Прохоренок Н.А.

PyQt. Создание оконных приложений на Python 3

Оглавление

Глава 1. Знакомство с PyQt

Рассматриваемые версии программ: PyQt 4.8.3 и Python 3.2.0.

1.1. Установка PyQt

Библиотека PyQt не входит в состав стандартной библиотеки Python. Прежде чем начать изучение основ, необходимо установить PyQt на компьютер.

1. Скачиваем программу установки PyQt-Py3.2-x86-gpl-4.8.3-1.exe со страницы **http://www.riverbankcomputing.co.uk/software/pyqt/download** и запускаем ее с помощью двойного щелчка на значке файла.
2. В открывшемся окне (рис. 1.1) нажимаем кнопку **Next**.
3. На следующем шаге (рис. 1.2) соглашаемся с лицензионным соглашением, нажимая кнопку **I Agree**.
4. В следующем диалоговом окне (рис. 1.3) можно выбрать компоненты, которые следует установить. Оставляем выбранными все компоненты и нажимаем кнопку **Next**.
5. На следующем шаге (рис. 1.4) задается путь к каталогу, в котором расположен интерпретатор Python (C:\Python32\). Нажимаем кнопку **Install** для запуска процесса установки PyQt.
6. После завершения установки будет выведено окно, изображенное на рис. 1.5. Нажимаем кнопку **Finish** для выхода из программы установки.

Рис. 1.1. Установка PyQt. Шаг 1

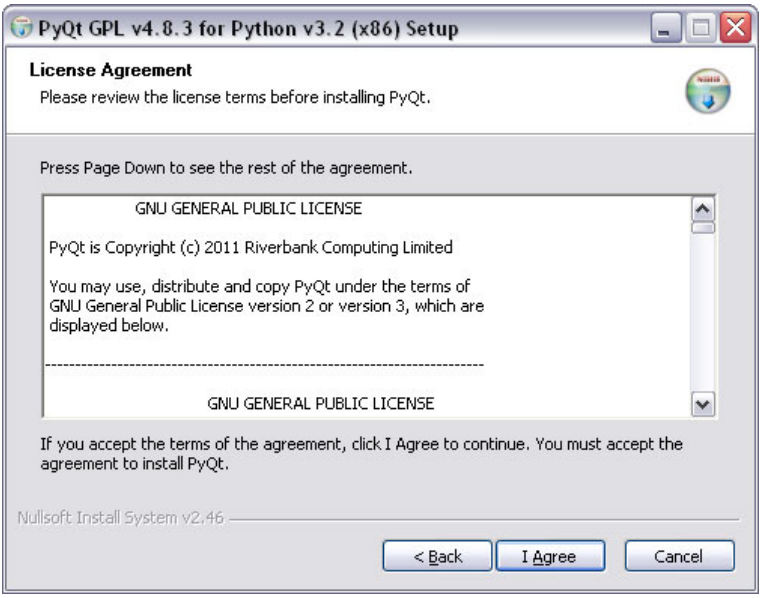

Рис. 1.2. Установка PyQt. Шаг 2

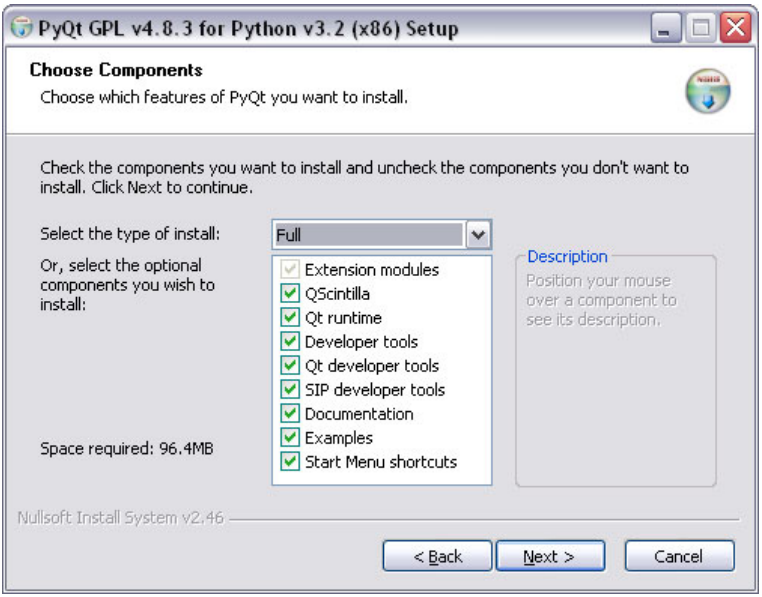

Рис. 1.3. Установка PyQt. Шаг 3

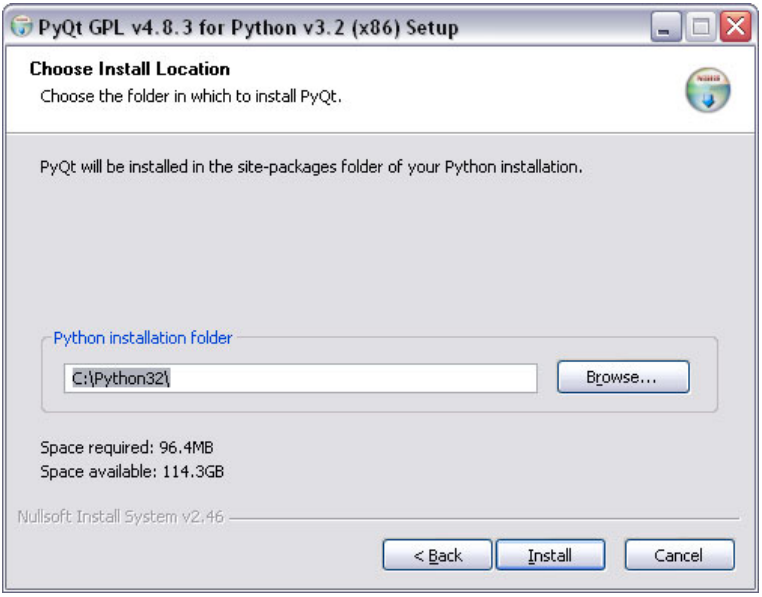

Рис. 1.4. Установка PyQt. Шаг 4

Листинги на странице http://unicross.narod.ru/pyqt/

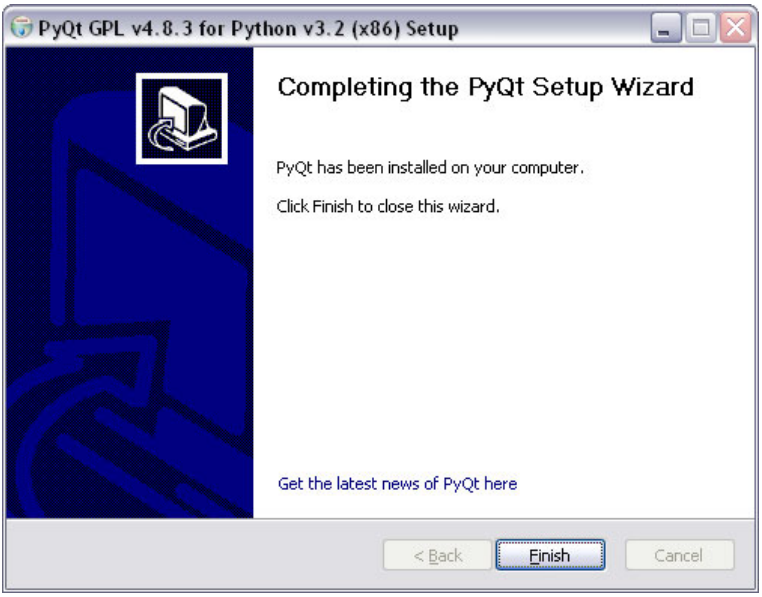

Рис. 1.5. Установка PyQt. Шаг 5

В результате установки все необходимые файлы будут скопированы в папку C:\Python32\Lib\site-packages\PyQt4\, а в начало системной переменной **PATH** добавлен путь к папке C:\Python32\Lib\site-packages\PyQt4\bin. В папке bin расположены программы Designer, Linguist и Assistant, а также библиотеки динамической компоновки (например, QtCore4.dll, QtGui4.dll), необходимые для нормального функционирования программы, написанной на PyQt. Кроме того, в папке bin находится библиотека libmySQL.dll, предназначенная для доступа к базе данных MySQL. Так как путь к папке C:\Python32\Lib\site-packages\PyQt4\bin добавляется в самое начало переменной **PATH**, библиотека libmySQL.dll будет всегда подгружаться из этой папки во всех программах. Если вы занимаетесь Web-программированием и подключаетесь к MySQL из PHP версии 5.2, то возможны проблемы с несоответствием версий библиотеки libmySQL.dll. Если проблема возникает, то следует переместить путь к папке bin, например, в самый конец переменной **PATH**.

Чтобы проверить правильность установки выведем версии PyQt и Qt (листинг 1.1).

Листинг 1.1. Проверка правильности установки PyQt

```
>>> from PyQt4 import QtCore
>>> QtCore.PYQT_VERSION_STR
'4.8.3'
```

Листинги на странице http://unicross.narod.ru/pyqt/

```
>>> QtCore.QT_VERSION_STR
'4.7.1'
```

1.2. Первая программа

При изучении языков и технологий принято начинать с программы, выводящей надпись "Привет, мир!". Не будем нарушать традицию и создадим окно с приветствием и кнопкой для закрытия окна (листинг 1.2).

Листинг 1.2. Первая программа на PyQt

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Первая программа на PyQt")
window.resize(300, 70)
label = QtGui.QLabel("<center>Привет, мир!</center>")
btnQuit = QtGui.QPushButton("&Закрыть окно")
vbox = QtGui.QVBoxLayout()
vbox.addWidget(label)
vbox.addWidget(btnQuit)
window.setLayout(vbox)
QtCore.QObject.connect(btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
window.show()
sys.exit(app.exec_())
```

Для создания файла с программой можно по-прежнему пользоваться редактором IDLE. Однако, в IDLE не работает автодополнение кода для PyQt, поэтому названия классов и методов придется набирать вручную. Многим программистам это не нравится. Кроме того, запуск оконного приложения из IDLE (нажатием клавиши <F5>) приводит к очень неприятным ошибкам и даже аварийному завершению работы редактора. Поэтому запускать оконные приложения следует двойным щелчком на значке файла.

Вместо редактора IDLE для редактирования и запуска программ на PyQt советую воспользоваться редактором Eclipse и модулем PyDev. В этом случае при

использовании точечной нотации будет автоматически выводиться список классов, методов и атрибутов. Кроме того, при выделении метода в списке можно посмотреть, какие параметры принимает метод, и что он возвращает. Запуск программы из Eclipse выполняется очень просто. Достаточно нажать кнопку на панели инструментов. Рассмотрение возможностей Eclipse выходит за рамки этой книги, поэтому изучать редактор вам придется самостоятельно.

До сих пор мы создавали файлы с расширением py и все результаты выполнения программы выводили в окно консоли. Оконное приложение также можно сохранить с расширением py, но при запуске помимо основного окна будет дополнительно выводиться окно консоли. На этапе отладки в окно консоли можно выводить отладочную информацию (этим способом мы будем пользоваться в дальнейших примерах). Чтобы избавиться от окна консоли следует сохранять файл с расширением pyw. Попробуйте создать два файла с различным расширением и запустить их с помощью двойного щелчка на значке.

1.3. Структура программы

Запускать программу мы научились, теперь рассмотрим код из листинга 1.2 построчно. В первой строке указывается кодировка файла. Так как кодировка UTF-8 является в Python 3 кодировкой модулей по умолчанию, эту строку можно и не указывать. Во второй строке подключаются модули `QtCore` и `QtGui`. Модуль `QtCore` содержит классы не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные модули PyQt. Модуль `QtGui` содержит классы, реализующие компоненты пользовательского интерфейса, например, надписи, кнопки, текстовые поля и др. В третьей строке производится подключение модуля `sys`, из которого нам потребуется список параметров из командной строки (`argv`), а также функция `exit()`, позволяющая завершить выполнение программы.

Инструкция

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

создает объект приложения с помощью класса `QApplication`. Конструктор этого класса принимает список параметров, переданных в командной строке. Следует помнить, что в программе всегда должен быть объект приложения, причем обязательно только один. Может показаться, что после создания объекта он больше нигде не используется в программе, однако с помощью этого объекта осуществляется управление приложением незаметно для нас. Получить доступ к этому объекту из любого места в программе можно через атрибут `qApp` из модуля `QtGui`. Например, вывести список параметров, переданных в командной строке, можно так:

```
print(QtGui.qApp.argv())
```

Следующая инструкция

```
window = QtGui.QWidget()
```

создает объект окна с помощью класса `QWidget`. Этот класс наследуют практически все классы, реализующие компоненты графического интерфейса. Поэтому любой компонент, не имеющий родителя, обладает своим собственным окном.

Инструкция

```
window.setWindowTitle("Первая программа на PyQt")
```

задает текст, который будет выводиться в заголовке окна. Следующая инструкция

```
window.resize(300, 70)
```

задает минимальные размеры окна. В первом параметре метода `resize()` указывается ширина окна, а во втором параметре — высота окна. Следует учитывать, что эти размеры не включают высоту заголовка окна и ширину границ, а также являются рекомендацией, т. е. если компоненты не помещаются, размеры окна будут увеличены.

Инструкция

```
label = QtGui.QLabel("<center>Привет, мир!</center>")
```

создает объект надписи. Текст надписи задается в качестве параметра в конструкторе класса `QLabel`. Обратите внимание на то, что внутри строки мы указали HTML-теги. В данном примере с помощью тега `<center>` произвели выравнивание текста по центру компонента. Возможность использования HTML-тегов и CSS-атрибутов является отличительной чертой библиотеки PyQt. Например, внутри надписи можно вывести таблицу или отобразить изображение. Это очень удобно.

Следующая инструкция

```
btnQuit = QtGui.QPushButton("&Закрыть окно")
```

создает объект кнопки. Текст, который будет отображен на кнопке, задается в качестве параметра в конструкторе класса `QPushButton`. Обратите внимание на символ `&` перед буквой "З". Таким образом задаются клавиши быстрого доступа. Если нажать одновременно клавишу <Alt> и клавишу с буквой, перед которой в строке указан символ `&`, то кнопка будет нажата.

Инструкция

```
vbox = QtGui.QVBoxLayout()
```

создает вертикальный контейнер. Все компоненты, добавляемые в этот контейнер, будут располагаться друг под другом в порядке добавления. Внутри контейнера автоматически производится подгонка размеров добавляемых компонентов под размеры контейнера. При изменении размеров контейнера будет произведено изменение размеров всех компонентов. В следующих двух инструкциях

```
vbox.addWidget(label)
vbox.addWidget(btnQuit)
```

с помощью метода `addWidget()` производится добавление объектов надписи и кнопки в вертикальный контейнер. Так как объект надписи добавляется первым, он будет расположен над кнопкой. При добавлении компонентов в контейнер, они автоматически становятся потомками контейнера.

Следующая инструкция

```
window.setLayout(vbox)
```

добавляет контейнер в основное окно с помощью метода `setLayout()`. Таким образом, контейнер становится потомком основного окна.

Инструкция

```
QtCore.QObject.connect(btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
```

назначает обработчик сигнала `clicked()`, который генерируется при нажатии кнопки. В первом параметре статического метода `connect()` указывается объект, генерирующий сигнал, а во втором параметре — название сигнала. В третьем параметре указывается объект, принимающий сигнал, а в четвертом параметре — метод этого объекта, который будет вызван при наступлении события. Этот метод принято называть *слотом*. В нашем примере получателем сигнала является объект приложения, доступный через атрибут `qApp`. При наступлении события будет вызван метод `quit()`, который завершит работу всего приложения.

Следующая инструкция

```
window.show()
```

отображает окно и все компоненты, которые мы ранее добавили. И, наконец, инструкция

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

запускает бесконечный цикл обработки событий. Инструкции, расположенные после вызова метода `exec_()`, будут выполнены только после завершения работы приложения. Так как результат выполнения метода `exec_()` мы передаем функции `exit()`, дальнейшее выполнение программы будет прекращено, а код возврата передан операционной системе.

1.4. ООП-стиль создания окна

Библиотека PyQt написана в объектно-ориентированном стиле (ООП-стиле) и содержит более 600 классов. Иерархия наследования всех классов имеет слишком большой размер, поэтому приводить ее в книге нет возможности. Тем не менее, чтобы показать зависимости, при описании компонентов иерархия наследования конкретного класса будет показываться. В качестве примера выведем базовые классы класса `QWidget`:

```
>>> from PyQt4 import QtGui, QtCore
```

```
>>> QtGui.QWidget.__bases__
(<class 'PyQt4.QtCore.QObject'>, <class 'PyQt4.QtGui.QPaintDevice'>)
```

Как видно из примера, класс `QWidget` наследует два класса — `QObject` и `QPaintDevice`. Класс `QObject` является классом верхнего уровня, его наследуют большинство классов в PyQt. В свою очередь класс `QWidget` является базовым классом для всех визуальных компонентов. Таким образом, класс `QWidget` наследует все свойства и методы базовых классов. Это обстоятельство следует учитывать при изучении документации, так как в ней описываются атрибуты и методы конкретного класса, а на унаследованные атрибуты и методы даются только ссылки.

В своих программах вы можете наследовать стандартные классы и добавлять новую функциональность. В качестве примера переделаем код из листинга 1.2 и создадим окно в ООП-стиле (листинг 1.3).

Листинг 1.3. ООП-стиль создания окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.label = QtGui.QLabel("Привет, мир!")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.btnQuit = QtGui.QPushButton("&Закрыть окно")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnQuit)
        self.setLayout(self.vbox)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()   # Создаем экземпляр класса
    window.setWindowTitle("ООП-стиль создания окна")
    window.resize(300, 70)
```

```
    window.show()          # Отображаем окно
    sys.exit(app.exec_()) # Запускаем цикл обработки событий
```

В первых двух строках как обычно указывается кодировка файла и подключаются модули `QtCore` и `QtGui`. Далее мы определяем класс `MyWindow`, который наследует класс `QWidget`:

```
class MyWindow(QtGui.QWidget):
```

Помимо класса `QWidget` можно наследовать и другие классы, которые являются наследниками класса `QWidget`, например, класс `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При наследовании класса `QDialog` окно будет выравниваться по центру экрана и иметь только две кнопки в заголовке окна — **Справка** и **Закрыть**. Кроме того, можно наследовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Наследование класса `QMainWindow` имеет свои отличия, которые мы будем рассматривать в отдельной главе.

Следующая инструкция

```
def __init__(self, parent=None):
```

создает конструктор класса. В качестве параметров конструктор принимает ссылку на экземпляр класса (`self`) и ссылку на родительский элемент (`parent`). Родительский элемент может отсутствовать, поэтому в определении конструктора параметру присваивается значение по умолчанию (`None`). Внутри метода `__init__()` вызывается конструктор базового класса и ему передается ссылка на родительский компонент:

```
QtGui.QWidget.__init__(self, parent)
```

Следующие инструкции внутри конструктора создают объекты надписи, кнопки и контейнера, затем добавляют компоненты в контейнер, а сам контейнер в основное окно. Следует обратить внимание на то, что объекты надписи и кнопки сохраняется в атрибутах экземпляра класса. В дальнейшем из методов класса можно управлять этими объектами, например, изменить текст надписи. Если объекты не сохранить, то получить к ним доступ будет не так просто.

Далее производится назначение обработчика нажатия кнопки:

```
self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
             QtGui.qApp.quit)
```

В отличие от предыдущего примера (листинг 1.2), метод `connect()` вызывается не через класс `QObject`, а через экземпляр нашего класса. Это возможно, так как мы наследовали класс `QWidget`, который в свою очередь является наследником класса `QObject` и наследует метод `connect()`. Еще одно отличие состоит в количестве параметров, которые принимает метод `connect()`. В первом и втором параметре как обычно указывается объект, генерирующий сигнал, и название обрабатываемого сигнала. В третьем параметре указывается ссылка на метод, который будет вызван при наступлении события. Это возможно, так как в языке Python функция является

обычным объектом, получить ссылку на который можно указав название функции без круглых скобок.

В предыдущем примере (листинг 1.2) мы выравнивали надпись с помощью HTML-разметки. Произвести аналогичную операцию позволяет также метод `setAlignment()`, которому следует передать значение атрибута `AlignCenter`:

```
self.label.setAlignment(QtCore.Qt.AlignCenter)
```

Создание объекта приложения и объекта класса `MyWindow` производится внутри условия:

```
if __name__ == "__main__":
```

Атрибут модуля `__name__` будет содержать значение `"__main__"` только в случае запуска модуля как главной программы. Если модуль импортировать, то атрибут будет содержать другое значение. Поэтому весь последующий код создания объекта приложения и объекта окна выполняется только при запуске программы с помощью двойного щелчка на значке файла. Может возникнуть вопрос, зачем это нужно? Одним из преимуществ ООП-стиля программирования является повторное использования кода. Следовательно, можно импортировать модуль и использовать класс `MyWindow` в другом приложении. Рассмотрим эту возможность на примере. Сохраняем код из листинга 1.3 в файле с названием MyWindow.py, а затем создаем в той же папке еще один файл (например, с названием test.pyw) и вставляем в него код из листинга 1.4.

Листинг 1.4. Повторное использование кода при ООП-стиле

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui, QtCore
import MyWindow

class MyDialog(QtGui.QDialog):
    def __init__(self, parent=None):
        QtGui.QDialog.__init__(self, parent)
        self.myWidget = MyWindow.MyWindow()
        self.myWidget.vbox.setMargin(0)
        self.button = QtGui.QPushButton("&Изменить надпись")
        mainBox = QtGui.QVBoxLayout()
        mainBox.addWidget(self.myWidget)
        mainBox.addWidget(self.button)
        self.setLayout(mainBox)
        self.connect(self.button, QtCore.SIGNAL("clicked()"),
```

```
                        self.on_clicked)

    def on_clicked(self):
        self.myWidget.label.setText("Новая надпись")
        self.button.setDisabled(True)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyDialog()
    window.setWindowTitle("Преимущество ООП-стиля")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

Теперь запустим файл test.pyw с помощью двойного щелчка на значке файла. В результате будет отображено окно с надписью и двумя кнопками. При нажатии кнопки **Изменить надпись** производится изменение текста надписи и кнопка деактивизируется. Нажатие кнопки **Закрыть окно** будет по-прежнему выполнять завершение выполнения приложения.

В этом примере мы создали класс `MyDialog`, который наследует класс `QDialog`. Поэтому при выводе окно автоматически выравнивается по центру экрана. Кроме того, в заголовке окна выводятся только две кнопки — **Справка** и **Закрыть**. Внутри конструктора мы создаем экземпляр класса `MyWindow` и сохраняем его в атрибуте `myWidget`:

```
self.myWidget = MyWindow.MyWindow()
```

С помощью этого атрибута можно получить доступ ко всем атрибутам класса `MyWindow`. Например, в следующей строке производится изменение отступа:

```
self.myWidget.vbox.setMargin(0)
```

В следующих инструкциях внутри конструктора производится создание объекта кнопки и контейнера, затем экземпляр класса `MyWindow` и объект кнопки добавляются в контейнер, а далее контейнер добавляется в основное окно.

Инструкция

```
self.connect(self.button, QtCore.SIGNAL("clicked()"),
             self.on_clicked)
```

назначает обработчик нажатия кнопки. В третьем параметре указывается ссылка на метод `on_clicked()` внутри которого производится изменение текста надписи (с помощью метода `setText()`) и деактивизация кнопки (с помощью метода

`setDisabled()`). Внутри метода `on_clicked()` доступен указатель `self`, через который можно получить доступ как к атрибутам класса `MyDialog`, так и к атрибутам класса `MyWindow`.

Таким образом, производится повторное использование ранее написанного кода. Мы создаем класс и сохраняем его внутри отдельного модуля. Чтобы протестировать модуль или использовать его как отдельное приложение, размещаем код создания объекта приложения и объекта окна внутри условия:

```
if __name__ == "__main__":
```

При запуске с помощью двойного щелка на значке файла производится выполнение кода как отдельного приложения. Если модуль импортируется, то создание объекта приложения не производится и мы можем использовать класс в других приложениях. Например, так как это было сделано в листинге 1.4 или путем наследования класса и добавления или переопределения методов.

В некоторых случаях использование ООП-стиля является обязательным. Например, чтобы обработать нажатие клавиши на клавиатуре, необходимо наследовать какой-либо класс и переопределить в нем метод с предопределенным названием. Какие методы необходимо переопределять мы рассмотрим при изучении обработки событий.

1.5. Создание окна с помощью программы Qt Designer

Если вы ранее пользовались Visual Studio или Delphi, то вспомните, что размещение компонентов на форме производили с помощью мыши. Щелкали левой кнопкой мыши на соответствующей кнопке на панели инструментов и перетаскивали компонент на форму. Далее с помощью инспектора свойств производили настройку значений некоторых свойств, а остальные свойства получали значения по умолчанию. При этом весь код генерировался автоматически. Произвести аналогичную операцию в PyQt позволяет программа Qt Designer, которая входит в состав установленных компонентов.

1.5.1. Создание формы

Для запуска программы Qt Designer в меню **Пуск** выбираем пункт **Программы | PyQt GPL v4.8.3 for Python v3.2 (x86) | Designer**. В окне **Новая форма** выделяем пункт **Widget** и нажимаем кнопку **Создать**. В результате откроется окно с пустой формой, на которую можно перетаскивать компоненты с панели **Панель виджетов** с помощью мыши.

В качестве примера добавим на форму надпись и кнопку. Для этого на панели **Панель виджетов** в группе **Display Widgets** щелкаем левой кнопкой мыши на пункте **Label** и, не отпуская кнопку мыши, перетаскиваем компонент на форму. Далее проделываем аналогичную операцию с компонентом **Push Button** и размещаем его ниже надписи.

Теперь выделяем одновременно надпись и кнопку, а затем щелкаем правой кнопкой мыши над любым компонентом и из контекстного меню выбираем пункт **Компоновка | Скомпоновать по вертикали**. Чтобы компоненты занимали всю область формы, щелкаем правой кнопкой мыши на свободном месте формы и из контекстного меню выбираем пункт **Компоновка | Скомпоновать по горизонтали**.

Теперь изменим некоторые свойства окна. Для этого в окне **Инспектор объектов** выделяем первый пункт и переходим в окно **Редактор свойств**. Находим свойство **objectName** и справа от свойства вводим значение `MyForm`. Далее находим свойство **geometry** и щелкаем мышью на плюсике слева, чтобы отобразить скрытые свойства. Задаем ширину равной `300`, а высоту равной `70`. Размеры формы автоматически изменятся. Указать текст, который будет отображаться в заголовке окна, позволяет свойство **windowTitle**.

Чтобы изменить свойства надписи следует вначале выделить компонент с помощью мыши или выделить соответствующий пункт в окне **Инспектор объектов**. Изменяем значение свойства **objectName** на `label`, в свойстве **text** указываем текст надписи, а в свойстве **alignment** задаем значение `AlignHCenter`. Теперь выделяем кнопку и изменяем значение свойства **objectName** на `btnQuit`, а в свойстве **text** указываем текст надписи. Изменить текст надписи можно также дважды щелкнув мышью на компоненте.

После окончания настройки формы и компонентов сохраняем форму в файл. Для этого в меню **Файл** выбираем пункт **Сохранить** и сохраняем файл под названием MyForm.ui. При необходимости внести какие-либо изменения этот файл можно открыть в программе Qt Designer, выбрав в меню **Файл** пункт **Открыть**.

1.5.2. Загрузка ui-файла в программе

Как вы можете убедиться, внутри ui-файла содержится текст в XML-формате, а не программа на языке Python. Следовательно, подключить файл с помощью инструкции `import` не получится. Чтобы использовать ui-файл внутри программы следует воспользоваться модулем `uic`, который входит в состав библиотеки PyQt. Прежде чем использовать функции из этого модуля необходимо подключить модуль с помощью инструкции:

```
from PyQt4 import uic
```

Для загрузки ui-файла предназначена функция `loadUi()`. Формат функции:

```
loadUi(<ui-файл>[, <Экземпляр класса>])
```

Если второй параметр не указан, то функция возвращает ссылку на объект формы. С помощью этой ссылки можно получить доступ к компонентам формы и, например, назначить обработчики сигналов (листинг 1.5). Названия компонентов задаются в программе Qt Designer в свойстве **objectName**.

Листинг 1.5. Использование функции loadUi(). Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui, uic
import sys
app = QtGui.QApplication(sys.argv)
window = uic.loadUi("MyForm.ui")
QtCore.QObject.connect(window.btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp.quit)
window.show()
sys.exit(app.exec_())
```

Если во втором параметре указать ссылку на экземпляр класса, то все компоненты формы будут доступны через указатель `self` (листинг 1.6).

Листинг 1.6. Использование функции loadUi(). Вариант 2

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui, uic

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        uic.loadUi("MyForm.ui", self)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Загрузить ui-файл позволяет также функция `loadUiType()`. Функция возвращает кортеж из двух элементов: ссылки на класс формы и ссылки на базовый класс. Так как функция возвращает ссылку на класс, а не на экземпляр класса, мы можем создать

множество экземпляров класса. После создания экземпляра класса формы необходимо вызвать метод `setupUi()` и передать ему указатель `self` (листинг 1.7).

Листинг 1.7. Использование функции loadUiType(). Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui, uic

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        Form, Base = uic.loadUiType("MyForm.ui")
        self.ui = Form()
        self.ui.setupUi(self)
        self.connect(self.ui.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Загрузить ui-файл можно вне класса, а затем указать класс формы во втором параметре в списке наследования. В этом случае наш класс наследует все методы класса формы. Для примера изменим определение класса `MyWindow` из предыдущего примера (листинг 1.8).

Листинг 1.8. Использование функции loadUiType(). Вариант 2

```
Form, Base = uic.loadUiType("MyForm.ui")
class MyWindow(QtGui.QWidget, Form):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)
```

1.5.3. Преобразование ui-файла в py-файл

Вместо подключения ui-файла можно автоматически сгенерировать код на языке Python. Для преобразования ui-файла в py-файл предназначена утилита pyuic4.bat, расположенная в каталоге C:\Python32\Lib\site-packages\PyQt4\bin. Запускаем командную строку и переходим в каталог, в котором находится ui-файл. Для генерации программы выполняем команду:

```
pyuic4.bat MyForm.ui -o ui_MyForm.py
```

В результате будет создан файл ui_MyForm.py, который мы можем подключить с помощью инструкции `import`. Внутри файла находится класс `Ui_MyForm` с двумя методами: `setupUi()` и `retranslateUi()`. При использовании процедурного стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему ссылку на экземпляр окна (листинг 1.9).

Листинг 1.9. Использование класса формы. Вариант 1

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys, ui_MyForm

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
ui = ui_MyForm.Ui_MyForm()
ui.setupUi(window)
QtCore.QObject.connect(ui.btnQuit, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp.quit)
window.show()
sys.exit(app.exec_())
```

При использовании ООП-стиля программирования следует создать экземпляр класса формы, а затем вызвать метод `setupUi()` и передать ему указатель `self` (листинг 1.10).

Листинг 1.10. Использование класса формы. Вариант 2

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import ui_MyForm
```

```
class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.ui = ui_MyForm.Ui_MyForm()
        self.ui.setupUi(self)
        self.connect(self.ui.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Класс формы можно указать во втором параметре в списке наследования. В этом случае наш класс наследует все методы класса формы. Для примера изменим определение класса `MyWindow` из предыдущего примера (листинг 1.11).

Листинг 1.11. Использование класса формы. Вариант 3

```
class MyWindow(QtGui.QWidget, ui_MyForm.Ui_MyForm):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setupUi(self)
        self.connect(self.btnQuit, QtCore.SIGNAL("clicked()"),
                     QtGui.qApp.quit)
```

Как видите и в PyQt можно создавать формы, размещать компоненты с помощью мыши, а затем непосредственно подключать ui-файл в программе или автоматически генерировать код с помощью утилиты pyuic4.bat. Все это очень удобно. Тем не менее, чтобы полностью владеть технологией необходимо уметь создавать код вручную. Поэтому в оставшейся части книги мы больше не будем рассматривать программу Qt Designer. Научиться "мышкотворчеству" вы сможете и самостоятельно без моей помощи.

1.6. Модули PyQt

В состав библиотеки PyQt входят следующие модули, объединенные в пакет `PyQt4`:

- `QtCore` — содержит классы не связанные с реализацией графического интерфейса. От этого модуля зависят все остальные модули;
- `QtGui` — содержит классы, реализующие компоненты пользовательского интерфейса, например, надписи, кнопки, текстовые поля и др.;
- `QtSql` — включает поддержку баз данных SQLite, MySQL и др.;
- `QtSvg` — позволяет работать с векторной графикой (SVG);
- `QtOpenGL` — обеспечивает поддержку OpenGL;
- `QtNetwork` — содержит классы, предназначенные для работы с сетью;
- `QtWebKit` — предназначен для отображения HTML-документов;
- `phonon` и `QtMultimedia` — содержат поддержку мультимедиа;
- `QtXml` и `QtXmlPatterns` — предназначены для обработки XML;
- `QtScript` и `QtScriptTools` — содержат поддержку языка сценариев Qt Scripts;
- `Qt` — включает классы из всех модулей сразу.

Помимо перечисленных модулей в состав пакета `PyQt4` входят также модули `QtDeclarative`, `QAxContainer`, `QtTest`, `QtHelp` и `QtDesigner`. Для подключения модулей используется следующий синтаксис:

```
from PyQt4 import <Названия модулей через запятую>
```

Пример подключения модулей `QtCore` и `QtGui`:

```
from PyQt4 import QtCore, QtGui
```

В этой книге мы будем рассматривать только модули `QtCore` и `QtGui`. Чтобы получить информацию по другим модулям обращайтесь к документации.

1.7. Типы данных в PyQt

Библиотека PyQt является надстройкой над библиотекой Qt, написанной на языке C++. Библиотека Qt содержит множество классов, которые расширяют стандартные типы данных языка C++. Они реализуют динамические массивы, ассоциативные массивы, множества и др. Все эти классы очень помогают при программировании на языке C++, но для языка Python они не представляют особого интереса, так как весь этот функционал содержат стандартные типы данных. Тем не менее, при чтении документации вы столкнетесь с этими типами, поэтому сейчас мы кратко рассмотрим основные типы:

➔ `QString` — Unicode-строка. В API версии 2 (используется по умолчанию) автоматически преобразуется в тип `str` и обратно. Класс доступен только если используется API версии 1. Чтобы указать версию следует в самом начале программы импортировать модуль `sip`, а затем вызвать функцию `setapi()`:

```
>>> import sip
>>> sip.setapi('QString', 1)
>>> from PyQt4 import QtCore
>>> s = QtCore.QString("строка")
>>> s
PyQt4.QtCore.QString('строка')
>>> str(s)
'строка'
```

➔ `QChar` — Unicode-символ. В API версии 2 (используется по умолчанию) автоматически преобразуется в тип `str`. Класс доступен только если используется API версии 1. Пример:

```
>>> import sip
>>> sip.setapi('QString', 1)
>>> from PyQt4 import QtCore
>>> QtCore.QChar("a")
PyQt4.QtCore.QChar(0x0061)
```

➔ `QStringList` — массив Unicode-строк. В API версии 2 (используется по умолчанию) автоматически преобразуется в список. Класс доступен только если используется API версии 1. Пример:

```
>>> import sip
>>> sip.setapi('QString', 1)
>>> from PyQt4 import QtCore
>>> L = QtCore.QStringList(["s1", "s2"])
>>> L
<PyQt4.QtCore.QStringList object at 0x00FD8618>
>>> list(L)
[PyQt4.QtCore.QString('s1'), PyQt4.QtCore.QString('s2')]
>>> [str(s) for s in list(L)]
['s1', 's2']
```

➔ `QByteArray` — массив байтов. Преобразуется в тип `bytes`. Пример:

```
>>> arr = QtCore.QByteArray(bytes("str", "cp1251"))
```

```
>>> arr
PyQt4.QtCore.QByteArray(b'str')
>>> bytes(arr)
b'str'
```

➔ `QVariant` — может хранить данные любого типа. Функциональность класса зависит от версии API. Если явно указана версия 1, то конструктор класса может принимать данные любого типа. Чтобы преобразовать данные, хранящиеся в экземпляре класса `QVariant`, в тип данных Python, нужно вызвать метод `toPyObject()`. Чтобы создать пустой объект, следует после названия класса указать пустые скобки. Пример:

```
>>> import sip
>>> sip.setapi('QVariant', 1)
>>> from PyQt4 import QtCore
>>> n = QtCore.QVariant(10)
>>> n
<PyQt4.QtCore.QVariant object at 0x00FD8D50>
>>> n.toPyObject()
10
>>> QtCore.QVariant() # Пустой объект
<PyQt4.QtCore.QVariant object at 0x00FD8420>
```

В API версии 2 (версия используется по умолчанию) все преобразования производятся автоматически. Обратите внимание на то, что создать экземпляр класса `QVariant` в API версии 2 нельзя. Попытка создания экземпляра класса приведет к исключению `TypeError`:

```
>>> from PyQt4 import QtCore
>>> QtCore.QVariant(10) # Создать экземпляр класса нельзя
... Фрагмент опущен ...
TypeError: PyQt4.QtCore.QVariant represents a mapped type
and cannot be instantiated
```

Если метод ожидает данные типа `QVariant`, то можно передать данные любого типа. При получении данных производится автоматическое преобразование объекта в тип, поддерживаемый языком Python. Чтобы создать пустой объект, следует воспользоваться классом `QPyNullVariant`. Конструктор класса принимает название типа из библиотеки Qt в виде строки (например, `"QString"`) или тип данных в языке Python (например, `str`). Класс содержит метод `isNull()`, который всегда возвращает значение `True`. Получить название типа данных в виде строки позволяет метод `typeName()`. Пример:

```
>>> from PyQt4 import QtCore
>>> nullVariant = QtCore.QPyNullVariant(int)
>>> nullVariant.isNull(), nullVariant.typeName()
(True, 'int')
>>> nullVariant = QtCore.QPyNullVariant("QString")
>>> nullVariant.isNull(), nullVariant.typeName()
(True, 'QString')
```

От версии API зависят также классы `QDate` (представление даты), `QTime` (представление времени), `QDateTime` (представление даты и времени), `QTextStream` (текстовый поток) и `QUrl`.

1.8. Управление основным циклом приложения

Для взаимодействия с системой и запуска обработчиков сигналов предназначен основной цикл приложения. После вызова метода `exec_()` программа переходит в бесконечный цикл. Инструкции, расположенные после вызова этого метода, будут выполнены только после завершения работы всего приложения. Чтобы завершить работу приложения необходимо вызвать слот `quit()` или метод `exit(returnCode=0)`. Так как программа находится внутри цикла, то вызвать эти методы можно только при наступлении какого-либо события, например, при нажатии пользователем кнопки. Цикл автоматически прерывается при нажатии пользователем кнопки **Закрыть** в заголовке окна.

После наступления события цикл прерывается и управление передается в обработчик. После завершения работы обработчика управление опять передается основному циклу приложения. Если внутри обработчика выполняется длительная операция, то программа перестает реагировать на события. В качестве примера изобразим длительный процесс с помощью функции `sleep()` из модуля `time` (листинг 1.12).

Листинг 1.12. Выполнение длительной операции

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys, time

def on_clicked():
    time.sleep(10) # "Засыпаем" на 10 секунд

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

```
button = QtGui.QPushButton("Запустить процесс")
button.resize(200, 40)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере при нажатии кнопки **Запустить процесс** вызывается функция `on_clicked()`, внутри которой мы "засыпаем" на десять секунд и тем самым прерываем основной цикл приложения. Попробуйте нажать кнопку, перекрыть окно другим окном, а затем заново его отобразить. В результате окно перестает реагировать на любые события и после перекрытия другим окном приложение не может выполнить перерисовку окна, пока не закончится выполнение процесса. Если в этот момент нажать кнопку **Закрыть** в заголовке окна, то будет выведено окно **Завершение программы**, с помощью которого система сообщает, что программа не отвечает и предлагает завершить ее.

При выполнении длительной операции ее можно разбить на несколько этапов и периодически запускать оборот основного цикла с помощью метода `processEvents()` объекта приложения. Переделаем предыдущую программу и с помощью цикла инсценируем длительную операцию, которая выполняется 20 секунд (листинг 1.13).

Листинг 1.13. Использование метода processEvents()

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys, time

def on_clicked():
    global button
    button.setDisabled(True)         # Делаем кнопку неактивной
    for i in range(1, 21):
        QtGui.qApp.processEvents()   # Запускаем оборот цикла
        time.sleep(1)                # "Засыпаем" на 1 секунду
        print("step -", i)
    button.setDisabled(False)        # Делаем кнопку активной

app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Запустить процесс")
```

```
button.resize(200, 40)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       on_clicked)
button.show()
sys.exit(app.exec_())
```

В этом примере длительная операция разбита на одинаковые этапы. После выполнения каждого этапа запускается оборот основного цикла приложения. Теперь при перекрытии окна и повторном его отображении окно будет перерисовано. Таким образом, приложение по-прежнему будет взаимодействовать с системой, хотя и с некоторой задержкой.

1.9. Многопоточные приложения

При обработке больших объемов данных не всегда можно равномерно разбить операцию на небольшие по времени этапы. Поэтому при использовании метода `processEvents()` возможны проблемы. Вместо использования метода `processEvents()` лучше вынести длительную операцию в отдельный поток. В этом случае длительная операция будет выполняться параллельно с основным циклом приложения и не будет его блокировать. В одном процессе можно запустить сразу несколько независимых потоков. Если процессор является многоядерным, то потоки будут равномерно распределены по ядрам. За счет этого можно не только избежать блокировки GUI-потока, но и значительно увеличить эффективность выполнения программы. Завершение основного цикла приложения приводит к завершению всех потоков.

1.9.1. Класс QThread. Создание потока

Для создания потока в PyQt предназначен класс `QThread`, который наследует класс `QObject`. Конструктор класса `QThread` имеет следующий формат:

```
QThread([parent=None])
```

Чтобы использовать потоки следует создать класс, который будет наследником класса `QThread`, и определить в нем метод `run()`. Код, расположенный в методе `run()`, будет выполняться в отдельном потоке. После завершения выполнения метода `run()` поток прекратит свое существование. Далее нужно создать экземпляр класса и вызвать метод `start()`, который после запуска потока вызовет метод `run()`. Обратите внимание на то, что если напрямую вызвать метод `run()`, то код будет выполняться в GUI-потоке, а не в отдельном потоке. Поэтому чтобы запустить поток необходимо вызвать именно метод `start()`, а не метод `run()`. Метод `start()` имеет следующий формат:

```
start([priority = QThread.InheritPriority])
```

Параметр `priority` задает приоритет выполнения потока по отношению к другим потокам. Следует учитывать, что при наличии потока с самым высоким приоритетом, поток с самым низким приоритетом может быть просто проигнорирован в некоторых операционных системах. Перечислим допустимые значения параметра (в порядке увеличения приоритета) и соответствующие им атрибуты из класса `QThread`:

- 0 — `QThread.IdlePriority` — самый низкий приоритет;
- 1 — `QThread.LowestPriority`;
- 2 — `QThread.LowPriority`;
- 3 — `QThread.NormalPriority`;
- 4 — `QThread.HighPriority`;
- 5 — `QThread.HighestPriority`;
- 6 — `QThread.TimeCriticalPriority` — самый высокий приоритет;
- 7 — `QThread.InheritPriority` — автоматический выбор приоритета (значение используется по умолчанию).

Задать приоритет потока позволяет также метод `setPriority(<Приоритет>)`. Узнать какой приоритет использует запущенный поток можно с помощью метода `priority()`.

После запуска потока генерируется сигнал `started()`, а после завершения — сигнал `finished()`. Назначив обработчики этим сигналам можно контролировать статус потока из основного цикла приложения. Если необходимо узнать текущий статус, то следует воспользоваться методами `isRunning()` и `isFinished()`. Метод `isRunning()` возвращает значение `True`, если поток выполняется, а метод `isFinished()` возвращает значение `True`, если поток закончил выполнение.

Потоки выполняются внутри одного процесса и имеют доступ ко всем глобальным переменным. Однако следует учитывать, что из потока нельзя изменять что-либо в GUI-потоке, например, выводить текст на надпись. Для изменения данных в GUI-потоке нужно использовать сигналы. Внутри потока с помощью метода `emit()` производим генерацию сигнала и передаем данные. В первом параметре метод принимает объект сигнала, а в остальных параметрах данные, которые будут переданы. Внутри основного потока назначаем обработчик этого сигнала и в обработчике изменяем что-либо в GUI-потоке.

Рассмотрим использование класса `QThread` на примере (листинг 1.14).

Листинг 1.14. Использование класса QThread

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
```

```
class MyThread(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        for i in range(1, 21):
            self.sleep(3)              # "Засыпаем" на 3 секунды
            # Передача данных из потока через сигнал
            self.emit(QtCore.SIGNAL("mysignal(QString)"), "i = %s" % i)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.label = QtGui.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button = QtGui.QPushButton("Запустить процесс")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.thread = MyThread()       # Создаем экземпляр класса
        self.connect(self.button, QtCore.SIGNAL("clicked()"),
                     self.on_clicked)
        self.connect(self.thread, QtCore.SIGNAL("started()"),
                     self.on_started)
        self.connect(self.thread, QtCore.SIGNAL("finished()"),
                     self.on_finished)
        self.connect(self.thread, QtCore.SIGNAL("mysignal(QString)"),
                     self.on_change, QtCore.Qt.QueuedConnection)
    def on_clicked(self):
        self.button.setDisabled(True)  # Делаем кнопку неактивной
        self.thread.start()            # Запускаем поток
    def on_started(self):               # Вызывается при запуске потока
        self.label.setText("Вызван метод on_started()")
    def on_finished(self):          # Вызывается при завершении потока
```

```
        self.label.setText("Вызван метод on_finished()")
        self.button.setDisabled(False) # Делаем кнопку активной
    def on_change(self, s):
        self.label.setText(s)


if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QThread")
    window.resize(300, 70)
    window.show()
    sys.exit(app.exec_())
```

Итак, мы создали класс `MyThread`, который является наследником класса `QThread`, и определили метод `run()`. Внутри метода `run()` производится имитация процесса с помощью цикла `for` и метода `sleep()`. Каждые три секунды выполняется генерация сигнала `mysignal(QString)` и передача текущего значения переменной `i`. Внутри конструктора класса `MyWindow` мы назначили обработчик этого сигнала с помощью инструкции:

```
self.connect(self.thread, QtCore.SIGNAL("mysignal(QString)"),
             self.on_change, QtCore.Qt.QueuedConnection)
```

В первом параметре указана ссылка на экземпляр класса `MyThread` (экземпляр создается в конструкторе класса `MyWindow` и сохраняется в атрибуте `thread`), а во втором параметре — объект сигнала. Обратите внимание на то, что название сигнала в методе `connect()` полностью совпадает с названием сигнала в методе `emit()`. В третьем параметре указана ссылка на метод, который будет вызван при наступлении события. Этот метод принимает один параметр и производит изменение текста надписи с помощью метода `setText()`. В четвертом параметре метода `connect()` с помощью значения `QtCore.Qt.QueuedConnection` указывается, что сигнал помещается в очередь обработки событий и обработчик должен выполняться в потоке приемника сигнала, т. е. в GUI-потоке. Из GUI-потока мы можем смело изменять свойства компонентов интерфейса.

Внутри конструктора класса `MyWindow` производится создание надписи и кнопки, а затем размещение их внутри вертикального контейнера. Далее выполняется создание экземпляра класса `MyThread` и сохранение его в атрибуте `thread`. С помощью этого атрибута мы можем управлять потоком и назначить обработчики сигналов `started()`, `finished()` и `mysignal(QString)`. Запуск потока производится с помощью метода `start()` внутри обработчика нажатия кнопки. Чтобы исключить повторный запуск

потока мы деактивизируем кнопку с помощью метода `setDisabled()`. После окончания работы потока внутри обработчика сигнала `finished()` делаем кнопку опять доступной.

Обратите внимание на то, что для имитации длительного процесса мы использовали статический метод `sleep()` из класса `QThread`, а не функцию `sleep()` из модуля `time`. Временно прервать выполнение потока позволяют следующие статические методы из класса `QThread`:

- `sleep()` — продолжительность задается в секундах:

```
QtCore.QThread.sleep(3)          # "Засыпаем" на 3 секунды
```

- `msleep()` — продолжительность задается в миллисекундах:

```
QtCore.QThread.msleep(3000)      # "Засыпаем" на 3 секунды
```

- `usleep()` — продолжительность задается в микросекундах:

```
QtCore.QThread.usleep(3000000) # "Засыпаем" на 3 секунды
```

1.9.2. Управление циклом внутри потока

Очень часто внутри потока одни и те же инструкции выполняются многократно. Например, при осуществлении мониторинга серверов в Интернете на каждой итерации цикла посылается запрос к одному серверу. В этом случае используется бесконечный цикл внутри метода `run()`. Выход из цикла производится после окончания опроса всех серверов. В некоторых случаях этот цикл необходимо прервать преждевременно при нажатии кнопки пользователем. Чтобы это стало возможным, в классе, реализующем поток, следует создать атрибут, который будет содержать флаг текущего состояния. Далее на каждой итерации цикла проверяем состояние флага. При изменении состояния прерываем выполнение цикла. Чтобы изменить значение атрибута создаем обработчик и связываем его с сигналом `clicked()` соответствующей кнопки. При нажатии кнопки внутри обработчика производим изменение значения атрибута. Пример запуска и остановки потока с помощью кнопок приведен в листинге 1.15.

Листинг 1.15. Запуск и остановка потока

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui


class MyThread(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.running = False   # Флаг выполнения
```

```
        self.count = 0
    def run(self):
        self.running = True
        while self.running:    # Проверяем значение флага
            self.count += 1
            self.emit(QtCore.SIGNAL("mysignal(QString)"),
                      "count = %s" % self.count)
            self.sleep(1)      # Имитируем процесс

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.label = QtGui.QLabel("Нажмите кнопку для запуска потока")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.btnStart = QtGui.QPushButton("Запустить поток")
        self.btnStop = QtGui.QPushButton("Остановить поток")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.btnStart)
        self.vbox.addWidget(self.btnStop)
        self.setLayout(self.vbox)
        self.thread = MyThread()
        self.connect(self.btnStart, QtCore.SIGNAL("clicked()"),
                     self.on_start)
        self.connect(self.btnStop, QtCore.SIGNAL("clicked()"),
                     self.on_stop)
        self.connect(self.thread, QtCore.SIGNAL("mysignal(QString)"),
                     self.on_change, QtCore.Qt.QueuedConnection)
    def on_start(self):
        if not self.thread.isRunning():
            self.thread.start()        # Запускаем поток
    def on_stop(self):
        self.thread.running = False    # Изменяем флаг выполнения
    def on_change(self, s):
        self.label.setText(s)
```

```
    def closeEvent(self, event):        # Вызывается при закрытии окна
        self.hide()                     # Скрываем окно
        self.thread.running = False     # Изменяем флаг выполнения
        self.thread.wait(5000)          # Даем время чтобы закончить
        event.accept()                  # Закрываем окно


if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Запуск и остановка потока")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

В этом примере внутри конструктора класса `MyThread` создается атрибут `running` и ему присваивается значение `False`. При запуске потока внутри метода `run()` значение атрибута изменяется на `True`. Затем запускается цикл, в котором атрибут указывается в качестве условия. Как только значение атрибута станет равным значению `False`, цикл будет остановлен.

Внутри конструктора класса `MyWindow` производится создание надписи, двух кнопок и экземпляра класса `MyThread`. Далее назначаются обработчики сигналов. При нажатии кнопки **Запустить поток** запустится метод `on_start()`, внутри которого с помощью метода `isRunning()` производится проверка текущего статуса потока. Если поток не запущен, то выполняется запуск потока с помощью метода `start()`. При нажатии кнопки **Остановить поток** запустится метод `on_stop()`, внутри которого атрибуту `running` присваивается значение `False`. Это значение является условием выхода из цикла внутри метода `run()`.

Путем изменения значения атрибута можно прервать выполнение цикла только в том случае, если закончится выполнение очередной итерации. Если поток длительное время ожидает какое-либо событие, например, ответ сервера, то можно так и не дождаться завершения потока. Чтобы прервать выполнение потока в любом случае, следует воспользоваться методом `terminate()`. Этим методом можно пользоваться только в крайнем случае, так как прерывание производится в любой части кода. При этом блокировки автоматически не снимаются. Кроме того, можно повредить данные, над которыми производились операции в момент прерывания. После вызова метода `terminate()` следует вызвать метод `wait()`.

При закрытии окна, приложение завершает работу. Завершение основного цикла приложения приводит к завершению всех потоков. Чтобы предотвратить повреждение

данных мы перехватываем событие закрытия окна и дожидаемся окончания выполнения. Чтобы перехватить событие необходимо внутри класса создать метод с предопределенным названием, в нашем случае с названием `closeEvent()`. Этот метод будет автоматически вызван при попытке закрыть окно. В качестве параметра метод принимает объект события `event`. Через этот объект можно получить дополнительную информацию о событии. Чтобы закрыть окно внутри метода `closeEvent()` следует вызвать метод `accept()` объекта события. Если необходимо предотвратить закрытие окна, то следует вызвать метод `ignore()`.

Внутри метода `closeEvent()` мы присваиваем атрибуту `running` значение `False`. Далее с помощью метода `wait()` даем возможность потоку нормально завершить работу. В качестве параметра метод `wait()` принимает количество миллисекунд, по истечении которых управление будет передано следующей инструкции. Необходимо учитывать, что это максимальное время. Если поток закончит работу раньше, то и метод закончит выполнение раньше. Метод `wait()` возвращает значение `True`, если поток успешно завершил работу, и `False` — в противном случае. Ожидание завершения потока занимает некоторое время, в течение которого окно будет по-прежнему видимым. Чтобы не вводить пользователя в заблуждение в самом начале метода `closeEvent()` мы скрываем окно с помощью метода `hide()`.

Каждый поток может иметь собственный цикл обработки сигналов, который запускается с помощью метода `exec_()`. В этом случае потоки могут обмениваться сигналами между собой. Чтобы прервать цикл следует вызвать слот `quit()` или метод `exit(returnCode=0)`. Рассмотрим обмен сигналами между потоками на примере (листинг 1.16).

Листинг 1.16. Обмен сигналами между потоками

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class Thread1(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.count = 0
    def run(self):
        self.exec_()          # Запускаем цикл обработки сигналов
    def on_start(self):
        self.count += 1
        self.emit(QtCore.SIGNAL("s1(int)"), self.count)
```

```
class Thread2(QtCore.QThread):
    def __init__(self, parent=None):
        QtCore.QThread.__init__(self, parent)
    def run(self):
        self.exec_()          # Запускаем цикл обработки сигналов
    def on_change(self, i):
        i += 10
        self.emit(QtCore.SIGNAL("s2(const QString&)"), "%d" % i)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.label = QtGui.QLabel("Нажмите кнопку")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button = QtGui.QPushButton("Сгенерировать сигнал")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.thread1 = Thread1()
        self.thread2 = Thread2()
        self.thread1.start()
        self.thread2.start()
        self.connect(self.button, QtCore.SIGNAL("clicked()"),
                     self.thread1.on_start)
        self.connect(self.thread1, QtCore.SIGNAL("s1(int)"),
                     self.thread2.on_change)
        self.connect(self.thread2, QtCore.SIGNAL("s2(const QString&)"),
                     self.label, QtCore.SLOT("setText(const QString&)"))

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
```

```
window.setWindowTitle("Обмен сигналами между потоками")
window.resize(300, 70)
window.show()
sys.exit(app.exec_())
```

В этом примере мы создали классы `Thread1`, `Thread2` и `MyWindow`. Первые два класса предназначены для создания потоков. Внутри этих классов в методе `run()` вызывается метод `exec_()`, который запускает цикл обработки событий. Внутри конструктора класса `MyWindow` производится создание надписи, кнопки и экземпляров классов `Thread1` и `Thread2`. Далее выполняется запуск сразу двух потоков.

