Lecture 14: PyQT (2)

QAction

- Qt menu support display of text, and can also contain any standard Qt widget. However, for buttons the best approach is to make use of the QAction system to place buttons on the menu.

- QAction is a class that provides a way to describe abstract user interfaces. What this means in English, is that you can define multiple interface elements within a single object, unified by the effect that interacting with that element has.

QAction (2)

- Without QAction you would have to define this in multiple places. But with QAction you can define a single QAction, defining the triggered action, and then add this action to both the menu and the toolbar. Each QAction has names, status messages, icons and signals that you can connect to (and much more).

QAction (3)

- Signal:
  - triggered: This signal is emitted when an action is activated by the user; for example, when the user clicks a menu option, toolbar button, or presses an action's shortcut key combination

QAction (4)

- toolbar = QToolBar("My main toolbar")
- self.addToolBar(toolbar)
- button_action = QAction("Your button", self)
- button_action.setStatusTip("This is your button")
- button_action.triggered.connect(self.onMyToolBarButtonClick)
- toolbar.addAction(button_action)
- def onMyToolBarButtonClick(self, s):
  - print("click", s)

QDialog

- Dialogs are useful GUI components that allow you to communicate with the user (hence the name dialog). They are commonly used for file Open/Save, settings, preferences, or for functions that do not fit into the main UI of the application.
- In Qt dialog boxes are handled by the QDialog class. To create a new dialog box simply create a new object of QDialog type passing in a parent widget, e.g. QMainWindow, as its parent

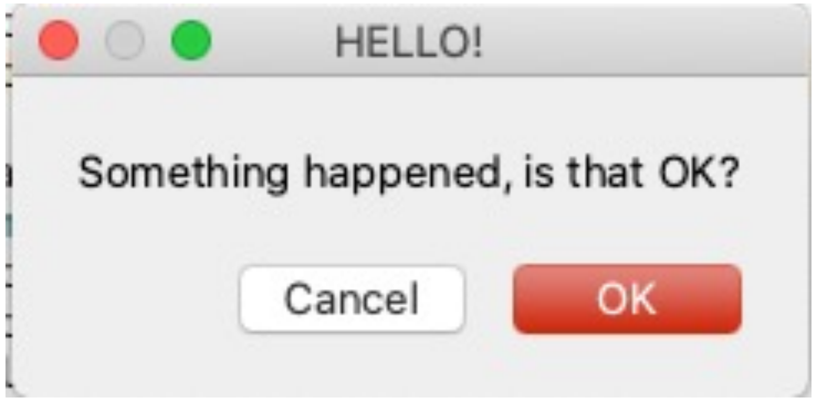
QDialog (2)

- button = QPushButton("Press me for a dialog!")
- button.clicked.connect(self.button_clicked)
- self.setCentralWidget(button)
- def button_clicked(self, s):
  - dlg = QDialog(self)
  - dlg.setWindowTitle("HELLO!")
  - dlg.exec ()

QDialog (3)

- class CustomDialog(QDialog):

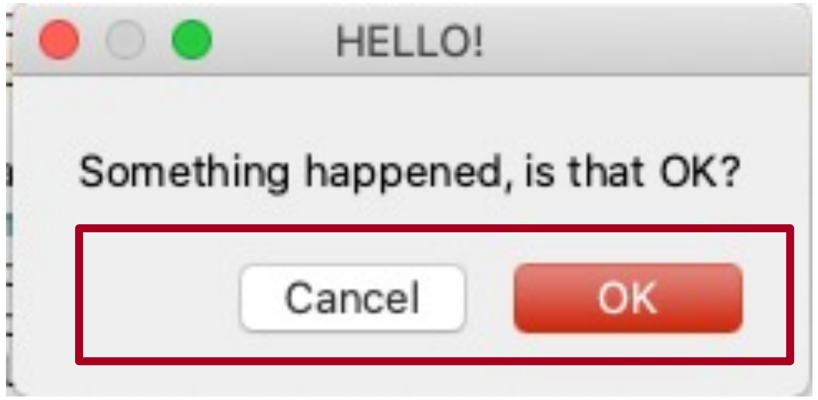
```
    def __init__(self, parent=None):
        super().__init__(parent)
        self.setWindowTitle("HELLO!")
        QBtn = QDialogButtonBox.StandardButton.Ok |
QDialogButtonBox.StandardButton.Cancel
        self.buttonBox = QDialogButtonBox(QBtn)
        self.buttonBox.accepted.connect(self.accept)
        self.buttonBox.rejected.connect(self.reject)
```


HELLO!

Something happened, is that OK?

QDialog (4)

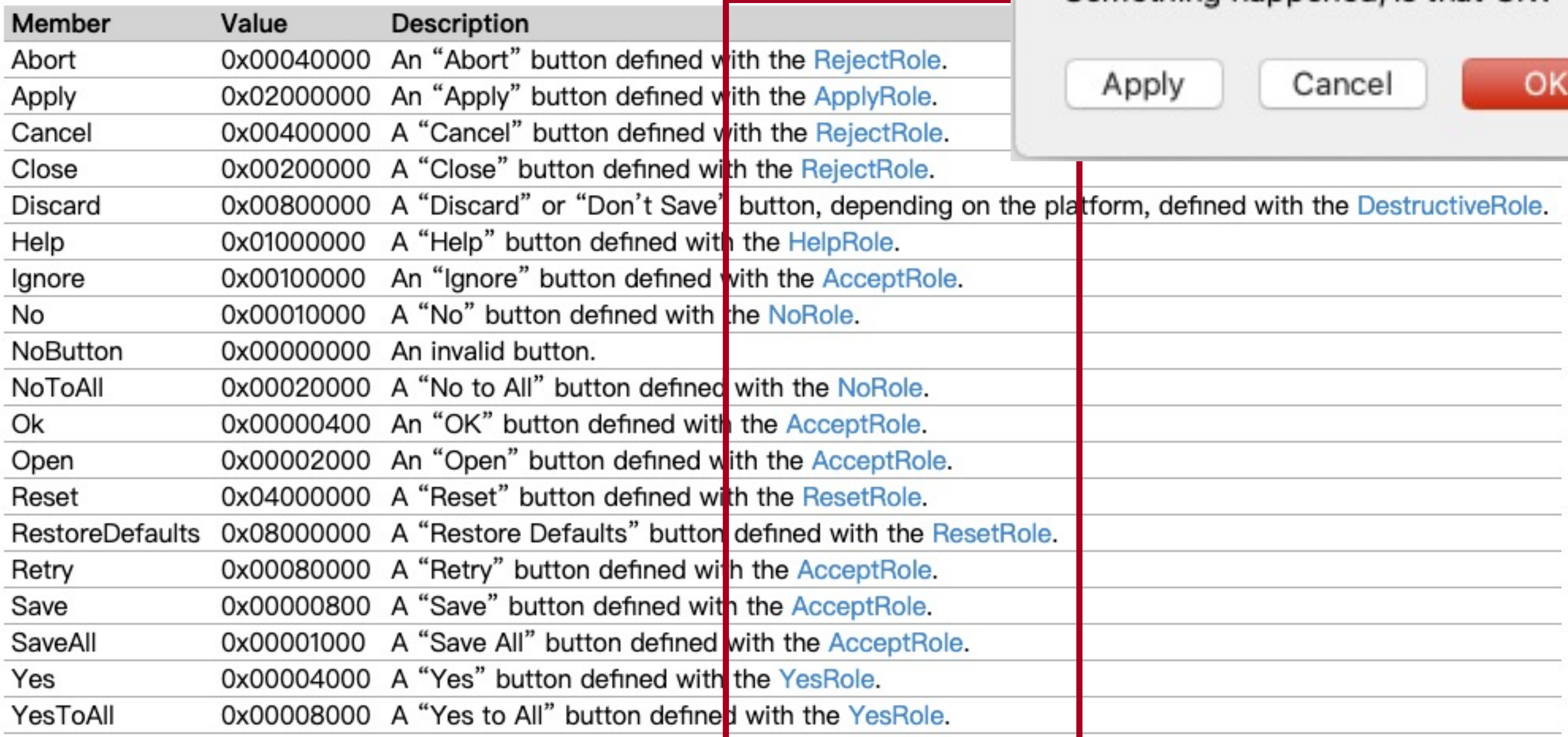
- QBtn = QDialogButtonBox.StandardButton.Ok | QDialogButtonBox.StandardButton.Cancel



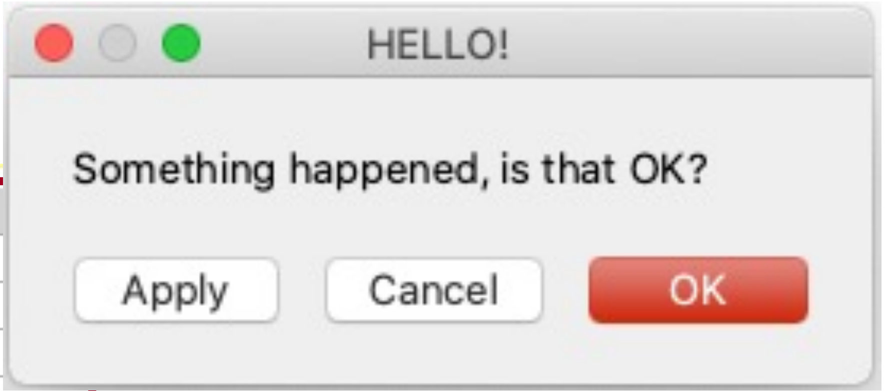
Something happened, is that OK?

