Application Creation

WEEK 4

Instructor: Ph.D. Oleg Tymchuk

Overview

Main Window

Window Geometry

Main Window

- A Main Window provides a framework for building an application's user interface
- Main windows have either a single (SDI) or multiple (MDI) document interface
- Main Window Classes:

QAction	Abstract user interface action that can be inserted into widgets
QActionGroup	Groups actions together
QWidgetAction	Extends QAction by an interface for inserting custom widgets into action based containers, such as toolbars
QDockWidget	Widget that can be docked inside a QMainWindow or floated as a top-level window on the desktop
QMainWindow	Main application window
QMdiArea	Area in which MDI windows are displayed
QMdiSubWindow	Subwindow class for QMdiArea
QMenu	Menu widget for use in menu bars, context menus, and other popup menus
QMenuBar	Horizontal menu bar
QSizeGrip	Resize handle for resizing top-level windows
QStatusBar	Horizontal bar suitable for presenting status information
QToolBar	Movable panel that contains a set of controls

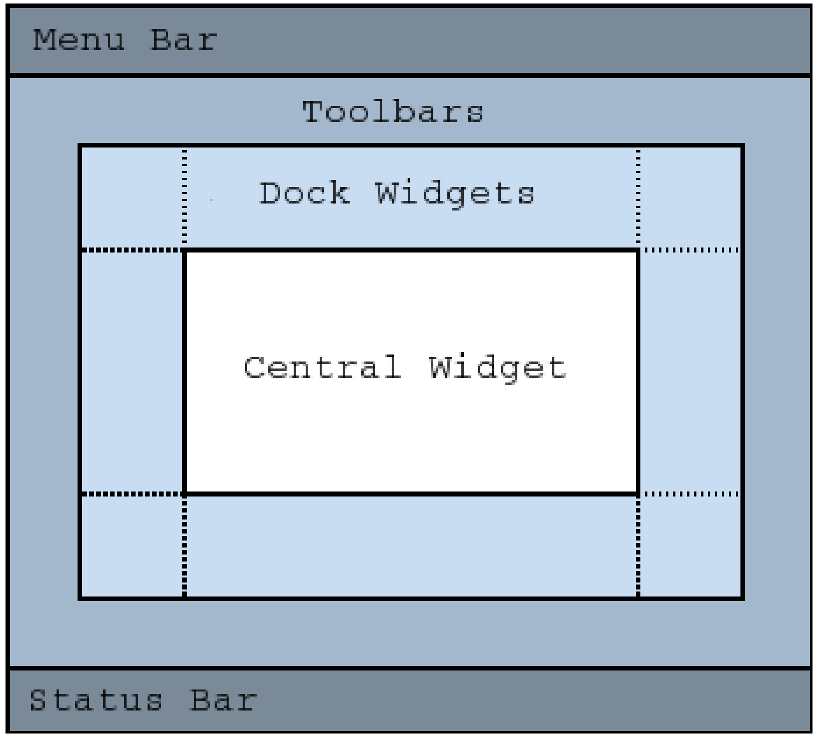
Main Window

The QMainWindow class provides a main application window

QMainWindow has its own layout to which you can add

- menu bar
- tool bars
- dockable widgets
- status bar

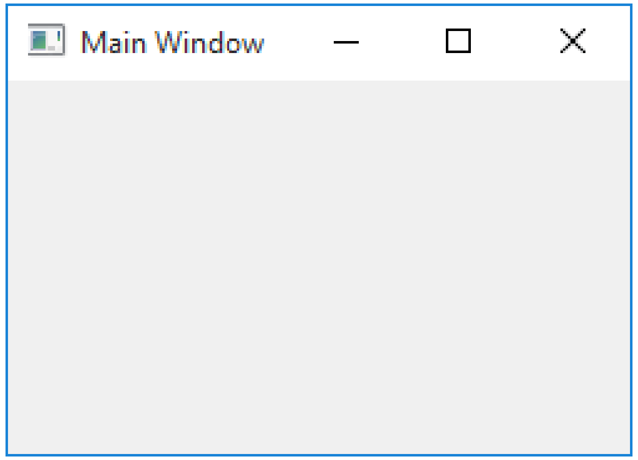
The center area can be occupied by any kind of QWidget



Note: Creating a main window without a central widget is not supported.
You must have a central widget even if it is just a placeholder.

