# Object Oriented Programming II

Dr. Burak Kaleci

March 31, 2019

# Content

# Introduction

- Widgets are basic building blocks of an application. PyQt5 has a wide range of various widgets, including label, buttons, boxes, sliders, or list boxes.

# QLabel

```
1 import sys
2 from PyQt5.QtWidgets import (QWidget, QLabel, QVBoxLayout, QApplication)
3
4 class Example(QWidget):
5
6     def __init__(self):
7         super().__init__()
8         self.initUI()
9
10    def initUI(self):
11
12        l1 = QLabel()
13        l2 = QLabel()
14        l3 = QLabel()
15        l1.setText("welcome to Python GUI Programming")
16        l2.setText("QLabel Example")
17        l3.setText("QVBoxLayout application")
18        vbox = QVBoxLayout()
19        vbox.addWidget(l1)
20        vbox.addStretch(1)
21        vbox.addWidget(l2)
22        vbox.addStretch(1)
23        vbox.addWidget(l3)
24        self.setLayout(vbox)
25        self.setWindowTitle("QLabel Demo")
26        self.show()
27
28 if __name__ == '__main__':
29
30    app = QApplication(sys.argv)
31    ex = Example()
32    sys.exit(app.exec_())
```

- A QLabel object acts as a placeholder to display non-editable text or image, or a movie of animated GIF. It can also be used as a mnemonic key for other widgets. Plain text, hyperlink or rich text can be displayed on the label.

# QLineEdit

```
1  import sys
2  from PyQt5.QtWidgets import (QWidget, QLabel, QLineEdit, QApplication)
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         self.lbl = QLabel(self)
12         qle = QLineEdit(self)
13         qle.move(60, 100)
14         self.lbl.move(60, 40)
15         qle.textChanged[str].connect(self.onChanged)
16         self.setGeometry(300, 300, 280, 170)
17         self.setWindowTitle('QLineEdit')
18         self.show()
19
20     def onChanged(self, text):
21         self.lbl.setText(text)
22         self.lbl.adjustSize()
23
24 if __name__ == '__main__':
25
26     app = QApplication(sys.argv)
27     ex = Example()
28     sys.exit(app.exec_())
```

- QLineEdit is a widget that allows to enter and edit a single line of plain text. There are undo and redo, cut and paste, and drag & drop functions available for the widget.
- This example shows a line edit widget and a label. The text that we key in the line edit is displayed immediately in the label widget.

# QLineEdit

```
1  import sys
2  from PyQt5.QtWidgets import (QWidget, QLabel, QLineEdit, QApplication)
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         self.lbl = QLabel(self)
12         qle = QLineEdit(self)
13         qle.move(60, 100)
14         self.lbl.move(60, 40)
15         qle.textChanged[str].connect(self.onChanged)
16         self.setGeometry(300, 300, 280, 170)
17         self.setWindowTitle('QLineEdit')
18         self.show()
19
20     def onChanged(self, text):
21         self.lbl.setText(text)
22         self.lbl.adjustSize()
23
24 if __name__ == '__main__':
25
26     app = QApplication(sys.argv)
27     ex = Example()
28     sys.exit(app.exec_())
```

- Line 12 creates the QLineEdit widget.
- If the text in the line edit widget changes, we call the onChanged() method (Line 15).
- Inside the onChanged() method, we set the typed text to the label widget. We call the adjustSize() method to adjust the size of the label to the length of the text (Lines 20-22).

# QPushButton

```
1 import sys
2 from PyQt5.QtWidgets import (QWidget, QLabel, QVBoxLayout, QApplication,QPushButton)
3
4 class Example(QWidget):
5
6     def __init__(self):
7         super().__init__()
8         self.initUI()
9
10    def initUI(self):
11        layout = QVBoxLayout()
12
13        self.b1 = QPushButton("Disabled")
14        self.b1.setEnabled(False)
15        layout.addWidget(self.b1)
16
17        self.b2 = QPushButton("&Default")
18        self.b2.setDefault(True)
19        self.b2.clicked.connect(self.buttonClicked)
20        layout.addWidget(self.b2)
21
22        self.lbl = QLabel(self)
23        layout.addWidget(self.lbl)
24
25        self.setLayout(layout)
26        self.setGeometry(300, 300, 280, 170)
27        self.setWindowTitle("Button Demo")
28        self.show()
29
30    def buttonClicked(self):
31
32        sender = self.sender()
33        self.lbl.setText(sender.text() + ' was pressed')
34
35 if __name__ == '__main__':
36
37     app = QApplication(sys.argv)
38     ex = Example()
39     sys.exit(app.exec_())
```

- In any GUI design, the command button is the most important and most often used control.
- Buttons with Save, Open, OK, Yes, No and Cancel etc. as caption are familiar to any computer user.
- In PyQt API, the QPushButton class object presents a button which when clicked can be programmed to invoke a certain function.

# QPushButton

```
1  import sys
2  from PyQt5.QtWidgets import (QWidget, QLabel, QVBoxLayout, QApplication,QPushButton)
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         layout = QVBoxLayout()
12
13         self.b1 = QPushButton("Disabled")
14         self.b1.setEnabled(False)
15         layout.addWidget(self.b1)
16
17         self.b2 = QPushButton("&Default")
18         self.b2.setDefault(True)
19         self.b2.clicked.connect(self.buttonClicked)
20         layout.addWidget(self.b2)
21
22         self.lbl = QLabel(self)
23         layout.addWidget(self.lbl)
24
25         self.setLayout(layout)
26         self.setGeometry(300, 300, 280, 170)
27         self.setWindowTitle("Button Demo")
28         self.show()
29
30     def buttonClicked(self):
31
32         sender = self.sender()
33         self.lbl.setText(sender.text() + ' was pressed')
34
35 if __name__ == '__main__':
36
37     app = QApplication(sys.argv)
38     ex = Example()
39     sys.exit(app.exec_())
```

- Button b1 is set to be disabled by using setEnabled() method (Line 14).
- Button b2 is set to default button by setDefault() method.
- Shortcut to its caption is created by prefixing & to the caption (&Default). As a result, by using the keyboard combination Alt+D, connected slot method will be called.

# QPushButton

```
1  import sys
2  from PyQt5.QtWidgets import (QWidget, QLabel, QVBoxLayout, QApplication,QPushButton)
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         layout = QVBoxLayout()
12
13         self.b1 = QPushButton("Disabled")
14         self.b1.setEnabled(False)
15         layout.addWidget(self.b1)
16
17         self.b2 = QPushButton("&Default")
18         self.b2.setDefault(True)
19         self.b2.clicked.connect(self.buttonClicked)
20         layout.addWidget(self.b2)
21
22         self.lbl = QLabel(self)
23         layout.addWidget(self.lbl)
24
25         self.setLayout(layout)
26         self.setGeometry(300, 300, 280, 170)
27         self.setWindowTitle("Button Demo")
28         self.show()
29
30     def buttonClicked(self):
31
32         sender = self.sender()
33         self.lbl.setText(sender.text() + ' was pressed')
34
35 if __name__ == '__main__':
36
37     app = QApplication(sys.argv)
38     ex = Example()
39     sys.exit(app.exec_())
```

- Button b2 is connected to buttonClicked slot method.
- Since the function is intended to retrieve caption of the clicked button, the buttonClicked is called and button name is appeared in a label.

