

Table of Contents

- Introduction
- PyQt5 date and time
- First programs in PyQt5
- Menus and toolbars in PyQt5
- Layout management in PyQt5
- Events and signals in PyQt5
- Dialogs in PyQt5
- PyQt5 widgets
- PyQt5 widgets II
- Drag and drop in PyQt5
- Painting in PyQt5
- Custom widgets in PyQt5
- Mini Project: Tetris

1. Introduction

PyQt is a GUI widgets toolkit. It is a Python interface for **Qt**, one of the most powerful, and popular cross-platform GUI library. PyQt was developed by RiverBank Computing Ltd. The latest version of PyQt can be downloaded from its official website – riverbankcomputing.com

PyQt API is a set of modules containing a large number of classes and functions. While **QtCore** module contains non-GUI functionality for working with file and directory etc., **QtGui** module contains all the graphical controls. In addition, there are modules for working with XML (**QtXml**), SVG (**QtSvg**), and SQL (**QtSql**), etc.

This is an introductory PyQt5 tutorial. The purpose of this tutorial is to get you started with the PyQt5 toolkit.

About PyQt5

PyQt5 is a set of Python bindings for Qt5 application framework from Digia. Qt library is one of the most powerful GUI libraries. The official home site for PyQt5 is www.riverbankcomputing.co.uk/news. PyQt5 is developed by Riverbank Computing.

PyQt5 is implemented as a set of Python modules. It has over 620 classes and 6000 functions and methods. It is a multiplatform toolkit which runs on all major operating systems, including Unix, Windows, and Mac OS. PyQt5 is dual licensed. Developers can choose between a GPL and a commercial license.

PyQt5 installation

We can install PyQt5 with the `pip` tool.

PyQt5 modules

PyQt5's classes are divided into several modules, including the following:

- QtCore
- QtGui
- QtWidgets
- QtMultimedia
- QtBluetooth
- QtNetwork

- QtPositioning
- Enginio
- QtWebSockets
- QtWebEngine
- QtWebEngineCore
- QtWebEngineWidgets
- QtXml
- QtSvg
- QtSql
- QtTest

The `QtCore` module contains the core non-GUI functionality. This module is used for working with time, files and directories, various data types, streams, URLs, mime types, threads or processes. The `QtGui` contains classes for windowing system integration, event handling, 2D graphics, basic imaging, fonts and text. The `QtWidgets` module contains classes that provide a set of UI elements to create classic desktop-style user interfaces. The `QtMultimedia` contains classes to handle multimedia content and APIs to access camera and radio functionality.

The `QtBluetooth` module contains classes to scan for devices and connect and interact with them. The `QtNetwork` module contains the classes for network programming. These classes facilitate the coding of TCP/IP and UDP clients and servers by making the network programming easier and more portable. The `QtPositioning` contains classes to determine a position by using a variety of possible sources, including satellite, Wi-Fi, or a text file. The `Enginio` module implements the client-side library for accessing the Qt Cloud Services Managed Application Runtime. The `QtWebSockets` module contains classes that implement the WebSocket protocol. The `QtWebEngine` module provides classes for integrating QML Web Engine objects with Python. The `QtWebEngineCore` contains the core Web Engine classes. The `QtWebEngineWidgets` contains the Chromium based web browser.

The `QtXml` contains classes for working with XML files. This module provides implementation for both SAX and DOM APIs. The `QtSvg` module provides classes for displaying the contents of SVG files. Scalable Vector Graphics (SVG) is a language for describing two-dimensional graphics and graphical applications in XML. The `QtSql` module provides classes for working with databases. The `QtTest` contains functions that enable unit testing of PyQt5 applications.

PyQt5 version

There are strings which hold the version of Qt and PyQt5.

Qt0101.py	
1	<code>from PyQt5.QtCore import QT_VERSION_STR</code>
2	<code>from PyQt5.Qt import PYQT_VERSION_STR</code>
3	
4	<code>print(QT_VERSION_STR)</code>
5	<code>print(PYQT_VERSION_STR)</code>

2. PyQt5 date and time

This part of the PyQt5 tutorial shows how to work with date and time in PyQt5.

QDate, QTime, QDateTime

PyQt5 has `QDate`, `QDateTime`, `QTime` classes to work with date and time. The `QDate` is a class for working with a calendar date in the Gregorian calendar. It has methods for determining the date, comparing, or manipulating dates. The `QTime` class works with a clock time. It provides methods for comparing time, determining the time and various other time manipulating methods. The `QDateTime` is a class that combines both `QDate` and `QTime` objects into one object.

Current date and time

PyQt5 has `currentDate`, `currentTime` and `currentDateTime` methods for determining current date and time.

```
Qt0201.py
1 from PyQt5.QtCore import QDate, QTime, QDateTime, Qt
2
3 now = QDate.currentDate()
4
5 print(now.toString(Qt.ISODate))
6 print(now.toString(Qt.DefaultLocaleLongDate))
7
8 datetime = QDateTime.currentDateTime()
9
10 print(datetime.toString())
11
12 time = QTime.currentTime()
13
14 print(time.toString(Qt.DefaultLocaleLongDate))
```

The example prints the current date, date and time, and time in various formats.

```
now = QDate.currentDate()
```

The `currentDate` method returns the current date.

```
print(now.toString(Qt.ISODate))
print(now.toString(Qt.DefaultLocaleLongDate))
```

The date is printed in two different formats by passing the values `Qt.ISODate` and `Qt.DefaultLocaleLongDate` to the `toString` method.

```
datetime = QDateTime.currentDateTime()
```

The `currentDateTime` returns the current date and time.

```
time = QDateTime.currentTime()
```

Finally, the `currentTime` method returns the current time.

Output:

```
2020-11-02
Monday, November 2, 2020
Mon Nov 2 03:13:34 2020
3:13:34 AM
```

UTC time

Our planet is a sphere; it revolves round its axis. The Earth rotates towards the east, so the Sun rises at different times in different locations. The Earth rotates once in about 24 hours. Therefore, the world was divided into 24 time zones. In each time zone, there is a different local time. This local time is often further modified by the daylight saving.

There is a pragmatic need for one global time. One global time helps to avoid confusion about time zones and daylight saving time. The UTC (Universal Coordinated time) was chosen to be the primary time standard. UTC is used in aviation, weather forecasts, flight plans, air traffic control clearances, and maps. Unlike local time, UTC does not change with a change of seasons.

Qt0202.py

```
1 from PyQt5.QtCore import QDateTime, Qt
2
3 now = QDateTime.currentTime()
4
5 print('Local datetime: ', now.toString(Qt.ISODate))
6 print('Universal datetime: ', now.toUTC().toString(Qt.ISODate))
7
8 print(f'The offset from UTC is: {now.offsetFromUtc()} seconds')
```

The example determines the current universal and local date and time.

```
print('Local datetime: ', now.toString(Qt.ISODate))
```

The `currentDateTime` method returns the current date and time expressed as local time. We can use the `toLocalTime` to convert a universal time into a local time.

```
print('Universal datetime: ', now.toUTC().toString(Qt.ISODate))
```

We get the universal time with the `toUTC` method from the date time object.

```
print(f'The offset from UTC is: {now.offsetFromUtc()} seconds')
```

The `offsetFromUtc` gives the difference between universal time and local time in seconds.

Output:

```
Local datetime: 2020-11-02T03:17:40
Universal datetime: 2020-11-01T19:17:40Z
The offset from UTC is: 28800 seconds
```

Number of days

The number of days in a particular month is returned by the `daysInMonth` method and the number of days in a year by the `daysInYear` method.

Qt0203.py

```
1 from PyQt5.QtCore import QDate, Qt
2
3 now = QDate.currentDate()
4
5 d = QDate(1945, 5, 7)
6
7 print(f'Days in month: {d.daysInMonth()}')
8 print(f'Days in year: {d.daysInYear()}')
```

The example prints the number of days in a month and year for the chosen date.

Output:

```
Days in month: 31
Days in year: 365
```

Difference in days

The `daysTo` method returns the number of days from a date to another date.

```
Qt0204.py
1 from PyQt5.QtCore import QDate
2
3 xmas1 = QDate(2019, 12, 24)
4 xmas2 = QDate(2020, 12, 24)
5
6 now = QDate.currentDate()
7
8 dayspassed = xmas1.daysTo(now)
9 print(f'{dayspassed} days have passed since last XMas')
10
11 nofdays = now.daysTo(xmas2)
12 print(f'There are {nofdays} days until next XMas')
```

The example calculates the number of days passed from the last XMas and the number of days until the next XMas.

Output:

```
314 days have passed since last XMas
There are 52 days until next XMas
```

Datetime arithmetic

We often need to add or subtract days, seconds, or years to a datetime value.

```
Qt0205.py
1 from PyQt5.QtCore import QDateTime, Qt
2
3 now = QDateTime.currentDateTime()
4
5 print(f'Today:', now.toString(Qt.ISODate))
6 print(f'Adding 12 days: {now.addDays(12).toString(Qt.ISODate)}')
7 print(f'Subtracting 22 days: {now.addDays(-22).toString(Qt.ISODate)}')
8
9 print(f'Adding 50 seconds: {now.addSecs(50).toString(Qt.ISODate)}')
10 print(f'Adding 3 months: {now.addMonths(3).toString(Qt.ISODate)}')
11 print(f'Adding 12 years: {now.addYears(12).toString(Qt.ISODate)}')
```

The example determines the current datetime and add or subtract days, seconds, months, and years.

Output:

```
Today: 2020-11-02T03:26:02
Adding 12 days: 2020-11-14T03:26:02
Subtracting 22 days: 2020-10-11T03:26:02
Adding 50 seconds: 2020-11-02T03:26:52
Adding 3 months: 2021-02-02T03:26:02
Adding 12 years: 2032-11-02T03:26:02
```

Daylight saving time

Daylight saving time (DST) is the practice of advancing clocks during summer months so that evening daylight lasts longer. The time is adjusted forward one hour in the beginning of spring and adjusted backward in the autumn to standard time.

Qt0206.py

```
1 from PyQt5.QtCore import QDateTime, QTimeZone, Qt
2
3 now = QDateTime.currentDateTime()
4
5 print(f'Time zone: {now.timeZoneAbbreviation()}')
6
7 if now.isDaylightTime():
8     print(f'The current date falls into DST time')
9 else:
10    print(f'The current date does not fall into DST time')
```

The example checks if the datetime is in the daylight saving time.

```
print(f'Time zone: {now.timeZoneAbbreviation()}')
```

The `timeZoneAbbreviation()` method returns the time zone abbreviation for the datetime.

```
if now.isDaylightTime():
    ...
```

The `isDaylightTime()` returns if the datetime falls in daylight saving time.

Output:

```
Time zone: Malay Peninsula Standard Time
The current date does not fall into DST time
```

The current date falls into DST time The program was executed in Bratislava, which is a city in Central Europe, during summer. Central European Summer Time (CEST) is 2 hours ahead of universal time. This time zone is a daylight saving time zone and is used in Europe and Antarctica. The standard time, which is used in winter, is Central European Time (CET).

Unix epoch

An epoch is an instant in time chosen as the origin of a particular era. For example in western Christian countries the time epoch starts from day 0, when Jesus was born. Another example is the French Republican Calendar which was used for twelve years. The epoch was the beginning of the Republican Era which was proclaimed on September 22, 1792, the day the First Republic was declared and the monarchy abolished.

Computers have their epochs too. One of the most popular is the Unix epoch. The Unix epoch is the time 00:00:00 UTC on 1 January 1970 (or 1970-01-01T00:00:00Z ISO 8601). The date and time in a computer is determined according to the number of seconds or clock ticks that have elapsed since the defined epoch for that computer or platform.

Unix time is the number of seconds elapsed since Unix epoch.

```
$ date +%s
1589617100
```

Unix date command can be used to get the Unix time. At this particular moment, 1589617100 seconds have passed since the Unix epoch.

Qt0207.py

```
1 from PyQt5.QtCore import QDateTime, Qt
2
3 now = QDateTime.currentDateTime()
4
5 unix_time = now.toSecsSinceEpoch()
6 print(unix_time)
7
8 d = QDateTime.fromSecsSinceEpoch(unix_time)
9 print(d.toString(Qt.ISODate))
```

The example prints the Unix time and converts it back to the QDateTime.

```
now = QDateTime.currentDateTime()
```

First, we retrieve the current date and time.

```
unix_time = now.toSecsSinceEpoch()
```

The `toSecsSinceEpoch()` returns the Unix time.

```
d = QDateTime.fromSecsSinceEpoch(unix_time)
```

With the `fromSecsSinceEpoch()` we convert the Unix time to `QDateTime`.

Output:

```
1604259114
2020-11-02T03:31:54
```

Julian day

Julian day refers to a continuous count of days since the beginning of the Julian Period. It is used primarily by astronomers. It should not be confused with the Julian calendar. The Julian Period started in 4713 BC. The Julian day number 0 is assigned to the day starting at noon on January 1, 4713 BC.

The Julian Day Number (JDN) is the number of days elapsed since the beginning of this period. The Julian Date (JD) of any instant is the Julian day number for the preceding noon plus the fraction of the day since that instant. (Qt does not compute this fraction.) Apart from astronomy, Julian dates are often used by military and mainframe programs.

Qt0208.py

```
1 from PyQt5.QtCore import QDate, Qt
2
3 now = QDate.currentDate()
4
5 print('Gregorian date for today:', now.toString(Qt.ISODate))
6 print('Julian day for today:', now.toJulianDay())
```

In the example, we compute the Gregorian date and the Julian day for today.

```
print('Julian day for today:', now.toJulianDay())
```

The Julian day is returned with the `toJulianDay()` method.

Output:

```
Gregorian date for today: 2020-11-02
Julian day for today: 2459156
```

Historical battles

With Julian day it is possible to do calculations that span centuries.

```
Qt0209.py
1 from PyQt5.QtCore import QDate, Qt
2
3 borodino_battle = QDate(1812, 9, 7)
4 slavkov_battle = QDate(1805, 12, 2)
5
6 now = QDate.currentDate()
7
8 j_today = now.toJulianDay()
9 j_borodino = borodino_battle.toJulianDay()
10 j_slavkov = slavkov_battle.toJulianDay()
11
12 d1 = j_today - j_slavkov
13 d2 = j_today - j_borodino
14
15 print(f'Days since Slavkov battle: {d1}')
16 print(f'Days since Borodino battle: {d2}')
```

The example counts the number of days passed since two historical events.

```
borodino_battle = QDate(1812, 9, 7)
slavkov_battle = QDate(1805, 12, 2)
```

We have two dates of battles of the Napoleonic era.

```
j_today = now.toJulianDay()
j_borodino = borodino_battle.toJulianDay()
j_slavkov = slavkov_battle.toJulianDay()
```

We compute the Julian days for today and for the Battles of Slavkov and Borodino.

```
d1 = j_today - j_slavkov
d2 = j_today - j_borodino
```

We compute the number of days passed since the two battles.

```
Days since Slavkov battle: 78498
Days since Borodino battle: 76027
```

When we run this script, 78498 days have passed since the Slavkov battle, and 76027 since the Borodino battle.

3. First programs in PyQt5

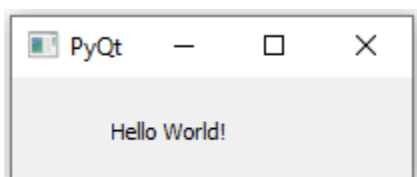
In this part of the PyQt5 tutorial we learn some basic functionality. The examples show a tooltip and an icon, close a window, show a message box and center a window on the desktop.

PyQt5 simple example

This is a simple example showing a small window. Yet we can do a lot with this window. We can resize it, maximise it or minimise it. This requires a lot of coding. Someone already coded this functionality. Because it is repeated in most applications, there is no need to code it over again. PyQt5 is a high level toolkit. If we would code in a lower level toolkit, the following code example could easily have hundreds of lines.

```
Qt0301.py
1 """
2 In this example, we create a simple window in PyQt5.
3 """
4 import sys
5 from PyQt5.QtWidgets import QApplication, QWidget
6
7 def main():
8     app = QApplication(sys.argv)
9
10    w = QWidget()
11    w.resize(250, 150)
12    w.move(300, 300)
13    w.setWindowTitle('Simple')
14    w.show()
15
16    sys.exit(app.exec_())
17
18 if __name__ == '__main__':
19     main()
```

The above code example shows a small window on the screen.



```
import sys
from PyQt5.QtWidgets import QApplication, QWidget
```

Here we provide the necessary imports. The basic widgets are located in `PyQt5.QtWidgets` module.

```
app = QApplication(sys.argv)
```

Every PyQt5 application must create an application object.

The `sys.argv` parameter is a list of arguments from a command line. Python scripts can be run from the shell. It is a way how we can control the startup of our scripts.

```
w = QWidget()
```

The `QWidget` widget is the base class of all user interface objects in PyQt5. We provide the default constructor for `QWidget`. The default constructor has no parent. A widget with no parent is called a window.

```
w.resize(250, 150)
```

The `resize()` method resizes the widget. It is 250px wide and 150px high.

```
w.move(300, 300)
```

The `move()` method moves the widget to a position on the screen at `x=300, y=300` coordinates.

```
w.setWindowTitle('Simple')
```

We set the title of the window with `setWindowTitle()`. The title is shown in the titlebar.

```
w.show()
```

The `show()` method displays the widget on the screen. A widget is first created in memory and later shown on the screen.

```
sys.exit(app.exec_())
```

Finally, we enter the mainloop of the application. The event handling starts from this point. The mainloop receives events from the window system and dispatches them to the application widgets. The mainloop ends if we call the `exit()` method or the main widget is destroyed. The `sys.exit()` method ensures a clean exit. The environment will be informed how the application ended.

The `exec_()` method has an underscore. It is because the `exec` is a Python keyword. And thus, `exec_()` was used instead.

An application icon

The application icon is a small image which is usually displayed in the top left corner of the titlebar. In the following example we will show how we do it in PyQt5. We will also introduce some new methods.

Some environments do not display icons in the titlebars. We need to enable them.

```
Qt0302.py
1  """
2  This example shows an icon in the titlebar of the window.
3  """
4  import sys
5  from PyQt5.QtWidgets import QApplication, QWidget
6  from PyQt5.QtGui import QIcon
7
8  class Example(QWidget):
9      def __init__(self):
10         super().__init__()
11         self.initUI()
12
13         def initUI(self):
14             self.setGeometry(300, 300, 300, 220)
15             self.setWindowTitle('Icon')
16             self.setWindowIcon(QIcon('web.png'))
17             self.show()
18
19 def main():
20     app = QApplication(sys.argv)
21     ex = Example()
22     sys.exit(app.exec_())
23
24 if __name__ == '__main__':
25     main()
```

The previous example was coded in a procedural style. Python programming language supports both procedural and object oriented programming styles. Programming in PyQt5 means programming in OOP.

```
class Example(QWidget):
    def __init__(self):
        super().__init__()
        ...
```

Three important things in object oriented programming are classes, data, and methods. Here we create a new class called Example. The Example class inherits from the QWidget class. This means that we call two constructors: the first one for the Example class and the second one for the inherited class. The super() method returns the parent object of the Example class and we call its constructor. The __init__() method is a constructor method in Python language.

```
self.initUI()
```

The creation of the GUI is delegated to the `initUI()` method.

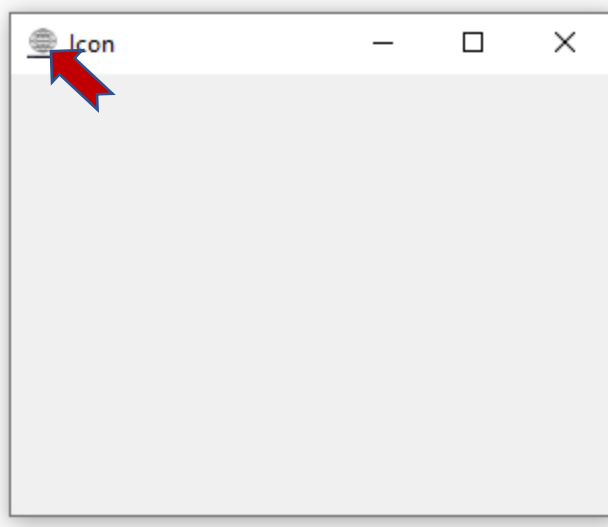
```
self.setGeometry(300, 300, 300, 220)
self.setWindowTitle('Icon')
self.setWindowIcon(QIcon('web.png'))
```

All three methods have been inherited from the `QWidget` class. The `setGeometry()` does two things: it locates the window on the screen and sets its size. The first two parameters are the x and y positions of the window. The third is the width and the fourth is the height of the window. In fact, it combines the `resize()` and `move()` methods in one method. The last method sets the application icon. To do this, we have created a `QIcon` object. The `QIcon` receives the path to our icon to be displayed.

```
def main():
    app = QApplication(sys.argv)
    ex = Example()
    sys.exit(app.exec_())
```

The application and example objects are created. The main loop is started.

Output:



Showing a tooltip in PyQt5

We can provide a balloon help for any of our widgets.

```
Qt0303.py
1  """
2  This example shows a tooltip on a window and a button.
3  """
4  import sys
5  from PyQt5.QtWidgets import (QWidget, QToolTip,
6                               QPushButton, QApplication)
7  from PyQt5.QtGui import QFont
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         QToolTip.setFont(QFont('SansSerif', 10))
16
17         self.setToolTip('This is a <b>QWidget</b> widget')
18
19         btn = QPushButton('Button', self)
20         btn.setToolTip('This is a <b>QPushButton</b> widget')
21         btn.resize(btn.sizeHint())
22         btn.move(50, 50)
23
24         self.setGeometry(300, 300, 300, 200)
25         self.setWindowTitle('Tooltips')
26         self.show()
27
28     def main():
29         app = QApplication(sys.argv)
30         ex = Example()
31         sys.exit(app.exec_())
32
33 if __name__ == '__main__':
34     main()
```

In this example, we show a tooltip for two PyQt5 widgets.

```
QToolTip.setFont(QFont('SansSerif', 10))
```

This static method sets a font used to render tooltips. We use a 10pt SansSerif font.

```
self.setToolTip('This is a <b>QWidget</b> widget')
```

To create a tooltip, we call the `setToolTip()` method. We can use rich text formatting.