- StandardButton List https://www.riverbankcomputing.com/static/Docs/PyQt6/api/qtwidgets/qdialogbuttonbox.html#

QDialog (5)



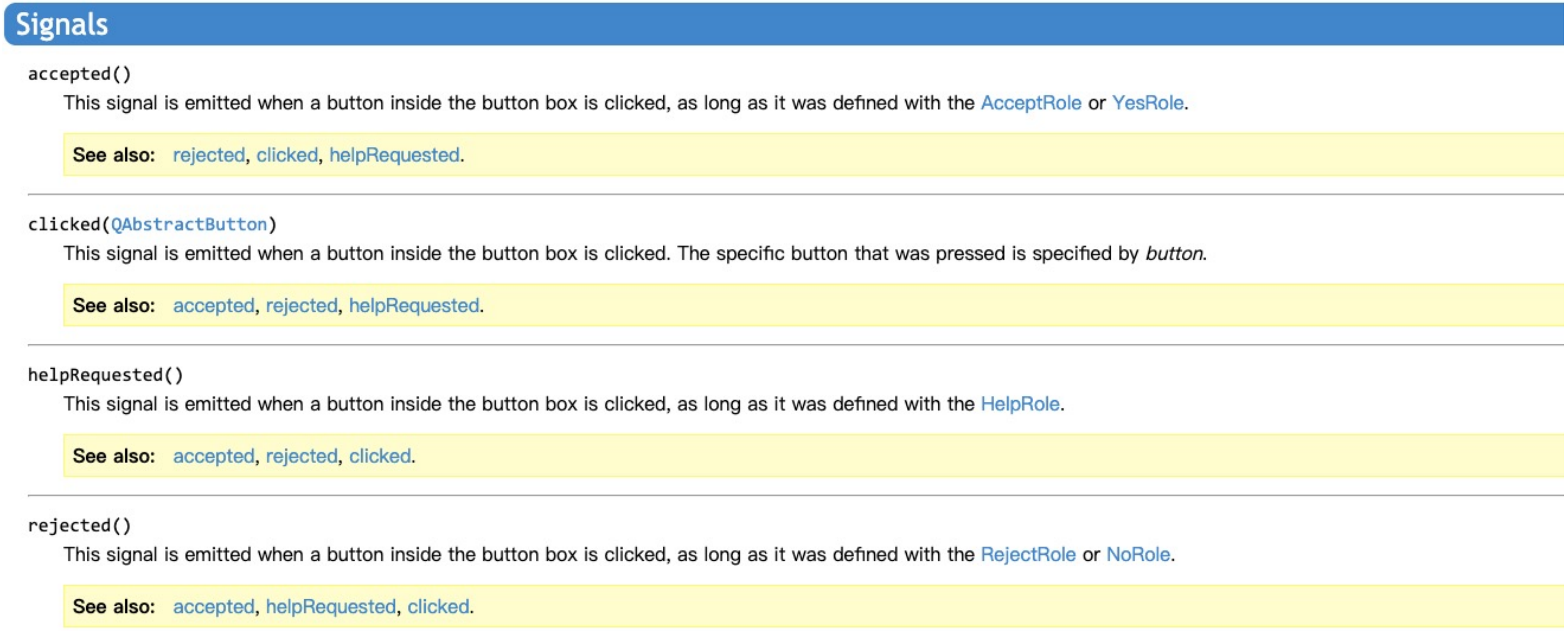
Member	Value	Description
Abort	0x00040000	An “Abort” button defined with the RejectRole.
Apply	0x02000000	An “Apply” button defined with the ApplyRole.
Cancel	0x00400000	A “Cancel” button defined with the RejectRole.
Close	0x00200000	A “Close” button defined with the RejectRole.
Discard	0x00800000	A “Discard” or “Don’t Save” button, depending on the platform, defined with the DestructiveRole.
Help	0x01000000	A “Help” button defined with the HelpRole.
Ignore	0x00100000	An “Ignore” button defined with the AcceptRole.
No	0x00010000	A “No” button defined with the NoRole.
NoButton	0x00000000	An invalid button.
NoToAll	0x00020000	A “No to All” button defined with the NoRole.
Ok	0x00000400	An “OK” button defined with the AcceptRole.
Open	0x00002000	An “Open” button defined with the AcceptRole.
Reset	0x04000000	A “Reset” button defined with the ResetRole.
RestoreDefaults	0x08000000	A “Restore Defaults” button defined with the ResetRole.
Retry	0x00080000	A “Retry” button defined with the AcceptRole.
Save	0x00000800	A “Save” button defined with the AcceptRole.
SaveAll	0x00001000	A “Save All” button defined with the AcceptRole.
Yes	0x00004000	A “Yes” button defined with the YesRole.
YesToAll	0x00008000	A “Yes to All” button defined with the YesRole.



HELLO!

Something happened, is that OK?

QDialog (6)



Signals

```
accepted()
```

This signal is emitted when a button inside the button box is clicked, as long as it was defined with the AcceptRole or YesRole.

See also: rejected, clicked, helpRequested.

```
clicked(QAbstractButton)
```

This signal is emitted when a button inside the button box is clicked. The specific button that was pressed is specified by *button*.

See also: accepted, rejected, helpRequested.

```
helpRequested()
```

This signal is emitted when a button inside the button box is clicked, as long as it was defined with the HelpRole.

See also: accepted, rejected, clicked.

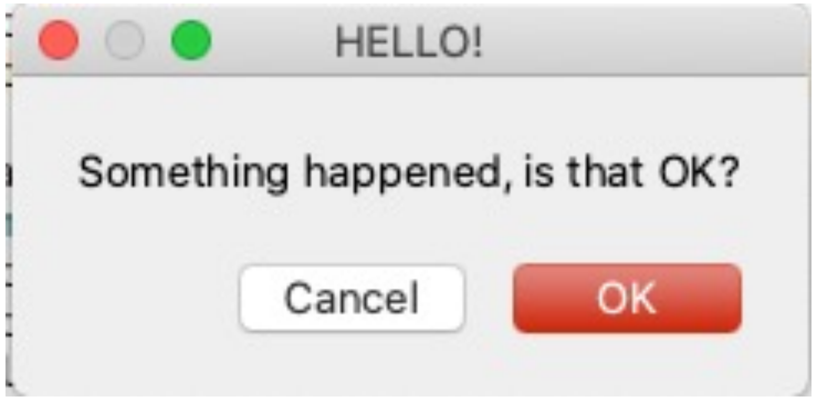
```
rejected()
```

This signal is emitted when a button inside the button box is clicked, as long as it was defined with the RejectRole or NoRole.

See also: accepted, helpRequested, clicked.

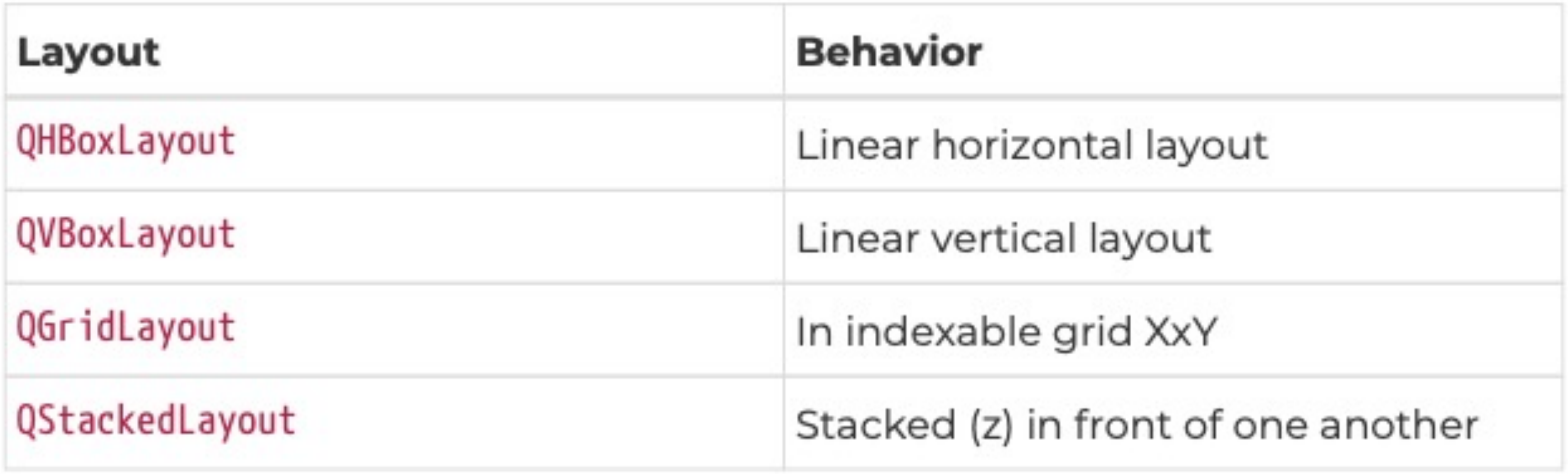
QDialog (7)

- self.buttonBox.accepted.connect(self.accept)
  self.buttonBox.rejected.connect(self.reject)
- self.reject () # defined in QDialog
  - Hides the modal dialog and sets the result code to Rejected.
- self.accept() # defined in QDialog
  - Hides the modal dialog and sets the result code to Accepted.



Something happened, is that OK?

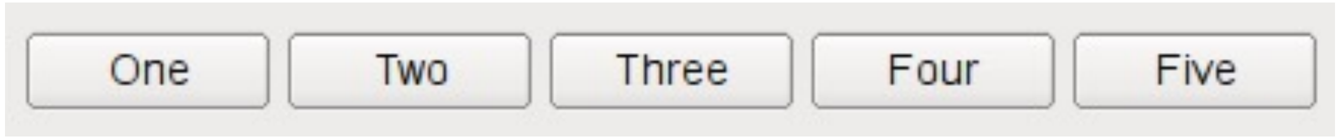
Layout

- There are 4 basic layouts available in Qt.
- 

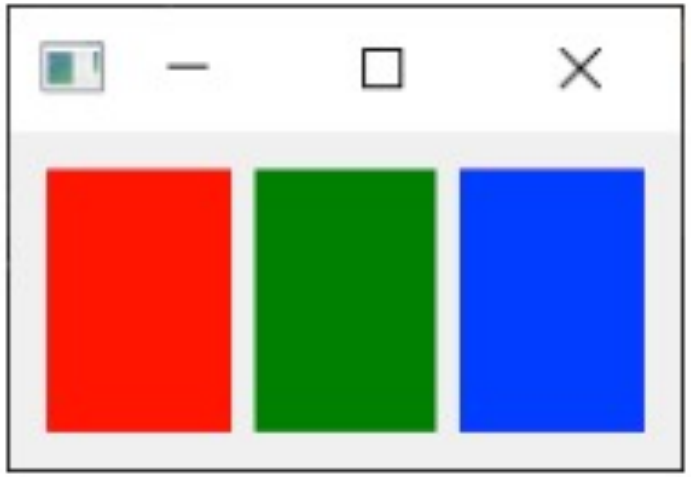
Layout	Behavior
`QHBoxLayout`	Linear horizontal layout
`QVBoxLayout`	Linear vertical layout
`QGridLayout`	In indexable grid XxY
`QStackedLayout`	Stacked (z) in front of one another

Layout (2)

- QHBoxLayout
- The QHBoxLayout class lines up widgets horizontally.



