Object Oriented Programming II

Dr. Burak Kaleci

March 22, 2019

Content

Modules

PyQt5

Introduction

- As your program gets longer, you may want to split it into several files for easier maintenance.
- You may also want to use a handy function that you've written in several programs without copying its definition into each program.
- To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.
- **Such a file is called a module**; definitions from a module can be imported into other modules or into the main module.
- A module is a file containing Python definitions and statements. The file name is the module name with the suffix .py appended.

Introduction

- For instance, create a file called fibo.py in the current directory with the following contents:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

- Now enter the Python interpreter and import this module with the following command:

  import fibo

- This does not enter the names of the functions defined in fibo directly in the current symbol table; it only enters the module name fibo there.

Introduction

- Using the module name you can access the functions:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

- Within a module, the module's name (as a string) is available as the value of the global variable __name__.
- If you intend to use a function often you can assign it to a local name:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Introduction

- There is a variant of the import statement that imports names from a module directly into the importing module's symbol table. For example:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- This does not introduce the module name from which the imports are taken in the local symbol table **(so in the example, fibo is not defined)**.
- There is even a variant to import all names that a module defines:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- This imports all names except those beginning with an underscore (_).
- In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Introduction

- If the module name is followed by as, then the name following as is bound directly to the imported module.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

- This is effectively importing the module in the same way that import fibo will do, with the only difference of it being available as fib.
- It can also be used when utilising from with similar effects:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Executing modules as scripts

- A module can contain executable statements as well as function definitions.
- These statements are intended to initialize the module. They are executed only the first time the module name is encountered in an import statement.
- They are also run if the file is executed as a script.
- When you run a Python module with:

  python fibo.py ¡arguments¿
- The code in the module will be executed, just as if you imported it, but with the __name__ set to "__main__".
- That means that by adding this code at the end of your module:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

Executing modules as scripts

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

- you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the "main" file:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

- If the module is imported, the code is not run:

  import fibo

- This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

Introduction

- PyQt is a library that lets you use the Qt GUI framework from Python. Qt itself is written in C++. By using it from Python, you can build applications much more quickly while not sacrificing much of the speed of C++.
- PyQt5 refers to the most recent version 5 of Qt. You may still find the occasional mention of (Py)Qt4 on the web, but it is old and no longer supported.
- **An interesting new competitor to PyQt is Qt for Python.** Its API is virtually identical. Unlike PyQt, it is licensed under the LGPL and can thus be used for free in commercial projects. **It's backed by the Qt company, and thus likely the future.** We use PyQt here because it is more mature. Since the APIs are so similar, you can easily switch your apps to Qt for Python later.

PyQt5 modules

- QtCore
- QtGui
- QtWidgets
- QtMultimedia
- QtBluetooth
- QtNetwork
- QtPositioning
- Enginio
- QtWebSockets
- QtWebKit
- QtWebKitWidgets
- QtXml
- QtSvg
- QtSql
- QtTest

PyQt5 modules

- 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.

PyQt5 modules

- 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 **QtWebKit** contains classes for a web browser implementation based on the WebKit2 library. The **QtWebKitWidgets** contains classes for a WebKit1 based implementation of a web browser for use in QtWidgets based applications.
- 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.

First Example

```
2 import sys
3 from PyQt5.QtWidgets import QApplication, QWidget
4
5 if __name__ == '__main__':
6
7     app = QApplication(sys.argv)
8
9     w = QWidget()
10    w.resize(250, 150)
11    w.move(300, 300)
12    w.setWindowTitle('Simple')
13    w.show()
14
15    sys.exit(app.exec_())
```

- In lines 2 and 3, we provide the necessary imports. The basic widgets are located in PyQt5.QtWidgets module.
- Line 7 creates an application object. 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.
- 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.

First Example

```
2 import sys
3 from PyQt5.QtWidgets import QApplication, QWidget
4
5 if __name__ == '__main__':
6
7     app = QApplication(sys.argv)
8
9     w = QWidget()
10    w.resize(250, 150)
11    w.move(300, 300)
12    w.setWindowTitle('Simple')
13    w.show()
14
15    sys.exit(app.exec_())
```

- The resize() method resizes the widget. It is 250px wide and 150px high.
- The move() method moves the widget to a position on the screen at x=300, y=300 coordinates.
- We set the title of the window with setWindowTitle(). The title is shown in the titlebar.
- The show() method displays the widget on the screen. A widget is first created in memory and later shown on the screen.
- 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.