Main Window. Example

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


class Example(QMainWindow):

    def __init__(self):
        super(Example, self).__init__()
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Main Window')

if __name__ == '__main__':
    app = QApplication(sys.argv)
    ex = Example()
    ex.show()
    sys.exit(app.exec())
```



Main Window. WindowFlags

Enum WindowFlags is used to specify various window-system properties for the widget

Constant	Description
QtCore.Qt.Window	Indicates that the widget is a window, usually with a window system frame and a title bar, irrespective of whether the widget has a parent or not. Note that it is not possible to unset this flag if the widget does not have a parent
QtCore.Qt.Dialog	Indicates that the widget is a window that should be decorated as a dialog (i.e., typically no maximize or minimize buttons in the title bar)
QtCore.Qt.Popup	Indicates that the widget is a pop-up top-level window, i.e. that it is modal, but has a window system frame appropriate for pop-up menus
QtCore.Qt.Tool	Indicates that the widget is a tool window. A tool window is often a small window with a smaller than usual title bar and decoration, typically used for collections of tool buttons
QtCore.Qt.ToolTip	Indicates that the widget is a tooltip. This is used internally to implement tooltips
QtCore.Qt.SplashScreen	Indicates that the window is a splash screen. This is the default type for QSplashScreen

Main Window. WindowFlags

- The CustomizeWindowHint flag is used to enable customization of the window controls

- This flag must be set to allow the WindowTitleHint, WindowSystemMenuHint, WindowMinimizeButtonHint, WindowMaximizeButtonHint and WindowCloseButtonHint flags to be changed

Constant	Description
QtCore.Qt.WindowTitleHint	Gives the window a title bar
QtCore.Qt.WindowSystemMenuHint	Adds a window system menu, and possibly a close button (for example on Mac)
QtCore.Qt.WindowMinimizeButtonHint	Adds a minimize button
QtCore.Qt.WindowMaximizeButtonHint	Adds a maximize button
QtCore.Qt.WindowCloseButtonHint	Adds a close button
QtCore.Qt.WindowContextHelpButtonHint	Adds a context help button to dialogs
QtCore.Qt.WindowStaysOnTopHint	Informs the window system that the window should stay on top of all other windows
QtCore.Qt.WindowStaysOnBottomHint	Informs the window system that the window should stay on bottom of all other windows
QtCore.Qt.WindowType_Mask	A mask for extracting the window type part of the window flags

Main Window. Example 1/5

The example consists of two classes:

- ControllerWindow is the main application window that allows the user to choose among the available window flags, and displays the effect on a separate preview window
- PreviewWindow is a custom window

ControllerWindow Class Definition

```
class ControllerWindow(QtWidgets.QMainWindow):
    # Constructor of ControllerWindow.
    # Create check boxes for three hints.
    def __init__(self):...

    # Create a check box.
    def create_check_box(self, text):...

    # Refresh the preview window.
    def update_hint(self):…
```

PreviewWindow Class Definition

```
class PreviewWindow
(QtWidgets.QMainWindow):
    # Constructor of PreviewWindow.
```

Main Window. Example 2/5

- ControllerWindow Class Implementation

```
def __init__(self):
    super(ControllerWindow, self).__init__()

    self.previewWindow = None
    # Creating the group of check boxes containing the available window flags
    self.windowMinimizeButtonCheckBox = self.create_check_box('Window minimize button')
    self.windowMaximizeButtonCheckBox = self.create_check_box('Window maximize button')
    self.windowCloseButtonCheckBox = self.create_check_box('Window close button')
    layout = QtWidgets.QHBoxLayout()
    layout.addWidget(self.windowMinimizeButtonCheckBox)
    layout.addWidget(self.windowMaximizeButtonCheckBox)
    layout.addWidget(self.windowCloseButtonCheckBox)

    self.centerWidget = QtWidgets.QWidget()
    self.setCentralWidget(self.centerWidget)
    self.centerWidget.setLayout(layout)
    self.setWindowTitle('Window Flags')
```

Main Window. Example 3/5

