

# Pyqt

## 1. 3 种主窗口

PyQt5是一个强大的跨平台GUI框架，它提供了多种不同类型的主窗口类，以满足不同的应用需求。下面是PyQt5中最常见的几种主窗口类型及其创建方式的简介：

### 1.1 QMainWindow

QMainWindow是用于创建具有菜单栏、工具栏、状态栏和中心窗口部件（如文本编辑器、画布或其他自定义布局）的应用程序的主窗口。

```
from PyQt5.QtWidgets import QApplication, QMainWindow

app = QApplication([])
window = QMainWindow()
window.setWindowTitle('QMainWindow Example')
window.show()
app.exec_()
```

### 1.2 QWidget

QWidget是所有用户界面对象的基类。当作为顶层窗口使用时，QWidget提供了一个简单的带有标题栏和边框的窗口。它可以用来创建不需要菜单栏、工具栏或状态栏的简单应用程序。

```
from PyQt5.QtWidgets import QApplication, QWidget

app = QApplication([])
window = QWidget()
window.setWindowTitle('QWidget Example')
window.show()
app.exec_()
```

### 1.3 QDialog

QDialog是用于创建对话框窗口的类。这些窗口通常用于短暂的、特定任务（如设置选项、文件选择等）。

对话框类型	QDialog 类
颜色对话框	QColorDialog
文件对话框	QFileDialog
字体对话框	QFontDialog
输入对话框	QInputDialog
消息对话框	QMessageBox
进度对话框	QProgressDialog
错误信息对话框	QErrorMessage
向导对话框	QWizardPage

```
from PyQt5.QtWidgets import QApplication, QDialog

app = QApplication([])
dialog = QDialog()
dialog.setWindowTitle('QDialog Example')
dialog.exec_()
```


高级特性

每种窗口类型都可以通过添加布局、控件、事件处理等来扩展其功能。例如，QMainWindow可以包含如下元素：

- 菜单栏（QMenuBar）
- 工具栏（QToolBar）
- 状态栏（QStatusBar）
- 中心部件（通常是一个或多个自定义的QWidget）
- 停靠窗口（QDockWidget）

自定义窗口

在PyQt5中，可以通过继承现有的窗口类（如QMainWindow、QWidget或QDialog）来创建自定义窗口，并重写特定方法或添加新的属性和功能，以满足特定的应用需求。

 **Tip**

选择正确的窗口类型对于创建有效和用户友好的应用程序至关重要。QMainWindow适用于大多数标准的桌面应用程序，而QDialog适合于模态对话框。如果你需要更多的自由度和定制化，可以选择QWidget作为你的起点

2. 2 种弹簧

在PyQt5中，“弹簧”（Spacer）指的是在布局中使用的空间填充物，主要用于在控件之间添加额外的空白区域或者推动控件到窗口的一边。在PyQt5中，这种弹簧效果通常是通过QSpacerItem来实现的，这些弹簧可以在水平或垂直布局中使用。

## 2.1 水平弹簧 (Horizontal Spacer)

水平弹簧用于在水平布局中添加空间或推动控件。例如，你可以用它来将一个按钮推到窗口的右边。

```
from PyQt5.QtWidgets import QApplication, QWidget, QHBoxLayout, QPushButton, QSpacerItem,
QSizePolicy

app = QApplication([])
window = QWidget()
layout = QHBoxLayout(window)

button = QPushButton("Button")

# 创建一个水平弹簧
spacer = QSpacerItem(40, 20, QSizePolicy.Expanding, QSizePolicy.Minimum)

layout.addWidget(button)
layout.addItem(spacer)

window.setLayout(layout)
window.show()
app.exec_()
```

在这个例子中，`QSpacerItem` 接受四个参数：宽度、高度、水平尺寸策略和垂直尺寸策略。`QSizePolicy.Expanding` 确保弹簧会尽可能地扩展。

## 2.2 垂直弹簧 (Vertical Spacer)

垂直弹簧用于在垂直布局中添加空间或推动控件。例如，你可以用它来将控件推到窗口的底部。

```
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QSpacerItem,
QSizePolicy

app = QApplication([])
window = QWidget()
layout = QVBoxLayout(window)

button = QPushButton("Button")

# 创建一个垂直弹簧
spacer = QSpacerItem(20, 40, QSizePolicy.Minimum, QSizePolicy.Expanding)

layout.addItem(spacer)
layout.addWidget(button)

window.setLayout(layout)
window.show()
app.exec_()
```

### Note

#### 组合使用

你可以在同一个布局中同时使用水平和垂直弹簧来控制控件的位置。这在创建复杂的用户界面时特别有用。

#### 注意事项

- 弹簧的尺寸策略（`QSizePolicy`）对于控制其行为非常重要。例如，`QSizePolicy.Expanding` 会使得弹簧尽可能地扩展，而 `QSizePolicy.Fixed` 则会保持其大小不变。
- 弹簧不是实际的控件，而是布局中的一种工具，用于控制空间和排列。

使用弹簧可以有效地控制布局中控件的位置和布局的整体外观，是创建专业和用户友好的GUI应用程序的关键技术之一。

### 3. 5 种布局

在PyQt5中，布局管理器是用来控制窗口中控件（widgets）的位置和大小的。使用布局管理器而不是手动定位控件可以让界面自动适应不同的窗口大小和显示设置。PyQt5提供了多种布局管理器来满足不同的布局需求。

#### 3.1 水平布局（`QHBoxLayout`）

水平布局，即`QHBoxLayout`，是PyQt5中的一种布局方式，用于按水平方向排列窗口部件（Widgets）。这种布局特别适合于需要并排显示一系列控件的情况，如工具栏按钮、水平排列的表单元素等。使用`QHBoxLayout`可以轻松实现元素的左对齐、右对齐或均匀分布，同时确保在窗口大小变化时各元素的相对位置保持一致。

##### 创建和使用步骤：

创建布局对象：首先实例化一个`QHBoxLayout`对象。

添加控件：通过`addWidget`方法将需要水平排列的控件添加到布局中。

设置布局：将这个布局对象应用到一个容器（如`QWidget`或`QMainWindow`）上。

调整布局属性：可以设置间距、对齐方式等属性以满足具体需求。

```
from PyQt5.QtWidgets import QApplication, QWidget, QHBoxLayout, QPushButton

app = QApplication([])
window = QWidget()
layout = QHBoxLayout()

# 添加控件
button1 = QPushButton('Button 1')
button2 = QPushButton('Button 2')
layout.addWidget(button1)
layout.addWidget(button2)

# 应用布局
window.setLayout(layout)
window.show()
app.exec_()
```

##### 管理控件：

- 添加控件：使用`addWidget`将控件添加到布局中。
- 移除控件：要移除控件，可以使用`removeWidget`方法，但需注意，这并不会删除控件本身，只是将其从布局中移除。
- 调整控件间距：使用`setSpacing`方法可以调整控件之间的间距。
- 设置伸缩因子：可以为各控件设置不同的伸缩因子（stretch factor），以控制它们在布局中的相对大小。0

### 3.2 垂直布局 (QVBoxLayout)

垂直布局，即QVBoxLayout，是PyQt5中用于按垂直方向排列控件的布局管理器。与水平布局相对，垂直布局将控件从上到下依次排列，适用于需要垂直堆叠控件的场景。这种布局方式在管理控件大小和位置方面极为有效，尤其是在处理具有不同大小的控件时，可以确保它们垂直对齐且在窗口大小改变时自动调整。

#### 创建和使用步骤：

1. **实例化QVBoxLayout：** 首先创建一个QVBoxLayout对象。
2. **添加控件：** 通过addWidget方法将控件添加到布局中。
3. **应用布局：** 将布局设置到一个容器（如QWidget）上。
4. **调整属性：** 可以根据需要设置控件间距、伸缩因子等属性。

```
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton

app = QApplication([])
window = QWidget()
layout = QVBoxLayout()

# 添加控件
button1 = QPushButton('Button 1')
button2 = QPushButton('Button 2')
layout.addWidget(button1)
layout.addWidget(button2)

# 应用布局
window.setLayout(layout)
window.show()
app.exec_()
```

### 3.3 网格布局 (QGridLayout)

网格布局，即QGridLayout，是PyQt5中提供的一种布局方式，允许开发者将控件放置在网格的行和列中。这种布局极大地增加了界面设计的灵活性，适合于需要精确控制控件位置和大小复杂界面设计。网格布局通过行和列的概念，允许控件跨越多个行或列，这在创建形式多样的用户界面时非常有用。

#### 创建和使用方法：

1. **实例化QGridLayout：** 首先创建一个QGridLayout对象。
2. **添加控件：** 使用addWidget方法，并指定控件的行、列位置，以及它所跨越的行数和列数。
3. **应用布局：** 将布局设置到容器（如QWidget）上。
4. **调整属性：** 可以设置行列间距、对齐方式等，以满足具体设计需求。

```
from PyQt5.QtWidgets import QApplication, QWidget, QGridLayout, QPushButton

app = QApplication([])
window = QWidget()
layout = QGridLayout()

# 添加控件
button1 = QPushButton('Button 1')
button2 = QPushButton('Button 2')
```

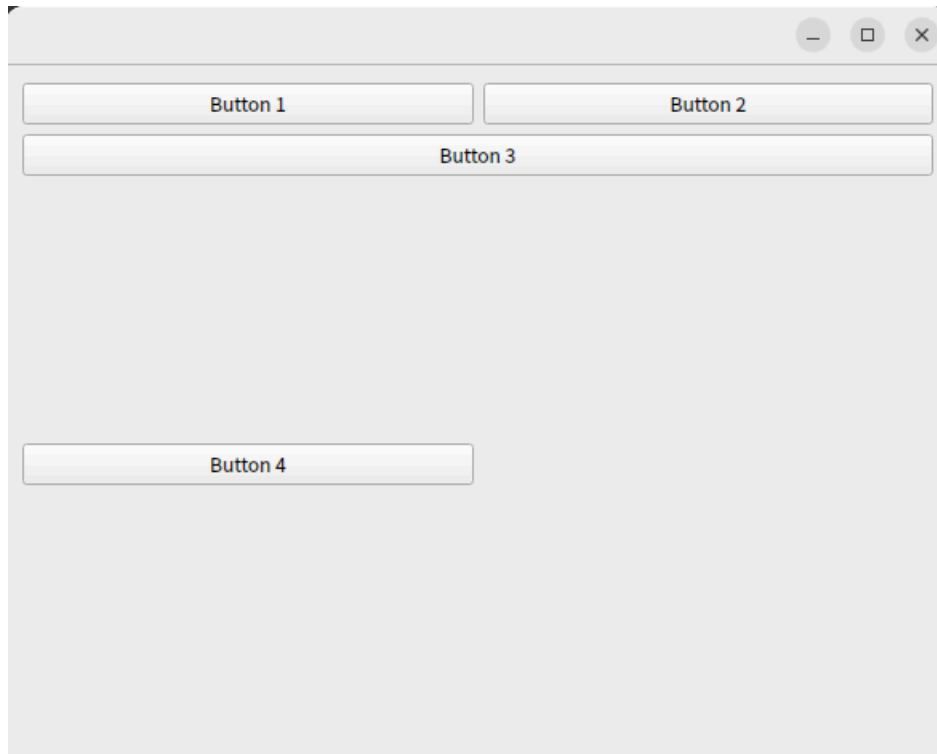
```

button3 = QPushButton('Button 3')
button4 = QPushButton('Button 4')

layout.addWidget(button1, 0, 0) # 第一行, 第一列
layout.addWidget(button2, 0, 1) # 第一行, 第二列
layout.addWidget(button3, 1, 0, 1, 2) # 第二行, 跨越两列
layout.addWidget(button4, 2, 0, 2, 1) # 跨越两行, 第一列

# 应用布局
window.setLayout(layout)
window.show()
app.exec_()

```



### 3.4 表单布局 (QFormLayout)

表单布局，即QFormLayout，是PyQt5提供了一种专门用于表单设计的布局方式。它以行为单位组织控件，每行通常包含一个标签（Label）和一个输入控件（如文本框、下拉列表等）。这种布局非常适合于需要快速创建清晰、结构化表单界面的应用场景，如用户注册、数据输入、设置配置等。

#### 创建和使用方法：

1. **实例化QFormLayout：** 首先创建一个QFormLayout对象。
2. **添加表单行：** 使用addRow方法添加表单的行，可以同时添加标签和对应的输入控件。
3. **应用布局：** 将此布局应用到容器（如QWidget）上。
4. **调整布局属性：** 可以设置行间距、对齐方式等，以满足具体的设计需求。

```

from PyQt5.QtWidgets import QApplication, QWidget, QFormLayout, QLineEdit, QLabel

app = QApplication([])
window = QWidget()
layout = QFormLayout()

# 添加表单行
layout.addRow(QLabel('Name:'), QLineEdit())

```

```
layout.addRow(QLabel('Email:'), QLineEdit())
layout.addRow(QLabel('Age:'), QLineEdit())

# 应用布局
window.setLayout(layout)
window.show()
app.exec_()
```

### 表单布局的作用：

用户输入：QFormLayout提供了一种高效的方法来收集用户输入，特别适用于需要用户填写多个字段的情况。

数据收集：在数据收集和处理的应用中，表单布局能够提供清晰的界面，帮助用户准确无误地输入数据。

布局一致性：通过标准化的行结构，表单布局确保整个表单的一致性和专业性，提升用户体验。

易于维护和扩展：在后续需要添加更多字段或调整表单结构时，QFormLayout使得维护和扩展变得更加简单。

总体来说，QFormLayout在PyQt5中为开发者提供了一种简洁、有效的方式来创建结构化和用户友好的表单界面，特别适合于需要大量用户输入和数据收集的应用。

### 3.5 堆叠布局 (QStackedLayout)

堆叠布局，即QStackedLayout，是PyQt5中的一种布局方式，它允许开发者在同一位置堆叠多个控件或布局，但一次只显示其中一个。这种布局特别适用于需要在同一界面区域中切换显示不同内容的场景，例如向导界面、多页表单、或切换不同功能模块的应用程序。

#### 创建和使用方法：

- 1.实例化QStackedLayout：创建一个QStackedLayout对象。
- 2.添加控件或布局：使用addWidget或addLayout方法添加多个控件或布局。
- 3.切换视图：使用setCurrentIndex方法来切换显示的控件或布局。
- 4.应用布局：将堆叠布局应用到一个容器（如QWidget）上。

```
from PyQt5.QtWidgets import QApplication, QWidget, QStackedLayout, QPushButton, QLabel
```

```
app = QApplication([])
window = QWidget()
stackedLayout = QStackedLayout()

# 创建几个要切换的控件
label1 = QLabel('View 1')
label2 = QLabel('View 2')
button = QPushButton('Switch View')

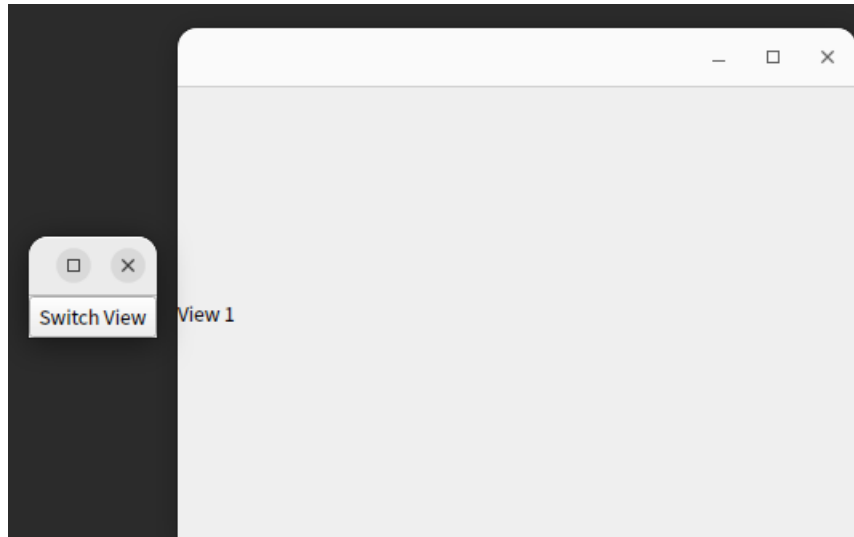
# 添加控件到堆叠布局
stackedLayout.addWidget(label1)
stackedLayout.addWidget(label2)

# 切换视图的函数
def switch_view():
    stackedLayout.setCurrentIndex((stackedLayout.currentIndex() + 1) % 2)

# 按钮点击切换视图
button.clicked.connect(switch_view)

# 设置堆叠布局
window.setLayout(stackedLayout)
window.show()

button.show()
app.exec_()
```







在此示例中，我们创建了两个标签控件和一个按钮，这些控件被添加到QStackedLayout中。通过点击按钮，可以在两个标签视图间切换。

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QWidget, QPushButton, QLabel,
QVBoxLayout, QStackedLayout

class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Stacked Layout Example")

        self.stacked_layout = QStackedLayout()
        page1 = self.create_page("Page 1 Content", "Next Page")
        page2 = self.create_page("Page 2 Content", "Next Page")
        page3 = self.create_page("Page 3 Content", "Next Page")
        self.stacked_layout.addWidget(page1)
        self.stacked_layout.addWidget(page2)
        self.stacked_layout.addWidget(page3)

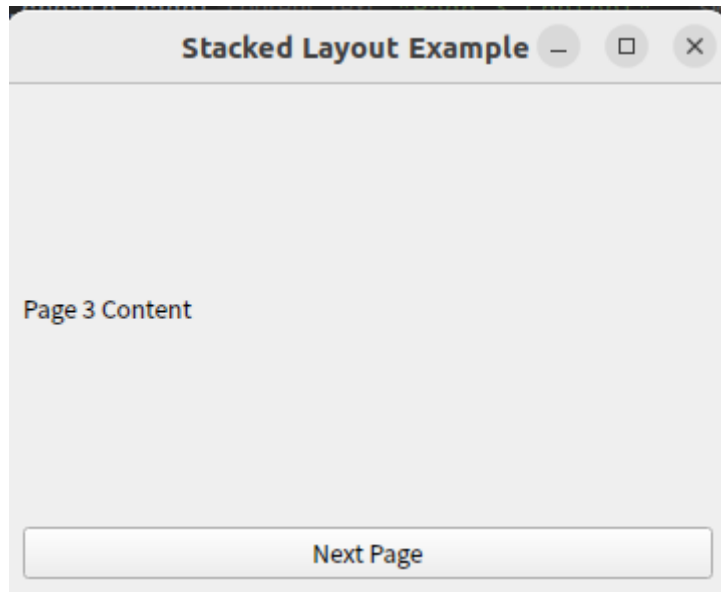
        central_widget = QWidget()
        central_widget.setLayout(self.stacked_layout)
        self.setCentralWidget(central_widget)

    def create_page(self, content_text, switch_button_text):
        layout = QVBoxLayout()
        content_label = QLabel(content_text)
        switch_button = QPushButton(switch_button_text)
        switch_button.clicked.connect(self.switch_page)
        layout.addWidget(content_label)
        layout.addWidget(switch_button)

        page = QWidget()
        page.setLayout(layout)
        return page

    def switch_page(self):
        # 切换页面
        current_index = self.stacked_layout.currentIndex()
        next_index = (current_index + 1) % 3 # 切换到下一页（循环切换）
        self.stacked_layout.setCurrentIndex(next_index)
```

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