В следующей инструкции сигнал нажатия кнопки соединяется с методом `on_start()` первого потока. Внутри этого метода производится какая-либо операция (в нашем случае увеличение значения атрибута `count`), а затем с помощью метода `emit()` генерируется сигнал `s1(int)` и во втором параметре передается результат выполнения метода. Сигнал `s1(int)` соединен с методом `on_change()` второго потока. Внутри этого метода также производится какая-либо операция, а затем генерируется сигнал `s2(const QString&)` и передается результат выполнения метода. В свою очередь сигнал `s2(const QString&)` соединен со слотом `setText(const QString&)` объекта надписи. Таким образом, при нажатии кнопки **Сгенерировать сигнал** вначале будет вызван метод `on_start()` из класса `Thread1`, затем метод `on_change()` из класса `Thread2`, а потом результат выполнения отобразится внутри надписи в окне.

1.9.3. Модуль queue. Создание очереди заданий

В предыдущем разделе мы рассмотрели возможность обмена сигналами между потоками. Теперь предположим, что запущены десять потоков, которые ожидают задания в бесконечном цикле. Как передать задание одному потоку, а не всем сразу? И как определить, какому потоку передать задание? Можно, конечно, создать список в глобальном пространстве имен и добавлять задания в этот список. Но в этом случае придется решать вопрос о совместном использовании одного ресурса сразу десятью потоками. Ведь, если потоки будут получать задания одновременно, то одно задание могут получить сразу несколько потоков, а какому-либо потоку не хватит заданий и возникнет исключительная ситуация. Говоря простым языком, возникает ситуация, когда вы пытаетесь сесть на стул, а другой человек одновременно пытается вытащить этот стул из под вас. Думаете, что успеете сесть?

Модуль `queue`, входящий в состав стандартной библиотеки Python, позволяет решить эту проблему. Модуль содержит несколько классов, которые реализуют разного рода потоко-безопасные очереди. Перечислим эти классы:

➔ `Queue` — очередь (первым пришел, первым вышел). Формат конструктора:

```
Queue([maxsize=0])
```

Пример:

```
>>> import queue
>>> q = queue.Queue()
>>> q.put_nowait("elem1")
>>> q.put_nowait("elem2")
>>> q.get_nowait()
'elem1'
>>> q.get_nowait()
'elem2'
```

➔ `LifoQueue` — стек (последним пришел, первым вышел). Формат конструктора:

```
LifoQueue([maxsize=0])
```

Пример:

```
>>> q = queue.LifoQueue()
>>> q.put_nowait("elem1")
>>> q.put_nowait("elem2")
>>> q.get_nowait()
'elem2'
>>> q.get_nowait()
'elem1'
```

➔ `PriorityQueue` — очередь с приоритетами. При получении значения возвращается элемент с наивысшим приоритетом (наименьшим значением в первом параметре кортежа). Элементы очереди должны быть кортежами, в которых первым элементом является число, означающее приоритет, а вторым — значение элемента. Формат конструктора класса:

```
PriorityQueue([maxsize=0])
```

Пример:

```
>>> import queue
>>> q = queue.PriorityQueue()
>>> q.put_nowait((10, "elem1"))
>>> q.put_nowait((3, "elem2"))
>>> q.put_nowait((12, "elem3"))
>>> q.get_nowait()
(3, 'elem2')
>>> q.get_nowait()
(10, 'elem1')
```

```
>>> q.get_nowait()
(12, 'elem3')
```

Параметр `maxsize` задает максимальное количество элементов, которое может содержать очередь. Если параметр равен нулю (значение по умолчанию) или отрицательному значению, то размер очереди неограничен.

Экземпляры этих классов содержат следующие методы:

- `put(<Элемент>[, block=True][, timeout=None])` — добавляет элемент в очередь. Если в параметре `block` указано значение `True`, то поток будет ожидать возможности добавления элемента. В параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось добавить, то возбуждается исключение `queue.Full`;
- `put_nowait(<Элемент>)` — добавление элемента без ожидания. Эквивалентно: `put(<Элемент>, False)`
- `get([block=True][, timeout=None])` — удаляет и возвращает элемент из очереди. Если в параметре `block` указано значение `True`, то поток будет ожидать возможности получения элемента. В параметре `timeout` можно указать максимальное время ожидания в секундах. Если элемент не удалось получить, то возбуждается исключение `queue.Empty`;
- `get_nowait()` — эквивалентно инструкции `get(False)`;
- `join()` — блокирует поток, пока не будут обработаны все задания в очереди. Другие потоки после обработки текущего задания должны вызывать метод `task_done()`. Как только все задания будут обработаны, поток будет разблокирован;
- `task_done()` — этот метод должны вызывать потоки после обработки задания;
- `qsize()` — возвращает приблизительное количество элементов в очереди. Так как доступ к очереди имеют сразу несколько потоков, доверять этому значению не следует. В любой момент времени количество элементов может измениться;
- `empty()` — возвращает `True`, если очередь пуста, и `False` — в противном случае;
- `full()` — возвращает `True`, если очередь содержит элементы, и `False` — в противном случае.

Рассмотрим использование очереди в многопоточном приложении на примере (листинг 1.17).

Листинг 1.17. Использование модуля queue

```
# -*- coding: utf-8 -*-
```

```
from PyQt4 import QtCore, QtGui
import queue

class MyThread(QtCore.QThread):
    def __init__(self, id, queue, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
        self.queue = queue
    def run(self):
        while True:
            task = self.queue.get()        # Получаем задание
            self.sleep(5)                  # Имитируем обработку
            self.emit(QtCore.SIGNAL("taskDone(int, int)"),
                      task, self.id)       # Передаем данные обратно
            self.queue.task_done()

class MyWindow(QtGui.QPushButton):
    def __init__(self):
        QtGui.QPushButton.__init__(self)
        self.setText("Раздать задания")
        self.queue = queue.Queue()         # Создаем очередь
        self.threads = []
        for i in range(1, 3):              # Создаем потоки и запускаем
            thread = MyThread(i, self.queue)
            self.threads.append(thread)
            self.connect(thread, QtCore.SIGNAL("taskDone(int, int)"),
                         self.on_task_done, QtCore.Qt.QueuedConnection)
            thread.start()
        self.connect(self, QtCore.SIGNAL("clicked()"),
                     self.on_add_task)
    def on_add_task(self):
        for i in range(0, 11):
            self.queue.put(i)              # Добавляем задания в очередь
    def on_task_done(self, data, id):
        print(data, "- id =", id)          # Выводим обработанные данные
```

```
if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование модуля queue")
    window.resize(300, 30)
    window.show()
    sys.exit(app.exec_())
```

В этом примере конструктор класса `MyThread` принимает уникальный идентификатор (`id`) и ссылку на очередь (`queue`), которые сохраняются в одноименных атрибутах класса. В методе `run()` внутри бесконечного цикла производится получение элемента из очереди с помощью метода `get()`. Если очередь пуста, то поток будет ожидать, пока не появится хотя бы один элемент. Далее производится обработка задания (в нашем случае просто задержка), а затем обработанные данные передаются главному потоку через сигнал `taskDone(int, int)`. В следующей инструкции с помощью метода `task_done()` указывается, что задание было обработано.

Главный поток реализуется с помощью класса `MyWindow`. Обратите внимание на то, что производится наследование класса `QPushButton` (класс кнопки), а не класса `QWidget`. Все визуальные компоненты являются наследниками класса `QWidget`, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В нашем случае используется только кнопка, поэтому можно сразу наследовать класс `QPushButton`.

Внутри конструктора класса `MyWindow` с помощью метода `setText()` задается текст надписи на кнопке. Далее создается экземпляр класса `Queue` и сохраняется в атрибуте `queue`. В следующей инструкции производится создание списка, в котором будут храниться ссылки на объекты потоков. Сами объекты потоков создаются внутри цикла (в нашем случае создаются два объекта) и добавляются в список. Внутри цикла производится также назначение обработчика сигнала `taskDone(int, int)` и запуск потока с помощью метода `start()`. Далее назначается обработчик нажатия кнопки.

При нажатии кнопки **Раздать задания** вызывается метод `on_add_task()`, внутри которого производится добавление заданий в конец очереди. После добавления заданий потоки выходят из цикла ожидания и каждый из них получает одно уникальное задание. После окончания обработки потоки генерируют сигнал `taskDone(int, int)` и вызывают метод `task_done()`, информирующий об окончании обработки задания. Главный поток получает сигнал и вызывает метод `on_task_done()`, внутри которого через параметры будут доступны обработанные данные. Так как метод расположен в GUI-потоке мы можем изменять свойства компонентов и, например, добавить результат в список или таблицу. В нашем примере результат просто выводится в окно консоли. Чтобы увидеть сообщения следует сохранить файл с расширением py, а не pyw. После

окончания обработки задания потоки снова получают задания. Если очередь будет пуста, то потоки перейдут в режим ожидания заданий.

1.9.4. Классы QMutex и QMutexLocker

Как вы уже знаете, совместное использование одного ресурса сразу несколькими потоками может привести к непредсказуемому поведению программы. Следовательно, внутри программы необходимо предусмотреть возможность блокировки ресурса одним потоком и ожидание разблокировки другим потоком. В один момент времени только один поток должен иметь доступ к ресурсу.

Реализовать блокировку ресурса в PyQt позволяют классы `QMutex` и `QMutexLocker` из модуля `QtCore`. Конструктор класса `QMutex` имеет следующий формат:

```
QMutex([mode=QtCore.QMutex.NonRecursive])
```

Необязательный параметр `mode` может принимать значения `NonRecursive` (поток может запросить блокировку только единожды; после снятия, блокировка может быть запрошена снова; значение по умолчанию) и `Recursive` (поток может запросить блокировку несколько раз; чтобы полностью снять блокировку следует вызвать метод `unlock()` соответствующее количество раз). Конструктор возвращает объект, через который доступны следующие методы:

- `lock()` — устанавливает блокировку. Если ресурс был заблокирован другим потоком, то работа текущего потока приостанавливается до снятия блокировки;
- `tryLock([<Время ожидания>])` — устанавливает блокировку. Если блокировка была успешно установлена, то метод возвращает значение `True`. Если ресурс заблокирован другим потоком, то метод возвращает значение `False` без ожидания возможности установить блокировку. Максимальное время ожидания в миллисекундах можно указать в качестве необязательного параметра. Если в параметре указано отрицательное значение, то метод `tryLock()` аналогичен методу `lock()`;
- `unlock()` — снимает блокировку.

Рассмотрим использование класса `QMutex` на примере (листинг 1.18).

Листинг 1.18. Использование класса QMutex

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui


class MyThread(QtCore.QThread):
```

```
    x = 10                                  # Атрибут класса
    mutex = QtCore.QMutex()                 # Мьютекс
    def __init__(self, id, parent=None):
        QtCore.QThread.__init__(self, parent)
        self.id = id
    def run(self):
        self.change_x()
    def change_x(self):
        MyThread.mutex.lock()               # Блокируем
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        MyThread.mutex.unlock()             # Снимаем блокировку

class MyWindow(QtGui.QPushButton):
    def __init__(self):
        QtGui.QPushButton.__init__(self)
        self.setText("Запустить")
        self.thread1 = MyThread(1)
        self.thread2 = MyThread(2)
        self.connect(self, QtCore.SIGNAL("clicked()"), self.on_start)
    def on_start(self):
        if not self.thread1.isRunning(): self.thread1.start()
        if not self.thread2.isRunning(): self.thread2.start()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Использование класса QMutex")
    window.resize(300, 30)
    window.show()
```

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

В этом примере внутри класса `MyThread` мы создали атрибут `x`, который доступен всем экземплярам класса. Изменение значения атрибута в одном потоке повлечет изменение значения и в другом потоке. Если потоки будут изменять значение одновременно, то предсказать текущее значение атрибута становится невозможно. Следовательно, изменять значение можно только после установки блокировки.

Чтобы обеспечить блокировку, внутри класса `MyThread` создается экземпляр класса `QMutex` и сохраняется в атрибуте `mutex`. Обратите внимание на то, что сохранение производится в атрибуте класса, а не в атрибуте экземпляра класса. Чтобы блокировка сработала, необходимо, чтобы защищаемый атрибут и мьютекс находились в одной области видимости. Далее все содержимое метода `change_x()`, в котором производится изменение атрибута `x`, указывается между вызовами методов `lock()` и `unlock()`. Таким образом, гарантируется, что все инструкции внутри метода `change_x()` будут выполнены сначала одним потоком и только потом другим потоком.

Внутри конструктора класса `MyWindow` производится создание двух объектов класса `MyThread` и назначение обработчика нажатия кнопки. После нажатия кнопки **Запустить** будет вызван метод `on_start()`, внутри которого производится запуск сразу двух потоков одновременно, при условии, что потоки не были запущены ранее. В результате мы получим следующий результат в окне консоли:

```
x = 10 id = 1
x = 15 id = 1
x = 49 id = 1
x = 49 id = 2
x = 54 id = 2
x = 88 id = 2
```

Как видно из результата, сначала изменение атрибута производил поток с идентификатором `1`, а лишь затем поток с идентификатором `2`. Если блокировку не указать, то результат будет другим:

```
x = 10 id = 1
x = 15 id = 2
x = 20 id = 1
x = 54 id = 1
x = 54 id = 2
x = 88 id = 2
```

В этом случае поток с идентификатором 2 изменил значение атрибута `x` до окончания выполнения метода `change_x()` в потоке с идентификатором 1.

При возникновении исключения внутри метода `change_x()` ресурс останется заблокированным, так как поток управления не дойдет до вызова метода `unlock()`. Кроме того, можно забыть вызвать метод `unlock()` по случайности, что также приведет к вечной блокировке. Исключить подобную ситуацию позволяет класс `QMutexLocker`. Конструктор этого класса принимает объект мьютекса и устанавливает блокировку. После выхода из области видимости будет вызван деструктор класса, внутри которого блокировка автоматически снимается. Следовательно, если создать экземпляр класса `QMutexLocker` в начале метода, то после выхода из метода блокировка автоматически снимется. Переделаем метод `change_x()` из класса `MyThread` и используем класс `QMutexLocker` (листинг 1.19).

Листинг 1.19. Использование класса QMutexLocker

```
    def change_x(self):
        ml = QtCore.QMutexLocker(MyThread.mutex)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        # Блокировка автоматически снимется
```

При использовании класса `QMutexLocker` следует помнить о разнице между областями видимости в языках C++ и Python. В языке C++ область видимости ограничена блоком. Блоком может быть как функция, так и просто область, ограниченная фигурными скобками. Таким образом, если переменная объявлена внутри блока условного оператора, например, `if`, то при выходе из этого блока переменная уже не будет видна:

```
if (условие) {
    int x = 10; // Объявляем переменную
    // ...
}
// Здесь переменная x уже не видна!
```

В языке Python область видимости гораздо шире. Если мы объявим переменную внутри условного оператора, то она будет видна и после выхода из этого блока:

```
if условие:
    x = 10      # Объявляем переменную
    # ...
```

```
# Здесь переменная x видна
```

Таким образом, область видимости локальной переменной в языке Python ограничена функцией, а не любым блоком. Класс `QMutexLocker` в PyQt поддерживает протокол менеджеров контекста, который позволяет ограничить область видимости блоком инструкции `with...as`. Этот протокол гарантирует снятие блокировки, даже при наличии исключения внутри инструкции `with...as`. Переделаем метод `change_x()` из класса `MyThread` еще раз и используем инструкцию `with...as` (листинг 1.20).

Листинг 1.20. Использование инструкции with...as

```
def change_x(self):
    with QtCore.QMutexLocker(MyThread.mutex):
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 5
        self.sleep(2)
        print("x =", MyThread.x, "id =", self.id)
        MyThread.x += 34
        print("x =", MyThread.x, "id =", self.id)
        # Блокировка автоматически снимется
```

Теперь, когда вы уже знаете о возможности блокировки ресурса, следует сделать несколько замечаний. Установка и снятие блокировки занимают некоторый промежуток времени, тем самым, снижая эффективность всей программы. Поэтому встроенные типы данных не обеспечивают безопасную работу в многопоточном приложении. Прежде, чем использовать блокировки, подумайте, может быть в вашем приложении они и не нужны. Второе замечание относится к доступу к защищенному ресурсу из GUI-потока. Ожидание снятия блокировки может заблокировать GUI-поток и приложение перестанет реагировать на события. Поэтому в этом случае следует использовать сигналы, а не прямой доступ. И последнее замечание относится к *взаимной блокировке*. Если первый поток, владея ресурсом `A`, захочет получить доступ к ресурсу `B`, а второй поток, владея ресурсом `B`, захочет получить доступ к ресурсу `A`, то оба потока будут ждать снятия блокировки вечно. В этой ситуации следует предусмотреть возможность временного освобождения ресурсов одним из потоков после превышения периода ожидания.

Примечание

Для синхронизации и координации потоков предназначены также классы `QSemaphore` и `QWaitCondition`. За подробной информацией по этим классам обращайтесь к документации PyQt, а также к примерам, расположенным в папке C:\Python32\Lib\site-packages\PyQt4\examples\threads. Следует также помнить, что в стандартную библиотеку языка Python входят модули `multiprocessing` и `threading`, которые позволяют

работать с потоками в любом приложении. Однако при использовании PyQt нужно отдать предпочтение классу `QThread`, так как он позволяет работать с сигналами.

1.10. Вывод заставки

В больших приложениях загрузка начальных данных может занимать продолжительное время. На это время принято выводить окно-заставку, внутри которого отображается процесс загрузки. По окончании загрузки окно-заставка скрывается и отображается главное окно приложения. Для отображения окна-заставки в PyQt предназначен класс `QSplashScreen` из модуля `QtGui`. Конструктор класса имеет следующий формат:

```
QSplashScreen([<Родитель>, ][<Изображение>[, <Тип окна>]])
```

Первый необязательный параметр позволяет указать родительский элемент. Во втором параметре указывается ссылка на изображение (экземпляр класса `QPixmap`), которое будет изображаться на заставке. Конструктору класса `QPixmap` можно передать путь к файлу с изображением. Третий параметр предназначен для указания типа окна, например, чтобы заставка отображалась поверх всех остальных окон, следует передать флаг `QtCore.Qt.WindowStaysOnTopHint`. Экземпляр класса `QSplashScreen` содержит следующие методы:

- `show()` — отображает заставку;
- `finish(<Ссылка на окно>)` — закрывает заставку. В качестве параметра указывается ссылка на главное окно приложения;
- `showMessage(<Сообщение>[, <Выравнивание>[, <Цвет>]])` — выводит сообщение. Во втором параметре указывается местоположение надписи в окне. По умолчанию надпись выводится в левом верхнем углу окна. В качестве значения можно указать комбинацию следующих флагов через оператор `|`: `AlignTop` (по верху), `AlignCenter` (по центру вертикали и горизонтали), `AlignBottom` (по низу), `AlignHCenter` (по центру горизонтали), `AlignVCenter` (по центру вертикали), `AlignLeft` (по левой стороне), `AlignRight` (по правой стороне). В третьем параметре указывается цвет текста. В качестве значения можно указать константу из класса `QtCore.Qt` (например, `black` (по умолчанию), `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.);
- `clearMessage()` — стирает надпись;
- `setPixmap(<Изображение>)` — позволяет изменить изображение в окне. В качестве параметра указывается экземпляр класса `QPixmap`;
- `pixmap()` — возвращает экземпляр класса `QPixmap`.

Пример вывода заставки показан в листинге 1.21.

Листинг 1.21. Вывод заставки

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
from PyQt4.QtCore import Qt, SIGNAL
import time

class MyWindow(QtGui.QPushButton):
    def __init__(self):
        QtGui.QPushButton.__init__(self)
        self.setText("Закрыть окно")
        self.connect(self, SIGNAL("clicked()"), QtGui.qApp.quit)
    def load_data(self, sp):
        for i in range(1, 11):          # Имитируем процесс
            time.sleep(2)               # Что-то загружаем
            sp.showMessage("Загрузка данных... {0}%".format(i * 10),
                           Qt.AlignHCenter | Qt.AlignBottom, Qt.black)
            QtGui.qApp.processEvents()  # Запускаем оборот цикла

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    splash = QtGui.QSplashScreen(QtGui.QPixmap("img.png"))
    splash.showMessage("Загрузка данных... 0%",
                       Qt.AlignHCenter | Qt.AlignBottom, Qt.black)
    splash.show()                       # Отображаем заставку
    QtGui.qApp.processEvents()          # Запускаем оборот цикла
    window = MyWindow()
    window.setWindowTitle("Использование класса QSplashScreen")
    window.resize(300, 30)
    window.load_data(splash)            # Загружаем данные
    window.show()
    splash.finish(window)               # Скрываем заставку
    sys.exit(app.exec_())
```

1.11. Доступ к документации

Библиотека PyQt содержит свыше 670 классов. Если описывать один класс на одной странице, то объем книги по PyQt будет более 670 страниц. Однако, уложиться в одну страницу практически невозможно, следовательно, объем вырастет в два, а то и три раза. Издать книгу такого большого объема не представляется возможным, поэтому в этой книге мы рассмотрим только наиболее часто используемые возможности библиотеки PyQt, те возможности, которые вы будете использовать каждый день в своей практике. Чтобы получить полную информацию следует обращаться к документации. Где ее найти мы сейчас и рассмотрим.

Самая последняя версия документации в формате HTML доступна на сайте **http://www.riverbankcomputing.co.uk/**. С этого же сайта можно загрузить последнюю версию библиотеки. Документация входит также в состав дистрибутива и устанавливается в папку C:\Python32\Lib\site-packages\PyQt4\doc\html. Чтобы отобразить содержание следует открыть в браузере файл index.html. Полный список классов расположен в файле classes.html, а список всех модулей — в файле modules.html. Каждое название класса или модуля является ссылкой на страницу с подробным описанием. Помимо документации в состав дистрибутива входят примеры, которые устанавливаются в папку C:\Python32\Lib\site-packages\PyQt4\examples. Обязательно изучите эти примеры.

Глава 2. Управление окном приложения

Создать окно и управлять им позволяет класс `QWidget`. Класс `QWidget` наследует два класса — `QObject` и `QPaintDevice`. В свою очередь класс `QWidget` является базовым классом для всех визуальных компонентов, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В этой главе мы рассмотрим методы класса `QWidget` применительно к окну верхнего уровня, однако следует помнить, что те же самые методы можно применять и к любым компонентам. Например, метод, позволяющий управлять размерами окна, можно использовать и для изменения размеров компонента, имеющего родителя. Тем не менее, некоторые методы имеет смысл использовать только для окон верхнего уровня, например, метод, позволяющий изменить текст в заголовке окна, не имеет смысла использовать в обычных компонентах.

Для создания окна верхнего уровня помимо класса `QWidget` можно использовать и другие классы, которые являются наследниками класса `QWidget`, например, класс `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При использовании класса `QDialog` окно будет выравниваться по центру экрана и иметь только две кнопки в заголовке окна — **Справка** и **Закрыть**. Кроме того, можно использовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Использование классов `QDialog` и `QMainWindow` имеет свои отличия, которые мы будем рассматривать в отдельных главах.

2.1. Создание и отображение окна

Самый простой способ создать пустое окно, показан в листинге 2.1.

Листинг 2.1. Создание и отображение окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()                    # Создаем окно
window.setWindowTitle("Заголовок окна")     # Указываем заголовок
window.resize(300, 50)                      # Минимальные размеры
window.show()                               # Отображаем окно
```

Листинги на странице http://unicross.narod.ru/pyqt/

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

Конструктор класса `QWidget` имеет следующий формат:

```
QWidget([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Если в параметре `flags` указан тип окна, то компонент, имея родителя, будет обладать своим собственным окном, но будет привязан к родителю. Это позволяет, например, создать модальное окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `flags` мы рассмотрим в следующем разделе.

Указать ссылку на родительский компонент уже после создания объекта позволяет метод `setParent()`. Формат метода:

```
setParent(<Родитель>[, <Тип окна>])
```

Получить ссылку на родительской компонент можно с помощью метода `parentWidget()`. Если компонент не имеет родителя, то метод возвращает значение `None`.

Для изменения текста в заголовке окна предназначен метод `setWindowTitle()`. Формат метода:

```
setWindowTitle(<Текст, отображаемый в заголовке>)
```

После создания окна необходимо вызвать метод `show()`, чтобы окно и все дочерние компоненты отобразились на экране. Для сокрытия окна предназначен метод `hide()`. Для отображения и сокрытия компонентов можно также воспользоваться методом `setVisible(<Флаг>)`. Если в параметре указано значение `True`, то компонент будет отображен, а если значение `False`, то компонент будет скрыт. Пример отображения окна и всех дочерних компонентов:

```
window.setVisible(True)
```

Проверить, видим компонент в настоящее время или нет, позволяет метод `isVisible()`. Метод возвращает `True`, если компонент видим, и `False` — в противном случае. Кроме того, можно воспользоваться методом `isHidden()`. Метод возвращает `True`, если компонент скрыт, и `False` — в противном случае.

2.2. Указание типа окна

При использовании класса `QWidget` по умолчанию окно создается с заголовком, в котором расположены: иконка, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Закрыть**. Указать другой тип создаваемого окна позволяет метод `setWindowFlags()` или параметр `flags` в конструкторе класса `QWidget`. Обратите внимание на то, что

метод `setWindowFlags()` должен вызываться перед отображением окна. Формат метода:

```
setWindowFlags(<Тип окна>)
```

В параметре `<Тип окна>` можно указать следующие атрибуты из класса `QtCore.Qt`:

- `Widget` — тип по умолчанию для класса `QWidget`;
- `Window` — указывает, что компонент является окном, независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком, в котором расположены кнопки **Свернуть**, **Развернуть** и **Закрыть**. По умолчанию размеры окна можно изменять с помощью мыши;
- `Dialog` — диалоговое окно. Окно выводится с рамкой и заголовком, в котором расположены кнопки **Справка** и **Закрыть**. Размеры окна можно изменять с помощью мыши. Пример указания типа для диалогового окна:

  ```
  window.setWindowFlags(QtCore.Qt.Dialog)
  ```

- `Sheet`;
- `Drawer`;
- `Popup` — указывает, что окно является всплывающим меню. Окно выводится без рамки и заголовка. Кроме того, окно может отбрасывать тень. Изменить размеры окна с помощью мыши нельзя;
- `Tool` — сообщает, что окно является панелью инструментов. Окно выводится с рамкой и заголовком (меньшем по высоте, чем обычное окно), в котором расположена кнопка **Закрыть**. Размеры окна можно изменять с помощью мыши;
- `ToolTip` — указывает, что окно является всплывающей подсказкой. Окно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя;
- `SplashScreen` — сообщает, что окно является заставкой. Окно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя. Значение по умолчанию для класса `QSplashScreen`;
- `Desktop` — указывает, что окно является рабочим столом. Окно вообще не отображается на экране. Значение по умолчанию для класса `QDesktopWidget`;
- `SubWindow` — сообщает, что окно является дочерним компонентом, независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком (меньшем по высоте, чем обычное окно) без кнопок. Изменить размеры окна с помощью мыши нельзя.

Для определения типа окна из программы предназначен метод `windowType()`:

```
window.setWindowFlags(QtCore.Qt.Window)
print(window.windowType() == QtCore.Qt.Window) # True
```

Кроме того, можно воспользоваться атрибутом `WindowType_Mask`:

```
>>> from PyQt4 import QtCore
>>> flag = QtCore.Qt.Window
>>> (flag & QtCore.Qt.WindowType_Mask) == QtCore.Qt.Window
True
```

Для окон верхнего уровня можно дополнительно указать следующие атрибуты из класса `QtCore.Qt` через оператор `|` (перечислены только наиболее часто используемые атрибуты; полный список смотрите в документации):

- `MSWindowsFixedSizeDialogHint` — запрещает изменение размеров окна. Изменить размеры с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной;
- `FramelessWindowHint` — убирает рамку и заголовок окна. Изменять размеры окна и перемещать его нельзя;
- `CustomizeWindowHint` — убирает рамку и заголовок окна, но добавляет эффект объемности. Размеры окна можно изменять с помощью мыши;
- `WindowTitleHint` — добавляет заголовок окна. Выведем окно фиксированного размера с заголовком, в котором находится только текст:

```
window.setWindowFlags(QtCore.Qt.Window |
                      QtCore.Qt.FramelessWindowHint |
                      QtCore.Qt.WindowTitleHint)
```

- `WindowSystemMenuHint` — добавляет оконное меню и кнопку **Закрыть**;
- `WindowMinimizeButtonHint` — кнопка **Свернуть** в заголовке окна делается активной, а кнопка **Развернуть** — неактивной;
- `WindowMaximizeButtonHint` — кнопка **Развернуть** в заголовке окна делается активной, а кнопка **Свернуть** — неактивной;
- `WindowMinMaxButtonsHint` — кнопки **Свернуть** и **Развернуть** в заголовке окна делаются активными;
- `WindowCloseButtonHint` — добавляет кнопку **Закрыть**;
- `WindowContextHelpButtonHint` — добавляет кнопку **Справка**. Кнопки **Свернуть** и **Развернуть** в заголовке окна не отображаются;
- `WindowStaysOnTopHint` — сообщает системе, что окно всегда должно отображаться поверх всех других окон;
- `WindowStaysOnBottomHint` — сообщает системе, что окно всегда должно расположено позади всех других окон.

Получить все установленные флаги из программы позволяет метод `windowFlags()`. Пример определения типа окна:

```
window.setWindowFlags(QtCore.Qt.Window)
print((window.windowFlags() & QtCore.Qt.WindowType_Mask) ==
      QtCore.Qt.Window) # True
```

2.3. Изменение и получение размеров окна

Для изменения размеров окна предназначены следующие методы:

- `resize(<Ширина>, <Высота>)` — изменяет текущий размер окна. Если содержимое окна не помещается в установленный размер, то размер будет выбран так, чтобы компоненты поместились без искажения, при условии, что используются менеджеры геометрии. Следовательно, заданный размер может не соответствовать реальному размеру окна. Если используется абсолютное позиционирование, то компоненты могут оказаться наполовину или полностью за пределами видимой части окна. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

  ```
  window.resize(100, 70)
  window.resize(QtCore.QSize(100, 70))
  ```

- `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — изменяет одновременно положение компонента и его текущий размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента), а третий и четвертый параметры — ширину и высоту. В качестве параметра можно также указать экземпляр класса `QRect`. Пример:

  ```
  window.setGeometry(100, 100, 100, 70)
  window.setGeometry(QtCore.QRect(100, 100, 100, 70))
  ```

- `setFixedSize(<Ширина>, <Высота>)` — задает фиксированный размер. Изменить размеры окна с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

  ```
  window.setFixedSize(100, 70)
  window.setFixedSize(QtCore.QSize(100, 70))
  ```

- `setFixedWidth(<Ширина>)` — задает фиксированный размер только по ширине. Изменить ширину окна с помощью мыши нельзя;

- `setFixedHeight(<Высота>)` — задает фиксированный размер только по высоте. Изменить высоту окна с помощью мыши нельзя;

- `setMinimumSize(<Ширина>, <Высота>)` — задает минимальный размер. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.setMinimumSize(100, 70)
window.setMinimumSize(QtCore.QSize(100, 70))
```

- `setMinimumWidth(<Ширина>)` и `setMinimumHeight(<Высота>)` — задают минимальный размер только по ширине и высоте соответственно;
- `setMaximumSize(<Ширина>, <Высота>)` — задает максимальный размер. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.setMaximumSize(100, 70)
window.setMaximumSize(QtCore.QSize(100, 70))
```

- `setMaximumWidth(<Ширина>)` и `setMaximumHeight(<Высота>)` — задают максимальный размер только по ширине и высоте соответственно;
- `setBaseSize(<Ширина>, <Высота>)` — задает базовые размеры. В качестве параметра можно также указать экземпляр класса `QSize`. Пример:

```
window.setBaseSize(500, 500)
window.setBaseSize(QtCore.QSize(500, 500))
```

- `adjustSize()` — подгоняет размеры компонента под содержимое. При этом учитываются рекомендуемые размеры, возвращаемые методом `sizeHint()`.

Получить размеры позволяют следующие методы:

- `width()` и `height()` — возвращают текущую ширину и высоту соответственно:

```
window.resize(50, 70)
print(window.width(), window.height())                # 50 70
```

- `size()` — возвращает экземпляр класса `QSize`, содержащий текущие размеры:

```
window.resize(50, 70)
print(window.size().width(), window.size().height()) # 50 70
```

- `minimumSize()` — возвращает экземпляр класса `QSize`, содержащий минимальные размеры;
- `minimumWidth()` и `minimumHeight()` — возвращают минимальную ширину и высоту соответственно;
- `maximumSize()` — возвращает экземпляр класса `QSize`, содержащий максимальные размеры;
- `maximumWidth()` и `maximumHeight()` — возвращают максимальную ширину и высоту соответственно;
- `baseSize()` — возвращает экземпляр класса `QSize`, содержащий базовые размеры;

- `sizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого размера;
- `minimumSizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый минимальный размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого минимального размера;
- `rect()` — возвращает экземпляр класса `QRect`, содержащий координаты прямоугольника, в который вписан компонент. Пример:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.rect()
print(rect.left(), rect.top())      # 0 0
print(rect.width(), rect.height()) # 100 70
```

- `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты относительно родительского компонента. Пример:

```
window.setGeometry(QtCore.QRect(100, 100, 100, 70))
rect = window.geometry()
print(rect.left(), rect.top())      # 100 100
print(rect.width(), rect.height()) # 100 70
```

При изменении и получении размеров окна следует учитывать, что:

- размеры не включают высоту заголовка окна и ширину границ;
- размер компонентов может изменяться в зависимости от настроек стиля. Например, на разных компьютерах может быть указан шрифт разного наименования и размера. Поэтому от указания фиксированных размеров лучше отказаться;
- размер окна может изменяться в промежутке между получением значения и действиями, выполняющими обработку этих значений в программе. Например, сразу после получения размера пользователь изменил размеры окна с помощью мыши.

Чтобы получить размеры окна, включая высоту заголовка и ширину границ, следует воспользоваться методом `frameSize()`. Метод возвращает экземпляр класса `QSize`. Обратите внимание на то, что полные размеры окна доступны только после отображения окна. До этого момента размеры совпадают с размерами окна без учета высоты заголовка и ширины границ. Пример получения полного размера окна:

```
window.resize(200, 70)                    # Задаем размеры
# ...
window.show()                             # Отображаем окно
print(window.width(), window.height())    # 200 70
```

```
print(window.frameSize().width(),
      window.frameSize().height())       # 208 104
```

Чтобы получить координаты окна с учетом высоты заголовка и ширины границ следует воспользоваться методом `frameGeometry()`. Обратите внимание на то, что полные размеры окна доступны только после отображения окна. Метод возвращает экземпляр класса `QRect`. Пример:

```
window.setGeometry(100, 100, 200, 70)
# ...
window.show()                            # Отображаем окно
rect = window.geometry()
print(rect.left(), rect.top())           # 100 100
print(rect.width(), rect.height())       # 200 70
rect = window.frameGeometry()
print(rect.left(), rect.top())           # 96 70
print(rect.width(), rect.height())       # 208 104
```

2.4. Местоположение окна на экране

Задать местоположение окна на экране монитора позволяют следующие методы:

- `move(<X>, <Y>)` — изменяет положение компонента относительно родителя. Метод учитывает высоту заголовка и ширину границ. В качестве параметра можно также указать экземпляр класса `QPoint`. Пример вывода окна в левом верхнем углу экрана:

  ```
  window.move(0, 0)
  window.move(QtCore.QPoint(0, 0))
  ```

- `setGeometry(<X>, <Y>, <Ширина>, <Высота>)` — изменяет одновременно положение компонента и его текущий размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента), а третий и четвертый параметры — ширину и высоту. Обратите внимание на то, что метод не учитывает высоту заголовка и ширину границ. Поэтому, если указать координаты `(0, 0)`, то заголовок окна и левая граница будут за пределами экрана. В качестве параметра можно также указать экземпляр класса `QRect`. Пример:

  ```
  window.setGeometry(100, 100, 100, 70)
  window.setGeometry(QtCore.QRect(100, 100, 100, 70))
  ```

Обратите внимание

Начало координат расположено в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз.

Получить позицию окна позволяют следующие методы:

- `x()` и `y()` — возвращают координаты левого верхнего угла окна относительно родителя по осям `X` и `Y` соответственно. Методы учитывают высоту заголовка и ширину границ. Пример:

```
window.move(10, 10)
print(window.x(), window.y())              # 10 10
```

- `pos()` — возвращает экземпляр класса `QPoint`, содержащий координаты левого верхнего угла окна относительно родителя. Метод учитывает высоту заголовка и ширину границ. Пример:

```
window.move(10, 10)
print(window.pos().x(), window.pos().y()) # 10 10
```

- `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты относительно родительского компонента. Обратите внимание на то, что метод не учитывает высоту заголовка и ширину границ. Пример:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.geometry()
print(rect.left(), rect.top())             # 14 40
print(rect.width(), rect.height())         # 300 100
```

- `frameGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты с учетом высоты заголовка и ширины границ. Обратите внимание на то, что полные размеры окна доступны только после отображения окна. Пример:

```
window.resize(300, 100)
window.move(10, 10)
rect = window.frameGeometry()
print(rect.left(), rect.top())             # 10 10
print(rect.width(), rect.height())         # 308 134
```

Для отображения окна по центру экрана или у правой или нижней границы необходимо знать размеры экрана. Для получения размеров экрана вначале следует вызвать статический метод `QApplication.desktop()`, который возвращает ссылку на компонент рабочего стола. Получить размеры экрана позволяют следующие методы этого объекта:

- `width()` — возвращает ширину всего экрана в пикселах;
- `height()` — возвращает высоту всего экрана в пикселах:

```
desktop = QtGui.QApplication.desktop()
print(desktop.width(), desktop.height()) # 1280 1024
```

➔ `screenGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты всего экрана:

```
desktop = QtGui.QApplication.desktop()
rect = desktop.screenGeometry()
print(rect.left(), rect.top())           # 0 0
print(rect.width(), rect.height())       # 1280 1024
```

➔ `availableGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты только доступной части экрана (без размера **Панели задач**):

```
desktop = QtGui.QApplication.desktop()
rect = desktop.availableGeometry()
print(rect.left(), rect.top())           # 0 0
print(rect.width(), rect.height())       # 1280 994
```

Пример отображения окна примерно по центру экрана показан в листинге 2.2.

Листинг 2.2. Вывод окна примерно по центру экрана

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
desktop = QtGui.QApplication.desktop()
x = (desktop.width() - window.width()) // 2
y = (desktop.height() - window.height()) // 2
window.move(x, y)
window.show()
sys.exit(app.exec_())
```

В этом примере мы воспользовались методами `width()` и `height()`, которые не учитывают высоту заголовка и ширину границ. В большинстве случаев этого способа достаточно. Если необходима точность при выравнивании, то для получения размеров окна можно воспользоваться методом `frameSize()`. Однако, этот метод возвращает корректные значения только после отображения окна. Если код выравнивания по центру расположить после вызова метода `show()`, то окно отобразится вначале в одном

месте экрана, а затем переместится в центр, что вызовет неприятное мелькание. Чтобы исключить мелькание следует вначале отобразить окно за пределами экрана, а затем переместить окно в центр экрана (листинг 2.3).

Листинг 2.3. Вывод окна точно по центру экрана

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Вывод окна по центру экрана")
window.resize(300, 100)
window.move(window.width() * -2, 0)
window.show()
desktop = QtGui.QApplication.desktop()
x = (desktop.width() - window.frameSize().width()) // 2
y = (desktop.height() - window.frameSize().height()) // 2
window.move(x, y)
sys.exit(app.exec_())
```

Этот способ можно также использовать для выравнивания окна по правому краю экрана. Например, чтобы расположить окно в правом верхнем углу экрана следует заменить код, выравнивающий окно по центру, из предыдущего примера следующим кодом:

```
desktop = QtGui.QApplication.desktop()
x = desktop.width() - window.frameSize().width()
window.move(x, 0)
```

Если попробовать вывести окно в правом нижнем углу, то может возникнуть проблема, так как в операционной системе Windows в нижней части экрана располагается **Панель задач**. В итоге окно будет частично расположено под **Панелью задач**. Чтобы при размещении окна учитывать местоположение **Панели задач** необходимо использовать метод `availableGeometry()`. Получить высоту **Панели задач**, расположенной в нижней части экрана, можно, например, так:

```
desktop = QtGui.QApplication.desktop()
taskBarHeight = (desktop.screenGeometry().height() -
                 desktop.availableGeometry().height())
```

Следует также заметить, что в некоторых операционных системах **Панель задач** может быть прикреплена к любой стороне экрана. Кроме того, экран может быть разделен на несколько рабочих столов. Все это необходимо учитывать при размещении окна. За подробной информацией обращайтесь к документации по классу `QDesktopWidget`.

2.5. Указание координат и размеров

В двух предыдущих разделах были упомянуты классы `QPoint`, `QSize` и `QRect`. Класс `QPoint` описывает координаты точки, класс `QSize` — размеры, а класс `QRect` — координаты и размеры прямоугольной области. Рассмотрим эти классы более подробно.

> ***Примечание***
> Классы `QPoint`, `QSize` и `QRect` предназначены для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать классы `QPointF`, `QSizeF` и `QRectF` соответственно.

2.5.1. Класс QPoint. Координаты точки

Класс `QPoint` из модуля `QtCore` описывает координаты точки. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
QPoint()
QPoint(<X>, <Y>)
QPoint(<Экземпляр класса QPoint>)
```

Первый конструктор создает экземпляр класса с нулевыми координатами:

```
>>> from PyQt4 import QtCore
>>> p = QtCore.QPoint()
>>> p.x(), p.y()
(0, 0)
```

Второй конструктор позволяет явно указать координаты точки:

```
>>> p = QtCore.QPoint(10, 88)
>>> p.x(), p.y()
(10, 88)
```

Третий конструктор создает новый экземпляр на основе другого экземпляра:

```
>>> p = QtCore.QPoint(QtCore.QPoint(10, 88))
>>> p.x(), p.y()
(10, 88)
```

Через экземпляр класса доступны следующие методы:

- `x()` и `y()` — возвращают координаты по осям `X` и `Y` соответственно;
- `setX()` и `setY()` — задают координаты по осям `X` и `Y` соответственно;
- `isNull()` — возвращает `True`, если координаты равны нулю, и `False` — в противном случае:

```
>>> p = QtCore.QPoint()
>>> p.isNull()
True
>>> p.setX(10); p.setY(88)
>>> p.x(), p.y()
(10, 88)
```

- `manhattanLength()` — возвращает сумму абсолютных значений координат:

```
>>> QtCore.QPoint(10, 88).manhattanLength()
98
```

Над двумя экземплярами класса `QPoint` определены операции `+`, `+=`, `-` (минус), `-=`, `==` и `!=`. Для смены знака координат можно воспользоваться унарным оператором `-`. Кроме того, экземпляр класса `QPoint` можно умножить или разделить на вещественное число (операторы `*`, `*=`, `/` и `/=`). Пример:

```
>>> p1 = QtCore.QPoint(10, 20); p2 = QtCore.QPoint(5, 9)
>>> p1 + p2, p1 - p2
(PyQt4.QtCore.QPoint(15, 29), PyQt4.QtCore.QPoint(5, 11))
>>> p1 * 2.5, p1 / 2.0
(PyQt4.QtCore.QPoint(25, 50), PyQt4.QtCore.QPoint(5, 10))
>>> -p1, p1 == p2, p1 != p2
(PyQt4.QtCore.QPoint(-10, -20), False, True)
```

2.5.2. Класс QSize. Размеры прямоугольной области

Класс `QSize` из модуля `QtCore` описывает размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
QSize()
QSize(<Ширина>, <Высота>)
QSize(<Экземпляр класса QSize>)
```

Первый конструктор создает экземпляр класса с отрицательной шириной и высотой. Второй конструктор позволяет явно указать ширину и высоту. Третий конструктор создает новый экземпляр на основе другого экземпляра.

```
>>> from PyQt4 import QtCore
```

```
>>> s1=QtCore.QSize(); s2=QtCore.QSize(10, 55); s3=QtCore.QSize(s2)
>>> s1
PyQt4.QtCore.QSize(-1, -1)
>>> s2, s3
(PyQt4.QtCore.QSize(10, 55), PyQt4.QtCore.QSize(10, 55))
```

Через экземпляр класса доступны следующие методы:

- `width()` и `height()` — возвращают ширину и высоту соответственно;
- `setWidth()` и `setHeight()` — задают ширину и высоту соответственно:

```
>>> s = QtCore.QSize()
>>> s.setWidth(10), s.setHeight(55)
(None, None)
>>> s.width(), s.height()
(10, 55)
```

- `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;
- `isValid()` — возвращает `True`, если ширина и высота больше или равны нулю, и `False` — в противном случае;
- `isEmpty()` — возвращает `True`, если один параметр (ширина или высота) меньше или равен нулю, и `False` — в противном случае;
- `scale()` — производит изменение размеров области в соответствии со значением параметра `<Тип преобразования>`. Метод изменяет текущий объект и ничего не возвращает. Форматы метода:

```
scale(<Экземпляр класса QSize>, <Тип преобразования>)
scale(<Ширина>, <Высота>, <Тип преобразования>)
```

В параметре `<Тип преобразования>` могут быть указаны следующие атрибуты из класса `QtCore.Qt` или соответствующие им целые значения:

- `IgnoreAspectRatio` — `0` — непосредственно изменяет размеры без сохранения пропорций сторон;
- `KeepAspectRatio` — `1` — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
- `KeepAspectRatioByExpanding` — `2` — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

Если новая ширина или высота имеет значение `0`, то размеры изменяются непосредственно без сохранения пропорций, вне зависимости от значения параметра `<Тип преобразования>`.

Примечание

Чтобы полностью понять принцип действия атрибутов `KeepAspectRatio` и `KeepAspectRatioByExpanding` откройте изображение qimage-scaling.png, которое расположено в папке C:\Python32\Lib\site-packages\PyQt4\doc\html\images.