# Toggle(Radio) button

```
1  from PyQt5.QtWidgets import (QWidget, QPushButton, QFrame, QApplication)
2  from PyQt5.QtGui import QColor
3  import sys
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10
11     def initUI(self):
12
13         self.col = QColor(0, 0, 0)
14
15         redb = QPushButton('Red', self)
16         redb.setCheckable(True)
17         redb.move(10, 10)
18         redb.clicked[bool].connect(self.setColor)
19
20         greenb = QPushButton('Green', self)
21         greenb.setCheckable(True)
22         greenb.move(10, 60)
23         greenb.clicked[bool].connect(self.setColor)
24
25         blueb = QPushButton('Blue', self)
26         blueb.setCheckable(True)
27         blueb.move(10, 110)
28         blueb.clicked[bool].connect(self.setColor)
29
30         self.square = QFrame(self)
31         self.square.setGeometry(150, 20, 100, 100)
32         self.square.setStyleSheet("QWidget { background-color: %s }" %
33             self.col.name())
34
35         self.setGeometry(300, 300, 280, 170)
36         self.setWindowTitle('Toggle button')
37         self.show()
```

- A toggle button is a QPushButton in a special mode. It is a button that has two states: pressed and not pressed.
- We toggle between these two states by clicking on it. There are situations where this functionality fits well.
- In our example, we create three toggle buttons and a QWidget. We set the background colour of the QWidget to black (Line 13).

# Toggle(Radio) button

```
1  from PyQt5.QtWidgets import (QWidget, QPushButton, QFrame, QApplication)
2  from PyQt5.QtGui import QColor
3  import sys
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10
11     def initUI(self):
12
13         self.col = QColor(0, 0, 0)
14
15         redb = QPushButton('Red', self)
16         redb.setCheckable(True)
17         redb.move(10, 10)
18         redb.clicked[bool].connect(self.setColor)
19
20         greenb = QPushButton('Green', self)
21         greenb.setCheckable(True)
22         greenb.move(10, 60)
23         greenb.clicked[bool].connect(self.setColor)
24
25         blueb = QPushButton('Blue', self)
26         blueb.setCheckable(True)
27         blueb.move(10, 110)
28         blueb.clicked[bool].connect(self.setColor)
29
30         self.square = QFrame(self)
31         self.square.setGeometry(150, 20, 100, 100)
32         self.square.setStyleSheet("QWidget { background-color: %s }" %
33             self.col.name())
34
35         self.setGeometry(300, 300, 280, 170)
36         self.setWindowTitle('Toggle button')
37         self.show()
```

- The toggle buttons will toggle the red, green, and blue parts of the colour value. The background colour depends on which toggle buttons is pressed.
- To create a toggle button, we create a QPushButton and make it checkable by calling the setCheckable() method (Lines 15 and 16).
- We connect a clicked signal to our user defined method. We use the clicked signal that operates with a Boolean value (Line 18).

# Toggle(Radio) button

```
40      def setColor(self, pressed):
41
42          source = self.sender()
43
44          if pressed:
45              val = 255
46          else: val = 0
47
48          if source.text() == "Red":
49              self.col.setRed(val)
50          elif source.text() == "Green":
51              self.col.setGreen(val)
52          else:
53              self.col.setBlue(val)
54
55          self.square.setStyleSheet("QFrame { background-color: %s }" %
56              self.col.name())
57
58
59  if __name__ == '__main__':
60
61      app = QApplication(sys.argv)
62      ex = Example()
63      sys.exit(app.exec_())
```

- We get the button which was toggled (Line 42).
- In case it is a red button, we update the red part of the colour accordingly (Lines 48-49).
- We use style sheets to change the background colour. The stylesheet is updated with setStyleSheet() method (Lines 55-56).

# QCheckBox

```
1  from PyQt5.QtWidgets import QWidget, QCheckBox, QApplication
2  from PyQt5.QtCore import Qt
3  import sys
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10
11     def initUI(self):
12         cb = QCheckBox('Show title', self)
13         cb.move(20, 20)
14         cb.toggle()
15         cb.stateChanged.connect(self.changeTitle)
16
17         self.setGeometry(300, 300, 250, 150)
18         self.setWindowTitle('QCheckBox')
19         self.show()
20
21     def changeTitle(self, state):
22
23         if state == Qt.Checked:
24             self.setWindowTitle('QCheckBox')
25         else:
26             self.setWindowTitle(' ')
27
28 if __name__ == '__main__':
29
30     app = QApplication(sys.argv)
31     ex = Example()
32     sys.exit(app.exec_())
```

- A QCheckBox is a widget that has two states: on and off. It is a box with a label.
- Checkboxes are typically used to represent features in an application that can be enabled or disabled.
- In our example, we will create a checkbox that will toggle the window title.
- Line 12 creates a QCheckBox object.

# QCheckBox

```
1  from PyQt5.QtWidgets import QWidget, QCheckBox, QApplication
2  from PyQt5.QtCore import Qt
3  import sys
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10
11     def initUI(self):
12         cb = QCheckBox('Show title', self)
13         cb.move(20, 20)
14         cb.toggle()
15         cb.stateChanged.connect(self.changeTitle)
16
17         self.setGeometry(300, 300, 250, 150)
18         self.setWindowTitle('QCheckBox')
19         self.show()
20
21     def changeTitle(self, state):
22
23         if state == Qt.Checked:
24             self.setWindowTitle('QCheckBox')
25         else:
26             self.setWindowTitle(' ')
27
28 if __name__ == '__main__':
29
30     app = QApplication(sys.argv)
31     ex = Example()
32     sys.exit(app.exec_())
```

- We have set the window title, so we also check the checkbox (Line 14).
- We connect the user defined changeTitle() method to the stateChanged signal. The changeTitle() method will toggle the window title (Line 15).
- The state of the widget is given to the changeTitle() method in the state variable.
- If the widget is checked, we set a title of the window. Otherwise, we set an empty string to the titlebar (Lines 21-26).

# QComboBox

```
1  from PyQt5.QtWidgets import (QWidget, QLabel, QComboBox, QApplication)
2  import sys
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         self.lbl = QLabel("Ubuntu", self)
12
13         combo = QComboBox(self)
14         combo.addItem("Ubuntu")
15         combo.addItem("Mandriva")
16         combo.addItem("Fedora")
17         combo.addItem("Arch")
18         combo.addItem("Gentoo")
19
20         combo.move(50, 50)
21         self.lbl.move(50, 150)
22
23         combo.activated[str].connect(self.onActivated)
24
25         self.setGeometry(300, 300, 300, 200)
26         self.setWindowTitle('QComboBox')
27         self.show()
28
29     def onActivated(self, text):
30         self.lbl.setText(text)
31         self.lbl.adjustSize()
32
33 if __name__ == '__main__':
34
35     app = QApplication(sys.argv)
36     ex = Example()
37     sys.exit(app.exec_())
```

- QComboBox is a widget that allows a user to choose from a list of options.
- The example shows a QComboBox and a QLabel. The combo box has a list of five options. These are the names of Linux distros.
- The label widget displays the selected option from the combo box.
- We create a QComboBox widget with five options (Lines 13-18).