```
import sys
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QIcon
from PyQt5.QtWidgets import (QApplication, QMainWindow, QStackedLayout, QWidget,
                             QToolBar, QToolButton, QStyle, QColorDialog, QFontDialog,
                             QVBoxLayout, QGroupBox, QRadioButton)

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

        # 设置窗口标题
        self.setWindowTitle('实战PyQt5: QStackedLayout Demo!')
        # 设置窗口大小
        self.resize(480, 360)

        self.initUi()

    def initUi(self):
        toolBar = QToolBar(self)
        self.addToolBar(Qt.LeftToolBarArea, toolBar)

        btnColor = self.createButton('颜色对话框')
        btnColor.clicked.connect(lambda: self.onButtonClicked(0))
        toolBar.addWidget(btnColor)
        btnFont = self.createButton('字体对话框')
        btnFont.clicked.connect(lambda: self.onButtonClicked(1))
        toolBar.addWidget(btnFont)
```

```

btnUser = self.createButton('分组部件')
btnUser.clicked.connect(lambda: self.onButtonClicked(2))
toolBar.addWidget(btnUser)

mainWidget = QWidget(self)

self.mainLayout = QStackedLayout(mainWidget)

# 添加三个widget,演示三个页面之间的切换

# 颜色对话框
self.mainLayout.addWidget(QColorDialog(self))
# 字体对话框
self.mainLayout.addWidget(QFontDialog(self))
# 自定义控件
self.mainLayout.addWidget(self.createExclusiveGroup())

mainWidget.setLayout(self.mainLayout)
# 设置中心窗口
self.setCentralWidget(mainWidget)

def createButton(self, text):
    icon = QApplication.style().standardIcon(QStyle.SP_DesktopIcon)
    btn = QToolButton(self)
    btn.setText(text)
    btn.setIcon(icon)
    btn.setToolButtonStyle(Qt.ToolButtonTextUnderIcon)

    return btn

def onButtonClicked(self, index):
    if index < self.mainLayout.count():
        self.mainLayout.setCurrentIndex(index)

def createExclusiveGroup(self):
    groupBox = QGroupBox('Exclusive Radio Buttons', self)

    radio1 = QRadioButton('&Radio Button 1', self)
    radio1.setChecked(True)
    radio2 = QRadioButton('R&adio button 2', self)
    radio3 = QRadioButton('Ra&dio button 3', self)

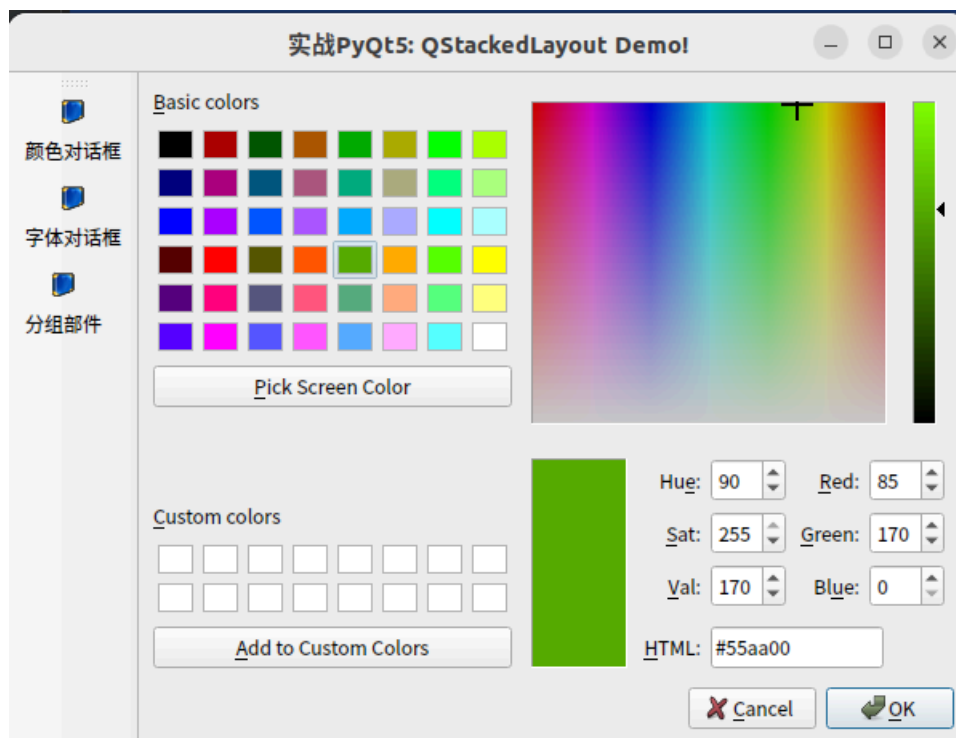
    vLayout = QVBoxLayout(groupBox)
    vLayout.addWidget(radio1)
    vLayout.addWidget(radio2)
    vLayout.addWidget(radio3)
    vLayout.addStretch(1)

    groupBox.setLayout(vLayout)

    return groupBox

if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = DemoStackedLayout()
    window.show()
    sys.exit(app.exec())

```



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

# Form implementation generated from reading ui file 'home.ui'
#
# Created by: PyQt5 UI code generator 5.15.6
#
# WARNING: Any manual changes made to this file will be lost when pyuic5 is
# run again. Do not edit this file unless you know what you are doing.

from PyQt5 import QtCore, QtGui, QtWidgets

class Ui_MainWindow(object):
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(733, 529)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.groupBox = QtWidgets.QGroupBox(self.centralwidget)
        self.groupBox.setGeometry(QtCore.QRect(0, 0, 761, 131))
        self.groupBox.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 1, y2: 0,
stop: 0 #011624, stop: 0.5 #004a72, stop: 1 #011624);")
        self.groupBox.setTitle("")
        self.groupBox.setObjectName("groupBox")
        self.horizontalLayoutWidget = QtWidgets.QWidget(self.groupBox)
        self.horizontalLayoutWidget.setGeometry(QtCore.QRect(-1, 50, 751, 80))
        self.horizontalLayoutWidget.setObjectName("horizontalLayoutWidget")
        self.horizontalLayout = QtWidgets.QHBoxLayout(self.horizontalLayoutWidget)
        self.horizontalLayout.setContentsMargins(0, 0, 0, 0)
        self.horizontalLayout.setObjectName("horizontalLayout")
        spacerItem = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem)
        spacerItem1 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem1)
```

```

        spacerItem2 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem2)
        spacerItem3 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem3)
        self.pushButton_3 = QtWidgets.QPushButton(self.horizontalLayoutWidget)
        font = QtGui.QFont()
        font.setPointSize(12)
        self.pushButton_3.setFont(font)
        self.pushButton_3.setStyleSheet("color: rgb(255, 255, 255);")
        self.pushButton_3.setObjectName("pushButton_3")
        self.horizontalLayout.addWidget(self.pushButton_3)
        spacerItem4 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem4)
        self.pushButton_2 = QtWidgets.QPushButton(self.horizontalLayoutWidget)
        font = QtGui.QFont()
        font.setPointSize(12)
        self.pushButton_2.setFont(font)
        self.pushButton_2.setStyleSheet("color: rgb(255, 255, 255);")
        self.pushButton_2.setObjectName("pushButton_2")
        self.horizontalLayout.addWidget(self.pushButton_2)
        spacerItem5 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem5)
        self.pushButton = QtWidgets.QPushButton(self.horizontalLayoutWidget)
        font = QtGui.QFont()
        font.setPointSize(12)
        self.pushButton.setFont(font)
        self.pushButton.setStyleSheet("color: rgb(255, 255, 255);")
        self.pushButton.setObjectName("pushButton")
        self.horizontalLayout.addWidget(self.pushButton)
        spacerItem6 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem6)
        spacerItem7 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem7)
        spacerItem8 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem8)
        spacerItem9 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem9)
        self.stackedWidget = QtWidgets.QStackedWidget(self.centralwidget)
        self.stackedWidget.setGeometry(QtCore.QRect(0, 130, 761, 401))
        self.stackedWidget.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 1, y2:
0, stop: 0 #011624, stop: 0.5 #004a72, stop: 1 #011624);")
        self.stackedWidget.setObjectName("stackedWidget")
        self.page = QtWidgets.QWidget()
        self.page.setObjectName("page")
        self.graphicsView = QtWidgets.QGraphicsView(self.page)
        self.graphicsView.setGeometry(QtCore.QRect(70, 0, 301, 192))
        self.graphicsView.setStyleSheet("background-color: rgb(85, 170, 0);")
        self.graphicsView.setObjectName("graphicsView")
        self.graphicsView_2 = QtWidgets.QGraphicsView(self.page)
        self.graphicsView_2.setGeometry(QtCore.QRect(390, 0, 301, 192))

```

```

self.graphicsView_2.setStyleSheet("background-color: rgb(170, 85, 255);")
self.graphicsView_2.setObjectName("graphicsView_2")
self.horizontalLayoutWidget_2 = QtWidgets.QWidget(self.page)
self.horizontalLayoutWidget_2.setGeometry(QtCore.QRect(69, 229, 621, 131))
self.horizontalLayoutWidget_2.setObjectName("horizontalLayoutWidget_2")
self.horizontalLayout_2 = QtWidgets.QHBoxLayout(self.horizontalLayoutWidget_2)
self.horizontalLayout_2.setContentsMargins(0, 0, 0, 0)
self.horizontalLayout_2.setObjectName("horizontalLayout_2")
self.graphicsView_6 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
self.graphicsView_6.setStyleSheet("background-color: rgb(255, 85, 0);")
self.graphicsView_6.setObjectName("graphicsView_6")
self.horizontalLayout_2.addWidget(self.graphicsView_6)
self.graphicsView_5 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
self.graphicsView_5.setStyleSheet("background-color: rgb(0, 85, 127);")
self.graphicsView_5.setObjectName("graphicsView_5")
self.horizontalLayout_2.addWidget(self.graphicsView_5)
self.graphicsView_4 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
self.graphicsView_4.setStyleSheet("background-color: rgb(170, 0, 127);")
self.graphicsView_4.setObjectName("graphicsView_4")
self.horizontalLayout_2.addWidget(self.graphicsView_4)
self.graphicsView_3 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
self.graphicsView_3.setStyleSheet("background-color: rgb(170, 85, 127);")
self.graphicsView_3.setObjectName("graphicsView_3")
self.horizontalLayout_2.addWidget(self.graphicsView_3)
self.stackedWidget.addWidget(self.page)
self.page_2 = QtWidgets.QWidget()
self.page_2.setObjectName("page_2")
self.checkBox_2 = QtWidgets.QCheckBox(self.page_2)
self.checkBox_2.setGeometry(QtCore.QRect(80, 100, 161, 75))
self.checkBox_2.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")
self.checkBox_2.setObjectName("checkBox_2")
self.checkBox = QtWidgets.QCheckBox(self.page_2)
self.checkBox.setGeometry(QtCore.QRect(80, 10, 161, 75))
self.checkBox.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")
self.checkBox.setObjectName("checkBox")
self.checkBox_3 = QtWidgets.QCheckBox(self.page_2)
self.checkBox_3.setGeometry(QtCore.QRect(80, 190, 161, 75))
self.checkBox_3.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")
self.checkBox_3.setObjectName("checkBox_3")
self.checkBox_4 = QtWidgets.QCheckBox(self.page_2)
self.checkBox_4.setGeometry(QtCore.QRect(80, 280, 161, 75))
self.checkBox_4.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")
self.checkBox_4.setObjectName("checkBox_4")
self.stackedWidget.addWidget(self.page_2)
self.page_3 = QtWidgets.QWidget()
self.page_3.setObjectName("page_3")
self.stackedWidget.addWidget(self.page_3)
MainWindow.setCentralWidget(self.centralwidget)

self.retranslateUi(MainWindow)
self.stackedWidget.setCurrentIndex(1)
QtCore.QMetaObject.connectSlotsByName(MainWindow)

```

```

def retranslateUi(self, MainWindow):

```

```

_translate = QtCore.QCoreApplication.translate
MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
self.pushButton_3.setText(_translate("MainWindow", "单源识别"))
self.pushButton_2.setText(_translate("MainWindow", "多源融合"))
self.pushButton.setText(_translate("MainWindow", "协同感知"))
self.checkBox_2.setText(_translate("MainWindow", "近红外"))
self.checkBox.setText(_translate("MainWindow", "可见光"))
self.checkBox_3.setText(_translate("MainWindow", "热红外"))
self.checkBox_4.setText(_translate("MainWindow", "高光谱"))

```

## 堆叠布局的应用：

向导界面：可以用于创建多步骤向导，用户可以在不同的步骤页面间切换。

多功能应用程序：在具有多个功能模块的应用程序中，可以用堆叠布局来切换不同的模块视图。

空间节约：当界面空间有限时，堆叠布局可以有效地在同一位置显示不同内容，节约空间。

QStackedLayout因其能够动态切换显示内容而成为PyQt5布局管理中的一个强大工具，尤其适用于需要高度动态界面的应用程序。

## 4. 4 种项目部件(Item Widget)

在PyQt5中，item widget是指可以放置在某些特定控件中的小部件，这些控件如QListWidget、QTableWidget和QTreeWidget支持直接在其单元格内放置widget。Item widget允许你在这些控件的每个项里放置更复杂的控件，例如按钮、复选框、下拉菜单等。

### 4.1 QListWidgetItem

QListWidgetItem 用于在 QListWidget 中表示每个列表项。它可以包含文本、图标等，并且可以设置为可选、可编辑等。

```

from PyQt5.QtWidgets import QApplication, QListWidget, QListWidgetItem

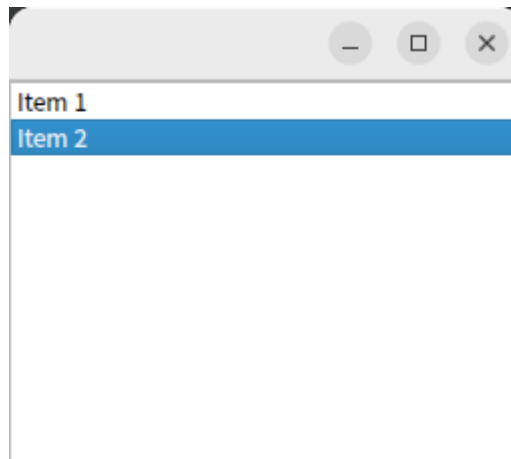
app = QApplication([])
list_widget = QListWidget()

item = QListWidgetItem("Item 1")
list_widget.addItem(item)

item = QListWidgetItem("Item 2")
list_widget.addItem(item)

list_widget.show()
app.exec_()

```



## 4.2 QTableWidgetItem

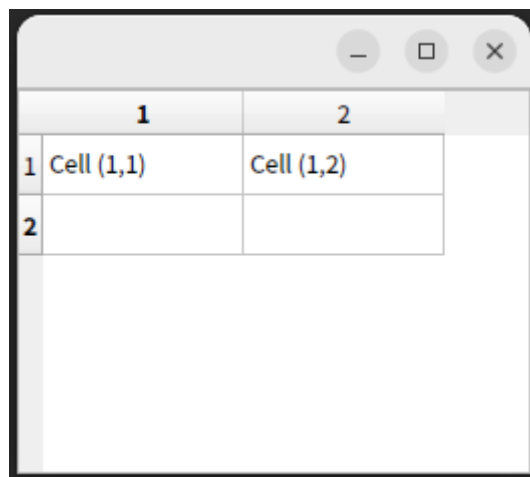
`QTableWidgetItem` 用于在 `QTableWidget` 中表示单元格。与 `QListWidgetItem` 类似，它可以包含文本和图标，并具有多种状态和属性。

```
from PyQt5.QtWidgets import QApplication, QTableWidgetItem

app = QApplication([])
table_widget = QTableWidgetItem(2, 2) # 2行2列

table_widget.setItem(0, 0, QTableWidgetItem("Cell (1,1)"))
table_widget.setItem(0, 1, QTableWidgetItem("Cell (1,2)"))

table_widget.show()
app.exec_()
```



## 4.3 QTreeWidgetItem

`QTreeWidgetItem` 用于在 `QTreeWidget` 中表示树形结构的每个节点。它可以有一个或多个子项，并且可以包含文本、图标等。

```
from PyQt5.QtWidgets import QApplication, QTreeWidgetItem

app = QApplication([])
tree_widget = QTreeWidgetItem()
tree_widget.setHeaderLabels(["Column 1", "Column 2", "Column 3"])

parent_item = QTreeWidgetItem(["Parent"])
tree_widget.addTopLevelItem(parent_item)
```

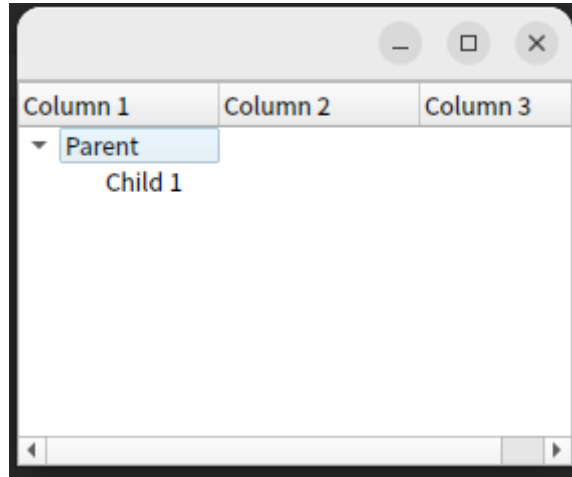


```

child_item = QTreeWidgetItem(["Child 1"])
parent_item.addChild(child_item)

tree_widget.show()
app.exec_()

```



#### 4.4 使用widgets作为Item

在 QTableWidget 或 QListWidget 中，你还可以直接将widget作为item插入。

```

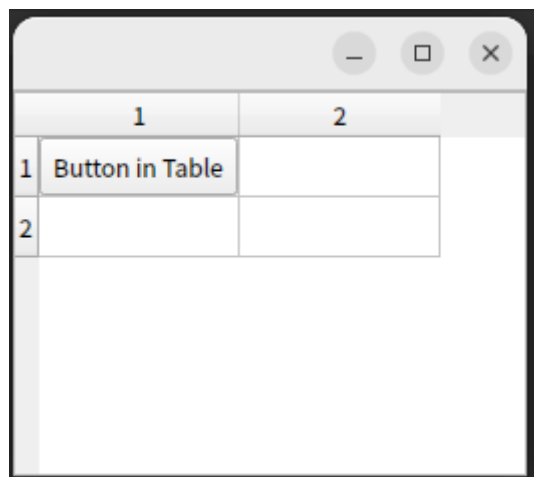
from PyQt5.QtWidgets import QApplication, QTableWidget, QPushButton

app = QApplication([])
table_widget = QTableWidget(2, 2)

# 在表格中插入按钮
button = QPushButton("Button in Table")
table_widget.setCellWidget(0, 0, button)

table_widget.show()
app.exec_()

```



使用item widget可以创建更动态和交互式的列表、表格和树形视图。

但是，大量使用widget作为item可能会影响性能，特别是在处理大型数据集时。

对于复杂的数据展示需求，考虑使用基于模型的视图（如 QListView, QTableView, QTreeView 以及它们的对应模型），这些通常提供更好的性能和灵活性。