Примеры:

```
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.IgnoreAspectRatio); s
PyQt4.QtCore.QSize(70, 60)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatio); s
PyQt4.QtCore.QSize(70, 28)
>>> s = QtCore.QSize(50, 20)
>>> s.scale(70, 60, QtCore.Qt.KeepAspectRatioByExpanding); s
PyQt4.QtCore.QSize(150, 60)
```

➔ `boundedTo(<Экземпляр класса QSize>)` — возвращает экземпляр класса `QSize`, который содержит минимальную ширину и высоту из текущих размеров и размеров, указанных в параметре. Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.boundedTo(QtCore.QSize(400, 5))
PyQt4.QtCore.QSize(50, 5)
>>> s.boundedTo(QtCore.QSize(40, 50))
PyQt4.QtCore.QSize(40, 20)
```

➔ `expandedTo(<Экземпляр класса QSize>)` — возвращает экземпляр класса `QSize`, который содержит максимальную ширину и высоту из текущих размеров и размеров, указанных в параметре. Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.expandedTo(QtCore.QSize(400, 5))
PyQt4.QtCore.QSize(400, 20)
>>> s.expandedTo(QtCore.QSize(40, 50))
PyQt4.QtCore.QSize(50, 50)
```

➔ `transpose()` — меняет значения местами. Метод изменяет текущий объект и ничего не возвращает. Пример:

```
>>> s = QtCore.QSize(50, 20)
>>> s.transpose(); s
PyQt4.QtCore.QSize(20, 50)
```

Над двумя экземплярами класса `QSize` определены операции +, +=, – (минус), –=, == и !=. Кроме того, экземпляр класса `QSize` можно умножить или разделить на вещественное число (операторы *, *=, / и /=). Пример:

```
>>> s1 = QtCore.QSize(50, 20); s2 = QtCore.QSize(10, 5)
>>> s1 + s2, s1 - s2
(PyQt4.QtCore.QSize(60, 25), PyQt4.QtCore.QSize(40, 15))
>>> s1 * 2.5, s1 / 2
(PyQt4.QtCore.QSize(125, 50), PyQt4.QtCore.QSize(25, 10))
>>> s1 == s2, s1 != s2
(False, True)
```

2.5.3. Класс QRect. Координаты и размеры прямоугольной области

Класс `QRect` из модуля `QtCore` описывает координаты и размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
QRect()
QRect(<left>, <top>, <Ширина>, <Высота>)
QRect(<Координаты левого верхнего угла>, <Размеры>)
QRect(<Координаты левого верхнего угла>,
      <Координаты правого нижнего угла>)
QRect(<Экземпляр класса QRect>)
```

Первый конструктор создает экземпляр класса со значениями по умолчанию. Второй и третий конструкторы позволяют указать координаты левого верхнего угла и размеры области. Во втором конструкторе значения указываются отдельно. В третьем конструкторе координаты задаются с помощью класса `QPoint`, а размеры — с помощью класса `QSize`. Четвертый конструктор позволяет указать координаты левого верхнего угла и правого нижнего угла. В качестве значений указываются экземпляры класса `QPoint`. Пятый конструктор создает новый экземпляр на основе другого экземпляра. Пример:

```
>>> from PyQt4 import QtCore
>>> r = QtCore.QRect()
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(0, 0, -1, -1, 0, 0)
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
```

```
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QSize(400, 300))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> r = QtCore.QRect(QtCore.QPoint(10, 15), QtCore.QPoint(409, 314))
>>> r.left(), r.top(), r.right(), r.bottom(), r.width(), r.height()
(10, 15, 409, 314, 400, 300)
>>> QtCore.QRect(r)
PyQt4.QtCore.QRect(10, 15, 400, 300)
```

Изменить значения уже после создания экземпляра позволяют следующие методы:

- `setLeft(<X1>)`, `setX(<X1>)`, `setTop(<Y1>)` и `setY(<Y1>)` — задают координаты левого верхнего угла по осям `X` и `Y`. Пример:

```
>>> r = QtCore.QRect()
>>> r.setLeft(10); r.setTop(55); r
PyQt4.QtCore.QRect(10, 55, -10, -55)
>>> r.setX(12); r.setY(81); r
PyQt4.QtCore.QRect(12, 81, -12, -81)
```

- `setRight(<X2>)` и `setBottom(<Y2>)` — задают координаты правого нижнего угла по осям `X` и `Y`. Пример:

```
>>> r = QtCore.QRect()
>>> r.setRight(12); r.setBottom(81); r
PyQt4.QtCore.QRect(0, 0, 13, 82)
```

- `setTopLeft(<QPoint>)` — задает координаты левого верхнего угла;
- `setTopRight(<QPoint>)` — задает координаты правого верхнего угла;
- `setBottomLeft(<QPoint>)` — задает координаты левого нижнего угла;
- `setBottomRight(<QPoint>)` — задает координаты правого нижнего угла:

```
>>> r = QtCore.QRect()
>>> r.setTopLeft(QtCore.QPoint(10, 5))
>>> r.setBottomRight(QtCore.QPoint(39, 19)); r
PyQt4.QtCore.QRect(10, 5, 30, 15)
>>> r.setTopRight(QtCore.QPoint(39, 5))
>>> r.setBottomLeft(QtCore.QPoint(10, 19)); r
PyQt4.QtCore.QRect(10, 5, 30, 15)
```

- `setWidth(<Ширина>)`, `setHeight(<Высота>)` и `setSize(<Экземпляр класса QSize>)` — задают ширину и высоту области;
- `setRect(<X1>, <Y1>, <Ширина>, <Высота>)` — задает координаты левого верхнего угла и размеры области;
- `setCoords(<X1>, <Y1>, <X2>, <Y2>)` — задает координаты левого верхнего угла и правого нижнего угла. Пример:

```
>>> r = QtCore.QRect()
>>> r.setRect(10, 10, 100, 500); r
PyQt4.QtCore.QRect(10, 10, 100, 500)
>>> r.setCoords(10, 10, 109, 509); r
PyQt4.QtCore.QRect(10, 10, 100, 500)
```

Переместить область при изменении координат позволяют следующие методы:

- `moveTo(<X1>, <Y1>)`, `moveTo(<QPoint>)`, `moveLeft(<X1>)` и `moveTop(<Y1>)` — перемещают координаты левого верхнего угла:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTo(0, 0); r
PyQt4.QtCore.QRect(0, 0, 400, 300)
>>> r.moveTo(QtCore.QPoint(10, 10)); r
PyQt4.QtCore.QRect(10, 10, 400, 300)
>>> r.moveLeft(5); r.moveTop(0); r
PyQt4.QtCore.QRect(5, 0, 400, 300)
```

- `moveRight(<X2>)` и `moveBottom(<Y2>)` — перемещают координаты правого нижнего угла;
- `moveTopLeft(<QPoint>)` — перемещает координаты левого верхнего угла;
- `moveTopRight(<QPoint>)` — перемещает координаты правого верхнего угла;
- `moveBottomLeft(<QPoint>)` — перемещает координаты левого нижнего угла;
- `moveBottomRight(<QPoint>)` — перемещает координаты правого нижнего угла. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.moveTopLeft(QtCore.QPoint(0, 0)); r
PyQt4.QtCore.QRect(0, 0, 400, 300)
>>> r.moveBottomRight(QtCore.QPoint(599, 499)); r
PyQt4.QtCore.QRect(200, 200, 400, 300)
```

- `moveCenter(<QPoint>)` — перемещает координаты центра;

- `translate(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translate(<QPoint>)` — перемещают координаты левого верхнего угла относительно текущего значения координат:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.translate(20, 15); r
PyQt4.QtCore.QRect(20, 15, 400, 300)
>>> r.translate(QtCore.QPoint(10, 5)); r
PyQt4.QtCore.QRect(30, 20, 400, 300)
```

- `translated(<Сдвиг по оси X>, <Сдвиг по оси Y>)` и `translated(<QPoint>)` — метод аналогичен методу `translate()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий;

- `adjust(<X1>, <Y1>, <X2>, <Y2>)` — сдвигает координаты левого верхнего угла и правого нижнего угла относительно текущих значений координат:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.adjust(10, 5, 10, 5); r
PyQt4.QtCore.QRect(10, 5, 400, 300)
```

- `adjusted(<X1>, <Y1>, <X2>, <Y2>)` — метод аналогичен методу `adjust()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий.

Для получения значений предназначены следующие методы:

- `left()` и `x()` — возвращают координату левого верхнего угла по оси `X`;
- `top()` и `y()` — возвращают координату левого верхнего угла по оси `Y`;
- `right()` и `bottom()` — возвращают координаты правого нижнего угла по осям `X` и `Y` соответственно;
- `width()` и `height()` — возвращают ширину и высоту соответственно;
- `size()` — возвращает размеры в виде экземпляра класса `QSize`. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.left(), r.top(), r.x(), r.y(), r.right(), r.bottom()
(10, 15, 10, 15, 409, 314)
>>> r.width(), r.height(), r.size()
(400, 300, PyQt4.QtCore.QSize(400, 300))
```

- `topLeft()` — возвращает координаты левого верхнего угла;
- `topRight()` — возвращает координаты правого верхнего угла;
- `bottomLeft()` — возвращает координаты левого нижнего угла;

➔ `bottomRight()` — возвращает координаты правого нижнего угла. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.topLeft(), r.topRight()
(PyQt4.QtCore.QPoint(10, 15), PyQt4.QtCore.QPoint(409, 15))
>>> r.bottomLeft(), r.bottomRight()
(PyQt4.QtCore.QPoint(10, 314), PyQt4.QtCore.QPoint(409, 314))
```

➔ `center()` — возвращает координаты центра области. Например, вывести окно по центру доступной области экрана можно так:

```
desktop = QtGui.QApplication.desktop()
window.move(desktop.availableGeometry().center() -
            window.rect().center())
```

➔ `getRect()` — возвращает кортеж с координатами левого верхнего угла и размерами области;

➔ `getCoords()` — возвращает кортеж с координатами левого верхнего угла и правого нижнего угла. Пример:

```
>>> r = QtCore.QRect(10, 15, 400, 300)
>>> r.getRect(), r.getCoords()
((10, 15, 400, 300), (10, 15, 409, 314))
```

Прочие методы:

➔ `isNull()` — возвращает `True`, если ширина и высота равны нулю, и `False` — в противном случае;

➔ `isValid()` — возвращает `True`, если `left() < right()` и `top() < bottom()`, и `False` — в противном случае;

➔ `isEmpty()` — возвращает `True`, если `left() > right()` или `top() > bottom()`, и `False` — в противном случае;

➔ `normalized()` — исправляет ситуацию, при которой `left() > right()` или `top() > bottom()` и возвращает новый экземпляр класса `QRect`. Пример:

```
>>> r = QtCore.QRect(QtCore.QPoint(409, 314),
                     QtCore.QPoint(10, 15))
>>> r
PyQt4.QtCore.QRect(409, 314, -398, -298)
>>> r.normalized()
PyQt4.QtCore.QRect(10, 15, 400, 300)
```

➔ `contains(<QPoint>[, <Флаг>])` и `contains(<X>, <Y>[, <Флаг>])` — возвращает `True`, если точка с указанными координатами расположена внутри области или на ее границе, и `False` — в противном случае. Если во втором параметре указано значение `True`, то точка должна быть расположена только внутри области, а не на ее границе. Значение параметра по умолчанию — `False`. Пример:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(0, 10), r.contains(0, 10, True)
(True, False)
```

➔ `contains(<QRect>[, <Флаг>])` — возвращает `True`, если указанная область расположена внутри текущей области или на ее краю, и `False` — в противном случае. Если во втором параметре указано значение `True`, то указанная область должна быть расположена только внутри текущей области, а не на ее краю. Значение параметра по умолчанию — `False`. Пример:

```
>>> r = QtCore.QRect(0, 0, 400, 300)
>>> r.contains(QtCore.QRect(0, 0, 20, 5))
True
>>> r.contains(QtCore.QRect(0, 0, 20, 5), True)
False
```

➔ `intersects(<QRect>)` — возвращает `True`, если указанная область пересекается с текущей областью, и `False` — в противном случае;

➔ `intersect(<QRect>)` и `intersected(<QRect>)` — возвращают область, которая расположена на пересечении текущей и указанной областей. Пример:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.intersects(QtCore.QRect(10, 10, 20, 20))
True
>>> r.intersect(QtCore.QRect(10, 10, 20, 20))
PyQt4.QtCore.QRect(10, 10, 10, 10)
>>> r.intersected(QtCore.QRect(10, 10, 20, 20))
PyQt4.QtCore.QRect(10, 10, 10, 10)
```

➔ `unite(<QRect>)` и `united(<QRect>)` — возвращают область, которая охватывает текущую и указанную области. Пример:

```
>>> r = QtCore.QRect(0, 0, 20, 20)
>>> r.unite(QtCore.QRect(30, 30, 20, 20))
PyQt4.QtCore.QRect(0, 0, 50, 50)
>>> r.united(QtCore.QRect(30, 30, 20, 20))
PyQt4.QtCore.QRect(0, 0, 50, 50)
```

Над двумя экземплярами класса `QRect` определены операции `&` и `&=` (пересечение), `|` и `|=` (объединение), `in` (проверка на вхождение), == и `!=`. Пример:

```
>>> r1, r2 = QtCore.QRect(0, 0, 20, 20), QtCore.QRect(10, 10, 20, 20)
>>> r1 & r2, r1 | r2
(PyQt4.QtCore.QRect(10, 10, 10, 10), PyQt4.QtCore.QRect(0, 0, 30, 30))
>>> r1 in r2, r1 in QtCore.QRect(0, 0, 30, 30)
(False, True)
>>> r1 == r2, r1 != r2
(False, True)
```

2.6. Разворачивание и сворачивание окна

В заголовке окна расположены кнопки **Свернуть** и **Развернуть**, с помощью которых можно свернуть окно в значок на **Панели задач** или максимально развернуть окно. Выполнить подобные действия из программы позволяют следующие методы класса `QWidget`:

- `showMinimized()` — сворачивает окно на **Панель задач**. Эквивалентно нажатию кнопки **Свернуть** в заголовке окна;
- `showMaximized()` — разворачивает окно до максимального размера. Эквивалентно нажатию кнопки **Развернуть** в заголовке окна;
- `showFullScreen()` — включает полноэкранный режим отображения окна. Окно отображается без заголовка и границ;
- `showNormal()` — отменяет сворачивание, максимальный размер и полноэкранный режим;
- `activateWindow()` — делает окно активным (т. е. имеющим фокус ввода). В Windows, если окно было ранее свернуто в значок на **Панель задач**, то оно автоматически не будет отображено на экране. В этом случае станет активным только значок на **Панели задач**;
- `setWindowState(<Флаги>)` — изменяет статус окна в зависимости от переданных флагов. В качестве параметра указывается комбинация следующих атрибутов из класса `QtCore.Qt` через побитовые операторы:
  - `WindowNoState` — нормальное состояние окна;
  - `WindowMinimized` — окно свернуто;
  - `WindowMaximized` — окно максимально развернуто;
  - `WindowFullScreen` — полноэкранный режим;

- `WindowActive` — окно имеет фокус ввода, т. е. является активным.

Например, включить полноэкранный режим можно так:

```
window.setWindowState((window.windowState() &
    ~(QtCore.Qt.WindowMinimized | QtCore.Qt.WindowMaximized))
    | QtCore.Qt.WindowFullScreen)
```

Проверить текущий статус окна позволяют следующие методы:

➔ `isMinimized()` — возвращает `True`, если окно свернуто, и `False` — в противном случае;

➔ `isMaximized()` — возвращает `True`, если окно раскрыто до максимальных размеров, и `False` — в противном случае;

➔ `isFullScreen()` — возвращает `True`, если включен полноэкранный режим, и `False` — в противном случае;

➔ `isActiveWindow()` — возвращает `True`, если окно имеет фокус ввода, и `False` — в противном случае;

➔ `windowState()` — возвращает комбинацию флагов, обозначающих текущий статус окна. Пример проверки использования полноэкранного режима:

```
if window.windowState() & QtCore.Qt.WindowFullScreen:
    print("Полноэкранный режим")
```

Пример разворачивания и сворачивания окна приведен в листинге 2.4.

Листинг 2.4. Разворачивание и сворачивание окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.btnMin = QtGui.QPushButton("Свернуть")
        self.btnMax = QtGui.QPushButton("Развернуть")
        self.btnFull = QtGui.QPushButton("Полный экран")
        self.btnNormal = QtGui.QPushButton("Нормальный размер")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.btnMin)
        vbox.addWidget(self.btnMax)
```

```
        vbox.addWidget(self.btnFull)
        vbox.addWidget(self.btnNormal)
        self.setLayout(vbox)
        self.connect(self.btnMin, QtCore.SIGNAL("clicked()"),
                     self.on_min)
        self.connect(self.btnMax, QtCore.SIGNAL("clicked()"),
                     self.on_max)
        self.connect(self.btnFull, QtCore.SIGNAL("clicked()"),
                     self.on_full)
        self.connect(self.btnNormal, QtCore.SIGNAL("clicked()"),
                     self.on_normal)
    def on_min(self):
        self.showMinimized()
    def on_max(self):
        self.showMaximized()
    def on_full(self):
        self.showFullScreen()
    def on_normal(self):
        self.showNormal()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.setWindowTitle("Разворачивание и сворачивание окна")
    window.resize(300, 100)
    window.show()
    sys.exit(app.exec_())
```

2.7. Управление прозрачностью окна

Сделать окно полупрозрачным позволяет метод `setWindowOpacity()` из класса `QWidget`. Формат метода:

```
setWindowOpacity(<Вещественное число от 0.0 до 1.0>)
```

В качестве параметра указывается вещественное число от `0.0` до `1.0`. Число `0.0` соответствует полностью прозрачному окну, а число `1.0` — отсутствию прозрачности. Для получения степени прозрачности окна из программы предназначен метод `windowOpacity()`, который возвращает вещественное число от `0.0` до `1.0`. Выведем окно со степенью прозрачности `0.5` (листинг 2.5).

Листинг 2.5. Полупрозрачное окно

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Полупрозрачное окно")
window.resize(300, 100)
window.setWindowOpacity(0.5)
window.show()
print(window.windowOpacity()) # Выведет: 0.4980392156862745
sys.exit(app.exec_())
```

2.8. Модальные окна

Модальным называется окно, которое не позволяет взаимодействовать с другими окнами в том же приложении. Пока модальное окно не будет закрыто, сделать активным другое окно нельзя. Например, если в программе Microsoft Word выбрать пункт меню **Файл | Сохранить как**, то откроется модальное диалоговое окно, позволяющее выбрать путь и название файла. Пока это окно не будет закрыто, вы не сможете взаимодействовать с главным окном приложения.

Указать, что окно является модальным, позволяет метод `setWindowModality(<Флаг>)` из класса `QWidget`. В качестве параметра могут быть указаны следующие атрибуты из класса `QtCore.Qt` или соответствующие им значения:

- `NonModal` — `0` — окно не является модальным;
- `WindowModal` — `1` — окно блокирует только родительские окна в пределах иерархии;
- `ApplicationModal` — `2` — окно блокирует все окна в приложении.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение позволяет метод `windowModality()`. Проверить, является ли окно модальным, можно с помощью метода `isModal()`. Метод возвращает `True`, если окно является модальным, и `False` — в противном случае.

Создадим два независимых окна. В первом окне разместим кнопку, при нажатии которой откроем модальное окно. Это модальное окно будет блокировать только первое окно, но не второе. При открытии модального окна отобразим его примерно по центру родительского окна (листинг 2.6).

Листинг 2.6. Модальные окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

def show_modal_window():
    global modalWindow
    global window1
    modalWindow = QtGui.QWidget(window1, QtCore.Qt.Window)
    modalWindow.setWindowTitle("Модальное окно")
    modalWindow.resize(200, 50)
    modalWindow.setWindowModality(QtCore.Qt.WindowModal)
    modalWindow.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
    modalWindow.move(window1.geometry().center() -
         modalWindow.rect().center() - QtCore.QPoint(4, 30))
    modalWindow.show()

app = QtGui.QApplication(sys.argv)
window1 = QtGui.QWidget()
window1.setWindowTitle("Обычное окно")
window1.resize(300, 100)
button = QtGui.QPushButton("Открыть модальное окно")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       show_modal_window)
vbox = QtGui.QVBoxLayout()
```

```
vbox.addWidget(button)
window1.setLayout(vbox)
window1.show()

window2 = QtGui.QWidget()
window2.setWindowTitle("Это окно не будет блокировано при WindowModal")
window2.resize(500, 100)
window2.show()

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

Если запустить приложение и нажать кнопку **Открыть модальное окно**, то откроется окно, выровненное примерно (произвести точное выравнивание вы сможете самостоятельно) по центру родительского окна. При этом получить доступ к родительскому окну можно только при закрытии модального окна, а второе окно блокировано не будет. Если заменить атрибут `WindowModal` атрибутом `ApplicationModal`, то оба окна будут блокированы.

Обратите внимание на то, что в конструктор модального окна мы передали ссылку на первое окно и атрибут `Window`. Если ссылку не указать, то окно блокировано не будет, а если атрибут не указать, то окно вообще не откроется. Кроме того, мы объявили переменную `modalWindow` глобальной, иначе при достижении конца функции переменная выйдет из области видимости и окно будет автоматически удалено. Чтобы объект окна автоматически удалялся при закрытии окна атрибуту `WA_DeleteOnClose` в методе `setAttribute()` было присвоено значение `True`.

Модальные окна в большинстве случаев являются диалоговыми. Для работы с диалоговыми окнами в PyQt предназначен класс `QDialog`, который автоматически выравнивает окно по центру экрана или по центру родительского окна. Кроме того, этот класс предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и многое другое. Подробно класс `QDialog` мы будем изучать в отдельной главе.

2.9. Смена иконки в заголовке окна

По умолчанию в левом верхнем углу окна отображается стандартная иконка. Отобразить другую иконку в заголовке окна позволяет метод `setWindowIcon()` из класса `QWidget`. В качестве параметра метод принимает экземпляр класса `QIcon`. Чтобы загрузить иконку из файла следует передать путь к файлу конструктору класса. Если указан относительный путь, то поиск файла будет производиться относительно текущего рабочего каталога. Получить список поддерживаемых форматов файлов можно с помощью метода `supportedImageFormats()` из класса `QImageReader`.

Метод возвращает список экземпляров класса `QByteArray`. Получим список поддерживаемых форматов:

```
from PyQt4 import QtGui
for i in QtGui.QImageReader.supportedImageFormats():
    print(str(i, "ascii").upper(), end=" ")
```

Результат выполнения на моем компьютере:

```
BMP GIF ICO JPEG JPG MNG PBM PGM PNG PPM SVG SVGZ TIF TIFF XBM XPM
```

Если для окна не задана иконка, то будет использоваться иконка приложения, установленная с помощью метода `setWindowIcon()` из класса `QApplication`. В качестве параметра метод принимает экземпляр класса `QIcon`.

Вместо загрузки иконки из файла можно воспользоваться одной из встроенных иконок. Загрузить стандартную иконку позволяет следующий код:

```
ico = window.style().standardIcon(QtGui.QStyle.SP_MessageBoxCritical)
window.setWindowIcon(ico)
```

Посмотреть список всех встроенных иконок можно в документации к классу `QStyle` (C:/Python32/Lib/site-packages/PyQt4/doc/html/qstyle.html#StandardPixmap-enum).

В качестве примера создаем иконку в формате PNG размером 16 на 16 пикселей и сохраняем ее в одной папке с программой, а далее устанавливаем эту иконку для окна и для всего приложения (листинг 2.7).

Листинг 2.7. Смена иконки в заголовке окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Смена иконки в заголовке окна")
window.resize(300, 100)
window.setWindowIcon(QtGui.QIcon("icon.png")) # Иконка для окна
app.setWindowIcon(QtGui.QIcon("icon.png"))    # Иконка приложения
window.show()
sys.exit(app.exec_())
```

2.10. Изменение цвета фона окна

Чтобы изменить цвет фона окна (или компонента) следует установить палитру с настроенной ролью `Window` (или `Background`). Цветовая палитра содержит цвета для каждой роли и состояния компонента. Указать состояние компонента позволяют следующие атрибуты из класса `QPalette` и соответствующие им значения:

- `Active` и `Normal` — `0` — компонент активен (окно находится в фокусе ввода);
- `Disabled` — `1` — компонент недоступен;
- `Inactive` — `2` — компонент неактивен (окно находится вне фокуса ввода).

Получить текущую палитру компонента позволяет метод `palette()`. Чтобы изменить цвет для какой-либо роли и состояния следует воспользоваться методом `setColor()` из класса `QPalette`. Формат метода:

```
setColor([<Состояние>, ]<Роль>, <Цвет>)
```

В параметре `<Роль>` указывается для какого элемента изменяется цвет. Например, атрибут `Window` (или `Background`) изменяет цвет фона, а `WindowText` (или `Foreground`) — цвет текста. Полный список атрибутов смотрите в документации по классу `QPalette`.

В параметре `<Цвет>` указывается цвет элемента. В качестве значения можно указать константу из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.).

После настройки палитры необходимо вызвать метод `setPalette()` и передать ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку необходимо передать значение `True` методу `setAutoFillBackground()`.

Изменить цвет фона можно также с помощью CSS-атрибута `background-color`. Для этого следует передать таблицу стилей в метод `setStyleSheet()`. Таблицы стилей могут быть внешними (подключение через командную строку), установленными на уровне приложения (с помощью метода `setStyleSheet()` из класса `QApplication`) или установленными на уровне компонента (с помощью метода `setStyleSheet()` из класса `QWidget`). Атрибуты, установленные последними, обычно перекрывают значения аналогичных атрибутов, указанных ранее. Если вы занимались Web-программированием, то CSS вам уже знаком, а если нет, то придется дополнительно изучить HTML и CSS.

Создадим окно с надписью. Для активного окна установим зеленый цвет, а для неактивного — красный. Цвет фона надписи сделаем белым. Для изменения фона окна будем устанавливать палитру, а для изменения цвета фона надписи — CSS-атрибут `background-color` (листинг 2.8).

Листинги на странице http://unicross.narod.ru/pyqt/

Листинг 2.8. Изменение цвета фона окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Изменение цвета фона окна")
window.resize(300, 100)
pal = window.palette()
pal.setColor(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QColor("#008800"))
pal.setColor(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
             QtGui.QColor("#ff0000"))
window.setPalette(pal)
label = QtGui.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignCenter)
label.setStyleSheet("background-color: #ffffff;")
label.setAutoFillBackground(True)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

2.11. Использование изображения в качестве фона

В качестве фона окна (или компонента) можно использовать изображение. Для этого необходимо получить текущую палитру компонента с помощью метода `palette()`, а затем вызвать метод `setBrush()` из класса `QPalette`. Формат метода:

```
setBrush([<Состояние>, ]<Роль>, <Экземпляр класса QBrush>)
```

Первые два параметра аналогичны соответствующим параметрам в методе `setColor()`, который мы рассматривали в предыдущем разделе. В третьем параметре указывается экземпляр класса `QBrush`. Форматы конструктора класса:

```
QBrush(<Стиль кисти>)
```

```
QBrush(<Цвет>, <Стиль кисти>=SolidPattern)
QBrush(<Цвет>, <Экземпляр класса QPixmap>)
QBrush(<Экземпляр класса QPixmap>)
QBrush(<Экземпляр класса QImage>)
```

В параметре `<Стиль кисти>` указывается атрибуты из класса `QtCore.Qt`, задающие стиль кисти, например, `NoBrush`, `SolidPattern`, `Dense1Pattern`, `Dense2Pattern`, `Dense3Pattern`, `Dense4Pattern`, `Dense5Pattern`, `Dense6Pattern`, `Dense7Pattern`, `CrossPattern` и др. С помощью этого параметра можно сделать цвет сплошным (`SolidPattern`) или имеющим текстуру (например, атрибут `CrossPattern` задает текстуру в виде сетки).

В параметре `<Цвет>` указывается цвет кисти. В качестве значения можно указать константу из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Например, установка сплошного цвета фона окна выглядит так:

```
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
    QtGui.QBrush(QtGui.QColor("#008800"), QtCore.Qt.SolidPattern))
window.setPalette(pal)
```

Параметры `<Экземпляр класса QPixmap>` и `<Экземпляр класса QImage>` позволяют передать объекты изображений. Конструкторы этих классов принимают путь к файлу, который может быть как абсолютным, так и относительным.

После настройки палитры необходимо вызвать метод `setPalette()` и передать ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку необходимо передать значение `True` в метод `setAutoFillBackground()`.

Указать, что изображение используется в качестве фона, можно также с помощью CSS-атрибутов `background` и `background-image`. С помощью CSS-атрибута `background-repeat` можно дополнительно указать режим повтора фонового рисунка. Он может принимать значения `repeat`, `repeat-x` (повтор только по горизонтали), `repeat-y` (повтор только по вертикали) и `no-repeat` (не повторяется).

Создадим окно с надписью. Для активного окна установим одно изображение (с помощью изменения палитры), а для надписи другое изображение (с помощью CSS-атрибута `background-image`) (листинг 2.9).

Листинг 2.9. Использование изображения в качестве фона

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Изображение в качестве фона")
window.resize(300, 100)
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QBrush(QtGui.QPixmap("img1.png")))
window.setPalette(pal)
label = QtGui.QLabel("Текст надписи")
label.setAlignment(QtCore.Qt.AlignCenter)
label.setAutoFillBackground(True)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(label)
window.setLayout(vbox)
window.show()
sys.exit(app.exec_())
```

2.12. Создание окна произвольной формы

Чтобы создать окно произвольной формы следует выполнить следующие шаги:

- создать изображение нужной формы с прозрачным фоном и сохранить его, например, в формате PNG;
- создать экземпляр класса `QPixmap`, передав конструктору класса абсолютный или относительный путь к изображению;
- установить изображение в качестве фона окна с помощью палитры;
- отделить альфа-канал с помощью метода `mask()` из класса `QPixmap`;
- передать маску в метод `setMask()` объекта окна;
- убрать рамку окна, например, передав комбинацию следующих флагов:
  `QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint`

Если для создания окна используется класс `QLabel`, то вместо установки палитры можно передать экземпляр класса `QPixmap` в метод `setPixmap()`, а маску в метод `setMask()`.

В качестве примера создадим круглое окно с кнопкой, с помощью которой можно закрыть окно. Окно выведем без заголовка и границ (листинг 2.10).

Листинг 2.10. Создание окна произвольной формы

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowFlags(QtCore.Qt.Window | QtCore.Qt.FramelessWindowHint)
window.setWindowTitle("Создание окна произвольной формы")
window.resize(300, 300)
pixmap = QtGui.QPixmap("fon.png")
pal = window.palette()
pal.setBrush(QtGui.QPalette.Normal, QtGui.QPalette.Window,
             QtGui.QBrush(pixmap))
pal.setBrush(QtGui.QPalette.Inactive, QtGui.QPalette.Window,
             QtGui.QBrush(pixmap))
window.setPalette(pal)
window.setMask(pixmap.mask())
button = QtGui.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 135)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
window.show()
sys.exit(app.exec_())
```

2.13. Всплывающие подсказки

При работе с программой у пользователя могут возникать вопросы для чего предназначен тот или иной компонент. Обычно для информирования пользователя используются надписи, расположенные над компонентом или перед ним. Но часто место в окне либо ограничено, либо вывод таких надписей испортит весь дизайн окна. В таких случаях принято выводить текст подсказки в отдельном окне без рамки при наведении указателя мыши на компонент. После выведения указателя окно должно автоматически закрываться.

В PyQt нет необходимости создавать окно с подсказкой самому и следить за перемещениями указателя мыши. Весь процесс автоматизирован и максимально упрощен. Чтобы создать всплывающие подсказки для окна или любого другого компонента и управлять ими, нужно воспользоваться следующими методами из класса `QWidget`:

- `setToolTip(<Текст>)` — задает текст вплывающей подсказки. В качестве параметра можно указать простой текст или текст в формате HTML. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку;
- `toolTip()` — возвращает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст справки. Обычно этот метод используется для вывода информации большего объема, чем во всплывающей подсказке. Чтобы отобразить текст справки необходимо сделать компонент активным и нажать комбинацию клавиш <Shift>+<F1>. У диалоговых окон в заголовке окна есть кнопка **Справка**. После нажатия этой кнопки вид курсора изменится на стрелку со знаком вопроса. Чтобы отобразить текст справки в этом случае следует щелкнуть мышью на компоненте. В качестве параметра можно указать простой текст или текст в формате HTML. Чтобы отключить вывод подсказки достаточно передать в этот метод пустую строку;
- `whatsThis()` — возвращает текст справки.

Создадим окно с кнопкой и зададим для них текст вплывающих подсказок и текст справки (листинг 2.11).

Листинг 2.11. Всплывающие подсказки

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

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

```
window = QtGui.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Всплывающие подсказки")
window.resize(300, 70)
button = QtGui.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
button.setToolTip("Это всплывающая подсказка для кнопки")
window.setToolTip("Это всплывающая подсказка для окна")
button.setWhatsThis("Это справка для кнопки")
window.setWhatsThis("Это справка для окна")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       QtGui.qApp, QtCore.SLOT("quit()"))
window.show()
sys.exit(app.exec_())
```

2.14. Закрытие окна из программы

В предыдущих разделах для закрытия окна мы использовали слот `quit()` и метод `exit(returnCode=0)` объекта приложения. Однако эти методы не только закрывают текущее окно, но и завершают выполнение всего приложения. Чтобы закрыть только текущее окно следует воспользоваться методом `close()` из класса `QWidget`. Метод возвращает значение `True`, если окно успешно закрыто, и `False` — в противном случае. Закрыть сразу все окна приложения позволяет слот `closeAllWindows()` из класса `QApplication`.

Если для окна атрибут `WA_DeleteOnClose` из класса `QtCore.Qt` установлен в истинное значение, то после закрытия окна объект окна будет автоматически удален. Если атрибут имеет ложное значение, то окно просто скрывается. Значение атрибута можно изменить с помощью метода `setAttribute()`:

```
window.setAttribute(QtCore.Qt.WA_DeleteOnClose, True)
```

После вызова метода `close()` или нажатия кнопки **Закрыть** в заголовке окна генерируется событие `QEvent.Close`. Если внутри класса определить метод с предопределенным названием `closeEvent()`, то это событие можно перехватить и обработать. В качестве параметра метод принимает объект класса `QCloseEvent`, который содержит методы `accept()` (позволяет закрыть окно) и `ignore()` (запрещает закрытие окна). Вызывая эти методы можно контролировать процесс закрытия окна.

В качестве примера закроем окно при нажатии на кнопку (листинг 2.12).

Листинг 2.12. Закрытие окна из программы

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget(flags=QtCore.Qt.Dialog)
window.setWindowTitle("Закрытие окна из программы")
window.resize(300, 70)
button = QtGui.QPushButton("Закрыть окно", window)
button.setFixedSize(150, 30)
button.move(75, 20)
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       window, QtCore.SLOT("close()"))
window.show()
sys.exit(app.exec_())
```

Глава 3. Обработка сигналов и событий

При взаимодействии пользователя с окном происходят события. В ответ на события система генерирует определенные сигналы. *Сигналы* — это своего рода извещения системы о том, что пользователь выполнил какое-либо действие или в самой системе возникло некоторое условие. Сигналы являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик сигнала, как удалить обработчик, а также уметь правильно обработать событие. Какие сигналы генерирует тот или иной компонент мы будем рассматривать при изучении конкретного компонента.

3.1. Назначение обработчиков сигналов

Чтобы обработать какой-либо сигнал необходимо сопоставить ему функцию или метод класса, которые будут вызваны при наступлении события. Назначить обработчик позволяет статический метод `connect()` из класса `QObject`. Форматы метода:

```
connect(<Объект>, <Сигнал>, <Обработчик>[, <ConnectionType>])
connect(<Объект1>, <Сигнал>, <Объект2>, <Слот>[, <ConnectionType>])
connect(<Объект1>, <Сигнал>, <Объект2>, <Сигнал>[, <ConnectionType>])
```

Кроме того, существует обычный (не статический) метод `connect()`:

```
<Объект2>.connect(<Объект1>, <Сигнал>, <Слот>[, <ConnectionType>])
```

Первый формат позволяет назначить обработчик сигнала `<Сигнал>`, возникшего при изменении статуса объекта `<Объект>`. Если обработчик успешно назначен, то метод возвращает значение `True`. Для одного сигнала можно назначить несколько обработчиков, которые будут вызываться в порядке назначения в программе. В параметре `<Сигнал>` указывается результат выполнения функции `SIGNAL()`. Формат функции:

```
QtCore.SIGNAL("<Название сигнала>([Тип параметров])")
```

Каждый компонент имеет определенный набор сигналов, например, при щелчке на кнопке генерируется сигнал `clicked(bool=0)`. Внутри круглых скобок могут быть указаны типы параметров, которые передаются в обработчик. Если параметров нет, то указываются только круглые скобки. Пример указания сигнала без параметров:

```
QtCore.SIGNAL("clicked()")
```

В этом случае обработчик не принимает никаких параметров. Указание сигнала с параметром выглядит следующим образом:

```
QtCore.SIGNAL("clicked(bool)")
```

В этом случае обработчик должен принимать один параметр, значение которого всегда будет равно `0` (`False`), так как это значение по умолчанию для сигнала `clicked()`.

В параметре `<Обработчик>` можно указать:

- ссылку на пользовательскую функцию;
- ссылку на метод класса;
- ссылку на экземпляр класса. В этом случае внутри класса должен существовать метод `__call__()`.

Пример обработки щелчка на кнопке приведен в листинге 3.1.

Листинг 3.1. Варианты назначения пользовательского обработчика

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

def on_clicked():
    print("Кнопка нажата. Функция on_clicked()")

class MyClass():
    def __init__(self, x=0):
        self.x = x
    def __call__(self):
        print("Кнопка нажата. Метод MyClass.__call__()")
        print("x =", self.x)
    def on_clicked(self):
        print("Кнопка нажата. Метод MyClass.on_clicked()")

obj = MyClass()
app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Нажми меня")
# Назначаем обработчиком функцию
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), on_clicked)
# Назначаем обработчиком метод класса
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       obj.on_clicked)
```

```
# Передача параметра в обработчик
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), MyClass(10))
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), MyClass(5))
button.show()
sys.exit(app.exec_())
```

Результат выполнения в окне консоли при одном щелчке на кнопке:

```
Кнопка нажата. Функция on_clicked()
Кнопка нажата. Метод MyClass.on_clicked()
Кнопка нажата. Метод MyClass.__call__()
x = 10
Кнопка нажата. Метод MyClass.__call__()
x = 5
```

Второй формат метода `connect()` назначает в качестве обработчика метод Qt-объекта `<Объект2>`. Обычно используется для назначения стандартного метода из класса, входящего в состав библиотеки Qt. В качестве примера при щелчке на кнопке завершим работу приложения (листинг 3.2).

Листинг 3.2. Завершение работы приложения при щелчке на кнопке

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Завершить работу")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       app,    QtCore.SLOT("quit()"))
button.show()
sys.exit(app.exec_())
```

Как видно из примера, в третьем параметре метода `connect()` указывается объект приложения, а в четвертом параметре в функцию `SLOT()` передается название метода `quit()` в виде строки. Благодаря гибкости языка Python данное назначение обработчика можно записать иначе:

```
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"), app.quit)
```

Третий формат метода `connect()` позволяет передать сигнал другому объекту. Рассмотрим передачу сигнала на примере (листинг 3.3).

Листинг 3.3. Передача сигнала другому объекту

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.button1 = QtGui.QPushButton("Кнопка 1. Нажми меня")
        self.button2 = QtGui.QPushButton("Кнопка 2")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.resize(300, 100)
        # Передача сигнала от кнопки 1 к кнопке 2
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.button2, QtCore.SIGNAL('clicked()'))
        # Способ 1 (4 параметра)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     self, QtCore.SLOT("on_clicked_button2()"))
        # Способ 2 (3 параметра)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button2()"))
    @QtCore.pyqtSlot()
    def on_clicked_button2(self):
        print("Сигнал получен кнопкой 2")

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

В этом примере мы создали класс `MyWindow`, который наследует класс `QtGui.QWidget`. В методе инициализации `__init__()` вначале вызывается конструктор базового класса и создаются две кнопки. Далее создается вертикальный контейнер и в него добавляются объекты кнопок с помощью метода `addWidget()`. С помощью метода `setLayout()` вертикальный контейнер добавляется в основное окно. Затем назначаются обработчики событий для кнопок. Обратите внимание на то, что метод `connect()` вызывается как метод нашего класса. Это возможно потому, что большинство PyQt-классов наследуют класс `QObject`, в котором определен метод `connect()`. Обработка нажатия кнопки производится с помощью метода `on_clicked_button2()`, который превращен декоратором `@QtCore.pyqtSlot()` в одноименный слот.

При нажатии первой кнопки производится вызов первого обработчика, который перенаправляет сигнал на вторую кнопку. Назначение перенаправления, соответствующее третьему формату метода `connect()`, выглядит так:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             self.button2, QtCore.SIGNAL('clicked()'))
```

После перенаправления сигнала вызывается обработчик второй кнопки. Для второй кнопки мы назначили обработчик двумя способами. Первый способ соответствует второму формату метода `connect()`:

```
self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button2()"))
```

Второй способ соответствует четвертому формату метода `connect()`:

```
self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             QtCore.SLOT("on_clicked_button2()"))
```

Необязательный параметр `<ConnectionType>` определяет тип соединения между сигналом и обработчиком. На этот параметр следует обратить особое внимание при использовании нескольких потоков в приложении, так как изменять GUI-поток из другого потока нельзя. В параметре можно указать одно из следующих атрибутов из класса `QtCore.Qt`:

- `AutoConnection` — `0` — значение по умолчанию. Если источник сигнала и обработчик находятся в одном потоке, то эквивалентно значению `DirectConnection`, а если в разных потоках — то `QueuedConnection`;
- `DirectConnection` — `1` — обработчик вызывается сразу после генерации сигнала. Обработчик выполняется в потоке источника сигнала;
- `QueuedConnection` — `2` — сигнал помещается в очередь обработки событий. Обработчик выполняется в потоке приемника сигнала;
- `BlockingQueuedConnection` — `4` — аналогично значению `QueuedConnection`, но пока сигнал не обработан поток будет заблокирован. Обратите внимание на то, что

источник сигнала и обработчик должны быть обязательно расположены в разных потоках;

- `UniqueConnection` — `0x80` — аналогично значению `AutoConnection`, но обработчик можно назначить только если он не был назначен ранее. Например, если изменить способы назначения обработчика из предыдущего примера для кнопки `button2` следующим образом, то второй обработчик назначен не будет:

```
st = self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button2()"),
             QtCore.Qt.UniqueConnection)
print(st)
st = self.connect(self.button2, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button2()"),
             QtCore.Qt.UniqueConnection)
print(st)
```

Результат:

```
True
False
```

- `AutoCompatConnection` — `3` — значение использовалось по умолчанию в Qt 3.

3.2. Блокировка и удаление обработчика

Для блокировки и удаления обработчиков предназначены следующие методы из класса `QObject`:

- `blockSignals(<Флаг>)` — временно блокирует прием сигналов, если параметр имеет значение `True`, и снимает блокировку, если параметр имеет значение `False`. Метод возвращает логическое представление предыдущего состояния соединения;
- `signalsBlocked()` — метод возвращает значение `True`, если блокировка установлена, и `False` — в противном случае;
- `disconnect()` — удаляет обработчик. Метод является статическим и доступен без создания экземпляра класса. Форматы метода:

```
disconnect(<Объект>, <Сигнал>, <Обработчик>)
disconnect(<Объект1>, <Сигнал>, <Объект2>, <Слот>)
```

Первый формат метода `disconnect()` позволяет удалить `<Обработчик>` сигнала. В параметре `<Обработчик>` можно указать ссылку на пользовательскую функцию или метод класса. Второй формат позволяет удалить слот, связанный с объектом. Если обработчик успешно удален, то метод `disconnect()` возвращает значение `True`.

Значения, указанные в методе `disconnect()`, должны совпадать со значениями, используемыми при назначении обработчика. Например, если обработчик назначался таким образом:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             self, QtCore.SLOT("on_clicked_button1()"))
```

или таким:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             QtCore.SLOT("on_clicked_button1()"))
```

то удалить его можно следующим образом:

```
self.disconnect(self.button1, QtCore.SIGNAL("clicked()"),
                self, QtCore.SLOT("on_clicked_button1()"))
```

Эти два способа назначения обработчика являются эквивалентными. В первом способе вызывается статический метод `connect()`. В этом случае ссылка на экземпляр класса передается явным образом в третьем параметре. Во втором способе используется метод класса `QObject`. В этом случае ссылка на экземпляр класса передается неявным образом. Метод `disconnect()` является статическим методом, поэтому ссылку на экземпляр класса необходимо передавать явным образом через третий параметр.

Создадим окно с четырьмя кнопками (листинг 3.4). Для кнопки **Нажми меня** назначим обработчик для сигнала `clicked()`. Чтобы информировать о нажатии кнопки выведем сообщение в окно консоли. Для кнопок **Блокировать**, **Разблокировать** и **Удалить обработчик** создадим обработчики, которые будут изменять статус обработчика для кнопки **Нажми меня**.

Листинг 3.4. Блокировка и удаление обработчика

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Блокировка и удаление обработчика")
        self.resize(300, 150)
        self.button1 = QtGui.QPushButton("Нажми меня")
        self.button2 = QtGui.QPushButton("Блокировать")
        self.button3 = QtGui.QPushButton("Разблокировать")
        self.button4 = QtGui.QPushButton("Удалить обработчик")
```

```
        self.button3.setEnabled(False)
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        vbox.addWidget(self.button3)
        vbox.addWidget(self.button4)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button1()"))
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button2()"))
        self.connect(self.button3, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button3()"))
        self.connect(self.button4, QtCore.SIGNAL("clicked()"),
                     QtCore.SLOT("on_clicked_button4()"))
    @QtCore.pyqtSlot()
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
    @QtCore.pyqtSlot()
    def on_clicked_button2(self):
        self.button1.blockSignals(True)
        self.button2.setEnabled(False)
        self.button3.setEnabled(True)
    @QtCore.pyqtSlot()
    def on_clicked_button3(self):
        self.button1.blockSignals(False)
        self.button2.setEnabled(True)
        self.button3.setEnabled(False)
    @QtCore.pyqtSlot()
    def on_clicked_button4(self):
        self.disconnect(self.button1,
                        QtCore.SIGNAL("clicked()"), self,
                        QtCore.SLOT("on_clicked_button1()"))
        self.button2.setEnabled(False)
        self.button3.setEnabled(False)
```

```
        self.button4.setEnabled(False)


if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Если после отображения окна нажать кнопку **Нажми меня**, то в окно консоли будет выведена строка "Нажата кнопка button1". Нажатие кнопки **Блокировать** производит блокировку обработчика. Теперь при нажатии кнопки **Нажми меня** никаких сообщений в окно консоли не выводится. Отменить блокировку можно с помощью кнопки **Разблокировать**. Нажатие кнопки **Удалить обработчик** производит полное удаление обработчика. В этом случае чтобы обрабатывать нажатие кнопки **Нажми меня** необходимо заново назначить обработчик.

Отключить генерацию сигнала можно также сделав компонент неактивным с помощью следующих методов из класса `QWidget`:

- `setEnabled(<Флаг>)` — если в параметре указано значение `False`, то компонент станет неактивным. Чтобы сделать компонент опять активным следует передать значение `True`;
- `setDisabled(<Флаг>)` — если в параметре указано значение `True`, то компонент станет неактивным. Чтобы сделать компонент опять активным следует передать значение `False`.

Проверить, активен компонент или нет, позволяет метод `isEnabled()`. Метод возвращает значение `True`, если компонент активен, и `False` — в противном случае.

3.3. Генерация сигнала из программы

В некоторых случаях необходимо вызвать сигнал из программы. Например, при заполнении последнего текстового поля и нажатии клавиши <Enter> можно имитировать нажатие кнопки и тем самым запустить обработчик этого сигнала. Выполнить генерацию сигнала из программы позволяет метод `emit()` из класса `QObject`. Формат метода:

```
<Объект>.emit(<Сигнал>[, <Данные через запятую>])
```

В параметре `<Сигнал>` указывается результат выполнения функции `SIGNAL()`. Формат функции:

```
QtCore.SIGNAL("<Название сигнала>([Тип параметров])")
```

Через второй необязательный параметр можно передать дополнительные данные через запятую. Метод всегда вызывается через объект, которому посылается сигнал.

В качестве примера создадим окно с двумя кнопками (листинг 3.5). Для этих кнопок назначим обработчики сигнала `clicked()` (нажатие кнопки). Внутри обработчика щелчка на первой кнопке сгенерируем два сигнала. Первый сигнал будет имитировать нажатие второй кнопки, а второй сигнал будет пользовательским, привязанным ко второй кнопке. Внутри обработчиков выведем сообщения в окно консоли.

Листинг 3.5. Генерация сигнала из программы

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Генерация сигнала из программы")
        self.resize(300, 100)
        self.button1 = QtGui.QPushButton("Нажми меня")
        self.button2 = QtGui.QPushButton("Кнопка 2")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button1)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button2)
        self.connect(self.button2, QtCore.SIGNAL("mysignal"),
                     self.on_mysignal)
    def on_clicked_button1(self):
        print("Нажата кнопка button1")
        # Генерируем сигналы
        self.button2.emit(QtCore.SIGNAL("clicked(bool)"), False)
        self.button2.emit(QtCore.SIGNAL("mysignal"), 10, 20)
    def on_clicked_button2(self):
```

```
        print("Нажата кнопка button2")
    def on_mysignal(self, x, y):
        print("Обработан пользовательский сигнал mysignal()")
        print("x =", x, "y =", y)


if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Результат выполнения после нажатия первой кнопки:

```
Нажата кнопка button1
Нажата кнопка button2
Обработан пользовательский сигнал mysignal()
x = 10 y = 20
```

Назначение обработчиков нажатия кнопки производится обычным образом, а вот назначение обработчика пользовательского сигнала имеет свои отличия. Как видно из примера, после названия пользовательского сигнала нет круглых скобок. В этом случае при генерации сигнала мы можем передать в обработчик произвольное количество значений любого типа. В нашем примере мы передаем два числа:

```
self.button2.emit(QtCore.SIGNAL("mysignal"), 10, 20)
```

При использовании пользовательских сигналов допускается также указывать тип передаваемых данных внутри круглых скобок. Пример назначения обработчика с двумя параметрами:

```
self.connect(self.button2, QtCore.SIGNAL("mysignal(int, int)"),
             self.on_mysignal)
```

Генерация сигнала выглядит так:

```
self.button2.emit(QtCore.SIGNAL("mysignal(int, int)"), 10, 20)
```

Вместо конкретного типа можно указать тип `PyQt_PyObject`. В этом случае при генерации сигнала допускается передавать данные любого типа. Пример назначения обработчика с одним параметром:

```
self.connect(self.button2, QtCore.SIGNAL("mysignal(PyQt_PyObject)"),
             self.on_mysignal)
```

Пример генерации сигнала:

```
self.button2.emit(QtCore.SIGNAL("mysignal(PyQt_PyObject)"), 20)
```

```
self.button2.emit(QtCore.SIGNAL("mysignal(PyQt_PyObject)"), [1, "2"])
```

Сгенерировать сигнал можно не только с помощью метода `emit()`. Некоторые компоненты предоставляют методы, которые посылают сигнал. Например, у кнопок существует метод `click()`. Используя этот метод инструкцию:

```
self.button2.emit(QtCore.SIGNAL("clicked(bool)"), False)
```

можно записать следующим образом:

```
self.button2.click()
```

Более подробно такие методы мы будем рассматривать при изучении конкретных компонентов.

3.4. Новый стиль назначения и удаления обработчиков

В PyQt существует альтернативный способ назначения и удаления обработчиков, а также генерации сигнала. Для назначения обработчика используется следующие форматы метода `connect()`:

```
<Компонент>.<Сигнал>.connect(<Обработчик>[, type=<ConnectionType>])
<Компонент>.<Сигнал>[<Тип>].connect(<Обработчик>[,
                                    type=<ConnectionType>])
```

В параметре `<Компонент>` указывается компонент, для которого назначается обработчик сигнала `<Сигнал>`. Название сигнала указывается без круглых скобок. Если одноименные сигналы отличаются типом параметров, то внутри квадратных скобок можно указать для какого типа назначается обработчик. В параметре `<Обработчик>` передается ссылка на функцию или метод класса, а в именованном параметре `type` можно дополнительно указать тип соединения. По умолчанию параметр `type` имеет значение `AutoConnection`. Пример:

```
self.button1.clicked.connect(self.on_clicked_button1)
self.button2.clicked[bool].connect(self.on_clicked_button2)
```

Для удаления обработчика используется следующие форматы метода `disconnect()`:

```
<Компонент>.<Сигнал>.disconnect([<Обработчик>])
<Компонент>.<Сигнал>[<Тип>].disconnect([<Обработчик>])
```

Если параметр `<Обработчик>` не указан, то удаляются все обработчики, назначенные ранее, в противном случае удаляется только указанный обработчик. Пример:

```
self.button1.clicked.disconnect()
self.button2.clicked[bool].disconnect(self.on_clicked_button2)
```

Для генерации сигнала используется следующие форматы метода `emit()`:

```
<Компонент>.<Сигнал>.emit([<Данные>])
<Компонент>.<Сигнал>[<Тип>].emit([<Данные>])
```

Пример генерации сигнала:

```
self.button2.clicked.emit(False)
self.button2.clicked[bool].emit(False)
```

Для регистрации пользовательских сигналов используется функция `pyqtSignal()` из модуля `QtCore`. Формат функции:

```
<Объект сигнала> = pyqtSignal(*types[, name])
```

В параметре `types` через запятую задаются названия типов данных в Python, например, `bool` или `int`, которые принимает метод:

```
mysignal = QtCore.pyqtSignal(int, name="mysignal")
```

При указании типа данных C++ его название необходимо указать в виде строки:

```
mysignal = QtCore.pyqtSignal("QString", name="mysignal")
```

Если метод не принимает параметров, то параметр `types` не указывается. Сигнал может иметь несколько перегруженных версий. В этом случае типы параметров указываются внутри квадратных скобок. Пример сигнала передающего данные типа `int` или `str`:

```
mysignal = QtCore.pyqtSignal([int], [str], name="mysignal")
```

В именованном параметре `name` можно передать название сигнала в виде строки. Это название будет использоваться вместо названия метода. Если параметр `name` не задан, то название сигнала будет совпадать с названием метода. Метод возвращает объект сигнала.