# QComboBox

```
1  from PyQt5.QtWidgets import (QWidget, QLabel, QComboBox, QApplication)
2  import sys
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         self.lbl = QLabel("Ubuntu", self)
12
13         combo = QComboBox(self)
14         combo.addItem("Ubuntu")
15         combo.addItem("Mandriva")
16         combo.addItem("Fedora")
17         combo.addItem("Arch")
18         combo.addItem("Gentoo")
19
20         combo.move(50, 50)
21         self.lbl.move(50, 150)
22
23         combo.activated[str].connect(self.onActivated)
24
25         self.setGeometry(300, 300, 300, 200)
26         self.setWindowTitle('QComboBox')
27         self.show()
28
29     def onActivated(self, text):
30         self.lbl.setText(text)
31         self.lbl.adjustSize()
32
33 if __name__ == '__main__':
34
35     app = QApplication(sys.argv)
36     ex = Example()
37     sys.exit(app.exec_())
```

- Upon an item selection, we call the onActivated() method (Line 23).
- Inside the method, we set the text of the chosen item to the label widget. We adjust the size of the label (Lines 29-31).

# QSpinBox

```
1 import sys
2 from PyQt5.QtWidgets import (QWidget, QLabel, QVBoxLayout, QApplication,QSpinBox)
3
4 class Example(QWidget):
5
6     def __init__(self):
7         super().__init__()
8         self.initUI()
9
10    def initUI(self):
11        layout = QVBoxLayout()
12        self.l1 = QLabel("current value:")
13        layout.addWidget(self.l1)
14        self.sp = QSpinBox()
15        layout.addWidget(self.sp)
16        self.sp.valueChanged.connect(self.valuechange)
17
18        self.setLayout(layout)
19        self.setGeometry(300, 300, 280, 170)
20        self.setWindowTitle("SpinBox Demo")
21        self.show()
22
23    def valuechange(self):
24        self.l1.setText("current value:"+str(self.sp.value()))
25
26 if __name__ == '__main__':
27
28    app = QApplication(sys.argv)
29    ex = Example()
30    sys.exit(app.exec_())
```

- A QSpinBox object presents the user with a textbox which displays an integer with up/down button on its right.
- The value in the textbox increases/decreases if the up/down button is pressed.
- By default, the integer number in the box starts with 0, goes upto 99 and changes by step 1.

# QSpinBox

```
1  import sys
2  from PyQt5.QtWidgets import (QWidget, QLabel, QVBoxLayout, QApplication,QSpinBox)
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         layout = QVBoxLayout()
12         self.l1 = QLabel("current value:")
13         layout.addWidget(self.l1)
14         self.sp = QSpinBox()
15         layout.addWidget(self.sp)
16         self.sp.valueChanged.connect(self.valuechange)
17
18         self.setLayout(layout)
19         self.setGeometry(300, 300, 280, 170)
20         self.setWindowTitle("SpinBox Demo")
21         self.show()
22
23     def valuechange(self):
24         self.l1.setText("current value:"+str(self.sp.value()))
25
26 if __name__ == '__main__':
27
28     app = QApplication(sys.argv)
29     ex = Example()
30     sys.exit(app.exec_())
```

- QSpinBox object emits valueChanged() signal every time when up/own button is pressed. The associated slot function can retrieve current value of the widget by value() method.
- In our example, a label (l1) and spinbox (sp) put in vertical layout of a top window.
- The valueChanged() signal is connected to valuechange() method (Line 16).
- The valueChange() function displays the current value as caption of the label (Line 24).

# QSlider

```
1 from PyQt5.QtWidgets import (QWidget, QSlider, QLabel,
2         QApplication,QVBoxLayout)
3 from PyQt5.QtGui import QFont
4 from PyQt5.QtCore import Qt
5 import sys
6
7 class Example(QWidget):
8
9     def __init__(self):
10        super().__init__()
11        self.initUI()
12
13    def initUI(self):
14        layout = QVBoxLayout()
15        self.l1 = QLabel("Hello")
16        layout.addWidget(self.l1)
17
18        self.sl = QSlider(Qt.Horizontal, self)
19        self.sl.setMinimum(10)
20        self.sl.setMaximum(30)
21        self.sl.setValue(20)
22        self.sl.setTickPosition(QSlider.TicksBelow)
23        self.sl.setTickInterval(5)
24        layout.addWidget(self.sl)
25        self.sl.valueChanged.connect(self.valuechange)
26        self.setLayout(layout)
27        self.setGeometry(300, 300, 280, 170)
28        self.setWindowTitle('QSlider')
29        self.show()
30
31    def valuechange(self):
32      size = self.sl.value()
33      self.l1.setFont(QFont("Arial",size))
34
35 if __name__ == '__main__':
36    app = QApplication(sys.argv)
37    ex = Example()
38    sys.exit(app.exec_())
```

- A QSlider is a widget that has a simple handle. This handle can be pulled back and forth.
- This way we are choosing a value for a specific task.
- Sometimes using a slider is more natural than entering a number or using a spin box.
- A slider control can be displayed in horizontal or vertical manner by mentioning the orientation in the constructor.

# QSlider

```
1  from PyQt5.QtWidgets import (QWidget, QSlider, QLabel,
2          QApplication,QVBoxLayout)
3  from PyQt5.QtGui import QFont
4  from PyQt5.QtCore import Qt
5  import sys
6
7  class Example(QWidget):
8
9      def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         layout = QVBoxLayout()
15         self.l1 = QLabel("Hello")
16         layout.addWidget(self.l1)
17
18         self.sl = QSlider(Qt.Horizontal, self)
19         self.sl.setMinimum(10)
20         self.sl.setMaximum(30)
21         self.sl.setValue(20)
22         self.sl.setTickPosition(QSlider.TicksBelow)
23         self.sl.setTickInterval(5)
24         layout.addWidget(self.sl)
25         self.sl.valueChanged.connect(self.valuechange)
26         self.setLayout(layout)
27         self.setGeometry(300, 300, 280, 170)
28         self.setWindowTitle('QSlider')
29         self.show()
30
31     def valuechange(self):
32       size = self.sl.value()
33       self.l1.setFont(QFont("Arial",size))
34
35 if __name__ == '__main__':
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- In our example A Label and a horizontal slider is placed in a vertical layout. Sliders valueChanged() signal is connected to valuechange() method (Line 25).
- The slot function valuechange() reads current value of the slider and uses it as the size of font for labels caption (Lines 32 and 33).

# QTab