An application icon

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

- 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.

An application icon

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

- The creation of the GUI is delegated to the initUI() method.
- The setGeometry() does two things: it locates the window on the screen and sets it 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.
- 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.

Closing a window

```
1 import sys
2 from PyQt5.QtWidgets import QWidget, QPushButton, QApplication
3
4 class Example(QWidget):
5
6     def __init__(self):
7         super().__init__()
8         self.initUI()
9
10    def initUI(self):
11
12        qbtn = QPushButton('Quit', self)
13        qbtn.clicked.connect(QApplication.instance().quit)
14        qbtn.resize(qbtn.sizeHint())
15        qbtn.move(50, 50)
16
17        self.setGeometry(300, 300, 250, 150)
18        self.setWindowTitle('Quit button')
19        self.show()
20
21 if __name__ == '__main__':
22
23    app = QApplication(sys.argv)
24    ex = Example()
25    sys.exit(app.exec_())
```

- 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 top level windows.

Closing a window

```
1 import sys
2 from PyQt5.QtWidgets import QWidget, QPushButton, QApplication
3
4 class Example(QWidget):
5
6     def __init__(self):
7         super().__init__()
8         self.initUI()
9
10    def initUI(self):
11
12        qbtn = QPushButton('Quit', self)
13        qbtn.clicked.connect(QApplication.instance().quit)
14        qbtn.resize(qbtn.sizeHint())
15        qbtn.move(50, 50)
16
17        self.setGeometry(300, 300, 250, 150)
18        self.setWindowTitle('Quit button')
19        self.show()
20
21 if __name__ == '__main__':
22
23     app = QApplication(sys.argv)
24     ex = Example()
25     sys.exit(app.exec_())
```

- 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 Example widget, which is a QWidget by inheritance.
- 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.

Message Box

```
1  import sys
2  from PyQt5.QtWidgets import QWidget, QMessageBox, QDesktopWidget, QApplication
3
4  class Example(QWidget):
5
6      def __init__(self):
7          super().__init__()
8          self.initUI()
9
10     def initUI(self):
11         self.resize(250, 150)
12         self.center()
13         self.setWindowTitle('Message box')
14         self.show()
15
16     def center(self):
17         qr = self.frameGeometry()
18         cp = QDesktopWidget().availableGeometry().center()
19         qr.moveCenter(cp)
20         self.move(qr.topLeft())
21
22     def closeEvent(self, event):
23         reply = QMessageBox.question(self, 'Message',
24             "Are you sure to quit?", QMessageBox.Yes |
25             QMessageBox.No, QMessageBox.No)
26
27         if reply == QMessageBox.Yes:
28             event.accept()
29         else:
30             event.ignore()
31
32 if __name__ == '__main__':
33
34     app = QApplication(sys.argv)
35     ex = Example()
36     sys.exit(app.exec_())
```

Message Box

- By default, if we click on the x button on the title bar, 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.
- The QDesktopWidget class provides information about the user's desktop, including the screen size.
- The code that will center the window is placed in the custom center() method.
- We get a rectangle specifying the geometry of the main window. This includes any window frame.
- We figure out the screen resolution of our monitor. And from this resolution, we get the center point.
- 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.
- 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.

Message Box

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

Statusbar

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

- The QMainWindow class provides a main application window. This enables to create a classic application skeleton with a statusbar, toolbars, and a menubar.
- A statusbar is a widget that is used for displaying status information.
- The statusbar is created with the help of the QMainWindow widget.
- 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.