Создадим окно с тремя кнопками и назначим обработчики разными способами (листинг 3.6). При нажатии первой кнопки имитируем нажатие второй кнопки. При нажатии третьей кнопки удалим все обработчики и сделаем кнопку неактивной.

Листинг 3.6. Альтернативный способ назначения и удаления обработчиков

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    mysignal = QtCore.pyqtSignal([int], [str], name="mysignal")
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Обработка сигналов")
        self.resize(300, 100)
```

```
        self.button1 = QtGui.QPushButton("Нажми меня")
        self.button2 = QtGui.QPushButton("Кнопка 2")
        self.button3 = QtGui.QPushButton("Удалить обработчики")
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        vbox.addWidget(self.button3)
        self.setLayout(vbox)
        # Назначение обработчиков
        self.button1.clicked.connect(self.on_clicked_button1)
        self.button2.clicked[bool].connect(self.on_clicked_button2)
        self.button3.clicked.connect(self.on_clicked_button3)
        self.mysignal[int].connect(self.on_mysignal)
        self.mysignal[str].connect(self.on_mysignal)
    def on_clicked_button1(self, status):
        print("Нажата кнопка button1", status)
        # Генерация сигнала
        self.button2.clicked[bool].emit(False)
        self.mysignal[int].emit(10)
        self.mysignal[str].emit("строка")
    def on_clicked_button2(self, status):
        print("Нажата кнопка button2", status)
    def on_clicked_button3(self):
        print("Нажата кнопка button3")
        # Удаление обработчиков
        self.button1.clicked.disconnect()
        self.button2.clicked[bool].disconnect(
                                    self.on_clicked_button2)
        self.mysignal[int].disconnect()
        self.mysignal[str].disconnect()
        self.button3.setEnabled(False)
    def on_mysignal(self, n):
        print("Обработан пользовательский сигнал\n n =", n)

if __name__ == "__main__":
```

```
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Чтобы пользовательский метод сделать слотом, необходимо перед реализацией метода добавить декоратор `@pyqtSlot()`. Обратите внимание на то, что класс, в котором находится метод, должен быть наследником класса `QObject` или наследником класса, который в свою очередь наследует класс `QObject`. Формат декоратора:

```
@QtCore.pyqtSlot(*types, name=None, result=None)
```

В параметре `types` через запятую задаются названия типов данных в Python, например, `bool` или `int`, которые принимает метод. При указании типа данных C++ его название необходимо указать в виде строки. Если метод не принимает параметров, то параметр `types` не указывается. В именованном параметре `name` можно передать название слота в виде строки. Это название будет использоваться вместо названия метода. Если параметр `name` не задан, то название слота будет совпадать с названием метода. Именованный параметр `result` предназначен для указания типа данных, возвращаемых методом. Если параметр не задан, то метод ничего не возвращает. Чтобы создать перегруженную версию слота декоратор указывается последовательно несколько раз с разными типами данных. Пример использования декоратора `@pyqtSlot()` приведен в листинге 3.7.

Листинг 3.7. Использование декоратора @pyqtSlot()

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import sys

class MyClass(QtCore.QObject):
    def __init__(self):
        QtCore.QObject.__init__(self)
    @QtCore.pyqtSlot()
    def on_clicked(self):
        print("Кнопка нажата. Слот on_clicked()")
    @QtCore.pyqtSlot(bool, name="myslot")
    def on_clicked2(self, status):
        print("Кнопка нажата. Слот myslot(bool)", status)
```

```
obj = MyClass()
app = QtGui.QApplication(sys.argv)
button = QtGui.QPushButton("Нажми меня")
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked()"),
                       obj, QtCore.SLOT("on_clicked()"))
QtCore.QObject.connect(button, QtCore.SIGNAL("clicked(bool)"),
                       obj, QtCore.SLOT("myslot(bool)"))
button.show()
sys.exit(app.exec_())
```

3.5. Передача данных в обработчик

При назначении обработчика в метод `connect()` передается ссылка на функцию или метод. Если после названия функции указать внутри круглых скобок какой-либо параметр, то это приведет к вызову функции и вместо ссылки будет передан результат выполнения функции, что приведет к ошибке. Передать данные в обработчик можно следующими способами:

- ➔ указать lambda-функцию, внутри которой вызывается обработчик с параметром. Пример передачи числа `10`:

  ```
  self.connect(self.button1, QtCore.SIGNAL("clicked()"),
               lambda : self.on_clicked_button1(10))
  ```

  Если значение вычисляется динамически и передается с помощью переменной, то переменную следует указывать как значение по умолчанию в lambda-функции, иначе внутри lambda-функции сохраняется ссылка на переменную, а не ее значение. Пример:

  ```
  y = 10
  self.connect(self.button1, QtCore.SIGNAL("clicked()"),
               lambda x=y: self.on_clicked_button1(x))
  ```

- ➔ передать ссылку на экземпляр класса, внутри которого определен метод `__call__()`. Значение указывается при создании экземпляра класса. Пример класса:

  ```
  class MyClass():
      def __init__(self, x=0):
          self.x = x
      def __call__(self):
          print("x =", self.x)
  ```

Пример передачи параметра в обработчик:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             MyClass(10))
```

➔ передать ссылку на обработчик и данные в функцию `partial()` из модуля `functools`. Формат функции:

```
from functools import partial
partial(func[, *args][, **keywords])
```

Пример передачи параметра в обработчик:

```
self.connect(self.button1, QtCore.SIGNAL("clicked()"),
             partial(self.on_clicked_button1, 10))
```

Если при генерации сигнала передается предопределенное значение, то оно будет доступно в обработчике после остальных параметров. Назначим обработчик сигнала `clicked(bool)` и передадим число:

```
self.connect(self.button1, QtCore.SIGNAL("clicked(bool)"),
             partial(self.on_clicked_button1, 10))
```

Обработчик будет иметь следующий вид:

```
def on_clicked_button1(self, x, status):
    print("Нажата кнопка button1", x, status)
```

Результат выполнения:

```
Нажата кнопка button1 10 False
```

3.6. Использование таймеров

Таймеры позволяют через определенный интервал времени выполнять метод с предопределенным названием `timerEvent()`. Для назначения таймера используется метод `startTimer()` из класса `QObject`. Формат метода:

```
<Id> = <Объект>.startTimer(<Интервал>)
```

Метод `startTimer()` возвращает идентификатор таймера, с помощью которого можно остановить таймер. Параметр `<Интервал>` задает промежуток времени в миллисекундах, по истечении которого выполняется метод `timerEvent()`. Формат метода `timerEvent()`:

```
timerEvent(self, <Объект класса QTimerEvent>)
```

Внутри метода `timerEvent()` можно получить идентификатор таймера с помощью метода `timerId()` объекта класса `QTimerEvent`. Формат метода:

```
<Id> = <Объект класса QTimerEvent>.timerId()
```

Минимальное значение интервала зависит от операционной системы. Если в параметре <Интервал> указать значение 0, то таймер будет срабатывать много раз при отсутствии других необработанных событий.

Чтобы остановить таймер, необходимо воспользоваться методом killTimer() из класса QObject. Формат метода:

```
<Объект>.killTimer(<Id>)
```

В качестве параметра указывается идентификатор, возвращаемый методом startTimer().

Создадим часы в окне, которые будут отображать текущее системное время с точностью до секунды, и добавим возможность запуска и остановки часов с помощью кнопок (листинг 3.8).

Листинг 3.8. Вывод времени в окне с точностью до секунды

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import time

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Часы в окне")
        self.resize(200, 100)
        self.timer_id = 0
        self.label = QtGui.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button1 = QtGui.QPushButton("Запустить")
        self.button2 = QtGui.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button1)
```

```
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button2)
    def on_clicked_button1(self):
        self.timer_id = self.startTimer(1000) # 1 секунда
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        if self.timer_id:
            self.killTimer(self.timer_id)
            self.timer_id = 0
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def timerEvent(self, event):
        self.label.setText(time.strftime("%H:%M:%S"))

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Вместо методов `startTimer()` и `killTimer()` можно воспользоваться классом `QTimer` из модуля `QtCore`. Конструктор класса имеет следующий формат:

```
<Объект> = QTimer(parent=None)
```

Методы класса:

- `setInterval(<Интервал>)` — задает промежуток времени в миллисекундах, по истечении которого генерируется сигнал `timeout()`. Минимальное значение интервала зависит от операционной системы. Если в параметре `<Интервал>` указать значение `0`, то таймер будет срабатывать много раз при отсутствии других необработанных сигналов;
- `start([<Интервал>])` — запускает таймер. В необязательном параметре можно указать промежуток времени в миллисекундах. Если параметр не указан, то используется значение, возвращаемое методом `interval()`;
- `stop()` — останавливает таймер;

- `setSingleShot(<Флаг>)` — если в параметре указано значение `True`, то таймер будет срабатывать только один раз, в противном случае — многократно;
- `interval()` — возвращает установленный интервал;
- `timerId()` — возвращает идентификатор таймера или значение `-1`;
- `isSingleShot()` — возвращает значение `True`, если таймер будет срабатывать только один раз, и `False` — в противном случае;
- `isActive()` — возвращает значение `True`, если таймер генерирует сигналы, и `False` — в противном случае.

Переделаем предыдущий пример и используем класс `QTimer` вместо методов `startTimer()` и `killTimer()` (листинг 3.9).

Листинг 3.9. Использование класса QTimer

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui
import time

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.setWindowTitle("Использование класса QTimer")
        self.resize(200, 100)
        self.label = QtGui.QLabel("")
        self.label.setAlignment(QtCore.Qt.AlignCenter)
        self.button1 = QtGui.QPushButton("Запустить")
        self.button2 = QtGui.QPushButton("Остановить")
        self.button2.setEnabled(False)
        vbox = QtGui.QVBoxLayout()
        vbox.addWidget(self.label)
        vbox.addWidget(self.button1)
        vbox.addWidget(self.button2)
        self.setLayout(vbox)
        self.connect(self.button1, QtCore.SIGNAL("clicked()"),
                     self.on_clicked_button1)
        self.connect(self.button2, QtCore.SIGNAL("clicked()"),
```

```
                    self.on_clicked_button2)
        self.timer = QtCore.QTimer()
        self.connect(self.timer, QtCore.SIGNAL("timeout()"),
                    self.on_timeout);
    def on_clicked_button1(self):
        self.timer.start(1000) # 1 секунда
        self.button1.setEnabled(False)
        self.button2.setEnabled(True)
    def on_clicked_button2(self):
        self.timer.stop()
        self.button1.setEnabled(True)
        self.button2.setEnabled(False)
    def on_timeout(self):
        self.label.setText(time.strftime("%H:%M:%S"))


if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Кроме перечисленных методов в классе `QTimer` определен статический метод `singleShot()`, предназначенный для вызова указанной функции, метода или слота через определенный промежуток времени. Таймер срабатывает только один раз. Форматы метода:

```
QtCore.QTimer.singleShot(<Интервал>, <Обработчик>)
QtCore.QTimer.singleShot(<Интервал>, <Объект>, <Слот>)
```

Примеры использования статического метода `singleShot()`:

```
QtCore.QTimer.singleShot(1000, self.on_timeout)
QtCore.QTimer.singleShot(1000, QtGui.qApp, QtCore.SLOT("quit()"))
```

3.7. Перехват всех событий

В предыдущих разделах мы рассмотрели обработку сигналов, которые позволяют обмениваться сообщениями между компонентами. Обработка внешних событий, например, нажатий клавиш, осуществляется несколько иначе. Чтобы обработать

событие необходимо наследовать класс и переопределить в нем метод со специальным названием, например, чтобы обработать нажатие клавиши следует переопределить метод `keyPressEvent()`. Специальные методы принимают объект, содержащий детальную информацию о событии, например, код нажатой клавиши. Все эти объекты являются наследниками класса `QEvent` и наследуют следующие методы:

- `accept()` — устанавливает флаг, который является признаком согласия с дальнейшей обработкой события, например, если в методе `closeEvent()` вызывать метод `accept()` через объект события, то окно будет закрыто. Флаг обычно устанавливается по умолчанию;
- `ignore()` — сбрасывает флаг, что является запретом на дальнейшую обработку события, например, если в методе `closeEvent()` вызывать метод `ignore()` через объект события, то окно закрыто не будет;
- `setAccepted(<Флаг>)` — если в качестве параметра указано значение `True`, то флаг будет установлен (аналогично вызову метода `accept()`), а если `False` — то сброшен (аналогично вызову метода `ignore()`);
- `isAccepted()` — возвращает текущее состояние флага;
- `registerEventType([<Число>])` — позволяет зарегистрировать пользовательский тип события. Метод возвращает идентификатор зарегистрированного события. В качестве параметра можно указать значение в пределах от `QEvent.User` (`1000`) до `QEvent.MaxUser` (`65535`). Метод является статическим;
- `spontaneous()` — возвращает `True`, если событие сгенерировано системой, и `False`, если событие сгенерировано внутри программы искусственно;
- `type()` — возвращает тип события. Перечислим основные типы событий (полный список смотрите в документации по классу `QEvent`):
  - `0` — нет события;
  - `1` — `Timer` — событие таймера;
  - `2` — `MouseButtonPress` — нажата кнопка мыши;
  - `3` — `MouseButtonRelease` — отпущена кнопка мыши;
  - `4` — `MouseButtonDblClick` — двойной щелчок мышью;
  - `5` — `MouseMove` — перемещение мыши;
  - `6` — `KeyPress` — клавиша на клавиатуре нажата;
  - `7` — `KeyRelease` — клавиша на клавиатуре отпущена;
  - `8` — `FocusIn` — получен фокус ввода с клавиатуры;

- 9 — FocusOut — потерян фокус ввода с клавиатуры;
- 10 — Enter — указатель мыши входит в область компонента;
- 11 — Leave — указатель мыши покидает область компонента;
- 12 — Paint — перерисовка компонента;
- 13 — Move — позиция компонента изменилась;
- 14 — Resize — изменился размер компонента;
- 17 — Show — компонент отображен;
- 18 — Hide — компонент скрыт;
- 19 — Close — окно закрыто;
- 24 — WindowActivate — окно стало активным;
- 25 — WindowDeactivate — окно стало неактивным;
- 26 — ShowToParent — дочерний компонент отображен;
- 27 — HideToParent — дочерний компонент скрыт;
- 31 — Wheel — прокручено колесико мыши;
- 40 — Clipboard — содержимое буфера обмена изменено;
- 60 — DragEnter — указатель мыши входит в область компонента при операции перетаскивания;
- 61 — DragMove — производится операция перетаскивания;
- 62 — DragLeave — указатель мыши покидает область компонента при операции перетаскивания;
- 63 — Drop — операция перетаскивания завершена;
- 68 — ChildAdded — добавлен дочерний компонент;
- 69 — ChildPolished — производится настройка дочернего компонента;
- 71 — ChildRemoved — удален дочерний компонент;
- 74 — PolishRequest — компонент настроен;
- 75 — Polish — производится настройка компонента;
- 82 — ContextMenu — событие контекстного меню;
- 99 — ActivationChange — изменился статус активности окна верхнего уровня;
- 103 — WindowBlocked — окно блокировано модальным окном;

- 104 — WindowUnblocked — текущее окно разблокировано после закрытия модального окна;
- 105 — WindowStateChange — статус окна изменился;
- 121 — ApplicationActivate — приложение стало доступно пользователю;
- 122 — ApplicationDeactivate — приложение стало недоступно пользователю;
- 1000 — User — пользовательское событие;
- 65535 — MaxUser — максимальный идентификатор пользовательского события.

Перехват всех событий осуществляется с помощью метода с предопределенным названием event(self, <event>). Через параметр <event> доступен объект с дополнительной информацией о событии. Этот объект отличается для разных типов событий, например, для события MouseButtonPress объект будет экземпляром класса QMouseEvent, а для события KeyPress — экземпляром класса QKeyEvent. Какие методы содержат эти классы, мы рассмотрим в следующих разделах.

Внутри метода event() следует вернуть значение True, если событие принято, и False — в противном случае. Если возвращается значение True, то родительский компонент не получит событие. Чтобы продолжить распространение события необходимо вызвать метод event() базового класса и передать ему текущий объект события. Обычно это делается так:

```
return QtGui.QWidget.event(self, e)
```

В этом случае пользовательский класс является наследником класса QWidget и переопределяет метод event(). Если вы наследуете другой класс, то следует вызывать метод именно этого класса. Например, при наследовании класса QLabel инструкция будет выглядеть так:

```
return QtGui.QLabel.event(self, e)
```

Пример перехвата нажатия клавиши, щелчка мышью и закрытия окна показан в листинге 3.10.

Листинг 3.10. Перехват всех событий

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
```

```
        self.resize(300, 100)
    def event(self, e):
        if e.type() == QtCore.QEvent.KeyPress:
            print("Нажата клавиша на клавиатуре")
            print("Код:", e.key(), ", текст:", e.text())
        elif e.type() == QtCore.QEvent.Close:
            print("Окно закрыто")
        elif e.type() == QtCore.QEvent.MouseButtonPress:
            print("Щелчок мышью. Координаты:", e.x(), e.y())
        return QtGui.QWidget.event(self, e) # Отправляем дальше

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

3.8. События окна

Перехватывать все события следует только в самом крайнем случае. В обычных ситуациях нужно использовать методы, предназначенные для обработки определенного события, например, чтобы обработать закрытие окна достаточно переопределить метод `closeEvent()`. Какие методы требуется переопределять для обработки событий окна мы сейчас и рассмотрим.

3.8.1. Изменение состояния окна

Отследить изменение состояния окна (сворачивание, разворачивание, сокрытие и отображение) позволяют следующие методы:

- `changeEvent(self, <event>)` — вызывается при изменении состояния окна, приложения или компонента. Иными словами, метод вызывается не только при изменении статуса окна, но и при изменении заголовка окна, палитры, статуса активности окна верхнего уровня, языка, локали и др. (полный список смотрите в документации). При обработке события `WindowStateChange` через параметр `<event>` доступен экземпляр класса `QWindowStateChangeEvent`. Этот класс содержит только метод `oldState()`, с помощью которого можно получить предыдущий статус окна;

➔ `showEvent(self, <event>)` — вызывается при отображении компонента. Через параметр `<event>` доступен экземпляр класса `QShowEvent`;

➔ `hideEvent(self, <event>)` — вызывается при сокрытии компонента. Через параметр `<event>` доступен экземпляр класса `QHideEvent`.

Для примера выведем текущее состояние окна в консоль при сворачивании, разворачивании, сокрытии и отображении окна (листинг 3.11).

Листинг 3.11. Отслеживание состояния окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def changeEvent(self, e):
        if e.type() == QtCore.QEvent.WindowStateChange:
            if self.isMinimized():
                print("Окно свернуто")
            elif self.isMaximized():
                print("Окно раскрыто до максимальных размеров")
            elif self.isFullScreen():
                print("Полноэкранный режим")
            elif self.isActiveWindow():
                print("Окно находится в фокусе ввода")
        QtGui.QWidget.changeEvent(self, e) # Отправляем дальше
    def showEvent(self, e):
        print("Окно отображено")
        QtGui.QWidget.showEvent(self, e)   # Отправляем дальше
    def hideEvent(self, e):
        print("Окно скрыто")
        QtGui.QWidget.hideEvent(self, e)   # Отправляем дальше

if __name__ == "__main__":
```

```
import sys
app = QtGui.QApplication(sys.argv)
window = MyWindow()
window.show()
sys.exit(app.exec_())
```

3.8.2. Изменение положения окна и его размеров

При перемещении окна и изменении размеров вызываются следующие методы:

- `moveEvent(self, <event>)` — вызывается непрерывно при перемещении окна. Через параметр `<event>` доступен экземпляр класса `QMoveEvent`. Получить координаты окна позволяют следующие методы из класса `QMoveEvent`:
  - `pos()` — возвращает экземпляр класса `QPoint` с текущими координатами;
  - `oldPos()` — возвращает экземпляр класса `QPoint` с предыдущими координатами.
- `resizeEvent(self, <event>)` — вызывается непрерывно при изменении размеров окна. Через параметр `<event>` доступен экземпляр класса `QResizeEvent`. Получить размеры окна позволяют следующие методы из класса `QResizeEvent`:
  - `size()` — возвращает экземпляр класса `QSize` с текущими размерами;
  - `oldSize()` — возвращает экземпляр класса `QSize` с предыдущими размерами.

Пример обработки изменения положения окна и его размера показан в листинге 3.12.

Листинг 3.12. Изменение положения окна и его размера

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def moveEvent(self, e):
        print("x = {0}; y = {1}".format(e.pos().x(), e.pos().y()))
        QtGui.QWidget.moveEvent(self, e)   # Отправляем дальше
    def resizeEvent(self, e):
```

```
        print("w = {0}; h = {1}".format(e.size().width(),
                                        e.size().height()))
        QtGui.QWidget.resizeEvent(self, e) # Отправляем дальше


if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

3.8.3. Перерисовка окна или его части

Когда компонент (или его часть) становится видимым, требуется выполнить перерисовку компонента или только его части. В этом случае вызывается метод с названием `paintEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QPaintEvent`, который содержит следующие методы:

- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, которую требуется перерисовать;
- `region()` — возвращает экземпляр класса `QRegion` с регионом, требующим перерисовки.

С помощью этих методов можно получать координаты области, которая, например, была ранее перекрыта другим окном и теперь оказалась в зоне видимости. Перерисовывая только область, а не весь компонент можно достичь более эффективного расходования ресурсов компьютера. Следует также заметить, что в целях эффективности, последовательность событий перерисовки может быть объединена в одно событие с общей областью перерисовки.

В некоторых случаях перерисовку окна необходимо выполнить вне зависимости от внешних действий системы или пользователя, например, при изменении каких-либо значений нужно обновить график. Вызвать событие перерисовки компонента позволяют следующие методы из класса `QWidget`:

- `repaint()` — незамедлительно вызывает метод `paintEvent()` для перерисовки компонента, при условии, что компонент не скрыт и обновление не запрещено с помощью метода `setUpdatesEnabled()`. Форматы метода:

```
repaint()
repaint(<X>, <Y>, <Ширина>, <Высота>)
repaint(<Экземпляр класса QRect>)
repaint(<Экземпляр класса QRegion>)
```

➔ `update()` — посылает сообщение о необходимости перерисовки компонента, при условии, что компонент не скрыт и обновление не запрещено. Событие будет обработано на следующей итерации основного цикла приложения. Если посылаются сразу несколько сообщений, то они объединяются в одно сообщение. Благодаря этому можно избежать неприятного мерцания. Метод `update()` предпочтительнее использовать вместо метода `repaint()`. Форматы метода:

```
update()
update(<X>, <Y>, <Ширина>, <Высота>)
update(<Экземпляр класса QRect>)
update(<Экземпляр класса QRegion>)
```

3.8.4. Предотвращение закрытия окна

При нажатии кнопки **Закрыть** в заголовке окна или при вызове метода `close()` вызывается метод `closeEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QCloseEvent`. Чтобы предотвратить закрытие окна необходимо вызвать метод `ignore()` через объект события, в противном случае — метод `accept()`.

В качестве примера при нажатии кнопки **Закрыть** в заголовке окна выведем стандартное диалоговое окно с запросом подтверждения закрытия окна (листинг 3.13). Если пользователь нажимает кнопку **Yes**, то закроем окно, а если кнопку **No** или просто закрывает диалоговое окно, то прервем закрытие окна.

Листинг 3.13. Обработка закрытия окна

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
    def closeEvent(self, e):
        result = QtGui.QMessageBox.question(self,
                        "Подтверждение закрытия окна",
                        "Вы действительно хотите закрыть окно?",
                        QtGui.QMessageBox.Yes | QtGui.QMessageBox.No,
                        QtGui.QMessageBox.No)
        if result == QtGui.QMessageBox.Yes:
```

```
            e.accept()
            QtGui.QWidget.closeEvent(self, e)
        else:
            e.ignore()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

3.9. События клавиатуры

События клавиатуры очень часто обрабатываются внутри приложений. Например, при нажатии клавиши <F1> можно вывести справочную информацию, при нажатии клавиши <Enter> в однострочном текстовом поле перенести фокус ввода на другой компонент и т. д. Рассмотрим события клавиатуры подробно.

3.9.1. Установка фокуса ввода

В один момент времени только один компонент (или вообще не одного) может иметь фокус ввода. Для управления фокусом ввода предназначены следующие методы из класса `QWidget`:

➔ `setFocus([<Причина>])` — устанавливает фокус ввода, если компонент находится в активном окне. В параметре `<Причина>` можно указать причину изменения фокуса ввода. Указываются следующие атрибуты из класса `QtCore.Qt` или соответствующие им значения:

- `MouseFocusReason` — `0` — фокус изменен с помощью мыши;
- `TabFocusReason` — `1` — нажата клавиша <Tab>;
- `BacktabFocusReason` — `2` — нажата комбинация клавиш <Shift>+<Tab>;
- `ActiveWindowFocusReason` — `3` — окно стало активным или неактивным;
- `PopupFocusReason` — `4` — открыто или закрыто всплывающее окно;
- `ShortcutFocusReason` — `5` — нажата комбинация клавиш быстрого доступа;
- `MenuBarFocusReason` — `6` — фокус изменился из-за меню;
- `OtherFocusReason` — `7` — другая причина;

- `clearFocus()` — убирает фокус ввода с компонента;
- `hasFocus()` — возвращает значение `True`, если компонент находится в фокусе ввода, и `False` — в противном случае;
- `focusWidget()` — возвращает ссылку на последний компонент, для которого вызывался метод `setFocus()`. Для компонентов верхнего уровня возвращается ссылка на компонент, который получит фокус после того, как окно станет активным;
- `setFocusProxy(<QWidget>)` — позволяет указать ссылку на компонент, который будет получать фокус ввода вместо текущего компонента;
- `focusProxy()` — возвращает ссылку на компонент, который обрабатывает фокус ввода вместо текущего компонента. Если компонента нет, то метод возвращает значение `None`;
- `focusNextChild()` — находит следующий компонент, которому можно передать фокус и передает фокус. Аналогично нажатию клавиши <Tab>. Метод возвращает значение `True`, если компонент найден, и `False` — в противном случае;
- `focusPreviousChild()` — находит предыдущий компонент, которому можно передать фокус и передает фокус. Аналогично нажатию комбинации клавиш <Shift>+<Tab>. Метод возвращает значение `True`, если компонент найден, и `False` — в противном случае;
- `focusNextPrevChild(<Флаг>)` — если в параметре указано значение `True`, то метод аналогичен методу `focusNextChild()`. Если в параметре указано значение `False`, то метод аналогичен методу `focusPreviousChild()`. Метод возвращает значение `True`, если компонент найден, и `False` — в противном случае;
- `setTabOrder(<Компонент1>, <Компонент2>)` — позволяет задать последовательность смены фокуса при нажатии клавиши <Tab>. Метод является статическим. В параметре `<Компонент2>` указывается ссылка на компонент, на который переместится фокус с компонента `<Компонент1>`. Если компонентов много, то метод вызывается несколько раз. Пример указания цепочки перехода `widget1 -> widget2 -> widget3 -> widget4`:

```
QtGui.QWidget.setTabOrder(widget1, widget2)
QtGui.QWidget.setTabOrder(widget2, widget3)
QtGui.QWidget.setTabOrder(widget3, widget4)
```

- `setFocusPolicy(<Способ>)` — задает способ получения фокуса компонентом. В качестве параметра указываются следующие атрибуты из класса `QtCore.Qt` или соответствующие им значения:
  - `NoFocus` — `0` — компонент не может получать фокус;
  - `TabFocus` — `1` — получает фокус с помощью клавиши <Tab>;

- `ClickFocus` — `2` — получает фокус с помощью щелчка мышью;
- `StrongFocus` — `11` — получает фокус с помощью клавиши <Tab> и щелчка мышью;
- `WheelFocus` — `15` — получает фокус с помощью клавиши <Tab>, щелчка мышью и колесика мыши;

➔ `focusPolicy()` — возвращает текущий способ получения фокуса.

➔ `grabKeyboard()` — захватывает ввод с клавиатуры. Другие компоненты не будут получать события клавиатуры, пока не будет вызван метод `releaseKeyboard()`;

➔ `releaseKeyboard()` — освобождает захваченный ранее ввод с клавиатуры.

Получить ссылку на компонент, находящийся в фокусе ввода, позволяет статический метод `focusWidget()` из класса `QApplication`. Если ни один компонент не имеет фокуса ввода, то метод возвращает значение `None`. Не путайте этот метод с одноименным методом из класса `QWidget`.

Обработать получение и потерю фокуса ввода позволяют следующие методы:

➔ `focusInEvent(self, <event>)` — вызывается при получении фокуса ввода;

➔ `focusOutEvent(self, <event>)` — вызывается при потере фокуса ввода.

Через параметр `<event>` доступен экземпляр класса `QFocusEvent`, который содержит следующие методы:

➔ `gotFocus()` — возвращает значение `True`, если тип события `QEvent.FocusIn`, и `False` — в противном случае;

➔ `lostFocus()` — возвращает значение `True`, если тип события `QEvent.FocusOut`, и `False` — в противном случае;

➔ `reason()` — возвращает причину установки фокуса. Значение аналогично значению параметра `<Причина>` в методе `setFocus()`.

Создадим окно с кнопкой и двумя однострочными полями ввода (листинг 3.14). Для полей ввода обработаем получение и потерю фокуса ввода, а при нажатии кнопки установим фокус ввода на второе поле. Кроме того, зададим последовательность перехода при нажатии клавиши <Tab>.

Листинг 3.14. Установка фокуса ввода

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui


class MyLineEdit(QtGui.QLineEdit):
```

```
    def __init__(self, id, parent=None):
        QtGui.QLineEdit.__init__(self, parent)
        self.id = id
    def focusInEvent(self, e):
        print("Получен фокус полем", self.id)
        QtGui.QLineEdit.focusInEvent(self, e)
    def focusOutEvent(self, e):
        print("Потерян фокус полем", self.id)
        QtGui.QLineEdit.focusOutEvent(self, e)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.button = QtGui.QPushButton("Установить фокус на поле 2")
        self.line1 = MyLineEdit(1)
        self.line2 = MyLineEdit(2)
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.button)
        self.vbox.addWidget(self.line1)
        self.vbox.addWidget(self.line2)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
        # Задаем порядок обхода с помощью клавиши <Tab>
        QtGui.QWidget.setTabOrder(self.line1, self.line2)
        QtGui.QWidget.setTabOrder(self.line2, self.button)
    def on_clicked(self):
        self.line2.setFocus()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

3.9.2. Назначение клавиш быстрого доступа

Клавиши быстрого доступа (иногда их также называют "горячими" клавишами) позволяют установить фокус ввода с помощью нажатия специальной клавиши (например, <Alt> или <Ctrl>) и какой-либо дополнительной клавиши. Если после нажатия клавиш быстрого доступа в фокусе окажется кнопка (или пункт меню), то она будет нажата.

Чтобы задать клавиши быстрого доступа следует в тексте надписи указать символ `&` перед буквой. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент окажется в фокусе ввода. Некоторые компоненты, например, текстовое поле, не имеют надписи. Чтобы задать клавиши быстрого доступа для таких компонентов необходимо отдельно создать надпись и связать ее с компонентом с помощью метода `setBuddy(<Компонент>)` из класса `QLabel`. Если же создание надписи не представляется возможным, то можно воспользоваться следующими методами из класса `QWidget`:

- `grabShortcut(<Клавиши>[, <Контекст>])` — регистрирует клавиши быстрого доступа и возвращает идентификатор, с помощью которого можно управлять ими в дальнейшем. В параметре `<Клавиши>` указывается экземпляр класса `QKeySequence`. Создать экземпляр этого класса для комбинации <Alt>+<E> можно, например, так:

  ```
  QtGui.QKeySequence.mnemonic("&e")
  QtGui.QKeySequence("Alt+e")
  QtGui.QKeySequence(QtCore.Qt.ALT + QtCore.Qt.Key_E)
  ```

  В параметре `<Контекст>` можно указать атрибуты `WidgetShortcut`, `WidgetWithChildrenShortcut`, `WindowShortcut` (значение по умолчанию) и `ApplicationShortcut` из класса `QtCore.Qt`;

- `releaseShortcut(<ID>)` — удаляет комбинацию с идентификатором `<ID>`;

- `setShortcutEnabled(<ID>[, <Флаг>])` — если в качестве параметра `<Флаг>` указано значение `True` (значение по умолчанию), то клавиши быстрого доступа с идентификатором `<ID>` разрешены. Значение `False` запрещает использование клавиш быстрого доступа.

При нажатии клавиш быстрого доступа генерируется событие `QEvent.Shortcut`, которое можно обработать в методе `event(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QShortcutEvent`, который содержит следующие методы:

- `shortcutId()` — возвращает идентификатор комбинации клавиш;

- `isAmbiguous()` — возвращает значение `True`, если событие отправлено сразу нескольким компонентам, и `False` — в противном случае;

- `key()` — возвращает экземпляр класса `QKeySequence`.

Создадим окно с надписью, двумя однострочными текстовыми полями и кнопкой (листинг 3.15). Для первого текстового поля назначим комбинацию клавиш (<Alt>+<B>) через надпись, а для второго поля (<Alt>+<E>) — с помощью метода `grabShortcut()`. Для кнопки назначим комбинацию клавиш (<Alt>+<У>) обычным образом через надпись на кнопке.

Листинг 3.15. Назначение клавиш быстрого доступа

```
# -*- coding: utf-8 -*-
from PyQt4 import QtCore, QtGui

class MyLineEdit(QtGui.QLineEdit):
    def __init__(self, parent=None):
        QtGui.QLineEdit.__init__(self, parent)
        self.id = None
    def event(self, e):
        if e.type() == QtCore.QEvent.Shortcut:
            if self.id == e.shortcutId():
                self.setFocus(QtCore.Qt.ShortcutFocusReason)
                return True
        return QtGui.QLineEdit.event(self, e)

class MyWindow(QtGui.QWidget):
    def __init__(self, parent=None):
        QtGui.QWidget.__init__(self, parent)
        self.resize(300, 100)
        self.label = QtGui.QLabel("Устано&вить фокус на поле 1")
        self.lineEdit1 = QtGui.QLineEdit()
        self.label.setBuddy(self.lineEdit1)
        self.lineEdit2 = MyLineEdit()
        self.lineEdit2.id = self.lineEdit2.grabShortcut(
                            QtGui.QKeySequence.mnemonic("&e"))
        self.button = QtGui.QPushButton("&Убрать фокус с поля 1")
        self.vbox = QtGui.QVBoxLayout()
        self.vbox.addWidget(self.label)
        self.vbox.addWidget(self.lineEdit1)
```

```
        self.vbox.addWidget(self.lineEdit2)
        self.vbox.addWidget(self.button)
        self.setLayout(self.vbox)
        self.button.clicked.connect(self.on_clicked)
    def on_clicked(self):
        self.lineEdit1.clearFocus()

if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    window = MyWindow()
    window.show()
    sys.exit(app.exec_())
```

Помимо рассмотренных способов для назначения клавиш быстрого доступа можно воспользоваться классом `QShortcut`. В этом случае назначение клавиш для второго текстового поля будет выглядеть так:

```
self.lineEdit2 = QtGui.QLineEdit()
self.shc = QtGui.QShortcut(QtGui.QKeySequence.mnemonic("&e"), self)
self.shc.setContext(QtCore.Qt.WindowShortcut)
self.shc.activated.connect(self.lineEdit2.setFocus)
```

Назначить комбинацию быстрых клавиш позволяет также класс `QAction`. Назначение клавиш для второго текстового поля выглядит следующим образом:

```
self.lineEdit2 = QtGui.QLineEdit()
self.act = QtGui.QAction(self)
self.act.setShortcut(QtGui.QKeySequence.mnemonic("&e"))
self.act.triggered.connect(self.lineEdit2.setFocus)
self.addAction(self.act)
```

3.9.3. Нажатие и отпускание клавиши на клавиатуре

При нажатии и отпускании клавиши вызываются следующие методы:

- `keyPressEvent(self, <event>)` — вызывается при нажатии клавиши на клавиатуре. Если клавишу удерживать нажатой, то событие генерируется постоянно, пока клавиша не будет отпущена.
- `keyReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой клавиши.

Через параметр <event> доступен экземпляр класса QKeyEvent, который позволяет получить дополнительную информацию о событии. Класс QKeyEvent содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу QKeyEvent):

➔ key() — возвращает код нажатой клавиши. Пример определения клавиши:

```
if e.key() == QtCore.Qt.Key_B:
    print("Нажата клавиша <B>")
```

➔ text() — возвращает текстовое представление символа в кодировке Unicode. Если клавиша является специальной, то возвращается пустая строка;

➔ modifiers() — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с клавишей. Может содержать значения следующих атрибутов из класса QtCore.Qt (или комбинацию значений):

- NoModifier — модификаторы не нажаты;
- ShiftModifier — нажата клавиша <Shift>;
- ControlModifier — нажата клавиша <Ctrl>;
- AltModifier — нажата клавиша <Alt>;
- MetaModifier — нажата клавиша <Meta>;
- KeypadModifier;
- GroupSwitchModifier.

Пример определения модификатора <Shift>:

```
if e.modifiers() & QtCore.Qt.ShiftModifier:
    print("Нажат модификатор <Shift>")
```

➔ isAutoRepeat() — возвращает значение True, если событие вызвано повторно удержанием клавиши нажатой, и False — в противном случае;

➔ matches(<QKeySequence.StandardKey>) — возвращает значение True, если нажата специальная комбинация клавиш, соответствующая указанному значению, и False — в противном случае. В качестве значения указываются атрибуты из класса QKeySequence, например, QKeySequence.Copy для комбинации клавиш <Ctrl>+<C> (копировать). Полный список атрибутов смотрите в документации по классу QKeySequence. Пример:

```
if e.matches(QtGui.QKeySequence.Copy):
    print("Нажата комбинация <Ctrl>+<C>")
```

При обработке нажатия клавиш следует учитывать, что:

- компонент должен иметь возможность принимать фокус ввода. Некоторые компоненты не могут принимать фокус ввода по умолчанию, например, надпись. Чтобы изменить способ получения фокуса следует воспользоваться методом `setFocusPolicy(<Способ>)`, который мы рассматривали в *разд. 3.9.1*;
- чтобы захватить эксклюзивный ввод с клавиатуры следует воспользоваться методом `grabKeyboard()`, а чтобы освободить ввод — методом `releaseKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме клавиши <Tab> и комбинации <Shift>+<Tab>. Эти клавиши используются для передачи фокуса следующему и предыдущему компоненту соответственно. Перехватить нажатие этих клавиш можно только в методе `event(self, <event>)`;
- если событие обработано, то нужно вызвать метод `accept()` через объект события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

3.10. События мыши

События мыши обрабатываются не реже, чем события клавиатуры. С помощью специальных методов можно обработать нажатие и отпускание кнопки мыши, перемещение указателя, а также вхождение указателя в область компонента и выхода из этой области. В зависимости от ситуации можно изменить вид указателя, например, при выполнении длительной операции отобразить указатель в виде песочных часов. В этом разделе мы рассмотрим изменение вида указателя мыши, как для отдельного компонента, так и для всего приложения.

3.10.1. Нажатие и отпускание кнопки мыши

При нажатии и отпускании кнопки мыши вызываются следующие методы:

- `mousePressEvent(self, <event>)` — вызывается при нажатии кнопки мыши;
- `mouseReleaseEvent(self, <event>)` — вызывается при отпускании ранее нажатой кнопки мыши;
- `mouseDoubleClickEvent(self, <event>)` — вызывается при двойном щелчке мышью в области компонента. Следует учитывать, что двойному щелчку предшествуют другие события. Последовательность событий при двойном щелчке выглядит так:

```
Событие MouseButtonPress
Событие MouseButtonRelease
Событие MouseButtonDblClick
```

`Событие MouseButtonPress`

`Событие MouseButtonRelease`

Задать интервал двойного щелчка позволяет метод `setDoubleClickInterval()` из класса `QApplication`. Получить текущее значение интервала можно с помощью метода `doubleClickInterval()`.

Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, который позволяет получить дополнительную информацию о событии. Класс `QMouseEvent` содержит следующие методы:

- `x()` и `y()` — возвращают координаты по осям `X` и `Y` соответственно в пределах области компонента;
- `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами в пределах области компонента;
- `posF()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах области компонента;
- `globalX()` и `globalY()` — возвращают координаты по осям `X` и `Y` соответственно в пределах экрана;
- `globalPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана;
- `button()` — позволяет определить, какая кнопка мыши вызвала событие. Возвращает значение одного из следующих атрибутов из класса `QtCore.Qt`:
  - `NoButton` — `0` — кнопки не нажаты. Это значение возвращается методом `button()` при перемещении указателя мыши;
  - `LeftButton` — `1` — нажата левая кнопка мыши;
  - `RightButton` — `2` — нажата правая кнопка мыши;
  - `MidButton` и `MiddleButton` — `4` — нажата средняя кнопка мыши;
  - `XButton1` — `8`;
  - `XButton2` — `16`;
- `buttons()` — позволяет определить все кнопки, которые нажаты одновременно. Возвращает комбинацию значений атрибутов `LeftButton`, `RightButton` и `MidButton`. Пример определения кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:
    print("Нажата левая кнопка мыши")
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в *разд. 3.9.3*.

Если событие обработано, то нужно вызвать метод `accept()` через объект события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

Если для компонента атрибут `WA_NoMousePropagation` из класса `QtCore.Qt` установлен в истинное значение, то событие мыши не будет передаваться родительскому компоненту. Значение атрибута можно изменить с помощью метода `setAttribute()`:

```
self.setAttribute(QtCore.Qt.WA_NoMousePropagation, True)
```

По умолчанию событие мыши перехватывает компонент, над которым произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне компонента следует захватить мышь с помощью метода `grabMouse()`. Освободить захваченную ранее мышь позволяет метод `releaseMouse()`.

3.10.2. Перемещение указателя

Чтобы обработать перемещение указателя мыши необходимо переопределить метод `mouseMoveEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QMouseEvent`, который позволяет получить дополнительную информацию о событии. Методы этого класса мы уже рассматривали в предыдущем разделе. Следует учитывать, что метод `button()` при перемещении мыши возвращает значение атрибута `QtCore.Qt.NoButton`.

По умолчанию метод `mouseMoveEvent()` вызывается только в том случае, если при перемещении удерживается нажатой какая-либо кнопка мыши. Это сделано специально, чтобы не создавать лишних событий при обычном перемещении указателя мыши. Если необходимо обрабатывать любые перемещения указателя в пределах компонента, то следует вызвать метод `setMouseTracking()` из класса `QWidget` и передать ему значение `True`. Чтобы обработать все перемещения внутри окна нужно дополнительно захватить мышь с помощью метода `grabMouse()`.

Метод `pos()` объекта события возвращает позицию точки в системе координат текущего компонента. Чтобы преобразовать координаты точки в систему координат родительского компонента или в глобальную систему координат следует воспользоваться следующими методами из класса `QWidget`:

- `mapToGlobal(<QPoint>)` — преобразует координаты точки из системы координат компонента в глобальную систему координат. Метод возвращает экземпляр класса `QPoint`;
- `mapFromGlobal(<QPoint>)` — преобразует координаты точки из глобальной системы координат в систему координат компонента. Метод возвращает экземпляр класса `QPoint`;

- `mapToParent(<QPoint>)` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента. Если компонент не имеет родителя, то метод аналогичен методу `mapToGlobal()`. Метод возвращает экземпляр класса `QPoint`;
- `mapFromParent(<QPoint>)` — преобразует координаты точки из системы координат родительского компонента в систему координат данного компонента. Если компонент не имеет родителя, то метод аналогичен методу `mapFromGlobal()`. Метод возвращает экземпляр класса `QPoint`;
- `mapTo(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента `<QWidget>`. Метод возвращает экземпляр класса `QPoint`;
- `mapFrom(<QWidget>, <QPoint>)` — преобразует координаты точки из системы координат родительского компонента `<QWidget>` в систему координат данного компонента. Метод возвращает экземпляр класса `QPoint`.

3.10.3. Наведение и выведение указателя

Обработать наведение указателя мыши на компонент и выведение указателя позволяют следующие методы:

- `enterEvent(self, <event>)` — вызывается при наведении указателя мыши на область компонента;
- `leaveEvent(self, <event>)` — вызывается, когда указатель мыши покидает область компонента.

Через параметр `<event>` доступен экземпляр класса `QEvent`. Этот параметр не несет никакой дополнительной информации. Вполне достаточно знать, что указатель попал в область компонента или покинул ее.

3.10.4. Прокрутка колесика мыши

Некоторые мыши комплектуются колесиком, которое обычно используется для управления прокруткой некоторой области. Обработать поворот этого колесика позволяет метод `wheelEvent(self, <event>)`. Через параметр `<event>` доступен экземпляр класса `QWheelEvent`, который позволяет получить дополнительную информацию о событии. Класс `QWheelEvent` содержит следующие методы:

- `delta()` — возвращает расстояние поворота колесика. Положительное значение означает, что колесико повернуто вперед от пользователя, а отрицательное значение — что поворот осуществлен в обратном направлении;

- `orientation()` — возвращает ориентацию, в виде значения одного из следующих атрибутов из класса `QtCore.Qt`:
  - `Horizontal` — 1 — по горизонтали;
  - `Vertical` — 2 — по вертикали;
- `x()` и `y()` — возвращают координаты указателя в момент события по осям `X` и `Y` соответственно в пределах области компонента;
- `pos()` — возвращает экземпляр класса `QPoint` с целочисленными координатами указателями в момент события в пределах области компонента;
- `globalX()` и `globalY()` — возвращают координаты указателя в момент события по осям `X` и `Y` соответственно в пределах экрана;
- `globalPos()` — возвращает экземпляр класса `QPoint` с координатами указателя в момент события в пределах экрана;
- `buttons()` — позволяет определить кнопки, которые нажаты одновременно с поворотом колесика. Возвращает комбинацию значений атрибутов `LeftButton`, `RightButton` и `MidButton`. Пример определения кнопки мыши:

```
if e.buttons() & QtCore.Qt.LeftButton:
    print("Нажата левая кнопка мыши")
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в *разд. 3.9.3*.

Если событие обработано, то нужно вызвать метод `accept()` через объект события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

3.10.5. Изменение внешнего вида указателя мыши

Для изменения внешнего вида указателя мыши при вхождении указателя в область компонента предназначены следующие методы из класса `QWidget`:

- `setCursor(<Курсор>)` — задает внешний вид указателя мыши для компонента. В качестве параметра указывается экземпляр класса `QCursor` или следующие атрибуты из класса `QtCore.Qt`: `ArrowCursor` (стандартная стрелка), `UpArrowCursor` (стрелка, направленная вверх), `CrossCursor` (крестообразный указатель), `WaitCursor` (песочные часы), `IBeamCursor` (I-образный указатель), `SizeVerCursor` (стрелки, направленные вверх и вниз), `SizeHorCursor` (стрелки, направленные влево и вправо), `SizeBDiagCursor`, `SizeFDiagCursor`, `SizeAllCursor` (стрелки, направленные вверх, вниз, влево и вправо), `BlankCursor` (пустой указатель), `SplitVCursor`, `SplitHCursor`, `PointingHandCursor`

(указатель в виде руки), `ForbiddenCursor` (перечеркнутый круг), `OpenHandCursor` (разжатая рука), `ClosedHandCursor` (сжатая рука), `WhatsThisCursor` (стрелка с вопросительным знаком) и `BusyCursor` (стрелка с песочными часами). Пример:

```
self.setCursor(QtCore.Qt.WaitCursor)
```

- `unsetCursor()` — отменяет установку указателя для компонента. В результате внешний вид указателя мыши будет наследоваться от родительского компонента;
- `cursor()` — возвращает экземпляр класса `QCursor` с текущим курсором.

Управлять текущим видом курсора для всего приложения сразу можно с помощью следующих статических методов из класса `QApplication`:

- `setOverrideCursor(<Курсор>)` — задает внешний вид указателя мыши для всего приложения. В качестве параметра указывается экземпляр класса `QCursor` или специальные атрибуты из класса `QtCore.Qt`. Для отмены установки необходимо вызвать метод `restoreOverrideCursor()`;
- `restoreOverrideCursor()` — отменяет изменение внешнего вида курсора для всего приложения. Пример:

```
QtGui.QApplication.setOverrideCursor(QtCore.Qt.WaitCursor)
# Выполняем длительную операцию
QtGui.QApplication.restoreOverrideCursor()
```

- `changeOverrideCursor(<Курсор>)` — изменяет внешний вид указателя мыши для всего приложения. Если до вызова этого метода не вызывался метод `setOverrideCursor()`, то указанное значение игнорируется. В качестве параметра указывается экземпляр класса `QCursor` или специальные атрибуты из класса `QtCore.Qt`;
- `overrideCursor()` — возвращает экземпляр класса `QCursor` с текущим курсором или значение `None`.

Изменять внешний вид указателя мыши для всего приложения принято на небольшой промежуток времени, обычно на время выполнения какой-либо операции, в процессе которой приложение не может нормально реагировать на действия пользователя. Чтобы информировать об этом пользователя указатель принято выводить в виде песочных часов (атрибут `WaitCursor`).

Метод `setOverrideCursor()` может быть вызван несколько раз. В этом случае курсоры помещаются в стек. Каждый вызов метода `restoreOverrideCursor()` удаляет последний курсор, добавленный в стек. Для нормальной работы приложения необходимо вызывать методы `setOverrideCursor()` и `restoreOverrideCursor()` одинаковое количество раз.

Класс `QCursor` позволяет создать объект курсора с изображением любой формы. Чтобы загрузить изображение следует передать путь к файлу конструктору класса `QPixmap`.

Чтобы создать объект курсора необходимо передать конструктору класса `QCursor` в первом параметре экземпляр класса `QPixmap`, а во втором и третьем параметрах — координаты "горячей" точки. Пример создания и установки пользовательского курсора:

```
self.setCursor(QtGui.QCursor(QtGui.QPixmap("cursor.png"), 0, 0))
```

Класс `QCursor` содержит также два статических метода:

- `pos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши относительно экрана. Пример:

  ```
  print(QtGui.QCursor.pos().x(), QtGui.QCursor.pos().y())
  ```

- `setPos()` — позволяет задать позицию указателя мыши. Метод имеет два формата: `setPos(<X>, <Y>)` и `setPos(<QPoint>)`.

3.11. Технология drag & drop

Технология drag & drop позволяет обмениваться данными различных типов между компонентами как одного приложения, так и разных приложений, путем перетаскивания и сбрасывания объектов с помощью мыши. Типичным примером использования технологии служит перемещение файлов в программе Проводник в Windows. Чтобы переместить файл в другой каталог достаточно нажать левую кнопку мыши над значком файла и, не отпуская кнопку, перетащить файл на значок каталога, а затем отпустить кнопку мыши. Если необходимо скопировать файл, а не переместить, то следует дополнительно удерживать нажатой клавишу <Ctrl>.

3.11.1. Запуск перетаскивания

Операция перетаскивания состоит из двух частей. Первая часть запускает процесс, а вторая часть обрабатывает момент сброса объекта. Обе части могут обрабатываться как одним приложением, так и двумя разными приложениями. Запуск перетаскивания осуществляется следующим образом:

- внутри метода `mousePressEvent()` запоминаются координаты щелчка левой кнопкой мыши;
- внутри метода `mousePressEvent()` вычисляется пройденное расстояние или измеряется время операции. Это необходимо сделать, чтобы предотвратить случайное перетаскивание. Управлять задержкой позволяют следующие статические методы класса `QApplication`;
  - `startDragDistance()` — возвращает минимальное расстояние, после прохождения которого можно запускать операцию перетаскивания;
  - `setStartDragDistance(<Дистанция>)` — задает расстояние;

- `startDragTime()` — возвращает время задержки в миллисекундах перед запуском операции перетаскивания;
- `setStartDragTime(<Время>)` — задает время задержки;

➔ если пройдено минимальное расстояние или истек минимальный промежуток времени, то создается экземпляр класса `QDrag` и вызывается метод `exec_()`, который после завершения операции возвращает действие, выполненное с данными (например, данные скопированы или перемещены).