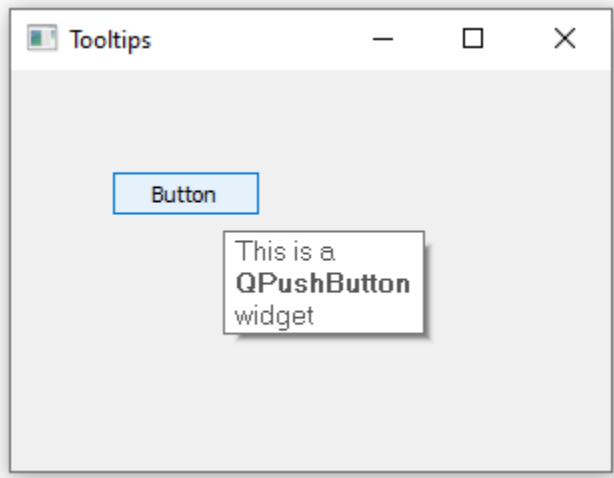
```
btn = QPushButton('Button', self)
btn.setToolTip('This is a <b>QPushButton</b> widget')
```

We create a push button widget and set a tooltip for it.

```
btn.resize(btn.sizeHint())  
btn.move(50, 50)
```

The button is being resized and moved on the window. The `sizeHint()` method gives a recommended size for the button.

Output:



Closing a window

The obvious way to close a window is to click on the x mark on the titlebar. In the next example, we show how we can programmatically close our window. We will briefly touch signals and slots.

The following is the constructor of a `QPushButton` widget that we use in our example.

```
QPushButton(string text, QWidget parent = None)
```

The `text` parameter is a text that will be displayed on the button. The `parent` is a widget on which we place our button. In our case it will be a `QWidget`. Widgets of an application form a hierarchy. In this hierarchy, most widgets have their parents. Widgets without parents are toplevel windows.

Qt0304.py

```
1 """
2 This program creates a quit button. When we press
3 the button, the application terminates.
4 """
5 import sys
6 from PyQt5.QtWidgets import QWidget, QPushButton, QApplication
7
8 class Example(QWidget):
9     def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         qbtn = QPushButton('Quit', self)
15         qbtn.clicked.connect(QApplication.instance().quit)
16         qbtn.resize(qbtn.sizeHint())
17         qbtn.move(50, 50)
18
19         self.setGeometry(300, 300, 350, 250)
20         self.setWindowTitle('Quit button')
21         self.show()
22
23 def main():
24     app = QApplication(sys.argv)
25     ex = Example()
26     sys.exit(app.exec_())
27
28 if __name__ == '__main__':
29     main()
```

In this example, we create a quit button. Upon clicking on the button, the application terminates.

```
qbtn = QPushButton('Quit', self)
```

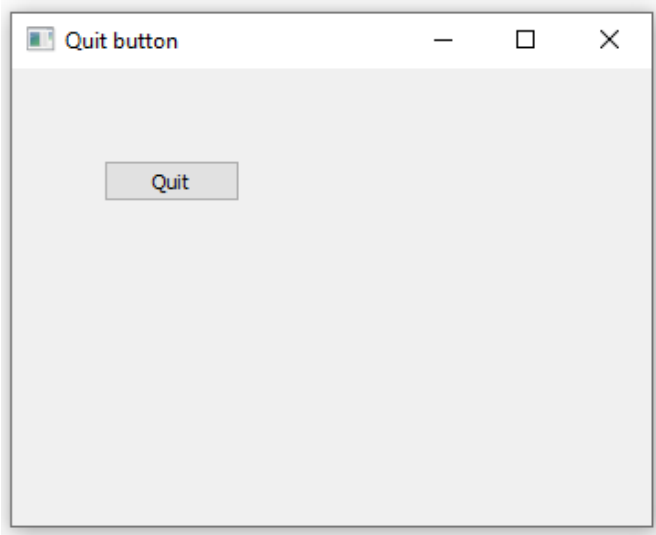
We create a push button. The button is an instance of the `QPushButton` class. The first parameter of the constructor is the label of the button. The second parameter is the parent widget. The parent widget is the `ExampleWidget`, which is a `QWidget` by inheritance.

```
qbtn.clicked.connect(QApplication.instance().quit)
```

The event processing system in PyQt5 is built with the signal & slot mechanism. If we click on the button, the signal `clicked` is emitted. The slot can be a Qt slot or any Python callable.

`QCoreApplication`, which is retrieved with `QApplication.instance()`, contains the main event loop—it processes and dispatches all events. The `clicked` signal is connected to the `quit()` method which terminates the application. The communication is done between two objects: the sender and the receiver. The sender is the push button, the receiver is the application object.

Output:



PyQt5 message box

By default, if we click on the `x` button on the titlebar, the `QWidget` is closed. Sometimes we want to modify this default behaviour. For example, if we have a file opened in an editor to which we did some changes. We show a message box to confirm the action.

Qt0305.py

```
1 """
2 This program shows a confirmation message box when we click on the close
3 button of the application window.
```

```

4  """
5  import sys
6  from PyQt5.QtWidgets import QWidget, QMessageBox, QApplication
7
8  class Example(QWidget):
9      def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         self.setGeometry(300, 300, 250, 150)
15         self.setWindowTitle('Message box')
16         self.show()
17
18     def closeEvent(self, event):
19         reply = QMessageBox.question(self, 'Message',
20                                     "Are you sure to quit?", QMessageBox.Yes |
21                                     QMessageBox.No, QMessageBox.No)
22
23         if reply == QMessageBox.Yes:
24             event.accept()
25         else:
26             event.ignore()
27
28     def main():
29         app = QApplication(sys.argv)
30         ex = Example()
31         sys.exit(app.exec_())
32
33 if __name__ == '__main__':
34     main()

```

If we close a `QWidget`, the `QCloseEvent` is generated. To modify the widget behaviour we need to reimplement the `closeEvent()` event handler.

```

reply = QMessageBox.question(self, 'Message',
                             "Are you sure to quit?", QMessageBox.Yes |
                             QMessageBox.No, QMessageBox.No)

```

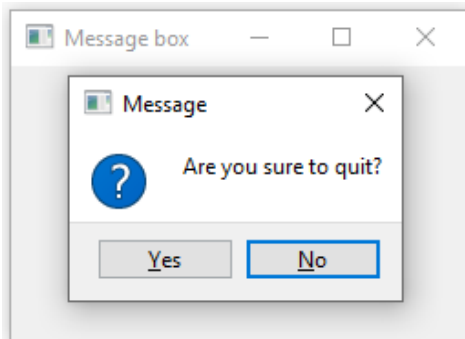
We show a message box with two buttons: Yes and No. The first string appears on the titlebar. The second string is the message text displayed by the dialog. The third argument specifies the combination of buttons appearing in the dialog. The last parameter is the default button. It is the button which has initially the keyboard focus. The return value is stored in the `reply` variable.

```

if reply == QtGui.QMessageBox.Yes:
    event.accept()
else:
    event.ignore()

```

Here we test the return value. If we click the Yes button, we accept the event which leads to the closure of the widget and to the termination of the application. Otherwise we ignore the close event.



Centering window on the screen

The following script shows how we can center a window on the desktop screen.

```
Qt0306.py
1  """
2  This program centers a window on the screen.
3  """
4  import sys
5  from PyQt5.QtWidgets import QWidget, QDesktopWidget, QApplication
6
7  class Example(QWidget):
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12         def initUI(self):
13             self.resize(250, 150)
14             self.center()
15
16             self.setWindowTitle('Center')
17             self.show()
18
19         def center(self):
20             qr = self.frameGeometry()
21             cp = QDesktopWidget().availableGeometry().center()
22             qr.moveCenter(cp)
23             self.move(qr.topLeft())
24
25     def main():
26         app = QApplication(sys.argv)
27         ex = Example()
28         sys.exit(app.exec_())
29
30     if __name__ == '__main__':
31         main()
```

The `QDesktopWidget` class provides information about the user's desktop, including the screen size.

```
self.center()
```

The code that will center the window is placed in the custom `center()` method.

```
qr = self.frameGeometry()
```

We get a rectangle specifying the geometry of the main window. This includes any window frame.

```
cp = QDesktopWidget().availableGeometry().center()
```

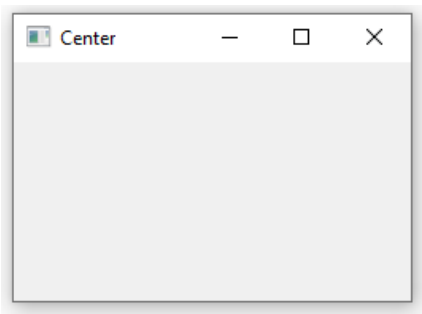
We figure out the screen resolution of our monitor. And from this resolution, we get the center point.

```
qr.moveCenter(cp)
```

Our rectangle has already its width and height. Now we set the center of the rectangle to the center of the screen. The rectangle's size is unchanged.

```
self.move(qr.topLeft())
```

We move the top-left point of the application window to the top-left point of the `qr` rectangle, thus centering the window on our screen.



4. Menus and toolbars in PyQt5

In this part of the PyQt5 tutorial, we create a statusbar, menubar and a toolbar. A menu is a group of commands located in a menubar. A toolbar has buttons with some common commands in the application. Statusbar shows status information, usually at the bottom of the application window.

QMainWindow

The `QMainWindow` class provides a main application window. This enables to create a classic application skeleton with a statusbar, toolbars, and a menubar.

Statusbar

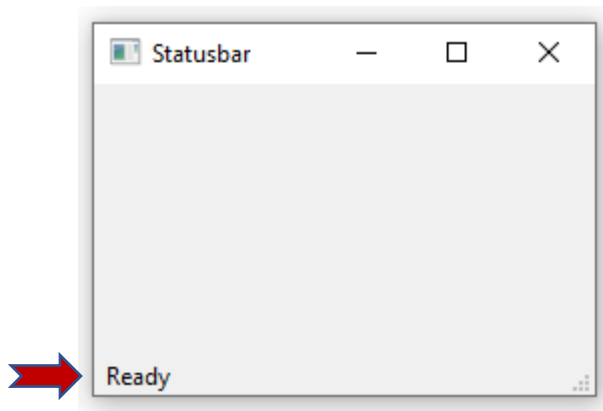
A statusbar is a widget that is used for displaying status information.

```
Qt0401.py
1 """
2 This program creates a statusbar.
3 """
4 import sys
5 from PyQt5.QtWidgets import QMainWindow, QApplication
6
7 class Example(QMainWindow):
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11
12    def initUI(self):
13        self.statusBar().showMessage('Ready')
14
15        self.setGeometry(300, 300, 250, 150)
16        self.setWindowTitle('Statusbar')
17        self.show()
18
19 def main():
20     app = QApplication(sys.argv)
21     ex = Example()
22     sys.exit(app.exec_())
23
24 if __name__ == '__main__':
25     main()
```

The statusbar is created with the help of the `QMainWindow` widget.

```
self.statusBar().showMessage('Ready')
```


To get the statusbar, we call the `statusBar()` method of the `QtGui.QMainWindow` class. The first call of the method creates a status bar. Subsequent calls return the statusbar object. The `showMessage()` displays a message on the statusbar.



PyQt5 simple menu

A menubar is a common part of a GUI application. It is a group of commands located in various menus. (Mac OS treats menubars differently. To get a similar outcome, we can add the following line: `menubar.setNativeMenuBar(False)`.)

```
Qt0402.py
1  """
2  This program creates a menubar. The menubar has one menu with an
3  exit action.
4  """
5  import sys
6  from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
7  from PyQt5.QtGui import QIcon
8
9  class Example(QMainWindow):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         exitAct = QAction(QIcon('exit.png'), '&Exit', self)
16         exitAct.setShortcut('Ctrl+Q')
17         exitAct.setStatusTip('Exit application')
18         exitAct.triggered.connect(qApp.quit)
19
20         self.statusBar()
21
22         menubar = self.menuBar()
23         fileMenu = menubar.addMenu('&File')
24         fileMenu.addAction(exitAct)
25
26         self.setGeometry(300, 300, 300, 200)
27         self.setWindowTitle('Simple menu')
```

```

28         self.show()
29
30
31     def main():
32         app = QApplication(sys.argv)
33         ex = Example()
34         sys.exit(app.exec_())
35
36     if __name__ == '__main__':
37         main()

```

In the above example, we create a menubar with one menu. This menu contains one action which terminates the application if selected. A statusbar is created as well. The action is accessible with the **Ctrl+Q** shortcut.

```

exitAct = QAction(QIcon('exit.png'), '&Exit', self)
exitAct.setShortcut('Ctrl+Q')
exitAct.setStatusTip('Exit application')

```

`QAction` is an abstraction for actions performed with a menubar, toolbar, or with a custom keyboard shortcut. In the above three lines, we create an action with a specific icon and an 'Exit' label. Furthermore, a shortcut is defined for this action. The third line creates a status tip which is shown in the statusbar when we hover a mouse pointer over the menu item.

```

exitAct.triggered.connect(qApp.quit)

```

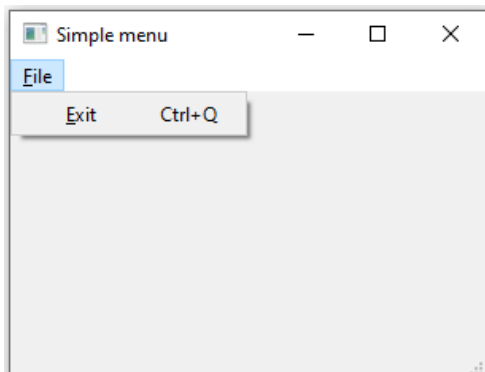
When we select this particular action, a triggered signal is emitted. The signal is connected to the `quit()` method of the `QApplication` widget. This terminates the application.

```

menubar = self.menuBar()
fileMenu = menubar.addMenu('&File')
fileMenu.addAction(exitAction)

```

The `menuBar()` method creates a menubar. We create a file menu with `addMenu()` and add the action with `addAction()`.



PyQt5 submenu

A submenu is a menu located inside another menu.

```
Qt0403.py
1  """
2  This program creates a submenu.
3  """
4  import sys
5  from PyQt5.QtWidgets import QMainWindow, QAction, QMenu, QApplication
6
7  class Example(QMainWindow):
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12         def initUI(self):
13             menubar = self.menuBar()
14             fileMenu = menubar.addMenu('File')
15
16             impMenu = QMenu('Import', self)
17             impAct = QAction('Import mail', self)
18             impMenu.addAction(impAct)
19
20             newAct = QAction('New', self)
21
22             fileMenu.addAction(newAct)
23             fileMenu.addMenu(impMenu)
24
25             self.setGeometry(300, 300, 300, 200)
26             self.setWindowTitle('Submenu')
27             self.show()
28
29         def main():
30             app = QApplication(sys.argv)
31             ex = Example()
32             sys.exit(app.exec_())
33
34         if __name__ == '__main__':
35             main()
```

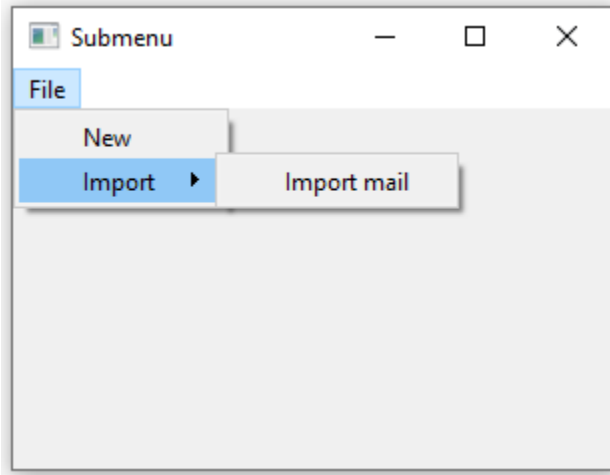
In the example, we have two menu items; one is located in the File menu and the other one in the File's Import submenu.

```
impMenu = QMenu('Import', self)
```

New menu is created with `QMenu`.

```
impAct = QAction('Import mail', self)
impMenu.addAction(impAct)
```

An action is added to the submenu with `addAction()`.



PyQt5 check menu

In the following example, we create a menu that can be checked and unchecked.

```
Qt0404.py
1  """
2  This program creates a checkable menu.
3  """
4  import sys
5  from PyQt5.QtWidgets import QMainWindow, QAction, QApplication
6
7  class Example(QMainWindow):
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12
13     def initUI(self):
14         self.statusbar = self.statusBar()
15         self.statusbar.showMessage('Ready')
16
17         menubar = self.menuBar()
18         viewMenu = menubar.addMenu('View')
19
20         viewStatAct = QAction('View statusbar', self, checkable=True)
21         viewStatAct.setStatusTip('View statusbar')
22         viewStatAct.setChecked(True)
23         viewStatAct.triggered.connect(self.toggleMenu)
24
25         viewMenu.addAction(viewStatAct)
26
27         self.setGeometry(300, 300, 300, 200)
28         self.setWindowTitle('Check menu')
29         self.show()
30
31     def toggleMenu(self, state):
32         if state:
```

```

33         self.statusbar.show()
34     else:
35         self.statusbar.hide()
36
37 def main():
38     app = QApplication(sys.argv)
39     ex = Example()
40     sys.exit(app.exec_())
41
42 if __name__ == '__main__':
43     main()

```

The code example creates a View menu with one action. The action shows or hides a statusbar. When the statusbar is visible, the menu item is checked.

```
viewStatAct = QAction('View statusbar', self, checkable=True)
```

With the checkable option we create a checkable menu.

```
viewStatAct.setChecked(True)
```

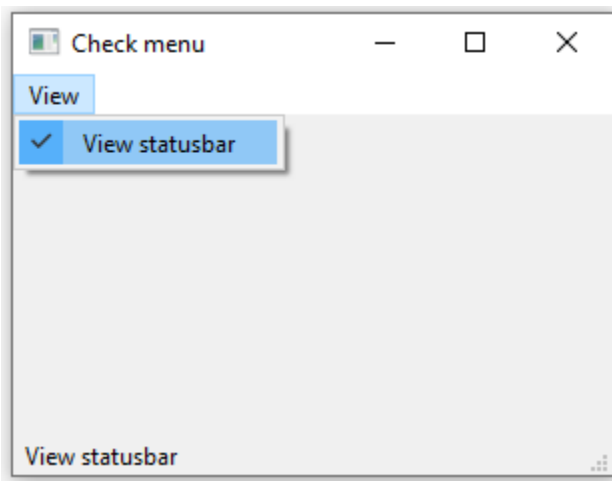
Since the statusbar is visible from the start, we check the action with `setChecked()` method.

```

def toggleMenu(self, state):
    if state:
        self.statusbar.show()
    else:
        self.statusbar.hide()

```

Depending on the state of the action, we show or hide the statusbar.



PyQt5 context menu

A context menu, also called a popup menu, is a list of commands that appears under some context. For example, in a Opera web browser when we right click on a web page, we get a context menu. Here we can reload a page, go back, or view a page source. If we right click on a toolbar, we get another context menu for managing toolbars.

```
Qt0405.py
1  """
2  This program creates a context menu.
3  """
4  import sys
5  from PyQt5.QtWidgets import QMainWindow, QApplication, QMenu, QApplication
6
7  class Example(QMainWindow):
8      def __init__(self):
9          super().__init__()
10         self.initUI()
11
12         def initUI(self):
13             self.setGeometry(300, 300, 300, 200)
14             self.setWindowTitle('Context menu')
15             self.show()
16
17         def contextMenuEvent(self, event):
18             cmenu = QMenu(self)
19
20             newAct = cmenu.addAction("New")
21             openAct = cmenu.addAction("Open")
22             quitAct = cmenu.addAction("Quit")
23             action = cmenu.exec_(self.mapToGlobal(event.pos()))
24
25             if action == quitAct:
26                 QApplication.quit()
27
28         def main():
29             app = QApplication(sys.argv)
30             ex = Example()
31             sys.exit(app.exec_())
32
33         if __name__ == '__main__':
34             main()
```

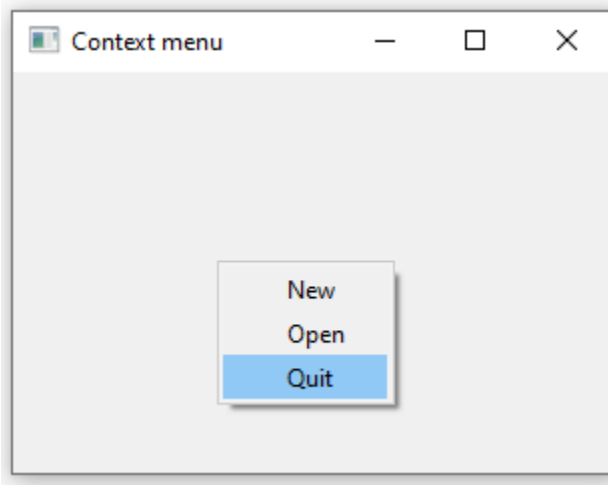
To work with a context menu, we have to reimplement the `contextMenuEvent()` method.

```
action = cmenu.exec_(self.mapToGlobal(event.pos()))
```

The context menu is displayed with the `exec_()` method. To get the coordinates of the mouse pointer from the event object. The `mapToGlobal()` method translates the widget coordinates to the global screen coordinates.

```
if action == quitAct:
    qApp.quit()
```

If the action returned from the context menu equals to quit action, we terminate the application.



PyQt5 toolbar

Menus group all commands that we can use in an application. Toolbars provide a quick access to the most frequently used commands.