Simple menu

```
1  import sys
2  from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
3  from PyQt5.QtGui import QIcon
4
5
6  class Example(QMainWindow):
7
8      def __init__(self):
9          super().__init__()
10
11         self.initUI()
12
13
14     def initUI(self):
15
16         exitAct = QAction(QIcon('exit.png'), '&Exit', self)
17         exitAct.setShortcut('Ctrl+Q')
18         exitAct.setStatusTip('Exit application')
19         exitAct.triggered.connect(qApp.quit)
20
21         self.statusBar()
22
23         menubar = self.menuBar()
24         fileMenu = menubar.addMenu('&File')
25         fileMenu.addAction(exitAct)
26
27         self.setGeometry(300, 300, 300, 200)
28         self.setWindowTitle('Simple menu')
29         self.show()
30
31
32 if __name__ == '__main__':
33
34     app = QApplication(sys.argv)
35     ex = Example()
36     sys.exit(app.exec_())
```

- A menubar is a common part of a GUI application. It is a group of commands located in various menus.
- In the example, we create a menubar with one menu. This menu will contain one action which will terminate the application if selected. A statusbar is created as well. The action is accessible with the Ctrl+Q shortcut.

Simple menu

```
1  import sys
2  from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
3  from PyQt5.QtGui import QIcon
4
5
6  class Example(QMainWindow):
7
8      def __init__(self):
9          super().__init__()
10
11         self.initUI()
12
13
14     def initUI(self):
15
16         exitAct = QAction(QIcon('exit.png'), '&Exit', self)
17         exitAct.setShortcut('Ctrl+Q')
18         exitAct.setStatusTip('Exit application')
19         exitAct.triggered.connect(qApp.quit)
20
21         self.statusBar()
22
23         menubar = self.menuBar()
24         fileMenu = menubar.addMenu('&File')
25         fileMenu.addAction(exitAct)
26
27         self.setGeometry(300, 300, 300, 200)
28         self.setWindowTitle('Simple menu')
29         self.show()
30
31
32 if __name__ == '__main__':
33
34     app = QApplication(sys.argv)
35     ex = Example()
36     sys.exit(app.exec_())
```

- QAction is an abstraction for actions performed with a menubar, toolbar, or with a custom keyboard shortcut.
- In lines 16-18, we create an action with a specific icon and an 'Exit' label. Furthermore, a shortcut is defined for this action. The line 18 creates a status tip which is shown in the statusbar when we hover a mouse pointer over the menu item.

Simple menu

```
1  import sys
2  from PyQt5.QtWidgets import QMainWindow, QAction, qApp, QApplication
3  from PyQt5.QtGui import QIcon
4
5
6  class Example(QMainWindow):
7
8      def __init__(self):
9          super().__init__()
10
11         self.initUI()
12
13
14     def initUI(self):
15
16         exitAct = QAction(QIcon('exit.png'), '&Exit', self)
17         exitAct.setShortcut('Ctrl+Q')
18         exitAct.setStatusTip('Exit application')
19         exitAct.triggered.connect(qApp.quit)
20
21         self.statusBar()
22
23         menubar = self.menuBar()
24         fileMenu = menubar.addMenu('&File')
25         fileMenu.addAction(exitAct)
26
27         self.setGeometry(300, 300, 300, 200)
28         self.setWindowTitle('Simple menu')
29         self.show()
30
31
32 if __name__ == '__main__':
33
34     app = QApplication(sys.argv)
35     ex = Example()
36     sys.exit(app.exec_())
```

- When we select this particular action, a triggered signal is emitted (Line 19).
- The signal is connected to the quit() method of the QApplication widget. This terminates the application.
- The menuBar() method creates a menubar. We create a file menu with addMenu() and add the action with addAction().

Submenu

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

- A submenu is a menu located inside another menu.
- 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.
- New menu is created with QMenu (Line 17).
- An action is added to the submenu with addAction() (Lines 18 and 19).

Layout management

- 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.
- 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
  - If we decide to change our layout, we must completely redo our layout, which is tedious and time consuming

Absolute positioning

```
1 import sys
2 from PyQt5.QtWidgets import QWidget, QLabel, QApplication
3
4 class Example(QWidget):
5
6     def __init__(self):
7         super().__init__()
8
9         self.initUI()
10
11
12     def initUI(self):
13
14         lbl1 = QLabel('Zetcode', 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
28 if __name__ == '__main__':
29
30     app = QApplication(sys.argv)
31     ex = Example()
32     sys.exit(app.exec_())
```

- 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.

