleTools):
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        super().__init__()
        self.main = QUiLoader().load('main_window.ui', self)

        # Connect the menuStyles QMenu to the dynamic theme switcher
        self.add_menu_theme(self.main, self.main.menuStyles)
```

This will populate `menuStyles` with all available `.xml` themes, and automatically apply the selected theme when clicked.

### 10.2.3 Create new themes

A built-in dock interface is available for customizing the current theme at runtime. This tool allows you to adjust colors, density, and other visual parameters interactively.

When you use this interface, a new theme file is generated and saved as `my_theme.xml` in the current working directory.

To enable it, call `show_dock_theme()`:

[ ]:
```
from qt_material import QtStyleTools
from PySide6.QtWidgets import QMainWindow
from PySide6.QtUiTools import QUiLoader

class RuntimeStylesheets(QMainWindow, QtStyleTools):
    def __init__(self):
        super().__init__()
        self.main = QUiLoader().load('main_window.ui', self)

        # Show the dock to create and export themes interactively
        self.show_dock_theme(self.main)
```

This feature is ideal for visually designing themes and exporting them without manually editing XML files.

## 10.3 Export Guide

This notebook demonstrates how to export a Qt-Material theme into local `.qss` and `.rcc` files for use in Python or C++ applications, even without needing the Python package at runtime.

### 10.3.1 Export theme

You can export a complete theme including styles and resources using the `export_theme()` function:

[ ]:
```
from qt_material import export_theme

extra = {
    # Button colors
    'danger': '#dc3545',
    'warning': '#ffc107',
    'success': '#17a2b8',

    # Font
    'font_family': 'monoespace',
    'font_size': '13px',
    'line_height': '13px',

    # Density Scale
    'density_scale': '0',

    # Environment hints
    'pyside6': True,
    'linux': True,

}

export_theme(theme='dark_teal.xml',
             qss='dark_teal.qss',
             rcc='resources.rcc',
             output='theme',
             prefix='icon:/',
             invert_secondary=False,
             extra=extra,
            )
```

This command will generate:

- `dark_teal.qss`: the compiled stylesheet
- `resources.rcc`: the icon archive
- A folder named `theme/` with all required icons

### 10.3.2 Using the exported theme in PySide6

[ ]:
```
import sys
from PySide6 import QtWidgets
from PySide6.QtCore import QDir

app = QtWidgets.QApplication(sys.argv)

# Load stylesheet
with open('dark_teal.qss', 'r') as file:
    app.setStyleSheet(file.read())
```

(continues on next page)

(continued from previous page)

```
# Register icon prefix
QDir.addSearchPath('icon', 'theme')

# Simple test window
window = QtWidgets.QMainWindow()
checkbox = QtWidgets.QCheckBox(window)
checkbox.setText('CheckBox')
window.show()
app.exec()
```

### 10.3.3 Cross-language support

The exported `.qss` and `.rcc` files can also be integrated into non-Python projects, such as C++/Qt applications, without requiring `qt-material` or Python at all.

### 10.3.4 New themes

Do you have a custom theme? Does it look great? Feel free to contribute! Create a pull request in the themes folder and share it with the community.