```
def create_check_box(self, text):
    # Creating QCheckBox widget
    check_box = QtWidgets.QCheckBox(text)
    # Connecting check box signal with slot
    check_box.clicked.connect(self.update_hint)
    return check_box

def update_hint(self):
    # Creating an empty WindowFlags
    flags = QtCore.Qt.WindowFlags()
    # Determining which one of the types that is checked and add it to flags
    if self.windowMinimizeButtonCheckBox.isChecked():
        flags |= QtCore.Qt.WindowMinimizeButtonHint
    if self.windowMaximizeButtonCheckBox.isChecked():
        flags |= QtCore.Qt.WindowMaximizeButtonHint
    if self.windowCloseButtonCheckBox.isChecked():
        flags |= QtCore.Qt.WindowCloseButtonHint
    # Creating the preview window
    self.previewWindow = PreviewWindow(flags)
    self.previewWindow.move(0, 0)
    self.previewWindow.show()
```

Main Window. Example 4/5

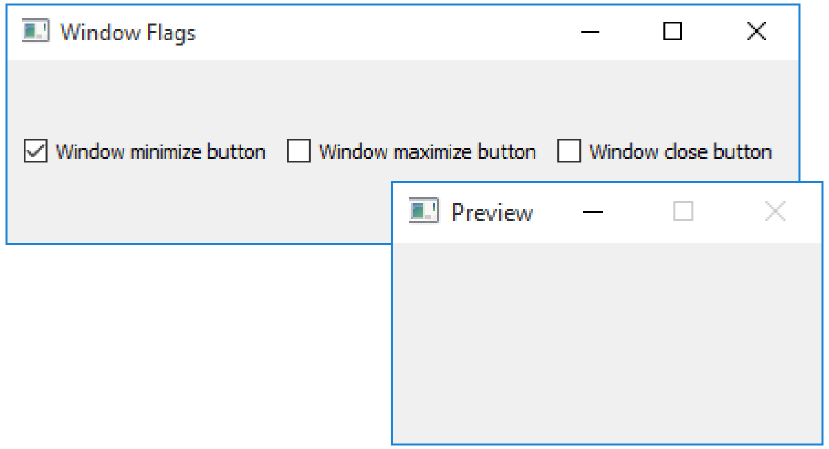
- PreviewWindow Class Implementation

```
class PreviewWindow (QtWidgets.QMainWindow):
    # Constructor of PreviewWindow.
    def __init__(self, flags, parent=None):
        super(PreviewWindow, self).__init__(parent)
        self.setWindowFlags(flags)
        self.setWindowTitle('Preview')
        self.setGeometry(0, 0, 200, 100)
```

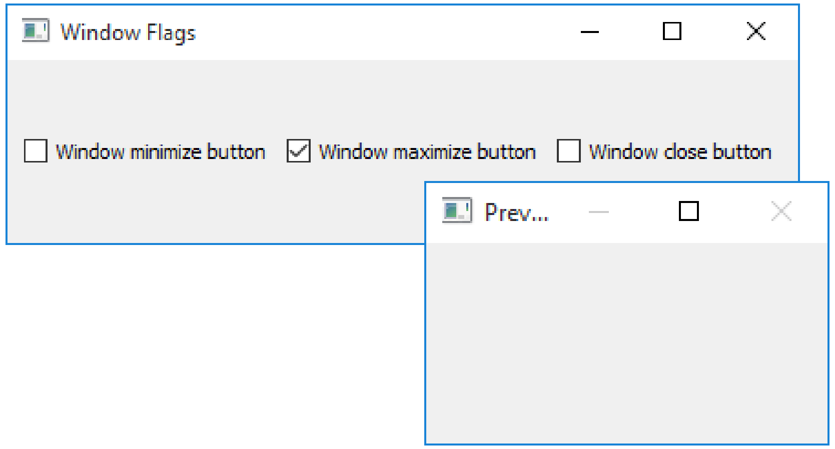
- Application Implementation