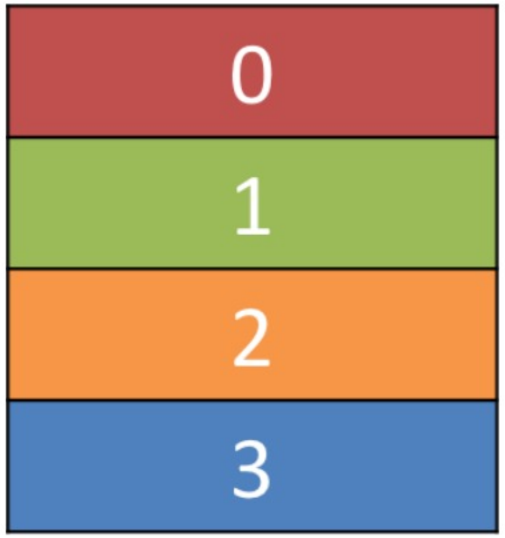
- layout = QHBoxLayout()
- layout.addWidget(Color("red"))
- layout.addWidget(Color("green"))
- layout.addWidget(Color("blue"))
- widget = QWidget()
- widget.setLayout(layout)



Layout (3)

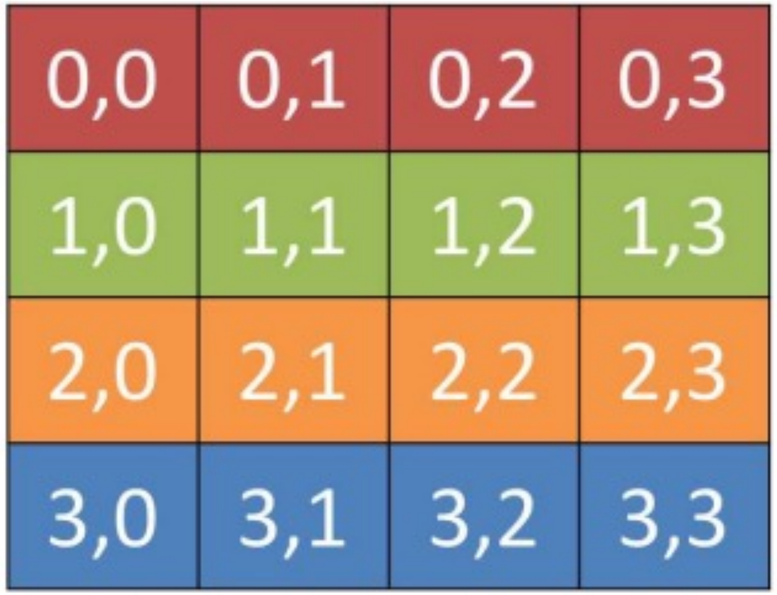
- QVBoxLayout
- With QVBoxLayout you arrange widgets one above the other linearly. Adding a widget adds it to the bottom of the column.

  - layout = QVBoxLayout()
  - layout.addWidget(Color("red"))
  - layout.addWidget(Color("green"))
  - layout.addWidget(Color("blue"))
  - widget = QWidget()
  - widget.setLayout(layout)



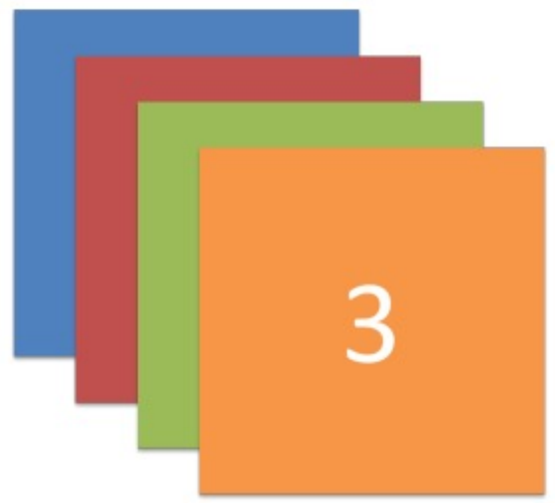
Layout (4)

- QGridLayout: widgets arranged in a grid
- QGridLayout allows you to position items specifically in a grid. You specify row and column positions for each widget. You can skip elements, and they will be left empty.



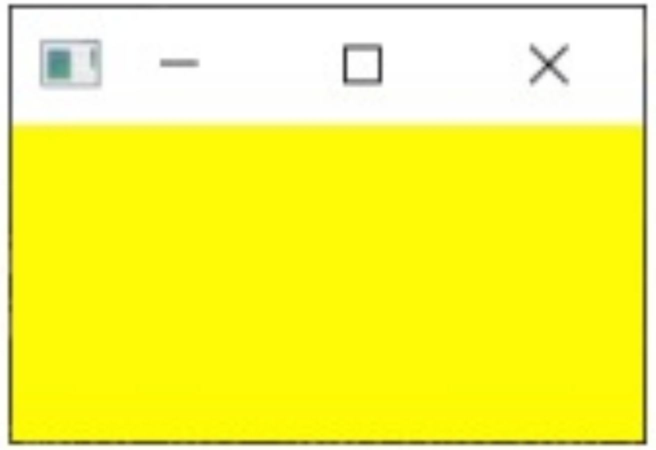
0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3
3,0	3,1	3,2	3,3

Layout (5)

- QStackedLayout
- This layout allows you to position elements directly in front of one another. You can then select which widget you want to show. You could use this for drawing layers in a graphics application.
- 

Layout (6)

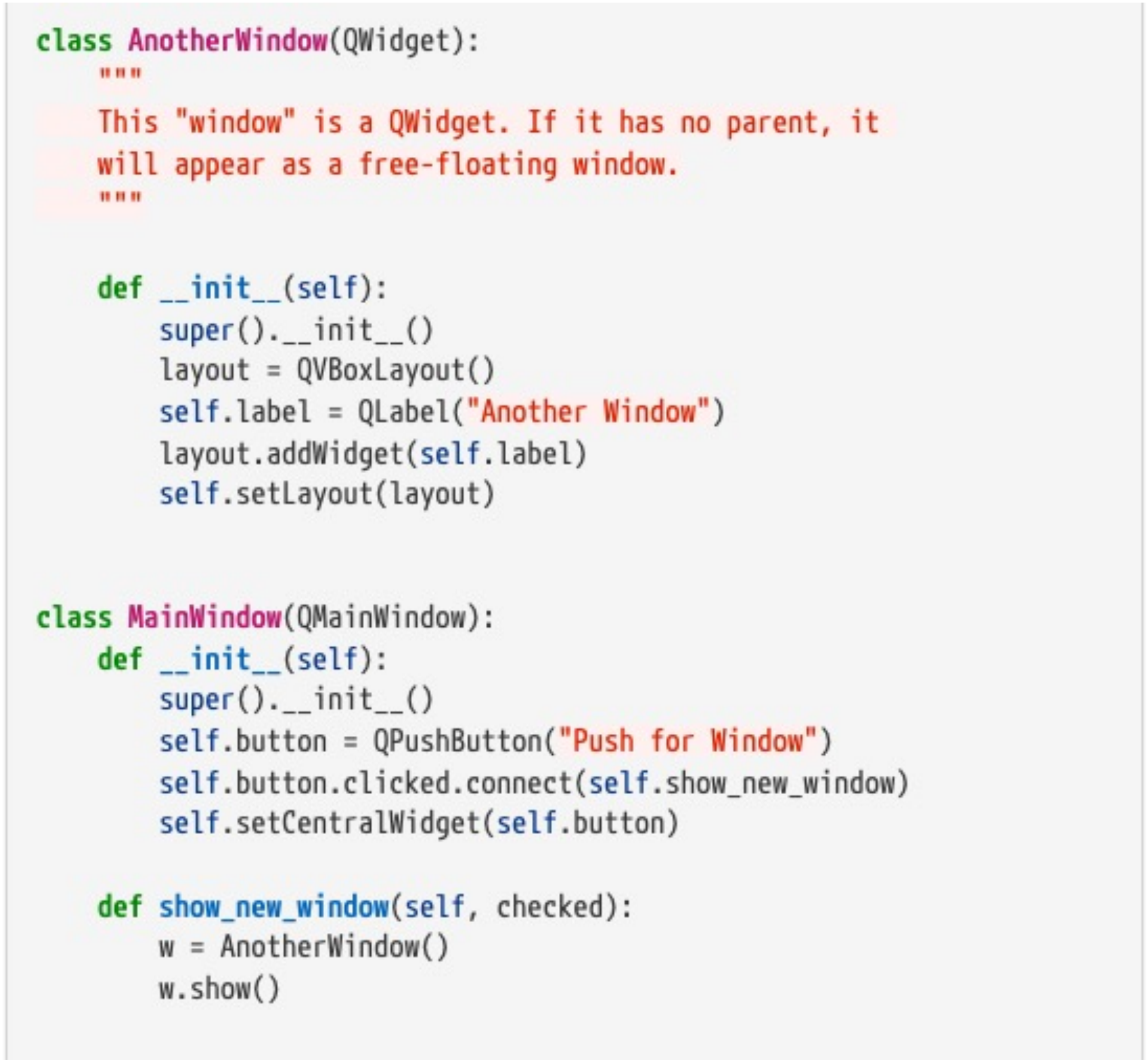
- layout = QStackedLayout()
- layout.addWidget(Color("red"))
- layout.addWidget(Color("green"))
- layout.addWidget(Color("blue"))
- layout.addWidget(Color("yellow"))