5. 6 种按钮

在PyQt5中，按钮是构建用户界面的基本元素之一，用于执行命令、启动功能或触发事件。PyQt5提供了多种类型的按钮，每种都适用于不同的场景和需求。

5.1 QPushButton

PushButton是PyQt5中最常用的控件之一，允许用户通过单击来执行操作。PushButton既可以显示文本，也可以显示图像。当该控件被单击时，它看起来的状态像是被按下，然后被释放。

PushButton控件对应PyQt5中的QPushButton类，该类的常用方法及说明如下表：

方法	说明
setText()	设置按钮呈现的文本
text()	获取按钮呈现的文本
setIcon()	设置按钮上的图标，可以将参数设置为：QtGui.QIcon('图标路径')
setIconSize()	设置按钮图标的大小，参数可以设置为：QtCore.QSize(int width, int height)
setEnabled()	设置按钮是否可用，参数设置为False时，按钮不可用
setShortcut()	设置按钮的快捷键，参数可以设置为键盘中的按键或者组合键，例如<Alt + O>

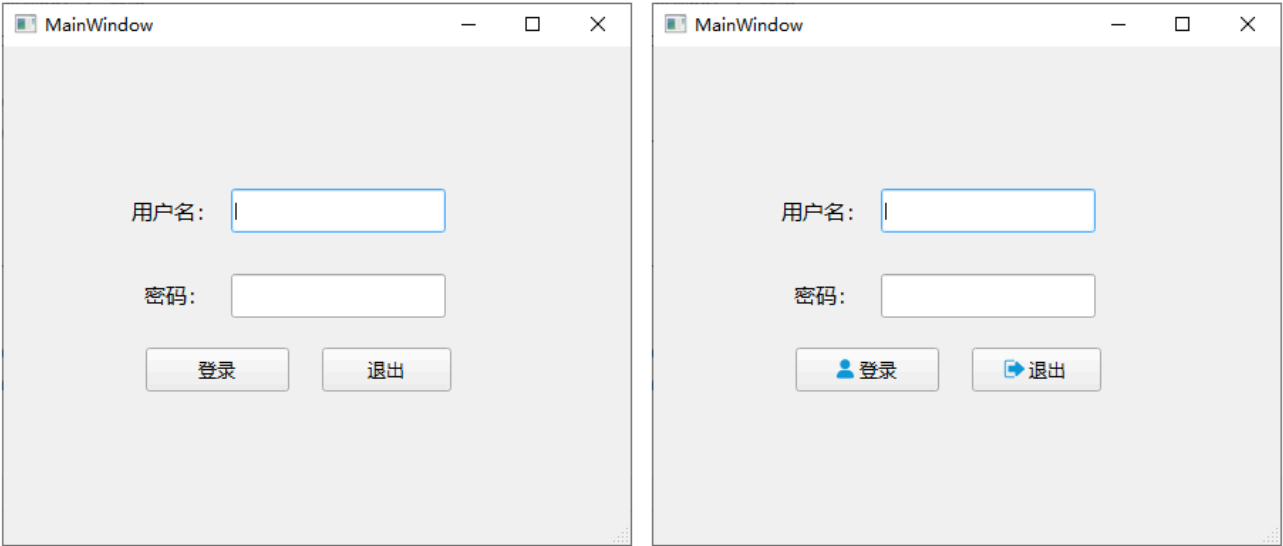
PushButton按钮中最常用的信号是clicked，作用是：当按钮被单击时，会发射该信号执行相应的操作。

实例：制作登录窗口

在Qt Designer中创建一个MainWindow窗口，添加“登录”和“退出”按钮。单击“登录”按钮时，弹出用户输入的用户名和密码。单击“退出”按钮时，则关闭当前登录窗口。

步骤1：从图标素材网站下载2个自己喜欢的图标，1个是登录，1个是退出，然后用代码设置到2个按钮中。当然，直接在按钮控件的属性编辑器中设置也是可以的。这里我们用代码进行演示：

```
self.pushButton.setIcon(QIcon(QPixmap('登录.png')))  
self.pushButton_2.setIcon(QIcon(QPixmap('退出登录.png')))
```



对比设置前后的效果，是不是高大上许多了呢？

步骤2：下面对密码输入框也就是lineEdit\_2进行一些限制设置，代码如下：

```
# 将输入框设置为密码模式
self.lineEdit_2.setEchoMode(QtWidgets.QLineEdit.Password)

# 设置输入框内只能输入8位数字
self.lineEdit_2.setValidator(QtGui.QIntValidator(10000000, 99999999))
```

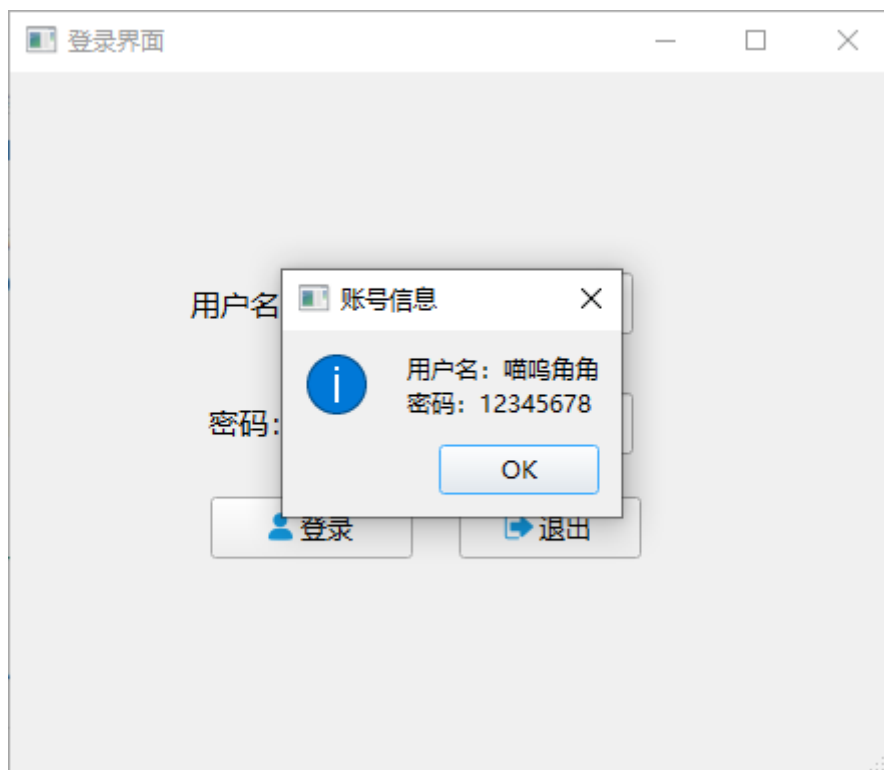
步骤3：自定义一个函数，使用information()方法弹出登录信息提示框，并将登录按钮和函数关联。代码如下：

```
# 将登录按钮和自定义函数login关联起来
self.pushButton.clicked.connect(self.login)

# 弹出登录信息提示框
def login(self):
    # 导入提示框需要用到的模块
    from PyQt5.QtWidgets import QMessageBox

    # 设置提示框内要显示的信息
    QMessageBox.information(self, '账号信息',
                            '用户名: ' + self.lineEdit.text() + '\n密码: ' +
self.lineEdit_2.text(),
                            QMessageBox.Ok)
```

运行上面代码，调出登录界面，输入信息后单击登录按钮，效果如下图：



我们也可以给登录按钮设置快捷键，方式1代码如下：

```
self.pushButton = QtWidgets.QPushButton('登录(&D)', self.centralwidget)
```

有Bug，而且是<Alt + D>组合键，很呆的方法，不推荐。

方法2：绑定小键盘的回车键，代码如下：

```
self.pushButton.setShortcut('enter')
self.pushButton.setShortcut('Enter')
```

代码简短，而且确实是一键登录，后面的参数enter / Enter皆可。弊端是只对数字小键盘里的回车键有效。

方法3：绑定键盘中间的回车键，代码如下：

```
# 参数return首字母大小写均可
self.pushButton.setShortcut('return')
self.pushButton.setShortcut('Return')
```

## 5.2 QRadioButton

QRadioButton是PyQt5中用于创建单选按钮的类，它继承自QAbstractButton类，间接继承自QWidget类。因此，QRadioButton可以像其他窗口部件一样进行布局和管理。作为常用的小部件之一，QRadioButton可用于在用户界面中提供单选选项。它可以和其他QRadioButton进行组合，形成一组互斥的选项，即同一时间只能选择其中一个。

## 5.3 QCheckBox

CheckBox复选框控件用来表示是否选取了某个选项条件，常用于为用户提供具有是或否、真或假的选项。它对应PyQt5中的QCheckBox类。

CheckBox控件的使用与RadioButton控件类似，但它是为用户提供“多选多”的选择。什么是“多选多”？就是CheckBox可以存在很多个，用户也可以同时选中很多个。而RadioButton控件是“多选一”，即：它可以存在很多个，但用户最终只能选择一个。

CheckBox控件有3种状态：选中、未选中、半选中。

### Note

除了常用的选中和未选中两种状态，CheckBox还提供了第三种状态来表明“没有变化”，也就是半选中状态。当需要为用户提供一个选中或者未选中复选框的选择时，这种状态还是很有用的哦~

如果你需要第三种状态，可以通过setTristate()来使它生效，并使用checkState()来查询当前的切换状态。

CheckBox最常用的信号是stateChanged()，即：状态改变时发射

QCheckBox类中的常用方法如下表：

方法	说明
setChecked()	设置复选框的状态，设置为True时表示选中复选框，设置为False时表示取消选中复选框
setText()	设置复选框的显示文本
text()	返回复选框的显示文本
isChecked()	检查复选框是否被选中
setTriState()	设置复选框为一个三态复选框

三态复选框有3种状态，如下表：

名称	值	说明
Qt.Checked	2	组件没有被选中（默认值）
Qt.PartiallyChecked	1	组件被半选中
Qt.Unchecked	0	组件被选中

### 实例讲解（1）

当CheckBox状态改变时，就把CheckBox的文本显示到Label中。这个代码怎么写？

```
# 当复选框为选中状态时，把文本显示在Label中
self.checkBox.stateChanged.connect(lambda: self.label.setText(self.checkBox.text()))
```

不用lambda的版本：

```
# 将CheckBox的状态改变和自定义函数get_value关联起来
self.checkBox.stateChanged.connect(self.get_value)

# 自定义一个函数，获取CheckBox的文本并设置到Label中
def get_value(self):
    self.label.setText(self.checkBox.text())
```

2种写法运行的结果都是一样的，如下图：



### 实例讲解（2）

当CheckBox是选中状态时就把它文本显示到Label，反之则不显示。就是你第一次点击，文本显示，第二次文本不显示，如此往复。

老实本分写法：

```
self.checkBox.stateChanged.connect(self.get_value)

def get_value(self):
    state = self.checkBox.isChecked()

    if state == True:
        self.label.setText(self.checkBox.text())

    if state == False:
        self.label.clear()
```

一本正经写法：

```
self.checkBox.stateChanged.connect(self.get_value)

def get_value(self):
    state = self.checkBox.isChecked()

    if state:
        self.label.setText(self.checkBox.text())

    if not state:
        self.label.clear()
```

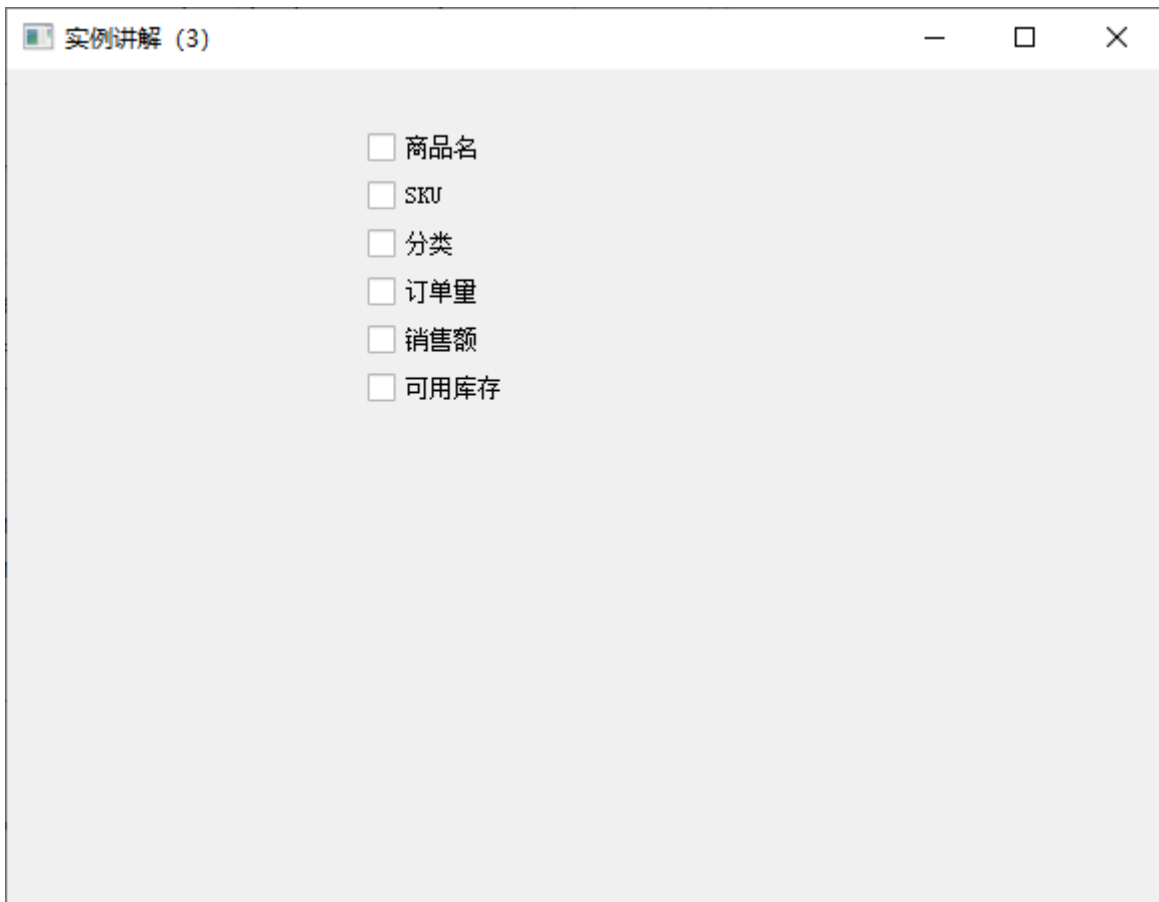
### 实例讲解 (3)

再举一个实际点的例子，假设现在有个列表`head_name = ['商品名', 'SKU', '分类', '订单量', '销售额', '可用库存']`，把里面的元素分别生成`CheckBox`，并添加到`verticalLayout`也就是垂直布局中，6个元素就是6个`CheckBox`

```
head_name = ['商品名', 'SKU', '分类', '订单量', '销售额', '可用库存']

# for循环遍历列表，用元素作为CheckBox的控件名
for name in head_name:
    ck = QCheckBox(name)
    self.verticalLayout.addWidget(ck)

# 在垂直布局中添加1个弹簧，避免生成的CheckBox太分散
self.www = QSpacerItem(0, 0, QSizePolicy.Minimum, QSizePolicy.Expanding)
self.verticalLayout.addItem(self.www)
```



#### 5.4 QPushButton

QPushButton 是一个通常用于工具栏的按钮，它可以显示一个图标。

##### Important

QPushButton和QPushButton的区别：

QPushButton 和 QPushButton 都是 PyQt 中的按钮控件，但它们有一些关键的区别：

1. \*\*外观和行为\*\*：

- QPushButton：是一个标准的按钮，可以显示文本或图片。当用户点击按钮时，它将显示被按下的状态，直到释放鼠标按钮。
- QPushButton：是一个更灵活的工具按钮，它可以显示文本、图片或两者。QPushButton 可以有多种状态，例如正常、悬停、按下等，并且可以显示菜单。

2. \*\*菜单功能\*\*：

- QPushButton：不支持直接显示菜单。
- QPushButton：可以设置一个菜单，当用户点击按钮时，可以弹出一个下拉菜单。这使得 QPushButton 非常适合需要在有限空间内提供多个选项的场景。

3. \*\*图标和文本\*\*：

- QPushButton：可以设置图标和文本，但不支持复杂的布局，如并排显示多个图标或文本。
- QPushButton：支持更复杂的布局，可以设置图标在文本的左侧或右侧，或者只有图标没有文本。

4. \*\*scale\*\*：

- QPushButton：通常在点击时不会改变大小。
- QPushButton：在某些情况下，如在 QPushButton 中，可以自动调整大小以适应可用空间。

5. \*\*使用场景\*\*：

- QPushButton：适用于简单的按钮需求，如对话框中的“确定”和“取消”按钮。
- QPushButton：适用于需要更多控制的场景，如工具栏中的按钮，可能需要显示菜单或图标和文本的复杂布局。

#### 6. \*\*样式表\*\*:

- 两者都支持样式表 (StyleSheet)，但 `QToolButton` 提供了更多的样式定制选项，以适应不同的状态和外观。

#### 7. \*\*继承关系\*\*:

- `QToolButton` 继承自 `QAbstractButton`，而 `QPushButton` 也继承自 `QAbstractButton`。这意味着它们共享一些基本的按钮功能，但在 `QToolButton` 中有更多的扩展。

在实际应用中，选择 `QPushButton` 或 `QToolButton` 取决于你的具体需求。如果你需要一个简单的按钮，`QPushButton` 可能是更好的选择。如果你需要一个可以显示菜单或具有更复杂布局的工具按钮，`QToolButton` 可能更适合。

## 5.5 QPushButton with Icon

QPushButton 也可以配置图标，使按钮更具视觉吸引力。

```
button = QPushButton()  
button.setIcon(QIcon('path/to/icon.png'))  
button.setText("Button with Icon")
```

## 5.6 QPushButton with Styles

可以自定义QPushButton的样式（例如颜色、字体、边框等）来更好地融入应用程序的整体风格。

```
button = QPushButton("Styled Button")  
button.setStyleSheet("background-color: blue; color: white; font: bold;")
```

### ⚠ Caution

- 按钮通常与信号和槽机制一起使用，以便在按钮被点击时执行特定的函数或方法。
- 不同类型的按钮适用于不同的使用场景，选择正确的类型可以提高应用程序的可用性。
- 通过样式表，你可以高度自定义按钮的外观，以适应应用程序的设计语言。

## 6. 10 种容器(Container)

### 6.1 QGroupBox - 分组界面元素

QGroupBox是PyQt5中的一个容器控件，它用于将一组相关的界面元素进行分组，从而在界面中创建有组织的部分。这个控件通常用于将一组复选框、单选按钮或其他控件组织在一起，以便用户可以更清晰地理解它们之间的关联性。