```
# Creating an application and main window
if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    controller = ControllerWindow()
    controller.show()
    sys.exit(app.exec())
```

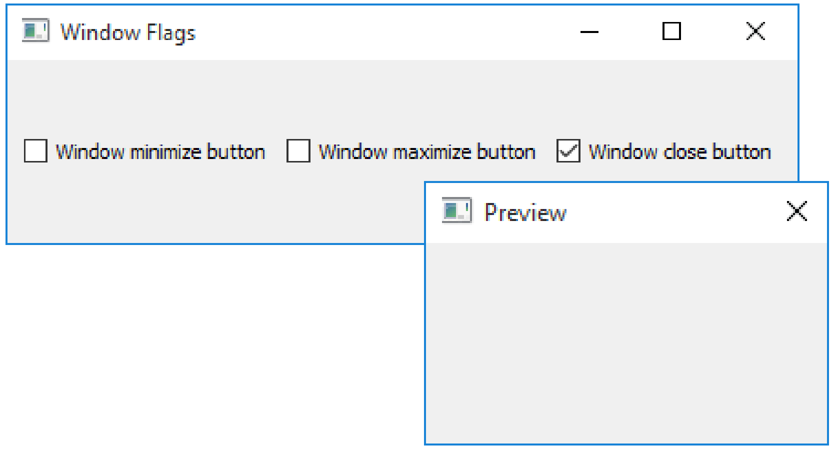
Main Window. Example 5/5



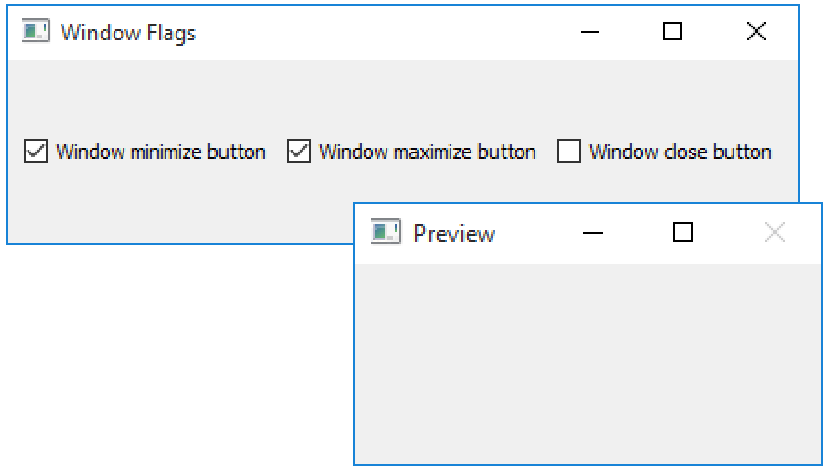
[✓] Window minimize button [ ] Window maximize button [ ] Window close button



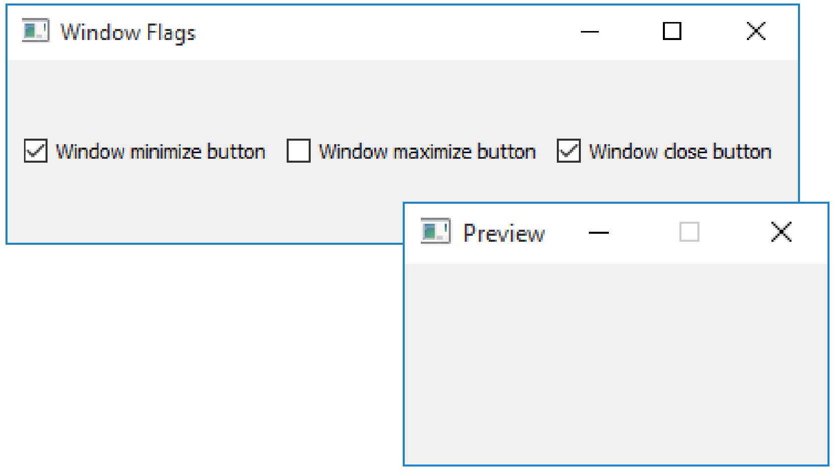
[ ] Window minimize button [✓] Window maximize button [ ] Window close button



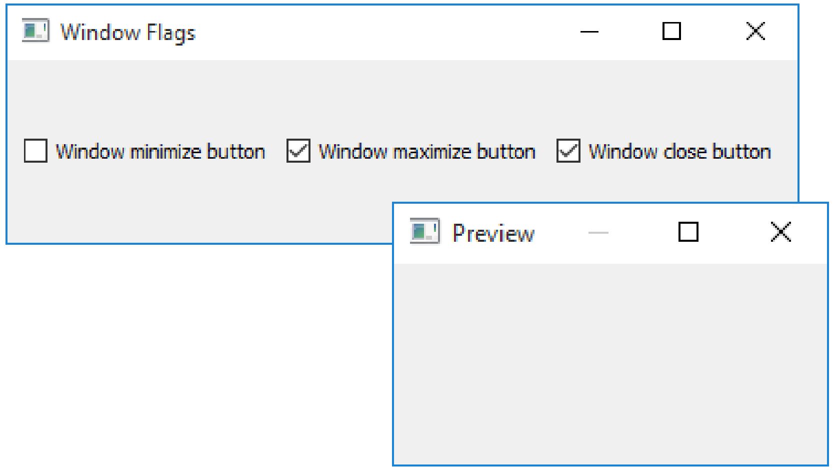
[ ] Window minimize button [ ] Window maximize button [✓] Window close button



[✓] Window minimize button [✓] Window maximize button [ ] Window close button



[✓] Window minimize button [ ] Window maximize button [✓] Window close button

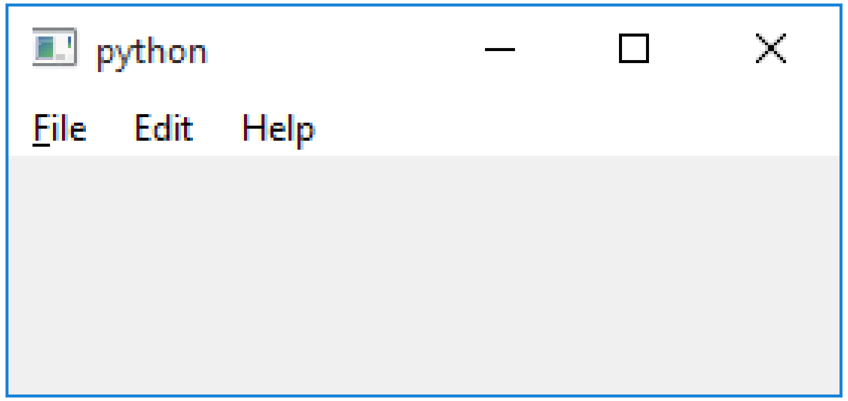


[ ] Window minimize button [✓] Window maximize button [✓] Window close button

Main Window. Creating Menus

- The QMenuBar class provides a horizontal menu bar
- A menu bar consists of a list of pull-down menu items
- addMenu() appends menu to the menu bar. Returns the menu's menuAction()
- The ampersand in the menu item's text sets Alt+F as a shortcut for this menu (you can use "&&" to get a real ampersand in the menu bar)