Window

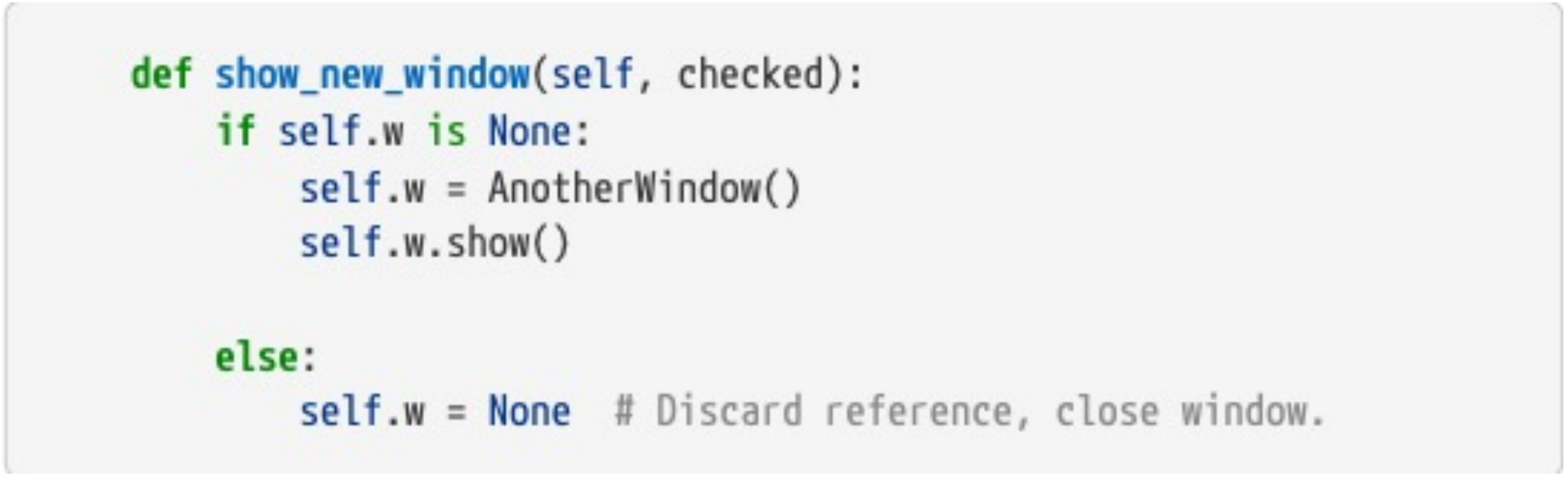
- To create a new window, you just need to create a new instance of a widget object without a parent.
- This can be any widget (technically any subclass of QWidget) including another QMainWindow if you prefer.
- There is no restriction on the number of QMainWindow instances you can have.

Window (2)



```python
class AnotherWindow(QWidget):
    """
    This "window" is a QWidget. If it has no parent, it
    will appear as a free-floating window.
    """

    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()
        self.label = QLabel("Another Window")
        layout.addWidget(self.label)
        self.setLayout(layout)


class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.show_new_window)
        self.setCentralWidget(self.button)

    def show_new_window(self, checked):
        w = AnotherWindow()
        w.show()
```

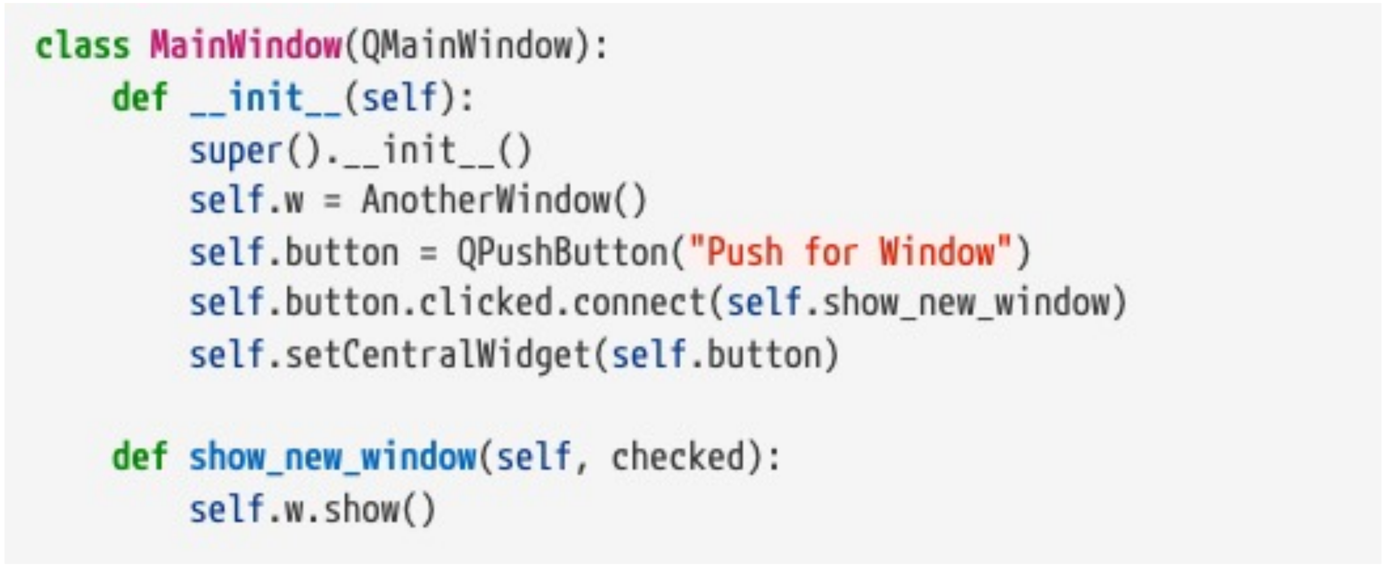
Window (3)

- Closing a window (Programming)
- If no reference to a window is kept, it will be discarded (and closed).
- By setting window to None (or any other value) the reference to the window will be lost, and the window will close.
- 

```
def show_new_window(self, checked):
    if self.w is None:
        self.w = AnotherWindow()
        self.w.show()

    else:
        self.w = None  # Discard reference, close window.
```

Window (4)

- Persistent Windows
- Sometimes you have a number of standard application windows. In this case it can often make more sense to create the additional windows first, then use .show() to display them when needed.



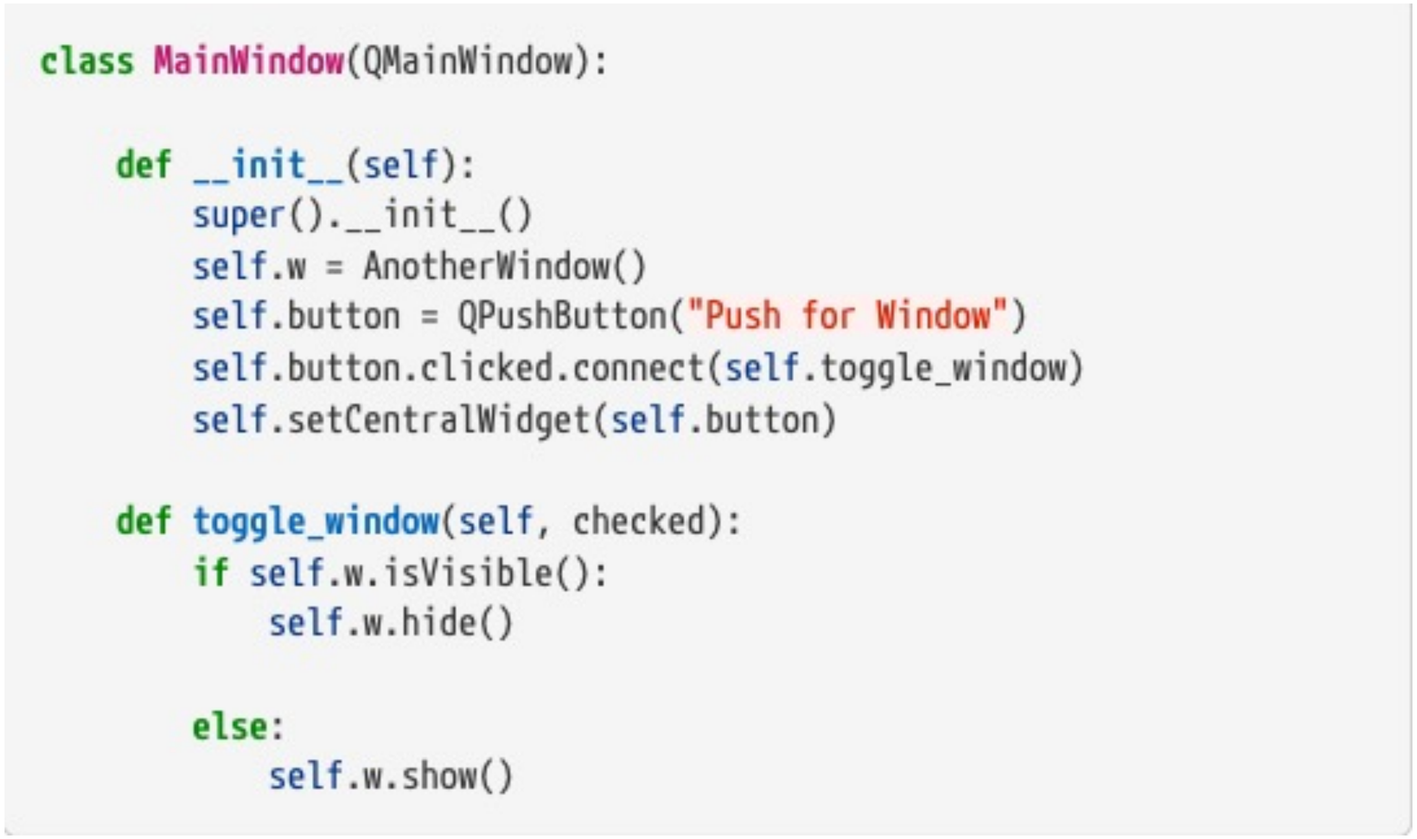
```python
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.w = AnotherWindow()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.show_new_window)
        self.setCentralWidget(self.button)

    def show_new_window(self, checked):
        self.w.show()
```

Window (5)

- Showing & hiding windows
- Once you have created a persistent window you can show and hide it without recreating it. Once hidden the window still exists, but will not be visible and accept mouse/other input.
- However you can continue to call methods on the window and update it's state – including changing it's appearance. Once re-shown any changes will be visible

Window (6)

- Showing & hiding windows
-



```
class MainWindow(QMainWindow):

    def __init__(self):
        super().__init__()
        self.w = AnotherWindow()
        self.button = QPushButton("Push for Window")
        self.button.clicked.connect(self.toggle_window)
        self.setCentralWidget(self.button)

    def toggle_window(self, checked):
        if self.w.isVisible():
            self.w.hide()

        else:
            self.w.show()
```

Connecting Signals between Windows

- Previously, we saw how it was possible to connect widgets together directly using signals and slots.
- All we needed was for the destination widget to have been created and to have a reference to via a variable.
- The same principle applies when connecting signals across windows—you can hook up signals in one window to slots in another, you just need to be able to access the slot.

Context Menus

- Context menus are small context-sensitive menus which typically appear when right clicking on a window.
- Qt has support for generating these menus, and widgets have a specific event used to trigger them.
- The .contextMenuEvent event is fired whenever a context menu is about to be shown, and is passed a single value event of type QContextMenuEvent.