Box layout

```
1  import sys
2  from PyQt5.QtWidgets import (QWidget, QPushButton,
3      QHBoxLayout, QVBoxLayout, QApplication)
4
5
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10
11         self.initUI()
12
13     def initUI(self):
14
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 if __name__ == '__main__':
34
35     app = QApplication(sys.argv)
36     ex = Example()
37     sys.exit(app.exec_())
```

- 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.
- 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 HBoxLayout and a QVBoxLayout.

Box layout

```
1  import sys
2  from PyQt5.QtWidgets import (QWidget, QPushButton,
3      QHBoxLayout, QVBoxLayout, QApplication)
4
5
6  class Example(QWidget):
7
8      def __init__(self):
9          super().__init__()
10
11         self.initUI()
12
13     def initUI(self):
14
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 if __name__ == '__main__':
34
35     app = QApplication(sys.argv)
36     ex = Example()
37     sys.exit(app.exec_())
```

- Lines 15 and 16 create two push buttons.
- Lines 18-21 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.
- 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. (Lines 23-25).
- Line 27 sets the main layout of the window.

QGridLayout

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

- QGridLayout is the most universal layout class. It divides the space into rows and columns.
- In our example, we create a grid of buttons.
- The instance of a QGridLayout is created and set to be the layout for the application window (Lines 14 and 15).
- Lines 17-21 define the labels used later for buttons.
- Line 23 creates a list of positions in the grid.
- In lines 24-28, buttons are created and added to the layout with the addWidget() method.

Review example

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

- Widgets can span multiple columns or rows in a grid. In the next example we illustrate this.
- 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.
- Lines 21 and 22 create a grid layout and set spacing between widgets.
- 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.

Events and signals

- GUI applications are event-driven. Events are generated mainly by the user of an application.
- But they can be generated by other means as well; e.g. an Internet connection, a window manager, or a timer.
- When we call the application's exec_() method, the application enters the main loop.
- The main loop fetches events and sends them to the objects.

Events and signals

- In the event model, there are three participants:
  - Event source
  - Event object
  - Event target
- The event source is the object whose state changes. It generates events.
- The event object (event) encapsulates the state changes in the event source.
- The event target is the object that wants to be notified. Event source object delegates the task of handling an event to the event target.
- PyQt5 has a unique signal and slot mechanism to deal with events. Signals and slots are used for communication between objects.
- A signal is emitted when a particular event occurs. A slot can be any Python callable. A slot is called when its connected signal is emitted.

Signals and slots

```
1  import sys
2  from PyQt5.QtCore import Qt
3  from PyQt5.QtWidgets import (QWidget, QLCDNumber, QSlider,
4      QVBoxLayout, QApplication)
5
6
7  class Example(QWidget):
8
9      def __init__(self):
10         super().__init__()
11
12         self.initUI()
13
14
15     def initUI(self):
16
17         lcd = QLCDNumber(self)
18         sld = QSlider(Qt.Horizontal, self)
19
20         vbox = QVBoxLayout()
21         vbox.addWidget(lcd)
22         vbox.addWidget(sld)
23
24         self.setLayout(vbox)
25         sld.valueChanged.connect(lcd.display)
26
27         self.setGeometry(300, 300, 250, 150)
28         self.setWindowTitle('Signal and slot')
29         self.show()
30
31
32 if __name__ == '__main__':
33
34     app = QApplication(sys.argv)
35     ex = Example()
36     sys.exit(app.exec_())
```

- In our example, we display a QtGui.QLCDNumber and a QtGui.QSlider. We change the lcd number by dragging the slider knob.
- Line 25 connects 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.

Reimplementing event handler

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

- In our example, we reimplement the keyPressEvent() event handler (Lines 20-23).
- If we click the Escape button, the application terminates.