Qt0406.py

```
1 """
2 This program creates a toolbar.
3 The toolbar has one action, which terminates the application,
4 if triggered.
5 """
6 import sys
7 from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
8 from PyQt5.QtGui import QIcon
9
10 class Example(QMainWindow):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         exitAct = QAction(QIcon('Exit.png'), 'Exit', self)
17         exitAct.setShortcut('Ctrl+Q')
18         exitAct.triggered.connect(qApp.quit)
19
20         self.toolbar = self.addToolBar('Exit')
21         self.toolbar.addAction(exitAct)
22
23         self.setGeometry(300, 300, 300, 200)
24         self.setWindowTitle('Toolbar')
```

```

25         self.show()
26
27     def main():
28         app = QApplication(sys.argv)
29         ex = Example()
30         sys.exit(app.exec_())
31
32     if __name__ == '__main__':
33         main()

```

In the above example, we create a simple toolbar. The toolbar has one tool action, an exit action which terminates the application when triggered.

```

exitAct = QAction(QIcon('exit24.png'), 'Exit', self)
exitAct.setShortcut('Ctrl+Q')
exitAct.triggered.connect(qApp.quit)

```

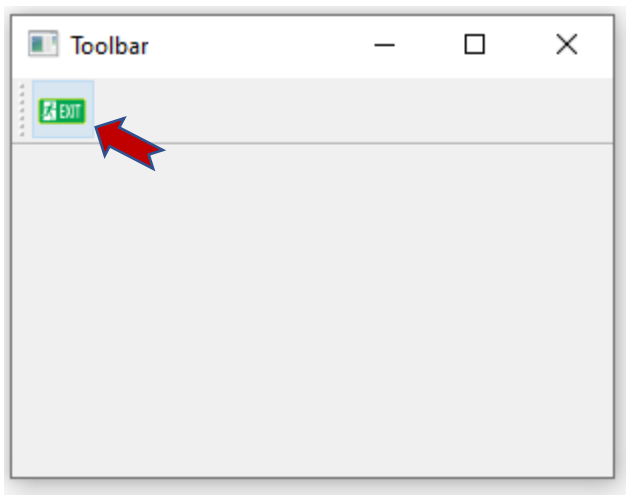
Similar to the menubar example above, we create an action object. The object has a label, icon, and a shortcut. A `quit()` method of the `QtGui.QMainWindow` is connected to the triggered signal.

```

self.toolbar = self.addToolBar('Exit')
self.toolbar.addAction(exitAction)

```

The toolbar is created with the `addToolBar()` method. We add an action object to the toolbar with `addAction()`.



PyQt5 main window

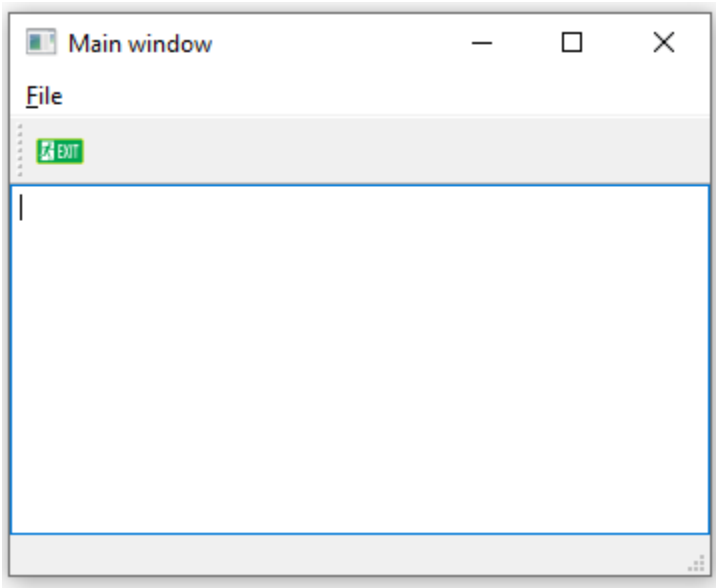
In the last example of this section, we create a menubar, toolbar, and a statusbar. We also create a central widget.

```
Qt0407.py
1  """
2  This program creates a skeleton of a classic GUI application
3  with a menubar, toolbar, statusbar, and a central widget.
4  """
5  import sys
6  from PyQt5.QtWidgets import QMainWindow, QTextEdit, QAction, QApplication
7  from PyQt5.QtGui import QIcon
8
9  class Example(QMainWindow):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         textEdit = QTextEdit()
16         self.setCentralWidget(textEdit)
17
18         exitAct = QAction(QIcon('Exit.png'), 'Exit', self)
19         exitAct.setShortcut('Ctrl+Q')
20         exitAct.setStatusTip('Exit application')
21         exitAct.triggered.connect(self.close)
22
23         self.statusBar()
24
25         menubar = self.menuBar()
26         fileMenu = menubar.addMenu('&File')
27         fileMenu.addAction(exitAct)
28
29         toolbar = self.addToolBar('Exit')
30         toolbar.addAction(exitAct)
31
32         self.setGeometry(300, 300, 350, 250)
33         self.setWindowTitle('Main window')
34         self.show()
35
36     def main():
37         app = QApplication(sys.argv)
38         ex = Example()
39         sys.exit(app.exec_())
40
41 if __name__ == '__main__':
42     main()
43
```

This code example creates a skeleton of a classic GUI application with a menubar, toolbar, and a statusbar.

```
textEdit = QTextEdit()  
self.setCentralWidget(textEdit)
```

Here we create a text edit widget. We set it to be the central widget of the `QMainWindow`. The central widget occupies all space that is left.



In this part of the PyQt5 tutorial, we worked with menus, toolbars, a statusbar, and a main application window.

5. Layout management in PyQt5

Layout management is the way how we place the widgets on the application window. We can place our widgets using *absolute positioning* or with *layout classes*. Managing the layout with layout managers is the preferred way of organizing our widgets.

Absolute positioning

The programmer specifies the position and the size of each widget in pixels. When you use absolute positioning, we have to understand the following limitations:

- The size and the position of a widget do not change if we resize a window
- Applications might look different on various platforms
- Changing fonts in our application might spoil the layout
- If we decide to change our layout, we must completely redo our layout, which is tedious and time consuming

The following example positions widgets in absolute coordinates.

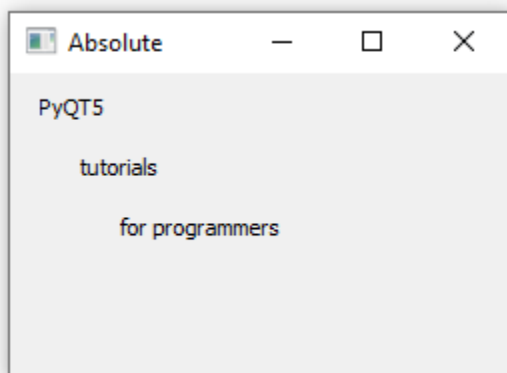
```
Qt0501.py
1 """
2 This example shows three labels on a window
3 using absolute positioning.
4 """
5 import sys
6 from PyQt5.QtWidgets import QWidget, QLabel, QApplication
7
8 class Example(QWidget):
9     def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         lbl1 = QLabel('PyQT5', self)
15         lbl1.move(15, 10)
16
17         lbl2 = QLabel('tutorials', self)
18         lbl2.move(35, 40)
19
20         lbl3 = QLabel('for programmers', self)
21         lbl3.move(55, 70)
22
23         self.setGeometry(300, 300, 250, 150)
24         self.setWindowTitle('Absolute')
25         self.show()
```

```
26
27 def main():
28     app = QApplication(sys.argv)
29     ex = Example()
30     sys.exit(app.exec_())
31
32 if __name__ == '__main__':
33     main()
```

We use the `move()` method to position our widgets. In our case these are labels. We position them by providing the x and y coordinates. The beginning of the coordinate system is at the left top corner. The x values grow from left to right. The y values grow from top to bottom.

```
lbl1 = QLabel('PyQT5', self)
lbl1.move(15, 10)
```

The label widget is positioned at $x=15$ and $y=10$.



PyQt5 QHBoxLayout

`QHBoxLayout` and `QVBoxLayout` are basic layout classes that line up widgets horizontally and vertically.

Imagine that we wanted to place two buttons in the right bottom corner. To create such a layout, we use one horizontal and one vertical box. To create the necessary space, we add a *stretch factor*.

```
Qt0502.py
1 """
2 In this example, we position two push buttons in the bottom-right
3 corner of the window.
4 """
5 import sys
6 from PyQt5.QtWidgets import (QWidget, QPushButton,
7                               QHBoxLayout, QVBoxLayout, QApplication)
8
9 class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         okButton = QPushButton("OK")
16         cancelButton = QPushButton("Cancel")
17
18         hbox = QHBoxLayout()
19         hbox.addStretch(1)
20         hbox.addWidget(okButton)
21         hbox.addWidget(cancelButton)
22
23         vbox = QVBoxLayout()
24         vbox.addStretch(1)
25         vbox.addLayout(hbox)
26
27         self.setLayout(vbox)
28
29         self.setGeometry(300, 300, 300, 150)
30         self.setWindowTitle('Buttons')
31         self.show()
32
33     def main():
34         app = QApplication(sys.argv)
35         ex = Example()
36         sys.exit(app.exec_())
37
38 if __name__ == '__main__':
39     main()
```

The example places two buttons in the bottom-right corner of the window. They stay there when we resize the application window. We use both a QHBoxLayout and a QVBoxLayout.

```
okButton = QPushButton("OK")
cancelButton = QPushButton("Cancel")
```

Here we create two push buttons.

```
hbox = QHBoxLayout()
hbox.addStretch(1)
hbox.addWidget(okButton)
hbox.addWidget(cancelButton)
```

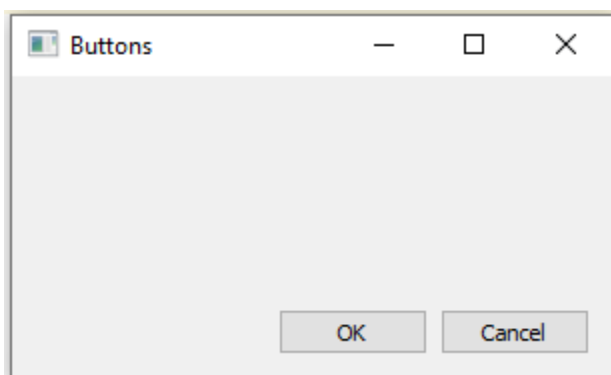
We create a horizontal box layout and add a stretch factor and both buttons. The stretch adds a stretchable space before the two buttons. This will push them to the right of the window.

```
vbox = QVBoxLayout()
vbox.addStretch(1)
vbox.addLayout(hbox)
```

The horizontal layout is placed into the vertical layout. The stretch factor in the vertical box will push the horizontal box with the buttons to the bottom of the window.

```
self.setLayout(vbox)
```

Finally, we set the main layout of the window.



PyQt5 QGridLayout

QGridLayout is the most universal layout class. It divides the space into rows and columns.

```
Qt0503.py
1  """
2  In this example, we create a skeleton of a calculator
3  using QGridLayout.
4  """
5  import sys
6  from PyQt5.QtWidgets import (QWidget, QGridLayout,
7                               QPushButton, QApplication)
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         grid = QGridLayout()
16         self.setLayout(grid)
17
18         names = ['Cls', 'Bck', '', 'Close',
19                '7', '8', '9', '/',
20                '4', '5', '6', '*',
21                '1', '2', '3', '-',
22                '0', '.', '=', '+']
23
24         positions = [(i, j) for i in range(5) for j in range(4)]
25
26         for position, name in zip(positions, names):
27             if name == '':
28                 continue
29             button = QPushButton(name)
30             grid.addWidget(button, *position)
31
32         self.move(300, 150)
33         self.setWindowTitle('Calculator')
34         self.show()
35
36     def main():
37         app = QApplication(sys.argv)
38         ex = Example()
39         sys.exit(app.exec_())
40
41 if __name__ == '__main__':
42     main()
```

In our example, we create a grid of buttons.

```
grid = QGridLayout()
self.setLayout(grid)
```

The instance of a `QGridLayout` is created and set to be the layout for the application window.

```
names = [ 'Cls', 'Bck', ' ', 'Close',  
          '7', '8', '9', '/',  
          '4', '5', '6', '*',  
          '1', '2', '3', '-',  
          '0', '.', '=', '+']
```

These are the labels used later for buttons.

```
positions = [(i,j) for i in range(5) for j in range(4)]
```

We create a list of positions in the grid.

```
for position, name in zip(positions, names):  
    if name == '':  
        continue  
    button = QPushButton(name)  
    grid.addWidget(button, *position)
```

Buttons are created and added to the layout with the `addWidget()` method.



Review example

Widgets can span multiple columns or rows in a grid. In the next example we illustrate this.

Qt0504.py

```
1 """
2 In this example, we create a bit more complicated window layout using
3 the QGridLayout manager.
4 """
5 import sys
6 from PyQt5.QtWidgets import (QWidget, QLabel, QLineEdit,
7                               QTextEdit, QGridLayout, QApplication)
8
9 class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         title = QLabel('Title')
16         author = QLabel('Author')
17         review = QLabel('Review')
18
19         titleEdit = QLineEdit()
20         authorEdit = QLineEdit()
21         reviewEdit = QTextEdit()
22
23         grid = QGridLayout()
24         grid.setSpacing(10)
25
26         grid.addWidget(title, 1, 0)
27         grid.addWidget(titleEdit, 1, 1)
28
29         grid.addWidget(author, 2, 0)
30         grid.addWidget(authorEdit, 2, 1)
31
32         grid.addWidget(review, 3, 0)
33         grid.addWidget(reviewEdit, 3, 1, 5, 1)
34
35         self.setLayout(grid)
36
37         self.setGeometry(300, 300, 350, 300)
38         self.setWindowTitle('Review')
39         self.show()
40
41     def main():
42         app = QApplication(sys.argv)
43         ex = Example()
44         sys.exit(app.exec_())
45
46 if __name__ == '__main__':
47     main()
```

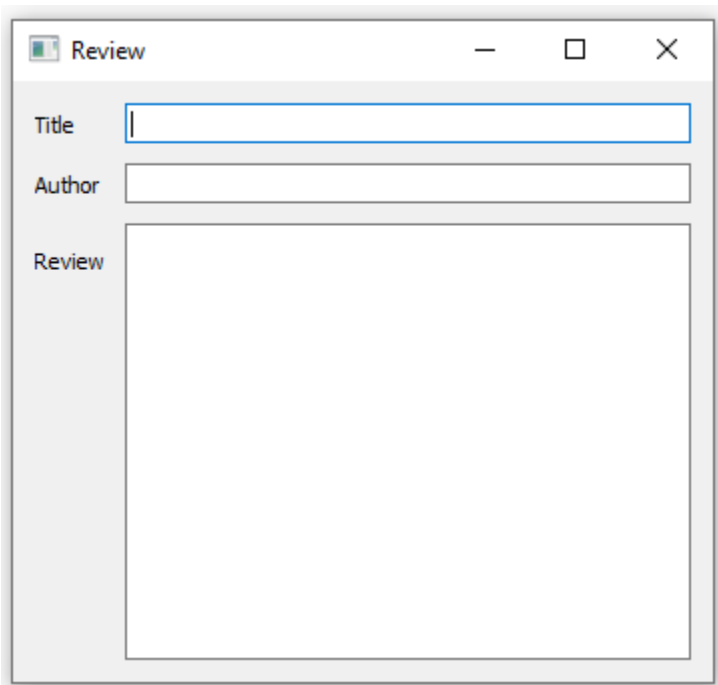
We create a window in which we have three labels, two line edits and one text edit widget. The layout is done with the `QGridLayout`.

```
grid = QGridLayout()  
grid.setSpacing(10)
```

We create a grid layout and set spacing between widgets.

```
grid.addWidget(reviewEdit, 3, 1, 5, 1)
```

If we add a widget to a grid, we can provide row span and column span of the widget. In our case, we make the `reviewEdit` widget span 5 rows.



This part of the PyQt5 tutorial was dedicated to layout management.


```

10 class Example(QWidget):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         lcd = QLCDNumber(self)
17         sld = QSlider(Qt.Horizontal, self)
18
19         vbox = QVBoxLayout()
20         vbox.addWidget(lcd)
21         vbox.addWidget(sld)
22
23         self.setLayout(vbox)
24         sld.valueChanged.connect(lcd.display)
25
26         self.setGeometry(300, 300, 250, 150)
27         self.setWindowTitle('Signal and slot')
28         self.show()
29
30 def main():
31     app = QApplication(sys.argv)
32     ex = Example()
33     sys.exit(app.exec_())
34
35 if __name__ == '__main__':
36     main()

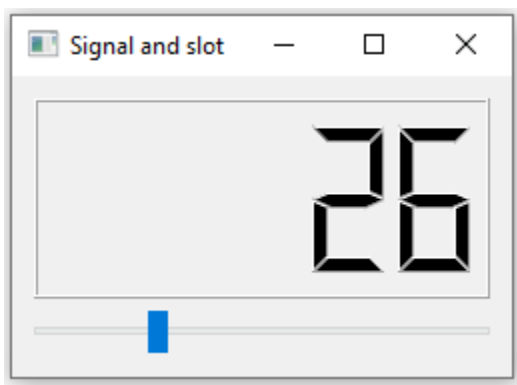
```

In our example, we display a `QtGui.QLCDNumber` and a `QtGui.QSlider`. We change the `lcd` number by dragging the slider knob.

```
sld.valueChanged.connect(lcd.display)
```

Here we connect a `valueChanged` signal of the slider to the `display` slot of the `lcd` number.

The *sender* is an object that sends a signal. The *receiver* is the object that receives the signal. The *slot* is the method that reacts to the signal.



PyQt5 reimplementing event handler

Events in PyQt5 are processed often by reimplementing event handlers.

```
Qt0602.py
1  """
2  In this example, we reimplement an event handler.
3  """
4  import sys
5  from PyQt5.QtCore import Qt
6  from PyQt5.QtWidgets import QWidget, QApplication
7
8  class Example(QWidget):
9      def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         self.setGeometry(300, 300, 250, 150)
15         self.setWindowTitle('Event handler')
16         self.show()
17
18     def keyPressEvent(self, e):
19         if e.key() == Qt.Key_Escape:
20             self.close()
21
22 def main():
23     app = QApplication(sys.argv)
24     ex = Example()
25     sys.exit(app.exec_())
26
27 if __name__ == '__main__':
28     main()
```

In our example, we reimplement the `keyPressEvent()` event handler.

```
def keyPressEvent(self, e):
    if e.key() == Qt.Key_Escape:
        self.close()
```

If we click the **Escape** button, the application terminates.

Event object in PyQt5

Event object is a Python object that contains a number of attributes describing the event. Event object is specific to the generated event type.

Qt0603.py

```
1 """
2 In this example, we display the x and y
3 coordinates of a mouse pointer in a label widget.
4 """
5 import sys
6 from PyQt5.QtCore import Qt
7 from PyQt5.QtWidgets import QWidget, QApplication, QGridLayout, QLabel
8
9 class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         grid = QGridLayout()
16
17         x = 0
18         y = 0
19
20         self.text = f'x: {x}, y: {y}'
21
22         self.label = QLabel(self.text, self)
23         grid.addWidget(self.label, 0, 0, Qt.AlignTop)
24
25         self.setMouseTracking(True)
26
27         self.setLayout(grid)
28
29         self.setGeometry(300, 300, 450, 300)
30         self.setWindowTitle('Event object')
31         self.show()
32
33     def mouseMoveEvent(self, e):
34         x = e.x()
35         y = e.y()
36
37         text = f'x: {x}, y: {y}'
38         self.label.setText(text)
39
40 def main():
41     app = QApplication(sys.argv)
42     ex = Example()
43     sys.exit(app.exec_())
44
45 if __name__ == '__main__':
46     main()
```

In this example, we display the x and y coordinates of a mouse pointer in a label widget.

```
self.text = f'x: {x}, y: {y}'  
self.label = QLabel(self.text, self)
```

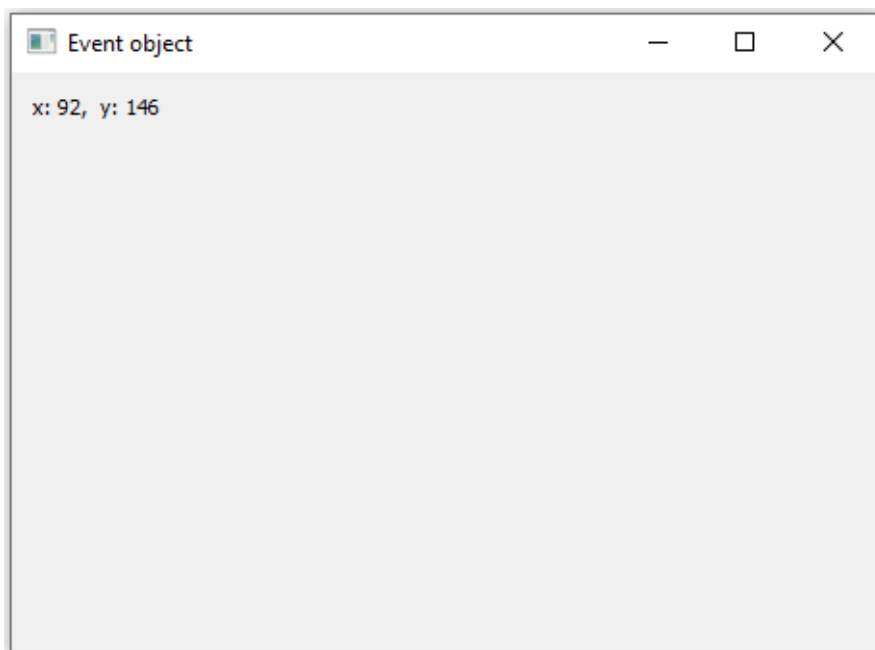
The x and y coordinates are displayed in a QLabel widget.

```
self.setMouseTracking(True)
```

Mouse tracking is disabled by default, so the widget only receives mouse move events when at least one mouse button is pressed while the mouse is being moved. If mouse tracking is enabled, the widget receives mouse move events even if no buttons are pressed.

```
def mouseMoveEvent(self, e):  
  
    x = e.x()  
    y = e.y()  
  
    text = f'x: {x}, y: {y}'  
    self.label.setText(text)
```

The `e` is the event object; it contains data about the event that was triggered; in our case, a mouse move event. With the `x()` and `y()` methods we determine the x and y coordinates of the mouse pointer. We build the string and set it to the label widget.