#### 主要用途:

- 组织和分组相关控件，以提高界面的可读性和可操作性。
- 允许用户在一组选项中进行选择。

#### QGroupBox的基本用法:

要创建一个QGroupBox，首先需要创建一个QGroupBox对象，然后将其他控件添加到该分组框内。以下是创建和配置QGroupBox的基本步骤:



```

# 导入必要的模块
from PyQt5.QtWidgets import QApplication, QMainWindow, QGroupBox, QVBoxLayout, QRadioButton

# 创建应用程序和主窗口
app = QApplication([])
window = QMainWindow()

# 创建一个QGroupBox
group_box = QGroupBox("选项")

# 创建一个垂直布局管理器
layout = QVBoxLayout()

# 向分组框中添加控件，例如单选按钮
radio_button1 = QRadioButton("选项1")
radio_button2 = QRadioButton("选项2")

layout.addWidget(radio_button1)
layout.addWidget(radio_button2)

# 将布局设置为分组框的布局
group_box.setLayout(layout)

# 将分组框添加到主窗口
window.setCentralWidget(group_box)

# 显示主窗口
window.show()
app.exec_()

```

在这个示例中，我们首先创建了一个QGroupBox，然后创建了一个垂直布局管理器，并将两个单选按钮添加到该布局中。最后，我们将布局设置为QGroupBox的布局，并将QGroupBox添加到主窗口中。

这样，我们就创建了一个简单的分组框，其中包含两个选项。用户可以通过单选按钮选择其中一个选项。

QGroupBox是一个强大的容器控件，可以帮助您组织和管理界面元素，提高用户体验。在实际应用中，您可以根据需要添加更多的控件和自定义分组框的样式。

## 6.2 QScrollArea - 添加滚动功能

QScrollArea是PyQt5中用于处理超出视图范围的内容的控件。它允许您将其他控件嵌套在其中，并在内容太大而无法完全显示时启用滚动条，以便用户可以滚动查看所有内容。

### 主要用途：

- 处理超出视图范围的内容，如大型文本、图像、表格等。
- 允许用户在内容过大时进行滚动查看。

### QScrollArea的基本用法：

要使用QScrollArea，首先创建一个QScrollArea对象，然后将要嵌套在其中的控件添加到QScrollArea中。以下是创建和配置QScrollArea的基本示例：

```

# 导入必要的模块
from PyQt5.QtWidgets import QApplication, QMainWindow, QScrollArea, QWidget, QVBoxLayout, QLabel

# 创建应用程序和主窗口

```

```
app = QApplication([])
window = QMainWindow()

# 创建一个QScrollArea
scroll_area = QScrollArea()

# 创建一个QWidget作为QScrollArea的子控件
scroll_content = QWidget()

# 创建一个垂直布局管理器
layout = QVBoxLayout()

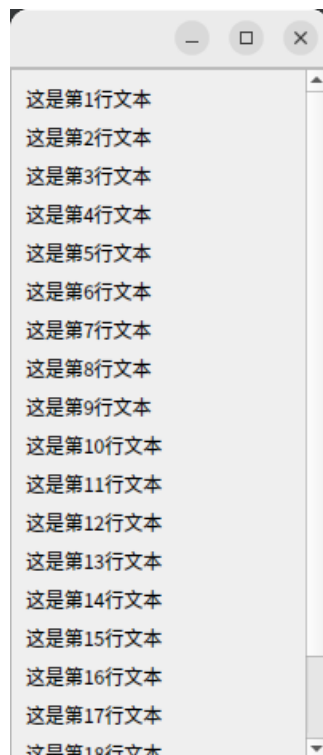
# 向垂直布局中添加内容，例如文本标签
for i in range(20):
    label = QLabel(f"这是第{i + 1}行文本")
    layout.addWidget(label)

# 将布局设置为子控件的布局
scroll_content.setLayout(layout)

# 将子控件设置为QScrollArea的小部件
scroll_area.setWidget(scroll_content)

# 将QScrollArea添加到主窗口
window.setCentralWidget(scroll_area)

# 显示主窗口
window.show()
app.exec_()
```



在这个示例中，我们首先创建了一个QScrollArea，然后创建了一个QWidget作为其子控件。在子控件内，我们创建了一个垂直布局，并向其中添加了多个文本标签作为示例内容。最后，我们将子控件设置为QScrollArea的小部件，这将启用滚动条以便用户可以滚动查看所有内容。

QScrollArea非常有用，特别是在处理大量内容或需要滚动查看的内容时。它可以帮助您在界面上添加滚动功能，以提高用户体验。根据需要，您可以将不同类型的控件嵌套在QScrollArea中。

### 6.3 QToolBox - 创造多功能工具箱

QToolBox是PyQt5中的一个控件，用于创建多功能工具箱界面，通常用于提供多个选项卡。它允许您在一个页面上组织多个子控件，每个子控件对应一个选项卡。用户可以通过切换选项卡来访问不同的子控件内容

#### 主要用途：

- 提供多个选项卡，每个选项卡包含不同的内容或工具。
- 组织和展示多个相关的子控件。

#### QToolBox的基本用法：

要使用QToolBox，首先创建一个QToolBox对象，然后添加多个子控件作为选项卡的内容。以下是创建和配置QToolBox的基本示例：

```
# 导入必要的模块
from PyQt5.QtWidgets import QApplication, QMainWindow, QToolBox, QLabel, QPushButton,
QVBoxLayout, QWidget

# 创建应用程序和主窗口
app = QApplication([])
window = QMainWindow()

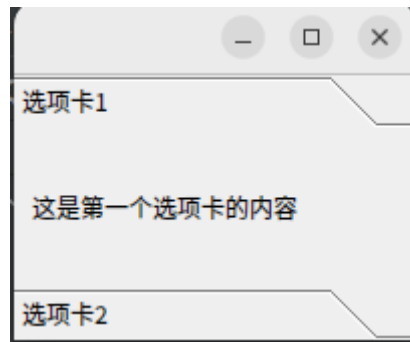
# 创建一个QToolBox
toolbox = QToolBox()

# 创建第一个选项卡
tab1 = QWidget()
label1 = QLabel("这是第一个选项卡的内容")
tab1.layout = QVBoxLayout()
tab1.layout.addWidget(label1)
tab1.setLayout(tab1.layout)
toolbox.addItem(tab1, "选项卡1")

# 创建第二个选项卡
tab2 = QWidget()
label2 = QLabel("这是第二个选项卡的内容")
button = QPushButton("按钮")
tab2.layout = QVBoxLayout()
tab2.layout.addWidget(label2)
tab2.layout.addWidget(button)
tab2.setLayout(tab2.layout)
toolbox.addItem(tab2, "选项卡2")

# 添加QToolBox到主窗口
window.setCentralWidget(toolbox)

# 显示主窗口
window.show()
app.exec_()
```



在这个示例中，我们首先创建了一个QToolBox，然后创建了两个不同的选项卡。每个选项卡是一个QWidget，包含了不同的子控件。我们使用toolbox.addItem()方法将这些选项卡添加到QToolBox中，并为每个选项卡指定了标题。用户可以通过点击选项卡标题来切换不同的内容。

QToolBox非常适合在界面中提供多个相关的选项，并以选项卡形式进行组织。您可以根据需要添加更多的选项卡，并在每个选项卡内放置不同类型的子控件，以满足您的应用需求。

#### 6.4 QTabWidget - 有效管理多标签界面

QTabWidget是PyQt5中用于创建和管理多标签界面的控件。它允许您在同一个窗口中组织多个标签页，每个标签页包含不同的内容或子控件。用户可以通过点击标签页来切换不同的内容，这在创建多功能界面时非常有用。

##### 主要用途：

- 提供多个标签页，每个标签页包含不同的内容或工具。
- 有效组织和管理多标签界面。

##### QTabWidget的基本用法：

要使用QTabWidget，首先创建一个QTabWidget对象，然后为其添加多个标签页。以下是创建和配置QTabWidget的基本示例：

```
# 导入必要的模块
from PyQt5.QtWidgets import QApplication, QMainWindow, QTabWidget, QWidget, QLabel, QPushButton,
QVBoxLayout

# 创建应用程序和主窗口
app = QApplication([])
window = QMainWindow()

# 创建一个QTabWidget
tab_widget = QTabWidget()

# 创建第一个标签页
tab1 = QWidget()
label1 = QLabel("这是标签页1的内容")
tab1.layout = QVBoxLayout()
tab1.layout.addWidget(label1)
tab1.setLayout(tab1.layout)
tab_widget.addTab(tab1, "标签页1")

# 创建第二个标签页
tab2 = QWidget()
label2 = QLabel("这是标签页2的内容")
button = QPushButton("按钮")
tab2.layout = QVBoxLayout()
tab2.layout.addWidget(label2)
tab2.layout.addWidget(button)
tab2.setLayout(tab2.layout)
```

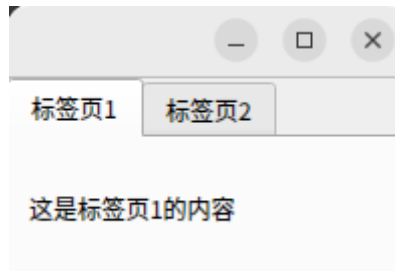
```

tab_widget.addTab(tab2, "标签页2")

# 添加QTabWidget到主窗口
window.setCentralWidget(tab_widget)

# 显示主窗口
window.show()
app.exec_()

```



在这个示例中，我们首先创建了一个QTabWidget，然后创建了两个不同的标签页（tab1和tab2）。每个标签页是一个QWidget，包含了不同的子控件。使用tab\_widget.addTab()方法，我们将这些标签页添加到QTabWidget中，并为每个标签页指定了标签标题。用户可以通过点击标签页标题来切换不同的内容。

QTabWidget非常适合在界面中提供多个标签页，每个标签页包含不同的功能或信息。您可以根据需要添加更多的标签页，并在每个标签页内放置不同类型的子控件，以满足您的应用需求。此外，您还可以自定义标签的外观和行为，使界面更具吸引力和易用性。

## 6.5 QStackedWidget - 实现堆叠界面

QStackedWidget是PyQt5中用于实现多页面布局的控件。它允许您将多个页面叠加在一起，但一次只显示一个页面。这在创建向导、选项卡式界面或多页面应用程序时非常有用。

### 主要用途：

- 创建多个页面，但只显示一个页面。
- 实现向导、选项卡式界面或多页面应用程序。

```

# 导入必要的模块
from PyQt5.QtWidgets import QApplication, QMainWindow, QStackedWidget, QWidget, QLabel,
QPushButton, QVBoxLayout

# 创建应用程序和主窗口
app = QApplication([])
window = QMainWindow()

# 创建一个QStackedWidget
stacked_widget = QStackedWidget()

# 创建第一个页面
page1 = QWidget()
label1 = QLabel("这是页面1的内容")
page1.layout = QVBoxLayout()
page1.layout.addWidget(label1)
page1.setLayout(page1.layout)
stacked_widget.addWidget(page1)
button = QPushButton("切换到页面2")
button.clicked.connect(lambda: stacked_widget.setCurrentIndex(1)) # 切换到页面2的按钮事件
page1.layout.addWidget(button)

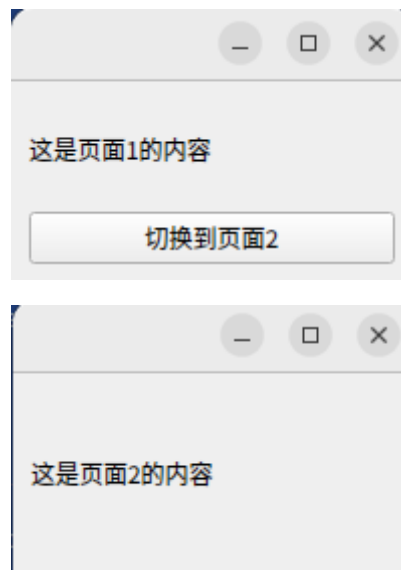
```

```
# 创建第二个页面
page2 = QWidget()
label2 = QLabel("这是页面2的内容")

page2.layout = QVBoxLayout()
page2.layout.addWidget(label2)
page2.setLayout(page2.layout)
stacked_widget.addWidget(page2)

# 添加QStackedWidget到主窗口
window.setCentralWidget(stackd_widget)

# 显示主窗口
window.show()
app.exec_()
```



在这个示例中，我们首先创建了一个QStackedWidget，然后创建了两个不同的页面（page1和page2）。使用stacked\_widget.addWidget()方法，我们将这些页面添加到QStackedWidget中。默认情况下，QStackedWidget会显示第一个添加的页面（即page1）。

我们还创建了一个按钮，点击该按钮将当前页面切换到page2。这是通过使用stacked\_widget.setCurrentIndex()方法来实现的。在这种方式下，您可以轻松地在不同的页面之间进行切换。

QStackedWidget非常适合在需要多个页面但一次只显示一个页面的情况下使用。这在创建向导、选项卡式界面或多步骤应用程序时非常有用。您可以根据需要添加更多的页面，并在每个页面中放置不同类型的子控件，以实现复杂的界面布局。

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

# Form implementation generated from reading ui file 'home.ui'
#
# Created by: PyQt5 UI code generator 5.15.6
#
# WARNING: Any manual changes made to this file will be lost when pyuic5 is
# run again. Do not edit this file unless you know what you are doing.
```

```
from PyQt5 import QtCore, QtGui, QtWidgets
```

```
class Ui_MainWindow(object):
```

```
    def setupUi(self, MainWindow):
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(733, 529)
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")
        self.groupBox = QtWidgets.QGroupBox(self.centralwidget)
        self.groupBox.setGeometry(QtCore.QRect(0, 0, 761, 131))
        self.groupBox.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 1, y2: 0,
stop: 0 #011624, stop: 0.5 #004a72, stop: 1 #011624);")
        self.groupBox.setTitle("")
        self.groupBox.setObjectName("groupBox")
        self.horizontalLayoutWidget = QtWidgets.QWidget(self.groupBox)
        self.horizontalLayoutWidget.setGeometry(QtCore.QRect(-1, 50, 751, 80))
        self.horizontalLayoutWidget.setObjectName("horizontalLayoutWidget")
        self.horizontalLayout = QtWidgets.QHBoxLayout(self.horizontalLayoutWidget)
        self.horizontalLayout.setContentsMargins(0, 0, 0, 0)
        self.horizontalLayout.setObjectName("horizontalLayout")
        spacerItem = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem)
        spacerItem1 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem1)
        spacerItem2 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem2)
        spacerItem3 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem3)
        self.pushButton_3 = QtWidgets.QPushButton(self.horizontalLayoutWidget)
        font = QtGui.QFont()
        font.setPointSize(12)
        self.pushButton_3.setFont(font)
        self.pushButton_3.setStyleSheet("color: rgb(255, 255, 255);")
        self.pushButton_3.setObjectName("pushButton_3")
        self.horizontalLayout.addWidget(self.pushButton_3)
        spacerItem4 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem4)
        self.pushButton_2 = QtWidgets.QPushButton(self.horizontalLayoutWidget)
        font = QtGui.QFont()
        font.setPointSize(12)
        self.pushButton_2.setFont(font)
        self.pushButton_2.setStyleSheet("color: rgb(255, 255, 255);")
        self.pushButton_2.setObjectName("pushButton_2")
        self.horizontalLayout.addWidget(self.pushButton_2)
        spacerItem5 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem5)
        self.pushButton = QtWidgets.QPushButton(self.horizontalLayoutWidget)
        font = QtGui.QFont()
        font.setPointSize(12)
        self.pushButton.setFont(font)
        self.pushButton.setStyleSheet("color: rgb(255, 255, 255);")
```

```

        self.pushButton.setObjectName("pushButton")
        self.horizontalLayout.addWidget(self.pushButton)
        spacerItem6 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem6)
        spacerItem7 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem7)
        spacerItem8 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem8)
        spacerItem9 = QtWidgets.QSpacerItem(40, 20, QtWidgets.QSizePolicy.Expanding,
QtWidgets.QSizePolicy.Minimum)
        self.horizontalLayout.addItem(spacerItem9)
        self.stackedWidget = QtWidgets.QStackedWidget(self.centralwidget)
        self.stackedWidget.setGeometry(QtCore.QRect(0, 130, 761, 401))
        self.stackedWidget.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 1, y2:
0, stop: 0 #011624, stop: 0.5 #004a72, stop: 1 #011624);")
        self.stackedWidget.setObjectName("stackedWidget")
        self.page = QtWidgets.QWidget()
        self.page.setObjectName("page")
        self.graphicsView = QtWidgets.QGraphicsView(self.page)
        self.graphicsView.setGeometry(QtCore.QRect(70, 0, 301, 192))
        self.graphicsView.setStyleSheet("background-color: rgb(85, 170, 0);")
        self.graphicsView.setObjectName("graphicsView")
        self.graphicsView_2 = QtWidgets.QGraphicsView(self.page)
        self.graphicsView_2.setGeometry(QtCore.QRect(390, 0, 301, 192))
        self.graphicsView_2.setStyleSheet("background-color: rgb(170, 85, 255);")
        self.graphicsView_2.setObjectName("graphicsView_2")
        self.horizontalLayoutWidget_2 = QtWidgets.QWidget(self.page)
        self.horizontalLayoutWidget_2.setGeometry(QtCore.QRect(69, 229, 621, 131))
        self.horizontalLayoutWidget_2.setObjectName("horizontalLayoutWidget_2")
        self.horizontalLayout_2 = QtWidgets.QHBoxLayout(self.horizontalLayoutWidget_2)
        self.horizontalLayout_2.setContentsMargins(0, 0, 0, 0)
        self.horizontalLayout_2.setObjectName("horizontalLayout_2")
        self.graphicsView_6 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
        self.graphicsView_6.setStyleSheet("background-color: rgb(255, 85, 0);")
        self.graphicsView_6.setObjectName("graphicsView_6")
        self.horizontalLayout_2.addWidget(self.graphicsView_6)
        self.graphicsView_5 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
        self.graphicsView_5.setStyleSheet("background-color: rgb(0, 85, 127);")
        self.graphicsView_5.setObjectName("graphicsView_5")
        self.horizontalLayout_2.addWidget(self.graphicsView_5)
        self.graphicsView_4 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
        self.graphicsView_4.setStyleSheet("background-color: rgb(170, 0, 127);")
        self.graphicsView_4.setObjectName("graphicsView_4")
        self.horizontalLayout_2.addWidget(self.graphicsView_4)
        self.graphicsView_3 = QtWidgets.QGraphicsView(self.horizontalLayoutWidget_2)
        self.graphicsView_3.setStyleSheet("background-color: rgb(170, 85, 127);")
        self.graphicsView_3.setObjectName("graphicsView_3")
        self.horizontalLayout_2.addWidget(self.graphicsView_3)
        self.stackedWidget.addWidget(self.page)
        self.page_2 = QtWidgets.QWidget()
        self.page_2.setObjectName("page_2")
        self.checkBox_2 = QtWidgets.QCheckBox(self.page_2)
        self.checkBox_2.setGeometry(QtCore.QRect(80, 100, 161, 75))
        self.checkBox_2.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")

```