Context Menus (2)

- To intercept the event, we simply override the object method with our new method of the same name. So in this case we can create a method on our MainWindow subclass with the name contextMenuEvent and it will receive all events of this type.

Context Menus (3)

```
import sys
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QAction
from PyQt6.QtWidgets import  QApplication, QLabel, QMainWindow, QMenu

class MainWindow(QMainWindow):
    def __init__(self):
          super().__init__()
    def contextMenuEvent(self, e):
         context = QMenu(self)
         context.addAction(QAction("test 1", self))
         context.addAction(QAction("test 2", self))
         context.addAction(QAction("test 3", self))
         context.exec(e.globalPos())

if __name__ == '__main__':
   app = QApplication(sys.argv)
   window = MainWindow()
   window.show()
   app.exec()
```

Context Menus (4)

- If you run the above code and right-click within the window, you'll see a context menu appear. You can set up .triggered slots on your menu actions as normal (and re-use actions defined for menus and toolbars).

Context Menus (5)

-
  ```
  def contextMenuEvent(self, e):
      context = QMenu(self)
      context.addAction(QAction("test 1", self))
      context.addAction(QAction("test 2", self))
      q3 = QAction("test 3", self)
      context.addAction(q3)
      q3.triggered.connect(self.q3_clicked)
      context.exec(e.globalPos())

  def q3_clicked(self):
      print("Q3 Clicked")
  ```

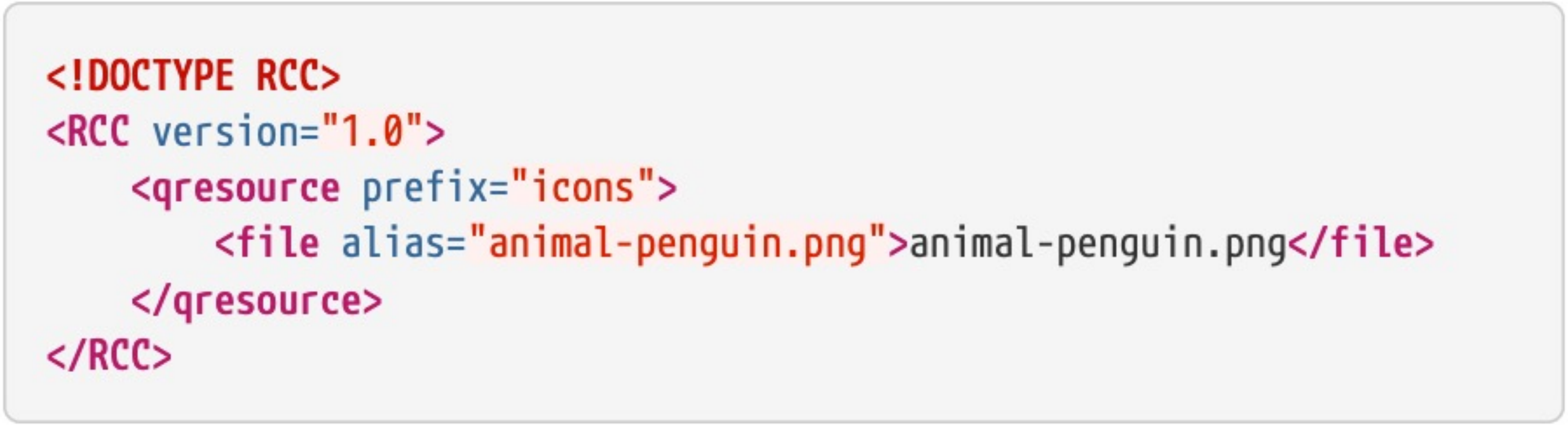
Qt Resource System

- Building applications takes more than just code. Usually your interface will need icons for actions, you may want to add illustrations or branding logos, or perhaps your application will need to load data files to pre-populate widgets.
- These data files are separate from the source code of your application but will ultimately need to be packaged and distributed with it in order for it to work.

Qt Resource System (2)

- Resources are bundled into Python files which can be distributed along with your source code, guaranteeing they will continue to work on other platforms.
- You can manage Qt resources through Qt Designer (or Qt Creator) and use resource library to add icons (and other graphics) to your apps.

Qt Resource System (3)

- The QRC file
- The core of the QR resource system is the resource file or QRC. The .qrc file is a simple XML file, which can be opened in any text editor.
- 

```
<!DOCTYPE RCC>
<RCC version="1.0">
    <qresource prefix="icons">
        <file alias="animal-penguin.png">animal-penguin.png</file>
    </qresource>
</RCC>
```

Qt Resource System (4)

```
<!DOCTYPE RCC>
<RCC version="1.0">
    <qresource prefix="icons">
        <file alias="animal-penguin.png">animal-penguin.png</file>
    </qresource>
</RCC>
```

- The name between the tags is the path to the file, relative to the resource file. The alias is the name which this resource will be known by from within your application.

Qt Resource System (5)

- Using a QRC file
- To use a .qrc file in your app, you first need to compile it to Python.
- To compile our resources.qrc file to a Python file named resources.py we can use—
- pyrcc6 resources.qrc -o resources.py

Qt Resource System (6)

- To use the resource file in our application we need to make a few small changes.
- Firstly, we need to import resources at the top of our app, to load the resources into the Qt resource system, and then
- Secondly we need to update the path to the icon file to use the resource path format as follows
- The prefix :/ indicates that this is a resource path. The first name "icons" is the prefix namespace and the filename is taken from the file alias, both as defined in resources.qrc file

Qt Resource System (7)

```
import sys

from PyQt5 import QtGui, QtWidgets

import resources  # Import the compiled resource file.


class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Hello World")
        b = QtWidgets.QPushButton("My button")

        icon = QtGui.QIcon(":/icons/penguin.png")
        b.setIcon(icon)
        self.setCentralWidget(b)

        self.show()
```

Resources in Qt Designer and Qt Creator

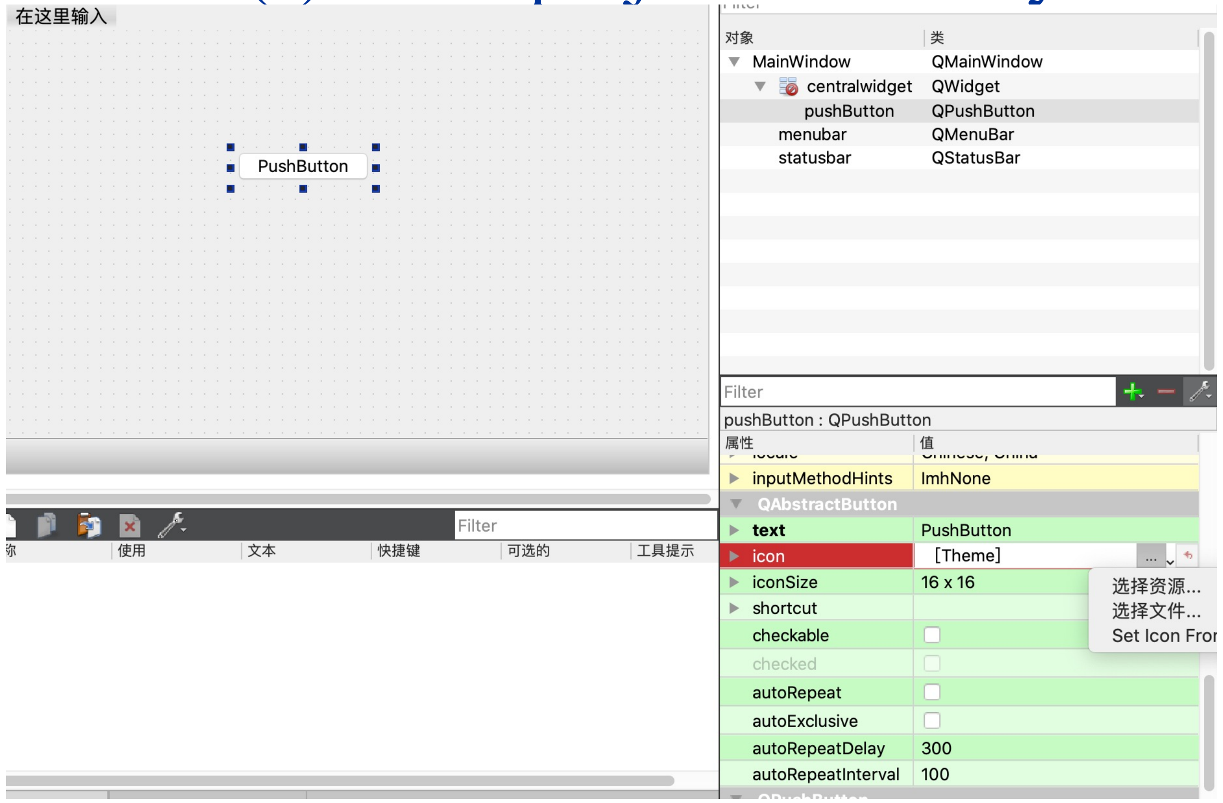
- While it’s fairly straightforward to manage your resources by editing the QRC file directly, Qt Designer can also be used to edit the resource library.

Resources in Qt Designer and Qt Creator (2)

- Adding Resources in Qt Designer
- If you’re using the standalone Qt Designer, the resource browser is available as a dockable widget, visible in the bottom right by default. If the Resource Browser is hidden you can show it through the "View" menu on the toolbar.

Resources in Qt Designer and Qt Creator (3)

- The Resource chooser window that appears allows you to pick icons from the resource file(s) in the project to use in your UI.