PyQt5 event sender

Sometimes it is convenient to know which widget is the sender of a signal. For this, PyQt5 has the `sender` method.

```
Qt0604.py
1  """
2  In this example, we determine the event sender
3  object.
4  """
5  import sys
6  from PyQt5.QtWidgets import QMainWindow, QPushButton, QApplication
7
8  class Example(QMainWindow):
9      def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         btn1 = QPushButton("Button 1", self)
15         btn1.move(30, 50)
16
17         btn2 = QPushButton("Button 2", self)
18         btn2.move(150, 50)
19
20         btn1.clicked.connect(self.buttonClicked)
21         btn2.clicked.connect(self.buttonClicked)
22
23         self.statusBar()
24
25         self.setGeometry(300, 300, 450, 350)
26         self.setWindowTitle('Event sender')
27         self.show()
28
29     def buttonClicked(self):
30         sender = self.sender()
31         self.statusBar().showMessage(sender.text() + ' was pressed')
32
33 def main():
34     app = QApplication(sys.argv)
35     ex = Example()
36     sys.exit(app.exec_())
37
38 if __name__ == '__main__':
39     main()
```

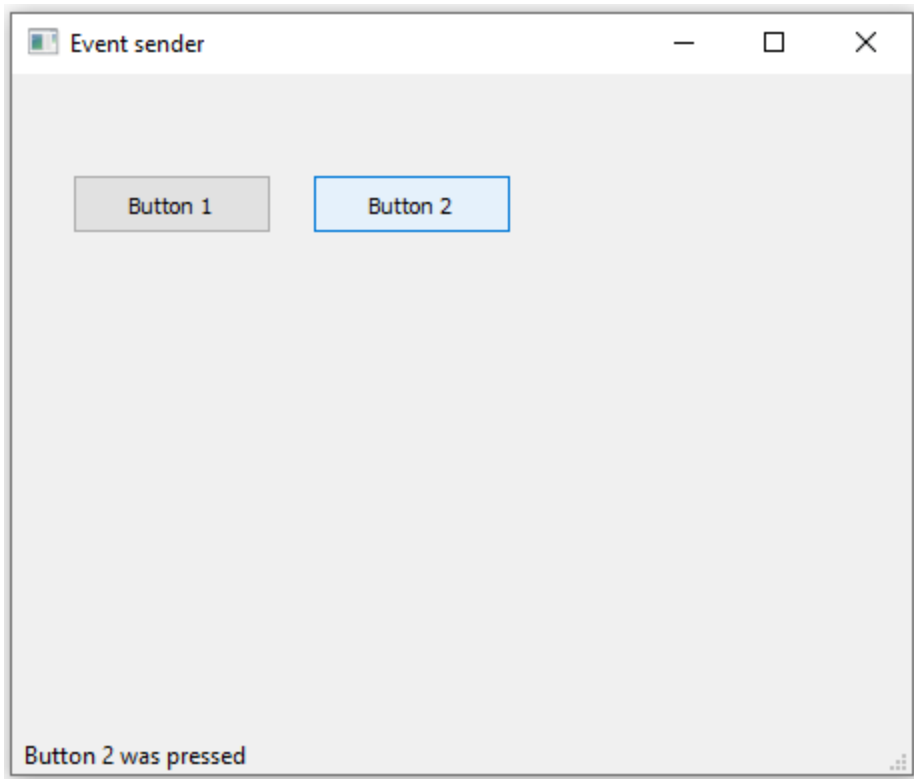
We have two buttons in our example. In the `buttonClicked` method we determine which button we have clicked by calling the `sender()` method.

```
btn1.clicked.connect(self.buttonClicked)
btn2.clicked.connect(self.buttonClicked)
```


Both buttons are connected to the same slot.

```
def buttonClicked(self):  
    sender = self.sender()  
    self.statusBar().showMessage(sender.text() + ' was pressed')
```

We determine the signal source by calling the `sender()` method. In the statusbar of the application, we show the label of the button being pressed.



PyQt5 emitting signals

Objects created from a `QObject` can emit signals. The following example shows how we to emit custom signals.

Qt0605.py

```
1 """  
2 In this example, we show how to emit a custom signal.  
3 """  
4 import sys  
5 from PyQt5.QtCore import pyqtSignal, QObject  
6 from PyQt5.QtWidgets import QMainWindow, QApplication  
7
```

```

8 class Communicate(QObject):
9     closeApp = pyqtSignal()
10
11 class Example(QMainWindow):
12     def __init__(self):
13         super().__init__()
14         self.initUI()
15
16     def initUI(self):
17         self.c = Communicate()
18         self.c.closeApp.connect(self.close)
19
20         self.setGeometry(300, 300, 450, 350)
21         self.setWindowTitle('Emit signal')
22         self.show()
23
24     def mousePressEvent(self, event):
25         self.c.closeApp.emit()
26
27 def main():
28     app = QApplication(sys.argv)
29     ex = Example()
30     sys.exit(app.exec_())
31
32 if __name__ == '__main__':
33     main()

```

We create a new signal called `closeApp`. This signal is emitted during a mouse press event. The signal is connected to the `close()` slot of the `QMainWindow`.

```

class Communicate(QObject):
    closeApp = pyqtSignal()

```

A signal is created with the `pyqtSignal()` as a class attribute of the external `Communicate` class.

```

self.c = Communicate()
self.c.closeApp.connect(self.close)

```

The custom `closeApp` signal is connected to the `close()` slot of the `QMainWindow`.

```

def mousePressEvent(self, event):
    self.c.closeApp.emit()

```

When we click on the window with a mouse pointer, the `closeApp` signal is emitted. The application terminates.

In this part of the PyQt5 tutorial, we have covered signals and slots.

7. Dialogs in PyQt5

A dialog is defined as a conversation between two or more persons. In a computer application a dialog is a window which is used to "talk" to the application. Dialogs are used for things such as getting data from users or changing application settings.

PyQt5 QInputDialog

QInputDialog provides a simple convenience dialog to get a single value from the user. The input value can be a string, a number, or an item from a list.

```
Qt0701.py
1  """
2  In this example, we receive data from a QInputDialog dialog.
3  """
4
5  from PyQt5.QtWidgets import (QWidget, QPushButton, QLineEdit,
6                               QInputDialog, QApplication)
7  import sys
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         self.btn = QPushButton('Dialog', self)
16         self.btn.move(20, 20)
17         self.btn.clicked.connect(self.showDialog)
18
19         self.le = QLineEdit(self)
20         self.le.move(130, 22)
21
22         self.setGeometry(300, 300, 450, 350)
23         self.setWindowTitle('Input dialog')
24         self.show()
25
26     def showDialog(self):
27         text, ok = QInputDialog.getText(self, 'Input Dialog',
28                                       'Enter your name:')
29         if ok:
30             self.le.setText(str(text))
31
32 def main():
33     app = QApplication(sys.argv)
34     ex = Example()
35     sys.exit(app.exec_())
36
37 if __name__ == '__main__':
38     main()
```

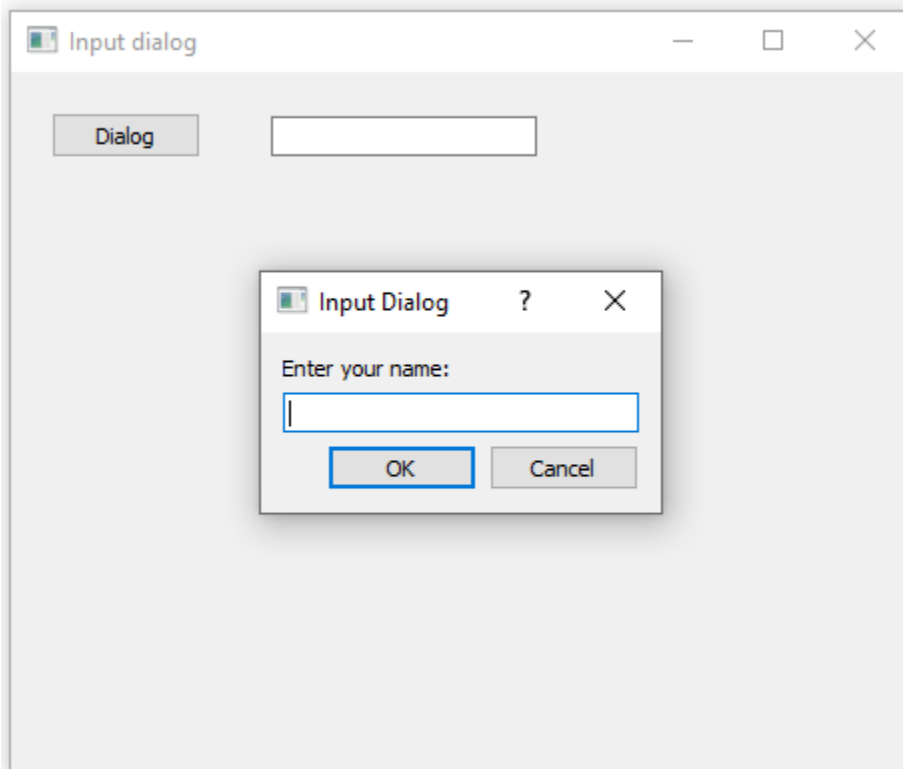
The example has a button and a line edit widget. The button shows the input dialog for getting text values. The entered text will be displayed in the line edit widget.

```
text, ok = QInputDialog.getText(self, 'Input Dialog',  
                               'Enter your name:')
```

This line displays the input dialog. The first string is a dialog title, the second one is a message within the dialog. The dialog returns the entered text and a boolean value. If we click the Ok button, the boolean value is true.

```
if ok:  
    self.le.setText(str(text))
```

The text that we have received from the dialog is set to the line edit widget with `setText()`.



PyQt5 QColorDialog

QColorDialog provides a dialog widget for selecting colour values.

```
Qt0702.py
1  """
2  In this example, we select a color value from the QColorDialog and
3  change the background color of a QFrame widget.
4  """
5  from PyQt5.QtWidgets import (QWidget, QPushButton, QFrame,
6                               QColorDialog, QApplication)
7  from PyQt5.QtGui import QColor
8  import sys
9
10 class Example(QWidget):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         col = QColor(0, 0, 0)
17
18         self.btn = QPushButton('Dialog', self)
19         self.btn.move(20, 20)
20
21         self.btn.clicked.connect(self.showDialog)
22
23         self.frm = QFrame(self)
24         self.frm.setStyleSheet("QWidget { background-color: %s }"
25                                % col.name())
26         self.frm.setGeometry(130, 22, 200, 200)
27
28         self.setGeometry(300, 300, 450, 350)
29         self.setWindowTitle('Color dialog')
30         self.show()
31
32     def showDialog(self):
33         col = QColorDialog.getColor()
34
35         if col.isValid():
36             self.frm.setStyleSheet('QWidget { background-color: %s }'
37                                    % col.name())
38
39 def main():
40     app = QApplication(sys.argv)
41     ex = Example()
42     sys.exit(app.exec_())
43
44 if __name__ == '__main__':
45     main()
```

The application example shows a push button and a QFrame. The widget background is set to black colour. Using the QColorDialog, we can change its background.

```
col = QColor(0, 0, 0)
```

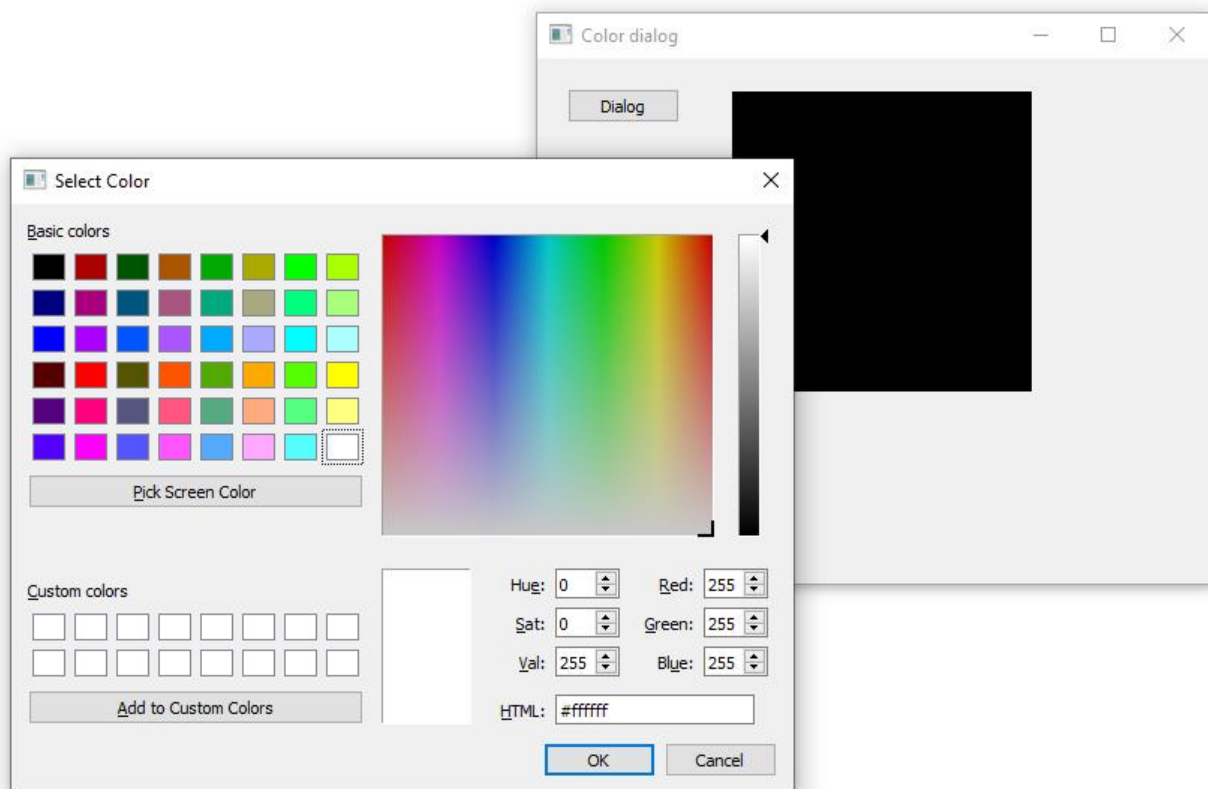
This is an initial colour of the `QFrame` background.

```
col = QColorDialog.getColor()
```

This line pops up the `QColorDialog`.

```
if col.isValid():  
    self.frm.setStyleSheet("QWidget { background-color: %s }"  
        % col.name())
```

We check if the colour is valid. If we click on the Cancel button, no valid colour is returned. If the colour is valid, we change the background colour using style sheets.



PyQt5 QFontDialog

QFontDialog is a dialog widget for selecting a font.

```
Qt0703.py
1  """
2  In this example, we select a font name and change the font
3  of a label.
4  """
5  from PyQt5.QtWidgets import (QWidget, QVBoxLayout, QPushButton,
6                               QSizePolicy, QLabel, QFontDialog, QApplication)
7  import sys
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         vbox = QVBoxLayout()
16
17         btn = QPushButton('Dialog', self)
18         btn.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
19         btn.move(20, 20)
20
21         vbox.addWidget(btn)
22
23         btn.clicked.connect(self.showDialog)
24
25         self.lbl = QLabel('Knowledge only matters', self)
26         self.lbl.move(130, 20)
27
28         vbox.addWidget(self.lbl)
29         self.setLayout(vbox)
30
31         self.setGeometry(300, 300, 450, 350)
32         self.setWindowTitle('Font dialog')
33         self.show()
34
35     def showDialog(self):
36         font, ok = QFontDialog.getFont()
37         if ok:
38             self.lbl.setFont(font)
39
40 def main():
41     app = QApplication(sys.argv)
42     ex = Example()
43     sys.exit(app.exec_())
44
45 if __name__ == '__main__':
46     main()
```

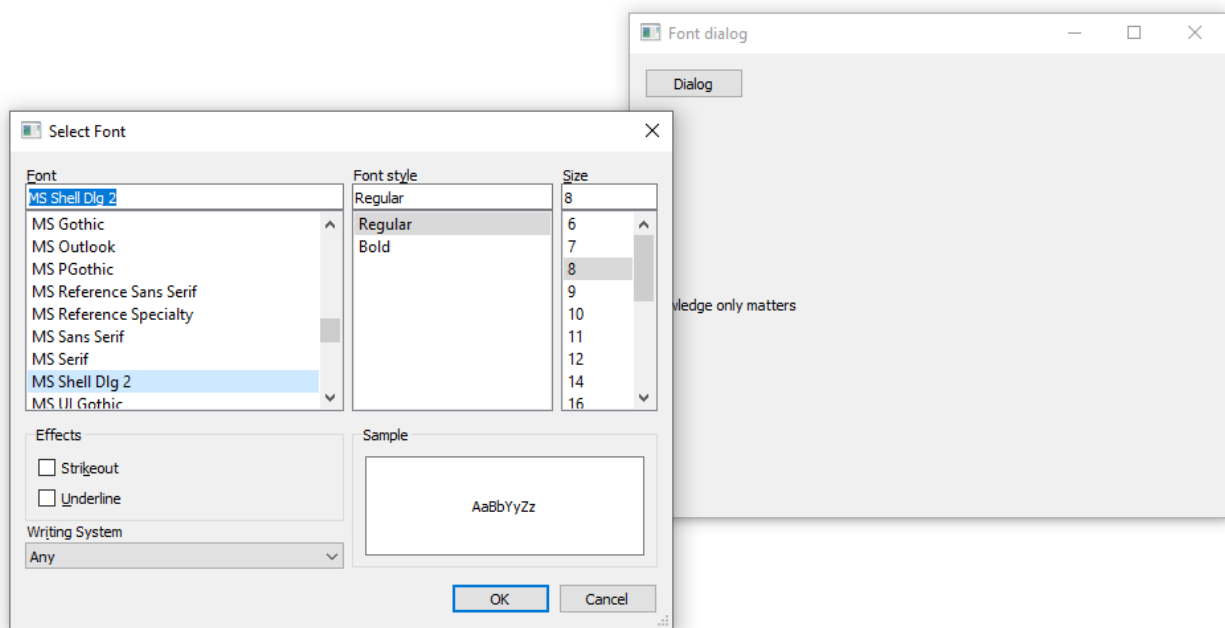
In our example, we have a button and a label. With the `QFontDialog`, we change the font of the label.

```
font, ok = QFontDialog.getFont()
```

Here we pop up the font dialog. The `getFont` method returns the font name and the `ok` parameter. It is equal to `True` if the user clicked `Ok`; otherwise it is `False`.

```
if ok:  
    self.label.setFont(font)
```

If we clicked `Ok`, the font of the label is changed with `setFont`.



PyQt5 QFileDialog

`QFileDialog` is a dialog that allows users to select files or directories. The files can be selected for both opening and saving.

Qt0704.py

```
1 """  
2 In this example, we select a file with a QFileDialog and  
3 display its contents in a QTextEdit.  
4 """  
5 from PyQt5.QtWidgets import (QMainWindow, QTextEdit,  
6                               QAction, QFileDialog, QApplication)  
7 from PyQt5.QtGui import QIcon  
8 import sys  
9 from pathlib import Path  
10
```



```

11 class Example(QMainWindow):
12     def __init__(self):
13         super().__init__()
14         self.initUI()
15
16     def initUI(self):
17         self.textEdit = QTextEdit()
18         self.setCentralWidget(self.textEdit)
19         self.statusBar()
20
21         openFile = QAction(QIcon('open.png'), 'Open', self)
22         openFile.setShortcut('Ctrl+O')
23         openFile.setStatusTip('Open new File')
24         openFile.triggered.connect(self.showDialog)
25
26         menubar = self.menuBar()
27         fileMenu = menubar.addMenu('&File')
28         fileMenu.addAction(openFile)
29
30         self.setGeometry(300, 300, 550, 450)
31         self.setWindowTitle('File dialog')
32         self.show()
33
34     def showDialog(self):
35         home_dir = str(Path.home())
36         fname = QFileDialog.getOpenFileName(self, 'Open file', home_dir)
37
38         if fname[0]:
39             f = open(fname[0], 'r')
40
41             with f:
42                 data = f.read()
43                 self.textEdit.setText(data)
44
45     def main():
46         app = QApplication(sys.argv)
47         ex = Example()
48         sys.exit(app.exec_())
49
50 if __name__ == '__main__':
51     main()
52

```

The example shows a menubar, centrally set text edit widget, and a statusbar. The menu item shows the `QFileDialog` which is used to select a file. The contents of the file are loaded into the text edit widget.

```

class Example(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initUI()

```

The example is based on the `QMainWindow` widget because we centrally set a text edit widget.

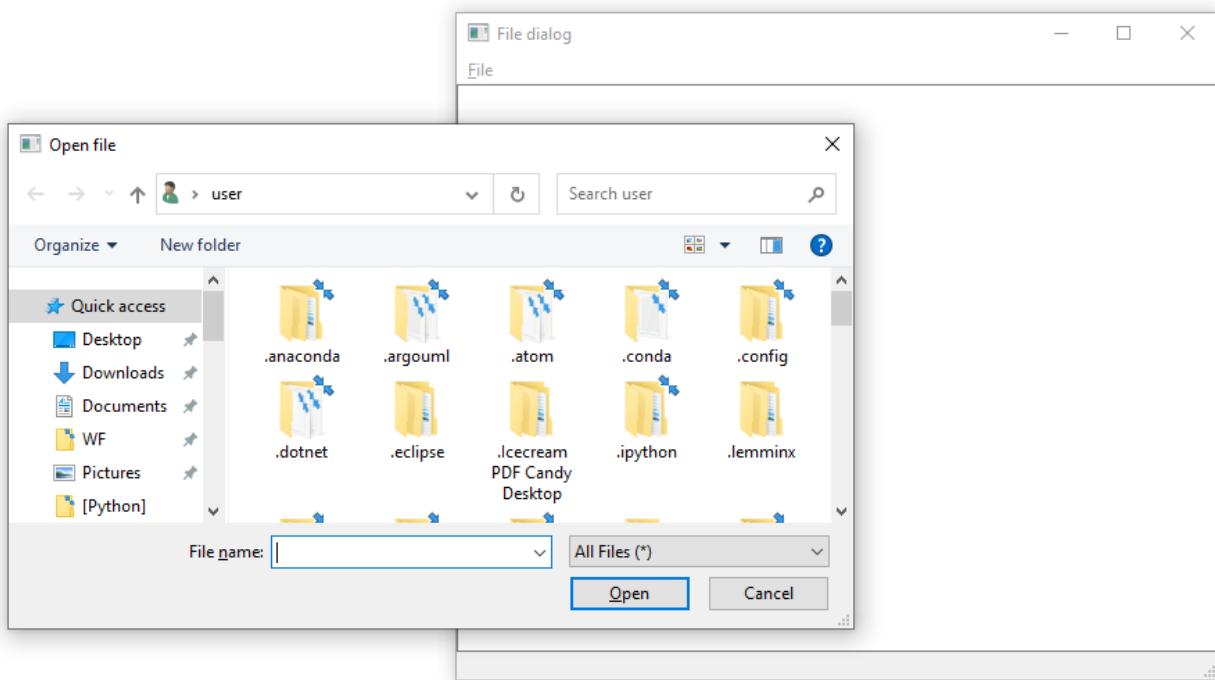
```
home_dir = str(Path.home())
fname = QFileDialog.getOpenFileName(self, 'Open file', home_dir)
```

We pop up the `QFileDialog`. The first string in the `getOpenFileName()` method is the caption. The second string specifies the dialog working directory. We use the `path` module to determine the user's home directory. By default, the file filter is set to All files (*).

```
if fname[0]:
    f = open(fname[0], 'r')

    with f:
        data = f.read()
        self.textEdit.setText(data)
```

The selected file name is read and the contents of the file are set to the text edit widget.