Создать экземпляр класса `QDrag` можно так:

```
<Объект> = QtGui.QDrag(<Ссылка на компонент>)
```

Класс `QDrag` содержит следующие методы:

➔ `exec_()` — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. Метод имеет два формата:

```
exec_(<Действия>=MoveAction)
exec_(<Действия>, <Действие по умолчанию>)
```

В параметре `<Действия>` указывается комбинация допустимых действий, а в параметре `<Действие по умолчанию>` — действие, которое используется, если не нажаты клавиши-модификаторы. Возможные действия могут быть заданы следующими атрибутами из класса `QtCore.Qt`: `CopyAction` (`1`; копирование), `MoveAction` (`2`; перемещение), `LinkAction` (`4`; ссылка), `IgnoreAction` (`0`; действие игнорировано), `TargetMoveAction` (`32770`). Пример:

```
act = drag.exec_(QtCore.Qt.MoveAction | QtCore.Qt.CopyAction,
                 QtCore.Qt.MoveAction)
```

➔ `start(<Действия>=CopyAction)` — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. В качестве параметра указывается комбинация допустимых действий;

➔ `setMimeData(<QMimeData>)` — позволяет задать перемещаемые данные. В качестве значения указывается экземпляр класса `QMimeData`. Пример передачи текста:

```
data = QtCore.QMimeData()
data.setText("Перетаскиваемый текст")
drag = QtGui.QDrag(self)
drag.setMimeData(data)
```

➔ `mimeData()` — возвращает экземпляр класса `QMimeData` с перемещаемыми данными;

- `setPixmap(<QPixmap>)` — задает изображение, которое будет перемещаться вместе с указателем мыши. В качестве параметра указывается экземпляр класса `QPixmap`. Пример:

```
drag.setPixmap(QtGui.QPixmap("pixmap.png"))
```

- `pixmap()` — возвращает экземпляр класса `QPixmap` с изображением, которое перемещается вместе с указателем;
- `setHotSpot(<QPoint>)` — задает координаты "горячей" точки на перемещаемом изображении. В качестве параметра указывается экземпляр класса `QPoint`. Пример:

```
drag.setHotSpot(QtCore.QPoint(20, 20))
```

- `hotSpot()` — возвращает экземпляр класса `QPoint` с координатами "горячей" точки на перемещаемом изображении;
- `setDragCursor(<QPixmap>, <Действие>)` — позволяет изменить внешний вид указателя мыши для действия, указанного во втором параметре. В первом параметре указывается экземпляр класса `QPixmap`, а во втором параметре — действия `CopyAction`, `MoveAction` или `LinkAction`. Если в первом параметре указан пустой объект класса `QPixmap`, то это позволит удалить ранее установленный вид указателя для действия. Пример изменения указателя для перемещения:

```
drag.setDragCursor(QtGui.QPixmap("cursor.png"),
                   QtCore.Qt.MoveAction)
```

- `source()` — возвращает ссылку на компонент-источник;
- `target()` — возвращает ссылку на компонент-приемник или значение `None`, если компонент находится в другом приложении.

Класс `QDrag` поддерживает два сигнала:

- `actionChanged(Qt::DropAction)` — генерируется при изменении действия;
- `targetChanged(QWidget *)` — генерируется при изменении принимающего компонента.

Пример назначения обработчиков сигналов:

```
self.connect(drag, QtCore.SIGNAL("actionChanged(Qt::DropAction)"),
             self.on_action_changed)
self.connect(drag, QtCore.SIGNAL("targetChanged(QWidget *)"),
             self.on_target_changed)
```

3.11.2. Класс QMimeData

Перемещаемые данные и сведения о MIME-типе должны быть представлены классом `QMimeData`. Экземпляр этого класса необходимо передать в метод `setMimeData()` класса `QDrag`. Создание экземпляра класса `QMimeData` выглядит так:

```
data = QtCore.QMimeData()
```

Класс `QMimeData` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QMimeData`):

- `setText(<Текст>)` — устанавливает текстовые данные (MIME-тип `text/plain`). Пример указания значения:
  ```
  data.setText("Перетаскиваемый текст")
  ```
- `text()` — возвращает текстовые данные (MIME-тип `text/plain`);
- `hasText()` — возвращает значение `True`, если объект содержит текстовые данные (MIME-тип `text/plain`), и `False` — в противном случае;
- `setHtml(<HTML-текст>)` — устанавливает текстовые данные в формате HTML (MIME-тип `text/html`). Пример указания значения:
  ```
  data.setHtml("<b>Перетаскиваемый HTML-текст</b>")
  ```
- `html()` — возвращает текстовые данные в формате HTML (MIME-тип `text/html`);
- `hasHtml()` — возвращает значение `True`, если объект содержит текстовые данные в формате HTML (MIME-тип `text/html`), и `False` — в противном случае;
- `setUrls(<Список URI-адресов>)` — устанавливает список URI-адресов (MIME-тип `text/uri-list`). В качестве значения указывается список с экземплярами класса `QUrl`. С помощью этого MIME-типа можно обработать перетаскивание файлов. Пример указания значения:
  ```
  data.setUrls([QtCore.QUrl("http://google.ru/")])
  ```
- `urls()` — возвращает список URI-адресов (MIME-тип `text/uri-list`). Пример получения первого URI-адреса:
  ```
  uri = e.mimeData().urls()[0].toString()
  ```
- `hasUrls()` — возвращает значение `True`, если объект содержит список URI-адресов (MIME-тип `text/uri-list`), и `False` — в противном случае;
- `setImageData(<Объект изображения>)` — устанавливает изображение (MIME-тип `application/x-qt-image`). В качестве значения можно указать, например, экземпляр класса `QImage` или `QPixmap`. Пример указания значения:
  ```
  data.setImageData(QtGui.QImage("pixmap.png"))
  data.setImageData(QtGui.QPixmap("pixmap.png"))
  ```

- `imageData()` — возвращает объект изображения (тип возвращаемого объекта зависит от типа объекта, указанного в методе `setImageData()`);
- `hasImage()` — возвращает значение `True`, если объект содержит изображение (MIME-тип `application/x-qt-image`), и `False` — в противном случае;
- `setData(<MIME-тип>, <Данные>)` — позволяет установить данные пользовательского MIME-типа. В первом параметре указывается MIME-тип в виде строки, а во втором параметре — экземпляр класса `QByteArray` с данными. Метод можно вызвать несколько раз с различными MIME-типами. Пример передачи текстовых данных:

```
data.setData("text/plain",
             QtCore.QByteArray(bytes("Данные", "utf-8")))
```

- `data(<MIME-тип>)` — возвращает экземпляр класса `QByteArray` с данными, соответствующими указанному MIME-типу;
- `hasFormat(<MIME-тип>)` — возвращает значение `True`, если объект содержит данные в указанном MIME-типе, и `False` — в противном случае;
- `formats()` — возвращает список с поддерживаемыми объектом MIME-типами;
- `removeFormat(<MIME-тип>)` — удаляет данные, соответствующие указанному MIME-типу;
- `clear()` — удаляет все данные и информацию о MIME-типе.

Если необходимо перетаскивать данные какого-либо специфического типа, то нужно наследовать класс `QMimeData` и переопределить методы `retrieveData()` и `formats()`. За подробной информацией по этому вопросу обращайтесь к документации.

3.11.3. Обработка сброса

Прежде чем обрабатывать перетаскивание и сбрасывание объекта необходимо сообщить системе, что компонент может обрабатывать эти события. Для этого внутри конструктора компонента следует вызвать метод `setAcceptDrops()` из класса `QWidget` и передать ему значение `True`:

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

Обработка перетаскивания и сброса объекта выполняется следующим образом:

- внутри метода `dragEnterEvent()` проверяется MIME-тип перетаскиваемых данных и действие. Если компонент способен обработать сброс этих данных и соглашается с предложенным действием, то необходимо вызвать метод `acceptProposedAction()` через объект события. Если нужно изменить действие,

то методу `setDropAction()` передается новое действие, а затем вызывается метод `accept()`, а не метод `acceptProposedAction()`;

- если необходимо ограничить область сброса некоторым участком компонента, то можно дополнительно определить метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области компонента. При согласии со сбрасыванием следует вызвать метод `accept()`, которому можно передать экземпляр класса `QRect` с координатами и размером участка. Если параметр указан, то при перетаскивании внутри участка метод `dragMoveEvent()` повторно вызываться не будет;
- внутри метода `dropEvent()` производится обработка сброса.

Обработать события, возникающие при перетаскивании и сбрасывании объектов, позволяют следующие методы:

- `dragEnterEvent(self, <event>)` — вызывается, когда перетаскиваемый объект входит в область компонента. Через параметр `<event>` доступен экземпляр класса `QDragEnterEvent`;
- `dragLeaveEvent(self, <event>)` — вызывается, когда перетаскиваемый объект покидает область компонента. Через параметр `<event>` доступен экземпляр класса `QDragLeaveEvent`;
- `dragMoveEvent(self, <event>)` — вызывается при перетаскивании объекта внутри области компонента. Через параметр `<event>` доступен экземпляр класса `QDragMoveEvent`;
- `dropEvent(self, <event>)` — вызывается при сбрасывании объекта в области компонента. Через параметр `<event>` доступен экземпляр класса `QDropEvent`.

Класс `QDragLeaveEvent` наследует класс `QEvent` и не несет никакой дополнительной информации. Достаточно просто знать, что перетаскиваемый объект покинул область компонента. Цепочка наследования остальных классов выглядит так:

`(QEvent, QMimeSource)` — `QDropEvent` — `QDragMoveEvent` — `QDragEnterEvent`

Класс `QMimeSource` признан устаревшим и используется только в целях совместимости с предыдущими версиями. Вместо класса `QMimeSource` следует использовать класс `QMimeData`. Класс `QDragEnterEvent` не содержит собственных методов, но наследует все методы классов `QDropEvent` и `QDragMoveEvent`.

Класс `QDropEvent` содержит следующие методы:

- `mimeData()` — возвращает экземпляр класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе;
- `pos()` — возвращает экземпляр класса `QPoint` с координатами сбрасывания объекта;

- `possibleActions()` — возвращает комбинацию возможных действий при сбрасывании. Пример определения значений:

```
if e.possibleActions() & QtCore.Qt.MoveAction:
    print("MoveAction")
if e.possibleActions() & QtCore.Qt.CopyAction:
    print("CopyAction")
```

- `proposedAction()` — возвращает действие по умолчанию при сбрасывании;
- `acceptProposedAction()` — устанавливает флаг готовности принять перемещаемые данные и согласия с действием, возвращаемым методом `proposedAction()`. Метод `acceptProposedAction()` (или метод `accept()`) необходимо вызвать внутри метода `dragEnterEvent()`, иначе метод `dropEvent()` вызван не будет;
- `setDropAction(<Действие>)` — позволяет изменить действие при сбрасывании. После изменения действия следует вызвать метод `accept()`, а не `acceptProposedAction()`;
- `dropAction()` — возвращает действие, которое должно быть выполнено при сбрасывании. Возвращаемое значение может не совпадать со значением, возвращаемым методом `proposedAction()`, если действие было изменено с помощью метода `setDropAction()`;
- `keyboardModifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы уже рассматривали в *разд. 3.9.3*;
- `mouseButtons()` — позволяет определить кнопки мыши, которые нажаты;
- `source()` — возвращает ссылку на компонент внутри приложения, являющийся источником события, или значение `None`.

Теперь рассмотрим методы класса `QDragMoveEvent`:

- `accept([<QRect>])` — устанавливает флаг, который является признаком согласия с дальнейшей обработкой события. В качестве параметра можно указать экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой сбрасывание будет принято;
- `ignore([<QRect>])` — сбрасывает флаг, что является запретом на дальнейшую обработку события. В качестве параметра можно указать экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой сбрасывание выполнить нельзя;

➔ `answerRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой произойдет сбрасывание, если событие будет принято.

Некоторые компоненты в PyQt по умолчанию поддерживают технологию drag & drop, например, в однострочное текстовое поле можно перетащить текст из другого приложения. Поэтому, прежде чем изобретать свой "велосипед", убедитесь, что поддержка технологии в компоненте не реализована.

3.12. Работа с буфером обмена

Помимо технологии drag & drop для обмена данными между приложениями используется буфер обмена. Одно приложение помещает данные в буфер обмена, а второе приложение (или то же самое) может извлечь их из буфера. Получить ссылку на глобальный объект буфера обмена позволяет статический метод `clipboard()` из класса `QApplication`:

```
clipboard = QtGui.QApplication.clipboard()
```

Класс `QClipboard` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QClipboard`):

➔ `setText(<Текст>[, <Режим>])` — копирует текст в буфер обмена;

➔ `text([<Режим>])` — возвращает текст из буфера обмена или пустую строку;

➔ `text(<Тип>[, <Режим>])` — возвращает кортеж из двух строк. Первый элемент кортежа содержит текст из буфера, а второй элемент — название типа. В параметре `<Тип>` могут быть указаны значения `"plain"` (простой текст), `"html"` (текст в формате HTML) или пустая строка;

➔ `setMimeData(<QMimeData>[, <Режим>])` — позволяет сохранить в буфере данные любого типа. В качестве первого параметра указывается экземпляр класса `QMimeData`. Этот класс мы уже рассматривали при изучении технологии drag & drop (см. *разд. 3.11.2*);

➔ `mimeData([<Режим>])` — возвращает экземпляр класса `QMimeData`;

➔ `clear([<Режим>])` — очищает буфер обмена.

В необязательном параметре `<Режим>` могут быть указаны атрибуты `Clipboard` (используется по умолчанию), `Selection` или `FindBuffer` из класса `QClipboard`.

Отследить изменение состояния буфера обмена позволяет сигнал `dataChanged()`. Назначить обработчик этого сигнала можно так:

```
self.connect(QtGui.qApp.clipboard(), QtCore.SIGNAL("dataChanged()"),
             self.on_change_clipboard)
```

3.13. Фильтрация событий

События можно перехватывать еще до того как они будут переданы компоненту. Для этого необходимо создать класс, который является наследником класса `QObject`, и переопределить метод `eventFilter(self, <Объект>, <event>)`. Через параметр `<Объект>` доступна ссылка на компонент, а через параметр `<event>` — объект с дополнительной информацией о событии. Этот объект отличается для разных типов событий, например, для события `MouseButtonPress` объект будет экземпляром класса `QMouseEvent`, а для события `KeyPress` — экземпляром класса `QKeyEvent`. Внутри метода `eventFilter()` следует вернуть значение `True`, если событие не должно быть передано дальше, и `False` — в противном случае. Пример фильтра, перехватывающего нажатие клавиши <B>:

```
class MyFilter(QtCore.QObject):
    def __init__(self, parent=None):
        QtCore.QObject.__init__(self, parent)
    def eventFilter(self, obj, e):
        if e.type() == QtCore.QEvent.KeyPress:
            if e.key() == QtCore.Qt.Key_B:
                print("Событие от клавиши <B> не дойдет до компонента")
                return True
        return QtCore.QObject.eventFilter(self, obj, e)
```

Далее следует создать экземпляр этого класса, передав в конструктор ссылку на компонент, а затем вызвать метод `installEventFilter()`, указав в качестве параметра ссылку на объект фильтра. Пример установки фильтра для надписи:

```
self.label.installEventFilter(MyFilter(self.label))
```

Метод `installEventFilter()` можно вызвать несколько раз, передавая ссылку на разные объекты фильтров. В этом случае первым будет вызван фильтр, который был добавлен последним. Кроме того, один фильтр можно установить сразу в нескольких компонентах. Ссылка на компонент, который является источником события, доступна через второй параметр метода `eventFilter()`.

Удалить фильтр позволяет метод `removeEventFilter(<Фильтр>)`. Если фильтр не был установлен, то ничего не происходит.

3.14. Искусственные события

Для создания искусственных событий применяются следующие статические методы из класса `QCoreApplication`:

➔ `sendEvent(<QObject>, <QEvent>)` — немедленно посылает событие компоненту и возвращает результат выполнения обработчика;

➔ `postEvent(<QObject>, <QEvent>)` — добавляет событие в очередь. Этот метод является потокобезопасным, следовательно, его можно использовать в многопоточных приложениях для обмена событиями между потоками.

В параметре `<QObject>` указывается ссылка на объект, которому посылается событие, а в параметре `<QEvent>` — объект события. Объект события может быть как экземпляром стандартного класса (например, экземпляром класса `QMouseEvent`), так и экземпляром пользовательского класса, который является наследником класса `QEvent`. Пример отправки события `QEvent.MouseButtonPress` компоненту `label`:

```
e = QtGui.QMouseEvent(QtCore.QEvent.MouseButtonPress,
                      QtCore.QPoint(5, 5), QtCore.Qt.LeftButton,
                      QtCore.Qt.LeftButton, QtCore.Qt.NoModifier)
QtCore.QCoreApplication.sendEvent(self.label, e)
```

Для отправки пользовательского события необходимо создать класс, который наследует класс `QEvent`. Внутри класса следует зарегистрировать пользовательское событие с помощью статического метода `registerEventType()` и сохранить идентификатор события в атрибуте класса. Пример:

```
class MyEvent(QtCore.QEvent):
    idType = QtCore.QEvent.registerEventType()
    def __init__(self, data):
        QtCore.QEvent.__init__(self, MyEvent.idType)
        self.data = data
    def get_data(self):
        return self.data
```

Пример отправки события класса `MyEvent` компоненту `label`:

```
QtCore.QCoreApplication.sendEvent(self.label, MyEvent("512"))
```

Обработать пользовательское событие можно с помощью метода `event(self, <event>)` или `customEvent(self, <event>)`. Пример:

```
def customEvent(self, e):
    if e.type() == MyEvent.idType:
        self.setText("Получены данные: {0}".format(e.get_data()))
```

Глава 4. Размещение нескольких компонентов в окне

При размещении нескольких компонентов в окне обычно возникает вопрос взаимного расположения компонентов, а также их минимальных размеров. Следует помнить, что по умолчанию размеры окна можно изменить, взявшись мышью на границу окна, а значит необходимо перехватывать событие изменения размеров и производить пересчет позиции и размера каждого компонента. Библиотека Qt избавляет нас от лишних проблем и предоставляет множество компонентов-контейнеров, которые производят перерасчет автоматически. Все, что от нас требуется — это выбрать нужный контейнер, добавить туда компоненты в определенном порядке, а затем поместить контейнер в окно или в другой контейнер.

4.1. Абсолютное позиционирование

Прежде чем изучать компоненты-контейнеры рассмотрим возможность абсолютного позиционирования компонентов в окне. Итак, если при создании компонента указана ссылка на родительский компонент, то он выводится в позицию с координатами `(0, 0)`. Иными словами, если мы добавим несколько компонентов, то все они отобразятся на одной и той же позиции. Последний добавленный компонент будет на вершине этой кучи, а остальные компоненты будут видны лишь частично или вообще не видны. Размеры добавляемых компонентов будут соответствовать их содержимому.

Для перемещения компонента можно воспользоваться методом `move()`, а для изменения размеров — методом `resize()`. Выполнить одновременное изменение позиции и размеров позволяет метод `setGeometry()`. Все эти методы, а также множество других методов, позволяющих изменять позицию и размеры, мы уже рассматривали в *разд. 2.3* и *2.4*. Если компонент не имеет родителя, то эти методы изменяют характеристики окна, а если при создании компонента указан родительский компонент, то методы изменяют характеристики только самого компонента.

Для примера выведем внутри окна надпись и кнопку, указав позицию и размеры для каждого компонента (листинг 4.1).

Листинг 4.1. Абсолютное позиционирование

```
# -*- coding: utf-8 -*-
from PyQt4 import QtGui
import sys
```

```
app = QtGui.QApplication(sys.argv)
window = QtGui.QWidget()
window.setWindowTitle("Абсолютное позиционирование")
window.resize(300, 120)
label = QtGui.QLabel("Текст надписи", window)
button = QtGui.QPushButton("Текст на кнопке", window)
label.setGeometry(10, 10, 280, 60)
button.resize(280, 30)
button.move(10, 80)
window.show()
sys.exit(app.exec_())
```

Абсолютное позиционирование имеет следующие недостатки:

- при изменении размеров окна необходимо пересчитывать и изменять характеристики всех компонентов вручную;
- при указании фиксированных размеров, надписи на компонентах могут выходить за пределы компонента. Помните, что в разных операционных системах используются разные стили оформления, в том числе и характеристики шрифта. Подогнав размеры в одной операционной системе можно прийти в ужас при виде приложения в другой операционной системе, где размер шрифта в два раза больше. Поэтому лучше вообще отказаться от указания фиксированных размеров или задавать размер и название шрифта для каждого компонента. Кроме того, приложение может поддерживать несколько языков интерфейса. Длина слов в разных языках отличается, что также станет причиной искажения компонентов.

4.2. Горизонтальное и вертикальное выравнивание

Компоненты-контейнеры (их еще называют *менеджерами компоновки*, *менеджерами геометрии*) лишены недостатков абсолютного позиционирования. При изменении размеров окна производится автоматическое изменение характеристик всех компонентов, добавленных в контейнер. Настройки шрифта при этом также учитываются, поэтому изменение размеров шрифта в два раза приведет только к увеличению компонентов и размеров окна.

Для автоматического выравнивания компонентов используются два класса:

- `QHBoxLayout` — выстраивает все добавляемые компоненты по горизонтали (по умолчанию слева направо). Конструктор класса имеет следующий формат:

```
<Объект> = QHBoxLayout([<Родитель>])
```

➔ QVBoxLayout — выстраивает все добавляемые компоненты по вертикали (по умолчанию сверху вниз). Формат конструктора класса:

```
<Объект> = QVBoxLayout([<Родитель>])
```

Иерархия наследования для классов QHBoxLayout и QVBoxLayout выглядит так:

```
(QObject, QLayoutItem) — QLayout — QBoxLayout — QHBoxLayout
(QObject, QLayoutItem) — QLayout — QBoxLayout — QVBoxLayout
```

Обратите внимание на то, что классы не являются наследниками класса QWidget, следовательно, они не обладают собственным окном и не могут использоваться отдельно. Поэтому контейнеры обязательно должны быть привязаны к родительскому компоненту. Передать ссылку на родительский компонент можно через конструктор классов QHBoxLayout и QVBoxLayout. Кроме того, можно передать ссылку на контейнер в метод setLayout() родительского компонента. После этого все компоненты, добавленные в контейнер, автоматически привязываются к родительскому компоненту. Типичный пример использования класса QHBoxLayout выглядит следующим образом:

```
window = QtGui.QWidget()                # Родительский компонент
button1 = QtGui.QPushButton("1")
button2 = QtGui.QPushButton("2")
hbox = QtGui.QHBoxLayout()              # Создаем контейнер
hbox.addWidget(button1)                 # Добавляем компоненты
hbox.addWidget(button2)
window.setLayout(hbox)                  # Передаем ссылку родителю
```

Добавить компоненты в контейнер и удалить их позволяют следующие методы:

➔ addWidget() — добавляет компонент в конец контейнера. Формат метода:

```
addWidget(<Компонент>[, stretch=0][, alignment=0])
```

В первом параметре указывается ссылка на компонент. Необязательный параметр stretch задает фактор растяжения для ячейки, а параметр alignment — выравнивание компонента внутри ячейки. Два последних параметра можно указывать в порядке следования или по именам в произвольном порядке:

```
hbox.addWidget(button1, 10, QtCore.Qt.AlignRight)
hbox.addWidget(button2, stretch=10)
hbox.addWidget(button3, alignment=QtCore.Qt.AlignRight)
```

➔ insertWidget() — добавляет компонент в указанную позицию контейнера. Формат метода:

```
insertWidget(<Индекс>, <Компонент>[, stretch=0][, alignment=0])
```

Если в первом параметре указано значение `0`, то компонент будет добавлен в начало контейнера. Если указано отрицательное значение, то компонент добавляется в конец контейнера. Другое значение указывает определенную позицию. Остальные параметры аналогичны параметрам метода `addWidget()`. Пример:

```
hbox.addWidget(button1)
hbox.insertWidget(-1, button2) # Добавление в конец
hbox.insertWidget(0, button3)  # Добавление в начало
```

- `removeWidget(<Компонент>)` — удаляет компонент из контейнера;
- `addLayout()` — добавляет другой контейнер в конец текущего контейнера. С помощью этого метода можно вкладывать один контейнер в другой, создавая таким образом структуру любой сложности. Формат метода:

  ```
  addLayout(<Контейнер>[, stretch=0])
  ```

- `insertLayout()` — добавляет другой контейнер в указанную позицию текущего контейнера. Если в первом параметре указано отрицательное значение, то контейнер добавляется в конец. Формат метода:

  ```
  insertLayout(<Индекс>, <Контейнер>[, stretch=0])
  ```

- `addSpacing(<Размер>)` — добавляет пустое пространство указанного размера в конец контейнера. Пример:

  ```
  hbox.addSpacing(100)
  ```

- `insertSpacing(<Индекс>, <Размер>)` — добавляет пустое пространство указанного размера в определенную позицию. Если в первом параметре указано отрицательное значение, то пространство добавляется в конец;
- `addStretch([stretch=0])` — добавляет пустое растягиваемое пространство с нулевым минимальным размером и фактором растяжения `stretch` в конец контейнера. Это пространство можно сравнить с пружиной, вставленной между компонентами, а параметр `stretch` с жесткостью пружины;
- `insertStretch(<Индекс>[, stretch=0])` — метод аналогичен методу `addStretch()`, но добавляет растягиваемое пространство в указанную позицию. Если в первом параметре указано отрицательное значение, то пространство добавляется в конец контейнера.

Параметр `alignment` в методах `addWidget()` и `insertWidget()` задает выравнивание компонента внутри ячейки. В этом параметре можно указать следующие атрибуты из класса `QtCore.Qt` или соответствующие им значения:

- `AlignLeft` — `1` — горизонтальное выравнивание по левому краю;
- `AlignRight` — `2` — горизонтальное выравнивание по правому краю;
- `AlignHCenter` — `4` — горизонтальное выравнивание по центру;

- `AlignJustify` — `8` — заполнение всего пространства;
- `AlignTop` — `32` — вертикальное выравнивание по верхнему краю;
- `AlignBottom` — `64` — вертикальное выравнивание по нижнему краю;
- `AlignVCenter` — `128` — вертикальное выравнивание по центру;
- `AlignCenter` — `AlignVCenter | AlignHCenter` — горизонтальное и вертикальное выравнивание по центру;
- `AlignAbsolute` — `16` — если в методе `setLayoutDirection()` из класса `QWidget` указан атрибут `QtCore.Qt.RightToLeft`, то атрибут `AlignLeft` задает выравнивание по правому краю, а атрибут `AlignRight` — по левому краю. Чтобы атрибут `AlignLeft` всегда соответствовал именно левому краю необходимо указать комбинацию `AlignAbsolute | AlignLeft`. Аналогично следует поступить с атрибутом `AlignRight`.

Можно задавать комбинацию атрибутов. В комбинации допускается указывать только один атрибут горизонтального выравнивания и только один атрибут вертикального выравнивания. Например, комбинация `AlignLeft | AlignTop` задает выравнивание по левому и верхнему краю. Противоречивые значения приводят к непредсказуемым результатам.

Помимо рассмотренных методов, контейнеры поддерживают следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setDirection(<Направление>)` — задает направление вывода компонентов. В параметре можно указать следующие атрибуты из класса `QBoxLayout`:
  - `LeftToRight` — `0` — слева направо (значение по умолчанию для горизонтального контейнера);
  - `RightToLeft` — `1` — справа налево;
  - `TopToBottom` — `2` — сверху вниз (значение по умолчанию для вертикального контейнера);
  - `BottomToTop` — `3` — снизу вверх;
- `setMargin(<Отступ>)` — задает величину отступа от границ контейнера до компонентов;
- `setSpacing(<Расстояние>)` — задает расстояние между компонентами.

4.3. Выравнивание по сетке

Помимо выравнивания компонентов по горизонтали и вертикали существует возможность размещения компонентов внутри ячеек сетки. Для выравнивания компонентов по сетке предназначен класс `QGridLayout`. Иерархия наследования:

```
(QObject, QLayoutItem) — QLayout — QGridLayout
```

Создать экземпляр класса `QGridLayout` можно следующим образом:

```
<Объект> = QGridLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент. Если параметр не указан, то необходимо передать ссылку на сетку в метод `setLayout()` родительского компонента. Типичный пример использования класса `QGridLayout` выглядит так:

```
window = QtGui.QWidget()                # Родительский компонент
button1 = QtGui.QPushButton("1")
button2 = QtGui.QPushButton("2")
button3 = QtGui.QPushButton("3")
button4 = QtGui.QPushButton("4")
grid = QtGui.QGridLayout()              # Создаем сетку
grid.addWidget(button1, 0, 0)           # Добавляем компоненты
grid.addWidget(button2, 0, 1)
grid.addWidget(button3, 1, 0)
grid.addWidget(button4, 1, 1)
window.setLayout(grid)                  # Передаем ссылку родителю
```

Добавить компоненты и удалить их позволяют следующие методы:

➔ `addWidget()` — добавляет компонент в указанную ячейку сетки. Метод имеет следующие форматы:

```
addWidget(<Компонент>, <Строка>, <Столбец>[, alignment=0])
addWidget(<Компонент>, <Строка>, <Столбец>, <Количество строк>,
          <Количество столбцов>[, alignment=0])
```

В первом параметре указывается ссылка на компонент, во втором параметре передается индекс строки, а в третьем — индекс столбца. Нумерация строк и столбцов начинается с нуля. Параметр `<Количество строк>` задает количество объединенных ячеек по вертикали, а параметр `<Количество столбцов>` — по горизонтали. Параметр `alignment` задает выравнивание компонента внутри ячейки. Значения, которые можно указать в этом параметре, мы рассматривали в предыдущем разделе. Пример:

```
grid = QtGui.QGridLayout()
grid.addWidget(button1, 0, 0, alignment=QtCore.Qt.AlignLeft)
grid.addWidget(button2, 0, 1, QtCore.Qt.AlignRight)
grid.addWidget(button3, 1, 0, 1, 2)
```

➔ `addLayout()` — добавляет контейнер в указанную ячейку сетки. Метод имеет следующие форматы:

```
addLayout(<Контейнер>, <Строка>, <Столбец>[, alignment=0])
addLayout(<Контейнер>, <Строка>, <Столбец>, <Количество строк>,
          <Количество столбцов>[, alignment=0])
```

В первом параметре указывается ссылка на контейнер. Остальные параметры аналогичны параметрам метода `addWidget()`.

Класс `QGridLayout` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

➔ `setRowMinimumHeight(<Индекс>, <Высота>)` — задает минимальную высоту строки с индексом `<Индекс>`;

➔ `setColumnMinimumWidth(<Индекс>, <Ширина>)` — задает минимальную ширину столбца с индексом `<Индекс>`;

➔ `setRowStretch(<Индекс>, <Фактор растяжения>)` — задает фактор растяжения для строки с индексом `<Индекс>`;

➔ `setColumnStretch(<Индекс>, <Фактор растяжения>)` — задает фактор растяжения для столбца с индексом `<Индекс>`;

➔ `setMargin(<Отступ>)` — задает величину отступа от границ сетки до компонентов;

➔ `setSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали и вертикали;

➔ `setHorizontalSpacing(<Значение>)` — задает расстояние между компонентами по горизонтали;

➔ `setVerticalSpacing(<Значение>)` — задает расстояние между компонентами по вертикали;

➔ `rowCount()` — возвращает количество строк сетки;

➔ `columnCount()` — возвращает количество столбцов сетки.

4.4. Выравнивание компонентов формы

Класс `QFormLayout` позволяет выравнивать компоненты формы. Контейнер по умолчанию состоит из двух столбцов. Первый столбец предназначен для вывода надписи, а второй столбец — для вывода компонента, например, текстового поля. При этом надпись связывается с компонентом, что позволяет назначать клавиши быстрого доступа, указав символ `&` перед буквой внутри текста надписи. После нажатия комбинации клавиш быстрого доступа (комбинация <Alt>+буква) в фокусе окажется компонент, расположенный справа от надписи. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QFormLayout
```

Создать экземпляр класса `QFormLayout` можно следующим образом:

```
<Объект> = QFormLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент. Если параметр не указан, то необходимо передать ссылку на контейнер в метод `setLayout()` родительского компонента. Типичный пример использования класса `QFormLayout` выглядит так:

```
window = QtGui.QWidget()
lineEdit = QtGui.QLineEdit()
textEdit = QtGui.QTextEdit()
button1 = QtGui.QPushButton("О&тправить")
button2 = QtGui.QPushButton("О&чистить")
hbox = QtGui.QHBoxLayout()
hbox.addWidget(button1)
hbox.addWidget(button2)
form = QtGui.QFormLayout()
form.addRow("&Название:", lineEdit)
form.addRow("&Описание:", textEdit)
form.addRow(hbox)
window.setLayout(form)
```

Класс `QFormLayout` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addRow()` — добавляет строку в конец контейнера. Форматы метода:

```
addRow(<Текст надписи>, <Компонент>)
addRow(<Текст надписи>, <Контейнер>)
addRow(<Компонент1>, <Компонент2>)
addRow(<Компонент>, <Контейнер>)
```

```
addRow(<Компонент>)
addRow(<Контейнер>)
```

В параметре `<Текст надписи>` можно указать текст, внутри которого перед какой-либо буквой указан символ `&`. В этом случае надпись связывается с компонентом, указанном во втором параметре. После нажатия комбинации клавиш быстрого доступа (комбинация <Alt>+буква) этот компонент окажется в фокусе ввода. Если в первом параметре указан экземпляр класса `QLabel`, то связь с компонентом необходимо устанавливать вручную, передав ссылку на компонент в метод `setBuddy()`. Если указан только один параметр, то компонент или контейнер займет сразу два столбца;

➔ `insertRow()` — добавляет строку в указанную позицию контейнера. Если указано отрицательное значение в первом параметре, то компонент добавляется в конец контейнера. Форматы метода:

```
insertRow(<Индекс>, <Текст надписи>, <Компонент>)
insertRow(<Индекс>, <Текст надписи>, <Контейнер>)
insertRow(<Индекс>, <Компонент1>, <Компонент2>)
insertRow(<Индекс>, <Компонент>, <Контейнер>)
insertRow(<Индекс>, <Компонент>)
insertRow(<Индекс>, <Контейнер>)
```

➔ `setFormAlignment(<Режим>)` — задает режим выравнивания формы. Допустимые значения мы рассматривали в *разд. 4.2*. Пример:

```
form.setFormAlignment(
     QtCore.Qt.AlignRight | QtCore.Qt.AlignBottom)
```

➔ `setLabelAlignment(<Режим>)` — задает режим выравнивания надписи. Допустимые значения мы рассматривали в *разд. 4.2*. Пример выравнивания по правому краю:

```
form.setLabelAlignment(QtCore.Qt.AlignRight)
```

➔ `setRowWrapPolicy(<Режим>)` — задает местоположение надписи. В качестве параметра указываются следующие атрибуты из класса `QFormLayout`:

- `DontWrapRows` — `0` — надписи расположены слева от компонентов;
- `WrapLongRows` — `1` — длинные надписи могут находиться выше компонентов, а короткие надписи — слева от компонентов;
- `WrapAllRows` — `2` — надписи расположены выше компонентов;

➔ `setFieldGrowthPolicy(<Режим>)` — задает режим управления размерами компонентов. В качестве параметра указываются следующие атрибуты из класса `QFormLayout`:

- FieldsStayAtSizeHint — 0 — размеры компонентов будут соответствовать рекомендуемым (возвращаемым методом sizeHint());
- ExpandingFieldsGrow — 1 — компоненты, для которых установлена политика изменения размеров QSizePolicy.Expanding или QSizePolicy.MinimumExpanding, будут занимать всю доступную ширину. Размеры остальных компонентов будут соответствовать рекомендуемым;
- AllNonFixedFieldsGrow — 2 — все компоненты (если это возможно) будут занимать всю доступную ширину;

➔ setMargin(<Отступ>) — задает величину отступа от границ формы до компонентов;

➔ setSpacing(<Значение>) — задает расстояние между компонентами по горизонтали и вертикали;

➔ setHorizontalSpacing(<Значение>) — задает расстояние между компонентами по горизонтали;

➔ setVerticalSpacing(<Значение>) — задает расстояние между компонентами по вертикали.

4.5. Классы QStackedLayout и QStackedWidget

Класс QStackedLayout реализует стек компонентов. В один момент времени показывается только один компонент. Иерархия наследования выглядит так:

(QObject, QLayoutItem) — QLayout — QStackedLayout

Создать экземпляр класса QStackedLayout можно следующим образом:

```
<Объект> = QStackedLayout([<Родитель>])
```

В необязательном параметре можно указать ссылку на родительский компонент или контейнер. Если параметр не указан, то необходимо передать ссылку на контейнер в метод setLayout() родительского компонента.

Класс QStackedLayout содержит следующие методы:

➔ setStackingMode(<Режим>) — задает режим отображения компонентов. В параметре могут быть указаны следующие атрибуты из класса QStackedLayout:

- StackOne — 0 — только один компонент видим (значение по умолчанию);
- StackAll — 1 — видны все компоненты;

➔ stackingMode() — возвращает режим отображения компонентов;

➔ addWidget(<Компонент>) — добавляет компонент в конец контейнера. Метод возвращает индекс добавленного компонента;

- `insertWidget(<Индекс>, <Компонент>)` — добавляет компонент в указанную позицию контейнера. Метод возвращает индекс добавленного компонента;
- `removeWidget(<Компонент>)` — удаляет компонент из контейнера;
- `count()` — возвращает количество компонентов внутри контейнера;
- `currentIndex()` — возвращает индекс видимого компонента;
- `currentWidget()` — возвращает ссылку на видимый компонент;
- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;
- `setCurrentIndex(int)` — делает видимым компонент с указанным в параметре индексом. Метод является слотом;
- `setCurrentWidget(QWidget *)` — делает видимым компонент, ссылка на который указана в параметре. Метод является слотом.

Класс `QStackedLayout` содержит следующие сигналы:

- `currentChanged(int)` — генерируется при изменении видимого компонента. Через параметр внутри обработчика доступен индекс нового компонента;
- `widgetRemoved(int)` — генерируется при удалении компонента из контейнера. Через параметр внутри обработчика доступен индекс компонента.

Класс `QStackedWidget` также реализует стек компонентов, но создает новый компонент, а не контейнер. Иерархия наследования выглядит так:

`(QObject, QPaintDevice) — QWidget — QFrame — QStackedWidget`

Создать экземпляр класса `QStackedWidget` можно следующим образом:

```
<Объект> = QStackedWidget([<Родитель>])
```

Класс `QStackedWidget` содержит методы `addWidget()`, `insertWidget()`, `removeWidget()`, `count()`, `currentIndex()`, `currentWidget()`, `widget()`, `setCurrentIndex()` и `setCurrentWidget()`, которые выполняют аналогичные действия, что и одноименные методы в классе `QStackedLayout`. Кроме того, класс `QStackedWidget` наследует все методы из базовых классов и содержит два дополнительных метода:

- `indexOf(<Компонент>)` — возвращает индекс компонента, ссылка на который указана в параметре;
- `__len__()` — возвращает количество компонентов. Метод вызывается при использовании функции `len()`, а также для проверки объекта на логическое значение.

Чтобы отследить изменения внутри компонента следует назначить обработчики сигналов `currentChanged(int)` и `widgetRemoved(int)`.

4.6. Класс QSizePolicy

Если в вертикальный контейнер большой высоты добавить надпись и кнопку, то под надпись будет выделено максимальное пространство, а кнопка займет пространство, достаточное для рекомендуемых размеров, которые возвращает метод `sizeHint()`. Управление размерами компонентов внутри контейнера определяется правилами, установленными с помощью класса `QSizePolicy`. Установить правила для компонента можно с помощью метода `setSizePolicy(<QSizePolicy>)` из класса `QWidget`, а получить значение с помощью метода `sizePolicy()`.

Создать экземпляр класса `QSizePolicy` можно следующим способом:

```
<Объект> = QSizePolicy([<Правило для горизонтали>,
                        <Правило для вертикали>[, <Тип компонента>]])
```

Если параметры не заданы, то размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`. В первом и втором параметрах указываются следующие атрибуты из класса `QSizePolicy`:

- `Fixed` — размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`;
- `Minimum` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента. Размер может быть увеличен компоновщиком;
- `Maximum` — размер, возвращаемый методом `sizeHint()`, является максимальным для компонента. Размер может быть уменьшен компоновщиком;
- `Preferred` — размер, возвращаемый методом `sizeHint()`, является предпочтительным, но может быть как увеличен, так и уменьшен;
- `Expanding` — размер, возвращаемый методом `sizeHint()`, может быть как увеличен, так и уменьшен. Компоновщик должен предоставить компоненту столько пространства, сколько возможно;
- `MinimumExpanding` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента. Компоновщик должен предоставить компоненту столько пространства, сколько возможно;
- `Ignored` — размер, возвращаемый методом `sizeHint()`, игнорируется. Компонент получит столько пространства, сколько возможно.

Изменить значения уже после создания экземпляра класса `QSizePolicy` позволяют методы `setHorizontalPolicy(<Правило для горизонтали>)` и `setVerticalPolicy(<Правило для вертикали>)`.

С помощью методов `setHorizontalStretch(<Фактор для горизонтали>)` и `setVerticalStretch(<Фактор для вертикали>)` можно указать фактор растяжения. Чем больше указанное значение относительно значения, заданного в других

компонентах, тем больше места будет выделяться под компонент. Этот параметр можно сравнить с жесткостью пружины.

Можно указать, что минимальная высота компонента зависит от его ширины. Для этого необходимо передать значение `True` в метод `setHeightForWidth(<Флаг>)`. Кроме того, следует переопределить метод `heightForWidth(<Ширина>)` в классе компонента. Метод должен возвращать размер высоты компонента в соответствии с указанной в параметре шириной.

4.7. Объединение компонентов в группу

Состояние некоторых компонентов может зависеть от состояния других компонентов, например, из нескольких переключателей можно выбрать только один. В этом случае компоненты объединяют в группу. Группа компонентов отображается внутри рамки, на границе которой выводится текст подсказки. Реализовать группу в PyQt позволяет класс `QGroupBox`. Иерархия наследования выглядит так:

`(QObject, QPaintDevice) — QWidget — QGroupBox`

Создать экземпляр класса `QGroupBox` можно следующим образом:

```
<Объект> = QGroupBox([<Родитель>])
<Объект> = QGroupBox(<Текст>[, <Родитель>])
```

В необязательном параметре `<Родитель>` можно указать ссылку на родительский компонент. Параметр `<Текст>` задает текст подсказки, которая отобразится на верхней границе рамки. Внутри текста подсказки символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы первый компонент внутри группы окажется в фокусе ввода.

После создания экземпляра класса `QGroupBox` следует добавить компоненты в какой-либо контейнер, а затем передать ссылку на контейнер в метод `setLayout()`. Типичный пример использования класса `QGroupBox` выглядит так:

```
window = QtGui.QWidget()
mainbox = QtGui.QVBoxLayout()
radio1 = QtGui.QRadioButton("&Да")
radio2 = QtGui.QRadioButton("&Нет")
box = QtGui.QGroupBox("&Вы знаете язык Python?") # Объект группы
hbox = QtGui.QHBoxLayout() # Контейнер для группы
hbox.addWidget(radio1)     # Добавляем компоненты
hbox.addWidget(radio2)
```

```
box.setLayout(hbox)          # Передаем ссылку на контейнер
mainbox.addWidget(box)       # Добавляем группу в главный контейнер
window.setLayout(mainbox)    # Передаем ссылку на главный контейнер в окно
radio1.setChecked(True)      # Выбираем первый переключатель
```

Класс `QGroupBox` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setTitle(<Текст>)` — задает текст подсказки;
- `setAlignment(<Выравнивание>)` — задает горизонтальное местоположение текста подсказки. В параметре указываются следующие атрибуты из класса `QtCore.Qt`: `AlignLeft`, `AlignHCenter` или `AlignRight`. Пример:

  ```
  box.setAlignment(QtCore.Qt.AlignRight)
  ```

- `setCheckable(<Флаг>)` — если в параметре указать значение `True`, то перед текстом подсказки будет отображен флажок. Если флажок установлен, то группа будет активной, а если флажок снят, то все компоненты внутри группы станут неактивными. По умолчанию флажок не отображается;
- `isCheckable()` — возвращает значение `True`, если флажок выводится перед надписью, и `False` — в противном случае;
- `setChecked(<Флаг>)` — если в параметре указать значение `True`, то флажок, отображаемый перед текстом подсказки, будет установлен. Значение `False` сбрасывает флажок;
- `isChecked()` — возвращает значение `True`, если флажок, отображаемый перед текстом подсказки, установлен, и `False` — в противном случае;
- `setFlat(<Флаг>)` — если в параметре указано значение `True`, то отображается только верхняя граница рамки, а если `False` — то все границы рамки;
- `isFlat()` — возвращает значение `True`, если отображается только верхняя граница рамки, и `False` — если все границы рамки.

Класс `QGroupBox` содержит следующие сигналы:

- `clicked(bool=0)` — генерируется при щелчке мышью на флажке, выводимом перед текстом подсказки. Если состояние флажка изменяется с помощью метода `setChecked()`, то сигнал не генерируется. Через параметр внутри обработчика доступно значение `True`, если флажок установлен, и `False` — если сброшен;
- `toggled(bool)` — генерируется при изменении статуса флажка, выводимого перед текстом подсказки. Через параметр внутри обработчика доступно значение `True`, если флажок установлен, и `False` — если сброшен.

4.8. Панель с рамкой

Класс `QFrame` расширяет возможности класса `QWidget` за счет добавления рамки различного стиля вокруг компонента. Этот класс наследуют в свою очередь некоторые компоненты, например, надписи, многострочные текстовые поля и др. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame
```

Конструктор класса `QFrame` имеет следующий формат:

```
<Объект> = QFrame([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Если в параметре `flags` указан тип окна, то компонент, имея родителя, будет обладать своим собственным окном, но будет привязан к родителю. Это позволяет, например, создать модальное окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `flags` мы уже рассматривали в *разд. 2.2*.

Класс `QFrame` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

➔ `setFrameShape(<Форма>)` — задает форму рамки. Могут быть указаны следующие атрибуты из класса `QFrame`:

- `NoFrame` — `0` — нет рамки;
- `Box` — `1` — прямоугольная рамка;
- `Panel` — `2` — панель, которая может быть выпуклой или вогнутой;
- `WinPanel` — `3` — панель со стилем, принятым в Windows. Ширина границы 2 пиксела. Панель может быть выпуклой или вогнутой;
- `HLine` — `4` — горизонтальная линия. Используется как разделитель;
- `VLine` — `5` — вертикальная линия без содержимого;
- `StyledPanel` — `6` — панель, внешний вид которой зависит от текущего стиля. Панель может быть выпуклой или вогнутой;

➔ `setFrameShadow(<Тень>)` — задает стиль тени. Могут быть указаны следующие атрибуты из класса `QFrame`:

- `Plain` — `16` — нет эффектов;
- `Raised` — `32` — панель отображается выпуклой;
- `Sunken` — `48` — панель отображается вогнутой;

➔ `setFrameStyle(<Стиль>)` — задает форму рамки и стиль тени одновременно. В качестве значения указывается комбинация атрибутов из класса `QFrame` через оператор `|`. Пример:

```
frame.setFrameStyle(QtGui.QFrame.Panel | QtGui.QFrame.Raised)
```

➔ `setLineWidth(<Ширина>)` — задает ширину линии рамки;

➔ `setMidLineWidth(<Ширина>)` — задает ширину средней линии рамки. Средняя линия используется для создания эффекта выпуклости и вогнутости и доступна только для форм рамки `Box`, `HLine` и `VLine`.

4.9. Панель с вкладками

Для создания панели с вкладками предназначен класс `QTabWidget`. Панель состоит из области заголовка с ярлыками и набора вкладок с различными компонентами. В один момент времени показывается содержимое только одной вкладки. Щелчок мышью на ярлыке в области заголовка приводит к отображению содержимого соответствующий вкладки. Иерархия наследования для класса `QTabWidget` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QTabWidget
```

Конструктор класса `QTabWidget` имеет следующий формат:

```
<Объект> = QTabWidget([<Родитель>])
```

В параметре `<Родитель>` указывается ссылка на родительский компонент. Если параметр не указан, то компонент будет обладать своим собственным окном. Типичный пример использования класса `QTabWidget` выглядит так:

```
window = QtGui.QWidget()
tab = QtGui.QTabWidget()
tab.addTab(QtGui.QLabel("Содержимое вкладки 1"), "Вкладка &1")
tab.addTab(QtGui.QLabel("Содержимое вкладки 2"), "Вкладка &2")
tab.addTab(QtGui.QLabel("Содержимое вкладки 3"), "Вкладка &3")
tab.setCurrentIndex(0)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(tab)
window.setLayout(vbox)
window.show()
```

Класс `QTabWidget` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

➔ `addTab()` — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Форматы метода:

```
addTab(<Компонент>, <Текст заголовка>)
addTab(<Компонент>, <QIcon>, <Текст заголовка>)
```

В параметре `<Компонент>` указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр `<Текст заголовка>` задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы соответствующая вкладка будет отображена. Параметр `<QIcon>` позволяет указать иконку (экземпляр класса `QIcon`), которая отобразится перед текстом в области заголовка. Пример указания стандартной иконки:

```
style = window.style()
icon = style.standardIcon(QtGui.QStyle.SP_DriveNetIcon)
tab.addTab(QtGui.QLabel("Содержимое вкладки 1"), icon,
           "Вкладка &1")
```

Пример загрузки иконки из файла:

```
icon = QtGui.QIcon("icon.png")
tab.addTab(QtGui.QLabel("Содержимое вкладки 1"), icon,
           "Вкладка &1")
```

➔ `insertTab()` — добавляет вкладку в указанную позицию. Метод возвращает индекс добавленной вкладки. Форматы метода:

```
insertTab(<Индекс>, <Компонент>, <Текст заголовка>)
insertTab(<Индекс>, <Компонент>, <QIcon>, <Текст заголовка>)
```

➔ `removeTab(<Индекс>)` — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;

➔ `clear()` — удаляет все вкладки, при этом компоненты, которые отображались на вкладках, не удаляются;

➔ `setTabText(<Индекс>, <Текст заголовка>)` — задает текст заголовка для вкладки с указанным индексом;

➔ `setElideMode(<Режим>)` — задает режим обрезки текста в названии вкладки, если он не помещается в отведенную область. В месте пропуска выводится троеточие. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `ElideLeft` — `0` — текст обрезается слева;
- `ElideRight` — `1` — текст обрезается справа;
- `ElideMiddle` — `2` — текст обрезается по середине;

- `ElideNone` — `3` — текст не обрезается;

➔ `tabText(<Индекс>)` — возвращает текст заголовка вкладки с указанным индексом;

➔ `setTabIcon(<Индекс>, <QIcon>)` — устанавливает иконку перед текстом в заголовке вкладки с указанным индексом. Во втором параметре указывается экземпляр класса `QIcon`;

➔ `setTabPosition(<Позиция>)` — задает позицию области заголовка. Могут быть указаны следующие атрибуты из класса `QTabWidget`:

- `North` — `0` — сверху;
- `South` — `1` — снизу;
- `West` — `2` — слева;
- `East` — `3` — справа.