```
1 import sys
2 from PyQt5.QtWidgets import (QMainWindow, QApplication,
3         QPushButton, QWidget, QVBoxLayout,QHBoxLayout,
4         QFormLayout,QTabWidget,QLineEdit,QLabel,QCheckBox)
5
6 class App(QMainWindow):
7
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11
12    def initUI(self):
13
14        self.setGeometry(300, 300, 280, 170)
15        self.setWindowTitle('Tab button')
16
17        self.table_widget = MyTableWidget(self)
18        self.setCentralWidget(self.table_widget)
19
20        self.show()
21
```

- If a form has too many fields to be displayed simultaneously, they can be arranged in different pages placed under each tab of a Tabbed Widget.
- The QTabWidget provides a tab bar and a page area. The page under the first tab is displayed and others are hidden.
- The user can view any page by clicking on the desired tab.

# QTab

```
1  import sys
2  from PyQt5.QtWidgets import (QMainWindow, QApplication,
3          QPushButton, QWidget, QVBoxLayout,QHBoxLayout,
4          QFormLayout,QTabWidget,QLineEdit,QLabel,QCheckBox)
5
6  class App(QMainWindow):
7
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12     def initUI(self):
13
14         self.setGeometry(300, 300, 280, 170)
15         self.setWindowTitle('Tab button')
16
17         self.table_widget = MyTableWidget(self)
18         self.setCentralWidget(self.table_widget)
19
20         self.show()
21
```

- In our example, there are two classes.
- Class App inherited from QMainWindow class.
- The App is responsible from the main window.
- Line 17 creates an objects for tab.

# QTab

```
22 class MyTableWidget(QWidget):
23
24     def __init__(self, parent):
25         super(QWidget, self).__init__(parent)
26         self.initTabs()
27
28     def initTabs(self):
29
30         self.layout = QVBoxLayout(self)
31         self.tabs = QTabWidget()
32         self.tab1 = QWidget()
33         self.tab2 = QWidget()
34         self.tabs.resize(300,200)
35         self.tabs.addTab(self.tab1,"Tab 1")
36         self.tabs.addTab(self.tab2,"Tab 2")
37         self.tab1.layout = QFormLayout(self)
38         self.tab1.layout.addRow("Name",QLineEdit())
39         self.tab1.layout.addRow("Address",QLineEdit())
40         self.tabs.setTabText(0,"Contact Details")
41         self.tab1.setLayout(self.tab1.layout)
42         self.tab2.layout = QHBoxLayout()
43         self.tab2.layout.addWidget(QLabel("subjects"))
44         self.tab2.layout.addWidget(QCheckBox("Physics"))
45         self.tab2.layout.addWidget(QCheckBox("Maths"))
46         self.tabs.setTabText(1,"Education Details")
47         self.tab2.setLayout(self.tab2.layout)
48
49         self.layout.addWidget(self.tabs)
50         self.setLayout(self.layout)
```

- To add a table to a window, we create MyTableWidget class.
- We initialize the tab screen by creating a QTabWidget and two QWidgets for tabs (Lines 31-34).
- We then add these tabs to the tab widget (Lines 35 and 36).
- Finally we add the tabs to the widget (Lines 49 and 50).
- The lines between 37 and 47 create content of the tabs.

# QSplitter

```
1 from PyQt5.QtWidgets import (QWidget, QHBoxLayout, QFrame,
2     QSplitter, QStyleFactory, QApplication,QTextEdit)
3 from PyQt5.QtCore import Qt
4 import sys
5
6 class Example(QWidget):
7
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11
12    def initUI(self):
13        hbox = QHBoxLayout(self)
14        topleft = QFrame()
15        topleft.setFrameShape(QFrame.StyledPanel)
16        bottom = QFrame()
17        bottom.setFrameShape(QFrame.StyledPanel)
18        splitter1 = QSplitter(Qt.Horizontal)
19        textedit = QTextEdit()
20        splitter1.addWidget(topleft)
21        splitter1.addWidget(textedit)
22        splitter1.setSizes([100,200])
23        splitter2 = QSplitter(Qt.Vertical)
24        splitter2.addWidget(splitter1)
25        splitter2.addWidget(bottom)
26        hbox.addWidget(splitter2)
27        self.setLayout(hbox)
28        QApplication.setStyle(QStyleFactory.create('Cleanlooks'))
29
30        self.setGeometry(300, 300, 300, 200)
31        self.setWindowTitle('QSplitter')
32        self.show()
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- QSplitter lets the user control the size of child widgets by dragging the boundary between its children. In our example, we show three QFrame widgets organized with two splitters.
- In our example, we have a splitter object, splitter1, in which a frame and QTextEdit object are horizontally added.
- This splitter object splitter1 and a bottom frame object are added in another splitter, splitter2, vertically. The object splitters is finally added in the top level window.

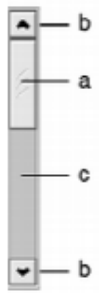
# QScrollBar

- A scrollbar control enables the user to access parts of the document that is outside the viewable area.
- It provides visual indicator to the current position.
- It has a slider by which a value between a preset range is set in analogous fashion.
- This value is usually correlated to bring a hidden data inside the viewport.

A slider

Two Scroll arrows

Page control



# QScrollBar

```
1 import sys
2 from PyQt5.QtWidgets import (QWidget, QLabel,QHBoxLayout,QScrollBar,QApplication)
3 from PyQt5.QtGui import QFont,QPalette,QColor
4
5 class Example(QWidget):
6
7     def __init__(self):
8         super().__init__()
9         self.initUI()
10
11    def initUI(self):
12        hbox = QHBoxLayout(self)
13        self.l1 = QLabel("Drag scrollbar sliders to change color")
14        self.l1.setFont(QFont("Arial",16))
15        hbox.addWidget(self.l1)
16        self.s1 = QScrollBar()
17        self.s1.setMaximum(255)
18        self.s1.sliderMoved.connect(self.sliderval)
19        self.s2 = QScrollBar()
20        self.s2.setMaximum(255)
21        self.s2.sliderMoved.connect(self.sliderval)
22        self.s3 = QScrollBar()
23        self.s3.setMaximum(255)
24        self.s3.sliderMoved.connect(self.sliderval)
25        hbox.addWidget(self.s1)
26        hbox.addWidget(self.s2)
27        hbox.addWidget(self.s3)
28        self.setGeometry(300, 300, 300, 200)
29        self.setWindowTitle('QScroll demo')
30        self.show()
31
32    def sliderval(self):
33        print (self.s1.value(),self.s2.value(), self.s3.value())
34        palette = QPalette()
35        c = QColor(self.s1.value(),self.s2.value(), self.s3.value(),255)
36        palette.setColor(QPalette.Foreground,c)
37        self.l1.setPalette(palette)
```

# QProgressBar

```
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12     def initUI(self):
13         self.pbar = QProgressBar(self)
14         self.pbar.setGeometry(30, 40, 200, 25)
15         self.btn = QPushButton('Start', self)
16         self.btn.move(40, 80)
17         self.btn.clicked.connect(self.doAction)
18         self.timer = QBasicTimer()
19         self.step = 0
20         self.setGeometry(300, 300, 280, 170)
21         self.setWindowTitle('QProgressBar')
22         self.show()
23
24     def timerEvent(self, e):
25
26         if self.step >= 100:
27             self.timer.stop()
28             self.btn.setText('Finished')
29             return
30
31         self.step = self.step + 1
32         self.pbar.setValue(self.step)
33
34     def doAction(self):
35
36         if self.timer.isActive():
37             self.timer.stop()
38             self.btn.setText('Start')
39         else:
40             self.timer.start(100, self)
41             self.btn.setText('Stop')
42
```

- A progress bar is a widget that is used when we process lengthy tasks.
- It is animated so that the user knows that the task is progressing.
- The QProgressBar widget provides a horizontal or a vertical progress bar in PyQt5 toolkit.
- The programmer can set the minimum and maximum value for the progress bar. The default values are 0 and 99.