In this part of the PyQt5 tutorial, we worked with dialogs.

8. PyQt5 widgets

Widgets are basic building blocks of an application. PyQt5 has a wide range of various widgets, including buttons, check boxes, sliders, or list boxes. In this section of the tutorial, we will describe several useful widgets: a `QCheckBox`, a `QPushButton` in toggle mode, a `QSlider`, a `QProgressBar`, and a `QCalendarWidget`.

PyQt5 QCheckBox

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.

Qt0801.py

```
1 """
2 In this example, a QCheckBox widget is used to toggle
3 the title of a window.
4 """
5 from PyQt5.QtWidgets import QWidget, QCheckBox, QApplication
6 from PyQt5.QtCore import Qt
7 import sys
8
9 class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         cb = QCheckBox('Show title', self)
16         cb.move(20, 20)
17         cb.toggle()
18         cb.stateChanged.connect(self.changeTitle)
19
20         self.setGeometry(300, 300, 350, 250)
21         self.setWindowTitle('QCheckBox')
22         self.show()
23
24     def changeTitle(self, state):
25         if state == Qt.Checked:
26             self.setWindowTitle('QCheckBox')
27         else:
28             self.setWindowTitle(' ')
29
30 def main():
31     app = QApplication(sys.argv)
32     ex = Example()
33     sys.exit(app.exec_())
34
```

```
35 if __name__ == '__main__':
36     main()
```

In our example, we will create a checkbox that will toggle the window title.

```
cb = QCheckBox('Show title', self)
```

This is a QCheckBox constructor.

```
cb.toggle()
```

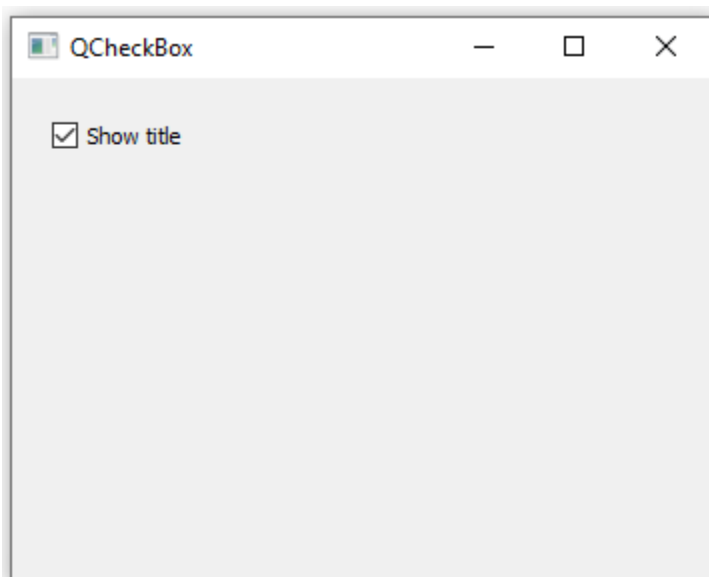
We have set the window title, so we also check the checkbox.

```
cb.stateChanged.connect(self.changeTitle)
```

We connect the user defined changeTitle() method to the stateChanged signal. The changeTitle() method will toggle the window title.

```
def changeTitle(self, state):
    if state == Qt.Checked:
        self.setWindowTitle('QCheckBox')
    else:
        self.setWindowTitle('')
```

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.



Toggle button

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.

```
Qt0802.py
1  """
2  In this example, we create three toggle buttons.
3  They will control the background color of a QFrame.
4  """
5  from PyQt5.QtWidgets import (QWidget, QPushButton,
6                               QFrame, QApplication)
7  from PyQt5.QtGui import QColor
8  import sys
9
10 class Example(QWidget):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         self.col = QColor(0, 0, 0)
17
18         redb = QPushButton('Red', self)
19         redb.setCheckable(True)
20         redb.move(10, 10)
21
22         redb.clicked[bool].connect(self.setColor)
23
24         greenb = QPushButton('Green', self)
25         greenb.setCheckable(True)
26         greenb.move(10, 60)
27
28         greenb.clicked[bool].connect(self.setColor)
29
30         blueb = QPushButton('Blue', self)
31         blueb.setCheckable(True)
32         blueb.move(10, 110)
33
34         blueb.clicked[bool].connect(self.setColor)
35
36         self.square = QFrame(self)
37         self.square.setGeometry(150, 20, 100, 100)
38         self.square.setStyleSheet("QWidget { background-color: %s }" %
39                                   self.col.name())
40
41         self.setGeometry(300, 300, 300, 250)
42         self.setWindowTitle('Toggle button')
43         self.show()
44
45     def setColor(self, pressed):
46         source = self.sender()
47
48         if pressed:
```

```

49         val = 255
50     else:
51         val = 0
52
53     if source.text() == "Red":
54         self.col.setRed(val)
55     elif source.text() == "Green":
56         self.col.setGreen(val)
57     else:
58         self.col.setBlue(val)
59
60     self.square.setStyleSheet("QFrame { background-color: %s }" %
61                               self.col.name())
62
63 def main():
64     app = QApplication(sys.argv)
65     ex = Example()
66     sys.exit(app.exec_())
67
68 if __name__ == '__main__':
69     main()

```

In our example, we create three toggle buttons and a `QWidget`. We set the background colour of the `QWidget` to black. 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.

```
self.col = QColor(0, 0, 0)
```

This is the initial, black colour value.

```

redb = QPushButton('Red', self)
redb.setCheckable(True)
redb.move(10, 10)

```

To create a toggle button, we create a `QPushButton` and make it checkable by calling the `setCheckable()` method.

```
redb.clicked[bool].connect(self.setColor)
```

We connect a `clicked` signal to our user defined method. We use the `clicked` signal that operates with a Boolean value.

```
source = self.sender()
```

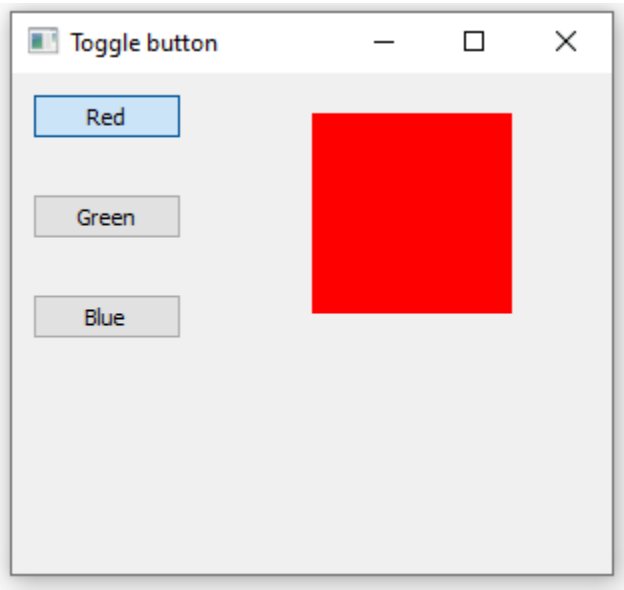
We get the button which was toggled.

```
if source.text() == "Red":  
    self.col.setRed(val)
```

In case it is a red button, we update the red part of the colour accordingly.

```
self.square.setStyleSheet("QFrame { background-color: %s }" %  
    self.col.name())
```

We use style sheets to change the background colour. The stylesheet is updated with `setStyleSheet()` method.



PyQt5 QSlider

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.

In our example we show one slider and one label. The label displays an image. The slider controls the label.

Qt0803.py

```
1  """
2  This example shows a QSlider widget.
3  """
4  from PyQt5.QtWidgets import (QWidget, QSlider,
5                               QLabel, QApplication)
6  from PyQt5.QtCore import Qt
7  from PyQt5.QtGui import QPixmap
8  import sys
9
10 class Example(QWidget):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         sld = QSlider(Qt.Horizontal, self)
17         sld.setFocusPolicy(Qt.NoFocus)
18         sld.setGeometry(30, 40, 200, 60)
19         sld.valueChanged[int].connect(self.changeValue)
20
21         self.label = QLabel(self)
22         self.label.setPixmap(QPixmap('Mute.png'))
23         self.label.setGeometry(250, 40, 80, 60)
24
25         self.setGeometry(300, 300, 350, 250)
26         self.setWindowTitle('QSlider')
27         self.show()
28
29     def changeValue(self, value):
30         if value == 0:
31
32             self.label.setPixmap(QPixmap('Mute.png'))
33         elif 0 < value <= 30:
34
35             self.label.setPixmap(QPixmap('Min.png'))
36         elif 30 < value < 80:
37
38             self.label.setPixmap(QPixmap('Med.png'))
39         else:
40
41             self.label.setPixmap(QPixmap('Max.png'))
42
43     def main():
44         app = QApplication(sys.argv)
45         ex = Example()
46         sys.exit(app.exec_())
47
48 if __name__ == '__main__':
49     main()
```

In our example we simulate a volume control. By dragging the handle of a slider, we change an image on the label.


```
sld = QSlider(Qt.Horizontal, self)
```

Here we create a horizontal `QSlider`.

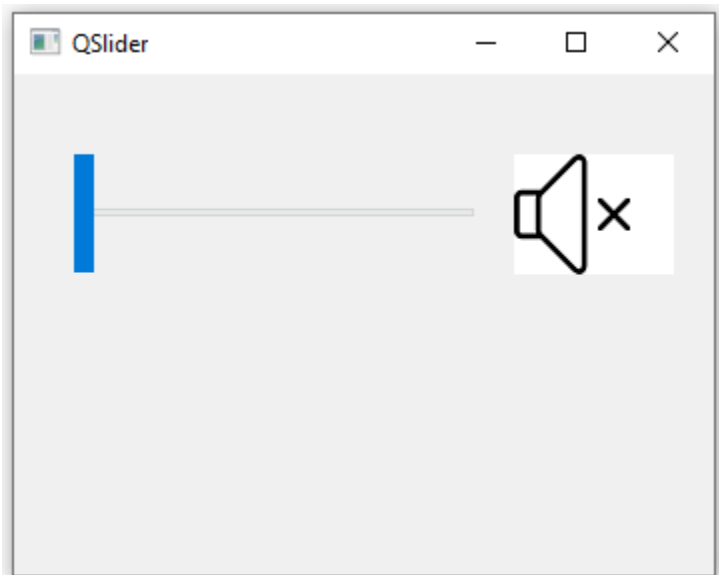
```
self.label = QLabel(self)
self.label.setPixmap(QPixmap('mute.png'))
```

We create a `QLabel` widget and set an initial mute image to it.

```
sld.valueChanged[int].connect(self.changeValue)
```

We connect the `valueChanged` signal to the user defined `changeValue()` method.

Based on the value of the slider, we set an image to the label. In the above code, we set the `mute.png` image to the label if the slider is equal to zero.



PyQt5 QProgressBar

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.

Qt0804.py

```
1  """
2  This example shows a QProgressBar widget.
3  """
4  from PyQt5.QtWidgets import (QWidget, QProgressBar,
5                               QPushButton, QApplication)
6  from PyQt5.QtCore import QBasicTimer
7  import sys
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         self.pbar = QProgressBar(self)
16         self.pbar.setGeometry(30, 40, 200, 25)
17
18         self.btn = QPushButton('Start', self)
19         self.btn.move(40, 80)
20         self.btn.clicked.connect(self.doAction)
21
22         self.timer = QBasicTimer()
23         self.step = 0
24
25         self.setGeometry(300, 300, 280, 170)
26         self.setWindowTitle('QProgressBar')
27         self.show()
28
29     def timerEvent(self, e):
30         if self.step >= 100:
31             self.timer.stop()
32             self.btn.setText('Finished')
33             return
34
35         self.step = self.step + 1
36         self.pbar.setValue(self.step)
37
38     def doAction(self):
39         if self.timer.isActive():
40             self.timer.stop()
41             self.btn.setText('Start')
42         else:
43             self.timer.start(100, self)
44             self.btn.setText('Stop')
45
46     def main():
47         app = QApplication(sys.argv)
48         ex = Example()
49         sys.exit(app.exec_())
50
51 if __name__ == '__main__':
52     main()
```

In our example we have a horizontal progress bar and a push button. The push button starts and stops the progress bar.

```
self.pbar = QProgressBar(self)
```

This is a QProgressBar constructor.

```
self.timer = QBasicTimer()
```

To activate the progress bar, we use a timer object.

```
self.timer.start(100, self)
```

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.

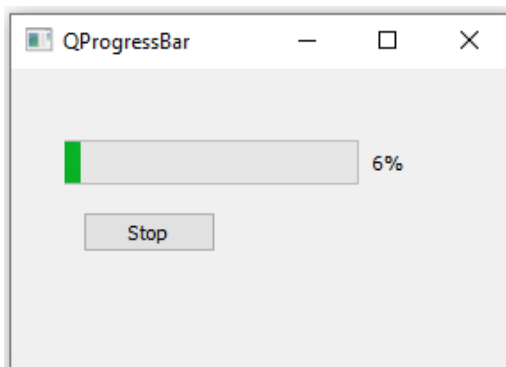
```
def timerEvent(self, e):
    if self.step >= 100:
        self.timer.stop()
        self.btn.setText('Finished')
        return

    self.step = self.step + 1
    self.pbar.setValue(self.step)
```

Each QObject and its descendants have a timerEvent() event handler. In order to react to timer events, we reimplement the event handler.

```
def doAction(self):
    if self.timer.isActive():
        self.timer.stop()
        self.btn.setText('Start')
    else:
        self.timer.start(100, self)
        self.btn.setText('Stop')
```

Inside the doAction() method, we start and stop the timer.



PyQt5 QCalendarWidget

A `QCalendarWidget` provides a monthly based calendar widget. It allows a user to select a date in a simple and intuitive way.

Qt0805.py

```
1 """
2 This example shows a QCalendarWidget widget.
3 """
4 from PyQt5.QtWidgets import (QWidget, QCalendarWidget,
5                               QLabel, QApplication, QVBoxLayout)
6 from PyQt5.QtCore import QDate
7 import sys
8
9 class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         vbox = QVBoxLayout(self)
16
17         cal = QCalendarWidget(self)
18         cal.setGridVisible(True)
19         cal.clicked[QDate].connect(self.showDate)
20
21         vbox.addWidget(cal)
22
23         self.lbl = QLabel(self)
24         date = cal.selectedDate()
25         self.lbl.setText(date.toString())
26
27         vbox.addWidget(self.lbl)
28
29         self.setLayout(vbox)
30
31         self.setGeometry(300, 300, 350, 300)
32         self.setWindowTitle('Calendar')
33         self.show()
34
35     def showDate(self, date):
36         self.lbl.setText(date.toString())
37
38 def main():
39     app = QApplication(sys.argv)
40     ex = Example()
41     sys.exit(app.exec_())
42
43 if __name__ == '__main__':
44     main()
```

The example has a calendar widget and a label widget. The currently selected date is displayed in the label widget.

```
cal = QCalendarWidget(self)
```

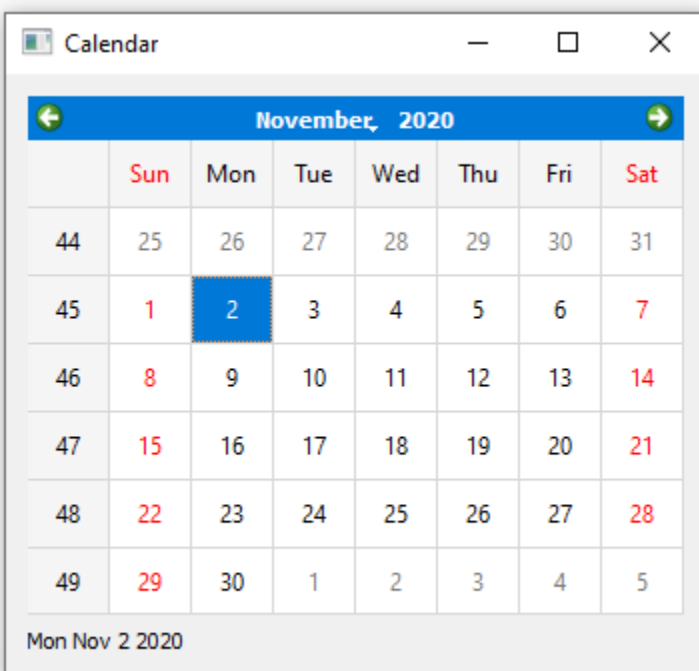
The `QCalendarWidget` is created.

```
cal.clicked[QDate].connect(self.showDate)
```

If we select a date from the widget, a `clicked[QDate]` signal is emitted. We connect this signal to the user defined `showDate()` method.

```
def showDate(self, date):  
    self.lbl.setText(date.toString())
```

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.



In this part of the PyQt5 tutorial, we covered the following widgets: `QCheckBox`, `QPushButton` in toggle mode, `QSlider`, `QProgressBar`, and `QCalendarWidget`.

9. PyQt5 widgets II

In this chapter we continue introducing PyQt5 widgets. We will COVER QPixmap, QLineEdit, QSplitter, and QComboBox.

PyQt5 QPixmap

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.

```
Qt0901.py
1  """
2  In this example, we display an image on the window.
3  """
4  from PyQt5.QtWidgets import (QWidget, QHBoxLayout,
5                               QLabel, QApplication)
6  from PyQt5.QtGui import QPixmap
7  import sys
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         hbox = QHBoxLayout(self)
16         pixmap = QPixmap('Durian.jpg')
17
18         lbl = QLabel(self)
19         lbl.setPixmap(pixmap)
20
21         hbox.addWidget(lbl)
22         self.setLayout(hbox)
23
24         self.move(300, 200)
25         self.setWindowTitle('King of Fruit')
26         self.show()
27
28     def main():
29         app = QApplication(sys.argv)
30         ex = Example()
31         sys.exit(app.exec_())
32
33 if __name__ == '__main__':
34     main()
```

In our example, we display an image on the window.

```
 pixmap = QPixmap('sid.jpg')
```

We create a `QPixmap` object. It takes the name of the file as a parameter.

```
 lbl = QLabel(self)  
 lbl.setPixmap(pixmap)
```

We put the pixmap into the `QLabel` widget.

Output:



PyQt5 QLineEdit

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.

```
Qt0902.py
1  """
2  This example shows text which is entered in a QLineEdit
3  in a QLabel widget.
4  """
5  import sys
6  from PyQt5.QtWidgets import (QWidget, QLabel,
7                               QLineEdit, QApplication)
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         self.lbl = QLabel(self)
16         qle = QLineEdit(self)
17
18         qle.move(60, 100)
19         self.lbl.move(60, 40)
20
21         qle.textChanged[str].connect(self.onChanged)
22
23         self.setGeometry(300, 300, 350, 250)
24         self.setWindowTitle('QLineEdit')
25         self.show()
26
27     def onChanged(self, text):
28         self.lbl.setText(text)
29         self.lbl.adjustSize()
30
31     def main():
32         app = QApplication(sys.argv)
33         ex = Example()
34         sys.exit(app.exec_())
35
36 if __name__ == '__main__':
37     main()
```

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.

```
qle = QLineEdit(self)
```

The QLineEdit widget is created.

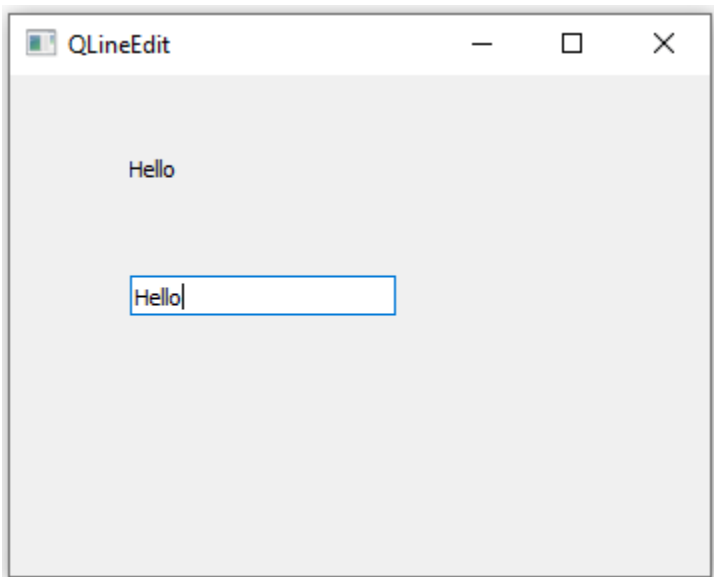

```
gle.textChanged[str].connect(self.onChanged)
```

If the text in the line edit widget changes, we call the `onChanged` method.

```
def onChanged(self, text):  
    self.lbl.setText(text)  
    self.lbl.adjustSize()
```

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.

Output:



PyQt5 QSplitter

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.