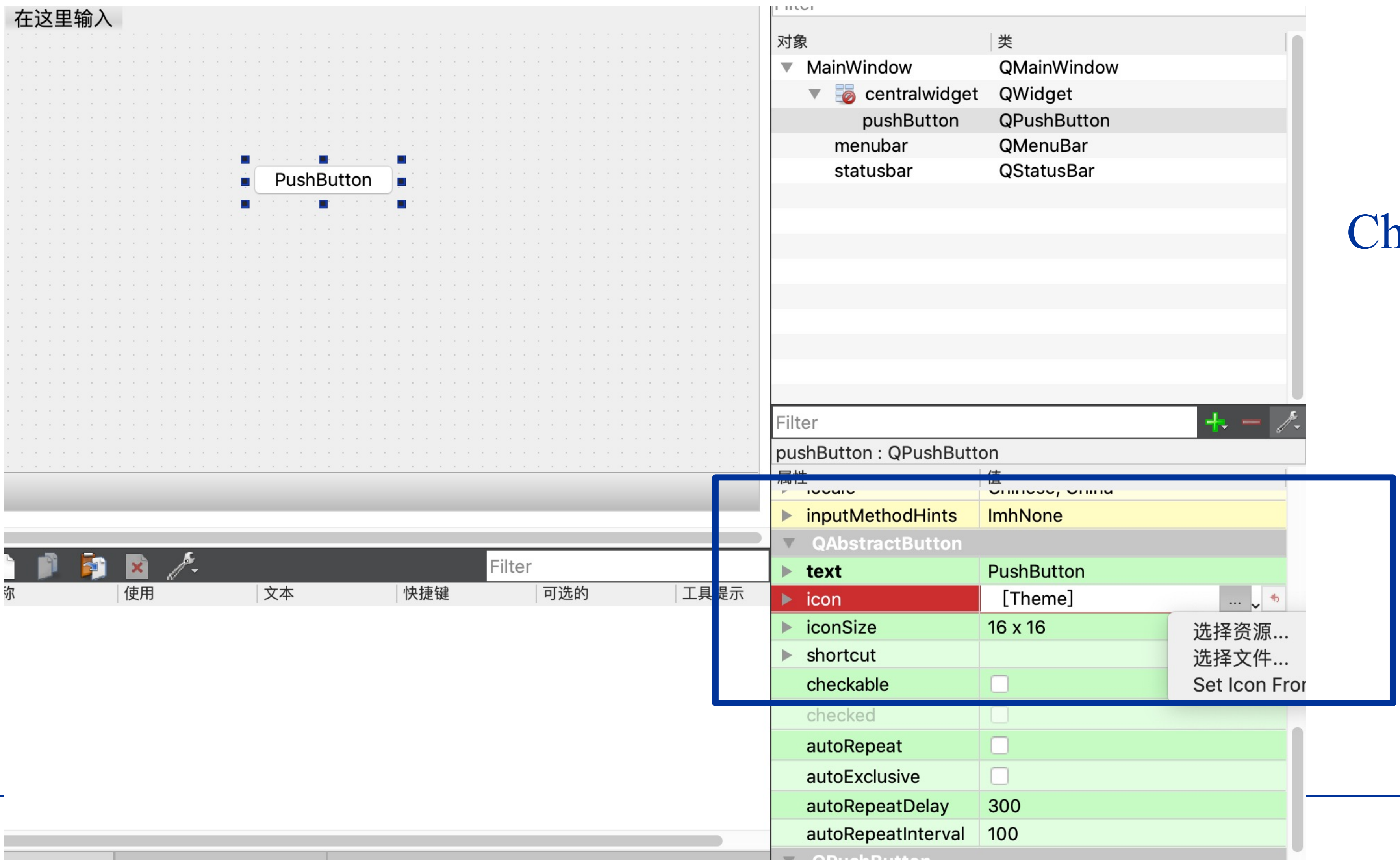


对象	类
MainWindow	QMainWindow
centralwidget	QWidget
pushButton	QPushButton
menubar	QMenuBar
statusbar	QStatusBar

pushButton : QPushButton

属性	值
inputMethodHints	ImhNone
QAbstractButton	
text	PushButton
icon	[Theme]
iconSize	16 x 16
shortcut	
checkable	☐
checked	☐
autoRepeat	☐
autoExclusive	☐
autoRepeatDelay	300
autoRepeatInterval	100

Resources in Qt Designer and Qt Creator (4)



在这里输入

PushButton

对象	类
MainWindow	QMainWindow
centralwidget	QWidget
pushButton	QPushButton
menubar	QMenuBar
statusbar	QStatusBar

pushButton : QPushButton

属性	值
inputMethodHints	ImhNone
QAbstractButton	
text	PushButton
icon	[Theme]
iconSize	16 x 16
shortcut	
checkable	☐
checked	☐
autoRepeat	☐
autoExclusive	☐
autoRepeatDelay	300
autoRepeatInterval	100

Choose Icon File

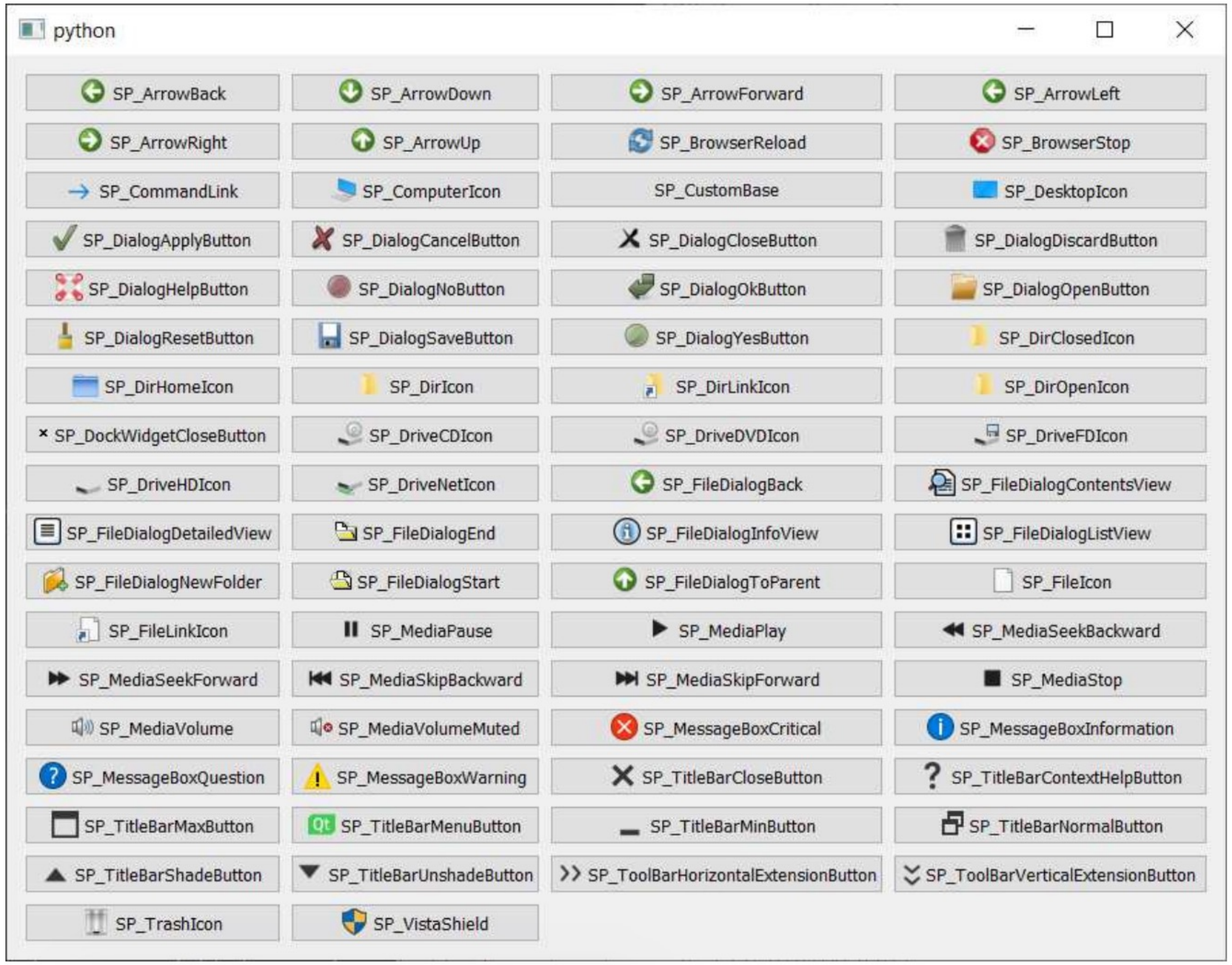
Icons

- Icons are small pictures which are used to aid navigation or understanding within a user interface. They are commonly found on buttons, either alongside or in place of text, or alongside actions in menus.
- By using easily recognizable indicators you can make your interface easier to use.
- In PyQt6 you have a number of different options for how to source and integrate icons into your application.

Qt Standard Icons

- The easiest way to add simple icons to your application is to use the built-in icons which ship with Qt itself. This small set of icons cover a number of standard use cases, from file operations, forward & backward arrows and message box indicators

Qt Standard Icons (2)