Пример указания значения:

```
tab.setTabPosition(QtGui.QTabWidget.South)
```

➔ `setTabShape(<Форма>)` — задает форму углов ярлыка вкладки в области заголовка. Могут быть указаны следующие атрибуты из класса `QTabWidget`:

- `Rounded` — `0` — скругленные углы (значение по умолчанию);
- `Triangular` — `1` — треугольная форма;

➔ `setTabsClosable(<Флаг>)` — если в качестве параметра указано значение `True`, то после текста заголовка будет отображена кнопка закрытия вкладки. При нажатии этой кнопки генерируется сигнал `tabCloseRequested(int)`;

➔ `setMovable(<Флаг>)` — если в качестве параметра указано значение `True`, то ярлыки вкладок можно перемещать с помощью мыши;

➔ `setDocumentMode(<Флаг>)` — если в качестве параметра указано значение `True`, то область компонента не будет отображаться как панель;

➔ `setUsesScrollButtons(<Флаг>)` — если в качестве параметра указано значение `True`, то когда все ярлыки вкладок не помещаются в область заголовка появляются две кнопки, с помощью которых можно прокручивать область заголовка, тем самым отображая только часть ярлыков. Значение `False` запрещает сокрытие ярлыков;

➔ `setTabToolTip(<Индекс>, <Текст>)` — задает текст вплывающей подсказки для ярлыка вкладки с указанным индексом;

➔ `setTabWhatsThis(<Индекс>, <Текст>)` — задает текст справки для ярлыка вкладки с указанным индексом;

- `setTabEnabled(<Индекс>, <Флаг>)` — если во втором параметре указано значение `False`, то вкладка с указанным в первом параметре индексом станет недоступной. Значение `True` делает вкладку доступной;
- `isTabEnabled(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом доступна, и `False` — в противном случае;
- `count()` — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции `len()`:

```
print(tab.count(), len(tab))
```

- `currentIndex()` — возвращает индекс видимой вкладки;
- `currentWidget()` — возвращает ссылку на компонент, расположенный на видимой вкладке;
- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;
- `indexOf(<Компонент>)` — возвращает индекс вкладки, на которой расположен компонент `<Компонент>`. Если компонент не найден возвращается значение `-1`;
- `setCurrentIndex(int)` — делает видимой вкладку с указанным в параметре индексом. Метод является слотом;
- `setCurrentWidget(QWidget *)` — делает видимым компонент, ссылка на который указана в параметре. Метод является слотом.

Класс `QTabWidget` содержит следующие сигналы:

- `currentChanged(int)` — генерируется при изменении вкладки. Через параметр внутри обработчика доступен индекс новой вкладки;
- `tabCloseRequested(int)` — генерируется при нажатии кнопки закрытия вкладки. Через параметр внутри обработчика доступен индекс вкладки.

4.10. Компонент "аккордеон"

Класс `QToolBox` позволяет создать компонент с несколькими вкладками. Изначально отображается содержимое только одной вкладки, а у остальных доступны только заголовки. После щелчка мышью на заголовке вкладки она открывается, а остальные сворачиваются. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QToolBox
```

Конструктор класса `QToolBox` имеет следующий формат:

```
<Объект> = QToolBox([parent=<Родитель>][, flags=<Тип окна>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, то компонент будет обладать своим собственным окном. В параметре flags может быть указан тип окна. Пример использования класса QToolBox:

```
window = QtGui.QWidget()
toolBox = QtGui.QToolBox()
toolBox.addItem(QtGui.QLabel("Содержимое вкладки 1"), "Вкладка &1")
toolBox.addItem(QtGui.QLabel("Содержимое вкладки 2"), "Вкладка &2")
toolBox.addItem(QtGui.QLabel("Содержимое вкладки 3"), "Вкладка &3")
toolBox.setCurrentIndex(0)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(toolBox)
window.setLayout(vbox)
window.show()
```

Класс QToolBox содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- addItem() — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Форматы метода:

  addItem(<Компонент>, <Текст заголовка>)

  addItem(<Компонент>, <QIcon>, <Текст заголовка>)

  В параметре <Компонент> указывается ссылка на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр <Текст заголовка> задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ &, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы соответствующая вкладка будет отображена. Параметр <QIcon> позволяет указать иконку (экземпляр класса QIcon), которая отобразится перед текстом в области заголовка;

- insertItem() — добавляет вкладку в указанную позицию. Метод возвращает индекс добавленной вкладки. Форматы метода:

  insertItem(<Индекс>, <Компонент>, <Текст заголовка>)

  insertItem(<Индекс>, <Компонент>, <QIcon>, <Текст заголовка>)

- removeItem(<Индекс>) — удаляет вкладку с указанным индексом, при этом компонент, который отображался на вкладке, не удаляется;

- `setItemText(<Индекс>, <Текст заголовка>)` — задает текст заголовка для вкладки с указанным индексом;
- `itemText(<Индекс>)` — возвращает текст заголовка вкладки с указанным индексом;
- `setItemIcon(<Индекс>, <QIcon>)` — устанавливает иконку перед текстом в заголовке вкладки с указанным индексом. Во втором параметре указывается экземпляр класса `QIcon`;
- `setItemToolTip(<Индекс>, <Текст>)` — задает текст вплывающей подсказки для ярлыка вкладки с указанным индексом;
- `setItemEnabled(<Индекс>, <Флаг>)` — если во втором параметре указано значение `False`, то вкладка с указанным в первом параметре индексом станет недоступной. Значение `True` делает вкладку доступной;
- `isItemEnabled(<Индекс>)` — возвращает значение `True`, если вкладка с указанным индексом доступна, и `False` — в противном случае;
- `count()` — возвращает количество вкладок. Получить количество вкладок можно также с помощью функции `len()`:

  ```
  print(toolBox.count(), len(toolBox))
  ```

- `currentIndex()` — возвращает индекс видимой вкладки;
- `currentWidget()` — возвращает ссылку на компонент, расположенный на видимой вкладке;
- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;
- `indexOf(<Компонент>)` — возвращает индекс вкладки, на которой расположен компонент `<Компонент>`. Если компонент не найден возвращается значение `-1`;
- `setCurrentIndex(int)` — делает видимой вкладку с указанным в параметре индексом. Метод является слотом;
- `setCurrentWidget(QWidget *)` — делает видимым компонент, ссылка на который указана в параметре. Метод является слотом.

При изменении вкладки генерируется сигнал `currentChanged(int)`. Через параметр внутри обработчика доступен индекс новой вкладки.

4.11. Панели с изменяемым размером

Класс `QSplitter` позволяет изменять размеры добавленных компонентов с помощью мыши, взявшись за границу между компонентами. Иерархия наследования выглядит так:

`(QObject, QPaintDevice) — QWidget — QFrame — QSplitter`

Конструктор класса `QSplitter` имеет два формата:

```
<Объект> = QSplitter([parent=<Родитель>])
<Объект> = QSplitter(<Ориентация>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Параметр `<Ориентация>` задает ориентацию размещения компонентов. Могут быть заданы атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) из класса `QtCore.Qt`. Если параметр не указан, то компоненты размещаются по горизонтали. Пример использования класса `QSplitter`:

```
window = QtGui.QWidget()
splitter = QtGui.QSplitter(QtCore.Qt.Vertical)
label1 = QtGui.QLabel("Содержимое компонента 1")
label2 = QtGui.QLabel("Содержимое компонента 2")
label1.setFrameStyle(QtGui.QFrame.Box | QtGui.QFrame.Plain)
label2.setFrameStyle(QtGui.QFrame.Box | QtGui.QFrame.Plain)
splitter.addWidget(label1)
splitter.addWidget(label2)
vbox = QtGui.QVBoxLayout()
vbox.addWidget(splitter)
window.setLayout(vbox)
window.show()
```

Класс `QSplitter` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addWidget(<Компонент>)` — добавляет компонент в конец контейнера;
- `insertWidget(<Индекс>, <Компонент>)` — добавляет компонент в указанную позицию. Если компонент был добавлен ранее, то он будет перемещен в новую позицию;
- `setOrientation(<Ориентация>)` — задает ориентацию размещения компонентов. Могут быть заданы атрибуты `Horizontal` (по горизонтали) или `Vertical` (по вертикали) из класса `QtCore.Qt`;

- `setHandleWidth(<Ширина>)` — задает ширину компонента-разделителя, взявшись за который мышью можно изменить размер области;
- `saveState()` — возвращает экземпляр класса `QByteArray` с размерами всех областей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода `restoreState(<QByteArray>)`;
- `setChildrenCollapsible(<Флаг>)` — если в параметре указано значение `False`, то пользователь не сможет уменьшить размеры всех компонентов до нуля. По умолчанию размер может быть нулевым, даже если установлены минимальные размеры компонента;
- `setCollapsible(<Индекс>, <Флаг>)` — значение `False` в параметре `<Флаг>` запрещает уменьшение размеров до нуля для компонента с указанным индексом;
- `setOpaqueResize (<Флаг>)` — если в качестве параметра указано значение `False`, то размеры компонентов изменятся только после окончания перемещения границы и отпускания кнопки мыши. В процессе перемещения мыши вместе с ней будет перемещаться специальный компонент в виде линии;
- `setStretchFactor(<Индекс>, <Фактор>)` — задает фактор растяжения для компонента с указанным индексом;
- `setSizes(<Список>)` — задает размеры всех компонентов. Для горизонтального контейнера указывается список со значениями ширины каждого компонента, а для вертикального контейнера — список со значениями высоты каждого компонента;
- `sizes()` — возвращает список с размерами (шириной или высотой). Пример:

```
print(splitter.sizes()) # Результат: [308, 15]
```

- `count()` — возвращает количество компонентов. Получить количество компонентов можно также с помощью функции `len()`:

```
print(splitter.count(), len(splitter))
```

- `widget(<Индекс>)` — возвращает ссылку на компонент, который расположен по указанному индексу, или значение `None`;
- `indexOf(<Компонент>)` — возвращает индекс области, в которой расположен компонент `<Компонент>`. Если компонент не найден возвращается значение `-1`.

При изменении размеров генерируется сигнал `splitterMoved(int,int)`. Через первый параметр внутри обработчика доступна новая позиция, а через второй параметр — индекс перемещаемого разделителя.

4.12. Область с полосами прокрутки

Класс `QScrollArea` реализует область с полосами прокрутки. Если компонент не помещается в размеры области, то автоматически отображаются полосы прокрутки. Изменение положения полос прокрутки с помощью мыши автоматически приводит к прокрутке содержимого области. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                                QAbstractScrollArea — QScrollArea
```

Конструктор класса `QScrollArea` имеет следующий формат:

```
<Объект> = QScrollArea([<Родитель>])
```

Класс `QScrollArea` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setWidget(<Компонент>)` — добавляет компонент в область прокрутки;
- `setWidgetResizable(<Флаг>)` — если в качестве параметра указано значение `True`, то при изменении размеров области будут изменяться и размеры компонента. Значение `False` запрещает изменение размеров компонента;
- `setAlignment(<Выравнивание>)` — задает местоположение компонента внутри области, когда размеры области больше размеров компонента. Пример:

  ```
  scrollArea.setAlignment(QtCore.Qt.AlignCenter)
  ```

- `ensureVisible(<X>, <Y>[, xMargin=50][, yMargin=50])` — прокручивает область к точке с координатами `(<X>, <Y>)` и полями `xMargin` и `yMargin`;
- `ensureWidgetVisible(<Компонент>[, xMargin=50][, yMargin=50])` — прокручивает область таким образом, чтобы `<Компонент>` был видим;
- `widget()` — возвращает ссылку на компонент, который расположен внутри области, или значение `None`;
- `takeWidget()` — удаляет компонент из области и возвращает ссылку на него. Сам компонент не удаляется.

Класс `QScrollArea` наследует следующие методы из класса `QAbstractScrollArea` (перечислены только основные методы; полный список смотрите в документации):

- `horizontalScrollBar()` — возвращает ссылку на горизонтальную полосу прокрутки (экземпляр класса `QScrollBar`);
- `verticalScrollBar()` — возвращает ссылку на вертикальную полосу прокрутки (экземпляр класса `QScrollBar`);
- `cornerWidget()` — возвращает ссылку на компонент, расположенный в правом нижнем углу между двумя полосами прокрутки, или значение `None`;

➔ `viewport()` — возвращает ссылку на окно области прокрутки;

➔ `setHorizontalScrollBarPolicy(<Режим>)` — устанавливает режим отображения горизонтальной полосы прокрутки;

➔ `setVerticalScrollBarPolicy(<Режим>)` — устанавливает режим отображения вертикальной полосы прокрутки. В параметре `<Режим>` могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `ScrollBarAsNeeded` — `0` — полоса прокрутки отображается только в том случае, если размеры компонента больше размеров области;
- `ScrollBarAlwaysOff` — `1` — полоса прокрутки никогда не отображается;
- `ScrollBarAlwaysOn` — `2` — полоса прокрутки всегда отображается;

➔ `setViewportMargins(<Слева>, <Сверху>, <Справа>, <Снизу>)` — задает отступ от границ области до компонента. По умолчанию отступы равны нулю.

Глава 5. Основные компоненты

Практически все компоненты пользовательского интерфейса наследуют классы `QObject` и `QWidget`. Следовательно, методы этих классов, которые мы рассматривали в предыдущих главах, доступны всем компонентам. Если компонент не имеет родителя, то он обладает собственным окном и, например, его положение отсчитывается относительно экрана. Если же компонент имеет родителя, то его положение отсчитывается относительно родительского компонента. Это обстоятельство важно учитывать при работе с компонентами. Обращайте внимание на иерархию наследования, которую мы будем показывать для каждого компонента.

5.1. Надпись

Надпись применяется для вывода подсказки пользователю, информирования пользователя о ходе выполнения операции, назначения клавиш быстрого доступа применительно к другому компоненту, а также для вывода изображений и анимации. Кроме того, надписи позволяют отображать текст в формате HTML, отформатированный с помощью CSS, что делает надпись самым настоящим браузером. В библиотеке PyQt надпись реализуется с помощью класса `QLabel`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLabel
```

Конструктор класса `QLabel` имеет два формата:

```
<Объект> = QLabel([parent=<Родитель>][, flags=<Тип окна>])
<Объект> = QLabel(<Текст>[, parent=<Родитель>][, flags=<Тип окна>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном, тип которого можно задать с помощью параметра `flags`. Параметр `<Текст>` позволяет задать текст, который будет отображен на надписи. Пример:

```
label = QtGui.QLabel("Текст надписи", flags=QtCore.Qt.Window)
label.resize(300, 50)
label.show()
```

Класс `QLabel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText(<Текст>)` — задает текст, который будет отображен на надписи. Можно указать как обычный текст, так и текст в формате HTML, который содержит форматирование с помощью CSS. Пример:

```
label.setText("Текст <b>полужирный</b>")
```

Перевод строки в простом тексте осуществляется с помощью символа `\n`, а в тексте в формате HTML с помощью тега `<br>`. Пример:

```
label.setText("Текст\nна двух строках")
```

Внутри текста символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент, ссылка на который передана в метод `setBuddy()`, окажется в фокусе ввода. Чтобы вывести символ `&` необходимо его удвоить. Если надпись не связана с другим компонентом, то символ `&` выводится в составе текста. Пример:

```
label = QtGui.QLabel("&Пароль")
lineEdit = QtGui.QLineEdit()
label.setBuddy(lineEdit)
```

Метод является слотом с сигнатурой `setText(const QString&);`

- `setNum(<Число>)` — преобразует целое или вещественное число в строку и отображает ее на надписи. Метод является слотом с сигнатурами `setNum(int)` и `setNum(double)`;
- `setWordWrap(<Флаг>)` — если в параметре указано значение `True`, то текст может переноситься на другую строку. По умолчанию перенос строк не осуществляется;
- `setTextFormat(<Режим>)` — задает режим отображения текста. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `PlainText` — `0` — простой текст;
  - `RichText` — `1` — форматированный текст;
  - `AutoText` — `2` — автоматическое определение (режим по умолчанию). Если текст содержит теги, то используется режим `RichText`, в противном случае — режим `PlainText`;
- `text()` — возвращает текст надписи;
- `setAlignment(<Режим>)` — задает режим выравнивания текста внутри надписи. Допустимые значения мы рассматривали в *разд. 4.2*. Пример:

  ```
  label.setAlignment(QtCore.Qt.AlignRight | QtCore.Qt.AlignBottom)
  ```

- `setOpenExternalLinks(<Флаг>)` — если в качестве параметра указано значение `True`, то щелчок на гиперссылке приведет к открытию браузера, используемого в системе по умолчанию, и загрузке указанной страницы. Пример:

  ```
  label.setText('<a href="http://google.ru/">Это гиперссылка</a>')
  ```

```
label.setOpenExternalLinks(True)
```

- `setBuddy(<Компонент>)` — позволяет связать надпись с другим компонентом. В этом случае в тексте надписи можно задавать клавиши быстрого доступа, указав символ `&` перед буквой или цифрой. После нажатия комбинации клавиш в фокусе ввода окажется компонент, ссылка на который передана в качестве параметра;
- `setPixmap(<QPixmap>)` — позволяет вывести изображение на надпись. В качестве параметра указывается экземпляр класса `QPixmap`. Метод является слотом с сигнатурой `setPixmap(const QPixmap&)`. Пример:

```
label.setPixmap(QtGui.QPixmap("foto.jpg"))
```

- `setPicture(<QPicture>)` — позволяет вывести рисунок. В качестве параметра указывается экземпляр класса `QPicture`. Метод является слотом с сигнатурой `setPicture(const QPicture&);`
- `setMovie(<QMovie>)` — позволяет вывести анимацию. В качестве параметра указывается экземпляр класса `QMovie`. Метод является слотом с сигнатурой `setMovie(QMovie *);`
- `setScaledContents(<Флаг>)` — если в параметре указано значение `True`, то при изменении размеров надписи размер содержимого также будет изменяться. По умолчанию изменение размеров содержимого не осуществляется;
- `setMargin(<Отступ>)` — задает отступ от рамки до содержимого надписи;
- `setIndent(<Отступ>)` — задает отступ от рамки до текста надписи в зависимости от значения выравнивания. Если выравнивание производится по левой стороне, то задает отступ слева, если по правой стороне, то справа и т. д.;
- `clear()` — удаляет содержимое надписи. Метод является слотом;
- `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом надписи. Можно указать следующие атрибуты (или их комбинацию через оператор `|`) из класса `QtCore.Qt`:
  - `NoTextInteraction` — `0` — пользователь не может взаимодействовать с текстом надписи;
  - `TextSelectableByMouse` — `1` — текст можно выделить и скопировать в буфер обмена;
  - `TextSelectableByKeyboard` — `2` — текст можно выделить с помощью клавиш на клавиатуре. Внутри надписи будет отображен текстовый курсор;
  - `LinksAccessibleByMouse` — `4` — на гиперссылке можно щелкнуть мышью и скопировать ее адрес;
  - `LinksAccessibleByKeyboard` — `8` — с гиперссылкой можно взаимодействовать с помощью клавиатуры. Перемещаться между

гиперссылками можно с помощью клавиши <Tab>, а переходить по гиперссылке при нажатии клавиши <Enter>;

- `TextEditable` — `16` — текст надписи можно редактировать;
- `TextEditorInteraction` — комбинация `TextSelectableByMouse | TextSelectableByKeyboard | TextEditable`;
- `TextBrowserInteraction` — комбинация `TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard`;

➔ `setSelection(<Индекс>, <Длина>)` — выделяет фрагмент длиной `<Длина>`, начиная с позиции `<Индекс>`;

➔ `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено;

➔ `selectedText()` — возвращает выделенный текст или пустую строку;

➔ `hasSelectedText()` — возвращает значение `True`, если существует выделенный фрагмент, и `False` — в противном случае.

Класс `QLabel` содержит следующие сигналы:

➔ `linkActivated(const QString&)` — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен URL-адрес;

➔ `linkHovered(const QString&)` — генерируется при наведении указателя мыши на гиперссылку и выведении указателя. Через параметр внутри обработчика доступен URL-адрес или пустая строка.

5.2. Командная кнопка

Командная кнопка является наиболее часто используемым компонентом. При нажатии кнопки внутри обработчика обычно выполняется какая-либо операция. Кнопка реализуется с помощью класса `QPushButton`. Иерархия наследования:

`(QObject, QPaintDevice) — QWidget — QAbstractButton — QPushButton`

Конструктор класса `QPushButton` имеет три формата:

```
<Объект> = QPushButton([parent=<Родитель>])
<Объект> = QPushButton(<Текст>[, parent=<Родитель>])
<Объект> = QPushButton(<QIcon>, <Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Параметр `<Текст>` позволяет задать текст, который будет отображен на кнопке, а параметр `<QIcon>` позволяет добавить перед текстом иконку.

Класс `QPushButton` наследует следующие методы из класса `QAbstractButton` (перечислены только основные методы; полный список смотрите в документации):

- `setText(<Текст>)` — задает текст, который будет отображен на кнопке. Внутри текста символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы кнопка будет нажата. Чтобы вывести символ `&` необходимо его удвоить;
- `text()` — возвращает текст, отображаемый на кнопке;
- `setShortcut(<QKeySequence>)` — задает комбинацию клавиш быстрого доступа. Примеры указания значения:

```
button.setShortcut("Alt+B")
button.setShortcut(QtGui.QKeySequence.mnemonic("&B"))
button.setShortcut(QtGui.QKeySequence("Alt+B"))
button.setShortcut(
       QtGui.QKeySequence(QtCore.Qt.ALT + QtCore.Qt.Key_E))
```

- `setIcon(<QIcon>)` — позволяет вставить иконку перед текстом;
- `setIconSize(<QSize>)` — задает размеры иконки. В качестве параметра указывается экземпляр класса `QSize`. Метод является слотом с сигнатурой `setIconSize(const QSize&)`;
- `setAutoRepeat(<Флаг>)` — если в качестве параметра указано значение `True`, то сигнал `clicked()` будет периодически генерироваться пока кнопка находится в нажатом состоянии. Примером являются кнопки, изменяющие значение полосы прокрутки;
- `animateClick([<Интервал>])` — имитирует нажатие кнопки пользователем. После нажатия кнопка находится в этом состоянии указанный промежуток времени, по истечении которого кнопка отпускается. Значение указывается в миллисекундах. Если параметр не указан, то интервал равен 100 миллисекундам. Метод является слотом;
- `click()` — имитирует нажатие кнопки без анимации. Метод является слотом;
- `toggle()` — переключает кнопку. Метод является слотом;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка является переключателем, который может находиться в двух состояниях — установленном и не установленном;

- `setChecked(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка-переключатель будет находиться в установленном состоянии. Метод является слотом с сигнатурой `setChecked(bool)`;
- `isChecked()` — возвращает значение `True`, если кнопка находится в установленном состоянии, и `False` — в противном случае;
- `setAutoExclusive(<Флаг>)` — если в качестве параметра указано значение `True`, то внутри группы только одна кнопка-переключатель может быть установлена;
- `setDown(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка будет находиться в нажатом состоянии;
- `isDown()` — возвращает значение `True`, если кнопка находится в нажатом состоянии, и `False` — в противном случае.

Кроме перечисленных состояний кнопка может находиться в неактивном состоянии. Для этого необходимо передать значение `False` в метод `setEnabled()` из класса `QWidget`. Проверить активна кнопка или нет позволяет метод `isEnabled()`. Метод возвращает значение `True`, если кнопка находится в активном состоянии, и `False` — в противном случае.

Класс `QAbstractButton` содержит следующие сигналы:

- `pressed()` — генерируется при нажатии кнопки;
- `released()` — генерируется при отпускании ранее нажатой кнопки;
- `clicked(bool=0)` — генерируется при нажатии, а затем отпускании кнопки мыши над кнопкой. Именно для этого сигнала обычно назначают обработчики;
- `toggled(bool)` — генерируется при переключении кнопки. Через параметр внутри обработчика доступно текущее состояние кнопки.

Класс `QPushButton` содержит дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setFlat(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка будет отображаться без границ;
- `setAutoDefault(<Флаг>)` — если в качестве параметра указано значение `True`, то кнопка может быть нажата с помощью клавиши <Enter>, при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>. В диалоговых окнах для всех кнопок по умолчанию указано значение `True`, а для остальных окон — значение `False`;
- `setDefault(<Флаг>)` — задает кнопку по умолчанию. Метод работает только в диалоговых окнах. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент, например, на текстовое поле;

- `setMenu(<QMenu>)` — устанавливает всплывающее меню, которое будет отображаться при нажатии кнопки. В качестве параметра указывается экземпляр класса `QMenu`;
- `menu()` — возвращает ссылку на всплывающее меню или значение `None`;
- `showMenu()` — отображает всплывающее меню. Метод является слотом.

5.3. Переключатель

Переключатели (иногда их называют радио-кнопками) обычно используются в группе. Внутри группы может быть включен только один переключатель. При попытке включить другой переключатель, ранее включенный переключатель автоматически отключается. Для объединения переключателей в группу можно воспользоваться классом `QGroupBox`, который мы уже рассматривали в *разд. 4.7*, а также классом `QButtonGroup`, который чаще используется для объединения кнопок-переключателей на панели инструментов. Переключатель реализуется с помощью класса `QRadioButton`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QRadioButton
```

Конструктор класса `QRadioButton` имеет два формата:

```
<Объект> = QRadioButton([parent=<Родитель>])
<Объект> = QRadioButton(<Текст>[, parent=<Родитель>])
```

Класс `QRadioButton` наследует все методы из класса `QAbstractButton` (см. *разд. 5.2*). Включить или отключить переключатель позволяет метод `setChecked()`, проверить текущий статус можно с помощью метода `isChecked()`, а чтобы перехватить переключение следует назначить обработчик сигнала `toggled(bool)`. Через параметр внутри обработчика доступно текущее состояние переключателя.

5.4. Флажок

Флажок предназначен для включения или выключения каких-либо опций настроек программы пользователем и может иметь несколько состояний: установлен, сброшен и частично установлен. Флажок реализуется с помощью класса `QCheckBox`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QCheckBox
```

Конструктор класса `QCheckBox` имеет два формата:

```
<Объект> = QCheckBox([parent=<Родитель>])
<Объект> = QCheckBox(<Текст>[, parent=<Родитель>])
```

Класс `QCheckBox` наследует все методы из класса `QAbstractButton` (см. *разд. 5.2*), а также добавляет несколько новых:

- `setCheckState(<Статус>)` — задает статус флажка. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок частично установлен;
  - `Checked` — `2` — флажок установлен;
- `checkState()` — возвращает текущий статус флажка;
- `setTristate([<Флаг>=True])` — если в качестве параметра указано значение `True` (значение по умолчанию), то флажок может поддерживать все три статуса. По умолчанию поддерживаются только статусы установлен и сброшен;
- `isTristate()` — возвращает значение `True`, если флажок поддерживает три статуса, и `False` — в противном случае.

Чтобы перехватить смену статуса флажка следует назначить обработчик сигнала `stateChanged(int)`. Через параметр внутри обработчика доступен текущий статус флажка.

Если используется флажок, поддерживающий только два состояния, то установить или сбросить флажок позволяет метод `setChecked()`, проверить текущий статус можно с помощью метода `isChecked()`, а чтобы перехватить изменение статуса следует назначить обработчик сигнала `toggled(bool)`. Через параметр внутри обработчика доступен текущий статус флажка.

5.5. Однострочное текстовое поле

Однострочное текстовое поле предназначено для ввода и редактирования текста небольшого объема. С его помощью можно также отобразить вводимые символы в виде звездочек (например, чтобы скрыть пароль) или вообще не отображать их (например, чтобы скрыть длину пароля). Поле по умолчанию поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Однострочное текстовое поле реализуется с помощью класса `QLineEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QLineEdit
```

Конструктор класса `QLineEdit` имеет два формата:

```
<Объект> = QLineEdit([parent=<Родитель>])
<Объект> = QLineEdit(<Текст>[, parent=<Родитель>])
```

В параметре parent указывается ссылка на родительский компонент. Если параметр не указан или имеет значение None, то компонент будет обладать своим собственным окном. Параметр <Текст> позволяет задать текст, который будет отображен в однострочном текстовом поле.

5.5.1. Основные методы и сигналы

Класс QLineEdit содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setEchoMode(<Режим>) — задает режим отображения текста. Могут быть указаны следующие атрибуты из класса QLineEdit:
  - Normal — 0 — показывать символы как они были введены;
  - NoEcho — 1 — не показывать вводимые символы;
  - Password — 2 — вместо символов указывать символ *;
  - PasswordEchoOnEdit — 3 — показывать символы при вводе, а при потере фокуса отображать символ *;
- setCompleter(<QCompleter>) — позволяет предлагать возможные варианты значений, начинающихся с введенных пользователем символов. В качестве параметра указывается экземпляр класса QCompleter. Пример:

```
lineEdit = QtGui.QLineEdit()
arr = ["кадр", "каменный", "камень", "камера"]
completer = QtGui.QCompleter(arr, window)
lineEdit.setCompleter(completer)
```

- setReadOnly(<Флаг>) — если в качестве параметра указано значение True, то поле будет доступно только для чтения;
- isReadOnly() — возвращает значение True, если поле доступно только для чтения, и False — в противном случае;
- setAlignment(<Выравнивание>) — задает выравнивание текста внутри поля;
- setMaxLength(<Количество>) — задает максимальное количество символов;
- setFrame(<Флаг>) — если в качестве параметра указано значение False, то поле будет отображаться без рамки;
- setDragEnabled(<Флаг>) — если в качестве параметра указано значение True, то режим перетаскивания текста из текстового поля с помощью мыши будет включен. По умолчанию однострочное текстовое поле только принимает перетаскиваемый текст;

- `setPlaceholderText(<Текст>)` — задает текст подсказки пользователю, который будет выводиться в поле, когда оно не содержит значения и находится вне фокуса ввода;
- `setText(<Текст>)` — вставляет указанный текст в поле. Метод является слотом с сигнатурой `setText(const QString&)`;
- `insert(<Текст>)` — вставляет текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален;
- `text()` — возвращает текст, содержащийся в текстовом поле;
- `displayText()` — возвращает текст, который видит пользователь. Результат зависит от режима отображения, заданного с помощью метода `setEchoMode()`, например, в режиме `Password` строка будет состоять из символов `*`;
- `selectedText()` — возвращает выделенный фрагмент или пустую строку;
- `clear()` — удаляет весь текст из поля. Метод является слотом;
- `backspace()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, то удаляет символ, стоящий слева от текстового курсора;
- `del_()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, то удаляет символ, стоящий справа от текстового курсора;
- `setSelection(<Индекс>, <Длина>)` — выделяет фрагмент длиной `<Длина>`, начиная с позиции `<Индекс>`. Во втором параметре можно указать отрицательное значение;
- `selectAll()` — выделяет весь текст в поле. Метод является слотом;
- `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено;
- `hasSelectedText()` — возвращает значение `True`, если поле содержит выделенный фрагмент, и `False` — в противном случае;
- `deselect()` — снимает выделение;
- `setCursorPosition(<Индекс>)` — задает положение текстового курсора;
- `cursorPosition()` — возвращает текущее положение текстового курсора;
- `cursorForward(<Флаг>, steps=1)` — перемещает текстовый курсор вперед на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, то фрагмент выделяется;
- `cursorBackward(<Флаг>, steps=1)` — перемещает текстовый курсор назад на указанное во втором параметре количество символов. Если в первом параметре указано значение `True`, то фрагмент выделяется;

- `cursorWordForward(<Флаг>)` — перемещает текстовый курсор вперед на одно слово. Если в параметре указано значение `True`, то фрагмент выделяется;
- `cursorWordBackward(<Флаг>)` — перемещает текстовый курсор назад на одно слово. Если в параметре указано значение `True`, то фрагмент выделяется;
- `home(<Флаг>)` — перемещает текстовый курсор в начало поля. Если в параметре указано значение `True`, то фрагмент выделяется;
- `end(<Флаг>)` — перемещает текстовый курсор в конец поля. Если в параметре указано значение `True`, то фрагмент выделяется;
- `cut()` — копирует выделенный текст в буфер обмена, а затем удаляет его, при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена, при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора, при условии, что поле доступно для редактирования. Метод является слотом;
- `undo()` — отменяет последнюю операцию ввода пользователем, при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;
- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `createStandardContextMenu()` — создает стандартное меню, которые отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню следует создать класс, наследующий класс `QLineEdit`, и переопределить метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню.

Класс `QLineEdit` содержит следующие сигналы:

- `cursorPositionChanged(int,int)` — генерируется при перемещении текстового курсора. Внутри обработчика через первый параметр доступна старая позиция курсора, а через второй параметр — новая позиция;
- `editingFinished()` — генерируется при нажатии клавиши <Enter> или потере полем фокуса ввода;

- `returnPressed()` — генерируется при нажатии клавиши <Enter>;
- `selectionChanged()` — генерируется при изменении выделения;
- `textChanged(const QString&)` — генерируется при изменении текста внутри поля пользователем или программно. Внутри обработчика через параметр доступно новое значение;
- `textEdited(const QString&)` — генерируется при изменении текста внутри поля пользователем. Сигнал не генерируется при изменении текста с помощью метода `setText()`. Внутри обработчика через параметр доступно новое значение.

5.5.2. Ввод данных по маске

С помощью метода `setInputMask(<Маска>)` можно ограничить ввод символов, допустимым диапазоном значений. В качестве значения указывается строка, имеющая следующий формат:

`"<Последовательность символов>[;<Символ-заполнитель>]"`

В первом параметре указывается комбинация из следующих специальных символов:

- `9` — обязательна цифра от `0` до `9`;
- `0` — разрешена, но не обязательна цифра от `0` до `9`;
- `D` — обязательна цифра от `1` до `9`;
- `d` — разрешена, но не обязательна цифра от `1` до `9`;
- `B` — обязательна цифра `0` или `1`;
- `b` — разрешена, но не обязательна цифра `0` или `1`;
- `H` — обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `h` — разрешен, но не обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `#` — разрешена, но не обязательна цифра или знак плюс или минус;
- `A` — обязательна буква в любом регистре;
- `a` — разрешена, но не обязательна буква;
- `N` — обязательна буква в любом регистре или цифра от `0` до `9`;
- `n` — разрешена, но не обязательна буква или цифра от `0` до `9`;
- `X` — обязателен любой символ;
- `x` — разрешен, но не обязателен любой символ;
- `>` — все последующие буквы переводятся в верхний регистр;

- `<` — все последующие буквы переводятся в нижний регистр;
- `!` — отключает изменение регистра;
- `\` — используется для отмены действия спецсимволов.

Все остальные символы трактуются как есть. В необязательном параметре `<Символ-заполнитель>` можно указать символ, который будет отображаться в поле, обозначая место ввода. Если параметр не указан, то символом является пробел. Пример:

```
lineEdit.setInputMask("Дата: 99.B9.9999;_")  # Дата: __.__.____
lineEdit.setInputMask("Дата: 99.B9.9999;#")  # Дата: ##.##.####
lineEdit.setInputMask("Дата: 99.B9.9999 г.") # Дата:   .  .     г.
```

Проверить соответствие введенных данных маске позволяет метод `hasAcceptableInput()`. Если данные соответствуют маске, то метод возвращает значение `True`, а в противном случае — `False`.

5.5.3. Контроль ввода

Контролировать ввод данных позволяет метод `setValidator(<QValidator>)`. В качестве значения указывается экземпляр класса, наследующего класс `QValidator`. Существуют следующие стандартные классы, позволяющие контролировать ввод данных:

- `QIntValidator` — допускает ввод только целых чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

  ```
  QIntValidator([parent=None])
  QIntValidator(<Начальное значение>, <Конечное значение>,
                <Родитель>)
  ```

  Пример ограничения ввода диапазоном целых чисел от `0` до `100`:

  ```
  lineEdit.setValidator(QtGui.QIntValidator(0, 100, window))
  ```

- `QDoubleValidator` — допускает ввод только вещественных чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

  ```
  QDoubleValidator([parent=None])
  QDoubleValidator(<Начальное значение>, <Конечное значение>,
                   <Количество цифр после точки>, <Родитель>)
  ```

  Пример ограничения ввода диапазоном вещественных чисел от `0.0` до `100.0`. и двумя цифрами после десятичной точки:

  ```
  lineEdit.setValidator(
           QtGui.QDoubleValidator(0.0, 100.0, 2, window))
  ```

Чтобы позволить вводить числа в экспоненциальной форме необходимо передать значение атрибута `ScientificNotation` в метод `setNotation()`. Если передать значение атрибута `StandardNotation`, то число должно быть только в десятичной форме. Пример:

```
validator = QtGui.QDoubleValidator(0.0, 100.0, 2, window)
validator.setNotation(QtGui.QDoubleValidator.StandardNotation)
lineEdit.setValidator(validator)
```

➔ `QRegExpValidator` — позволяет проверить данные на соответствие шаблону регулярного выражения. Форматы конструктора:

```
QRegExpValidator([parent=None])
QRegExpValidator(<Экземпляр класса QRegExp>, <Родитель>)
```

Пример ввода только цифр от `0` до `9`:

```
validator = QtGui.QRegExpValidator(
                    QtCore.QRegExp("[0-9]+"), window)
lineEdit.setValidator(validator)
```

Обратите внимание на то, что производится проверка полного соответствия шаблону, поэтому символы `^` и `$` явным образом указывать не нужно.

Проверить соответствие введенных данных условию позволяет метод `hasAcceptableInput()`. Если данные соответствуют условию, то метод возвращает значение `True`, а в противном случае — `False`.

5.6. Многострочное текстовое поле

Многострочное текстовое поле предназначено для ввода и редактирования, как простого текста, так и текста в формате HTML. Поле по умолчанию поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Многострочное текстовое поле реализуется с помощью класса `QTextEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                          QAbstractScrollArea — QTextEdit
```

Конструктор класса `QTextEdit` имеет два формата:

```
<Объект> = QTextEdit([parent=<Родитель>])
<Объект> = QTextEdit(<Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Если параметр не указан или имеет значение `None`, то компонент будет обладать своим собственным окном. Параметр `<Текст>` позволяет задать текст в формате HTML, который будет отображен в текстовом поле.

Примечание
Класс `QTextEdit` предназначен для отображения как простого текста, так и текста в формате HTML. Если поддержка HTML не нужна, то следует воспользоваться классом `QPlainTextEdit`, который оптимизирован для работы с простым текстом большого объема.

5.6.1. Основные методы и сигналы

Класс `QTextEdit` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText(<Текст>)` — вставляет указанный текст в поле. Текст может быть простым или в формате HTML. Метод является слотом с сигнатурой `setText(const QString&);`
- `setPlainText(<Текст>)` — вставляет простой текст. Метод является слотом с сигнатурой `setPlainText(const QString&);`
- `setHtml(<Текст>)` — вставляет текст в формате HTML. Метод является слотом с сигнатурой `setHtml(const QString&);`
- `insertPlainText(<Текст>)` — вставляет простой текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален. Метод является слотом с сигнатурой `insertPlainText(const QString&);`
- `insertHtml(<Текст>)` — вставляет текст в формате HTML в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален. Метод является слотом с сигнатурой `insertHtml(const QString&);`
- `append(<Текст>)` — добавляет новый абзац с указанным текстом в формате HTML в конец поля;
- `setDocumentTitle(<Текст>)` — задает текст заголовка документа (для тега `<title>`);
- `documentTitle()` — возвращает текст заголовка (из тега `<title>`);
- `toPlainText()` — возвращает простой текст, содержащийся в текстовом поле;
- `toHtml()` — возвращает текст в формате HTML;
- `clear()` — удаляет весь текст из поля. Метод является слотом;
- `selectAll()` — выделяет весь текст в поле. Метод является слотом;
- `zoomIn([range=1])` — увеличивает размер шрифта. Метод является слотом;
- `zoomOut([range=1])` — уменьшает размер шрифта. Метод является слотом;

- `cut()` — копирует выделенный текст в буфер обмена, а затем удаляет его из поля, при условии, что есть выделенный фрагмент. Метод является слотом;
- `copy()` — копирует выделенный текст в буфер обмена, при условии, что есть выделенный фрагмент. Метод является слотом;
- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора, при условии, что поле доступно для редактирования. Метод является слотом;
- `canPaste()` — возвращает `True`, если из буфера обмена можно вставить текст, и `False` — в противном случае;
- `setAcceptRichText(<Флаг>)` — если в качестве параметра указано значение `True`, то в поле можно будет вставить текст в формате HTML из буфера обмена или при помощи перетаскивания. Значение `False` отключает эту возможность;
- `acceptRichText()` — возвращает значение `True`, если в поле можно вставить текст в формате HTML, и `False` — в противном случае;
- `undo()` — отменяет последнюю операцию ввода пользователем, при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве значения указано значение `True`, то операции отмены и повтора действий разрешены, а если `False` — то запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `createStandardContextMenu([<QPoint>])` — создает стандартное меню, которые отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню следует создать класс, наследующий класс `QTextEdit`, и переопределить метод `contextMenuEvent(self, <event>)`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню;
- `ensureCursorVisible()` — прокручивает область таким образом, чтобы текстовый курсор оказался в зоне видимости;
- `find(<Текст>[, <Режим>])` — производит поиск фрагмента (по умолчанию в прямом направлении без учета регистра символов) в текстовом поле. Если фрагмент найден, то он выделяется и метод возвращает значение `True`, в противном случае — значение `False`. В необязательном параметре `<Режим>` можно указать комбинацию (через оператор `|`) следующих атрибутов из класса `QTextDocument`:

- `FindBackward` — `1` — поиск в обратном направлении, а не в прямом;
- `FindCaseSensitively` — `2` — поиск с учетом регистра символов;
- `FindWholeWords` — `4` — поиск слов целиком, а не фрагментов;

➔ `print_(<QPrinter>)` — отправляет содержимое текстового поля на печать. В качестве параметра указывается экземпляр класса `QPrinter`. Пример печати в файл в формате PDF:

```
printer = QtGui.QPrinter()
printer.setOutputFormat(QtGui.QPrinter.PdfFormat)
printer.setOutputFileName("mypdf.pdf")
textEdit.print_(printer)
```

Класс `QTextEdit` содержит следующие сигналы:

➔ `currentCharFormatChanged(const QTextCharFormat&)` — генерируется при изменении формата. Внутри обработчика через параметр доступен новый формат;

➔ `cursorPositionChanged()` — генерируется при изменении положения текстового курсора;

➔ `selectionChanged()` — генерируется при изменении выделения;

➔ `textChanged()` — генерируется при изменении текста внутри поля;

➔ `copyAvailable(bool)` — генерируется при изменении возможности скопировать фрагмент. Внутри обработчика через параметр доступно значение `True`, если фрагмент можно скопировать, и `False` — в противном случае;

➔ `undoAvailable(bool)` — генерируется при изменении возможности отменить операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно отменить операцию ввода, и `False` — в противном случае;

➔ `redoAvailable(bool)` — генерируется при изменении возможности повторить отмененную операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно повторить отмененную операцию ввода, и `False` — в противном случае.

5.6.2. Изменение настроек поля

Для изменения настроек предназначены следующие методы из класса `QTextEdit` (перечислены только основные методы; полный список смотрите в документации):

➔ `setTextInteractionFlags(<Режим>)` — задает режим взаимодействия пользователя с текстом. Можно указать следующие атрибуты (или их комбинацию через оператор `|`) из класса `QtCore.Qt`:

- `NoTextInteraction` — `0` — пользователь не может взаимодействовать с текстом;
- `TextSelectableByMouse` — `1` — текст можно выделить и скопировать в буфер обмена;
- `TextSelectableByKeyboard` — `2` — текст можно выделить с помощью клавиш на клавиатуре. Внутри поля будет отображен текстовый курсор;
- `LinksAccessibleByMouse` — `4` — на гиперссылке можно щелкнуть мышью и скопировать ее адрес;
- `LinksAccessibleByKeyboard` — `8` — с гиперссылкой можно взаимодействовать с помощью клавиатуры. Перемещаться между гиперссылками можно с помощью клавиши <Tab>, а переходить по гиперссылке при нажатии клавиши <Enter>;
- `TextEditable` — `16` — текст можно редактировать;
- `TextEditorInteraction` — комбинация `TextSelectableByMouse | TextSelectableByKeyboard | TextEditable`;
- `TextBrowserInteraction` — комбинация `TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard`;

➔ `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, то поле будет доступно только для чтения;

➔ `isReadOnly()` — возвращает значение `True`, если поле доступно только для чтения, и `False` — в противном случае;

➔ `setLineWrapMode(<Режим>)` — задает режим переноса строк. В качестве значения могут быть указаны следующие атрибуты из класса `QTextEdit`:

- `NoWrap` — `0` — перенос строк не производится;
- `WidgetWidth` — `1` — перенос строки при достижении ширины поля;
- `FixedPixelWidth` — `2` — перенос строки при достижении фиксированной ширины в пикселах, которую можно задать с помощью метода `setLineWrapColumnOrWidth()`;
- `FixedColumnWidth` — `3` — перенос строки при достижении фиксированной ширины в буквах, которую можно задать с помощью метода `setLineWrapColumnOrWidth()`;

➔ `setLineWrapColumnOrWidth(<Значение>)` — задает ширину колонки;

➔ `setWordWrapMode(<Режим>)` — задает режим переноса по словам. В качестве значения могут быть указаны следующие атрибуты из класса `QTextOption`:

- `NoWrap` — `0` — перенос по словам не производится;
- `WordWrap` — `1` — перенос строк только по словам;
- `ManualWrap` — `2` — аналогичен режиму `NoWrap`;
- `WrapAnywhere` — `3` — перенос строки может быть внутри слова;
- `WrapAtWordBoundaryOrAnywhere` — `4` — по возможности перенос по словам. Если это невозможно, то перенос строки может быть внутри слова;

➔ `setOverwriteMode(<Флаг>)` — если в качестве параметра указано значение `True`, то вводимый текст будет замещать ранее введенный. Значение `False` отключает замещение;

➔ `overwriteMode()` — возвращает значение `True`, если вводимый текст замещает ранее введенный, и `False` — в противном случае;

➔ `setAutoFormatting(<Режим>)` — задает режим автоматического форматирования. В качестве значения могут быть указаны следующие атрибуты из класса `QTextEdit`:

- `AutoNone` — автоматическое форматирование не используется;
- `AutoBulletList` — автоматически создавать маркированный список при вводе пользователем в начале строки символа `*`;
- `AutoAll` — включить все режимы. В Qt версии 4.7 эквивалентно режиму `AutoBulletList`;

➔ `setCursorWidth(<Ширина>)` — задает ширину текстового курсора;

➔ `setTabChangesFocus(<Флаг>)` — если в качестве параметра указано значение `False`, то с помощью нажатия клавиши <Tab> можно вставить символ табуляции в поле. Если указано значение `True`, то клавиша <Tab> используется для передачи фокуса между компонентами;

➔ `setTabStopWidth(<Ширина>)` — задает ширину символа табуляции в пикселах;

➔ `tabStopWidth()` — возвращает ширину символа табуляции в пикселах.

5.6.3. Изменение характеристик текста и фона

Для изменения характеристик текста и фона предназначены следующие методы из класса `QTextEdit` (перечислены только основные методы; полный список смотрите в документации):

➔ `setCurrentFont(<QFont>)` — задает текущий шрифт. Метод является слотом с сигнатурой `setCurrentFont(const QFont&)`. В качестве параметра указывается экземпляр класса `QFont`. Конструктор класса `QFont` имеет следующий формат:

```
<Шрифт> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1]
                [, italic=False])
```

В первом параметре указывается название шрифта в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: число от `0` до `99` или значение атрибутов `Light`, `Normal`, `DemiBold`, `Bold` или `Black` из класса `QFont`. Если в параметре `italic` указано значение `True`, то шрифт будет курсивным;

- `currentFont()` — возвращает экземпляр класса `QFont` с текущими характеристиками шрифта;
- `setFontFamily(<Название шрифта>)` — задает название текущего шрифта. Метод является слотом с сигнатурой `setFontFamily(const QString&);`
- `fontFamily()` — возвращает название текущего шрифта;
- `setFontPointSize(<Размер>)` — задает размер текущего шрифта. Метод является слотом с сигнатурой `setFontPointSize(qreal);`
- `fontPointSize()` — возвращает размер текущего шрифта;
- `setFontWeight(<Жирность>)` — задает жирность текущего шрифта. Метод является слотом с сигнатурой `setFontWeight(int);`
- `fontWeight()` — возвращает жирность текущего шрифта;
- `setFontItalic(<Флаг>)` — если в качестве параметра указано значение `True`, то шрифт будет курсивным. Метод является слотом с сигнатурой `setFontItalic(bool);`
- `fontItalic()` — возвращает `True`, если шрифт курсивный, и `False` — в противном случае;
- `setFontUnderline(<Флаг>)` — если в качестве параметра указано значение `True`, то текст будет подчеркнутым. Метод является слотом с сигнатурой `setFontUnderline(bool);`
- `fontUnderline()` — возвращает `True`, если текст подчеркнутый, и `False` — в противном случае;
- `setTextColor(<QColor>)` — задает цвет текущего текста. В качестве значения можно указать константу из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Метод является слотом с сигнатурой `setTextColor(const QColor&);`
- `textColor()` — возвращает экземпляр класса `QColor` с цветом текущего текста;

- `setTextBackgroundColor(<QColor>)` — задает цвет фона. В качестве значения можно указать атрибут из класса `QtCore.Qt` (например, `black`, `white` и т. д.) или экземпляр класса `QColor` (например, `QColor("red")`, `QColor("#ff0000")`, `QColor(255, 0, 0)` и др.). Метод является слотом с сигнатурой `setTextBackgroundColor(const QColor&);`
- `textBackgroundColor()` — возвращает экземпляр класса `QColor` с цветом фона;
- `setAlignment(<Выравнивание>)` — задает горизонтальное выравнивание текста внутри абзаца. Допустимые значения мы рассматривали в *разд. 4.2*. Метод является слотом с сигнатурой `setAlignment(Qt::Alignment);`
- `alignment()` — возвращает значение выравнивания текста внутри абзаца.

Задать формат символов можно также с помощью класса `QTextCharFormat`, который содержит дополнительные настройки. После создания экземпляра класса его следует передать в метод `setCurrentCharFormat(<QTextCharFormat>)`. Получить экземпляр класса с текущими настройками позволяет метод `currentCharFormat()`. За подробной информацией по классу `QTextCharFormat` обращайтесь к документации.

5.6.4. Класс QTextDocument

Класс `QTextDocument` реализует документ, который отображается в многострочном текстовом поле. Получить ссылку на текущий документ позволяет метод `document()` из класса `QTextEdit`. Установить новый документ можно с помощью метода `setDocument(<QTextDocument>)`. Иерархия наследования:

```
QObject — QTextDocument
```

Конструктор класса `QTextDocument` имеет два формата:

```
<Объект> = QTextDocument([parent=<Родитель>])
<Объект> = QTextDocument(<Текст>[, parent=<Родитель>])
```

В параметре `parent` указывается ссылка на родительский компонент. Параметр `<Текст>` позволяет задать текст в простом формате (не в HTML-формате), который будет отображен в текстовом поле.

Класс `QTextDocument` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPlainText(<Текст>)` — вставляет простой текст;
- `setHtml(<Текст>)` — вставляет текст в формате HTML;
- `toPlainText()` — возвращает простой текст, содержащийся в документе;

- `toHtml([<QByteArray>])` — возвращает текст в формате HTML. В качестве параметра можно указать кодировку документа, которая будет выведена в теге `<meta>`;
- `clear()` — удаляет весь текст из документа;
- `isEmpty()` — возвращает значение `True`, если документ пустой, и `False` — в противном случае;
- `isModified()` — возвращает значение `True`, если документ был изменен, и `False` — в противном случае;
- `undo()` — отменяет последнюю операцию ввода пользователем, при условии, что отмена возможна. Метод является слотом;
- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом;
- `isUndoAvailable()` — возвращает значение `True`, если можно отменить последнюю операцию ввода, и `False` — в противном случае;
- `isRedoAvailable()` — возвращает значение `True`, если можно повторить последнюю отмененную операцию ввода, и `False` — в противном случае;
- `setUndoRedoEnabled(<Флаг>)` — если в качестве значения указано значение `True`, то операции отмены и повтора действий разрешены, а если `False` — то запрещены;
- `isUndoRedoEnabled()` — возвращает значение `True`, если операции отмены и повтора действий разрешены, и `False` — если запрещены;
- `availableUndoSteps()` — возвращает количество возможных операций отмены;
- `availableRedoSteps()` — возвращает количество возможных повторов отмененных операций;
- `clearUndoRedoStacks([stacks=UndoAndRedoStacks])` — очищает список возможных отмен и/или повторов. В качестве параметра можно указать следующие атрибуты из класса `QTextDocument`:
  - `UndoStack` — только список возможных отмен;
  - `RedoStack` — только список возможных повторов;
  - `UndoAndRedoStacks` — очищаются оба списка;
- `print_(<QPrinter>)` — отправляет содержимое документа на печать. В качестве параметра указывается экземпляр класса `QPrinter`;
- `find()` — производит поиск фрагмента в документе. Метод возвращает экземпляр класса `QTextCursor`. Если фрагмент не найден, то объект курсора будет нулевым.