Qt0903.py

```
1  """
2  This example shows how to use QSplitter widget.
3  """
4  import sys
5
6  from PyQt5.QtCore import Qt
7  from PyQt5.QtWidgets import (QWidget, QHBoxLayout, QFrame,
8                               QSplitter, QApplication)
9
10 class Example(QWidget):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         hbox = QHBoxLayout(self)
17
18         topleft = QFrame(self)
19         topleft.setFrameShape(QFrame.StyledPanel)
20
21         topright = QFrame(self)
22         topright.setFrameShape(QFrame.StyledPanel)
23
24         bottom = QFrame(self)
25         bottom.setFrameShape(QFrame.StyledPanel)
26
27         splitter1 = QSplitter(Qt.Horizontal)
28         splitter1.addWidget(topleft)
29         splitter1.addWidget(topright)
30
31         splitter2 = QSplitter(Qt.Vertical)
32         splitter2.addWidget(splitter1)
33         splitter2.addWidget(bottom)
34
35         hbox.addWidget(splitter2)
36         self.setLayout(hbox)
37
38         self.setGeometry(300, 300, 450, 400)
39         self.setWindowTitle('QSplitter')
40         self.show()
41
42 def main():
43     app = QApplication(sys.argv)
44     ex = Example()
45     sys.exit(app.exec_())
46
```

```
47 if __name__ == '__main__':  
48     main()
```

In our example, we have three frame widgets and two splitters. Note that under some themes, the splitters may not be visible very well.

```
topleft = QFrame(self)  
topleft.setStyleSheet(QFrame.StyledPanel)
```

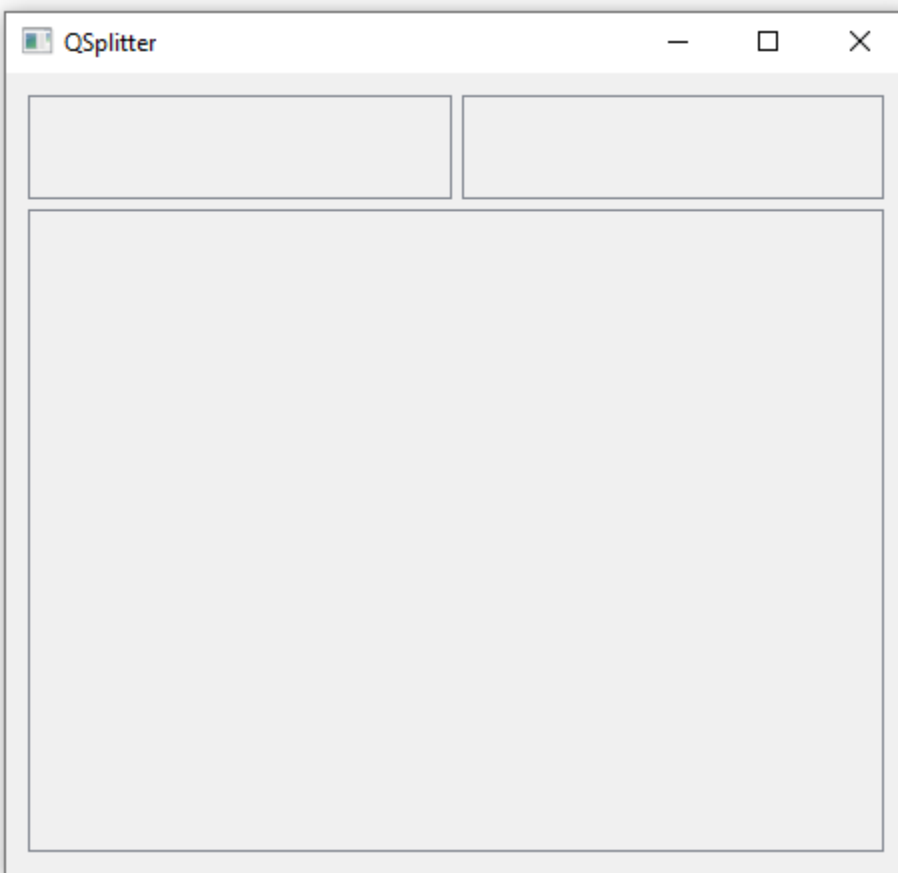
We use a styled frame in order to see the boundaries between the `QFrame` widgets.

```
splitter1 = QSplitter(Qt.Horizontal)  
splitter1.addWidget(topleft)  
splitter1.addWidget(topright)
```

We create a `QSplitter` widget and add two frames into it.

We can also add a splitter to another splitter widget.

Output:



PyQt5 QComboBox

QComboBox is a widget that allows a user to choose from a list of options.

```
Qt0904.py
1  """
2  This example shows how to use a QComboBox widget.
3  """
4  import sys
5  from PyQt5.QtWidgets import (QWidget, QLabel,
6                               QComboBox, QApplication)
7
8  class Example(QWidget):
9      def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         self.lbl = QLabel('Ubuntu', self)
15
16         combo = QComboBox(self)
17         combo.addItem('Ubuntu')
18         combo.addItem('Mandriva')
19         combo.addItem('Fedora')
20         combo.addItem('Arch')
21         combo.addItem('Gentoo')
22
23         combo.move(50, 50)
24         self.lbl.move(50, 150)
25
26         combo.activated[str].connect(self.onActivated)
27
28         self.setGeometry(300, 300, 450, 400)
29         self.setWindowTitle('QComboBox')
30         self.show()
31
32     def onActivated(self, text):
33         self.lbl.setText(text)
34         self.lbl.adjustSize()
35
36 def main():
37     app = QApplication(sys.argv)
38     ex = Example()
39     sys.exit(app.exec_())
40
41 if __name__ == '__main__':
42     main()
```

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.

```
combo = QComboBox(self)
combo.addItem('Ubuntu')
combo.addItem('Mandriva')
combo.addItem('Fedora')
combo.addItem('Arch')
combo.addItem('Gentoo')
```

We create a `QComboBox` widget with five options.

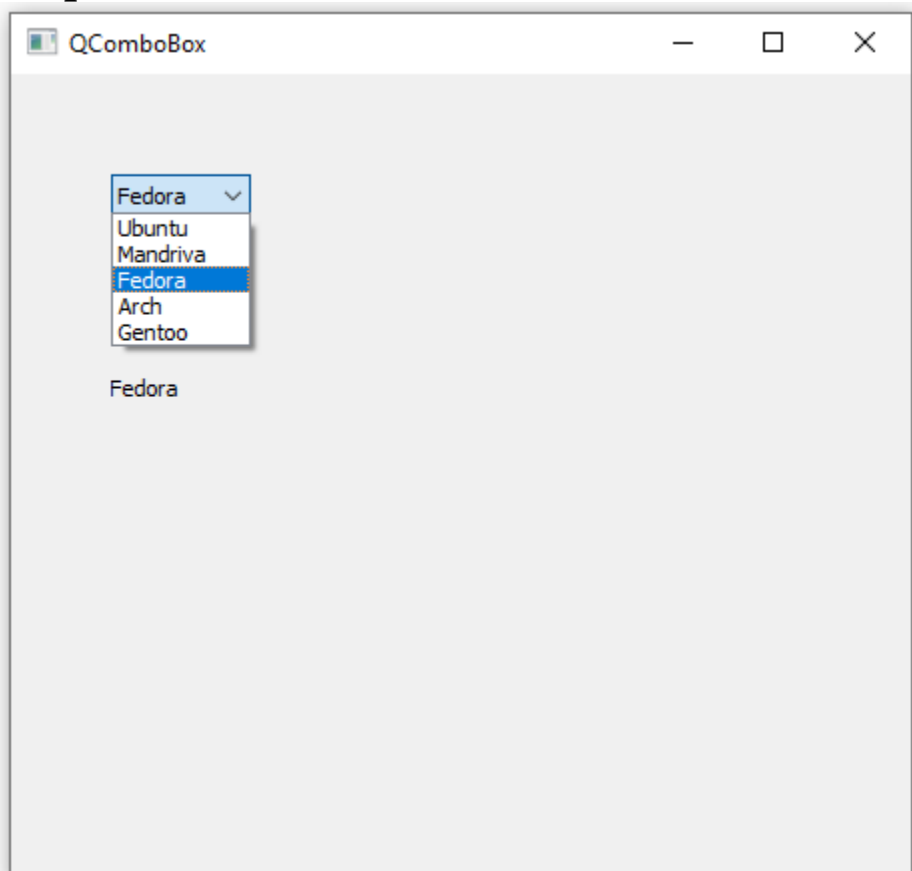
```
combo.activated[str].connect(self.onActivated)
```

Upon an item selection, we call the `onActivated()` method.

```
def onActivated(self, text):
    self.lbl.setText(text)
    self.lbl.adjustSize()
```

Inside the method, we set the text of the chosen item to the label widget. We adjust the size of the label.

Output:



10. Drag and drop in PyQt5

In this part of the PyQt5 tutorial, we will talk about drag & drop operations.

In computer graphical user interfaces, drag-and-drop is the action of (or support for the action of) clicking on a virtual object and dragging it to a different location or onto another virtual object. In general, it can be used to invoke many kinds of actions, or create various types of associations between two abstract objects.

Drag and drop is part of the graphical user interface. Drag and drop operations enable users to do complex things intuitively.

Usually, we can drag and drop two things: data or some graphical objects. If we drag an image from one application to another, we drag and drop binary data. If we drag a tab in Firefox and move it to another place, we drag and drop a graphical component.

QDrag

QDrag provides support for MIME-based drag and drop data transfer. It handles most of the details of a drag and drop operation. The transferred data is contained in a QMimeData object.

Simple drag and drop example in PyQt5

In the first example, we have a QLineEdit and a QPushButton. We drag plain text from the line edit widget and drop it onto the button widget. The button's label will change.

```
Qt1001.py
1 """
2 This is a simple drag and drop example.
3 """
4 import sys
5 from PyQt5.QtWidgets import (QPushButton, QWidget,
6                               QLineEdit, QApplication)
7
8 class Button(QPushButton):
9     def __init__(self, title, parent):
10         super().__init__(title, parent)
11         self.setAcceptDrops(True)
12
13     def dragEnterEvent(self, e):
14         if e.mimeData().hasFormat('text/plain'):
15             e.accept()
```

```

16         else:
17             e.ignore()
18
19     def dropEvent(self, e):
20         self.setText(e.mimeData().text())
21
22 class Example(QWidget):
23     def __init__(self):
24         super().__init__()
25         self.initUI()
26
27     def initUI(self):
28         edit = QLineEdit('', self)
29         edit.setDragEnabled(True)
30         edit.move(30, 65)
31
32         button = Button("Button", self)
33         button.move(190, 65)
34
35         self.setWindowTitle('Simple drag and drop')
36         self.setGeometry(300, 300, 300, 150)
37
38     def main():
39         app = QApplication(sys.argv)
40         ex = Example()
41         ex.show()
42         app.exec_()
43
44 if __name__ == '__main__':
45     main()

```

The example presents a simple drag & drop operation.

```

class Button(QPushButton):
    def __init__(self, title, parent):
        super().__init__(title, parent)
        ...

```

In order to drop text on the `QPushButton` widget, we must reimplement some methods. Therefore, we create our own `Button` class which will inherit from the `QPushButton` class.

```
self.setAcceptDrops(True)
```

We enable drop events for the widget with `setAcceptDrops()`.

```

def dragEnterEvent(self, e):
    if e.mimeData().hasFormat('text/plain'):
        e.accept()
    else:
        e.ignore()

```

First, we reimplement the `dragEnterEvent()` method. We inform about the data type that we accept. In our case it is plain text.

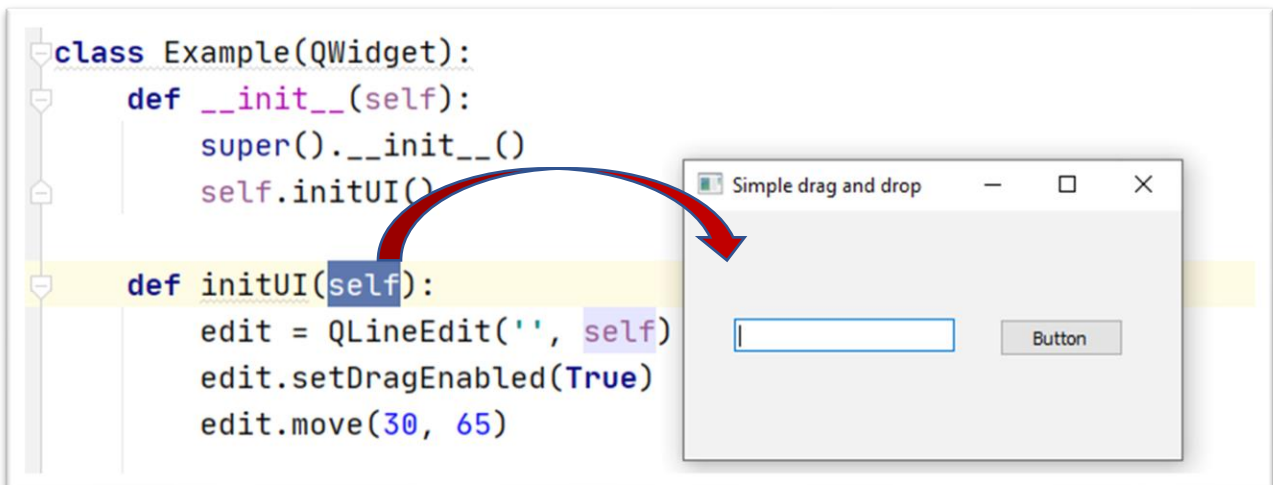
```
def dragEnterEvent(self, e):  
    self.setText(e.mimeData().text())
```

By reimplementing the `dropEvent()` method we define what happens at the drop event. Here we change the text of the button widget.

```
edit = QLineEdit('', self)  
edit.setDragEnabled(True)
```

The `QLineEdit` widget has a built-in support for drag operations. All we need to do is to call the `setDragEnabled()` method to activate it.

Output:



The screenshot displays a Qt IDE interface. On the left, a code editor shows the following Python code:

```
class Example(QWidget):  
    def __init__(self):  
        super().__init__()  
        self.initUI()  
  
    def initUI(self):  
        edit = QLineEdit('', self)  
        edit.setDragEnabled(True)  
        edit.move(30, 65)
```

The `initUI(self):` method definition is highlighted in yellow. A red arrow points from this code to a window titled "Simple drag and drop" on the right. The window contains a text input field and a button labeled "Button".

Drag and drop a button widget

The following example demonstrates how to drag and drop a button widget.

Qt1002.py

```
1 """
2 In this program, we can press on a button with a left mouse
3 click or drag and drop the button with the right mouse click.
4 """
5 import sys
6 from PyQt5.QtCore import Qt, QMimeData
7 from PyQt5.QtGui import QDrag
8 from PyQt5.QtWidgets import QPushButton, QWidget, QApplication
9
10 class Button(QPushButton):
11     def __init__(self, title, parent):
12         super().__init__(title, parent)
13
14     def mouseMoveEvent(self, e):
15         if e.buttons() != Qt.RightButton:
16             return
17
18         mimeData = QMimeData()
19
20         drag = QDrag(self)
21         drag.setMimeData(mimeData)
22         drag.setHotSpot(e.pos() - self.rect().topLeft())
23
24         dropAction = drag.exec_(Qt.MoveAction)
25
26     def mousePressEvent(self, e):
27         super().mousePressEvent(e)
28
29         if e.button() == Qt.LeftButton:
30             print('press')
31
32
33 class Example(QWidget):
34     def __init__(self):
35         super().__init__()
36         self.initUI()
37
38     def initUI(self):
39         self.setAcceptDrops(True)
40
41         self.button = Button('Button', self)
42         self.button.move(100, 65)
43
44         self.setWindowTitle('Click or Move')
45         self.setGeometry(300, 300, 550, 450)
46
47     def dragEnterEvent(self, e):
48         e.accept()
49
```

```

50     def dropEvent(self, e):
51         position = e.pos()
52         self.button.move(position)
53
54         e.setDropAction(Qt.MoveAction)
55         e.accept()
56
57     def main():
58         app = QApplication(sys.argv)
59         ex = Example()
60         ex.show()
61         app.exec_()
62
63     if __name__ == '__main__':
64         main()

```

In our code example, we have a QPushButton on the window. If we click on the button with a left mouse button, the 'press' message is printed to the console. By right clicking and moving the button, we perform a drag and drop operation on the button widget.

```

class Button(QPushButton):
    def __init__(self, title, parent):
        super().__init__(title, parent)

```

We create a Button class which derives from the QPushButton. We also reimplement two methods of the QPushButton: the mousePressEvent() and the mouseMoveEvent(). The mouseMoveEvent() method is the place where the drag and drop operation begins.

```

if e.buttons() != Qt.RightButton:
    return

```

Here we decide that we can perform drag and drop only with a right mouse button. The left mouse button is reserved for clicking on the button.

```

mimeType = QMimeData()

drag = QDrag(self)
drag.setMimeData(mimeType)
drag.setHotSpot(e.pos() - self.rect().topLeft())

```

The QDrag object is created. The class provides support for MIME-based drag and drop data transfer.

```

dropAction = drag.exec_(Qt.MoveAction)

```

The exec_() method of the drag object starts the drag and drop operation.

```

def mousePressEvent(self, e):
    super().mousePressEvent(e)

```

```
if e.button() == Qt.LeftButton:  
    print('press')
```

We print 'press' to the console if we left click on the button with the mouse. Notice that we call `mousePressEvent()` method on the parent as well. Otherwise, we would not see the button being pushed.

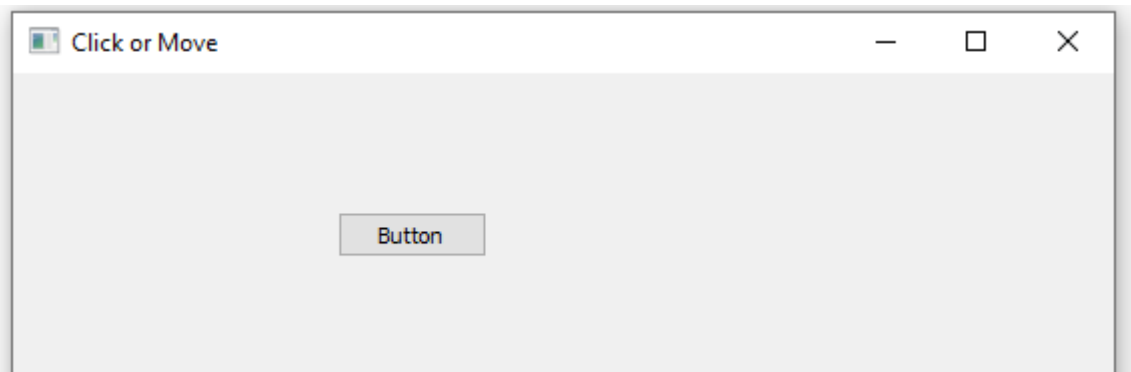
```
position = e.pos()  
self.button.move(position)
```

In the `dropEvent()` method we specify what happens after we release the mouse button and finish the drop operation. In our case, we find out the current mouse pointer position and move the button accordingly.

```
e.setDropAction(Qt.MoveAction)  
e.accept()
```

We specify the type of the drop action with `setDropAction()`. In our case it is a move action.

Output:



This part of the PyQt5 tutorial was dedicated to drag and drop operations.

11. Painting in PyQt5

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

`QPainter` performs low-level painting on widgets and other paint devices. It can draw everything from simple lines to complex shapes.

The `paintEvent` method

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.

PyQt5 draw text

We begin with drawing some Unicode text on the client area of a window.

```
Qt1101.py
1 """
2 In this example, we draw text in Russian Cylliric.
3 """
4 import sys
5 from PyQt5.QtWidgets import QWidget, QApplication
6 from PyQt5.QtGui import QPainter, QColor, QFont
7 from PyQt5.QtCore import Qt
8
9 class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         self.text = "My Chinese name is 凌志光"
16
17         self.setGeometry(300, 300, 350, 50)
18         self.setWindowTitle('Drawing text')
19         self.show()
20
21     def paintEvent(self, event):
22         qp = QPainter()
23         qp.begin(self)
```

```

24     self.drawText(event, qp)
25     qp.end()
26
27     def drawText(self, event, qp):
28         qp.setPen(QColor(168, 34, 3))
29         qp.setFont(QFont('Decorative', 10))
30         qp.drawText(event.rect(), Qt.AlignCenter, self.text)
31
32     def main():
33         app = QApplication(sys.argv)
34         ex = Example()
35         sys.exit(app.exec_())
36
37     if __name__ == '__main__':
38         main()

```

In our example, we draw some text in Cyrillic. The text is vertically and horizontally aligned.

```

def paintEvent(self, event):
...

```

Drawing is done within the paint event.

```

qp = QPainter()
qp.begin(self)
self.drawText(event, qp)
qp.end()

```

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.

```

qp.setPen(QColor(168, 34, 3))
qp.setFont(QFont('Decorative', 10))

```

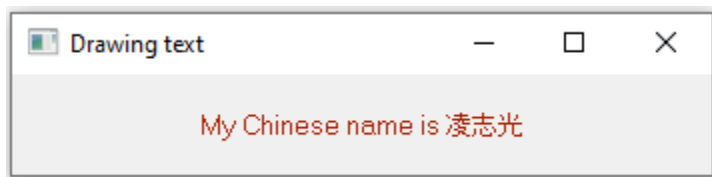
Here we define a pen and a font which are used to draw the text.

```

qp.drawText(event.rect(), Qt.AlignCenter, self.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.



PyQt5 draw points

A point is the most simple graphics object that can be drawn. It is a small spot on the window.

Qt1102.py

```
1  """
2  In the example, we draw randomly 1000 red points
3  on the window.
4  """
5  from PyQt5.QtWidgets import QWidget, QApplication
6  from PyQt5.QtGui import QPainter
7  from PyQt5.QtCore import Qt
8  import sys, random
9
10 class Example(QWidget):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         self.setGeometry(300, 300, 300, 190)
17         self.setWindowTitle('Points')
18         self.show()
19
20     def paintEvent(self, e):
21         qp = QPainter()
22         qp.begin(self)
23         self.drawPoints(qp)
24         qp.end()
25
26     def drawPoints(self, qp):
27         qp.setPen(Qt.red)
28         size = self.size()
29
30         if size.height() <= 1 or size.height() <= 1:
31             return
32
33         for i in range(1000):
34             x = random.randint(1, size.width() - 1)
35             y = random.randint(1, size.height() - 1)
36             qp.drawPoint(x, y)
37
38 def main():
39     app = QApplication(sys.argv)
40     ex = Example()
41     sys.exit(app.exec_())
42
43 if __name__ == '__main__':
44     main()
```

In our example, we draw randomly 1000 red points on the client area of the window.

```
qp.setPen(Qt.red)
```

We set the pen to red colour. We use a predefined `Qt.red` colour constant.

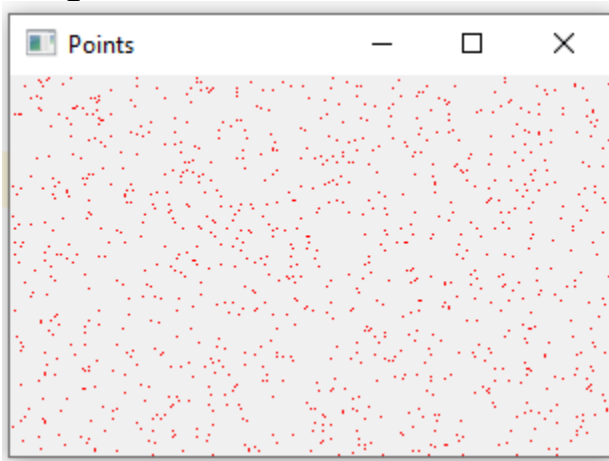
```
size = self.size()
```

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.