```
...

class Example(QMainWindow):
    def __init__(self, parent=None):
        super(Example, self).__init__(parent)

        main_menu = self.menuBar()
        main_menu.addMenu('&File')
        main_menu.addMenu('Edit')
        main_menu.addMenu('Help')

...
```



Main Window. Creating Menus

QActions are added to the menus, which display them as menu items

The QAction class provides an abstract user interface action that can be inserted into widgets

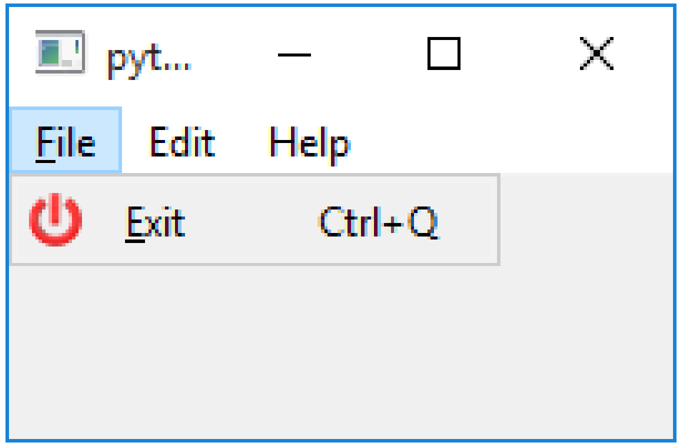
- may contain an icon, menu text, a shortcut, status text, and a tooltip
- emits signal triggered on execution
- connected slot performs action
- added to menus, toolbar, key shortcuts
- each performs same way
- regardless of user interface used

Actions are added to widgets using QWidget.addAction()

Note that an action must be added to a widget before it can be used

Main Window. Creating Menus