Event object

```
1  import sys
2  from PyQt5.QtCore import Qt
3  from PyQt5.QtWidgets import QWidget, QApplication, QGridLayout, QLabel
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10
11     def initUI(self):
12         grid = QGridLayout()
13
14         x = 0
15         y = 0
16         self.text = "x: {0},  y: {1}".format(x, y)
17         self.label = QLabel(self.text, self)
18         grid.addWidget(self.label, 0, 0, Qt.AlignTop)
19
20         self.setMouseTracking(True)
21         self.setLayout(grid)
22
23         self.setGeometry(300, 300, 350, 200)
24         self.setWindowTitle('Event object')
25         self.show()
26
27     def mouseMoveEvent(self, e):
28         x = e.x()
29         y = e.y()
30
31         text = "x: {0},  y: {1}".format(x, y)
32         self.label.setText(text)
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- Event object is a Python object that contains a number of attributes describing the event. Event object is specific to the generated event type.
- In this example, we display the x and y coordinates of a mouse pointer in a label widget.
- The x and y coordinates are displayed in a QLabel widget (Lines 16-17).

Event object

```
1  import sys
2  from PyQt5.QtCore import Qt
3  from PyQt5.QtWidgets import QWidget, QApplication, QGridLayout, QLabel
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10
11     def initUI(self):
12         grid = QGridLayout()
13
14         x = 0
15         y = 0
16         self.text = "x: {0},  y: {1}".format(x, y)
17         self.label = QLabel(self.text, self)
18         grid.addWidget(self.label, 0, 0, Qt.AlignTop)
19
20         self.setMouseTracking(True)
21         self.setLayout(grid)
22
23         self.setGeometry(300, 300, 350, 200)
24         self.setWindowTitle('Event object')
25         self.show()
26
27     def mouseMoveEvent(self, e):
28         x = e.x()
29         y = e.y()
30
31         text = "x: {0},  y: {1}".format(x, y)
32         self.label.setText(text)
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- 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.
- 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 (Lines 27-32).

Event sender

```
1  import sys
2  from PyQt5.QtWidgets import QMainWindow, QPushButton, QApplication
3
4  class Example(QMainWindow):
5
6      def __init__(self):
7          super().__init__()
8
9          self.initUI()
10
11
12     def initUI(self):
13
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, 290, 150)
26         self.setWindowTitle('Event sender')
27         self.show()
28
29     def buttonClicked(self):
30
31         sender = self.sender()
32         self.statusBar().showMessage(sender.text() + ' was pressed')
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- Sometimes it is convenient to know which widget is the sender of a signal. For this, PyQt5 has the sender() method.
- We have two buttons in our example. In the buttonClicked() method we determine which button we have clicked by calling the sender() method.
- In lines 20 and 21 Both buttons are connected to the same slot.
- Lines 29-32 determine the signal source by calling the sender() method. In the statusbar of the application, we show the label of the button being pressed.

Emitting signals

```
1  import sys
2  from PyQt5.QtCore import pyqtSignal, QObject
3  from PyQt5.QtWidgets import QMainWindow, QApplication
4
5
6  class Communicate(QObject):
7
8      closeApp = pyqtSignal()
9
10
11 class Example(QMainWindow):
12
13     def __init__(self):
14         super().__init__()
15
16         self.initUI()
17
18
19     def initUI(self):
20
21         self.c = Communicate()
22         self.c.closeApp.connect(self.close)
23
24         self.setGeometry(300, 300, 290, 150)
25         self.setWindowTitle('Emit signal')
26         self.show()
27
28
29     def mousePressEvent(self, event):
30
31         self.c.closeApp.emit()
32
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- Objects created from a QObject can emit signals. The following example shows how we to emit custom signals.
- 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.
- A signal is created with the pyqtSignal() as a class attribute of the external Communicate class (Lines 6-8).
- The custom closeApp signal is connected to the close() slot of the QMainWindow (Lines 21-22).
- When we click on the window with a mouse pointer, the closeApp signal is emitted. The application terminates (Lines 29-31).

Dialogs

- Dialog windows or dialogs are an indispensable part of most modern GUI applications.
- 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.
- A dialog is used to input data, modify data, change the application settings etc.