```

self.checkBox_2.setObjectName("checkBox_2")
self.checkBox = QtWidgets.QCheckBox(self.page_2)
self.checkBox.setGeometry(QtCore.QRect(80, 10, 161, 75))
self.checkBox.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")
self.checkBox.setObjectName("checkBox")
self.checkBox_3 = QtWidgets.QCheckBox(self.page_2)
self.checkBox_3.setGeometry(QtCore.QRect(80, 190, 161, 75))
self.checkBox_3.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")
self.checkBox_3.setObjectName("checkBox_3")
self.checkBox_4 = QtWidgets.QCheckBox(self.page_2)
self.checkBox_4.setGeometry(QtCore.QRect(80, 280, 161, 75))
self.checkBox_4.setStyleSheet("background: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
stop: 0 #0e3c62, stop: 0.5 #124d7e, stop: 1 #1963a0);")
self.checkBox_4.setObjectName("checkBox_4")
self.stackedWidget.addWidget(self.page_2)
self.page_3 = QtWidgets.QWidget()
self.page_3.setObjectName("page_3")
self.stackedWidget.addWidget(self.page_3)
MainWindow.setCentralWidget(self.centralwidget)

self.retranslateUi(MainWindow)
self.stackedWidget.setCurrentIndex(1)
QtCore.QMetaObject.connectSlotsByName(MainWindow)

self.pushButton_3.clicked.connect(lambda:
self.stackedWidget.setCurrentWidget(self.page))
self.pushButton_2.clicked.connect(lambda:
self.stackedWidget.setCurrentWidget(self.page_2))
self.pushButton.clicked.connect(lambda:
self.stackedWidget.setCurrentWidget(self.page_3))

def retranslateUi(self, MainWindow):
    _translate = QtCore.QCoreApplication.translate
    MainWindow.setWindowTitle(_translate("MainWindow", "MainWindow"))
    self.pushButton_3.setText(_translate("MainWindow", "单源识别"))
    self.pushButton_2.setText(_translate("MainWindow", "多源融合"))
    self.pushButton.setText(_translate("MainWindow", "协同感知"))
    self.checkBox_2.setText(_translate("MainWindow", "近红外"))
    self.checkBox.setText(_translate("MainWindow", "可见光"))
    self.checkBox_3.setText(_translate("MainWindow", "热红外"))
    self.checkBox_4.setText(_translate("MainWindow", "高光谱"))

```

## 6.6 QFrame - 为元素提供框架

QFrame是PyQt5中的一个控件，主要用于为其他控件提供框架和背景。它可以改善界面的视觉效果，为元素之间创建分隔线或边框。

### 主要用途：

- 为其他控件提供框架、边框和背景。
- 改进界面的视觉效果。

### QFrame的基本用法：

要使用QFrame，首先创建一个QFrame对象，然后将其他控件添加到该QFrame中。以下是一个简单示例，演示如何使用QFrame为两个QLabel提供框架：

```
# 导入必要的模块
from PyQt5.QtWidgets import QApplication, QMainWindow, QFrame, QLabel, QVBoxLayout, QWidget

# 创建应用程序和主窗口
app = QApplication([])
window = QMainWindow()

# 创建一个QFrame作为容器
frame = QFrame()

# 创建两个QLabel
label1 = QLabel("标签1")
label2 = QLabel("标签2")

# 创建一个垂直布局
layout = QVBoxLayout()

# 将QLabel添加到布局中
layout.addWidget(label1)
layout.addWidget(label2)

# 设置QFrame的布局为垂直布局
frame.setLayout(layout)

# 设置QFrame的边框样式
frame.setFrameShape(QFrame.Box)
frame.setLineWidth(2) # 设置边框宽度

# 添加QFrame到主窗口
window.setCentralWidget(frame)

# 显示主窗口
window.show()
app.exec_()
```



在这个示例中，我们首先创建了一个QFrame，然后在该QFrame中创建了两个QLabel。我们使用布局将这两个QLabel添加到QFrame中，然后设置QFrame的边框样式为Box，同时指定了边框的宽度。

QFrame的主要作用是其他控件提供框架和边框，这可以增强界面的可读性和美观性。您可以根据需要设置不同的边框样式和背景颜色，以满足设计需求。在实际应用中，QFrame通常与其他控件一起使用，以创建具有更好外观和分隔效果的界面元素。

## 6.7 QWidget - 基础控件和容器

QWidget是PyQt5中的基础控件，同时也是许多其他控件的父类。它是构建用户界面的基本构建块，具有丰富的功能和灵活性。

### 主要功能：

- 作为其他控件的父类，提供了基本的控件功能。
- 可用于创建自定义控件和容器。

### QWidget的基本用法：

在PyQt5中，几乎所有的用户界面元素都是QWidget或其子类。您可以使用QWidget创建窗口、对话框、按钮、文本框等等。以下是一个简单的示例，演示如何创建一个窗口并在其中添加一个按钮：

```
# 导入必要的模块
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton

# 创建应用程序和主窗口
app = QApplication([])
window = QMainWindow()

# 创建一个QWidget作为窗口的中央部件
central_widget = QWidget()

# 创建一个按钮
button = QPushButton("点击我")

# 将按钮添加到QWidget中
central_widget.setCentralWidget(button)

# 设置QWidget为主窗口的中央部件
window.setCentralWidget(central_widget)

# 显示主窗口
window.show()
app.exec_()
```

在这个示例中，我们首先创建了一个QWidget作为窗口的中央部件，然后创建了一个按钮，并将按钮添加到QWidget中。最后，我们将QWidget设置为主窗口的中央部件，以便显示在窗口中央。

QWidget的灵活性使其成为自定义控件和容器的理想选择。您可以继承QWidget类并重写其方法，以创建自定义控件，也可以将多个QWidget组合在一起以创建复杂的用户界面。QWidget还提供了丰富的绘图和事件处理功能，可以满足各种界面设计需求。

总之，QWidget是PyQt5中的基础控件，它在用户界面设计中扮演着重要的角色，可以用于创建各种类型的界面元素，同时也是自定义控件和容器的基础。

## 6.8 QMdiArea - 多文档界面

QMdiArea（多文档界面区域）是PyQt5中用于管理多个内嵌窗口的控件。它适用于那些需要在单个应用程序中打开和管理多个文档或子窗口的情况。QMdiArea提供了一种方便的方式来组织和管理这些文档，使用户能够轻松切换和操作它们。

### 主要功能：

- 允许在单个应用程序中同时打开多个文档窗口。

- 提供选项卡式或窗口式的界面布局。
- 支持文档的排列、最小化、最大化和关闭等操作。

#### QMdiArea的基本用法：

以下是一个示例，演示如何使用QMdiArea创建一个多文档界面，其中包含多个内嵌窗口：

```
from PyQt5.QtWidgets import QApplication, QMainWindow, QMdiArea, QMdiSubWindow, QTextEdit

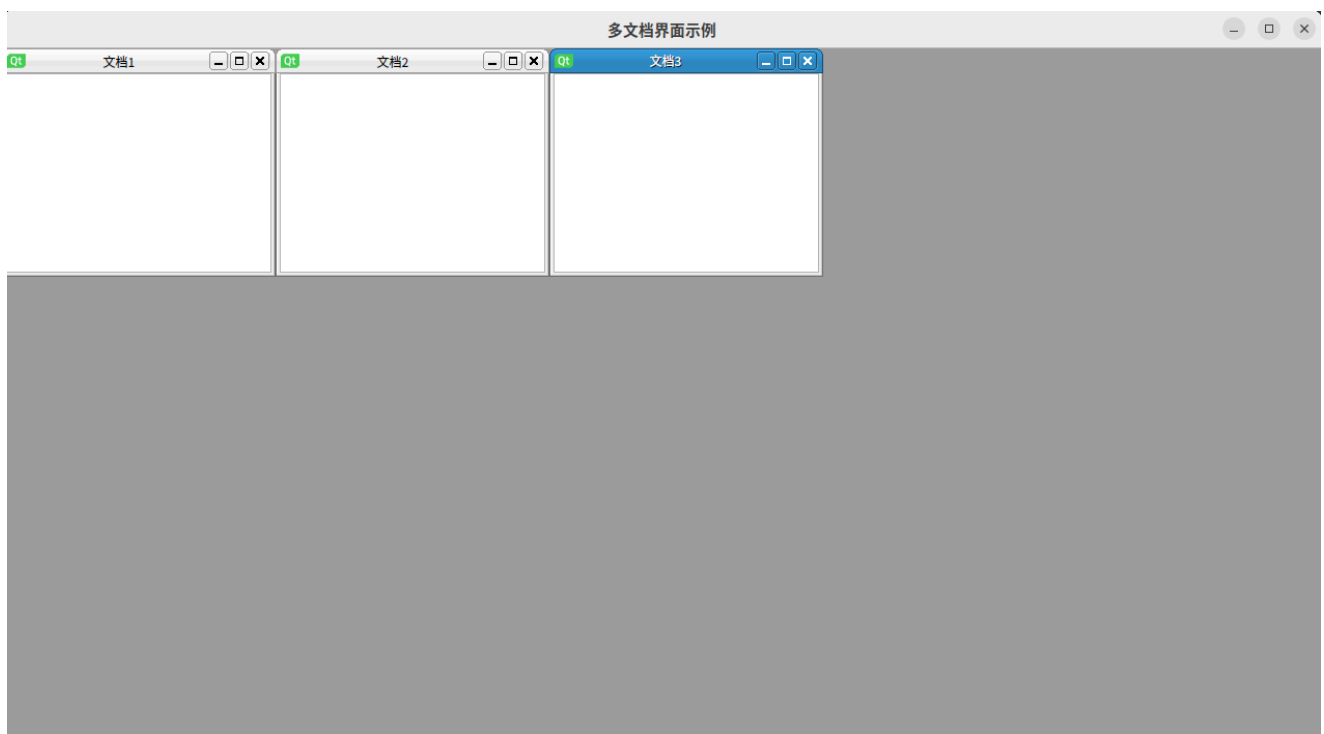
app = QApplication([])

# 创建主窗口
main_window = QMainWindow()
main_window.setWindowTitle("多文档界面示例")

# 创建QMdiArea作为主窗口的中央部件
mdi_area = QMdiArea()
main_window.setCentralWidget(mdi_area)

# 创建多个内嵌窗口
for i in range(1, 4):
    sub_window = QMdiSubWindow()
    sub_window.setWindowTitle(f"文档{i}")
    text_edit = QTextEdit()
    sub_window.setWidget(text_edit)
    mdi_area.addSubWindow(sub_window)

main_window.show()
app.exec_()
```



在这个示例中，我们首先创建了一个主窗口，并将QMdiArea设置为主窗口的中央部件。然后，我们创建了三个内嵌窗口，每个窗口包含一个文本编辑器（使用QTextEdit）。最后，我们使用addSubWindow方法将这些内嵌窗口添加到QMdiArea中。

QMdiArea还支持多种布局选项，您可以根据需要选择选项卡式或窗口式的界面布局。用户可以方便地切换、排列和管理多个文档窗口，使QMdiArea成为处理多文档应用程序的理想选择。

总之，QMDiArea是PyQt5中用于创建多文档界面的控件，它允许在单个应用程序中管理多个文档窗口，提供了丰富的功能来满足多文档应用程序的需求。

## 6.9 QDockWidget - 可停靠和可移动的控件

QDockWidget是PyQt5中用于创建可停靠和可移动的控件的控件。它通常用于创建应用程序的工具栏、面板或其他可停靠部件，使用户能够根据需要重新排列和停靠它们。

### 主要功能：

允许创建可停靠的控件，如工具栏、属性面板等。

用户可以将这些可停靠控件拖动到主窗口的不同位置。

提供了灵活的布局选项，包括停靠在主窗口的四个边缘或浮动在主窗口上方。

### QDockWidget的基本用法：

以下是一个示例，演示如何使用QDockWidget创建一个可停靠的工具栏：

```
from PyQt5.QtWidgets import QApplication, QMainWindow, QDockWidget, QTextEdit, QPushButton

app = QApplication([])

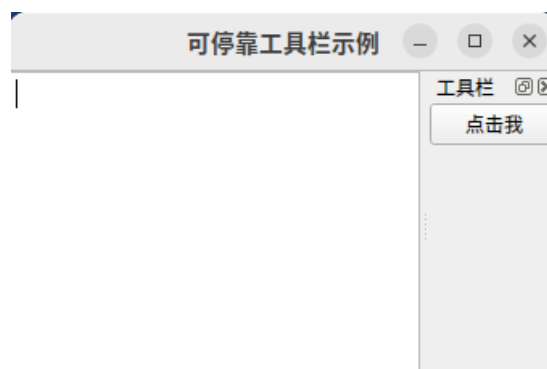
# 创建主窗口
main_window = QMainWindow()
main_window.setWindowTitle("可停靠工具栏示例")

# 创建文本编辑器
text_edit = QTextEdit()
main_window.setCentralWidget(text_edit)

# 创建QDockWidget作为可停靠的工具栏
dock_widget = QDockWidget("工具栏", main_window)
button = QPushButton("点击我")
dock_widget.setWidget(button)

# 将工具栏停靠在主窗口的右侧
main_window.addDockWidget(2, dock_widget)

main_window.show()
app.exec_()
```



工具栏可以移出

在这个示例中，我们首先创建了一个主窗口，并在其中央部件设置了一个文本编辑器。然后，我们创建了一个QDockWidget，将一个按钮添加到其中作为工具栏的内容。最后，我们使用addDockWidget方法将工具栏停靠在主窗口的右侧。

用户可以拖动工具栏到主窗口的不同位置，也可以将其浮动在主窗口上方。这使得QDockWidget非常适合创建可自定义布局的应用程序，用户可以根据需要重新排列控件，以满足其工作流程。

总之，QDockWidget是PyQt5中用于创建可停靠和可移动的控件的控件，它为应用程序提供了灵活的布局选项，允许用户根据需要重新排列和停靠控件，提高了界面的自定义性和用户体验。

## 6.10 QAxWidget - 集成ActiveX控件

QAxWidget是PyQt5中用于集成ActiveX控件的控件。ActiveX控件是一种微软开发的技术，用于在Windows平台上创建交互式 and 可视化的控件，如网页浏览器、媒体播放器、Office文档等。使用QAxWidget，您可以在PyQt5应用程序中嵌入和使用这些ActiveX控件。

### 主要功能：

- 允许在PyQt5应用程序中嵌入和使用ActiveX控件。
- 支持与ActiveX控件的交互和通信。
- 可以在窗口中显示Web浏览器、Office文档、媒体播放器等。

### QAxWidget的基本用法：

以下是一个示例，演示如何使用QAxWidget嵌入Internet Explorer作为ActiveX控件：

```
from PyQt5.QtWidgets import QApplication, QMainWindow, QAxWidget

app = QApplication([])

# 创建主窗口
main_window = QMainWindow()
main_window.setWindowTitle("ActiveX控件示例")

# 创建QAxWidget用于嵌入Internet Explorer
ax_widget = QAxWidget(main_window)
ax_widget.setControl("Shell.Explorer")
ax_widget.show()

main_window.setCentralWidget(ax_widget)
main_window.show()

app.exec_()
```

在这个示例中，我们首先创建了一个主窗口，并在其中央部件设置了一个QAxWidget。然后，我们使用setControl方法指定要嵌入的ActiveX控件，这里是"Shell.Explorer"，即Internet Explorer。最后，我们将QAxWidget设置为主窗口的中央部件，以显示嵌入的Internet Explorer。

通过使用QAxWidget，您可以在PyQt5应用程序中嵌入各种ActiveX控件，从而实现与这些控件的交互和使用，为您的应用程序增加更多的功能和可视化元素。

总之，QAxWidget是PyQt5中用于集成ActiveX控件的控件，它使您能够在应用程序中嵌入和使用各种ActiveX控件，从而扩展应用程序的功能和可视化元素。

### Tip

容器和控件在PyQt5界面设计中扮演着重要的角色，它们帮助我们组织和展示应用程序的用户界面元素。

QGroupBox允许我们创建分组框，用于组织和分类界面元素，提高用户体验。

QScrollArea允许我们添加滚动功能，以处理超出视图范围的内容，确保用户能够访问所有信息。

QToolBox是一个多功能的工具箱，用于提供多个选项卡界面，适用于各种应用场景。

QTabWidget用于有效管理多标签界面，可自定义标签的外观和行为，为用户提供更好的导航体验。

QStackedWidget用于实现多页面布局，允许我们在不同的界面之间进行切换，以响应用户的操作。

QFrame可为元素提供框架，改进界面的视觉效果，增强用户界面的吸引力。

QWidget是其他控件的基础，我们可以自定义和扩展它，以创建自定义控件。

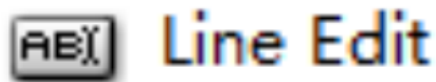
QMdiArea用于创建多文档界面，管理多个内嵌窗口，适用于开发多文档应用程序。

QDockWidget是可停靠和可移动的控件，可用于创建灵活的布局，允许用户自定义界面。

## 7. 1 3 种输入控件(Input Widgets)

### 7.1 QLineEdit - 基础文本输入

LineEdit是单行文本框，可以从字面意思理解，即：它只能输入单行字符串。它在Qt Designer设计器中的图标如下图所示：



LineEdit控件对应PyQt5中的QLineEdit类，该类的常用方法和说明如下表所示：

方法	说明
setText()	设置文本框内容
text()	获取文本框内容
setPlaceholderText()	设置文本框内浮现文字
setMaxLength()	设置允许文本框内输入字符的最大长度
setAlignment()	设置文本对齐方式
setReadOnly()	设置文本框内容为只读模式
setFocus()	使文本框得到焦点
setEchoMode()	设置文本框显示字符的模式，有以下4种模式： 1. QLineEdit.Normal：正常显示输入的字符，这是默认设置 2. QLineEdit.NoEcho：不显示任何输入的字符，适用于即使符合密码长度也需要保密的密码 3. QLineEdit.Password：显示与平台相关的密码掩码字符，而不是实际输入的字符 4. QLineEdit.PasswordEchoOnEdit：在编辑时显示字符，失去焦点后显示密码掩码字符
setValidator()	设置文本框验证器，有以下3种模式： 1. QIntValidator：限制输入整数 2. QDoubleValidator：限制输入小数 3. QRegExpValidator：检查输入是否符合设置的正则表达式
setInputMask()	设置掩码，掩码通常由掩码字符和分隔符组成，后面可以跟一个分号和空白字符，空白字符在编辑完成后会从文本框中删除，常用的掩码有以下3种形式： 1. 日期掩码：0000-00-00 2. 时间掩码：00:00:00 3. 序列号掩码：>AAAAA-A AAAA-A AAAA-A AAAA-A AAAA;#
clear()	清除文本框内容

QLineEdit类的常用信号及说明如下表所示：

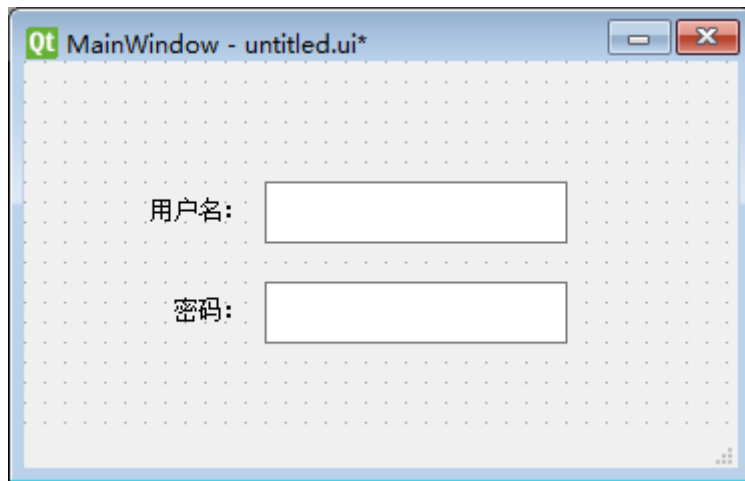
信号	说明
textChanged()	当更改文本框中的内容时发射该信号
editingFinished()	当文本框中的内容编辑结束时发射该信号，以按下<Enter>键为编辑结束标志

**实例：包括用户名和密码的登录窗口**

使用LineEdit控件，并结合Label控件制作一个简单的登录窗口，其中包含用户名和密码输入框。密码要求是8位数字，并且以掩码形式显示，步骤如下：

步骤1. 打开Qt Designer设计器，根据需求，从工具箱向主窗口中放入两个Label控件与两个LineEdit控件，然后分别将两个Label控件的text值修改为“用户名：”和“密码：”。如下图所示：





步骤2. 设计完成后，保存为.ui文件，使用PyUIC工具将其转换为.py文件，并在表示密码的LineEdit文本框下面使用setEchoMode()方法将其设置为密码文本，同时使用setValidator()方法为其设置验证器，控制只能输入8位数字，代码如下：

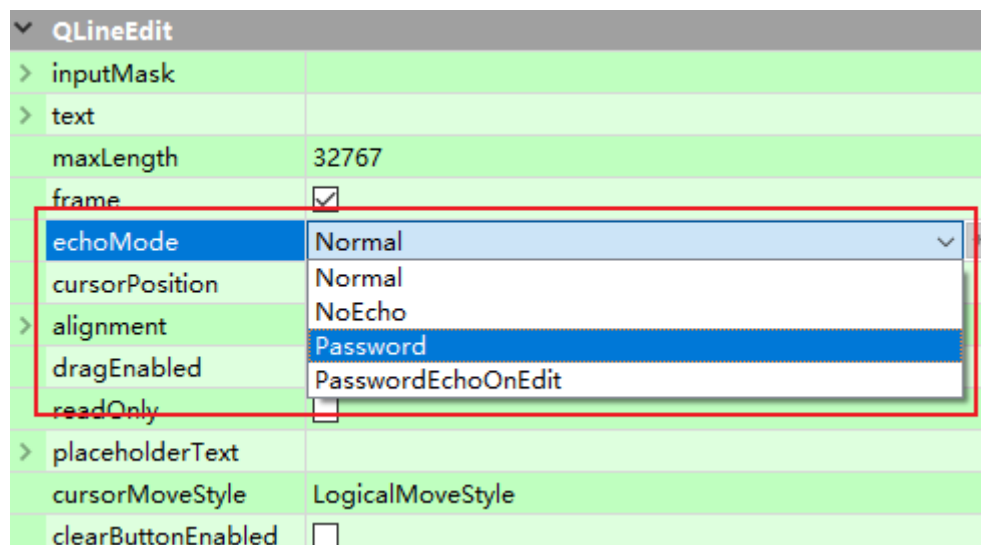
```
# 设置文本框为密码
self.lineEdit_2.setEchoMode(QtWidgets.QLineEdit.Password)