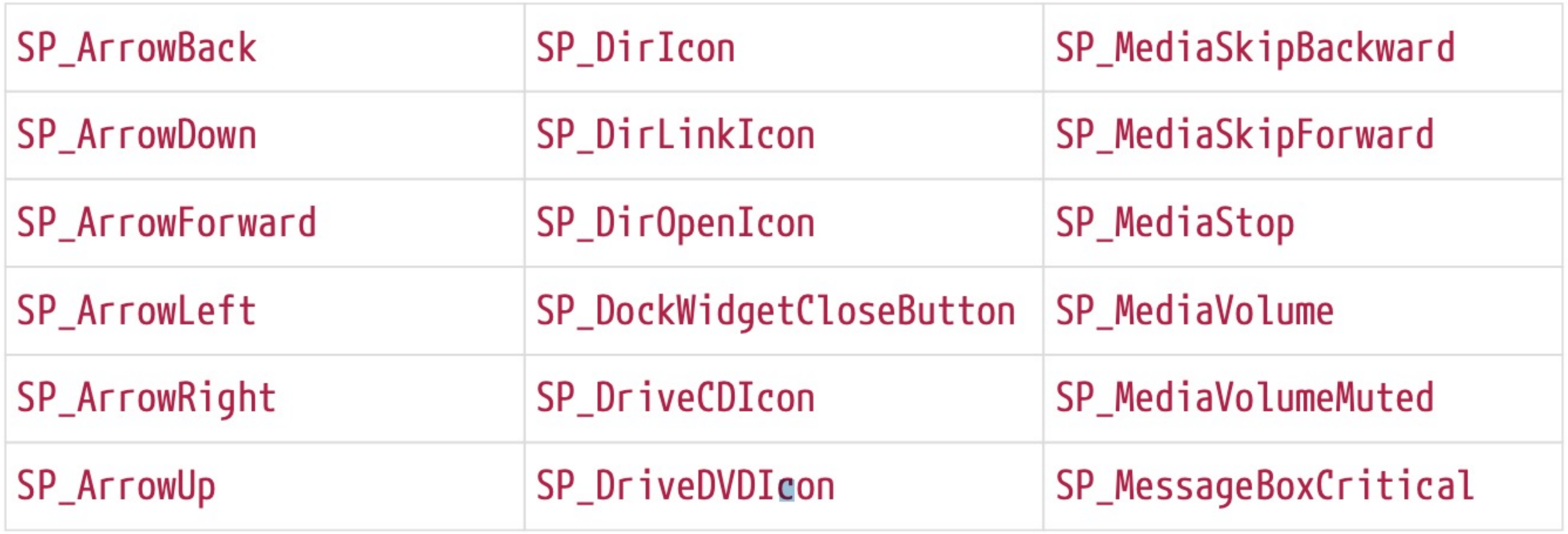


python

SP_ArrowBack	SP_ArrowDown	SP_ArrowForward	SP_ArrowLeft
SP_ArrowRight	SP_ArrowUp	SP_BrowserReload	SP_BrowserStop
SP_CommandLink	SP_ComputerIcon	SP_CustomBase	SP_DesktopIcon
SP_DialogApplyButton	SP_DialogCancelButton	SP_DialogCloseButton	SP_DialogDiscardButton
SP_DialogHelpButton	SP_DialogNoButton	SP_DialogOkButton	SP_DialogOpenButton
SP_DialogResetButton	SP_DialogSaveButton	SP_DialogYesButton	SP_DirClosedIcon
SP_DirHomeIcon	SP_DirIcon	SP_DirLinkIcon	SP_DirOpenIcon
SP_DockWidgetCloseButton	SP_DriveCDIcon	SP_DriveDVDIcon	SP_DriveFDIcon
SP_DriveHDIcon	SP_DriveNetIcon	SP_FileDialogBack	SP_FileDialogContentsView
SP_FileDialogDetailedView	SP_FileDialogEnd	SP_FileDialogInfoView	SP_FileDialogListView
SP_FileDialogNewFolder	SP_FileDialogStart	SP_FileDialogToParent	SP_FileIcon
SP_FileLinkIcon	SP_MediaPause	SP_MediaPlay	SP_MediaSeekBackward
SP_MediaSeekForward	SP_MediaSkipBackward	SP_MediaSkipForward	SP_MediaStop
SP_MediaVolume	SP_MediaVolumeMuted	SP_MessageBoxCritical	SP_MessageBoxInformation
SP_MessageBoxQuestion	SP_MessageBoxWarning	SP_TitleBarCloseButton	SP_TitleBarContextHelpButton
SP_TitleBarMaxButton	SP_TitleBarMenuButton	SP_TitleBarMinButton	SP_TitleBarNormalButton
SP_TitleBarShadeButton	SP_TitleBarUnshadeButton	SP_ToolBarHorizontalExtensionButton	SP_ToolBarVerticalExtensionButton
SP_TrashIcon	SP_VistaShield		

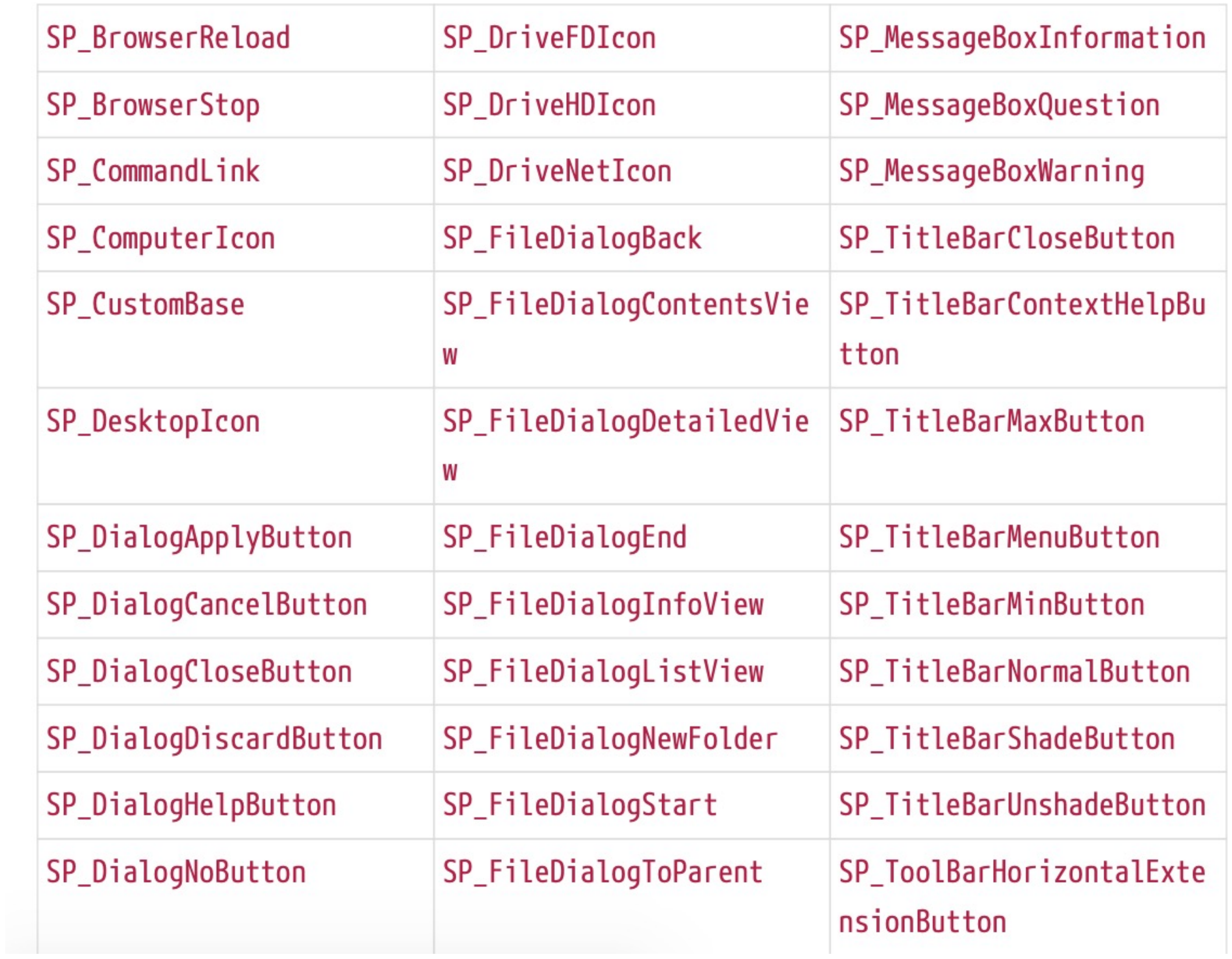
Qt Standard Icons (3)

- style = button.style() # Get the QStyle object from the widget.
- icon = style.standardIcon(QStyle.SP_MessageBoxCritical) button.setIcon(icon)



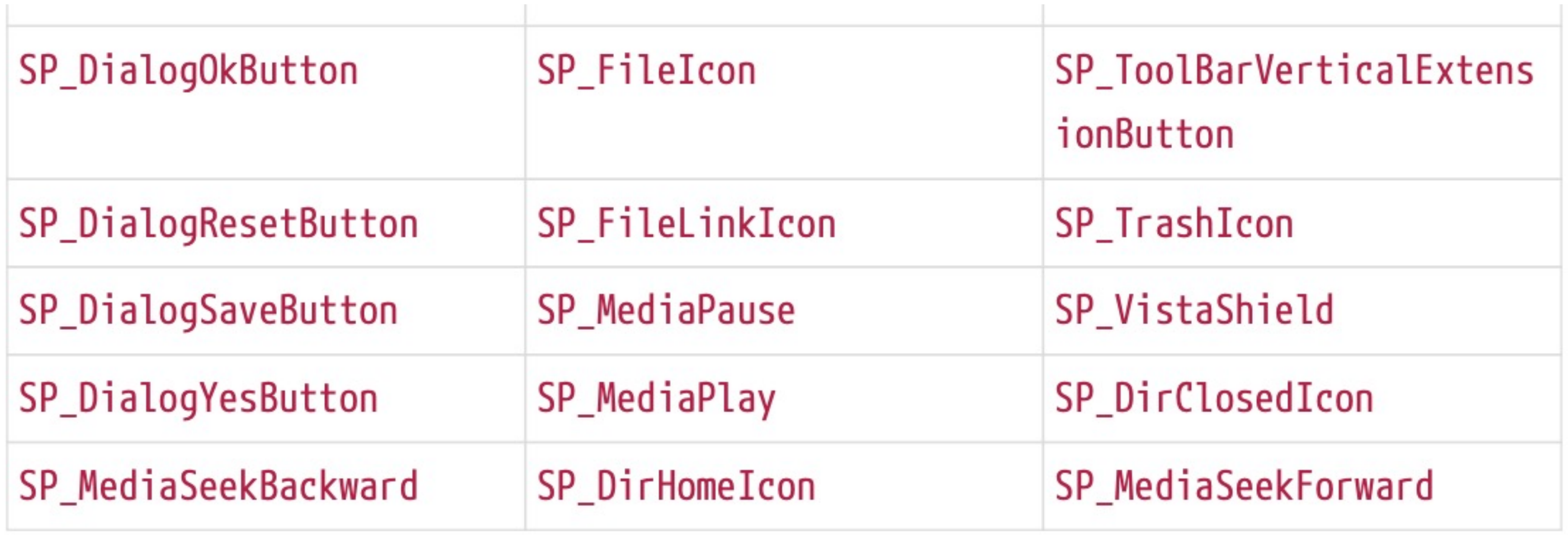
SP_ArrowBack	SP_DirIcon	SP_MediaSkipBackward
SP_ArrowDown	SP_DirLinkIcon	SP_MediaSkipForward
SP_ArrowForward	SP_DirOpenIcon	SP_MediaStop
SP_ArrowLeft	SP_DockWidgetCloseButton	SP_MediaVolume
SP_ArrowRight	SP_DriveCDIcon	SP_MediaVolumeMuted
SP_ArrowUp	SP_DriveDVDIcon	SP_MessageBoxCritical

Qt Standard Icons (4)



SP_BrowserReload	SP_DriveFDIcon	SP_MessageBoxInformation
SP_BrowserStop	SP_DriveHDIcon	SP_MessageBoxQuestion
SP_CommandLink	SP_DriveNetIcon	SP_MessageBoxWarning
SP_ComputerIcon	SP_FileDialogBack	SP_TitleBarCloseButton
SP_CustomBase	SP_FileDialogContentsView	SP_TitleBarContextHelpButton
SP_DesktopIcon	SP_FileDialogDetailedView	SP_TitleBarMaxButton
SP_DialogApplyButton	SP_FileDialogEnd	SP_TitleBarMenuButton
SP_DialogCancelButton	SP_FileDialogInfoView	SP_TitleBarMinButton
SP_DialogCloseButton	SP_FileDialogListView	SP_TitleBarNormalButton
SP_DialogDiscardButton	SP_FileDialogNewFolder	SP_TitleBarShadeButton
SP_DialogHelpButton	SP_FileDialogStart	SP_TitleBarUnshadeButton
SP_DialogNoButton	SP_FileDialogToParent	SP_ToolBarHorizontalExtensionButton

Qt Standard Icons (5)



SP_DialogOkButton	SP_FileIcon	SP_ToolBarVerticalExtensionButton
SP_DialogResetButton	SP_FileLinkIcon	SP_TrashIcon
SP_DialogSaveButton	SP_MediaPause	SP_VistaShield
SP_DialogYesButton	SP_MediaPlay	SP_DirClosedIcon
SP_MediaSeekBackward	SP_DirHomeIcon	SP_MediaSeekForward

The Model View Architecture

- Model–View–Controller (MVC) is an architectural pattern used for developing user interfaces.