Проверить успешность операции можно с помощью метода `isNull()` объекта курсора. Форматы метода:

```
find(<Текст>[, position=0][, options=0])
find(<QRegExp>[, position=0][, options=0])
find(<Текст>, <QTextCursor>[, options=0])
find(<QRegExp>, <QTextCursor>[, options=0])
```

Параметр `<Текст>` задает искомый фрагмент, а параметр `<QRegExp>` позволяет указать регулярное выражение. По умолчанию обычный поиск производится без учета регистра символов в прямом направлении, начиная с позиции `position` или от текстового курсора, указанного в параметре `<QTextCursor>`. Поиск по регулярному выражению по умолчанию производится с учетом регистра символов. Чтобы поиск производился без учета регистра необходимо передать атрибут `QtCore.Qt.CaseInsensitive` в метод `setCaseSensitivity()`. В необязательном параметре `options` можно указать комбинацию (через оператор `|`) следующих атрибутов из класса `QTextDocument`:

- `FindBackward` — `1` — поиск в обратном направлении, а не в прямом;
- `FindCaseSensitively` — `2` — поиск с учетом регистра символов. При использовании регулярного выражения значение игнорируется;
- `FindWholeWords` — `4` — поиск слов целиком, а не фрагментов;

➔ `setDefaultFont(<QFont>)` — задает шрифт по умолчанию для документа. В качестве параметра указывается экземпляр класса `QFont`. Конструктор класса `QFont` имеет следующий формат:

```
<Шрифт> = QFont(<Название шрифта>[, pointSize=-1][, weight=-1]
                [, italic=False])
```

В первом параметре указывается название шрифта в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: число от `0` до `99` или значение атрибутов `Light`, `Normal`, `DemiBold`, `Bold` или `Black` из класса `QFont`. Если в параметре `italic` указано значение `True`, то шрифт будет курсивным;

➔ `setDefaultStyleSheet(<CSS>)` — устанавливает таблицу стилей CSS по умолчанию для документа;

➔ `setDocumentMargin(<Отступ>)` — задает отступ от краев поля до текста;

➔ `documentMargin()` — возвращает величину отступа от краев поля до текста;

➔ `setMaximumBlockCount(<Количество>)` — задает максимальное количество текстовых блоков в документе. Если количество блоков становится больше указанного значения, то первый блок будет удален;

- `maximumBlockCount()` — возвращает максимальное количество текстовых блоков;
- `characterCount()` — возвращает количество символов в документе;
- `lineCount()` — возвращает количество абзацев в документе;
- `blockCount()` — возвращает количество текстовых блоков в документе;
- `firstBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит первый текстовый блок документа;
- `lastBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит последний текстовый блок документа;
- `findBlock(<Индекс символа>)` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа, включающий символ с указанным индексом;
- `findBlockByNumber(<Индекс блока>)` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа с указанным индексом.

Класс `QTextDocument` содержит следующие сигналы:

- `undoAvailable(bool)` — генерируется при изменении возможности отменить операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно отменить операцию ввода, и `False` — в противном случае;
- `redoAvailable(bool)` — генерируется при изменении возможности повторить отмененную операцию ввода. Внутри обработчика через параметр доступно значение `True`, если можно повторить отмененную операцию ввода, и `False` — в противном случае.
- `undoCommandAdded()` — генерируется при добавлении операции ввода в список возможных отмен;
- `blockCountChanged(int)` — генерируется при изменении количества текстовых блоков. Внутри обработчика через параметр доступно новое количество текстовых блоков;
- `cursorPositionChanged(const QTextCursor&)` — генерируется при изменении позиции текстового курсора из-за операции редактирования. Обратите внимание на то, что при простом перемещении текстового курсора сигнал не генерируется;
- `contentsChange(int,int,int)` — генерируется при изменении текста. Внутри обработчика через первый параметр доступен индекс позиции внутри документа, через второй параметр — количество удаленных символов, а через третий параметр — количество добавленных символов;
- `contentsChanged()` — генерируется при любом изменении документа;
- `modificationChanged(bool)` — генерируется при изменении статуса документа.

5.6.5. Класс QTextCursor

Класс `QTextCursor` реализует текстовый курсор, выделение и позволяет изменять документ. Конструктор класса `QTextCursor` имеет следующие форматы:

```
<Объект> = QTextCursor()
<Объект> = QTextCursor(<QTextDocument>)
<Объект> = QTextCursor(<QTextFrame>)
<Объект> = QTextCursor(<QTextBlock>)
<Объект> = QTextCursor(<QTextCursor>)
```

Создать текстовый курсор, установить его в документе и управлять им позволяют следующие методы из класса `QTextEdit`:

- `textCursor()` — возвращает видимый в данный момент текстовый курсор (экземпляр класса `QTextCursor`). Чтобы изменения затронули текущий документ необходимо передать этот объект в метод `setTextCursor()`;
- `setTextCursor(<QTextCursor>)` — устанавливает текстовый курсор, ссылка на который указана в качестве параметра;
- `cursorForPosition(<QPoint>)` — возвращает текстовый курсор, который соответствует позиции, указанной в качестве параметра. Позиция задается с помощью экземпляра класса `QPoint` в координатах области;
- `moveCursor(<Позиция>[, mode=MoveAnchor])` — перемещает текстовый курсор внутри документа. В первом параметре можно указать следующие атрибуты из класса `QTextCursor`:
  - `NoMove` — `0` — не перемещать курсор;
  - `Start` — `1` — в начало документа;
  - `Up` — `2` — на одну строку вверх;
  - `StartOfLine` — `3` — в начало текущей строки;
  - `StartOfBlock` — `4` — в начало текущего текстового блока;
  - `StartOfWord` — `5` — в начало текущего слова;
  - `PreviousBlock` — `6` — в начало предыдущего текстового блока;
  - `PreviousCharacter` — `7` — сдвинуть на один символ влево;
  - `PreviousWord` — `8` — в начало предыдущего слова;
  - `Left` — `9` — сдвинуть на один символ влево;
  - `WordLeft` — `10` — влево на одно слово;
  - `End` — `11` — в конец документа;

- `Down` — `12` — на одну строку вниз;
- `EndOfLine` — `13` — в конец текущей строки;
- `EndOfWord` — `14` — в конец текущего слова;
- `EndOfBlock` — `15` — в конец текущего текстового блока;
- `NextBlock` — `16` — в начало следующего текстового блока;
- `NextCharacter` — `17` — сдвинуть на один символ вправо;
- `NextWord` — `18` — в начало следующего слова;
- `Right` — `19` — сдвинуть на один символ вправо;
- `WordRight` — `20` — в начало следующего слова.

Помимо перечисленных атрибутов существуют также атрибуты `NextCell`, `PreviousCell`, `NextRow` и `PreviousRow` позволяющие перемещать текстовый курсор внутри таблицы. В необязательном параметре `mode` можно указать следующие атрибуты из класса `QTextCursor`:

- `MoveAnchor` — `0` — если существует выделенный фрагмент, то выделение будет снято и текстовый курсор переместится в новое место (значение по умолчанию);
- `KeepAnchor` — `1` — фрагмент текста от старой позиции курсора до новой будет выделен.

Класс `QTextCursor` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

➔ `isNull()` — возвращает значение `True`, если объект курсора является нулевым (создан с помощью конструктора без параметра), и `False` — в противном случае;

➔ `setPosition(<Позиция>[, mode=MoveAnchor])` — перемещает текстовый курсор внутри документа. В первом параметре указывается позиция внутри документа. Необязательный параметр `mode` аналогичен одноименному параметру в методе `moveCursor()` из класса `QTextEdit`;

➔ `movePosition(<Позиция>[, mode=MoveAnchor][, n=1])` — перемещает текстовый курсор внутри документа. Параметры `<Позиция>` и `mode` аналогичны одноименным параметрам в методе `moveCursor()` из класса `QTextEdit`. Необязательный параметр `n` позволяет указать количество перемещений, например, переместить курсор на 10 символов вперед можно так:

```
cur = textEdit.textCursor()
cur.movePosition(QtGui.QTextCursor.NextCharacter,
                 mode=QtGui.QTextCursor.MoveAnchor, n=10)
textEdit.setTextCursor(cur)
```

Метод `movePosition()` возвращает значение `True`, если операция успешно выполнена указанное количество раз. Если было выполнено меньшее количество перемещений (например, из-за достижения конца документа), то метод возвращает значение `False`;

- `position()` — возвращает позицию текстового курсора внутри документа;
- `positionInBlock()` — возвращает позицию текстового курсора внутри блока;
- `block()` — возвращает экземпляр класса `QTextBlock`, который описывает текстовый блок, содержащий курсор;
- `blockNumber()` — возвращает индекс текстового блока, содержащего курсор;
- `atStart()` — возвращает значение `True`, если текстовый курсор находится в начале документа, и `False` — в противном случае;
- `atEnd()` — возвращает значение `True`, если текстовый курсор находится в конце документа, и `False` — в противном случае;
- `atBlockStart()` — возвращает значение `True`, если текстовый курсор находится в начале блока, и `False` — в противном случае;
- `atBlockEnd()` — возвращает значение `True`, если текстовый курсор находится в конце блока, и `False` — в противном случае;
- `select(<Режим>)` — выделяет фрагмент в документе в соответствии с указанным режимом. В качестве параметра можно указать следующие атрибуты из класса `QTextCursor`:
  - `WordUnderCursor` — `0` — выделяет слово, в котором расположен курсор;
  - `LineUnderCursor` — `1` — выделяет текущую строку;
  - `BlockUnderCursor` — `2` — выделяет текущий текстовый блок;
  - `Document` — `3` — выделяет весь документ;
- `hasSelection()` — возвращает значение `True`, если существует выделенный фрагмент, и `False` — в противном случае;
- `hasComplexSelection()` — возвращает значение `True`, если выделенный фрагмент содержит сложное форматирование, а не просто текст, и `False` — в противном случае;
- `clearSelection()` — снимает выделение;
- `selectionStart()` — возвращает начальную позицию выделенного фрагмента;
- `selectionEnd()` — возвращает конечную позицию выделенного фрагмента;

- `selectedText()` — возвращает текст выделенного фрагмента. Обратите внимание, если выделенный фрагмент занимает несколько строк, то вместо символа перевода строки вставляется символ с кодом `\u2029`. Попытка вывести этот символ в окно консоли приведет к исключению, поэтому следует произвести замену символа с помощью метода `replace()`:

  ```
  print(cur.selectedText().replace("\u2029", "\n"))
  ```

- `selection()` — возвращает экземпляр класса `QTextDocumentFragment`, который описывает выделенный фрагмент. Получить текст позволяют методы `toPlainText()` (возвращает простой текст) и `toHtml()` (возвращает текст в формате HTML) из этого класса;
- `removeSelectedText()` — удаляет выделенный фрагмент;
- `deleteChar()` — если нет выделенного фрагмента, то удаляет символ справа от курсора, в противном случае удаляет выделенный фрагмент;
- `deletePreviousChar()` — если нет выделенного фрагмента, то удаляет символ слева от курсора, в противном случае удаляет выделенный фрагмент;
- `beginEditBlock()` и `endEditBlock()` — задают начало и конец блока инструкций. Эти инструкции могут быть отменены или повторены как единое целое с помощью методов `undo()` и `redo()`;
- `joinPreviousEditBlock()` — делает последующие инструкции частью предыдущего блока инструкций;
- `setKeepPositionOnInsert(<Флаг>)` — если в качестве параметра указано значение `True`, то после операции вставки курсор сохранит свою предыдущую позицию. По умолчанию позиция курсора при вставке изменяется;
- `insertText(<Текст>[, <QTextCharFormat>])` — вставляет простой текст;
- `insertHtml(<Текст>)` — вставляет текст в формате HTML.

С помощью методов `insertBlock()`, `insertFragment()`, `insertFrame()`, `insertImage()`, `insertList()` и `insertTable()` можно вставить различные элементы, например, изображения, списки и др. Изменить формат выделенного фрагмента позволяют методы `mergeBlockCharFormat()`, `mergeBlockFormat()` и `mergeCharFormat()`. За подробной информацией по этим методам обращайтесь к документации.

5.7. Текстовый браузер

Класс `QTextBrowser` расширяет возможности класса `QTextEdit` и реализует текстовый браузер с возможностью перехода по гиперссылкам при щелчке мышью. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                                QAbstractScrollArea — QTextEdit — QTextBrowser
```

Формат конструктора класса `QTextBrowser`:

```
<Объект> = QTextBrowser([parent=<Родитель>])
```

Класс `QTextBrowser` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setSource(<QUrl>)` — загружает ресурс. В качестве параметра указывается экземпляр класса `QUrl`. Метод является слотом с сигнатурой `setSource(const QUrl&);`
- `source()` — возвращает экземпляр класса `QUrl` с адресом текущего ресурса;
- `reload()` — перезагружает текущий ресурс. Метод является слотом;
- `home()` — загружает первый ресурс из списка истории. Метод является слотом;
- `backward()` — загружает предыдущий ресурс из списка истории. Метод является слотом;
- `forward()` — загружает следующий ресурс из списка истории. Метод является слотом;
- `backwardHistoryCount()` — возвращает количество предыдущих ресурсов;
- `forwardHistoryCount()` — возвращает количество следующих ресурсов;
- `isBackwardAvailable()` — возвращает значение `True`, если существует предыдущий ресурс в списке истории, и `False` — в противном случае;
- `isForwardAvailable()` — возвращает значение `True`, если существует следующий ресурс в списке истории, и `False` — в противном случае;
- `clearHistory()` — очищает список истории;
- `historyTitle(<Число>)` — если в качестве параметра указано отрицательное число, то возвращает заголовок предыдущего ресурса, если `0` — то заголовок текущего ресурса, а если положительное число — то заголовок следующего ресурса;
- `historyUrl(<Число>)` — если в качестве параметра указано отрицательное число, то возвращает URL-адрес (экземпляр класса `QUrl`) предыдущего ресурса, если `0` — то URL-адрес текущего ресурса, а если положительное число — то URL-адрес следующего ресурса;
- `setOpenLinks(<Флаг>)` — если в качестве параметра указано значение `True`, то автоматический переход по гиперссылкам разрешен (значение по умолчанию). Значение `False` запрещает переход.

Класс `QTextBrowser` содержит следующие сигналы:

- `anchorClicked(const QUrl&)` — генерируется при переходе по гиперссылке. Внутри обработчика через параметр доступен URL-адрес гиперссылки;
- `backwardAvailable(bool)` — генерируется при изменении статуса списка предыдущих ресурсов. Внутри обработчика через параметр доступен статус;
- `forwardAvailable(bool)` — генерируется при изменении статуса списка следующих ресурсов. Внутри обработчика через параметр доступен статус;
- `highlighted(const QUrl&)` — генерируется при наведении указателя мыши на гиперссылку и выведении его. Внутри обработчика через параметр доступен URL-адрес ссылки (экземпляр класса `QUrl`) или пустой объект;
- `highlighted(const QString&)` — генерируется при наведении указателя мыши на гиперссылку и выведении его. Внутри обработчика через параметр доступен URL-адрес ссылки в виде строки или пустая строка;
- `historyChanged()` — генерируется при изменении списка истории;
- `sourceChanged(const QUrl&)` — генерируется при загрузке нового ресурса. Внутри обработчика через параметр доступен URL-адрес ресурса.

Примечание

Если возможностей класса `QTextBrowser` вам покажется мало, то обратите внимание на модуль `QtWebKit`, который содержит множество классов в совокупности реализующих полноценный браузер с поддержкой HTML, XHTML, CSS и JavaScript.

5.8. Поля для ввода целых и вещественных чисел

Для ввода чисел предназначены классы `QSpinBox` (поле для ввода целых чисел) и `QDoubleSpinBox` (поле для ввода вещественных чисел). Поля могут содержать две кнопки, которые позволяют увеличивать и уменьшать значение внутри поля с помощью щелчка мышью. Иерархия наследования:

`(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QSpinBox`

`(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDoubleSpinBox`

Форматы конструкторов классов `QSpinBox` и `QDoubleSpinBox`:

```
<Объект> = QSpinBox([parent=<Родитель>])
<Объект> = QDoubleSpinBox([parent=<Родитель>])
```

Классы `QSpinBox` и `QDoubleSpinBox` наследуют следующие методы из класса `QAbstractSpinBox` (перечислены только основные методы; полный список смотрите в документации):

- `setButtonSymbols(<Режим>)` — задает режим отображения кнопок, предназначенных для изменения значения поля с помощью мыши. Можно указать следующие атрибуты из класса `QAbstractSpinBox`:
  - `UpDownArrows` — `0` — отображаются кнопки со стрелками;
  - `PlusMinus` — `1` — отображаются кнопки с символами `+` и `-`. Обратите внимание на то, что при использовании некоторых стилей данное значение может быть проигнорировано;
  - `NoButtons` — `2` — кнопки не отображаются;
- `setAlignment(<Режим>)` — задает режим выравнивания значения внутри поля;
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `True`, то значение внутри поля будет изменяться по кругу при нажатии кнопок, например, максимальное значение сменится минимальным;
- `setSpecialValueText(<Строка>)` — позволяет задать строку, которая будет отображаться внутри поля вместо минимального значения;
- `setReadOnly(<Флаг>)` — если в качестве параметра указано значение `True`, то поле будет доступно только для чтения;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, то поле будет отображаться без рамки;
- `stepDown()` — уменьшает значение на одно приращение. Метод является слотом;
- `stepUp()` — увеличивает значение на одно приращение. Метод является слотом;
- `stepBy(<Количество>)` — увеличивает (при положительном значении) или уменьшает (при отрицательном значении) значение поля на указанное количество приращений;
- `text()` — возвращает текст, содержащийся внутри поля;
- `clear()` — очищает поле. Метод является слотом;
- `selectAll()` — выделяет все содержимое поля. Метод является слотом.

Класс `QAbstractSpinBox` содержит сигнал `editingFinished()`, который генерируется при потере полем фокуса ввода или нажатии клавиши <Enter>.

Классы `QSpinBox` и `QDoubleSpinBox` содержат следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setValue(<Число>)` — задает значение поля. Метод является слотом с сигнатурами `setValue(int)` и `setValue(double)`;
- `value()` — возвращает целое или вещественное число, содержащееся в поле;
- `cleanText()` — возвращает целое или вещественное число в виде строки;

- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное допустимые значения;
- `setPrefix(<Текст>)` — задает текст, который будет отображаться внутри поля перед значением;
- `setSuffix(<Текст>)` — задает текст, который будет отображаться внутри поля после значения;
- `setSingleStep(<Число>)` — задает число, которое будет прибавляться или вычитаться из текущего значения поля на каждом шаге.

Класс `QDoubleSpinBox` содержит также метод `setDecimals(<Количество>)`, который задает количество цифр после десятичной точки.

Классы `QSpinBox` и `QDoubleSpinBox` содержат сигналы `valueChanged(int)` (только в классе `QSpinBox`), `valueChanged(double)` (только в классе `QDoubleSpinBox`) и `valueChanged(const QString&)` которые генерируются при изменении значения внутри поля. Внутри обработчика через параметр доступно новое значение в виде числа или строки в зависимости от типа параметра.

5.9. Поля для ввода даты и времени

Для ввода даты и времени предназначены классы `QDateTimeEdit` (поле для ввода даты и времени), `QDateEdit` (поле для ввода даты) и `QTimeEdit` (поле для ввода времени). Поля могут содержать две кнопки, которые позволяют увеличивать и уменьшать значение внутри поля с помощью щелчка мышью. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit —
                          QDateEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit —
                          QTimeEdit
```

Форматы конструкторов классов:

```
<Объект> = QDateTimeEdit([parent=<Родитель>])
<Объект> = QDateTimeEdit(<QDateTime>[, parent=<Родитель>])
<Объект> = QDateTimeEdit(<QDate>[, parent=<Родитель>])
<Объект> = QDateTimeEdit(<QTime>[, parent=<Родитель>])
<Объект> = QDateEdit([parent=<Родитель>])
<Объект> = QDateEdit(<QDate>[, parent=<Родитель>])
<Объект> = QTimeEdit([parent=<Родитель>])
```

```
<Объект> = QTimeEdit(<QTime>[, parent=<Родитель>])
```

В параметре `<QDateTime>` можно указать экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python. Преобразовать экземпляр класса `QDateTime` в экземпляр класса `datetime` позволяет метод `toPyDateTime()`.

В качестве параметра `<QDate>` можно указать экземпляр класса `QDate` или экземпляр класса `date` из языка Python. Преобразовать экземпляр класса `QDate` в экземпляр класса `date` позволяет метод `toPyDate()`.

В параметре `<QTime>` можно указать экземпляр класса `QTime` или экземпляр класса `time` из языка Python. Преобразовать экземпляр класса `QTime` в экземпляр класса `time` позволяет метод `toPyTime()`.

Класс `QDateTimeEdit` наследует все методы из класса `QAbstractSpinBox` (см. *разд. 5.8*) и дополнительно реализует следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setDateTime(<QDateTime>)` — устанавливает дату и время. В качестве параметра указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python. Метод является слотом с сигнатурой `setDateTime(const QDateTime&)`;
- `setDate(<QDate>)` — устанавливает дату. В качестве параметра указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python. Метод является слотом с сигнатурой `setDate(const QDate&)`;
- `setTime(<QTime>)` — устанавливает время. В качестве параметра указывается экземпляр класса `QTime` или экземпляр класса `time` из языка Python. Метод является слотом с сигнатурой `setTime(const QTime&)`;
- `dateTime()` — возвращает экземпляр класса `QDateTime` с датой и временем;
- `date()` — возвращает экземпляр класса `QDate` с датой;
- `time()` — возвращает экземпляр класса `QTime` со временем;
- `setDateTimeRange(<Минимум>, <Максимум>)` — задает минимальное и максимальное допустимые значения для даты и времени. В параметрах указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python;
- `setMinimumDateTime(<Минимум>)` и `setMaximumDateTime(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты и времени. В параметрах указывается экземпляр класса `QDateTime` или экземпляр класса `datetime` из языка Python;
- `setDateRange(<Минимум>, <Максимум>)`, `setMinimumDate(<Минимум>)` и `setMaximumDate(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты. В параметрах указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python;

- `setTimeRange(<Минимум>, <Максимум>)`, `setMinimumTime(<Минимум>)` и `setMaximumTime(<Максимум>)` — задают минимальное и максимальное допустимые значения для времени. В параметрах указывается экземпляр класса `QTime` или экземпляр класса `time` из языка Python;
- `setDisplayFormat(<Формат>)` — задает формат отображения даты и времени. В качестве параметра указывается строка, содержащая специальные символы. Пример указания строки формата:
  `dateTimeEdit.setDisplayFormat("dd.MM.yyyy HH:mm:ss")`
- `setTimeSpec(<Зона>)` — задает зону времени. В качестве параметра можно указать атрибуты `LocalTime`, `UTC` или `OffsetFromUTC` из класса `QtCore.Qt`;
- `setCalendarPopup(<Флаг>)` — если в качестве параметра указано значение `True`, то дату можно будет выбрать с помощью календаря;
- `setSelectedSection(<Секция>)` — выделяет указанную секцию. В качестве параметра можно указать атрибуты `NoSection`, `DaySection`, `MonthSection`, `YearSection`, `HourSection`, `MinuteSection`, `SecondSection`, `MSecSection` или `AmPmSection` из класса `QDateTimeEdit`;
- `setCurrentSection(<Секция>)` — делает указанную секцию текущей;
- `setCurrentSectionIndex(<Индекс>)` — делает секцию с указанным индексом текущей;
- `currentSection()` — возвращает тип текущей секции;
- `currentSectionIndex()` — возвращает индекс текущей секции;
- `sectionCount()` — возвращает количество секций внутри поля;
- `sectionAt(<Индекс>)` — возвращает тип секции по указанному индексу;
- `sectionText(<Секция>)` — возвращает текст указанной секции.

При изменении внутри поля значений даты или времени генерируются сигналы `timeChanged(const QTime&)`, `dateChanged(const QDate&)` и `dateTimeChanged(const QDateTime&)`. Внутри обработчиков через параметр доступно новое значение.

Классы `QDateEdit` (поле для ввода даты) и `QTimeEdit` (поле для ввода времени) созданы для удобства и отличаются от класса `QDateTimeEdit` только форматом отображаемых данных. Эти классы наследуют методы базовых классов и не добавляют больше никаких своих методов.

5.10. Календарь

Класс `QCalendarWidget` реализует календарь с возможностью выбора даты и перемещения по месяцам с помощью мыши и клавиатуры. Иерархия наследования:

`(QObject, QPaintDevice) — QWidget — QCalendarWidget`

Формат конструктора класса `QCalendarWidget`:

`<Объект> = QCalendarWidget([parent=<Родитель>])`

Класс `QCalendarWidget` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setSelectedDate(<QDate>)` — устанавливает указанную дату. В качестве параметра указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python. Метод является слотом с сигнатурой `setSelectedDate(const QDate&);`
- `selectedDate()` — возвращает экземпляр класса `QDate` с выбранной датой;
- `setDateRange(<Минимум>, <Максимум>)`, `setMinimumDate(<Минимум>)` и `setMaximumDate(<Максимум>)` — задают минимальное и максимальное допустимые значения для даты. В параметрах указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python;
- `setCurrentPage(<Год>, <Месяц>)` — делает текущей страницу с указанным годом и месяцем. Выбранная дата при этом не изменяется. Метод является слотом с сигнатурой `setCurrentPage(int,int);`
- `monthShown()` — возвращает месяц (число от `1` до `12`), отображаемый на текущей странице;
- `yearShown()` — возвращает год, отображаемый на текущей странице;
- `showSelectedDate()` — отображает страницу с выбранной датой. Выбранная дата при этом не изменяется. Метод является слотом;
- `showToday()` — отображает страницу с сегодняшней датой. Выбранная дата при этом не изменяется. Метод является слотом;
- `showPreviousMonth()` — отображает страницу с предыдущим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showNextMonth()` — отображает страницу со следующим месяцем. Выбранная дата при этом не изменяется. Метод является слотом;
- `showPreviousYear()` — отображает страницу с текущим месяцем в предыдущем году. Выбранная дата при этом не изменяется. Метод является слотом;
- `showNextYear()` — отображает страницу с текущим месяцем в следующем году. Выбранная дата при этом не изменяется. Метод является слотом;

- `setFirstDayOfWeek(<День>)` — задает первый день недели. По умолчанию используется воскресенье. Чтобы первым днем недели сделать понедельник следует в качестве параметра указать атрибут `Monday` из класса `QtCore.Qt`;
- `setNavigationBarVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то панель навигации выводиться не будет. Метод является слотом с сигнатурой `setNavigationBarVisible(bool)`;
- `setHorizontalHeaderFormat(<Формат>)` — задает формат горизонтального заголовка. В качестве параметра можно указать следующие атрибуты из класса `QCalendarWidget`:
  - `NoHorizontalHeader` — `0` — заголовок не отображается;
  - `SingleLetterDayNames` — `1` — отображается только первая буква из названия дня недели;
  - `ShortDayNames` — `2` — отображается сокращенное название дня недели;
  - `LongDayNames` — `3` — отображается полное название дня недели;
- `setVerticalHeaderFormat(<Формат>)` — задает формат вертикального заголовка. В качестве параметра можно указать следующие атрибуты из класса `QCalendarWidget`:
  - `NoVerticalHeader` — `0` — заголовок не отображается;
  - `ISOWeekNumbers` — `1` — отображается номер недели в году;
- `setGridVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то будут отображены линии сетки;
- `setSelectionMode(<Режим>)` — задает режим выделения даты. В качестве параметра можно указать следующие атрибуты из класса `QCalendarWidget`:
  - `NoSelection` — `0` — дата не может быть выбрана пользователем;
  - `SingleSelection` — `1` — может быть выбрана одна дата;
- `setHeaderTextFormat(<QTextCharFormat>)` — задает формат ячеек заголовка. В параметре указывается экземпляр класса `QTextCharFormat`;
- `setWeekdayTextFormat(<День недели>, <QTextCharFormat>)` — задает формат ячеек для указаного дня недели. В первом параметре указываются атрибуты `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday` или `Sunday` из класса `QtCore.Qt`, а во втором параметре — экземпляр класса `QTextCharFormat`;
- `setDateTextFormat(<QDate>, <QTextCharFormat>)` — задает формат ячейки с указанной датой. В первом параметре указывается экземпляр класса `QDate` или экземпляр класса `date` из языка Python, а во втором параметре — экземпляр класса `QTextCharFormat`.

Листинги на странице http://unicross.narod.ru/pyqt/

Класс QCalendarWidget содержит следующие сигналы:

- activated(const QDate&) — генерируется при двойном щелчке мышью или нажатии клавиши <Enter>. Внутри обработчика через параметр доступна дата;
- clicked(const QDate&) — генерируется при щелчке мышью на доступной дате. Внутри обработчика через параметр доступна выбранная дата;
- currentPageChanged(int,int) — генерируется при изменении страницы. Внутри обработчика через первый параметр доступен год, а через второй — месяц;
- selectionChanged() — генерируется при изменении выбранной даты пользователем или из программы.

5.11. Электронный индикатор

Класс QLCDNumber реализует электронный индикатор, в котором цифры и буквы отображаются отдельными сегментами, как на электронных часах или дисплее калькулятора. Индикатор позволяет отображать числа в двоичной, восьмеричной, десятичной и шестнадцатеричной системах исчисления. Иерархия наследования выглядит так:

(QObject, QPaintDevice) — QWidget — QFrame — QLCDNumber

Форматы конструктора класса QLCDNumber:

```
<Объект> = QLCDNumber([parent=<Родитель>])
<Объект> = QLCDNumber(<Количество цифр>[, parent=<Родитель>])
```

В параметре <Количество цифр> указывается количество отображаемых цифр. Если параметр не указан, то по умолчанию используется значение 5.

Класс QLCDNumber содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- display(<Значение>) — задает новое значение. В качестве параметра можно указать целое число, вещественное число или строку. Метод является слотом с сигнатурами display(int), display(double) и display(const QString&);
- checkOverflow(<Число>) — возвращает значение True, если целое или вещественное число, указанное в параметре, не может быть отображено индикатором. В противном случае возвращает значение False;
- intValue() — возвращает значение индикатора в виде целого числа;
- value() — возвращает значение индикатора в виде вещественного числа;
- setSegmentStyle(<Стиль>) — задает стиль индикатора. В качестве параметра можно указать атрибуты Outline, Filled или Flat из класса QLCDNumber;

➔ `setMode(<Режим>)` — задает режим отображения чисел. В качестве параметра можно указать следующие атрибуты из класса `QLCDNumber`:

- `Hex` — `0` — шестнадцатеричное значение;
- `Dec` — `1` — десятичное значение;
- `Oct` — `2` — восьмеричное значение;
- `Bin` — `3` — двоичное значение.

Вместо метода `setMode()` удобнее воспользоваться слотами `setHexMode()`, `setDecMode()`, `setOctMode()` и `setBinMode()`;

➔ `setSmallDecimalPoint(<Флаг>)` — если в качестве параметра указано значение `True`, то десятичная точка будет отображаться как отдельный элемент (при этом значение выводится более компактно без пробелов до и после точки), а если значение `False` — то десятичная точка будет занимать позицию цифры (значение используется по умолчанию). Метод является слотом с сигнатурой `setSmallDecimalPoint(bool)`;

➔ `setDigitCount(<Число>)` — задает количество отображаемых цифр. Если в методе `setSmallDecimalPoint()` указано значение `False`, то десятичная точка считается одной цифрой.

Класс `QLCDNumber` содержит сигнал `overflow()`, который генерируется при попытке задать значение, которое не может быть отображено индикатором.

5.12. Индикатор хода процесса

Класс `QProgressBar` реализует индикатор хода процесса, с помощью которого можно информировать пользователя о текущем состоянии выполнения длительной операции. Иерархия наследования выглядит так:

`(QObject, QPaintDevice) — QWidget — QProgressBar`

Формат конструктора класса `QProgressBar`:

```
<Объект> = QProgressBar([parent=<Родитель>])
```

Класс `QProgressBar` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

➔ `setValue(<Значение>)` — задает новое значение. Метод является слотом с сигнатурой `setValue(int)`;

➔ `value()` — возвращает текущее значение индикатора в виде числа;

➔ `text()` — возвращает текст, отображаемый на индикаторе или рядом с ним;

- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное значения. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы являются слотами с сигнатурами `setRange(int,int)`, `setMinimum(int)` и `setMaximum(int)`;
- `reset()` — сбрасывает значение индикатора. Метод является слотом;
- `setOrientation(<Ориентация>)` — задает ориентацию индикатора. В качестве значения указываются атрибуты `Horizontal` или `Vertical` из класса `QtCore.Qt`. Метод является слотом с сигнатурой `setOrientation(Qt::Orientation)`;
- `setTextVisible(<Флаг>)` — если в качестве параметра указано значение `False`, то текст с текущим значением индикатора отображаться не будет;
- `setTextDirection(<Направление>)` — задает направление вывода текста при вертикальной ориентации индикатора. Обратите внимание на то, что при использовании стилей "`windows`", "`windowsxp`" и "`macintosh`" при вертикальной ориентации текст вообще не отображается. В качестве значения указываются следующие атрибуты из класса `QProgressBar`:
  - `TopToBottom` — `0` — текст поворачивается на 90 градусов по часовой стрелке;
  - `BottomToTop` — `1` — текст поворачивается на 90 градусов против часовой стрелки;
- `setInvertedAppearance(<Флаг>)` — если в качестве параметра указано значение `True`, то направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево при горизонтальной ориентации).

При изменении значения индикатора генерируется сигнал `valueChanged(int)`. Внутри обработчика через параметр доступно новое значение.

5.13. Шкала с ползунком

Класс `QSlider` реализует шкалу с ползунком, который можно перемещать с помощью указателя мыши или клавиатуры. Иерархия наследования выглядит так:

`(QObject, QPaintDevice) — QWidget — QAbstractSlider — QSlider`

Форматы конструктора класса `QSlider`:

```
<Объект> = QSlider([parent=<Родитель>])
<Объект> = QSlider(<Ориентация>[, parent=<Родитель>])
```

Параметр `<Ориентация>` позволяет задать ориентацию шкалы. В качестве значения указываются атрибуты `Horizontal` или `Vertical` (значение по умолчанию) из класса `QtCore.Qt`.

Класс `QSlider` наследует следующие методы из класса `QAbstractSlider` (перечислены только основные методы; полный список смотрите в документации):

- `setValue(<Значение>)` — задает новое значение. Метод является слотом с сигнатурой `setValue(int)`;
- `value()` — возвращает установленное положение ползунка в виде числа;
- `setSliderPosition(<Значение>)` — задает текущее положение ползунка;
- `sliderPosition()` — возвращает текущее положение ползунка в виде числа. Если отслеживание перемещения ползунка включено (по умолчанию), то возвращаемое значение будет совпадать со значением, возвращаемым методом `value()`. Если отслеживание выключено, то при перемещении метод `sliderPosition()` вернет текущее положение, а метод `value()` — положение, которое имел ползунок до перемещения;
- `setRange(<Минимум>, <Максимум>)`, `setMinimum(<Минимум>)` и `setMaximum(<Максимум>)` — задают минимальное и максимальное значения;
- `setOrientation(<Ориентация>)` — задает ориентацию шкалы. В качестве значения указываются атрибуты `Horizontal` или `Vertical` из класса `QtCore.Qt`;
- `setSingleStep(<Значение>)` — задает значение, на которое сдвинется ползунок при нажатии клавиш со стрелками;
- `setPageStep(<Значение>)` — задает значение, на которое сдвинется ползунок при нажатии клавиш <PageUp> и <PageDown>, повороте колесика мыши или щелчке мышью на шкале;
- `setInvertedAppearance(<Флаг>)` — если в качестве параметра указано значение `True`, то направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево при горизонтальной ориентации).
- `setInvertedControls(<Флаг>)` — если в качестве параметра указано значение `False`, то при изменении направления увеличения значения будет изменено и направление перемещения ползунка при нажатии клавиш <Page Up> и <Page Down>, повороте колесика мыши и нажатии клавиш со стрелками вверх и вниз;
- `setTracking(<Флаг>)` — если в качестве параметра указано значение `True`, то отслеживание перемещения ползунка будет включено (значение по умолчанию). При этом сигнал `valueChanged(int)` будет генерироваться постоянно при перемещении ползунка. Если в качестве параметра указано значение `False`, то сигнал `valueChanged(int)` будет сгенерирован только при отпускании ползунка;

➔ `hasTracking()` — возвращает значение `True`, если отслеживание перемещения ползунка включено, и `False` — в противном случае.

Класс `QAbstractSlider` содержит следующие сигналы:

➔ `actionTriggered(int)` — генерируется, когда производится взаимодействие с ползунком, например, при нажатии клавиши <Page Up>. Внутри обработчика через параметр доступно произведенное действие, которое описывается атрибутами `SliderNoAction` (`0`), `SliderSingleStepAdd` (`1`), `SliderSingleStepSub` (`2`), `SliderPageStepAdd` (`3`), `SliderPageStepSub` (`4`), `SliderToMinimum` (`5`), `SliderToMaximum` (`6`) и `SliderMove` (`7`) из класса `QAbstractSlider`;

➔ `rangeChanged(int,int)` — генерируется при изменении диапазона значений. Внутри обработчика через первый параметр доступно новое минимальное значение, а через второй параметр — новое максимальное значение;

➔ `sliderPressed()` — генерируется при нажатии ползунка;

➔ `sliderMoved(int)` — генерируется постоянно при перемещении ползунка. Внутри обработчика через параметр доступно новое положение ползунка;

➔ `sliderReleased()` — генерируется при отпускании ранее нажатого ползунка;

➔ `valueChanged(int)` — генерируется при изменении значения. Внутри обработчика через параметр доступно новое значение.

Класс `QSlider` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

➔ `setTickPosition(<Позиция>)` — задает позицию рисок. В качестве параметра указываются следующие атрибуты из класса `QSlider`:

- `NoTicks` — без рисок;
- `TicksBothSides` — риски по обе стороны;
- `TicksAbove` — риски выводятся сверху;
- `TicksBelow` — риски выводятся снизу;
- `TicksLeft` — риски выводятся слева;
- `TicksRight` — риски выводятся справа;

➔ `setTickInterval(<Расстояние>)` — задает расстояние между рисками.

5.14. Класс QDial

Класс `QDial` реализует круглую шкалу с ползунком круглой или треугольной формы (вид зависит от используемого стиля), который можно перемещать по кругу с помощью

указателя мыши или клавиатуры. Компонент напоминает регулятор, используемый в различных устройствах для изменения или отображения каких-либо настроек. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QDial
```

Формат конструктора класса `QDial`:

```
<Объект> = QDial([parent=<Родитель>])
```

Класс `QDial` наследует все методы и сигналы из класса `QAbstractSlider` (см. *разд. 5.13*) и содержит несколько дополнительных методов (перечислены только основные методы; полный список смотрите в документации):

- `setNotchesVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то риски будут отображены. По умолчанию риски не выводятся. Метод является слотом с сигнатурой `setNotchesVisible(bool)`;
- `setNotchTarget(<Значение>)` — задает рекомендуемое количество пикселов между рисками. В качестве параметра указывается вещественное число;
- `setWrapping(<Флаг>)` — если в качестве параметра указано значение `True`, то начало шкалы будет совпадать с ее концом. По умолчанию между началом шкалы и концом расположено пустое пространство. Метод является слотом с сигнатурой `setWrapping(bool)`.

5.15. Полоса прокрутки

Класс `QScrollBar` реализует горизонтальную и вертикальную полосу прокрутки. Изменить значение можно с помощью нажатия кнопок, расположенных по краям полосы, щелчка мышью на полосе, путем перемещения ползунка, нажатия клавиш на клавиатуре, а также выбрав соответствующий пункт из контекстного меню. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QScrollBar
```

Форматы конструктора класса `QScrollBar`:

```
<Объект> = QScrollBar([parent=<Родитель>])
<Объект> = QScrollBar(<Ориентация>[, parent=<Родитель>])
```

Параметр `<Ориентация>` позволяет задать ориентацию полосы прокрутки. В качестве значения указываются атрибуты `Horizontal` или `Vertical` (значение по умолчанию) из класса `QtCore.Qt`.

Класс `QScrollBar` наследует все методы и сигналы из класса `QAbstractSlider` (см. *разд. 5.13*) и не содержит дополнительных методов.

Примечание

Полоса прокрутки редко используется отдельно. Гораздо удобнее воспользоваться областью с полосами прокрутки, которую реализует класс `QScrollArea` (см. *разд. 4.12*).

Глава 6. Списки и таблицы

В PyQt имеется широкий выбор компонентов, позволяющих отображать как одномерный список строк (в свернутом или развернутом состояниях), так и табличные данные. Кроме того, можно отобразить данные, которые имеют очень сложную структуру, например, иерархическую. Благодаря поддержке концепции "модель/представление", позволяющей отделить данные от их представления, одни и те же данные можно отображать сразу в нескольких компонентах без их дублирования.

6.1. Раскрывающийся список

Класс `QComboBox` реализует раскрывающийся список с возможностью выбора одного пункта. При щелчке мышью на поле появляется список возможных вариантов, а при выборе пункта список сворачивается. Иерархия наследования выглядит так:

`(QObject, QPaintDevice) — QWidget — QComboBox`

Формат конструктора класса `QComboBox`:

```
<Объект> = QComboBox([parent=<Родитель>])
```

6.1.1. Добавление, изменение и удаление элементов

Для добавления, изменения, удаления и получения значения элементов предназначены следующие методы из класса `QComboBox`:

- `addItem()` — добавляет один элемент в конец списка. Форматы метода:

  ```
  addItem(<Строка>[, <Данные>])
  addItem(<QIcon>, <Строка>[, <Данные>])
  ```

  В параметре `<Строка>` задается текст элемента списка, а в параметре `<QIcon>` — иконка, которая будет отображена перед текстом. Необязательный параметр `<Данные>` позволяет сохранить пользовательские данные, например, индекс в таблице базы данных;

- `addItems(<Список строк>)` — добавляет несколько элементов в конец списка;

- `insertItem()` — вставляет один элемент в указанную позицию списка. Все остальные элементы сдвигаются в конец списка. Форматы метода:

  ```
  insertItem(<Индекс>, <Строка>[, <Данные>])
  insertItem(<Индекс>, <QIcon>, <Строка>[, <Данные>])
  ```

- `insertItems(<Индекс>, <Список строк>)` — вставляет несколько элементов в указанную позицию списка. Все остальные элементы сдвигаются в конец списка;

- `insertSeparator(<Индекс>)` — вставляет разделительную линию в указанную позицию;
- `setItemText(<Индекс>, <Строка>)` — изменяет текст элемента с указанным индексом;
- `setItemIcon(<Индекс>, <QIcon>)` — изменяет иконку элемента с указанным индексом;
- `setItemData(<Индекс>, <Данные>[, role=UserRole])` — изменяет данные для элемента с указанным индексом. Необязательный параметр `role` позволяет указать роль, для которой задаются данные. Например, если указать атрибут `ToolTipRole` из класса `QtCore.Qt`, то данные задают текст вплывающей подсказки, которая будет отображена при наведении указателя мыши на элемент. По умолчанию изменяются пользовательские данные;
- `setCurrentIndex(<Индекс>)` — делает элемент с указанным индексом текущим. Метод является слотом с сигнатурой `setCurrentIndex(int)`;
- `currentIndex()` — возвращает индекс текущего элемента;
- `currentText()` — возвращает текст текущего элемента;
- `itemText(<Индекс>)` — возвращает текст элемента с указанным индексом;
- `itemData(<Индекс>[, role=UserRole])` — возвращает данные, сохраненные в роли `role` элемента с индексом `<Индекс>`;
- `count()` — возвращает общее количество элементов списка. Получить количество элементов можно также с помощью функции `len()`;
- `removeItem(<Индекс>)` — удаляет элемент с указанным индексом;
- `clear()` — удаляет все элементы списка.

6.1.2. Изменение настроек

Управлять настройками раскрывающегося списка позволяют следующие методы:

- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь сможет добавлять новые элементы в список, путем ввода текста в поле и последующего нажатия клавиши <Enter>;
- `setInsertPolicy(<Режим>)` — задает режим добавления нового элемента пользователем. В качестве параметра указываются следующие атрибуты из класса `QComboBox`:
  - `NoInsert` — `0` — элемент не будет добавлен;
  - `InsertAtTop` — `1` — элемент вставляется в начало списка;

- `InsertAtCurrent` — 2 — будет изменен текст текущего элемента;
- `InsertAtBottom` — 3 — элемент вставляется в конец списка;
- `InsertAfterCurrent` — 4 — элемент вставляется после текущего элемента;
- `InsertBeforeCurrent` — 5 — элемент вставляется перед текущим элементом;
- `InsertAlphabetically` — 6 — при вставке учитывается алфавитный порядок следования элементов;

➔ `setEditText(<Текст>)` — вставляет текст в поле редактирования. Метод является слотом с сигнатурой `setEditText(const QString&)`;

➔ `clearEditText()` — удаляет текст из поля редактирования. Метод является слотом;

➔ `setAutoCompletion(<Флаг>)` — если в качестве параметра указано значение `True`, то режим отображения возможных вариантов, начинающихся с введенных букв, будет включен. Значение `False` отключает отображение вариантов;

➔ `setCompleter(<QCompleter>)` — позволяет предлагать возможные варианты значений, начинающихся с введенных пользователем символов. В качестве параметра указывается экземпляр класса `QCompleter`;

➔ `setAutoCompletionCaseSensitivity(<Режим>)` — задает режим учета регистра символов при отображении возможных вариантов. При указании атрибута `CaseInsensitive` из класса `QtCore.Qt` регистр букв учитываться не будет. Атрибут `CaseSensitive` задает регистрозависимый режим;

➔ `autoCompletion()` — возвращает значение `True`, если режим отображения возможных вариантов включен, и `False` — в противном случае;

➔ `setValidator(<QValidator>)` — устанавливает контроль ввода. В качестве значения указывается экземпляр класса, наследующего класс `QValidator` (см. *разд. 5.5.3*);

➔ `setDuplicatesEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может добавить элемент с повторяющимся текстом. По умолчанию повторы запрещены;

➔ `setMaxCount(<Количество>)` — задает максимальное количество элементов в списке. Если до вызова метода количество элементов превышало указанное количество, то лишние элементы будут удалены;

➔ `setMaxVisibleItems(<Количество>)` — задает максимальное количество видимых элементов в раскрывающемся списке;

- `setMinimumContentsLength(<Количество>)` — задает минимальное количество отображаемых символов;
- `setSizeAdjustPolicy(<Режим>)` — устанавливает режим изменения ширины при изменении содержимого. В качестве параметра указываются следующие атрибуты из класса `QComboBox`:
  - `AdjustToContents` — `0` — ширина будет соответствовать содержимому;
  - `AdjustToContentsOnFirstShow` — `1` — ширина будет соответствовать ширине, используемой при первом отображении списка;
  - `AdjustToMinimumContentsLength` — `2` — `AdjustToContents` или `AdjustToContentsOnFirstShow`;
  - `AdjustToMinimumContentsLengthWithIcon` — `3` — используется значение минимальной ширины, которое установлено с помощью метода `setMinimumContentsLength()`, плюс ширина иконки;
- `setFrame(<Флаг>)` — если в качестве параметра указано значение `False`, то поле будет отображаться без рамки;
- `setIconSize(<QSize>)` — задает максимальный размер иконок;
- `showPopup()` — отображает список;
- `hidePopup()` — скрывает список.

6.1.3. Поиск элемента внутри списка

Произвести поиск элемента внутри списка позволяют методы `findText()` (поиск в тексте элемента) и `findData()` (поиск данных в указанной роли). Методы возвращают индекс найденного элемента или значение `-1`, если элемент не найден. Форматы методов:

```
findText(<Текст>[,  flags=MatchExactly | MatchCaseSensitive])
findData(<Данные>[, role=UserRole][,
                    flags=MatchExactly | MatchCaseSensitive])
```

Параметр `flags` задает режим поиска. В качестве значения можно указать комбинацию (через оператор `|`) следующих атрибутов из класса `QtCore.Qt`:

- `MatchExactly` — `0` — поиск полного соответствия;
- `MatchFixedString` — `8` — поиск полного соответствия внутри строки, выполняемый по умолчанию без учета регистра символов;
- `MatchContains` — `1` — поиск совпадения с любой частью;
- `MatchStartsWith` — `2` — совпадение с началом;

- `MatchEndsWith` — `3` — совпадение с концом;
- `MatchRegExp` — `4` — поиск с помощью регулярного выражения;
- `MatchWildcard` — `5` — используются подстановочные знаки;
- `MatchCaseSensitive` — `16` — поиск с учетом регистра символов;
- `MatchWrap` — `32` — поиск по кругу;
- `MatchRecursive` — `64` — просмотр всей иерархии.

6.1.3. Сигналы

Класс `QComboBox` содержит следующие сигналы:

- `activated(int)` и `activated(const QString&)` — генерируется при выборе пункта в списке (даже, если индекс не изменился) пользователем. Внутри обработчика доступен индекс или текст элемента;
- `currentIndexChanged(int)` и `currentIndexChanged(const QString&)` — генерируется при изменении текущего индекса. Внутри обработчика доступен индекс (значение `-1`, если список пуст) или текст элемента;
- `editTextChanged(const QString&)` — генерируется при изменении текста в поле. Внутри обработчика через параметр доступен новый текст;
- `highlighted(int)` и `highlighted(const QString&)` — генерируется при наведении указателя мыши на пункт в списке. Внутри обработчика доступен индекс или текст элемента.