```
qp.drawPoint(x, y)
```

We draw the point with the `drawPoint()` method.

Output:



PyQt5 colours

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.

Qt1103.py

```
1  """
2  This example draws three rectangles in three
3  different colours.
4  """
5  from PyQt5.QtWidgets import QWidget, QApplication
6  from PyQt5.QtGui import QPainter, QColor, QBrush
7  import sys
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         self.setGeometry(300, 300, 350, 100)
16         self.setWindowTitle('Colours')
17         self.show()
18
19     def paintEvent(self, e):
20         qp = QPainter()
21         qp.begin(self)
22         self.drawRectangles(qp)
23         qp.end()
24
25     def drawRectangles(self, qp):
26         col = QColor(0, 0, 0)
27         col.setNamedColor('#d4d4d4')
28         qp.setPen(col)
29
30         qp.setBrush(QColor(200, 0, 0))
31         qp.drawRect(10, 15, 90, 60)
32
33         qp.setBrush(QColor(255, 80, 0, 160))
34         qp.drawRect(130, 15, 90, 60)
35
36         qp.setBrush(QColor(25, 0, 90, 200))
37         qp.drawRect(250, 15, 90, 60)
38
39     def main():
40         app = QApplication(sys.argv)
41         ex = Example()
42         sys.exit(app.exec_())
43
```



```
44 if __name__ == '__main__':  
45     main()
```

In our example, we draw three coloured rectangles.

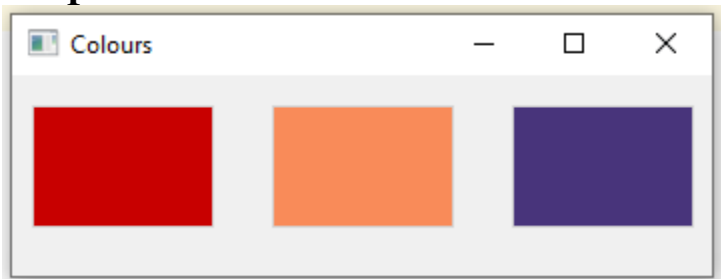
```
color = QColor(0, 0, 0)  
color.setNamedColor('#d4d4d4')
```

Here we define a colour using a hexadecimal notation.

```
qp.setBrush(QColor(200, 0, 0))  
qp.drawRect(10, 15, 90, 60)
```

Here we 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.

Output:



PyQt5 QPen

The `QPen` is an elementary graphics object. It is used to draw lines, curves and outlines of rectangles, ellipses, polygons, or other shapes.

```
Qt1104.py
1  """
2  In this example we draw 6 lines using different pen styles.
3  """
4  from PyQt5.QtWidgets import QWidget, QApplication
5  from PyQt5.QtGui import QPainter, QPen
6  from PyQt5.QtCore import Qt
7  import sys
8
9  class Example(QWidget):
10     def __init__(self):
11         super().__init__()
12         self.initUI()
13
14     def initUI(self):
15         self.setGeometry(300, 300, 280, 270)
16         self.setWindowTitle('Pen styles')
17         self.show()
18
19     def paintEvent(self, e):
20         qp = QPainter()
21         qp.begin(self)
22         self.drawLines(qp)
23         qp.end()
24
25     def drawLines(self, qp):
26         pen = QPen(Qt.black, 2, Qt.SolidLine)
27
28         qp.setPen(pen)
29         qp.drawLine(20, 40, 250, 40)
30
31         pen.setStyle(Qt.DashLine)
32         qp.setPen(pen)
33         qp.drawLine(20, 80, 250, 80)
34
35         pen.setStyle(Qt.DashDotLine)
36         qp.setPen(pen)
37         qp.drawLine(20, 120, 250, 120)
38
39         pen.setStyle(Qt.DotLine)
40         qp.setPen(pen)
41         qp.drawLine(20, 160, 250, 160)
42
43         pen.setStyle(Qt.DashDotDotLine)
44         qp.setPen(pen)
45         qp.drawLine(20, 200, 250, 200)
46
47         pen.setStyle(Qt.CustomDashLine)
48         pen.setDashPattern([1, 4, 5, 4])
```

```

49     qp.setPen(pen)
50     qp.drawLine(20, 240, 250, 240)
51
52     def main():
53         app = QApplication(sys.argv)
54         ex = Example()
55         sys.exit(app.exec_())
56
57     if __name__ == '__main__':
58         main()

```

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.

```
pen = QPen(Qt.black, 2, Qt.SolidLine)
```

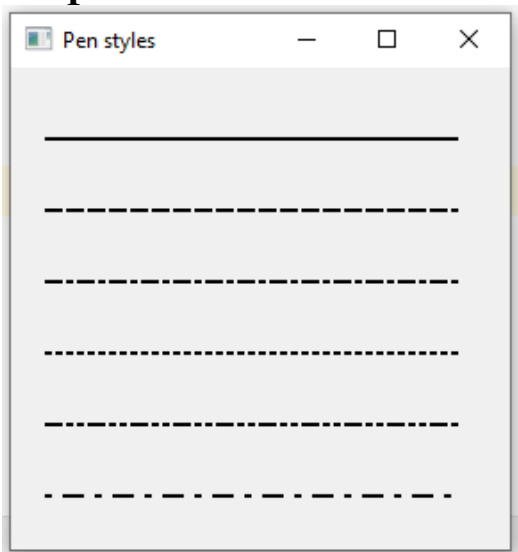
We create a `QPen` object. The colour is black. The width is set to 2 pixels so that we can see the differences between the pen styles. `Qt.SolidLine` is one of the predefined pen styles.

```
pen.setStyle(Qt.CustomDashLine)
pen.setDashPattern([1, 4, 5, 4])
qp.setPen(pen)

```

Here we 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 1 px dash, 4 px space, 5 px dash, 4 px space etc.

Output:



PyQt5 QBrush

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.

Qt1105.py

```
1  """
2  This example draws nine rectangles in different
3  brush styles.
4  """
5  from PyQt5.QtWidgets import QWidget, QApplication
6  from PyQt5.QtGui import QPainter, QBrush
7  from PyQt5.QtCore import Qt
8  import sys
9
10 class Example(QWidget):
11     def __init__(self):
12         super().__init__()
13         self.initUI()
14
15     def initUI(self):
16         self.setGeometry(300, 300, 355, 280)
17         self.setWindowTitle('Brushes')
18         self.show()
19
20     def paintEvent(self, e):
21         qp = QPainter()
22         qp.begin(self)
23         self.drawBrushes(qp)
24         qp.end()
25
26     def drawBrushes(self, qp):
27         brush = QBrush(Qt.SolidPattern)
28         qp.setBrush(brush)
29         qp.drawRect(10, 15, 90, 60)
30
31         brush.setStyle(Qt.Dense1Pattern)
32         qp.setBrush(brush)
33         qp.drawRect(130, 15, 90, 60)
34
35         brush.setStyle(Qt.Dense2Pattern)
36         qp.setBrush(brush)
37         qp.drawRect(250, 15, 90, 60)
38
39         brush.setStyle(Qt.DiagCrossPattern)
40         qp.setBrush(brush)
41         qp.drawRect(10, 105, 90, 60)
42
43         brush.setStyle(Qt.Dense5Pattern)
44         qp.setBrush(brush)
45         qp.drawRect(130, 105, 90, 60)
46
```

```

47     brush.setStyle(Qt.Dense6Pattern)
48     qp.setBrush(brush)
49     qp.drawRect(250, 105, 90, 60)
50
51     brush.setStyle(Qt.HorPattern)
52     qp.setBrush(brush)
53     qp.drawRect(10, 195, 90, 60)
54
55     brush.setStyle(Qt.VerPattern)
56     qp.setBrush(brush)
57     qp.drawRect(130, 195, 90, 60)
58
59     brush.setStyle(Qt.BDiagPattern)
60     qp.setBrush(brush)
61     qp.drawRect(250, 195, 90, 60)
62
63 def main():
64     app = QApplication(sys.argv)
65     ex = Example()
66     sys.exit(app.exec_())
67
68 if __name__ == '__main__':
69     main()

```

In our example, we draw nine different rectangles.

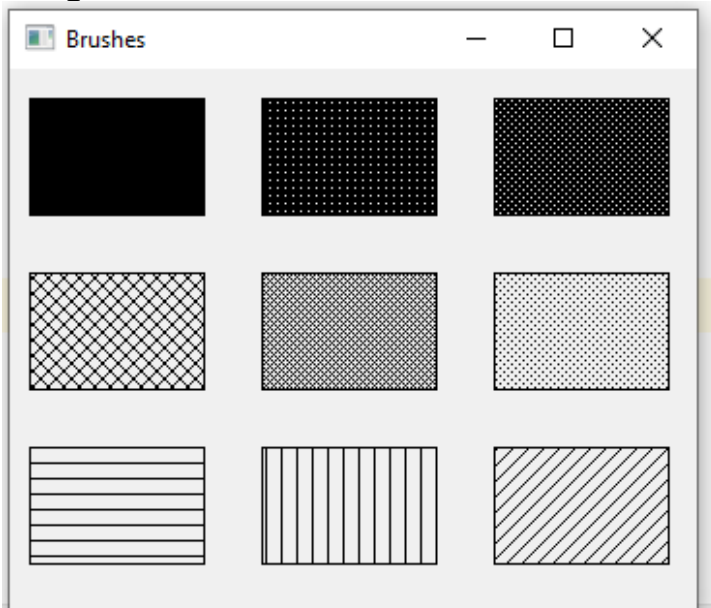
```

brush = QBrush(Qt.SolidPattern)
qp.setBrush(brush)
qp.drawRect(10, 15, 90, 60)

```

We define a brush object. We set it to the painter object and draw the rectangle by calling the `drawRect` method.

Output:



Bézier curve

Bézier curve is a cubic line. Bézier 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.

```
Qt1106.py
1  """
2  This program draws a Bézier curve with QPainterPath.
3  """
4  import sys
5  from PyQt5.QtGui import QPainter, QPainterPath
6  from PyQt5.QtWidgets import QWidget, QApplication
7
8  class Example(QWidget):
9      def __init__(self):
10         super().__init__()
11         self.initUI()
12
13     def initUI(self):
14         self.setGeometry(300, 300, 380, 250)
15         self.setWindowTitle('Bézier curve')
16         self.show()
17
18     def paintEvent(self, e):
19         qp = QPainter()
20         qp.begin(self)
21         qp.setRenderHint(QPainter.Antialiasing)
22         self.drawBezierCurve(qp)
23         qp.end()
24
25     def drawBezierCurve(self, qp):
26         path = QPainterPath()
27         path.moveTo(30, 30)
28         path.cubicTo(30, 30, 200, 350, 350, 30)
29
30         qp.drawPath(path)
31
32     def main():
33         app = QApplication(sys.argv)
34         ex = Example()
35         sys.exit(app.exec_())
36
37     if __name__ == '__main__':
38         main()
```

This example draws a Bézier curve.

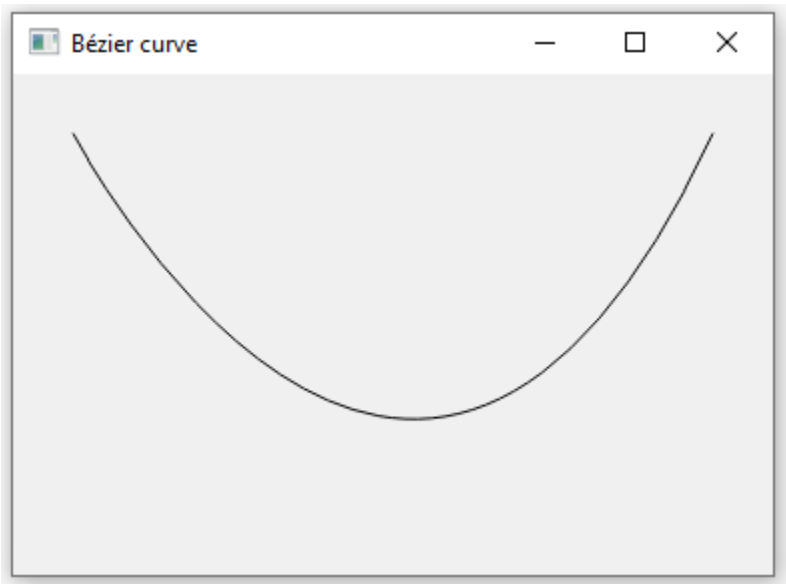
```
path = QPainterPath()
path.moveTo(30, 30)
path.cubicTo(30, 30, 200, 350, 350, 30)
```

We create a Bézier curve with `QPainterPath` `path`. The curve is created with `cubicTo()` method, which takes three points: starting point, control point, and ending point.

```
qp.drawPath(path)
```

The final path is drawn with `drawPath()` method.

Output:



In this part of the PyQt5 tutorial, we did some basic painting.

12. Custom widgets in PyQt5

PyQt5 has a rich set of widgets. However, no toolkit can provide all widgets that programmers might need in their applications. Toolkits usually provide only the most common widgets like buttons, text widgets, or sliders. If there is a need for a more specialised widget, we must create it ourselves.

Custom widgets are created by using the drawing tools provided by the toolkit. There are two basic possibilities: a programmer can modify or enhance an existing widget or he can create a custom widget from scratch.

PyQt5 burning widget

This is a widget that we can see in Nero, K3B, or other CD/DVD burning software.

```
Qt1201.py
1  """
2  In this example, we create a custom widget.
3  """
4
5  from PyQt5.QtWidgets import (QWidget, QSlider, QApplication,
6                               QHBoxLayout, QVBoxLayout)
7  from PyQt5.QtCore import QObject, Qt, pyqtSignal
8  from PyQt5.QtGui import QPainter, QFont, QColor, QPen
9  import sys
10
11
12  class Communicate(QObject):
13      updateBW = pyqtSignal(int)
14
15
16  class BurningWidget(QWidget):
17      def __init__(self):
18          super().__init__()
19
20          self.initUI()
21
22      def initUI(self):
23
24          self.setMinimumSize(1, 30)
25          self.value = 75
26          self.num = [75, 150, 225, 300, 375, 450, 525, 600, 675]
27
28      def setValue(self, value):
29
30          self.value = value
31
32      def paintEvent(self, e):
33
```



```

34     qp = QPainter()
35     qp.begin(self)
36     self.drawWidget(qp)
37     qp.end()
38
39     def drawWidget(self, qp):
40
41         MAX_CAPACITY = 700
42         OVER_CAPACITY = 750
43
44         font = QFont('Serif', 7, QFont.Light)
45         qp.setFont(font)
46
47         size = self.size()
48         w = size.width()
49         h = size.height()
50
51         step = int(round(w / 10))
52
53         till = int((w / OVER_CAPACITY) * self.value)
54         full = int((w / OVER_CAPACITY) * MAX_CAPACITY)
55
56         if self.value >= MAX_CAPACITY:
57
58             qp.setPen(QColor(255, 255, 255))
59             qp.setBrush(QColor(255, 255, 184))
60             qp.drawRect(0, 0, full, h)
61             qp.setPen(QColor(255, 175, 175))
62             qp.setBrush(QColor(255, 175, 175))
63             qp.drawRect(full, 0, till - full, h)
64
65         else:
66
67             qp.setPen(QColor(255, 255, 255))
68             qp.setBrush(QColor(255, 255, 184))
69             qp.drawRect(0, 0, till, h)
70
71         pen = QPen(QColor(20, 20, 20), 1,
72                  Qt.SolidLine)
73
74         qp.setPen(pen)
75         qp.setBrush(Qt.NoBrush)
76         qp.drawRect(0, 0, w - 1, h - 1)
77
78         j = 0
79
80         for i in range(step, 10 * step, step):
81
82             qp.drawLine(i, 0, i, 5)
83             metrics = qp.fontMetrics()
84             fw = metrics.width(str(self.num[j]))
85
86             x, y = int(i - fw/2), int(h / 2)
87             qp.drawText(x, y, str(self.num[j]))
88             j = j + 1
89
90

```

```

91 class Example(QWidget):
92
93     def __init__(self):
94         super().__init__()
95
96         self.initUI()
97
98     def initUI(self):
99
100         OVER_CAPACITY = 750
101
102         sld = QSlider(Qt.Horizontal, self)
103         sld.setFocusPolicy(Qt.NoFocus)
105         sld.setRange(1, OVER_CAPACITY)
105         sld.setValue(75)
106         sld.setGeometry(30, 40, 150, 30)
107
108         self.c = Communicate()
109         self.wid = BurningWidget()
110         self.c.updateBW[int].connect(self.wid.setValue)
111
112         sld.valueChanged[int].connect(self.changeValue)
113         hbox = QHBoxLayout()
114         hbox.addWidget(self.wid)
115         vbox = QVBoxLayout()
116         vbox.addStretch(1)
117         vbox.addLayout(hbox)
118         self.setLayout(vbox)
119
120         self.setGeometry(300, 300, 390, 210)
121         self.setWindowTitle('Burning widget')
122         self.show()
123
124     def changeValue(self, value):
125         self.c.updateBW.emit(value)
126         self.wid.repaint()
127
128
129 def main():
130     app = QApplication(sys.argv)
131     ex = Example()
132     sys.exit(app.exec_())
133
134
135 if __name__ == '__main__':
136     main()

```

In our example, we have a `QSlider` and a custom widget. The slider controls the custom widget. This widget shows graphically the total capacity of a medium and the free space available to us. The minimum value of our custom widget is 1, the maximum is `OVER_CAPACITY`. If we reach value `MAX_CAPACITY`, we begin drawing in red colour. This normally indicates overburning.

The burning widget is placed at the bottom of the window. This is achieved using one QHBoxLayout and one QVBoxLayout.

```
class BurningWidget(QWidget):  
    def __init__(self):  
        super().__init__()
```

The burning widget is based on the QWidget widget.

```
self.setMinimumSize(1, 30)
```

We change the minimum size (height) of the widget. The default value is a bit small for us.

```
font = QFont('Serif', 7, QFont.Light)  
qp.setFont(font)
```

We use a smaller font than the default one. This better suits our needs.

```
size = self.size()  
w = size.width()  
h = size.height()  
  
step = int(round(w / 10))  
  
till = int(((w / OVER_CAPACITY) * self.value))  
full = int(((w / OVER_CAPACITY) * MAX_CAPACITY))
```

We draw the widget dynamically. The greater is the window, the greater is the burning widget and vice versa. That is why we must calculate the size of the widget onto which we draw the custom widget. The `till` parameter determines the total size to be drawn. This value comes from the slider widget. It is a proportion of the whole area. The `full` parameter determines the point where we begin to draw in red colour.

The actual drawing consists of three steps. We draw the yellow or the red and yellow rectangle. Then we draw the vertical lines which divide the widget into several parts. Finally, we draw the numbers which indicate the capacity of the medium.

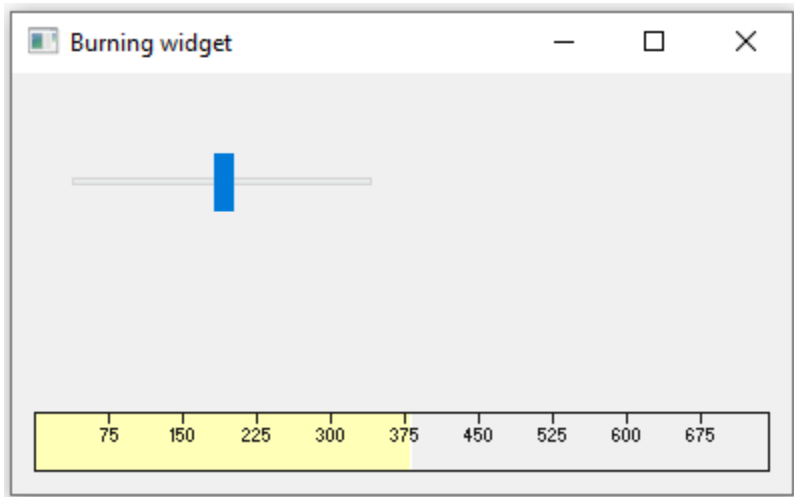
```
metrics = qp.fontMetrics()  
fw = metrics.width(str(self.num[j]))  
  
x, y = int(i - fw/2), int(h / 2)  
qp.drawText(x, y, str(self.num[j]))
```

We use font metrics to draw the text. We must know the width of the text in order to center it around the vertical line.

```
def changeValue(self, value):  
    self.c.updateBW.emit(value)  
    self.wid.repaint()
```

When we move the slider, the `changeValue()` method is called. Inside the method, we send a custom `updateBW` signal with a parameter. The parameter is the current value of the slider. The value is later used to calculate the capacity of the Burning widget to be drawn. The custom widget is then repainted.

Output:



In this part of the PyQT5 tutorial, we created a custom widget.

Mini Project: Tetris

In this chapter, we will create a Tetris game clone.

Tetris

The Tetris game is one of the most popular computer games ever created. The original game was designed and programmed by a Russian programmer *Alexey Pajitnov* in 1985. Since then, Tetris is available on almost every computer platform in lots of variations.

Tetris is called a falling block puzzle game. In this game, we have seven different shapes called *Tetrominoes*: a S-shape, a Z-shape, a T-shape, an L-shape, a Line-shape, a MirroredL-shape, and a Square-shape. Each of these shapes is formed with four squares. The shapes are falling down the board. The object of the Tetris game is to move and rotate the shapes so that they fit as much as possible. If we manage to form a row, the row is destroyed and we score. We play the Tetris game until we top out.



Figure: Tetrominoes

PyQt5 is a toolkit designed to create applications. There are other libraries which are targeted at creating computer games. Nevertheless, PyQt5 and other application toolkits can be used to create simple games.

Creating a computer game is a good way for enhancing programming skills.

The development

We do not have images for our Tetris game, we draw the Tetrominoes using the drawing API available in the PyQt5 programming toolkit. Behind every computer game, there is a mathematical model. So, it is in Tetris.

Some ideas behind the game:

- We use a `QtCore.QBasicTimer` to create a game cycle.
- The tetrominoes are drawn.
- The shapes move on a square by square basis (not pixel by pixel).
- Mathematically a board is a simple list of numbers.

The code consists of four classes: `Tetris`, `Board`, `Tetrominoe` and `Shape`. The `Tetris` class sets up the game. The `Board` is where the game logic is written. The `Tetrominoe` class contains names for all tetris pieces and the `Shape` class contains the code for a tetris piece.