```
...

class Example(QMainWindow):
   def __init__(self, parent=None):
       super(Example, self).__init__(parent)

       main_menu = self.menuBar()
       file_item = main_menu.addMenu('&File')
       edit_item = main_menu.addMenu('Edit')
       help_item = main_menu.addMenu('Help')

       exitAction = QAction(QtGui.QIcon('exit.png'), '&Exit', self)
       exitAction.setShortcut('Ctrl+Q')
       exitAction.setStatusTip('Exit application')
       exitAction.triggered.connect(self.close)

       file_item.addAction(exitAction)

...
```



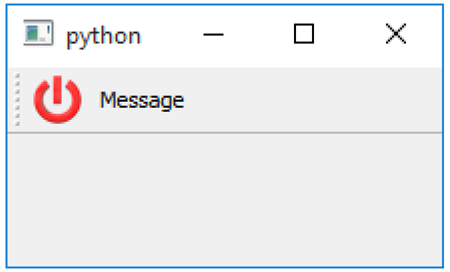
Main Window. Creating Toolbars

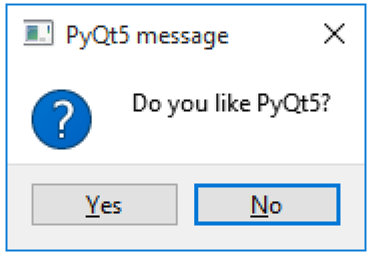
- Toolbars are implemented in the QToolBar class
- The QToolBar class provides a movable panel that contains a set of controls
- Toolbar buttons are added by adding actions, using addAction() or insertAction()
- Groups of buttons can be separated using addSeparator() or insertSeparator()
- If a toolbar button is not appropriate, a widget can be inserted instead using addWidget() or insertWidget()
- When a toolbar button is pressed, it emits the actionTriggered() signal
- A toolbar can be fixed in place in a particular area (e.g., at the top of the window), or it can be movable between toolbar areas; see setMovable(), isMovable(), allowedAreas() and isAreaAllowed()

You add a toolbar to a main window with addToolBar()

Main Window. Creating Toolbars

```
class Example(QMainWindow):
    def __init__(self, parent=None):
        super(Example, self).__init__(parent)

        self.toolbar = self.addToolBar('')
        exitAction = QAction(QtGui.QIcon('exit.png'), '&Exit', self)
        exitAction.setShortcut('Ctrl+Q')
        exitAction.setStatusTip('Exit application')
        exitAction.triggered.connect(self.close)
        self.toolbar.addAction(exitAction)

        messageAction = QAction('&Message', self)
        messageAction.setStatusTip('Message Box')
        messageAction.triggered.connect(self.show_message)
        self.toolbar.addAction(messageAction)
        # Toolbar is restricted to the top and bottom toolbar areas.
        self.toolbar.setAllowedAreas(QtCore.Qt.TopToolBarArea |
                                     QtCore.Qt.BottomToolBarArea)