6.2. Список для выбора шрифта

Класс `QFontComboBox` реализует раскрывающийся список с названиями шрифтов. Шрифт можно выбрать из списка или ввести название в поле, при этом будут отображаться названия, начинающиеся с введенных букв. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QComboBox — QFontComboBox
```

Формат конструктора класса `QFontComboBox`:

```
<Объект> = QFontComboBox([parent=<Родитель>])
```

Класс `QFontComboBox` наследует все методы и сигналы из класса `QComboBox` (см. *разд. 6.1*) и содержит несколько дополнительных методов:

- `setCurrentFont(<QFont>)` — делает текущим элемент, соответствующий указанному шрифту. В качестве параметра указывается экземпляр класса `QFont`. Метод является слотом с сигнатурой `setCurrentFont(const QFont&)`. Пример:

```
comboBox.setCurrentFont(QtGui.QFont("Verdana"))
```

➔ `currentFont()` — возвращает экземпляр класса `QFont`, с выбранным шрифтом. Пример вывода названия шрифта:

```
print(comboBox.currentFont().family())
```

➔ `setFontFilters(<Фильтр>)` — ограничивает список указанными типами шрифтов. В качестве параметра указывается комбинация следующих атрибутов из класса `QFontComboBox`:

- `AllFonts` — `0` — все типы шрифтов;
- `ScalableFonts` — `1` — масштабируемые шрифты;
- `NonScalableFonts` — `2` — не масштабируемые шрифты;
- `MonospacedFonts` — `4` — моноширинные шрифты;
- `ProportionalFonts` — `8` — пропорциональные шрифты.

Класс `QFontComboBox` содержит сигнал `currentFontChanged(const QFont&)`, который генерируется при изменении текущего шрифта. Внутри обработчика доступен экземпляр класса `QFont`, с текущим шрифтом.

6.3. Роли элементов

Каждый элемент списка содержит данные, распределенные по ролям. С помощью этих данных можно указать текст элемента, каким шрифтом и цветом отображается текст, задать текст всплывающей подсказки и многое другое. Перечислим роли элементов (атрибуты из класса `QtCore.Qt`):

➔ `DisplayRole` — `0` — отображаемые данные (обычно текст);

➔ `DecorationRole` — `1` — изображение (обычно иконка);

➔ `EditRole` — `2` — данные в виде, удобном для редактирования;

➔ `ToolTipRole` — `3` — текст всплывающей подсказки;

➔ `StatusTipRole` — `4` — текст для строки состояния;

➔ `WhatsThisRole` — `5` — текст для справки;

➔ `FontRole` — `6` — шрифт элемента. Указывается экземпляр класса `QFont`;

➔ `TextAlignmentRole` — `7` — выравнивание текста внутри элемента;

➔ `BackgroundRole` — `8` — фон элемента. Указывается экземпляр класса `QBrush`;

➔ `ForegroundRole` — `9` — цвет текста. Указывается экземпляр класса `QBrush`;

- `CheckStateRole` — `10` — статус флажка. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок частично установлен;
  - `Checked` — `2` — флажок установлен;
- `AccessibleTextRole` — `11` — текст для плагинов;
- `AccessibleDescriptionRole` — `12` — описание элемента;
- `SizeHintRole` — `13` — рекомендуемый размер элемента. Указывается экземпляр класса `QSize`;
- `UserRole` — `32` — любые пользовательские данные, например, индекс элемента в базе данных. Можно сохранить несколько данных, указав их в роли с индексом более `32`. Пример:

```
comboBox.setItemData(0, 50, role=QtCore.Qt.UserRole)
comboBox.setItemData(0, "Другие данные",
                            role=QtCore.Qt.UserRole + 1)
```

6.4. Модели

Для отображения данных в виде списков и таблиц применяется концепция "модель/представление", позволяющая отделить данные от их представления и избежать дублирования данных. В основе концепции лежат следующие составляющие:

- *модель* — является "оберткой" над данными. Позволяет добавлять, изменять и удалять данные, а также содержит методы для чтения данных и управления ими;
- *представление* — предназначено для отображения элементов модели. В нескольких представлениях можно установить одну модель;
- *модель выделения* — позволяет управлять выделением. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении;
- *промежуточная модель* — является прослойкой между моделью и представлением. Позволяет производить сортировку и фильтрацию данных на основе базовой модели без изменения порядка следования элементов в базовой модели;
- *делегат* — выполняет рисование каждого элемента в отдельности, а также позволяет произвести редактирование данных. В своих программах вы можете наследовать стандартные классы делегатов и полностью управлять отображением

данных и их редактированием. За подробной информацией по классам делегатов обращайтесь к документации.

6.4.1. Доступ к данным внутри модели

Доступ к данным внутри модели реализуется с помощью класса `QModelIndex`. Формат конструктора класса:

```
<Объект> = QModelIndex([<QModelIndex или QPersistentModelIndex>])
```

Если параметр не указан, то создается не валидный объект, который ни на что не указывает. Такой объект обычно возвращается, когда элемента не существует в модели. Наиболее часто экземпляр класса `QModelIndex` создается с помощью метода `index()` из класса модели или метода `currentIndex()` из класса `QAbstractItemView`.

Класс `QModelIndex` содержит следующие методы:

- `isValid()` — возвращает значение `True`, если объект является валидным, и `False` — в противном случае;
- `data([DisplayRole])` — возвращает данные, хранящиеся в указанной роли;
- `flags()` — содержит свойства элемента. Возвращает комбинацию следующих атрибутов из класса `QtCore.Qt`:
  - `NoItemFlags` — `0` — свойства не установлены;
  - `ItemIsSelectable` — `1` — можно выделить;
  - `ItemIsEditable` — `2` — можно редактировать;
  - `ItemIsDragEnabled` — `4` — можно перетаскивать;
  - `ItemIsDropEnabled` — `8` — можно сбрасывать перетаскиваемые данные;
  - `ItemIsUserCheckable` — `16` — может быть включен и выключен;
  - `ItemIsEnabled` — `32` — пользователь может взаимодействовать с элементом;
  - `ItemIsTristate` — `64` — флажок имеет три состояния;
- `row()` — возвращает индекс строки;
- `column()` — возвращает индекс столбца;
- `parent()` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на один уровень выше по иерархии. Если элемента нет, то возвращается не валидный экземпляр класса `QModelIndex`;
- `child(<Строка>, <Столбец>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на один уровень ниже по иерархии на указанной

позиции. Если элемента нет, то возвращается не валидный экземпляр класса `QModelIndex`;

- `sibling(<Строка>, <Столбец>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на том же уровне вложенности на указанной позиции. Если элемента нет, то возвращается не валидный экземпляр класса `QModelIndex`;
- `model()` — возвращает ссылку на модель.

Следует учитывать, что модель может измениться и экземпляр класса `QModelIndex` будет ссылаться на несуществующий уже элемент. Если необходимо сохранить ссылку на элемент, то следует воспользоваться классом `QPersistentModelIndex`, который содержит те же самые методы, но обеспечивает валидность ссылки.

6.4.2. Класс QStringListModel

Класс `QStringListModel` реализует одномерную модель, содержащую список строк. Модель можно отобразить с помощью классов `QListView` и `QComboBox`, передав в метод `setModel()`. Иерархия наследования:

`QObject — QAbstractItemModel — QAbstractListModel — QStringListModel`

Форматы конструктора класса `QStringListModel`:

```
<Объект> = QStringListModel([parent=<Родитель>])
<Объект> = QStringListModel(<Список строк>[, parent=<Родитель>])
```

Класс `QStringListModel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setStringList(<Список строк>)` — устанавливает список строк;
- `stringList()` — возвращает список строк, хранимых в модели;
- `insertRows(<Индекс>, <Количество>[, parent])` — вставляет указанное количество элементов в позицию, заданную первым параметром. Остальные элементы сдвигаются в конец списка. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeRows(<Индекс>, <Количество>[, parent])` — удаляет указанное количество элементов, начиная с позиции, заданной первым параметром. Метод возвращает значение `True`, если операция успешно выполнена;
- `setData(<QModelIndex>, <Значение>[, role=EditRole])` — задает значение для роли `role` элемента, на который указывает индекс `<QModelIndex>`. Метод возвращает значение `True`, если операция успешно выполнена;

- `data(<QModelIndex>, <Роль>)` — возвращает данные, хранимые в указанной роли элемента, на который ссылается индекс `<QModelIndex>`;
- `rowCount([parent])` — возвращает количество строк в модели;
- `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном.

Класс `QStringListModel` наследует метод `index()` из класса `QAbstractListModel`, который возвращает индекс (экземпляр класса `QModelIndex`) внутри модели. Формат метода:

```
index(<Строка>[, column=0][, parent = QModelIndex()])
```

6.4.3. Класс QStandardItemModel

Класс `QStandardItemModel` реализует двумерную (таблица) и иерархическую модели. Каждый элемент такой модели представлен классом `QStandardItem`. Модель можно отобразить с помощью классов `QTableView` и `QTreeView`, передав в метод `setModel()`. Иерархия наследования:

```
QObject — QAbstractItemModel — QStandardItemModel
```

Форматы конструктора класса `QStandardItemModel`:

```
<Объект> = QStandardItemModel([parent=<Родитель>])
<Объект> = QStandardItemModel(<Количество строк>, <Количество столбцов>
                              [, parent=<Родитель>])
```

Класс `QStandardItemModel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRowCount(<Количество строк>)` — задает количество строк;
- `setColumnCount(<Количество столбцов>)` — задает количество столбцов;
- `rowCount([parent=QModelIndex()])` — возвращает количество строк;
- `columnCount([parent=QModelIndex()])` — возвращает количество столбцов;
- `setItem(<Строка>, <Столбец>, <QStandardItem>)` — устанавливает элемент в указанную ячейку. Пример заполнения таблицы:

```
model = QtGui.QStandardItemModel(3, 4)
for row in range(0, 3):
    for column in range(0, 4):
        item = QtGui.QStandardItem(
```

```
                                "({0}, {1})".format(row, column))
        model.setItem(row, column, item)
view.setModel(model)
```

- `appendRow(<Список>)` — добавляет одну строку в конец модели. В качестве параметра указывается список экземпляров класса `QStandardItem`;
- `appendColumn(<Список>)` — добавляет один столбец в конец модели. В качестве параметра указывается список экземпляров класса `QStandardItem`;
- `insertRow(<Индекс строки>, <Список>)` — добавляет одну строку в указанную позицию модели. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`;
- `insertRow(<Индекс>[, parent=QModelIndex()])` — добавляет одну строку в указанную позицию модели. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько строк в указанную позицию модели. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertColumn(<Индекс столбца>, <Список>)` — добавляет один столбец в указанную позицию модели. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`;
- `insertColumn(<Индекс>[, parent=QModelIndex()])` — добавляет один столбец в указанную позицию. Метод возвращает значение `True`, если операция успешно выполнена;
- `insertColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — добавляет несколько столбцов в указанную позицию. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeRows(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество строк, начиная со строки, имеющей индекс `<Индекс>`. Метод возвращает значение `True`, если операция успешно выполнена;
- `removeColumns(<Индекс>, <Количество>[, parent=QModelIndex()])` — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс `<Индекс>`. Метод возвращает значение `True`, если операция успешно выполнена;
- `takeItem(<Строка>[, <Столбец>=0])` — удаляет указанный элемент из модели и возвращает его (экземпляр класса `QStandardItem`);
- `takeRow(<Индекс>)` — удаляет указанную строку из модели и возвращает ее (список экземпляров класса `QStandardItem`);

- `takeColumn(<Индекс>)` — удаляет указанный столбец из модели и возвращает его (список экземпляров класса `QStandardItem`);
- `clear()` — удаляет все элементы из модели;
- `item(<Строка>[, <Столбец>=0])` — возвращает ссылку на элемент (экземпляр класса `QStandardItem`), расположенный в указанной ячейке;
- `invisibleRootItem()` — возвращает ссылку на невидимый корневой элемент модели (экземпляр класса `QStandardItem`);
- `itemFromIndex(<QModelIndex>)` — возвращает ссылку на элемент (экземпляр класса `QStandardItem`), на который ссылается индекс `<QModelIndex>`;
- `index(<Строка>, <Столбец>[, parent=QModelIndex()])` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного в указанной ячейке;
- `indexFromItem(<QStandardItem>)` — возвращает индекс элемента (экземпляр класса `QModelIndex`), ссылка на который передана в качестве параметра;
- `setData(<QModelIndex>, <Значение>[, role=EditRole])` — задает значение для роли `role` элемента, на который указывает индекс `<QModelIndex>`. Метод возвращает значение `True`, если операция успешно выполнена;
- `data(<QModelIndex>[, role=DisplayRole])` — возвращает данные, хранимые в указанной роли элемента, на который ссылается индекс `<QModelIndex>`;
- `setHorizontalHeaderLabels(<Список строк>)` — задает заголовки столбцов. В качестве параметра указывается список строк;
- `setVerticalHeaderLabels(<Список строк>)` — задает заголовки строк. В качестве параметра указывается список строк;
- `setHorizontalHeaderItem(<Индекс>, <QStandardItem>)` — задает заголовок столбца. В первом параметре указывается индекс столбца, а во втором параметре экземпляр класса `QStandardItem`;
- `setVerticalHeaderItem(<Индекс>, <QStandardItem>)` — задает заголовок строки. В первом параметре указывается индекс строки, а во втором параметре экземпляр класса `QStandardItem`;
- `horizontalHeaderItem(<Индекс>)` — возвращает ссылку на указанный заголовок столбца (экземпляр класса `QStandardItem`);
- `verticalHeaderItem(<Индекс>)` — возвращает ссылку на указанный заголовок строки (экземпляр класса `QStandardItem`);
- `setHeaderData(<Индекс>, <Ориентация>, <Значение>[, role])` — задает данные для указанной роли заголовка. В первом параметре указывается индекс

строки или столбца, а во втором параметре — ориентация (атрибуты `Horizontal` или `Vertical` из класса `QtCore.Qt`). Если параметр `role` не указан, то используется значение `EditRole`. Метод возвращает значение `True`, если операция успешно выполнена;

- `headerData(<Индекс>, <Ориентация>[, role])` — возвращает данные, хранящиеся в указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором параметре — ориентация. Если параметр `role` не указан, то используется значение `DisplayRole`;
- `findItems(<Текст>[, flags=MatchExactly][, column=0])` — производит поиск элемента внутри модели в указанном в параметре `column` столбце. Допустимые значения параметра `flags` мы уже рассматривали в *разд. 6.1.3*. В качестве значения метод возвращает список элементов (список экземпляров класса `QStandardItem`) или пустой список;
- `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном;
- `setSortRole(<Роль>)` — задает роль (см. *разд. 6.3*), по которой производится сортировка;
- `parent(<QModelIndex>)` — возвращает индекс (экземпляр класса `QModelIndex`) родительского элемента. В качестве параметра указывается индекс (экземпляр класса `QModelIndex`) элемента-потомка;
- `hasChildren([parent=QModelIndex()])` — возвращает значение `True`, если существует элемент, расположенный на один уровень ниже по иерархии, и `False` — в противном случае.

При изменении значения элемента генерируется сигнал `itemChanged(QStandardItem *)`. Внутри обработчика через параметр доступна ссылка на элемент (экземпляр класса `QStandardItem`).

6.4.4. Класс QStandardItem

Каждый элемент модели `QStandardItemModel` представлен классом `QStandardItem`. Этот класс не только описывает элемент, но и позволяет создавать вложенные структуры. Форматы конструктора класса:

```
<Объект> = QStandardItem()
<Объект> = QStandardItem(<Текст>)
<Объект> = QStandardItem(<QIcon>, <Текст>)
<Объект> = QStandardItem(<Количество строк>[, <Количество столбцов>=1])
```

`<Объект> = QStandardItem(<QStandardItem>)`

Класс `QStandardItem` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRowCount(<Количество строк>)` — задает количество дочерних строк;
- `setColumnCount(<Количество столбцов>)` — задает количество дочерних столбцов;
- `rowCount()` — возвращает количество дочерних строк;
- `columnCount()` — возвращает количество дочерних столбцов;
- `row()` — возвращает индекс строки в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя;
- `column()` — возвращает индекс столбца в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя;
- `setChild(<Строка>, <Столбец>, <QStandardItem>)` — устанавливает элемент в указанную ячейку дочерней таблицы. Пример создания иерархии:

```
parent = QtGui.QStandardItem(3, 4)
parent.setText("Элемент-родитель")
for row in range(0, 3):
    for column in range(0, 4):
        item = QtGui.QStandardItem(
                    "({0}, {1})".format(row, column))
        parent.setChild(row, column, item)
model.appendRow(parent)
```

- `appendRow(<Список>)` — добавляет одну строку в конец дочерней таблицы. В качестве параметра указывается список экземпляров класса `QStandardItem`;
- `appendRow(<QStandardItem>)` — добавляет один элемент в конец дочерней таблицы. В качестве параметра указывается экземпляр класса `QStandardItem`;
- `appendRows(<Список>)` — добавляет несколько строк в конец дочерней таблицы. В качестве параметра указывается список экземпляров класса `QStandardItem`;
- `appendColumn(<Список>)` — добавляет один столбец в конец дочерней таблицы. В качестве параметра указывается список экземпляров класса `QStandardItem`;
- `insertRow(<Индекс строки>, <Список>)` — добавляет одну строку в указанную позицию дочерней таблицы. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`;

- `insertRow(<Индекс строки>, <QStandardItem>)` — добавляет один элемент в указанную позицию дочерней таблицы. В качестве второго параметра указывается экземпляр класса `QStandardItem`;
- `insertRows(<Индекс строки>, <Список>)` — добавляет несколько строк в указанную позицию дочерней таблицы. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`;
- `insertRows(<Индекс строки>, <Количество>)` — добавляет несколько строк в указанную позицию дочерней таблицы;
- `insertColumn(<Индекс столбца>, <Список>)` — добавляет один столбец в указанную позицию дочерней таблицы. В качестве параметра `<Список>` указывается список экземпляров класса `QStandardItem`;
- `insertColumns(<Индекс>, <Количество>)` — добавляет несколько столбцов в указанную позицию;
- `removeRow(<Индекс>)` — удаляет строку с указанным индексом;
- `removeRows(<Индекс>, <Количество>)` — удаляет указанное количество строк, начиная со строки, имеющей индекс `<Индекс>`;
- `removeColumn(<Индекс>)` — удаляет столбец с указанным индексом;
- `removeColumns(<Индекс>, <Количество>)` — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс `<Индекс>`;
- `takeChild(<Строка>[, <Столбец>= 0])` — удаляет указанный дочерний элемент и возвращает его (экземпляр класса `QStandardItem`);
- `takeRow(<Индекс>)` — удаляет указанную строку из дочерней таблицы и возвращает ее (список экземпляров класса `QStandardItem`);
- `takeColumn(<Индекс>)` — удаляет указанный столбец из дочерней таблицы и возвращает его (список экземпляров класса `QStandardItem`);
- `parent()` — возвращает ссылку на родительский элемент (экземпляр класса `QStandardItem`) или значение `None`;
- `child(<Строка>[, <Столбец>= 0])` — возвращает ссылку на дочерний элемент (экземпляр класса `QStandardItem`) или значение `None`;
- `hasChildren()` — возвращает значение `True`, если существует дочерний элемент, и `False` — в противном случае;
- `setData(<Значение>[, role=UserRole+1])` — устанавливает значение для указанной роли;
- `data([UserRole+1])` — возвращает значение, хранимое в указанной роли;

- `setText(<Текст>)` — задает текст элемента;
- `text()` — возвращает текст элемента;
- `setTextAlignment(<Выравнивание>)` — задает выравнивание текста внутри элемента;
- `setIcon(<QIcon>)` — задает иконку, которая будет отображена перед текстом;
- `setToolTip(<Текст>)` — задает текст всплывающей подсказки;
- `setWhatsThis(<Текст>)` — задает текст для справки;
- `setFont(<QFont>)` — задает шрифт элемента;
- `setBackground(<QBrush>)` — задает цвет фона;
- `setForeground(<QBrush>)` — задает цвет текста;
- `setCheckable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может взаимодействовать с флажком;
- `isCheckable()` — возвращает значение `True`, если пользователь может взаимодействовать с флажком, и `False` — в противном случае;
- `setCheckState(<Статус>)` — задает статус флажка. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:
  - `Unchecked` — `0` — флажок сброшен;
  - `PartiallyChecked` — `1` — флажок частично установлен;
  - `Checked` — `2` — флажок установлен;
- `checkState()` — возвращает текущий статус флажка;
- `setTristate(<Флаг>)` — если в качестве параметра указано значение `True`, то флажок может иметь три состояния: установлен, сброшен и частично установлен;
- `isTristate()` — возвращает значение `True`, если флажок может иметь три состояния, и `False` — в противном случае;
- `setFlags(<Флаги>)` — задает свойства элемента (см. *разд. 6.4.1*);
- `flags()` — возвращает значение установленных свойств элемента;
- `setSelectable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может выделить элемент;
- `setEditable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может редактировать текст элемента;
- `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то перетаскивание элемента разрешено;

- `setDropEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то сброс разрешен;
- `setEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может взаимодействовать с элементом. Значение `False` делает элемент недоступным;
- `clone()` — возвращает копию элемента (экземпляр класса `QStandardItem`);
- `index()` — возвращает индекс элемента (экземпляр класса `QModelIndex`);
- `model()` — возвращает ссылку на модель (экземпляр класса `QStandardItemModel`);
- `sortChildren(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку дочерней таблицы. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном.

6.5. Представления

Для отображения элементов модели предназначены следующие классы представлений:

- `ListView` — реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме того, с помощью этого класса можно отображать иконки;
- `QTableView` — реализует таблицу;
- `QTreeView` — реализует иерархический список.

Помимо этих классов для отображения элементов модели можно воспользоваться классами `QComboBox` (раскрывающийся список; см. *разд. 6.1*), `QListWidget` (простой список), `QTableWidget` (таблица) и `QTreeWidget` (иерархический список). Последние три класса нарушают концепцию "модель/представление", хотя и базируются на этой концепции. За подробной информацией по этим классам обращайтесь к документации.

6.5.1. Класс QAbstractItemView

Абстрактный класс `QAbstractItemView` является базовым классом для всех представлений. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                                QAbstractItemView
```

Класс `QAbstractItemView` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setCurrentIndex(<QModelIndex>)` — делает элемент с указанным индексом (экземпляр класса `QModelIndex`) текущим. Метод является слотом с сигнатурой `setCurrentIndex(const QModelIndex&);`
- `currentIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) текущего элемента;
- `setRootIndex(<QModelIndex>)` — задает корневой элемент. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом с сигнатурой `setRootIndex(const QModelIndex&);`
- `rootIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) корневого элемента;
- `setAlternatingRowColors(<Флаг>)` — если в качестве параметра указано значение `True`, то четные и нечетные строки будут иметь разный цвет фона;
- `setIndexWidget(<QModelIndex>, <QWidget>)` — устанавливает компонент в позицию, указанную индексом (экземпляр класса `QModelIndex`);
- `indexWidget(<QModelIndex>)` — возвращает ссылку на компонент, установленный ранее в позицию, указанную индексом (экземпляр класса `QModelIndex`);
- `setSelectionModel(<QItemSelectionModel>)` — устанавливает модель выделения;
- `selectionModel()` — возвращает модель выделения;
- `setSelectionMode(<Режим>)` — задает режим выделения элементов. В качестве параметра указываются следующие атрибуты из класса `QAbstractItemView`:
  - `NoSelection` — `0` — элементы не могут быть выделены;
  - `SingleSelection` — `1` — можно выделить только один элемент;
  - `MultiSelection` — `2` — можно выделить несколько элементов. Повторный щелчок на элементе снимает выделение;
  - `ExtendedSelection` — `3` — можно выделить несколько элементов, удерживая нажатой клавишу <Ctrl> или щелкнув мышью на элементе левой кнопкой мыши и перемещая мышь не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;
  - `ContiguousSelection` — `4` — можно выделить несколько элементов, щелкнув мышью на элементе левой кнопкой мыши и перемещая мышь не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;

➔ `setSelectionBehavior(<Режим>)` — задает режим выделения. В качестве параметра указываются следующие атрибуты из класса `QAbstractItemView`:

- `SelectItems` — `0` — выделяется отдельный элемент;
- `SelectRows` — `1` — выделяется строка целиком;
- `SelectColumns` — `2` — выделяется столбец целиком;

➔ `selectAll()` — выделяет все элементы. Метод является слотом;

➔ `clearSelection()` — снимает выделение. Метод является слотом;

➔ `setEditTriggers(<Режим>)` — задает действие, при котором производится начало редактирования текста элемента. В качестве параметра указывается комбинация следующих атрибутов из класса `QAbstractItemView`:

- `NoEditTriggers` — `0` — редактировать нельзя;
- `CurrentChanged` — `1` — при выделении элемента;
- `DoubleClicked` — `2` — при двойном щелчке мышью;
- `SelectedClicked` — `4` — при одинарном щелчке мышью на выделенном элементе;
- `EditKeyPressed` — `8` — при нажатии клавиши <F2>;
- `AnyKeyPressed` — `16` — при нажатии любой символьной клавиши;
- `AllEditTriggers` — `31` — при любом вышеперечисленном действии;

➔ `setIconSize(<QSize>)` — задает размер иконок. В качестве параметра указывается экземпляр класса `QSize`;

➔ `setTextElideMode(<Режим>)` — задает режим обрезки текста, если он не помещается в отведенную область. В месте пропуска выводится троеточие. Могут быть указаны следующие атрибуты из класса `QtCore.Qt`:

- `ElideLeft` — `0` — текст обрезается слева;
- `ElideRight` — `1` — текст обрезается справа;
- `ElideMiddle` — `2` — текст обрезается по середине;
- `ElideNone` — `3` — текст не обрезается;

➔ `setTabKeyNavigation(<Флаг>)` — если в качестве параметра указано значение `True`, то между элементами можно перемещаться с помощью клавиши <Tab>;

➔ `scrollTo(<QModelIndex>[, hint=EnsureVisible])` — прокручивает представление таким образом, чтобы элемент, на который ссылается индекс (экземпляр класса `QModelIndex`) был видим. В параметре `hint` указываются следующие атрибуты из класса `QAbstractItemView`:

- `EnsureVisible` — `0` — элемент должен быть в области видимости;
- `PositionAtTop` — `1` — элемент отображается в верхней части;
- `PositionAtBottom` — `2` — элемент отображается в нижней части;
- `PositionAtCenter` — `3` — элемент отображается в центре;

➔ `scrollToTop()` — прокручивает представление в самое начало. Метод является слотом;

➔ `scrollToBottom()` — прокручивает представление в самый конец. Метод является слотом;

➔ `setDragEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то перетаскивание элементов разрешено;

➔ `setDragDropMode(<Режим>)` — задает режим технологии drag & drop. В качестве параметра указываются следующие атрибуты из класса `QAbstractItemView`:

- `NoDragDrop` — `0` — drag & drop не поддерживается;
- `DragOnly` — `1` — поддерживается только перетаскивание;
- `DropOnly` — `2` — поддерживается только сбрасывание;
- `DragDrop` — `3` — поддерживается перетаскивание и сбрасывание;
- `InternalMove` — `4` — перетаскивание и сбрасывание самого элемента, а не его копии;

➔ `setDropIndicatorShown(<Флаг>)` — если в качестве параметра указано значение `True`, то позиция возможного сброса элемента будет выделена;

➔ `setAutoScroll(<Флаг>)` — если в качестве параметра указано значение `True`, то при перетаскивании пункта будет производиться автоматическая прокрутка;

➔ `setAutoScrollMargin(<Отступ>)` — задает расстояние от края области при достижении которого будет производиться автоматическая прокрутка области.

Класс `QAbstractItemView` содержит следующие сигналы:

➔ `activated(const QModelIndex&)` — генерируется при активизации элемента, например, путем нажатия клавиши <Enter>. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

➔ `pressed(const QModelIndex&)` — генерируется при нажатии кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

➔ `clicked(const QModelIndex&)` — генерируется при нажатии и отпускании кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

- `doubleClicked(const QModelIndex&)` — генерируется при двойном щелчке мышью над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);
- `entered(const QModelIndex&)` — генерируется при вхождении указателя мыши в область элемента. Чтобы сигнал сработал необходимо включить обработку перемещения указателя с помощью метода `setMouseTracking()` из класса `QWidget`. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);
- `viewportEntered()` — генерируется при вхождении указателя мыши в область компонента. Чтобы сигнал сработал необходимо включить обработку перемещения указателя с помощью метода `setMouseTracking()` из класса `QWidget`.

6.5.2. Класс QListView. Простой список

Класс `QListView` реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме того, с помощью этого класса можно отображать иконки. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                    QAbstractItemView — QListView
```

Формат конструктора класса `QListView`:

```
<Объект> = QListView([parent=<Родитель>])
```

Класс `QListView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 6.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel(<Модель>)` — устанавливает модель;
- `model()` — возвращает ссылку на модель;
- `setModelColumn(<Индекс>)` — задает индекс отображаемого столбца в табличной модели;
- `setViewMode(<Режим>)` — задает режим отображения элементов. В качестве параметра указываются следующие атрибуты из класса `QListView`:
  - `ListMode` — `0` — элементы размещаются сверху вниз, а иконки имеют маленькие размеры;
  - `IconMode` — `1` — элементы размещаются слева направо, а иконки имеют большие размеры. Элементы можно свободно перемещать с помощью мыши;
- `setMovement(<Режим>)` — задает режим перемещения элементов. В качестве параметра указываются следующие атрибуты из класса `QListView`:

- `Static` — `0` — пользователь не может перемещать элементы;
- `Free` — `1` — свободное перемещение;
- `Snap` — `2` — перемещение по сетке (размеры задаются методом `setGridSize()`);

➔ `setGridSize(<QSize>)` — задает размеры вспомогательной сетки;

➔ `setResizeMode(<Режим>)` — задает режим положения элементов при изменении размера списка. В качестве параметра указываются следующие атрибуты из класса `QListView`:

- `Fixed` — `0` — элементы остаются в том же положении;
- `Adjust` — `1` — положение элементов изменяется при изменении размеров;

➔ `setFlow(<Режим>)` — задает порядок вывода элементов. В качестве параметра указываются следующие атрибуты из класса `QListView`:

- `LeftToRight` — `0` — слева направо;
- `TopToBottom` — `1` — сверху вниз;

➔ `setWrapping(<Флаг>)` — если в качестве параметра указано значение `False`, то перенос элементов на новую строку (если они не помещаются в ширину области) запрещен;

➔ `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, то текст элемента может быть перенесен на другую строку;

➔ `setLayoutMode(<Режим>)` — задает режим размещения элементов. В качестве параметра указываются следующие атрибуты из класса `QListView`:

- `SinglePass` — `0` — элементы размещаются все сразу. Если список слишком большой, то окно будет заблокировано, пока все элементы не будут отображены;
- `Batched` — `1` — элементы размещаются блоками. Размер блока задается с помощью метода `setBatchSize(<Количество>)`;

➔ `setUniformItemSizes(<Флаг>)` — если в качестве параметра указано значение `True`, то все элементы будут иметь одинаковый размер;

➔ `setSpacing(<Отступ>)` — задает отступ вокруг элемента;

➔ `setSelectionRectVisible(<Флаг>)` — если в качестве параметра указано значение `True`, то при выделении будет отображаться вспомогательная рамка, показывающая область выделения. Метод доступен только при использовании режима множественного выделения;

- `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то строка с индексом, указанном в первом параметре, будет скрыта. Значение `False` отображает строку;
- `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список.

Класс `QListView` содержит сигнал `indexesMoved(const QModelIndexList&)` который генерируется при перемещении элементов. Внутри обработчика через параметр доступен список экземпляров класса `QModelIndex`.

6.5.3. Класс QTableView. Таблица

Класс `QTableView` реализует таблицу. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                    QAbstractItemView — QTableView
```

Формат конструктора класса `QTableView`:

```
<Объект> = QTableView([parent=<Родитель>])
```

Класс `QTableView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 6.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel(<Модель>)` — устанавливает модель;
- `model()` — возвращает ссылку на модель;
- `horizontalHeader()` — возвращает ссылку на горизонтальный заголовок (экземпляр класса `QHeaderView`);
- `verticalHeader()` — возвращает ссылку на вертикальный заголовок (экземпляр класса `QHeaderView`). Например, вывести таблицу без заголовков можно следующим образом:

  ```
  view.horizontalHeader().hide()
  view.verticalHeader().hide()
  ```

- `setRowHeight(<Индекс>, <Высота>)` — задает высоту строки с указанным в первом параметре индексом;
- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `rowHeight(<Индекс>)` — возвращает высоту строки;

- `columnWidth(<Индекс>)` — возвращает ширину столбца;
- `resizeRowToContents(<Индекс строки>)` — изменяет размер указанной строки таким образом, чтобы поместилось все содержимое. Метод является слотом с сигнатурой `resizeRowToContents(int)`;
- `resizeRowsToContents()` — изменяет размер всех строк таким образом, чтобы поместилось все содержимое. Метод является слотом;
- `resizeColumnToContents(<Индекс столбца>)` — изменяет размер указанного столбца таким образом, чтобы поместилось все содержимое. Метод является слотом с сигнатурой `resizeColumnToContents(int)`;
- `resizeColumnsToContents()` — изменяет размер всех столбцов таким образом, чтобы поместилось все содержимое. Метод является слотом;
- `setSpan(<Индекс строки>, <Индекс столбца>, <Количество строк>, <Количество столбцов>)` — растягивает элемент с указанными в первых двух параметрах индексами на заданное количество строк и столбцов. Происходит как бы объединение ячеек таблицы;
- `rowSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в строке, которое занимает элемент с указанными индексами;
- `columnSpan(<Индекс строки>, <Индекс столбца>)` — возвращает количество ячеек в столбце, которое занимает элемент с указанными индексами;
- `clearSpans()` — отменяет все объединения ячеек;
- `setRowHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то строка с индексом, указанном в первом параметре, будет скрыта. Значение `False` отображает строку;
- `hideRow(<Индекс>)` — скрывает строку с указанным индексом. Метод является слотом с сигнатурой `hideRow(int)`;
- `showRow(<Индекс>)` — отображает строку с указанным индексом. Метод является слотом с сигнатурой `showRow(int)`;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанном в первом параметре, будет скрыт. Значение `False` отображает столбец;
- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом с сигнатурой `hideColumn(int)`;
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом с сигнатурой `showColumn(int)`;

- `isRowHidden(<Индекс>)` — возвращает значение `True`, если строка с указанным индексом скрыта, и `False` — в противном случае;
- `isColumnHidden(<Индекс>)` — возвращает значение `True`, если столбец с указанным индексом скрыт, и `False` — в противном случае;
- `isIndexHidden(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) скрыт, и `False` — в противном случае;
- `selectRow(<Индекс>)` — выделяет строку с указанным индексом. Метод является слотом с сигнатурой `selectRow(int)`;
- `selectColumn(<Индекс>)` — выделяет столбец с указанным индексом. Метод является слотом с сигнатурой `selectColumn(int)`;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список;
- `setGridStyle(<Стиль>)` — задает стиль линий сетки. В качестве параметра указываются следующие атрибуты из класса `QtCore.Qt`:
  - `NoPen` — `0` — линии не выводятся;
  - `SolidLine` — `1` — сплошная линия;
  - `DashLine` — `2` — штриховая линия;
  - `DotLine` — `3` — пунктирная линия;
  - `DashDotLine` — `4` — штрих и точка, штрих и точка и т. д.;
  - `DashDotDotLine` — `5` — штрих и две точки, штрих и две точки и т. д.;
- `setShowGrid(<Флаг>)` — если в качестве параметра указано значение `True`, то сетка будет отображена, а если `False` — то скрыта. Метод является слотом с сигнатурой `setShowGrid(bool)`;
- `setSortingEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то столбцы можно сортировать с помощью щелчка мышью на заголовке столбца. При этом в заголовке показывается текущее направление сортировки;
- `setCornerButtonEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то с помощью кнопки в левом верхнем углу заголовка можно выделить всю таблицу. Значение `False` отключает кнопку;
- `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, то текст элемента может быть перенесен на другую строку;
- `sortByColumn(<Индекс столбца>[, AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса

`QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном.

6.5.4. Класс QTreeView. Иерархический список

Класс `QTreeView` реализует иерархический список. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                                    QAbstractItemView — QTreeView
```

Формат конструктора класса `QTreeView`:

```
<Объект> = QTreeView([parent=<Родитель>])
```

Класс `QTreeView` наследует все методы и сигналы из класса `QAbstractItemView` (см. *разд. 6.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel(<Модель>)` — устанавливает модель;
- `model()` — возвращает ссылку на модель;
- `header()` — возвращает ссылку на горизонтальный заголовок (экземпляр класса `QHeaderView`);
- `setColumnWidth(<Индекс>, <Ширина>)` — задает ширину столбца с указанным в первом параметре индексом;
- `columnWidth(<Индекс>)` — возвращает ширину столбца;
- `rowHeight(<QModelIndex>)` — возвращает высоту строки в которой находится элемент с указанным индексом (задается экземпляром класса `QModelIndex`);
- `resizeColumnToContents(<Индекс столбца>)` — изменяет ширину указанного столбца таким образом, чтобы поместилось все содержимое. Метод является слотом с сигнатурой `resizeColumnToContents(int)`;
- `setUniformRowHeights(<Флаг>)` — если в качестве параметра указано значение `True`, то все элементы будут иметь одинаковую высоту;
- `setHeaderHidden(<Флаг>)` — если в качестве параметра указано значение `True`, то заголовок будет скрыт. Значение `False` отображает заголовок;
- `isHeaderHidden()` — возвращает значение `True`, если заголовок скрыт, и `False` — в противном случае;
- `setColumnHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то столбец с индексом, указанном в первом параметре, будет скрыт. Значение `False` отображает столбец;

- `hideColumn(<Индекс>)` — скрывает столбец с указанным индексом. Метод является слотом с сигнатурой `hideColumn(int);`
- `showColumn(<Индекс>)` — отображает столбец с указанным индексом. Метод является слотом с сигнатурой `showColumn(int);`
- `isColumnHidden(<Индекс>)` — возвращает значение `True`, если столбец с указанным индексом скрыт, и `False` — в противном случае;
- `setRowHidden(<Индекс>, <QModelIndex>, <Флаг>)` — если в третьем параметре указано значение `True`, то строка с индексом `<Индекс>` и родителем `<QModelIndex>` будет скрыта. Значение `False` отображает строку;
- `isRowHidden(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с указанным индексом `<Индекс>` и родителем `<QModelIndex>` скрыта, и `False` — в противном случае;
- `isIndexHidden(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) скрыт, и `False` — в противном случае;
- `setExpanded(<QModelIndex>, <Флаг>)` — если во втором параметре указано значение `True`, то элементы, которые являются дочерними для элемента с указанным в первом параметре индексом, будут отображены, а если `False` — то скрыты. В первом параметре указывается экземпляр класса `QModelIndex`;
- `expand(<QModelIndex>)` — отображает элементы, которые являются дочерними для элемента с указанным индексом. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом с сигнатурой `expand(const QModelIndex&);`
- `expandToDepth(<Уровень>)` — отображает все дочерние элементы до указанного уровня. Метод является слотом с сигнатурой `expandToDepth(int);`
- `expandAll()` — отображает все дочерние элементы. Метод является слотом;
- `collapse(<QModelIndex>)` — скрывает элементы, которые являются дочерними для элемента с указанным индексом. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом с сигнатурой `collapse(const QModelIndex&);`
- `collapseAll()` — скрывает все дочерние элементы. Метод является слотом;
- `isExpanded(<QModelIndex>)` — возвращает значение `True`, если элементы, которые являются дочерними для элемента с указанным индексом, отображены, и `False` — в противном случае. В качестве параметра указывается экземпляр класса `QModelIndex`;

- `setItemsExpandable(<Флаг>)` — если в качестве параметра указано значение `False`, то пользователь не сможет отображать или скрывать дочерние элементы;
- `setAnimated(<Флаг>)` — если в качестве параметра указано значение `True`, то отображение и сокрытие дочерних элементов будет производиться с анимацией;
- `setIndentation(<Оступ>)` — задает отступ для дочерних элементов;
- `setRootIsDecorated(<Флаг>)` — если в качестве параметра указано значение `False`, то для элементов верхнего уровня не будут показываться компоненты, с помощью которых производится отображение и сокрытие дочерних элементов;
- `setSortingEnabled(<Флаг>)` — если в качестве параметра указано значение `True`, то столбцы можно сортировать с помощью щелчка мышью на заголовке столбца. При этом в заголовке показывается текущее направление сортировки;
- `sortByColumn(<Индекс столбца>[, AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном;
- `setWordWrap(<Флаг>)` — если в качестве параметра указано значение `True`, то текст элемента может быть перенесен на другую строку;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список.

Класс `QTreeView` содержит следующие сигналы:

- `expanded(const QModelIndex&)` — генерируется при отображении дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса `QModelIndex`) элемента;
- `collapsed(const QModelIndex&)` — генерируется при сокрытии дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса `QModelIndex`) элемента.

6.5.5. Класс QHeaderView. Заголовки строк и столбцов

Класс `QHeaderView` реализует заголовки строк и столбцов в представлениях `QTableView` и `QTreeView`. Получить ссылки на заголовки в классе `QTableView` позволяют методы `horizontalHeader()` и `verticalHeader()`, а для установки заголовков предназначены методы `setHorizontalHeader(<QHeaderView>)` и `setVerticalHeader(<QHeaderView>)`. Получить ссылку на заголовок в классе `QTreeView` позволяет метод `header()`, а для установки заголовка предназначен метод `setHeader(<QHeaderView>)`. Иерархия наследования:

(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea — QAbstractItemView — QHeaderView

Формат конструктора класса QHeaderView:

```
<Объект> = QHeaderView(<Ориентация>[, parent=<Родитель>])
```

Параметр <Ориентация> позволяет задать ориентацию заголовка. В качестве значения указываются атрибуты Horizontal или Vertical из класса QtCore.Qt.

Класс QHeaderView наследует все методы и сигналы из класса QAbstractItemView (см. *разд. 6.5.1*) и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- count() — возвращает количество секций в заголовке. Получить количество секций можно также с помощью функции len();
- setDefaultSectionSize(<Размер>) — задает размер секции по умолчанию;
- defaultSectionSize() — возвращает размер секций по умолчанию;
- setMinimumSectionSize(<Размер>) — задает минимальный размер секций;
- minimumSectionSize() — возвращает минимальный размер секций;
- resizeSection(<Индекс>, <Размер>) — изменяет размер секции с указанным индексом;
- sectionSize(<Индекс>) — возвращает размер секции с указанным индексом;
- setResizeMode(<Режим>) — задает режим изменения размеров для всех секций. В качестве параметра могут быть указаны следующие атрибуты из класса QHeaderView:
  - Interactive — 0 — размер может быть изменен пользователем или программно;
  - Stretch — 1 — секции автоматически равномерно распределяют свободное пространство между собой. Размер не может быть изменен ни пользователем, ни программно;
  - Fixed — 2 — размер может быть изменен только программно;
  - ResizeToContents — 3 — размер определяется автоматически по содержимому секции. Размер не может быть изменен ни пользователем, ни программно;
- setResizeMode(<Индекс>, <Режим>) — задает режим изменения размеров для секции с указанным индексом;
- setStretchLastSection(<Флаг>) — если в качестве параметра указано значение True, то последняя секция будет занимать все свободное пространство;

- `setCascadingSectionResizes(<Флаг>)` — если в качестве параметра указано значение `True`, то изменение размеров одной секции может привести к изменению размеров других секций;
- `setSectionHidden(<Индекс>, <Флаг>)` — если во втором параметре указано значение `True`, то секция с индексом, указанном в первом параметре, будет скрыта. Значение `False` отображает секцию;
- `hideSection(<Индекс>)` — скрывает секцию с указанным индексом;
- `showSection(<Индекс>)` — отображает секцию с указанным индексом;
- `isSectionHidden(<Индекс>)` — возвращает значение `True`, если секция с указанным индексом скрыта, и `False` — в противном случае;
- `sectionsHidden()` — возвращает значение `True`, если существует скрытая секция, и `False` — в противном случае;
- `hiddenSectionCount()` — возвращает количество скрытых секций;
- `setDefaultAlignment(<Выравнивание>)` — задает выравнивание текста внутри заголовков;
- `setHighlightSections(<Флаг>)` — если в качестве параметра указано значение `True`, то текст заголовка текущей секции будет выделен;
- `setClickable(<Флаг>)` — если в качестве параметра указано значение `True`, то заголовок будет реагировать на щелчок мышью, при этом выделяя все элементы секции;
- `setMovable(<Флаг>)` — если в качестве параметра указано значение `True`, то пользователь может перемещать секции с помощью мыши;
- `isMovable()` — возвращает значение `True`, если пользователь может перемещать секции с помощью мыши, и `False` — в противном случае;
- `moveSection(<Откуда>, <Куда>)` — позволяет переместить секцию. В параметрах указываются визуальные индексы;
- `swapSections(<Секция1>, <Секция2>)` — меняет две секции местами. В параметрах указываются визуальные индексы;
- `visualIndex(<logicalIndex>)` — преобразует логический (первоначальный порядок следования) индекс в визуальный (отображаемый порядок следования) индекс. Если преобразование прошло неудачно, то возвращается значение `-1`;
- `logicalIndex(<visualIndex>)` — преобразует визуальный (отображаемый порядок следования) индекс в логический (первоначальный порядок следования) индекс. Если преобразование прошло неудачно, то возвращается значение `-1`;

- `saveState()` — возвращает экземпляр класса `QByteArray` с текущими размерами и положением секций;
- `restoreState(<QByteArray>)` — восстанавливает размеры и положение секций на основе экземпляра класса `QByteArray`, возвращаемого методом `saveState()`.

Класс `QHeaderView` содержит следующие сигналы (перечислены только основные сигналы; полный список смотрите в документации):

- `sectionPressed(int)` — генерируется при нажатии левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionClicked(int)` — генерируется при нажатии и отпускании левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionDoubleClicked(int)` — генерируется при двойном щелчке мышью на заголовке секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionMoved(int, int, int)` — генерируется при изменении положения секции. Внутри обработчика через первый параметр доступен логический индекс секции, через второй параметр — старый визуальный индекс, а через третий — новый визуальный индекс;
- `sectionResized(int, int, int)` — генерируется непрерывно при изменении размера секции. Внутри обработчика через первый параметр доступен логический индекс секции, через второй параметр — старый размер, а через третий — новый размер.

6.6. Управление выделением элементов

Класс `QItemSelectionModel` реализует модель, позволяющую централизованно управлять выделением сразу в нескольких представлениях. Установить модель выделения позволяет метод `setSelectionModel(<QItemSelectionModel>)` из класса `QAbstractItemView`, а получить ссылку на модель можно с помощью метода `selectionModel()`. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении. Иерархия наследования выглядит так:

```
QObject — QItemSelectionModel
```

Форматы конструктора класса `QItemSelectionModel`:

```
<Объект> = QItemSelectionModel(<Модель>)
```

```
<Объект> = QItemSelectionModel(<Модель>, <Родитель>)
```

Класс `QItemSelectionModel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `hasSelection()` — возвращает значение `True`, если существует выделенный элемент, и `False` — в противном случае;
- `isSelected(<QModelIndex>)` — возвращает значение `True`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) выделен, и `False` — в противном случае;
- `isRowSelected(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с индексом `<Индекс>` и родителем `<QModelIndex>` выделена, и `False` — в противном случае;
- `isColumnSelected(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если столбец с индексом `<Индекс>` и родителем `<QModelIndex>` выделен, и `False` — в противном случае;
- `rowIntersectsSelection(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если строка с индексом `<Индекс>` и родителем `<QModelIndex>` содержит выделенный элемент, и `False` — в противном случае;
- `columnIntersectsSelection(<Индекс>, <QModelIndex>)` — возвращает значение `True`, если столбец с индексом `<Индекс>` и родителем `<QModelIndex>` содержит выделенный элемент, и `False` — в противном случае;
- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список;
- `selectedRows([<Индекс столбца>=0])` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов из указанного столбца. Элемент попадет в список только в том случае, если строка выделена полностью;
- `selectedColumns([<Индекс строки>=0])` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов из указанной строки. Элемент попадет в список только в том случае, если столбец выделен полностью;
- `selection()` — возвращает ссылку на экземпляр класса `QItemSelection`;
- `select(<QModelIndex>, <Режим>)` — изменяет выделение элемента с указанным индексом. Во втором параметре указываются следующие атрибуты (или их комбинация через оператор `|`) из класса `QItemSelectionModel`:
  - `NoUpdate` — без изменений;
  - `Clear` — снимает выделение всех элементов;
  - `Select` — выделяет элемент;

- `Deselect` — снимает выделение с элемента;
- `Toggle` — выделяет элемент, если он не выделен, или снимает выделение, если элемент был выделен;
- `Current` — изменяет выделение текущего элемента;
- `Rows` — индекс будет расширен так, чтобы охватить всю строку;
- `Columns` — индекс будет расширен так, чтобы охватить весь столбец;
- `SelectCurrent` — комбинация `Select | Current`;
- `ToggleCurrent` — комбинация `Toggle | Current`;
- `ClearAndSelect` — комбинация `Clear | Select`.

Метод является слотом с сигнатурой `select(const QModelIndex&, QItemSelectionModel::SelectionFlags);`

➔ `select(<QItemSelection>, <Режим>)` — изменяет выделение элементов. Метод является слотом с сигнатурой `select(const QItemSelection&, QItemSelectionModel::SelectionFlags);`

➔ `setCurrentIndex(<QModelIndex>, <Режим>)` — делает элемент текущим и изменяет режим выделения. Метод является слотом с сигнатурой `setCurrentIndex(const QModelIndex&, QItemSelectionModel::SelectionFlags);`

➔ `currentIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) текущего элемента;

➔ `clearSelection()` — снимает все выделения. Метод является слотом.

Класс `QItemSelectionModel` содержит следующие сигналы:

➔ `currentChanged(const QModelIndex&, const QModelIndex&)` — генерируется при изменении индекса текущего элемента. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй параметр — индекс нового элемента;

➔ `currentRowChanged(const QModelIndex&, const QModelIndex&)` — генерируется, когда текущий элемент перемещается в другую строку. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй параметр — индекс нового элемента;

➔ `currentColumnChanged(const QModelIndex&, const QModelIndex&)` — генерируется, когда текущий элемент перемещается в другой столбец. Внутри обработчика через первый параметр доступен индекс предыдущего элемента, а через второй параметр — индекс нового элемента;

➔ `selectionChanged(const QItemSelection&, const QItemSelection&)` — генерируется при изменении выделения.

6.7. Промежуточные модели

Как вы уже знаете, одну модель можно установить в нескольких представлениях. При этом изменение порядка следования элементов в одном представлении повлечет за собой изменение порядка следования элементов в другом представлении. Чтобы предотвратить изменение порядка следования элементов в базовой модели следует создать промежуточную модель с помощью класса `QSortFilterProxyModel` и установить ее в представлении. Иерархия наследования для класса `QSortFilterProxyModel` выглядит так:

```
QObject — QAbstractItemModel — QAbstractProxyModel —
                                    QSortFilterProxyModel
```

Формат конструктора класса `QSortFilterProxyModel`:

```
<Объект> = QSortFilterProxyModel([parent=<Родитель>])
```

Класс `QSortFilterProxyModel` наследует следующие методы из класса `QAbstractProxyModel` (перечислены только основные методы; полный список смотрите в документации):

➔ `setSourceModel(<Модель>)` — устанавливает базовую модель;

➔ `sourceModel()` — возвращает ссылку на базовую модель.

Класс `QSortFilterProxyModel` поддерживает основные методы обычных моделей и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

➔ `sort(<Индекс столбца>[, order=AscendingOrder])` — производит сортировку. Если во втором параметре указан атрибут `AscendingOrder` из класса `QtCore.Qt`, то сортировка производится в прямом порядке, а если `DescendingOrder` — то в обратном. Если в параметре `<Индекс столбца>` указать значение `-1`, то будет использован порядок следования элементов из базовой модели.

> ***Примечание***
> Чтобы включить сортировку столбцов пользователем следует передать значение `True` в метод `setSortingEnabled()` объекта представления.

➔ `setSortRole(<Роль>)` — задает роль (см. *разд. 6.3*), по которой производится сортировка. По умолчанию сортировка производится по роли `DisplayRole`;

- `setSortCaseSensitivity(<Режим>)` — если в качестве параметра указать атрибут `CaseInsensitive` из класса `QtCore.Qt`, то при сортировке не будет учитываться регистр символов, а если `CaseSensitive` — то регистр будет учитываться;
- `setSortLocaleAware(<Флаг>)` — если в качестве параметра указать значение `True`, то при сортировке будут учитываться настройки локали;
- `setFilterFixedString(<Фрагмент>)` — в результат попадут только строки, которые содержат заданный фрагмент. Если указать пустую строку, то в результат попадут все строки из базовой модели. Метод является слотом с сигнатурой `setFilterFixedString(const QString&);`
- `setFilterRegExp()` — производит фильтрацию элементов в соответствии с указанным регулярным выражением. Если указать пустую строку, то в результат попадут все строки из базовой модели. Форматы метода:

  `setFilterRegExp(<QRegExp>)`

  `setFilterRegExp(<Строка с шаблоном>)`

  В первом формате указывается экземпляр класса `QRegExp`, а во втором формате строка с шаблоном регулярного выражения. Второй формат метода является слотом с сигнатурой `setFilterRegExp(const QString&);`
- `setFilterWildcard(<Шаблон>)` — производит фильтрацию элементов в соответствии с указанной строкой, содержащей подстановочные знаки:
  - `?` — один любой символ;
  - `*` — нуль или более любых символов;
  - `[...]` — диапазон значений.

  Остальные символы трактуются как есть. Если в качестве параметра указать пустую строку, то в результат попадут все строки из базовой модели. Метод является слотом с сигнатурой `setFilterWildcard(const QString&);`
- `setFilterKeyColumn(<Индекс>)` — задает индекс столбца по которому будет производиться фильтрация. Если в качестве параметра указать значение `-1`, то будут просматриваться элементы во всех столбцах. По умолчанию фильтрация производится по первому столбцу;
- `setFilterRole(<Роль>)` — задает роль (см. *разд. 6.3*), по которой производится фильтрация. По умолчанию сортировка производится по роли `DisplayRole`;
- `setFilterCaseSensitivity(<Режим>)` — если в качестве параметра указать атрибут `CaseInsensitive` из класса `QtCore.Qt`, то при фильтрации не будет учитываться регистр символов, а если `CaseSensitive` — то регистр будет учитываться;

➔ `setDynamicSortFilter(<Флаг>)` — если в качестве параметра указано значение `True`, то при изменении базовой модели будет производиться повторная сортировка или фильтрация.