# QProgressBar

```
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12     def initUI(self):
13         self.pbar = QProgressBar(self)
14         self.pbar.setGeometry(30, 40, 200, 25)
15         self.btn = QPushButton('Start', self)
16         self.btn.move(40, 80)
17         self.btn.clicked.connect(self.doAction)
18         self.timer = QBasicTimer()
19         self.step = 0
20         self.setGeometry(300, 300, 280, 170)
21         self.setWindowTitle('QProgressBar')
22         self.show()
23
24     def timerEvent(self, e):
25
26         if self.step >= 100:
27             self.timer.stop()
28             self.btn.setText('Finished')
29             return
30
31         self.step = self.step + 1
32         self.pbar.setValue(self.step)
33
34     def doAction(self):
35
36         if self.timer.isActive():
37             self.timer.stop()
38             self.btn.setText('Start')
39         else:
40             self.timer.start(100, self)
41             self.btn.setText('Stop')
42
```

- In our example we have a horizontal progress bar and a push button. The push button starts and stops the progress bar.
- Line 13 creates a QProgressBar object.
- To activate the progress bar, we use a timer object (Line 18).
- To launch a timer event, we call its start() method. This method has two parameters: the timeout and the object which will receive the events.
- Each QObject and its descendants have a timerEvent() event handler. In order to react to timer events, we reimplement the event handler.

# QCalendarWidget

```
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12     def initUI(self):
13         vbox = QVBoxLayout(self)
14         cal = QCalendarWidget(self)
15         cal.setGridVisible(True)
16         cal.clicked[QDate].connect(self.showDate)
17         vbox.addWidget(cal)
18
19         self.lbl = QLabel(self)
20         date = cal.selectedDate()
21         self.lbl.setText(date.toString())
22
23         vbox.addWidget(self.lbl)
24
25         self.setLayout(vbox)
26         self.setGeometry(300, 300, 350, 300)
27         self.setWindowTitle('Calendar')
28         self.show()
29
30     def showDate(self, date):
31
32         self.lbl.setText(date.toString())
```

- A QCalendarWidget provides a monthly based calendar widget. It allows a user to select a date in a simple and intuitive way.
- The example has a calendar widget and a label widget. The currently selected date is displayed in the label widget.
- Line 14 creates a QCalendarWidget object.
- If we select a date from the widget, a clicked[QDate] signal is emitted. We connect this signal to the user defined showDate() method.
- We retrieve the selected date by calling the selectedDate() method. Then we transform the date object into string and set it to the label widget.

# QPixmap

```
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12     def initUI(self):
13
14         hbox = QHBoxLayout(self)
15         pixmap = QPixmap("redrock.jpg")
16         lbl = QLabel(self)
17         lbl.setPixmap(pixmap)
18         hbox.addWidget(lbl)
19         self.setLayout(hbox)
20
21         self.move(300, 200)
22         self.setWindowTitle('Red Rock')
23         self.show()
24
```

- A QPixmap is one of the widgets used to work with images.
- It is optimized for showing images on screen. In our code example, we will use the QPixmap to display an image on the window.
- In our example, we display an image on the window.
- We create a QPixmap object. It takes the name of the file as a parameter (Line 15).
- We put the pixmap into the QLabel widget (Lines 16 and 17).

# Introduction

- PyQt5 painting system is able to render vector graphics, images, and outline font-based text.
- Painting is needed in applications when we want to change or enhance an existing widget, or if we are creating a custom widget from scratch.
- To do the drawing, we use the painting API provided by the PyQt5 toolkit.

# QPainter

```
1 import sys
2 from PyQt5.QtWidgets import QWidget, QApplication
3 from PyQt5.QtGui import QPainter, QColor, QFont
4 from PyQt5.QtCore import Qt
5
6 class Example(QWidget):
7
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11
12    def initUI(self):
13        self.text = "PyQt Example"
14        self.setGeometry(300, 300, 280, 170)
15        self.setWindowTitle('Drawing text')
16        self.show()
17    def paintEvent(self, event):
18        qp = QPainter()
19        qp.begin(self)
20        self.drawText(event, qp)
21        qp.end()
22    def drawText(self, event, qp):
23
24        qp.setPen(QColor(168, 34, 3))
25        qp.setFont(QFont('Decorative', 10))
26        qp.drawText(event.rect(), Qt.AlignCenter, self.text)
```

- QPainter performs low-level painting on widgets and other paint devices. It can draw everything from simple lines to complex shapes.
- The painting is done within the paintEvent() method. The painting code is placed between the begin() and end() methods of the QPainter object. It performs low-level painting on widgets and other paint devices.
- We begin with drawing some Unicode text on the client area of a window.
- In our example, we draw some text in Cylliric. The text is vertically and horizontally aligned.

# QPainter

```
1 import sys
2 from PyQt5.QtWidgets import QWidget, QApplication
3 from PyQt5.QtGui import QPainter, QColor, QFont
4 from PyQt5.QtCore import Qt
5
6 class Example(QWidget):
7
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11
12    def initUI(self):
13        self.text = "PyQt Example"
14        self.setGeometry(300, 300, 280, 170)
15        self.setWindowTitle('Drawing text')
16        self.show()
17    def paintEvent(self, event):
18        qp = QPainter()
19        qp.begin(self)
20        self.drawText(event, qp)
21        qp.end()
22    def drawText(self, event, qp):
23
24        qp.setPen(QColor(168, 34, 3))
25        qp.setFont(QFont('Decorative', 10))
26        qp.drawText(event.rect(), Qt.AlignCenter, self.text)
```

- Drawing is done within the paint event (Lines 17-21).
- The QPainter class is responsible for all the low-level painting.
- All the painting methods go between begin() and end() methods.
- The actual painting is delegated to the drawText() method.
- Lines 23 and 24 define a pen and a font which are used to draw the text.
- The drawText() method draws text on the window. The rect() method of the paint event returns the rectangle that needs to be updated. With the Qt.AlignCenter we align the text in both dimensions.

# Drawing points

```
1 from PyQt5.QtWidgets import QWidget, QApplication
2 from PyQt5.QtGui import QPainter
3 from PyQt5.QtCore import Qt
4 import sys, random
5
6 class Example(QWidget):
7
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11    def initUI(self):
12        self.setGeometry(300, 300, 300, 190)
13        self.setWindowTitle('Points')
14        self.show()
15    def paintEvent(self, e):
16        qp = QPainter()
17        qp.begin(self)
18        self.drawPoints(qp)
19        qp.end()
20    def drawPoints(self, qp):
21        qp.setPen(Qt.red)
22        size = self.size()
23
24        for i in range(1000):
25            x = random.randint(1, size.width()-1)
26            y = random.randint(1, size.height()-1)
27            qp.drawPoint(x, y)
```

- A point is the most simple graphics object that can be drawn. It is a small spot on the window.
- In our example, we draw randomly 1000 red points on the client area of the window.
- We set the pen to red colour. We use a predefined Qt.red colour constant (Line 21).