    def show_message(self):
        QMessageBox.question(self, 'PyQt5 message', "Do you like PyQt5?",
                             QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
```





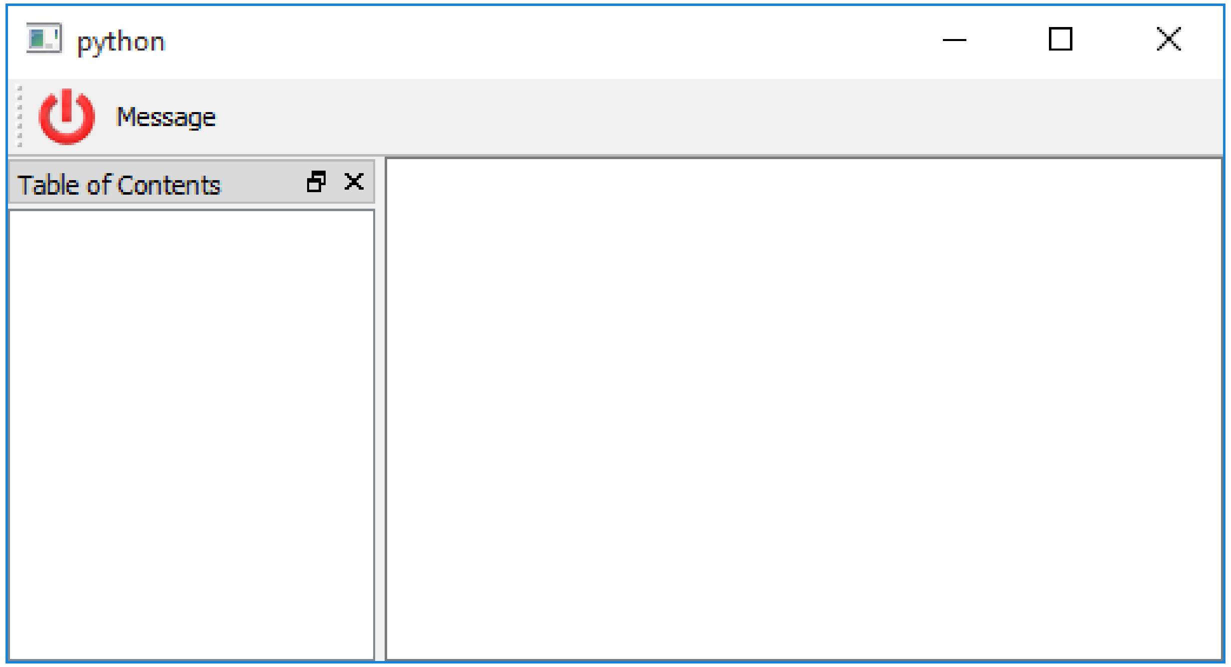
Do you like PyQt5?

Main Window. Creating Dock Widgets

- Dock widgets are implemented in the QDockWidget class
- The QDockWidget class provides a widget that can be docked inside a QMainWindow or floated as a top-level window on the desktop
- QDockWidget provides the concept of dock widgets, also know as tool palettes or utility windows
- Dock windows can be moved inside their current area, moved into new areas and floated (e.g., undocked) by the end-user. The QDockWidget API allows the programmer to restrict the dock widgets ability to move, float and close, as well as the areas in which they can be placed
- A QDockWidget consists of a title bar and the content area. The title bar displays the dock widgets window title, a float button and a close button. Depending on the state of the QDockWidget, the float and close buttons may be either disabled or not shown at all
- The visual appearance of the title bar and buttons is dependent on the style in use

Main Window. Creating Dock Widgets

```
...

contentsWindow = QDockWidget('Table of Contents', self)
contentsWindow.setAllowedAreas(QtCore.Qt.LeftDockWidgetArea)
self.addDockWidget(QtCore.Qt.LeftDockWidgetArea,
contentsWindow)

headingList = QListWidget(contentsWindow)
contentsWindow.setWidget(headingList)

self.setCentralWidget(QTextEdit())

...
```



Main Window. The Status Bar

- The QStatusBar class provides a horizontal bar suitable for presenting status information

- Each status indicator falls into one of three categories:

  Temporary - briefly occupies most of the status bar. Used to explain tool tip texts or menu entries, for example.
  Normal - occupies part of the status bar and may be hidden by temporary messages.
  Used to display the page and line number in a word processor, for example.
  Permanent - is never hidden. Used for important mode indications.

- QStatusBar lets you display all three types of indicators.

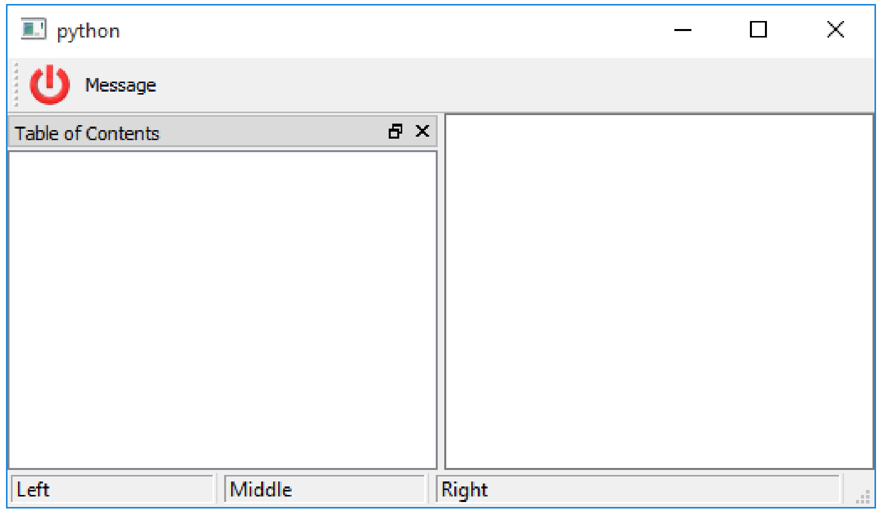
- The status bar can be retrieved using the QMainWindow.statusBar() function, and replaced using the QMainWindow.setStatusBar() function.

- Use the showMessage() slot to display a temporary message

- Normal and Permanent messages are displayed by creating a small widget (QLabel, QProgressBar or even QToolButton) and then adding it to the status bar using the addWidget() or the addPermanentWidget() function

Main Window. The Status Bar