- It divides an application into three interconnected parts, separating the internal representation of data from how it is presented to and accepted from the user.

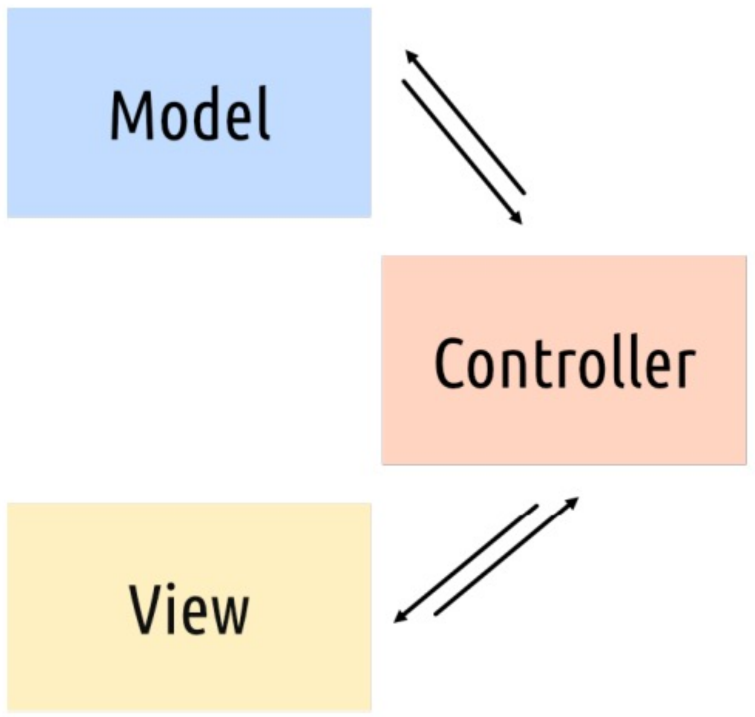
The Model View Architecture (2)

- The MVC pattern splits the interface into the following components —
- Model holds the data structure which the app is working with.
- View is any representation of information as shown to the user, whether graphical or tables. Multiple views of the same data are allowed.
- Controller accepts input from the user, transforms it into commands and applies these to the model or view

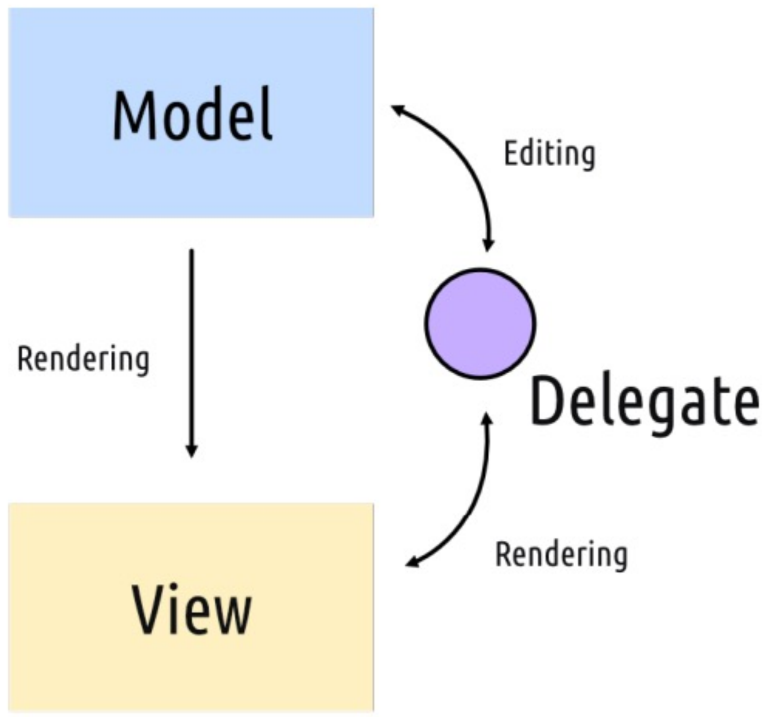
The Model View Architecture (3)

- In Qt land the distinction between the View & Controller gets a little murky.
- Qt accepts input events from the user via the OS and delegates these to the widgets (Controller) to handle.
- However, widgets also handle presentation of their own state to the user, putting them squarely in the View. Rather than agonize over where to draw the line, in Qt-speak the View and Controller are instead merged together creating a Model/ViewController architecture — called "Model View" for simplicity

The Model View Architecture (4)



Model View Controller architecture



Qt's Model/Views architecture

The Model View Architecture (5)

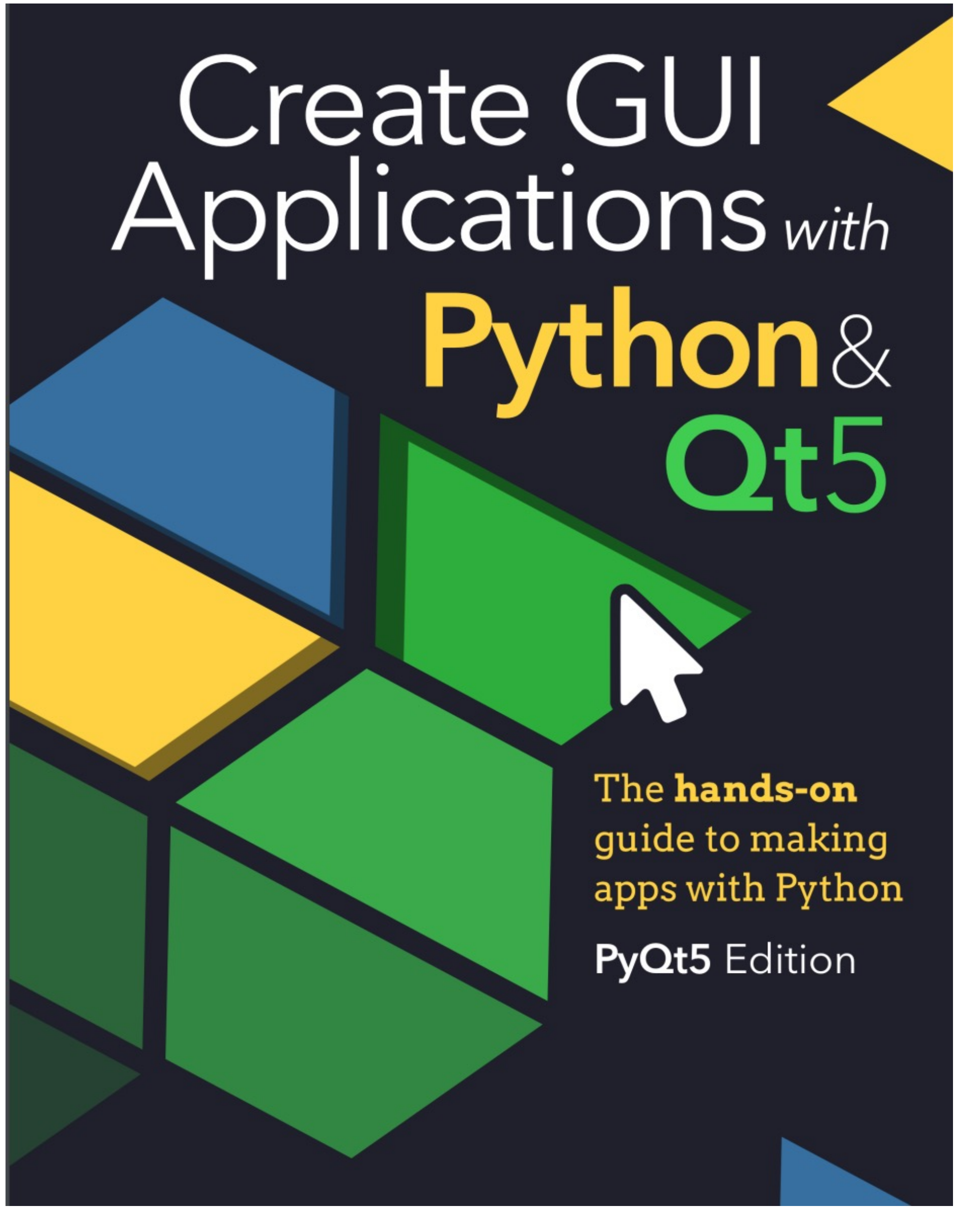
- The Model View
- The Model acts as the interface between the data store and the ViewController. The Model holds the data (or a reference to it) and presents this data through a standardized API which Views then consume and present to the user.
- Multiple Views can share the same data, presenting it in completely different ways.

The Model View Architecture (6)

- The two parts are essentially responsible for —
- 1. The model stores the data, or a reference to it and returns individual or ranges of records, and associated metadata or display instructions.
- 2. The view requests data from the model and displays what is returned on the widget.

Reference

- Create GUI Applications with Python & Qt5
- https://paddle.s3.amazonaws.com/fulfillment_downloads/16090/561130/D6eHc7VCSfGrmsulClgK_create-gui-applications-pyqt5.pdf


Create GUI Applications with Python & Qt5
The hands-on guide to making apps with Python
PyQt5 Edition