# Drawing points

```
1 from PyQt5.QtWidgets import QWidget, QApplication
2 from PyQt5.QtGui import QPainter
3 from PyQt5.QtCore import Qt
4 import sys, random
5
6 class Example(QWidget):
7
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11    def initUI(self):
12        self.setGeometry(300, 300, 300, 190)
13        self.setWindowTitle('Points')
14        self.show()
15    def paintEvent(self, e):
16        qp = QPainter()
17        qp.begin(self)
18        self.drawPoints(qp)
19        qp.end()
20    def drawPoints(self, qp):
21        qp.setPen(Qt.red)
22        size = self.size()
23
24        for i in range(1000):
25            x = random.randint(1, size.width()-1)
26            y = random.randint(1, size.height()-1)
27            qp.drawPoint(x, y)
```

- Each time we resize the window, a paint event is generated. We get the current size of the window with the size() method. We use the size of the window to distribute the points all over the client area of the window (Line 22).
- We draw the point with the drawPoint() method (Line 27).

# Colours

```
1 from PyQt5.QtWidgets import QWidget, QApplication
2 from PyQt5.QtGui import QPainter, QColor, QBrush
3 import sys
4
5 class Example(QWidget):
6
7     def __init__(self):
8         super().__init__()
9         self.initUI()
10    def initUI(self):
11        self.setGeometry(300, 300, 350, 100)
12        self.setWindowTitle('Colours')
13        self.show()
14    def paintEvent(self, e):
15        qp = QPainter()
16        qp.begin(self)
17        self.drawRectangles(qp)
18        qp.end()
19    def drawRectangles(self, qp):
20        col = QColor(0, 0, 0)
21        col.setNamedColor('#d4d4d4')
22        qp.setPen(col)
23        qp.setBrush(QColor(200, 0, 0))
24        qp.drawRect(10, 15, 90, 60)
25        qp.setBrush(QColor(255, 80, 0, 160))
26        qp.drawRect(130, 15, 90, 60)
27        qp.setBrush(QColor(25, 0, 90, 200))
28        qp.drawRect(250, 15, 90, 60)
29
```

- A colour is an object representing a combination of Red, Green, and Blue (RGB) intensity values. Valid RGB values are in the range from 0 to 255.
- We can define a colour in various ways. The most common are RGB decimal values or hexadecimal values.
- We can also use an RGBA value which stands for Red, Green, Blue, and Alpha. Here we add some extra information regarding transparency.
- Alpha value of 255 defines full opacity, 0 is for full transparency, e.g. the colour is invisible.

# Colours

```
1 from PyQt5.QtWidgets import QWidget, QApplication
2 from PyQt5.QtGui import QPainter, QColor, QBrush
3 import sys
4
5 class Example(QWidget):
6
7     def __init__(self):
8         super().__init__()
9         self.initUI()
10    def initUI(self):
11        self.setGeometry(300, 300, 350, 100)
12        self.setWindowTitle('Colours')
13        self.show()
14    def paintEvent(self, e):
15        qp = QPainter()
16        qp.begin(self)
17        self.drawRectangles(qp)
18        qp.end()
19    def drawRectangles(self, qp):
20        col = QColor(0, 0, 0)
21        col.setNamedColor('#d4d4d4')
22        qp.setPen(col)
23        qp.setBrush(QColor(200, 0, 0))
24        qp.drawRect(10, 15, 90, 60)
25        qp.setBrush(QColor(255, 80, 0, 160))
26        qp.drawRect(130, 15, 90, 60)
27        qp.setBrush(QColor(25, 0, 90, 200))
28        qp.drawRect(250, 15, 90, 60)
29
```

- In our example, we draw three coloured rectangles.
- Lines 20 and 21 define a colour using a hexadecimal notation.
- Lines 23 and 24 define a brush and draw a rectangle. A brush is an elementary graphics object which is used to draw the background of a shape.
- The drawRect() method accepts four parameters. The first two are x and y values on the axis. The third and fourth parameters are the width and height of the rectangle. The method draws the rectangle using the current pen and brush.

# QPen

```
1  from PyQt5.QtWidgets import QWidget, QApplication
2  from PyQt5.QtGui import QPainter, QPen
3  from PyQt5.QtCore import Qt
4  import sys
5
6  class Example(QWidget):
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10     def initUI(self):
11         self.setGeometry(300, 300, 280, 270)
12         self.setWindowTitle('Pen styles')
13         self.show()
14     def paintEvent(self, e):
15         qp = QPainter()
16         qp.begin(self)
17         self.drawLines(qp)
18         qp.end()
19     def drawLines(self, qp):
20         pen = QPen(Qt.black, 2, Qt.SolidLine)
21         qp.setPen(pen)
22         qp.drawLine(20, 40, 250, 40)
23         pen.setStyle(Qt.DashLine)
24         qp.setPen(pen)
25         qp.drawLine(20, 80, 250, 80)
26         pen.setStyle(Qt.DashDotLine)
27         qp.setPen(pen)
28         qp.drawLine(20, 120, 250, 120)
29         pen.setStyle(Qt.DotLine)
30         qp.setPen(pen)
31         qp.drawLine(20, 160, 250, 160)
32         pen.setStyle(Qt.DashDotDotLine)
33         qp.setPen(pen)
34         qp.drawLine(20, 200, 250, 200)
35         pen.setStyle(Qt.CustomDashLine)
36         pen.setDashPattern([1, 4, 5, 4])
37         qp.setPen(pen)
38         qp.drawLine(20, 240, 250, 240)
```

- The QPen is an elementary graphics object. It is used to draw lines, curves and outlines of rectangles, ellipses, polygons, or other shapes.
- In our example, we draw six lines. The lines are drawn in six different pen styles. There are five predefined pen styles. We can create also custom pen styles. The last line is drawn using a custom pen style.
- Line 20 creates a QPen object. The colour is black. The width is set to 2 pixels so that we can see the differences between the pen styles.