```
Tetris.py
1  """
2  This is a Tetris game clone.
3  """
4  import random
5  import sys
6
7  from PyQt5.QtCore import Qt, QBasicTimer, pyqtSignal
8  from PyQt5.QtGui import QPainter, QColor
9  from PyQt5.QtWidgets import QMainWindow, QFrame, QDesktopWidget, QApplication
10
11 class Tetris(QMainWindow):
12     def __init__(self):
13         super().__init__()
14         self.initUI()
15
16     def initUI(self):
17         """initiates application UI"""
18         self.tboard = Board(self)
19         self.setCentralWidget(self.tboard)
20
21         self.statusbar = self.statusBar()
22         self.tboard.msg2StatusBar[str].connect(self.statusbar.showMessage)
23
24         self.tboard.start()
25
26         self.resize(180, 380)
27         self.center()
28         self.setWindowTitle('Tetris')
29         self.show()
```

```

30
31     def center(self):
32         """centers the window on the screen"""
33         screen = QDesktopWidget().screenGeometry()
34         size = self.geometry()
35         self.move(int((screen.width() - size.width()) / 2),
36                 int((screen.height() - size.height()) / 2))
37
38
39 class Board(QFrame):
40     msg2Statusbar = pyqtSignal(str)
41
42     BoardWidth = 10
43     BoardHeight = 22
44     Speed = 300
45
46     def __init__(self, parent):
47         super().__init__(parent)
48         self.initBoard()
49
50     def initBoard(self):
51         """initiates board"""
52         self.timer = QTimer()
53         self.isWaitingAfterLine = False
54
55         self.curX = 0
56         self.curY = 0
57         self.numLinesRemoved = 0
58         self.board = []
59
60         self.setFocusPolicy(Qt.StrongFocus)
61         self.isStarted = False
62         self.isPaused = False
63         self.clearBoard()
64
65     def shapeAt(self, x, y):
66         """determines shape at the board position"""
67         return self.board[(y * Board.BoardWidth) + x]
68
69     def setShapeAt(self, x, y, shape):
70         """sets a shape at the board"""
71         self.board[(y * Board.BoardWidth) + x] = shape
72
73     def squareWidth(self):
74         """returns the width of one square"""
75         return self.contentsRect().width() // Board.BoardWidth
76
77     def squareHeight(self):
78         """returns the height of one square"""
79         return self.contentsRect().height() // Board.BoardHeight
80
81     def start(self):
82         """starts game"""
83         if self.isPaused:
84             return
85
86         self.isStarted = True

```

```

87     self.isWaitingAfterLine = False
88     self.numLinesRemoved = 0
89     self.clearBoard()
90
91     self.msg2StatusBar.emit(str(self.numLinesRemoved))
92
93     self.newPiece()
94     self.timer.start(Board.Speed, self)
95
96     def pause(self):
97         """pauses game"""
98         if not self.isStarted:
99             return
100
101         self.isPaused = not self.isPaused
102
103         if self.isPaused:
104             self.timer.stop()
105             self.msg2StatusBar.emit("paused")
106
107         else:
108             self.timer.start(Board.Speed, self)
109             self.msg2StatusBar.emit(str(self.numLinesRemoved))
110
111         self.update()
112
113     def paintEvent(self, event):
114         """paints all shapes of the game"""
115         painter = QPainter(self)
116         rect = self.contentsRect()
117
118         boardTop = rect.bottom() - Board.BoardHeight * self.squareHeight()
119
120         for i in range(Board.BoardHeight):
121             for j in range(Board.BoardWidth):
122                 shape = self.shapeAt(j, Board.BoardHeight - i - 1)
123
124                 if shape != Tetrominoe.NoShape:
125                     self.drawSquare(painter,
126                                     rect.left() + j * self.squareWidth(),
127                                     boardTop + i * self.squareHeight(), shape)
128
129                 if self.curPiece.shape() != Tetrominoe.NoShape:
130                     for i in range(4):
131                         x = self.curX + self.curPiece.x(i)
132                         y = self.curY - self.curPiece.y(i)
133                         self.drawSquare(painter, rect.left() +
134                                         x * self.squareWidth(), boardTop +
135                                         (Board.BoardHeight - y - 1) * self.squareHeight(),
136                                         self.curPiece.shape())
137
138     def keyPressEvent(self, event):
139         """processes key press events"""
140         if not self.isStarted or \
141            self.curPiece.shape() == Tetrominoe.NoShape:
142             super(Board, self).keyPressEvent(event)
143             return

```



```

144
145     key = event.key()
146
147     if key == Qt.Key_P:
148         self.pause()
149         return
150
151     if self.isPaused:
152         return
153
154     elif key == Qt.Key_Left:
155         self.tryMove(self.curPiece, self.curX - 1, self.curY)
156
157     elif key == Qt.Key_Right:
158         self.tryMove(self.curPiece, self.curX + 1, self.curY)
159
160     elif key == Qt.Key_Down:
161         self.tryMove(self.curPiece.rotateRight(), self.curX, self.curY)
162
163     elif key == Qt.Key_Up:
164         self.tryMove(self.curPiece.rotateLeft(), self.curX, self.curY)
165
166     elif key == Qt.Key_Space:
167         self.dropDown()
168
169     elif key == Qt.Key_D:
170         self.oneLineDown()
171
172     else:
173         super(Board, self).keyPressEvent(event)
174
175     def timerEvent(self, event):
176         """handles timer event"""
177         if event.timerId() == self.timer.timerId():
178             if self.isWaitingAfterLine:
179                 self.isWaitingAfterLine = False
180                 self.newPiece()
181             else:
182                 self.oneLineDown()
183         else:
184             super(Board, self).timerEvent(event)
185
186     def clearBoard(self):
187         """clears shapes from the board"""
188         for i in range(Board.BoardHeight * Board.BoardWidth):
189             self.board.append(Tetrominoe.NoShape)
190
191     def dropDown(self):
192         """drops down a shape"""
193         newY = self.curY
194
195         while newY > 0:
196             if not self.tryMove(self.curPiece, self.curX, newY - 1):
197                 break
198
199             newY -= 1
200

```

```

201     self.pieceDropped()
202
203     def oneLineDown(self):
204         """goes one line down with a shape"""
205         if not self.tryMove(self.curPiece, self.curX, self.curY - 1):
206             self.pieceDropped()
207
208     def pieceDropped(self):
209         """after dropping shape, remove full lines and create new shape"""
210         for i in range(4):
211             x = self.curX + self.curPiece.x(i)
212             y = self.curY - self.curPiece.y(i)
213             self.setShapeAt(x, y, self.curPiece.shape())
214
215         self.removeFullLines()
216
217         if not self.isWaitingAfterLine:
218             self.newPiece()
219
220     def removeFullLines(self):
221         """removes all full lines from the board"""
222         numFullLines = 0
223         rowsToRemove = []
224
225         for i in range(Board.BoardHeight):
226             n = 0
227             for j in range(Board.BoardWidth):
228                 if not self.shapeAt(j, i) == Tetrominoe.NoShape:
229                     n = n + 1
230
231             if n == 10:
232                 rowsToRemove.append(i)
233
234         rowsToRemove.reverse()
235
236         for m in rowsToRemove:
237             for k in range(m, Board.BoardHeight):
238                 for l in range(Board.BoardWidth):
239                     self.setShapeAt(l, k, self.shapeAt(l, k + 1))
240
241         numFullLines = numFullLines + len(rowsToRemove)
242
243         if numFullLines > 0:
244             self.numLinesRemoved = self.numLinesRemoved + numFullLines
245             self.msg2StatusBar.emit(str(self.numLinesRemoved))
246
247             self.isWaitingAfterLine = True
248             self.curPiece.setShape(Tetrominoe.NoShape)
249             self.update()
250
251     def newPiece(self):
252         """creates a new shape"""
253         self.curPiece = Shape()
254         self.curPiece.setRandomShape()
255         self.curX = Board.BoardWidth // 2 + 1
256         self.curY = Board.BoardHeight - 1 + self.curPiece.minY()
257

```

```

258     if not self.tryMove(self.curPiece, self.curX, self.curY):
259         self.curPiece.setShape(Tetrominoe.NoShape)
260         self.timer.stop()
261         self.isStarted = False
262         self.msg2StatusBar.emit("Game over")
263
264     def tryMove(self, newPiece, newX, newY):
265         """tries to move a shape"""
266
267         for i in range(4):
268             x = newX + newPiece.x(i)
269             y = newY - newPiece.y(i)
270
271             if x < 0 or x >= Board.BoardWidth or y < 0 or \
272                 y >= Board.BoardHeight:
273                 return False
274
275             if self.shapeAt(x, y) != Tetrominoe.NoShape:
276                 return False
277
278         self.curPiece = newPiece
279         self.curX = newX
280         self.curY = newY
281         self.update()
282
283         return True
284
285     def drawSquare(self, painter, x, y, shape):
286         """draws a square of a shape"""
287         colorTable = [0x000000, 0xCC6666, 0x66CC66, 0x6666CC,
288                     0xCCCC66, 0xCC66CC, 0x66CCCC, 0xDAAA00]
289
290         color = QColor(colorTable[shape])
291         painter.fillRect(x + 1, y + 1, self.squareWidth() - 2,
292                         self.squareHeight() - 2, color)
293
294         painter.setPen(color.lighter())
295         painter.drawLine(x, y + self.squareHeight() - 1, x, y)
296         painter.drawLine(x, y, x + self.squareWidth() - 1, y)
297
298         painter.setPen(color.darker())
299         painter.drawLine(x + 1, y + self.squareHeight() - 1,
300                         x + self.squareWidth() - 1, y +
301                         self.squareHeight() - 1)
302         painter.drawLine(x + self.squareWidth() - 1,
303                         y + self.squareHeight() - 1, x +
304                         self.squareWidth() - 1, y + 1)
305
306     class Tetrominoe(object):
307         NoShape = 0
308         ZShape = 1
309         SShape = 2
310         LineShape = 3
311         TShape = 4
312         SquareShape = 5
313         LShape = 6
314         MirroredLShape = 7

```

```

315
316 class Shape(object):
317     coordsTable = (
318         ((0, 0), (0, 0), (0, 0), (0, 0)),
319         ((0, -1), (0, 0), (-1, 0), (-1, 1)),
320         ((0, -1), (0, 0), (1, 0), (1, 1)),
321         ((0, -1), (0, 0), (0, 1), (0, 2)),
322         ((-1, 0), (0, 0), (1, 0), (0, 1)),
323         ((0, 0), (1, 0), (0, 1), (1, 1)),
324         ((-1, -1), (0, -1), (0, 0), (0, 1)),
325         ((1, -1), (0, -1), (0, 0), (0, 1))
326     )
327
328     def __init__(self):
329         self.coords = [[0, 0] for i in range(4)]
330         self.pieceShape = Tetrominoe.NoShape
331
332         self.setShape(Tetrominoe.NoShape)
333
334     def shape(self):
335         """returns shape"""
336         return self.pieceShape
337
338     def setShape(self, shape):
339         """sets a shape"""
340         table = Shape.coordsTable[shape]
341
342         for i in range(4):
343             for j in range(2):
344                 self.coords[i][j] = table[i][j]
345
346         self.pieceShape = shape
347
348     def setRandomShape(self):
349         """chooses a random shape"""
350         self.setShape(random.randint(1, 7))
351
352     def x(self, index):
353         """returns x coordinate"""
354         return self.coords[index][0]
355
356     def y(self, index):
357         """returns y coordinate"""
358         return self.coords[index][1]
359
360     def setX(self, index, x):
361         """sets x coordinate"""
362         self.coords[index][0] = x
363
364     def setY(self, index, y):
365         """sets y coordinate"""
366         self.coords[index][1] = y
367
368     def minX(self):
369         """returns min x value"""
370         m = self.coords[0][0]
371         for i in range(4):

```

```

372         m = min(m, self.coords[i][0])
373
374     return m
375
376     def maxX(self):
377         """returns max x value"""
378         m = self.coords[0][0]
379         for i in range(4):
380             m = max(m, self.coords[i][0])
381
382     return m
383
384     def minY(self):
385         """returns min y value"""
386         m = self.coords[0][1]
387         for i in range(4):
388             m = min(m, self.coords[i][1])
389
390     return m
391
392     def maxY(self):
393         """returns max y value"""
394         m = self.coords[0][1]
395         for i in range(4):
396             m = max(m, self.coords[i][1])
397
398     return m
399
400     def rotateLeft(self):
401         """rotates shape to the left"""
402         if self.pieceShape == Tetrominoe.SquareShape:
403             return self
404
405         result = Shape()
406         result.pieceShape = self.pieceShape
407
408         for i in range(4):
409             result.setX(i, self.y(i))
410             result.setY(i, -self.x(i))
411
412     return result
413
414     def rotateRight(self):
415         """rotates shape to the right"""
416         if self.pieceShape == Tetrominoe.SquareShape:
417             return self
418
419         result = Shape()
420         result.pieceShape = self.pieceShape
421
422         for i in range(4):
423             result.setX(i, -self.y(i))
424             result.setY(i, self.x(i))
425
426     return result
427
428     def main():

```

```
429     app = QApplication([])
430     tetris = Tetris()
431     sys.exit(app.exec_())
432
433     if __name__ == '__main__':
434         main()
```

The game is simplified a bit so that it is easier to understand. The game starts immediately after it is launched. We can pause the game by pressing the `p` key. The `space` key will drop the Tetris piece instantly to the bottom. The game goes at constant speed, no acceleration is implemented. The score is the number of lines that we have removed.

```
self.tboard = Board(self)
self.setCentralWidget(self.tboard)
```

An instance of the `Board` class is created and set to be the central widget of the application.

```
self.statusbar = self.statusBar()
self.tboard.msg2Statusbar[str].connect(self.statusbar.showMessage)
```

We create a statusbar where we will display messages. We will display three possible messages: the number of lines already removed, the paused message, or the game over message. The `msg2Statusbar` is a custom signal that is implemented in the `Board` class. The `showMessage` is a built-in method that displays a message on a statusbar.

```
self.tboard.start()
```

This line initiates the game.

```
class Board(QFrame):
    msg2Statusbar = pyqtSignal(str)
    ...
```

A custom signal is created with `pyqtSignal`. The `msg2Statusbar` is a signal that is emitted when we want to write a message or the score to the statusbar.

```
BoardWidth = 10
BoardHeight = 22
Speed = 300
```

These are `Board`'s class variables. The `BoardWidth` and the `BoardHeight` define the size of the board in blocks. The `Speed` defines the speed of the game. Each 300 ms a new game cycle will start.


```

...
self.curX = 0
self.curY = 0
self.numLinesRemoved = 0
self.board = []
...

```

In the `initBoard` method we initialize some important variables. The `self.board` variable is a list of numbers from 0 to 7. It represents the position of various shapes and remains of the shapes on the board.

```

def shapeAt(self, x, y):
    """determines shape at the board position"""

    return self.board[(y * Board.BoardWidth) + x]

```

The `shapeAt` method determines the type of a shape at a given block.

```

def squareWidth(self):
    """returns the width of one square"""

    return self.contentsRect().width() // Board.BoardWidth

```

The board can be dynamically resized. As a consequence, the size of a block may change. The `squareWidth` calculates the width of the single square in pixels and returns it. The `Board.BoardWidth` is the size of the board in blocks.

```

def pause(self):
    """pauses game"""
    if not self.isStarted:
        return

    self.isPaused = not self.isPaused

    if self.isPaused:
        self.timer.stop()
        self.msg2StatusBar.emit("paused")
    else:
        self.timer.start(Board.Speed, self)
        self.msg2StatusBar.emit(str(self.numLinesRemoved))

    self.update()

```

The `pause` method pauses the game. It stops the timer and displays a message on the statusbar.

```

def paintEvent(self, event):
    """paints all shapes of the game"""
    painter = QPainter(self)
    rect = self.contentsRect()
    ...

```


The painting happens in the `paintEvent` method. The `QPainter` is responsible for all low-level painting in PyQt5.

```
for i in range(Board.BoardHeight):
    for j in range(Board.BoardWidth):
        shape = self.shapeAt(j, Board.BoardHeight - i - 1)

        if shape != Tetrominoe.NoShape:
            self.drawSquare(painter,
                            rect.left() + j * self.squareWidth(),
                            boardTop + i * self.squareHeight(), shape)
```

The painting of the game is divided into two steps. In the first step, we draw all the shapes, or remains of the shapes that have been dropped to the bottom of the board. All the squares are remembered in the `self.board` list variable. The variable is accessed using the `shapeAt` method.

```
if self.curPiece.shape() != Tetrominoe.NoShape:
    for i in range(4):
        x = self.curX + self.curPiece.x(i)
        y = self.curY - self.curPiece.y(i)
        self.drawSquare(painter, rect.left() + x * self.squareWidth(),
                        boardTop + (Board.BoardHeight - y - 1) * self.squareHeight(),
                        self.curPiece.shape())
```

he next step is the drawing of the actual piece that is falling down.

```
elif key == Qt.Key_Right:
    self.tryMove(self.curPiece, self.curX + 1, self.curY)
```

In the `keyPressEvent` method we check for pressed keys. If we press the right arrow key, we try to move the piece to the right. We say try because the piece might not be able to move.

```
elif key == Qt.Key_Up:
    self.tryMove(self.curPiece.rotateLeft(), self.curX, self.curY)
```

The `Up` arrow key will rotate the falling piece to the left.

```
elif key == Qt.Key_Space:
    self.dropDown()
```

The `Space` key will drop the falling piece instantly to the bottom.

```
elif key == Qt.Key_D:
    self.oneLineDown()
```

Pressing the `d` key, the piece will go one block down. It can be used to accelerate the falling of a piece a bit.

```
def timerEvent(self, event):
    """handles timer event"""
    if event.timerId() == self.timer.timerId():
        if self.isWaitingAfterLine:
            self.isWaitingAfterLine = False
            self.newPiece()
        else:
            self.oneLineDown()

    else:
        super(Board, self).timerEvent(event)
```

In the timer event, we either create a new piece after the previous one was dropped to the bottom or we move a falling piece one line down.

```
def clearBoard(self):
    """clears shapes from the board"""
    for i in range(Board.BoardHeight * Board.BoardWidth):
        self.board.append(Tetrominoe.NoShape)
```

The `clearBoard` method clears the board by setting `Tetrominoe.NoShape` at each block of the board.

```
def removeFullLines(self):
    """removes all full lines from the board"""
    numFullLines = 0
    rowsToRemove = []

    for i in range(Board.BoardHeight):
        n = 0
        for j in range(Board.BoardWidth):
            if not self.shapeAt(j, i) == Tetrominoe.NoShape:
                n = n + 1

        if n == 10:
            rowsToRemove.append(i)

    rowsToRemove.reverse()

    for m in rowsToRemove:
        for k in range(m, Board.BoardHeight):
            for l in range(Board.BoardWidth):
                self.setShapeAt(l, k, self.shapeAt(l, k + 1))

    numFullLines = numFullLines + len(rowsToRemove)
    ...
```

If the piece hits the bottom, we call the `removeFullLines` method. We find out all full lines and remove them. We do it by moving all lines above the current full line to be removed one line down. Notice that we reverse the order of the lines to be removed. Otherwise, it would not work correctly. In our case we use a *naive gravity*. This means that the pieces may be floating above empty gaps.

```
def newPiece(self):
    """creates a new shape"""
    self.curPiece = Shape()
    self.curPiece.setRandomShape()
    self.curX = Board.BoardWidth // 2 + 1
    self.curY = Board.BoardHeight - 1 + self.curPiece.minY()

    if not self.tryMove(self.curPiece, self.curX, self.curY):
        self.curPiece.setShape(Tetrominoe.NoShape)
        self.timer.stop()
        self.isStarted = False
        self.msg2StatusBar.emit("Game over")
```

The `newPiece` method creates randomly a new tetris piece. If the piece cannot go into its initial position, the game is over.

```
def tryMove(self, newPiece, newX, newY):
    """tries to move a shape"""
    for i in range(4):
        x = newX + newPiece.x(i)
        y = newY - newPiece.y(i)
        if x < 0 or x >= Board.BoardWidth or y < 0 or y >= Board.BoardHeight:
            return False

        if self.shapeAt(x, y) != Tetrominoe.NoShape:
            return False
    self.curPiece = newPiece
    self.curX = newX
    self.curY = newY
    self.update()

    return True
```

In the `tryMove` method we try to move our shapes. If the shape is at the edge of the board or is adjacent to some other piece, we return `False`. Otherwise we place the current falling piece to a new position.

```
class Tetrominoe(object):
    NoShape = 0
    ZShape = 1
    SShape = 2
    LineShape = 3
    TShape = 4
    SquareShape = 5
    LShape = 6
    MirroredLShape = 7
```

The `Tetrominoe` class holds names of all possible shapes. We have also a `NoShape` for an empty space.

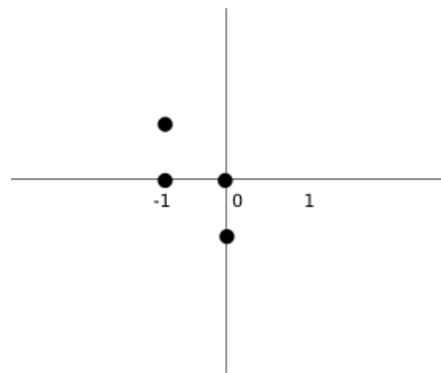
The `Shape` class saves information about a tetris piece.

```
class Shape(object):
    coordsTable = (
        ((0, 0), (0, 0), (0, 0), (0, 0)),
        ((0, -1), (0, 0), (-1, 0), (-1, 1)),
        ...
    )
    ...
```

The `coordsTable` tuple holds all possible coordinate values of our tetris pieces. This is a template from which all pieces take their coordinate values.

```
self.coords = [[0,0] for i in range(4)]
```

Upon creation we create an empty coordinates list. The list will save the coordinates of the tetris piece.



The above image will help understand the coordinate values a bit more. For example, the tuples `(0, -1)`, `(0, 0)`, `(-1, 0)`, `(-1, -1)` represent a Z-shape. The diagram illustrates the shape.

```
def rotateLeft(self):
    """rotates shape to the left"""
    if self.pieceShape == Tetrominoe.SquareShape:
        return self

    result = Shape()
    result.pieceShape = self.pieceShape

    for i in range(4):
        result.setX(i, self.y(i))
        result.setY(i, -self.x(i))

    return result
```

The `rotateLeft` method rotates a piece to the left. The square does not have to be rotated. That is why we simply return the reference to the current object. A new piece is created and its coordinates are set to the ones of the rotated piece.