QInputDialog

```
1  from PyQt5.QtWidgets import (QWidget, QPushButton, QLineEdit,
2      QInputDialog, QApplication)
3  import sys
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9
10         self.initUI()
11
12     def initUI(self):
13
14         self.btn = QPushButton('Dialog', self)
15         self.btn.move(20, 20)
16         self.btn.clicked.connect(self.showDialog)
17
18         self.le = QLineEdit(self)
19         self.le.move(130, 22)
20
21         self.setGeometry(300, 300, 290, 150)
22         self.setWindowTitle('Input dialog')
23         self.show()
24
25     def showDialog(self):
26
27         text, ok = QInputDialog.getText(self, 'Input Dialog',
28             'Enter your name:')
29
30         if ok:
31             self.le.setText(str(text))
32
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- 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.
- 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.

QInputDialog

```
1  from PyQt5.QtWidgets import (QWidget, QPushButton, QLineEdit,
2      QInputDialog, QApplication)
3  import sys
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9
10         self.initUI()
11
12     def initUI(self):
13
14         self.btn = QPushButton('Dialog', self)
15         self.btn.move(20, 20)
16         self.btn.clicked.connect(self.showDialog)
17
18         self.le = QLineEdit(self)
19         self.le.move(130, 22)
20
21         self.setGeometry(300, 300, 290, 150)
22         self.setWindowTitle('Input dialog')
23         self.show()
24
25     def showDialog(self):
26
27         text, ok = QInputDialog.getText(self, 'Input Dialog',
28             'Enter your name:')
29
30         if ok:
31             self.le.setText(str(text))
32
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- Line 27 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.
- The text that we have received from the dialog is set to the line edit widget with setText() (Lines 30 and 31).

QColorDialog

```
1 from PyQt5.QtWidgets import (QWidget, QPushButton, QFrame,
2     QColorDialog, QApplication)
3 from PyQt5.QtGui import QColor
4 import sys
5
6 class Example(QWidget):
7
8     def __init__(self):
9         super().__init__()
10        self.initUI()
11
12    def initUI(self):
13        col = QColor(0, 0, 0)
14
15        self.btn = QPushButton('Dialog', self)
16        self.btn.move(20, 20)
17        self.btn.clicked.connect(self.showDialog)
18
19        self.frm = QFrame(self)
20        self.frm.setStyleSheet("QWidget { background-color: %s }" % col.name())
21        self.frm.setGeometry(130, 22, 100, 100)
22
23        self.setGeometry(300, 300, 250, 180)
24        self.setWindowTitle('Color dialog')
25        self.show()
26
27    def showDialog(self):
28        col = QColorDialog.getColor()
29
30        if col.isValid():
31            self.frm.setStyleSheet("QWidget { background-color: %s }"
32                % col.name())
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- QColorDialog provides a dialog widget for selecting colour values.
- 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.
- Line 13 is an initial colour of the QFrame background.
- Line 28 pops up the QColorDialog.
- Lines 30-32 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.

QFontDialog

```
1  from PyQt5.QtWidgets import (QWidget, QVBoxLayout, QPushButton,
2      QSizePolicy, QLabel, QFontDialog, QApplication)
3  import sys
4
5  class Example(QWidget):
6
7      def __init__(self):
8          super().__init__()
9          self.initUI()
10
11     def initUI(self):
12
13         vbox = QVBoxLayout()
14         btn = QPushButton('Dialog', self)
15         btn.setSizePolicy(QSizePolicy.Fixed,QSizePolicy.Fixed)
16         btn.move(20, 20)
17         vbox.addWidget(btn)
18         btn.clicked.connect(self.showDialog)
19         self.lbl = QLabel('Knowledge only matters', self)
20         self.lbl.move(130, 20)
21         vbox.addWidget(self.lbl)
22         self.setLayout(vbox)
23
24         self.setGeometry(300, 300, 250, 180)
25         self.setWindowTitle('Font dialog')
26         self.show()
27
28     def showDialog(self):
29
30         font, ok = QFontDialog.getFont()
31         if ok:
32             self.lbl.setFont(font)
33
34 if __name__ == '__main__':
35
36     app = QApplication(sys.argv)
37     ex = Example()
38     sys.exit(app.exec_())
```

- QFontDialog is a dialog widget for selecting a font.
- In our example, we have a button and a label. With the QFontDialog, we change the font of the label.
- 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 (Line 30).
- If we clicked Ok, the font of the label is changed with setFont() (Lines 31 and 32).