# 限制只能输入8位数字
self.lineEdit_2.setValidator(QtGui.QIntValidator(10000000, 99999999))
```

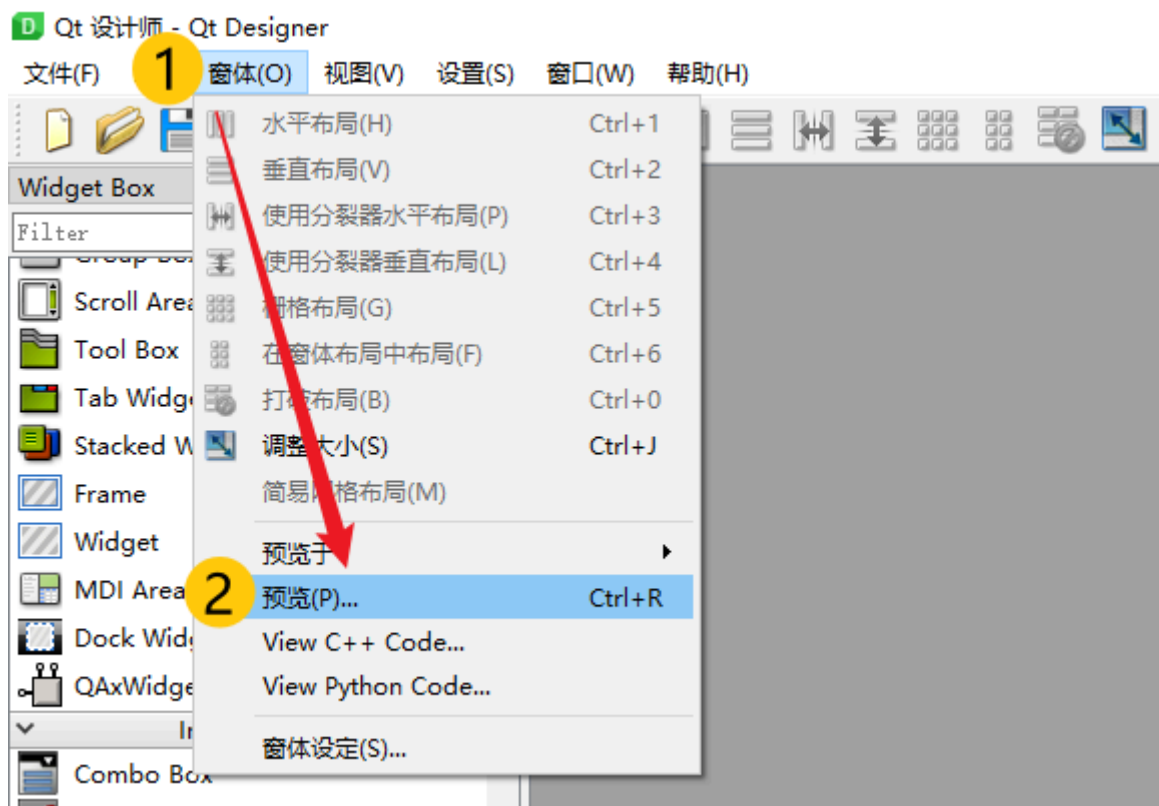
代码运行后并输入内容，效果如下图所示：



当然，这些属性也都可以直接在Qt Designer设计器的属性编辑器中设置，如下图所示：



预览有2种方式，方式1：按Ctrl + R进行预览，方式2：鼠标单击Qt Designer设计器顶部的菜单栏中“窗体(O)”，在下拉弹窗中再单击“预览(P)...”即可预览。如下图：



## textChanged信号

textChanged信号在一些要求输入值时实时执行操作的场景下使用，比如在网上购物时，更改购买商品的数量，总价就会实时变化。

如果你设计时遇到类似这样的功能需求，就可以通过LineEdit控件中的textChanged信号来实现。实例演示：假设牙刷2元1支，我需要购买100支。当我输入数量100时，下面标签控件实时显示出我需要支付200元。代码如下：

```
# 1
self.lineEdit.setValidator(QtGui.QIntValidator(10000000, 99999999))

# 2
self.lineEdit.textChanged.connect(self.total_price)

# 3
def total_price(self):

# 4
    n = self.lineEdit.text()

# 5
    buy_number = int(n)

# 6
    self.label_4.setText(str(buy_number*2))
```

因篇幅较长，详细注释如下：

# 1: 限制单行输入框只能输入8位数字。因为购买数量也必须是数字，所以这里代码不变。

# 2: 将单行输入框的textChanged信号和下面的自定义函数total\_price关联起来，关联函数是connect()。

# 3: 准备定义一个名为total\_price的函数，参数默认为self。

# 4: 获取单行输入框的内容并赋值给变量n。

# 5: 由于通过text()函数获取的数据类型是字符串，这里通过int()函数转换成整型。

# 6: label\_4是显示最终需要支付价格的标签控件，将数量buy\_number乘以价格2元之后得到一个总价格是整型，但setText()函数的参数类型是字符串，所以我们计算完还要将结果再转换回字符串。通过setText()函数将label\_4显示的内容变为需要支付的价格。

代码运行后，我们在输入框内输入数量100，下面的label\_4标签控件就实时显示出需要支付的钱啦~ 如下图所示：



### 输入掩码和验证器：

QLineEdit支持输入掩码和验证器，以限制用户的输入。例如，我们可以通过以下方式限制用户只能输入数字：

```
# 创建只允许输入数字的QLineEdit
numeric_line_edit = QLineEdit()
numeric_line_edit.setInputMask('D')
```

这将使QLineEdit只接受数字的输入。此外，可以使用验证器来执行更灵活的验证。例如，我们可以创建一个只允许输入0到100的验证器：

```
from PyQt5.QtGui import QIntValidator

# 创建验证器
validator = QIntValidator(0, 100)

# 将验证器应用于QLineEdit
numeric_line_edit.setValidator(validator)
```

通过这样的设置，用户只能输入介于0和100之间的整数。

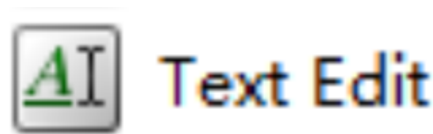
通过QLineEdit的输入掩码和验证器，我们能够在用户输入时实现更加精确和合法的文本控制，提高应用的稳定性和用户体验。

7.2 QTextEdit - 多行文本编辑

有单行输入框，那么就会有多行输入框，而TextEdit就是多行文本框控件。它主要用来显示多行的文本内容，当文本内容超出控件的显示范围时，该控件会自动显示滚动条。

另外，TextEdit控件不仅可以显示纯文本内容，还支持显示HTML网页。

TextEdit控件在Qt Designer设计器中的图标如下所示：



TextEdit控件对应PyQt5中的QTextEdit类，该类的常用方法及说明如下表：

方法	说明
setPlainText()	设置文本内容
toPlainText()	获取文本内容
setTextColor()	设置文本颜色，例： QtGui.QColor(255,0,0)
setTextBackgroundColor()	设置文本背景色，参数同setTextColor()
setHtml()	设置HTML文档内容
toHtml()	获取HTML文档内容
setWordWrapMode()	设置自动换行
clear()	清除所有内容

设置多行文本和HTML文本的对比显示

使用Qt Designer设计器创建一个MainWindow窗口，其中添加2个TextEdit控件，然后保存为.ui文件，使用PyUIC工具将其转换为.py文件。

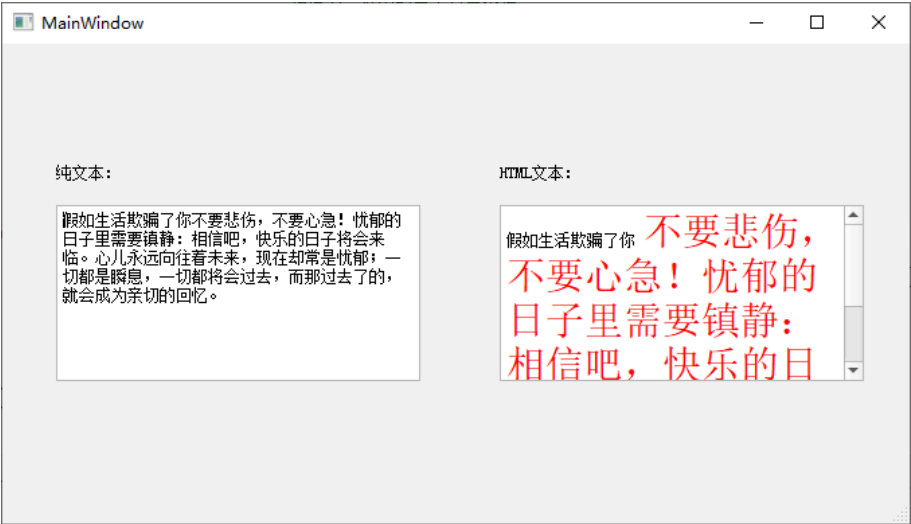
然后分别使用setPlainText()方法和setHtml()方法为2个TextEdit控件设置要显示的文本内容，代码如下：

```
# 设置纯文本显示
self.textEdit.setPlainText('假如生活欺骗了你'
                             '不要悲伤，不要心急！'
                             '忧郁的日子里需要镇静：'
                             '相信吧，快乐的日子将会来临。'
                             '心儿永远向往着未来，'
                             '现在却常是忧郁：'
                             '一切都是瞬息，'
                             '一切都将会过去，'
                             '而那过去了的，'
                             '就会成为亲切的回忆。')

# 设置HTML文本显示
self.textEdit_2.setHtml(
```

```
"假如生活欺骗了你 "
"<font color = 'red' size = 10>
不要悲伤，不要心急！忧郁的日子里需要镇静：相信吧，快乐的日子将会过去，而那过去了的，就会成为亲切的回忆。")
```

运行效果如下图：

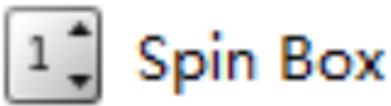


7.3 QSpinBox & QDoubleSpinBox - 数字输入

QSpinBox - 整数输入:

SpinBox是一个整数数字选择控件，该控件提供了一对上下箭头，用户可以单击上下箭头选择数值，也可以直接输入。

如果输入的数值大于设置的最大值，或者小于设置的最小值，SpinBox将不会接受输入。SpinBox控件在Qt Designer中的图标如下图所示：



SpinBox控件对应PyQt5中的QSpinBox类，该类常用方法及说明如下表：

方法	说明
setValue()	设置当前值
setMaximum()	设置最大值
setMinimum()	设置最小值
setRange()	设置取值范围（包括最大值与最小值）
setSingleStep()	设置单击上下箭头时的步长值
value()	获取控件中的值

在默认情况下，SpinBox控件的取值范围是0~99，步长值是1。

在单击SpinBox控件的上下箭头时，可以通过发射valueChanged信号，获取控件中的当前值

## 获取SpinBox中选择的数字

使用Qt Designer设计器创建一个MainWindow窗口，其中添加2个Label控件和1个SpinBox控件，然后保存为.ui文件，使用PyUIC工具将其转换为.py文件。

在转换后的.py文件中，分别设置数字选择控件的最小值、最大值和步长值。

代码如下：

```
# 设置最小值为0
self.spinBox.setMinimum(0)

# 设置最大值为100
self.spinBox.setMaximum(100)

# 设置步长值为2
self.spinBox.setSingleStep(2)
```

可以使用setRange()方法，简写上面的3行代码：

```
# 设置取值范围为：0~100
self.spinBox.setRange(0, 100)

# 设置步长值为2
self.spinBox.setSingleStep(2)
```

下面我们自定义一个get\_value函数，在函数体中我们使用value()方法获取SpinBox控件的当前值，并显示在Label标签控件中。代码如下：

```
def get_value(self):
    self.label_2.setText(str(self.spinBox.value()))
```

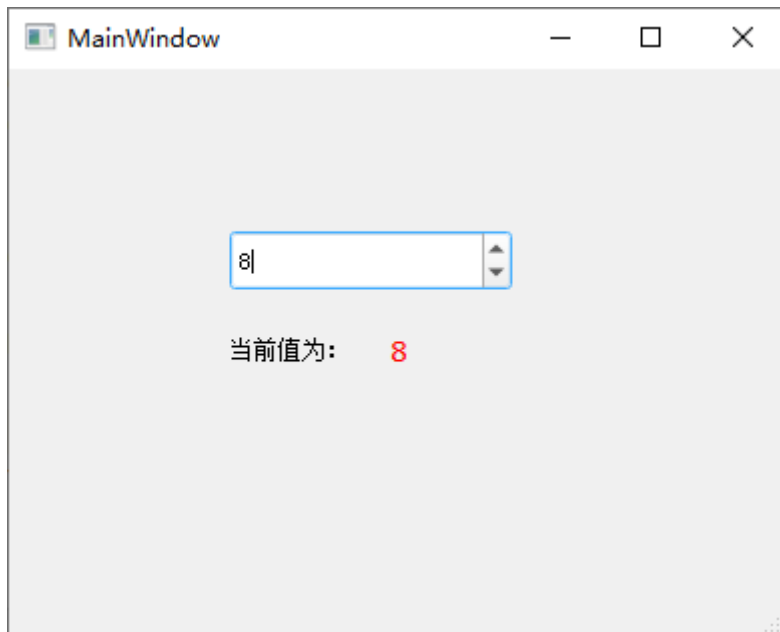
将SpinBox控件的valueChanged信号和自定义函数get\_value关联起来：

```
self.spinBox.valueChanged.connect(self.get_value)
```

### ⚠ Caution

**关联代码要写在自定义函数之前**

以上代码运行结果如下图所示：



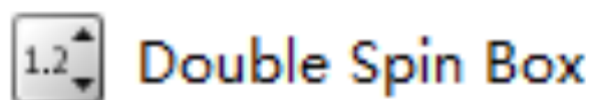
或使用：



### QDoubleSpinBox - 浮点数输入：

DoubleSpinBox 与 SpinBox 类似，区别在于 DoubleSpinBox 是用来选择小数数字，并且默认保留 2 位小数。DoubleSpinBox 对应 PyQt5 中的 QDoubleSpinBox 类。

DoubleSpinBox 控件在 Qt Designer 中的图标如下图所示：



### 设置 DoubleSpinBox 中的小数位数并获取选择的数字

使用 Qt Designer 创建一个 MainWindow 窗口，其中添加 2 个 Label 控件和 1 个 DoubleSpinBox 控件，然后保存为 .ui 文件，使用 PyUIC 工具将 .ui 文件转换为 .py 文件。

第一步，在 .py 文件中，分别设置小数数字选择控件的最小值、最大值、步长值，以及保留 2 位小数。代码如下：

```
# 设置取值范围为：0 ~ 99.99
self.doubleSpinBox.setRange(0, 99.99)

# 设置步长值为：0.01
self.doubleSpinBox.setSingleStep(0.01)

# 保留2位小数
self.doubleSpinBox.setDecimals(2)
```

第二步，自定义 get\_value 函数，函数体中使用 value() 方法获取 DoubleSpinBox 的当前值，并显示在 Label 中。代码如下：

```
def get_value(self):
    self.label_2.setText(str(self.doubleSpinBox.value()))
```

第三步，将DoubleSpinBox的valueChanged()和自定义函数get\_value关联。代码如下：

```
self.doubleSpinBox.valueChanged.connect(self.get_value)
```

以上代码运行后，效果如下图所示：



#### 7.4 QComboBox - 下拉选择列表

QComboBox是PyQt5中常用的控件之一，用于提供下拉选择列表的功能。以下是QComboBox的用途、功能以及如何进行基本操作的介绍。

##### 用途和功能:

QComboBox用于展示用户可以选择的多个选项，并允许用户从中选择一个或输入自定义内容。常见的用途包括选择列表、国家/地区选择、颜色选择等。

##### 基本操作:

以下是一些基本操作，演示如何使用QComboBox添加、移除和编辑项目。

```
from PyQt5.QtWidgets import QApplication, QWidget, QComboBox, QVBoxLayout

class ComboBoxApp(QWidget):
    def __init__(self):
        super().__init__()

        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        # 创建QComboBox
        combo_box = QComboBox()

        # 添加项目
        combo_box.addItem('Option 1')
```



```

combo_box.addItem('Option 2')
combo_box.addItem('Option 3')