```
...

# Creating left widget for statusbar.
statusLeft = QLabel("Left", self)
statusLeft.setFrameStyle(QFrame.Panel | QFrame.Sunken)
# Creating middle widget for statusbar
statusMiddle = QLabel("Middle", self)
statusMiddle.setFrameStyle(QFrame.Panel | QFrame.Sunken)
# Creating right widget for statusbar
statusRight = QLabel("Right", self)
statusRight.setFrameStyle(QFrame.Panel | QFrame.Sunken)

# Adds the given widget permanently to the status bar.
# The stretch parameter is used
# to compute a suitable size for the given widget.
self.statusBar().addPermanentWidget(statusLeft, 1)
self.statusBar().addPermanentWidget(statusMiddle, 1)
self.statusBar().addPermanentWidget(statusRight, 2)

...
```



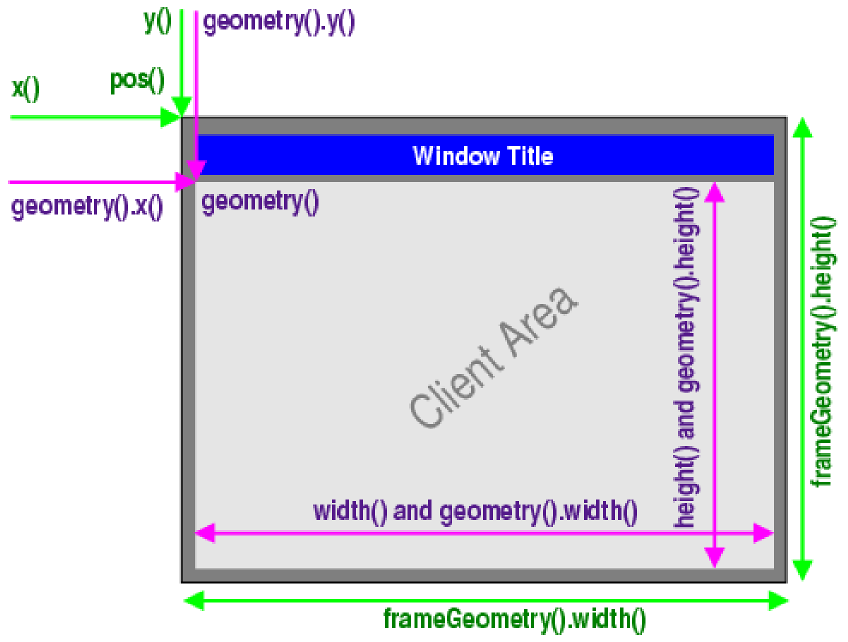
Window Geometry

QWidget provides several functions that deal with a widget's geometry.

Some of these functions operate on the pure client area (i.e. the window excluding the window frame), others include the window frame.

- **Including the window frame**
  x(), y(), frameGeometry(), pos(), and move()

- **Excluding the window frame**
  geometry(), width(), height(), rect(), and size()



Note that the distinction only matters for decorated top-level widgets.
For all child widgets, the frame geometry is equal to the widget's client geometry.