# QPen

```
1 from PyQt5.QtWidgets import QWidget, QApplication
2 from PyQt5.QtGui import QPainter, QPen
3 from PyQt5.QtCore import Qt
4 import sys
5
6 class Example(QWidget):
7     def __init__(self):
8         super().__init__()
9         self.initUI()
10    def initUI(self):
11        self.setGeometry(300, 300, 280, 270)
12        self.setWindowTitle('Pen styles')
13        self.show()
14    def paintEvent(self, e):
15        qp = QPainter()
16        qp.begin(self)
17        self.drawLines(qp)
18        qp.end()
19    def drawLines(self, qp):
20        pen = QPen(Qt.black, 2, Qt.SolidLine)
21        qp.setPen(pen)
22        qp.drawLine(20, 40, 250, 40)
23        pen.setStyle(Qt.DashLine)
24        qp.setPen(pen)
25        qp.drawLine(20, 80, 250, 80)
26        pen.setStyle(Qt.DashDotLine)
27        qp.setPen(pen)
28        qp.drawLine(20, 120, 250, 120)
29        pen.setStyle(Qt.DotLine)
30        qp.setPen(pen)
31        qp.drawLine(20, 160, 250, 160)
32        pen.setStyle(Qt.DashDotDotLine)
33        qp.setPen(pen)
34        qp.drawLine(20, 200, 250, 200)
35        pen.setStyle(Qt.CustomDashLine)
36        pen.setDashPattern([1, 4, 5, 4])
37        qp.setPen(pen)
38        qp.drawLine(20, 240, 250, 240)
```

- Qt.SolidLine is one of the predefined pen styles.
- Lines 35-37 define a custom pen style. We set a Qt.CustomDashLine pen style and call the setDashPattern() method.
- The list of numbers defines a style. There must be an even number of numbers. Odd numbers define a dash, even numbers space.
- The greater the number, the greater the space or the dash. Our pattern is 1px dash, 4px space, 5px dash, 4px space etc.

# QBrush

```
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11     def initUI(self):
12         self.setGeometry(300, 300, 355, 280)
13         self.setWindowTitle('Brushes')
14         self.show()
15     def paintEvent(self, e):
16         qp = QPainter()
17         qp.begin(self)
18         self.drawBrushes(qp)
19         qp.end()
20     def drawBrushes(self, qp):
21         brush = QBrush(Qt.SolidPattern)
22         qp.setBrush(brush)
23         qp.drawRect(10, 15, 90, 60)
24         brush.setStyle(Qt.Dense1Pattern)
25         qp.setBrush(brush)
26         qp.drawRect(130, 15, 90, 60)
27         brush.setStyle(Qt.Dense2Pattern)
28         qp.setBrush(brush)
29         qp.drawRect(250, 15, 90, 60)
30         brush.setStyle(Qt.DiagCrossPattern)
31         qp.setBrush(brush)
32         qp.drawRect(10, 105, 90, 60)
33         brush.setStyle(Qt.Dense5Pattern)
34         qp.setBrush(brush)
35         qp.drawRect(130, 105, 90, 60)
36         brush.setStyle(Qt.Dense6Pattern)
37         qp.setBrush(brush)
38         qp.drawRect(250, 105, 90, 60)
39         brush.setStyle(Qt.HorPattern)
40         qp.setBrush(brush)
41         qp.drawRect(10, 195, 90, 60)
42         brush.setStyle(Qt.VerPattern)
43         qp.setBrush(brush)
44         qp.drawRect(130, 195, 90, 60)
45         brush.setStyle(Qt.BDiagPattern)
46         qp.setBrush(brush)
47         qp.drawRect(250, 195, 90, 60)
```

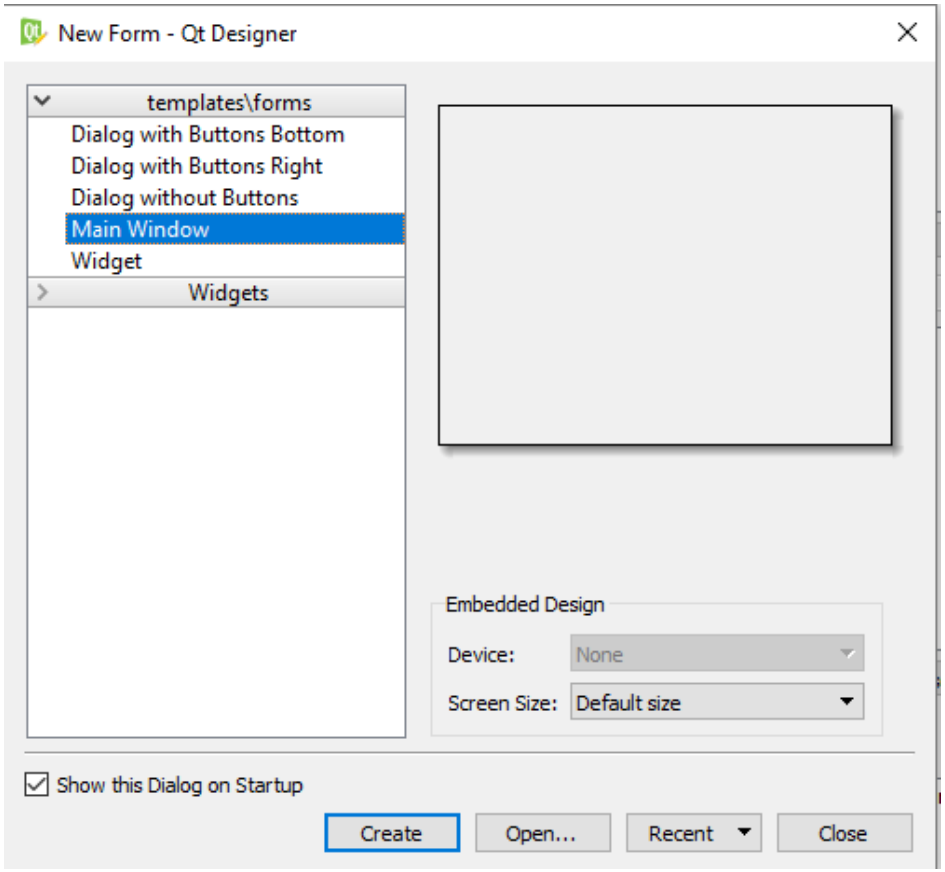
- QBrush is an elementary graphics object. It is used to paint the background of graphics shapes, such as rectangles, ellipses, or polygons.
- A brush can be of three different types: a predefined brush, a gradient, or a texture pattern.
- In our example, we draw nine different rectangles.
- Lines 21-23 define a brush object. We set it to the painter object and draw the rectangle by calling the drawRect() method.

# Bezier curve

```
1  from PyQt5.QtWidgets import QWidget, QApplication
2  from PyQt5.QtGui import QPainter, QPainterPath
3  from PyQt5.QtCore import Qt
4  import sys
5
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11     def initUI(self):
12         self.setGeometry(300, 300, 380, 250)
13         self.setWindowTitle('Bézier curve')
14         self.show()
15     def paintEvent(self, e):
16         qp = QPainter()
17         qp.begin(self)
18         qp.setRenderHint(QPainter.Antialiasing)
19         self.drawBezierCurve(qp)
20         qp.end()
21     def drawBezierCurve(self, qp):
22         path = QPainterPath()
23         path.moveTo(30, 30)
24         path.cubicTo(30, 30, 200, 350, 350, 30)
25
26         qp.drawPath(path)
```

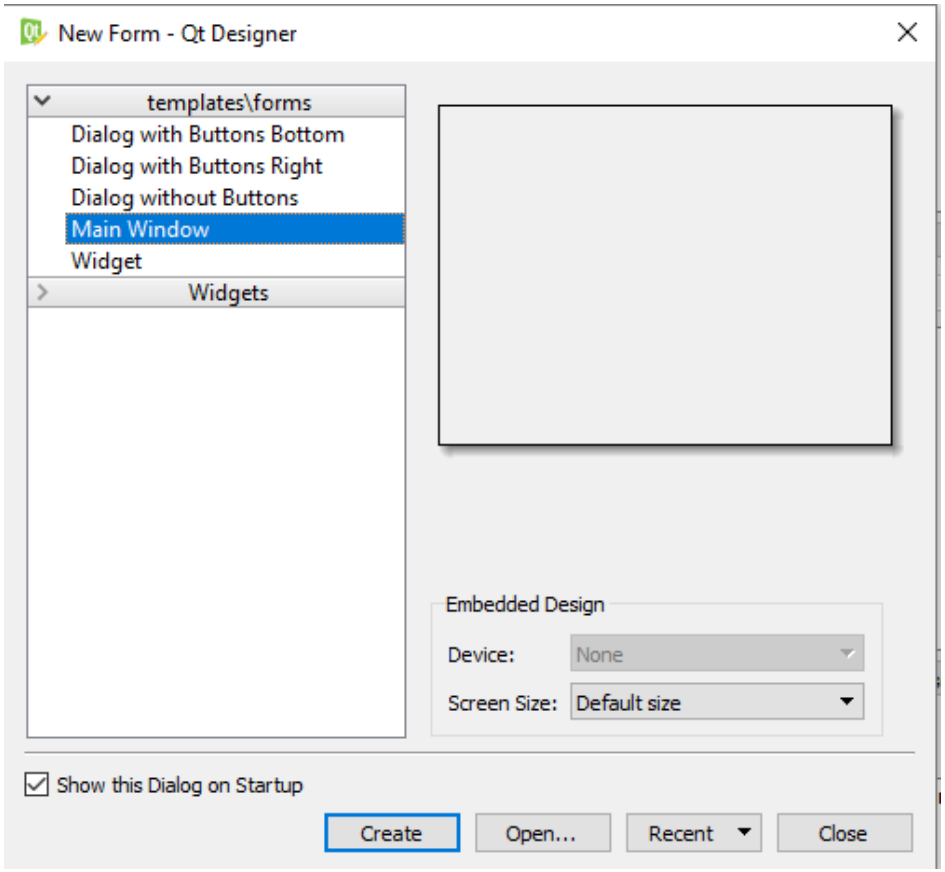
- Bezier curve is a cubic line. Bezier curve in PyQt5 can be created with QPainterPath.
- A painter path is an object composed of a number of graphical building blocks, such as rectangles, ellipses, lines, and curves.
- This example draws a Bezier curve.
- Lines 22-24 create a Bezier curve with QPainterPath path. The curve is created with cubicTo() method, which takes three points: starting point, control point, and ending point.
- The final path is drawn with drawPath() method (Line 26).

# Introduction




- Open designer.exe and you will see a dialog asking you about the form template you want.
- There are five templates available:
- Dialog with Buttons Bottom: Creates a form with OK and Cancel buttons at the bottom right of the form.
- Dialog with Buttons Right: Creates a form with OK and Cancel buttons at the top right of the form.
- Dialog without Buttons: Creates a blank form.
- Main Window: Creates a window with a menu bar and a toolbar and inherited from QMainWindow.

# Introduction




- Widget: Creates a widget which is inherited from QWidget class, unlike the Dialogs templates which inherit from QDialog class.
- QWidget is the base class for all GUI elements in the PyQt5.
- QDialog is used for asking the user about something, like asking the user to accept or reject something or maybe asking for an input and is based on QWidget.
- QMainWindow is the bigger template where you can place your toolbar, menu bar, status bar, and other widget and it doesnt have a built-in allowance for buttons like those in QDialog.

# Load .ui VS convert .ui to .py

- Open PyQt5 designer, and choose Main Window template and click create button.
- Then from the file menu, click save; PyQt5 designer will export your form into XML file with .ui extension.
- Now, in order to use this design, you have two ways:
  - Loading the .ui file in your Python code.
  - Converting the .ui file to a .py file using pyuic5.

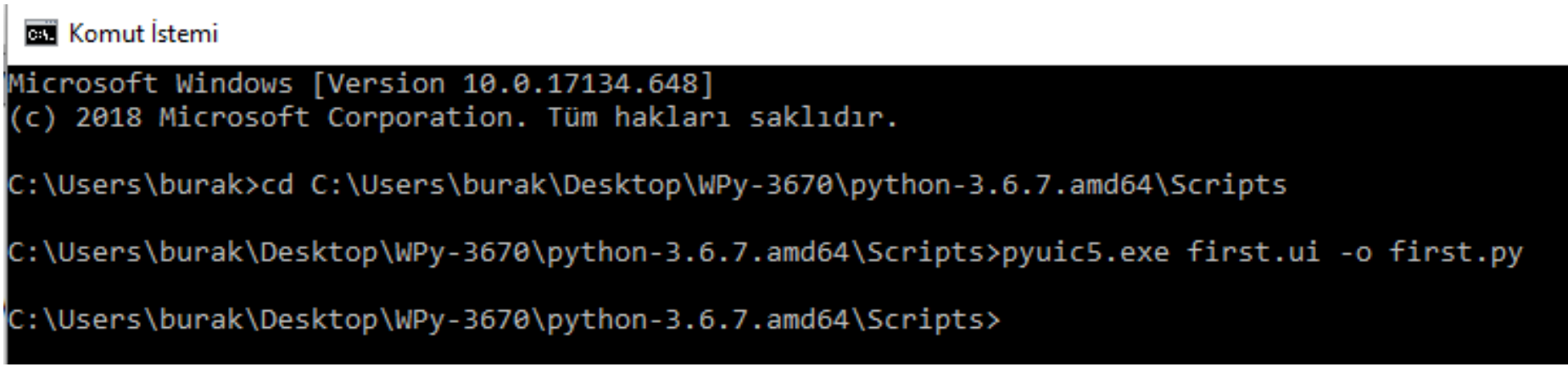
# Loading the .ui file in your Python code

- To load the .ui file in your Python code, you can use the loadUI() function from uic like this:

```
1 from PyQt5 import QtWidgets, uic
2 import sys
3
4 app = QtWidgets.QApplication([])
5 win = uic.loadUi("first.ui") #specify the location of your .ui file
6 win.show()
7 sys.exit(app.exec())
```

# Converting the .ui file to a .py file using pyuic5

- Now, lets try the second way by converting the .ui file to a Python code:




- Yes! A new file was created with the name first.py. Now, let's import that file to show our window.

# Converting the .ui file to a .py file using pyuic5

- If you run this code, you should see the same window again as we did in the first method.

```
1  from PyQt5 import QtWidgets
2
3  from first import Ui_MainWindow  # importing our generated file
4
5  import sys
6
7  class mywindow(QtWidgets.QMainWindow):
8
9      def __init__(self):
10
11         super(mywindow, self).__init__()
12
13         self.ui = Ui_MainWindow()
14
15         self.ui.setupUi(self)
16
17 app = QtWidgets.QApplication([])
18
19 application = mywindow()
20
21 application.show()
22
23 sys.exit(app.exec())
```