# 获取当前选择的项
current_item = combo_box.currentText()
print(f'Current Item: {current_item}')

# 移除指定位置的项
combo_box.removeItem(1)

# 插入新项到指定位置
combo_box.insertItem(1, 'New Option')

# 编辑指定位置的项
combo_box.setItemText(0, 'Updated Option')

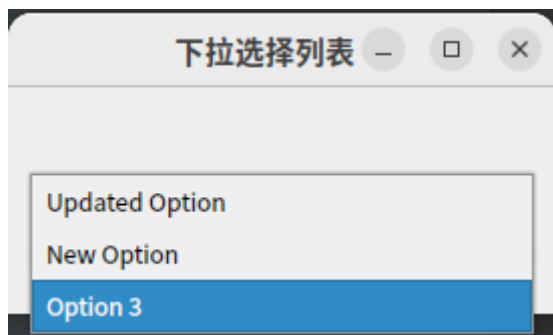
# 将QComboBox添加到布局
layout.addWidget(combo_box)

# 设置布局
self.setLayout(layout)

self.setWindowTitle('下拉选择列表')
self.show()

if __name__ == '__main__':
    app = QApplication([])
    window = ComboBoxApp()
    app.exec_()

```



在这个示例中，我们创建了一个包含三个选项的QComboBox，并演示了添加、移除、编辑项目的操作。通过currentText()方法，我们还可以获取当前选择的项。

### 可编辑项:

QComboBox还支持可编辑的模式，允许用户输入自定义内容。要启用可编辑模式，可以使用setEditable(True)方法。

```

# 启用可编辑模式
combo_box.setEditable(True)

```

启用可编辑模式后，用户可以在下拉框中输入文本，而不仅仅限于选择预定义的项目。

QComboBox是一个灵活且常用的控件，适用于许多用户界面场景。通过使用QComboBox，用户可以方便地从预定义的选项中进行选择或输入自定义内容。

7.5 QSlider & QDial - 滑动选择器和旋钮

QSlider和QDial是PyQt5中用于实现滑动选择和旋钮功能的控件。以下是这两个控件的使用方法和定制示例。

QSlider - 滑动选择器

QSlider用于通过滑块选择一个数值范围。

PyQt5中提供了2种滑块控件，分别是：水平滑块（HorizontalSlider）、垂直滑块（VerticalSlider）。

它们对应的类都是QSlider类，这个类中有1个 setOrientation() 方法，通过设置该方法的参数，就可以将滑块显示为水平或者垂直。

QSlider 滑块类的常用方法、常用信号

信号	说明
valueChanged	当滑块的值发生变化时发射
sliderPressed	当用户按下滑块时发射
sliderMoved	当用户拖动滑块时发射
sliderReleased	当用户释放滑块时发射

滑块只能控制整数范围，因此它不适用于需要准确的大范围取值的场景

以下是QSlider的基本用法和一些定制操作。

```
from PyQt5.QtWidgets import QApplication, QWidget, QSlider, QVBoxLayout
from PyQt5.QtCore import Qt

class SliderApp(QWidget):
    def __init__(self):
        super().__init__()

        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        # 创建QSlider
        slider = QSlider(Qt.Horizontal)

        # 设置范围
        slider.setMinimum(0)
        slider.setMaximum(100)

        # 设置初始值
        slider.setValue(50)

        # 设置刻度位置和间隔
        slider.setTickPosition(QSlider.TicksBelow)
        slider.setTickInterval(10)
```

```

# 连接值变化的信号到槽函数
slider.valueChanged.connect(self.on_slider_change)

# 将QSlider添加到布局
layout.addWidget(slider)

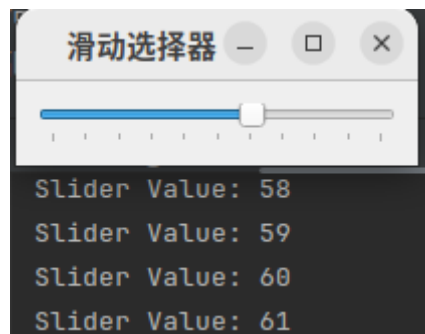
# 设置布局
self.setLayout(layout)

self.setWindowTitle('滑动选择器')
self.show()

def on_slider_change(self, value):
    print(f'Slider Value: {value}')

if __name__ == '__main__':
    app = QApplication([])
    window = SliderApp()
    app.exec_()

```



在这个示例中，我们创建了一个水平方向的QSlider，设置了范围、初始值和刻度。通过连接valueChanged信号到槽函数，我们可以捕获滑块值的变化。

要定制QSlider和QDial的样式，可以使用样式表（StyleSheet）。以下是一个简单的样式表示例，将滑块的背景颜色和滑块的样式进行了定制。

```

slider.setStyleSheet("QSlider::groove:horizontal { background: #b2bec3; height: 10px; } "
                    "QSlider::handle:horizontal { background: #d63031; width: 20px; margin: "
                    "-5px 0; }")

```



### 示例：创意的表情滑块

```

QSlider{
    background-color: transparent;
}

QSlider::add-page:horizontal {
    background: qlineargradient(x1:0, y1:0, x2:1, y2:0, stop:0 #ffffff, stop:1 #ab7bff);
}

```

```

QSlider::sub-page:horizontal {
background: qlineargradient(x1:0, y1:0, x2:1, y2:0, stop:0 #ff6699, stop:1 #ffffff);
}

QSlider::groove:horizontal{
height: 10px;
background-color: transparent;
}

QSlider::handle:horizontal {
image: url(/icon/atxmr-bwdq6.png);
background-color: transparent;
margin: -20px -20px -20px -20px;
}

QSlider::handle:horizontal:hover {
image: url(/icon/a8tyb-42rrh.png);
}

QSlider::handle:horizontal:pressed {
image: url(/icon/arv9v-8m1je.png);
}

```

## QDial - 旋钮

QDial用于通过旋转选择一个数值范围。以下是QDial的基本用法和一些定制操作。

```

from PyQt5.QtWidgets import QApplication, QWidget, QDial, QVBoxLayout
from PyQt5.QtCore import Qt

class DialApp(QWidget):
    def __init__(self):
        super().__init__()

        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        # 创建QDial
        dial = QDial()

        # 设置范围
        dial.setMinimum(0)
        dial.setMaximum(100)

        # 设置初始值
        dial.setValue(50)

        # 连接值变化的信号到槽函数
        dial.valueChanged.connect(self.on_dial_change)

        # 将QDial添加到布局
        layout.addWidget(dial)

        # 设置布局
        self.setLayout(layout)

```

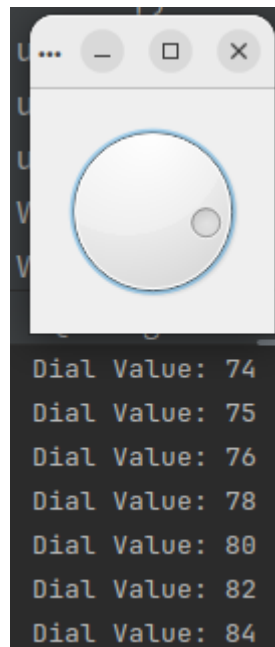
```

        self.setWindowTitle('旋钮')
        self.show()

    def on_dial_change(self, value):
        print(f'Dial Value: {value}')

if __name__ == '__main__':
    app = QApplication([])
    window = DialApp()
    app.exec_()

```



在这个示例中，我们创建了一个QDial，同样设置了范围和初始值，并连接了valueChanged信号。用户可以通过旋转旋钮来选择数值。

## 7.6 QDateTimeEdit、QDateEdit、QTimeEdit - 日期和时间输入

QDateTimeEdit、QDateEdit和QTimeEdit是PyQt5中用于实现日期和时间输入的控件。以下是这些控件的功能、格式化选项以及如何实现日期和时间选择的示例。

### QDateTimeEdit - 日期和时间选择

QDateTimeEdit允许用户选择日期和时间。以下是一个简单的示例：

```

from PyQt5.QtCore import QDateTime
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QDateTimeEdit

class DateTimeEditApp(QWidget):
    def __init__(self):
        super().__init__()

        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        # 创建QDateTimeEdit

```

```

date_time_edit = QDateTimeEdit(self)

# 设置日期和时间范围
date_time_edit.setMinimumDateTime(QDateTime.currentDateTime().addDays(-365))
date_time_edit.setMaximumDateTime(QDateTime.currentDateTime().addDays(365))

# 设置显示格式
date_time_edit.setDisplayFormat('yyyy-MM-dd HH:mm:ss')

# 连接日期时间变化的信号到槽函数
date_time_edit.dateTimeChanged.connect(self.on_date_time_changed)

# 将QDateTimeEdit添加到布局
layout.addWidget(date_time_edit)

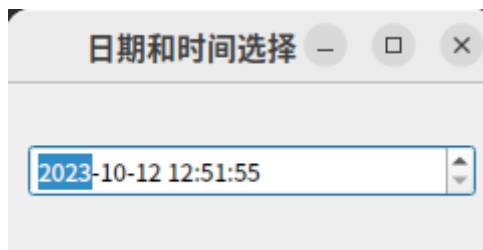
# 设置布局
self.setLayout(layout)

self.setWindowTitle('日期和时间选择')
self.show()

def on_date_time_changed(self, datetime):
    print(f'Selected Date and Time: {datetime.toString()}')

if __name__ == '__main__':
    app = QApplication([])
    window = DateTimeEditApp()
    app.exec_()

```



在这个示例中，我们创建了一个 `QDateTimeEdit`，并设置了日期和时间的范围以及显示格式。通过连接 `dateTimeChanged` 信号到槽函数，我们可以捕获日期和时间的变化。

### `QDateEdit` - 日期选择

`QDateEdit` 用于选择日期。以下是一个简单的示例：

```

from PyQt5.QtCore import QDate
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QDateEdit

class DateEditApp(QWidget):
    def __init__(self):
        super().__init__()

        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        # 创建QDateEdit

```

```

date_edit = QDateEdit(self)

# 设置日期范围
date_edit.setMinimumDate(QDate.currentDate().addDays(-365))
date_edit.setMaximumDate(QDate.currentDate().addDays(365))

# 设置显示格式
date_edit.setDisplayFormat('yyyy-MM-dd')

# 连接日期变化的信号到槽函数
date_edit.dateChanged.connect(self.on_date_changed)

# 将QDateEdit添加到布局
layout.addWidget(date_edit)

# 设置布局
self.setLayout(layout)

self.setWindowTitle('日期选择')
self.show()

def on_date_changed(self, date):
    print(f'Selected Date: {date.toString()}')

if __name__ == '__main__':
    app = QApplication([])
    window = DateEditApp()
    app.exec_()

```



在这个示例中，我们创建了一个QDateEdit，并设置了日期的范围以及显示格式。通过连接dateChanged信号到槽函数，我们可以捕获日期的变化。

## QTimeEdit - 时间选择

QTimeEdit用于选择时间。以下是一个简单的示例：

```

from PyQt5.QtCore import QTime
from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QTimeEdit

class TimeEditApp(QWidget):
    def __init__(self):
        super().__init__()

        self.init_ui()

    def init_ui(self):

```

```

layout = QVBoxLayout()

# 创建QTimeEdit
time_edit = QTimeEdit(self)

# 设置时间范围
time_edit.setMinimumTime(QTime.currentTime().addSecs(-3600))
time_edit.setMaximumTime(QTime.currentTime().addSecs(3600))

# 设置显示格式
time_edit.setDisplayFormat('hh:mm:ss')

# 连接时间变化的信号到槽函数
time_edit.timeChanged.connect(self.on_time_changed)

# 将QTimeEdit添加到布局
layout.addWidget(time_edit)

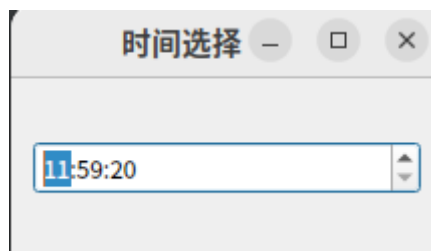
# 设置布局
self.setLayout(layout)

self.setWindowTitle('时间选择')
self.show()

def on_time_changed(self, time):
    print(f'Selected Time: {time.toString()}')

if __name__ == '__main__':
    app = QApplication([])
    window = TimeEditApp()
    app.exec_()

```



## 7.7 QProgressBar - 进度条展示

QProgressBar用于在PyQt5应用程序中展示任务进度。以下是关于QProgressBar的用途、自定义选项以及如何实现和更新进度条的示例。

```

from PyQt5.QtWidgets import QApplication, QWidget, QVBoxLayout, QPushButton, QProgressBar
from PyQt5.QtCore import QThread, pyqtSignal, QTimer

class WorkerThread(QThread):
    # 通过信号更新进度条
    progress_updated = pyqtSignal(int)

    def run(self):
        for i in range(101):
            # 模拟耗时任务
            self.msleep(50)
            # 发送进度信号

```



```
        self.progress_updated.emit(i)

class ProgressBarApp(QWidget):
    def __init__(self):
        super().__init__()

        self.init_ui()

    def init_ui(self):
        layout = QVBoxLayout()

        # 创建进度条
        self.progress_bar = QProgressBar(self)
        self.progress_bar.setMinimum(0)
        self.progress_bar.setMaximum(100)

        # 创建开始任务按钮
        start_button = QPushButton('开始任务', self)
        start_button.clicked.connect(self.start_task)

        # 将进度条和按钮添加到布局
        layout.addWidget(self.progress_bar)
        layout.addWidget(start_button)

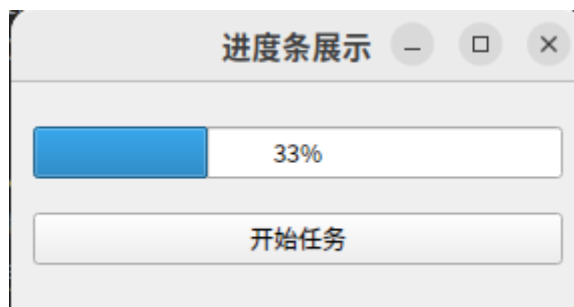
        # 设置布局
        self.setLayout(layout)

        self.setWindowTitle('进度条展示')
        self.show()

    def start_task(self):
        # 创建并启动工作线程
        self.worker_thread = WorkerThread(self)
        self.worker_thread.progress_updated.connect(self.update_progress)
        self.worker_thread.start()

    def update_progress(self, value):
        # 更新进度条的值
        self.progress_bar.setValue(value)

if __name__ == '__main__':
    app = QApplication([])
    window = ProgressBarApp()
    app.exec_()
```



在这个示例中，我们创建了一个简单的PyQt5应用程序，包括一个QProgressBar和一个按钮。当用户点击按钮时，会启动一个后台工作线程（WorkerThread），该线程模拟一个耗时任务，每完成一部分任务就发射一个信号（progress\_updated），通过这个信号更新主线程中的进度条。这样可以确保在执行耗时任务的同时，用户界面保持响应。

### 7.8 QLabel-标签

标签控件主要用于显示用户不能编辑的文本，标识窗体上的对象（例如：给文本框、列表框添加，描述信息等）。它对应PyQt5中的QLabel类，Label控件在本质上是QLabel类的一个对象。

#### 设置标签文本

有2种方法可以设置标签控件显示的文本：

方法1，直接在Qt Designer设计器的属性编辑器中设置它的text属性。如下图所示：

▼ QLabel	
> text	hello world
textFormat	AutoText
pixmap	
scaledContents	<input type="checkbox"/>
> alignment	AlignLeft, AlignVCenter
wordWrap	<input type="checkbox"/>
margin	0
indent	-1
openExternalLinks	<input type="checkbox"/>
> textInteractionFlags	LinksAccessibleByMouse
buddy	

方法2，通过Python代码进行设置：

```
# 通过setText()方法将标签控件的显示内容设为“hello world”
self.label.setText('hello world')
```

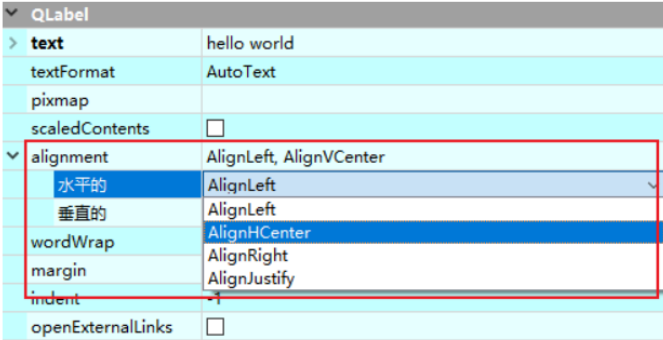
#### 设置标签控件内文本的对齐方式

PyQt5中支持设置标签中文本的对齐方式，主要用到alignment属性。在Qt Designer设计器的属性编辑器中展开alignment属性，可以看到有2个值，分别为Horizontal（水平的）和Vertical（垂直的）。

其中，Horizontal表示水平对齐，取值有4个，如下表所示：

取值	说明
AlignLeft	左对齐
AlignHCenter	水平居中对齐
AlignRight	右对齐
AlignJustify	两端对齐（效果同AlignLeft）

在Qt Designer设计器中的位置如下图所示：

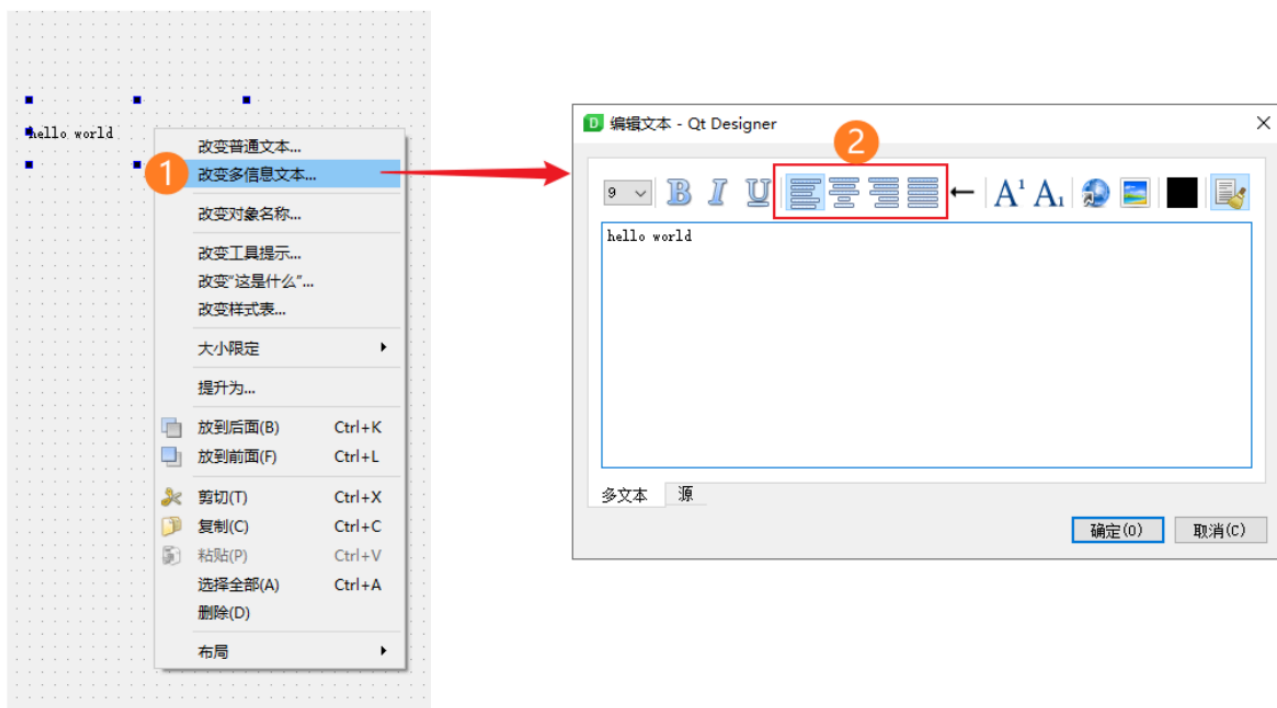


不同取值的效果如下图所示：



以上是在属性编辑器中直接设置其属性的方法，除此之外，还有2种方法：

方法2：我们单击窗口中的Label控件，然后再单击右键菜单中的“改变多信息文本”，弹出一个编辑文本的窗口，在这个窗口的顶栏中有4种对齐方式可以设置，从左到右分别是：左对齐，中央对齐，右对齐，自动调整。



方法3: Python代码设置, 写法都是一样的, 只需要修改末尾最后一个单词即可, 具体示例代码如下:

```
# AlignLeft: 左对齐
self.label.setAlignment(QtCore.Qt.AlignLeft)

# AlignHCenter: 水平居中对齐
self.label.setAlignment(QtCore.Qt.AlignHCenter)

# AlignRight: 右对齐
self.label.setAlignment(QtCore.Qt.AlignRight)

# AlignJustify: 两端对齐 (效果同AlignLeft)
self.label.setAlignment(QtCore.Qt.AlignJustify)
```

代码运行效果如下图所示 (为方便辨识, 我将标签控件的背景色改成了粉色):



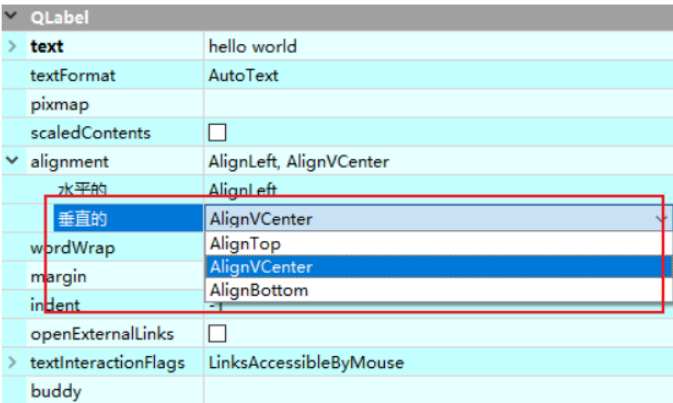
可以看到, 它们都是在顶部左对齐、水平居中对齐、右对齐, 而且都是贴边的, 这样很不美观, 也不是我们想要的结果。

所以就需结合Vertical了, Vertical是用来设置标签文本的垂直对齐方式, 取值有3个。如下表:

取值	说明
AlignTop	顶部对齐
AlignVCenter	垂直居中对齐
AlignBottom	底部对齐

在Qt Designer设计器中的位置如下图所示:

在Qt Designer设计器中的位置如下图所示：



不同取值的效果如下图所示：



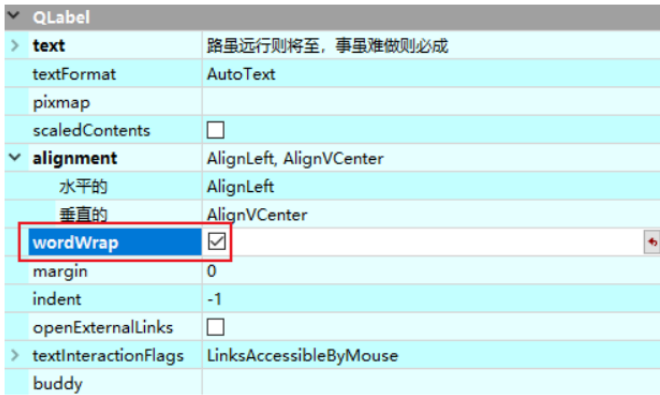
可以看到，Vertical是上中下变化，而Horizontal是左中右变化。所以只要将它们2个结合起来，就能实现真正的居中。示例代码如下：

```
# 水平垂直居中
self.label.setAlignment(QtCore.Qt.AlignHCenter | QtCore.Qt.AlignVCenter)
```

文本换行显示

假设将标签文本改为“路虽远行则将至，事虽难做则必成”，在标签宽度不足的情况下，系统会默认只显示前面部分文字。遇到这种情况时，我们就需要用到换行显示了。

换行显示是标签控件的属性之一，在Qt Designer设计器的属性编辑器中，将wordWrap属性后面的复选框选中即可实现换行显示。如下图所示：



```
# 通过QLabel类的setWordWrap()方法将文本设为换行显示
self.label.setWordWrap(True)
```

## 为标签设置超链接

为Label标签设置超链接时，可以直接在QLabel类的setText()方法中使用HTML中的[标签设置超链接文本](#)，然后将Label标签中的setOpenExternalLinks()方法设置为True，以便允许访问超链接，示例代码如下：

```
# 设置跳转链接和超链接文本（注意双引号和单引号）
self.label.setText("<a href = 'https://www.baidu.com/'> 百度一下，你就知道 </a>")

# 启用跳转
self.label.setOpenExternalLinks(True)
```

代码运行后，标签内的文本会变成蓝色，并且带有下划线。当你单击文本后，则会立即跳转到目标网址。

## 为标签设置图片

为Label标签设置图片时，需要使用QLabel类的setPixmap()方法，该方法中需要有一个QPixmap对象，表示图标对象。示例代码如下：

```
# 导入QPixmap类
from PyQt5.QtGui import QPixmap

# 通过setPixmap()方法将名为abc的图片在标签内显示出来
self.label.setPixmap(QPixmap('abc.png'))
```

在PyQt5中暂不支持该方法显示jpg格式的图片，你可以通过其它工具先将格式转换成png，然后再显示

## 为标签设置气泡提示信息

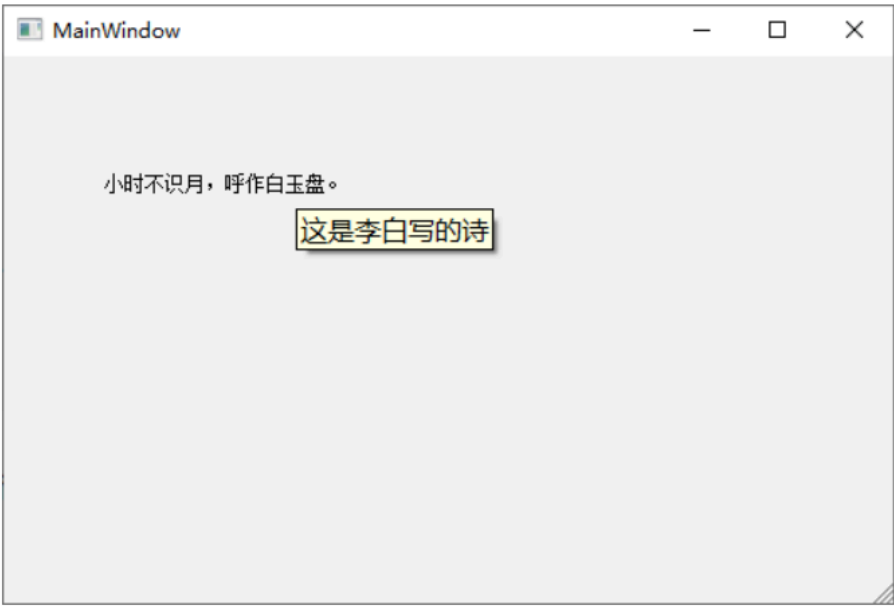
气泡提示就是当你把光标移动到某个控件上时，会弹出一个提示框。光标移开时，提示框又会自动消失，多数用于解释说明。**不仅局限于标签控件**

```
# 使用该方法需要导入2个模块：QToolTip和QFont
from PyQt5.QtWidgets import QToolTip
from PyQt5.QtGui import QFont

# 设置气泡提示内容为“这是李白写的诗”
self.label.setToolTip('这是李白写的诗')

# 设置提示文本的字体为微软雅黑，字号为12
QToolTip.setFont(QFont('微软雅黑', 12))
```

代码运行效果如下图所示：



7.9 QLCDNumber：液晶数字显示控件

QLCDNumber控件对应PyQt5中的QLCDNumber类，该类的常用方法及说明如下表：

方法	说明
setDigitCount()	设置可以显示的数字数量
setProperty()	设置相应属性的值
setMode()	设置显示数字的模式，有4种模式： 1. Bin：二进制 2. Oct：八进制 3.Dec：十进制 4. Hex：十六进制
setSegmentStyle()	设置显示样式，有3种样式： OutLine，Filled，Flat
value()	获取显示的数值

实例：设置液晶显示屏中的数字显示

使用Qt Designer创建一个MainWindow窗口，其中添加1个Label、1个LineEdit、1个QLCDNumber。其中，QLCDNumber用来显示LineEdit中输入的数字。

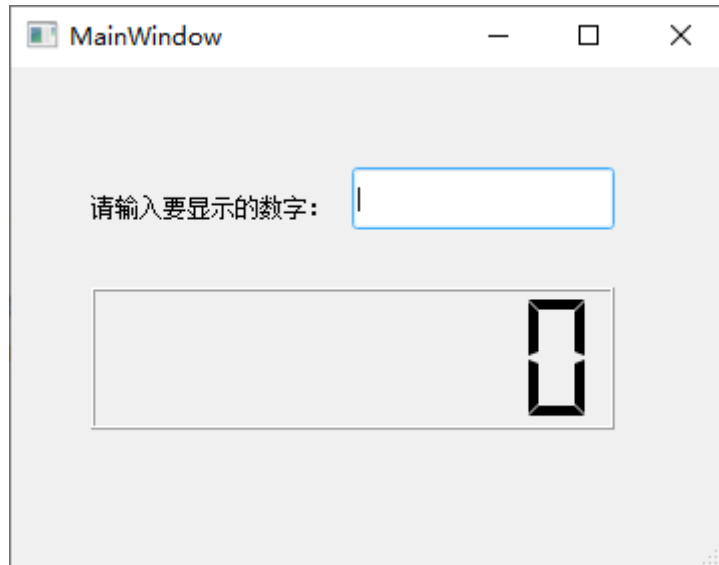
将设计完成的窗口保存为.ui文件，使用PyUIC工具将其转换成.py文件。在转化后的.py文件中，设置QLCDNumber的最大显示数字位数、显示样式及模式。代码如下：

```
# 设置最大显示7位数字
self.lcdNumber.setDigitCount(7)

# 设置以十进制显示数字
self.lcdNumber.setMode(QtWidgets.QLCDNumber.Dec)

# 设置数字显示屏的显示样式
self.lcdNumber.setSegmentStyle(QtWidgets.QLCDNumber.Flat)
```

运行看下效果：



接下来，自定义一个set\_value()函数，使用setProperty()方法为LCDNumber设置要显示的数字为LineEdit中输入的数字，代码如下：

```
def set_value(self):
    self.lcdNumber.setProperty('value', self.lineEdit.text())
```

再将自定义函数set\_value()和LineEdit的editingFinished信号关联起来。这个信号的作用是：当你输入完成后按键该信号就会发射。代码如下：

```
self.lineEdit.editingFinished.connect(self.set_value)
```

**关联代码要写在自定义函数之前**

以上代码运行后，我们在LineEdit中输入5201314，然后按键。效果如下图所示：





我们在前面的代码中限制了LCDNumber最大显示7位数字，如果超过7位，那么它会以科学计数法的形式进行显示。如下图所示：



## 8. 动画和图形效果

### 8.1 基础动画概念

动画是通过连续播放一系列静态图像来创造运动或变化的错觉。要理解和实现动画，需要掌握几个基本概念：

- 1.帧(Frame)：动画由多个帧组成，每帧是一个静态的图像。在连续快速播放时，这些静态图像产生运动的错觉。
- 2.时间轴(Timeline)：这是动画发生的时间框架。它定义了动画的开始、持续时间以及结束。在时间轴上，我们可以设定关键帧（特定的时间点），用来标记动画状态的改变。
- 3.缓动(Easing)：缓动是动画的速度变化。不是所有动画都以恒定速度播放。缓动可以让动画开始慢、然后加速，或者开始快后减速，这样可以创造更自然、更吸引人的动画效果。

#### 在PyQt5中创建和控制基本动画

PyQt5提供了一套完整的动画框架，可以轻松地创建和控制动画。以下是在PyQt5中实现基本动画的步骤：

 Important

1.**定义动画对象**：使用QPropertyAnimation类，您可以创建一个动画对象。这个类允许您选择要动画化的属性（例如，控件的位置、大小或颜色）。

## 2.设置动画属性：

- 目标对象：设置动画将作用的QWidget或其它对象。
- 属性名：设置您想要动画化的属性名，如"geometry"。
- 持续时间：定义动画从开始到结束所需的时间（以毫秒为单位）。
- 起始值和结束值：设置属性的起始值和结束值。这些值取决于动画的类型和您想要创建的效果

3.**应用缓动效果**：可以通过设置easing curve来控制动画的速度变化。PyQt5提供了多种预定义的缓动效果。

4.**启动动画**：通过调用动画对象的start()方法来开始动画。

例如，要创建一个简单的移动动画，您可以定义一个QPropertyAnimation对象，指定要移动的控件和位置属性，并设置动画的持续时间及起始和结束位置。通过调整这些参数，您可以创造出平滑且吸引人的动画效果。

## 8.2 QGraphicsView和QGraphicsItem

在PyQt5中，QGraphicsView和QGraphicsItem是创建复杂图形界面和动画的核心组件。它们是Qt的图形视图框架的一部分，提供了一个强大的场景-视图架构来管理和显示自定义的2D图形项。

### QGraphicsView

- **作用**：QGraphicsView是一个显示控件，用于展示QGraphicsScene中的内容。它相当于一个查看窗口，通过它可以观察场景中的图形项（QGraphicsItem）。
- **特点**：支持缩放和旋转视图、可以处理大量的自定义图形项、支持图形项的交互操作（如拖动和选择）。

### QGraphicsItem

- **作用**：QGraphicsItem代表场景中的每一个图形元素。它可以是任何形状（如矩形、圆形、文本甚至自定义的复杂形状）。
- **特点**：可以单独对每个项设置动画和变换（如移动、缩放、旋转），支持事件处理（如鼠标点击），可以组合多个项来创建复杂的图形结构

### 示例代码：创建和管理图形元素

```
from PyQt5.QtWidgets import QApplication, QGraphicsView, QGraphicsScene, QGraphicsItem
from PyQt5.QtGui import QBrush, QPen
from PyQt5.QtCore import QRectF
import sys

class ExampleItem(QGraphicsItem):
    def boundingRect(self):
        # 定义图形项的边界框
        return QRectF(0, 0, 100, 100)

    def paint(self, painter, option, widget):
        # 绘制一个简单的矩形
        painter.setPen(QPen())
        painter.setBrush(QBrush())
        painter.drawRect(0, 0, 100, 100)

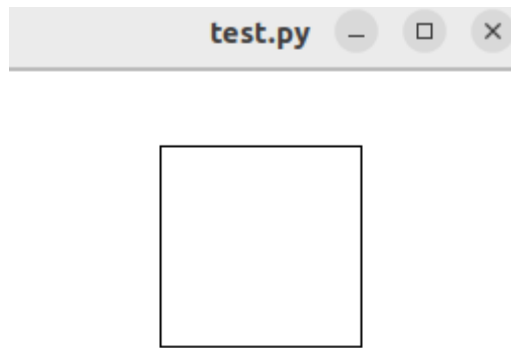
app = QApplication(sys.argv)

# 创建一个场景
scene = QGraphicsScene()
```

```
# 创建一个图形项并添加到场景中
item = ExampleItem()
scene.addItem(item)

# 创建一个视图，并将场景设置给它
view = QGraphicsView(scene)
view.show()

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



这段代码创建了一个简单的PyQt5应用程序，其中包括一个QGraphicsView来展示一个场景，以及一个自定义的QGraphicsItem。在这个例子中，我们定义了一个名为ExampleItem的类，它继承自QGraphicsItem并实现了boundingRect和paint方法来定义图形项的外观。然后我们将这个自定义项添加到场景中，并通过视图来展示这个场景。这只是一个简单的起点，但它展示了如何在PyQt5中使用图形视图框架来创建和管理图形元素。

### 8.3 属性动画

属性动画是指在一定时间内逐渐改变对象的属性（如位置、大小、颜色等）的动画。这种类型的动画通过平滑地过渡对象属性的值，创造出流畅且吸引人的视觉效果。在PyQt5中，属性动画可以通过QPropertyAnimation类来实现。

属性动画的关键概念：

- 属性(Property)：可以动画化的对象属性，例如，窗口控件的geometry、opacity或自定义属性。
- 持续时间(Duration)：动画从开始到结束所需的时间。
- 起始值和结束值：动画属性的初始值和最终值。
- 缓动曲线(Easing Curve)：控制动画速度变化的曲线，可以使动画更自然。

#### 示例代码：实现属性动画

下面的代码示例展示了如何使用PyQt5中的QPropertyAnimation来实现一个简单的属性动画，比如改变一个窗口控件的位置。

```
from PyQt5.QtWidgets import QApplication, QPushButton
from PyQt5.QtCore import QPropertyAnimation
from PyQt5.QtCore import QRect
import sys

app = QApplication(sys.argv)

# 创建一个按钮
button = QPushButton("Click Me", None)
button.show()
```

```
# 创建一个属性动画对象
anim = QPropertyAnimation(button, b"geometry")
anim.setDuration(1000) # 持续时间为1000毫秒

# 设置动画的起始值和结束值
anim.setStartValue(QRect(0, 0, 100, 30)) # 起始位置和大小
anim.setEndValue(QRect(250, 250, 100, 30)) # 结束位置和大小

# 启动动画
anim.start()

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

在这个例子中，我们创建了一个QPushButton控件，并定义了一个QPropertyAnimation对象来改变这个按钮的geometry属性。动画将在1000毫秒内将按钮从屏幕的一个位置平滑移动到另一个位置。通过调整动画的持续时间、起始值和结束值，可以创造出各种不同的过渡效果。

## 8.4 QAnimation类的使用

在PyQt5中，QAnimation类是一个虚拟基类，提供了创建和管理动画的框架。实际上，通常使用的是它的子类，如QPropertyAnimation和QSequentialAnimationGroup，来实现具体的动画效果。这些类使得动画的创建和管理变得更加简单和直观。

### 📌 Important

#### QPropertyAnimation和QSequentialAnimationGroup的区别

“QPropertyAnimation”和“QSequentialAnimationGroup”是Qt中用于创建动画的两个不同的类，它们在动画的创建和控制方面有着各自的特点和用途：

##### 1. QPropertyAnimation:

- “QPropertyAnimation”是用于动画化对象属性的类。它允许你对“QObject”或其派生类的任何可读写属性进行动画处理。这意味着，你可以创建平滑、连续的动画效果，比如控件的位置、大小、颜色等属性的变化。
- “QPropertyAnimation”继承自“QVariantAnimation”，因此它支持动画化任何可以表示为“QVariant”的属性。
- 使用“QPropertyAnimation”时，你可以直接对目标对象的属性进行动画设置，而不需要关心动画的顺序或组合。
- “QPropertyAnimation”可以设置开始值、结束值、持续时间、缓动曲线等，以实现丰富的动画效果。

##### 2. QSequentialAnimationGroup:

- “QSequentialAnimationGroup”是一个动画组，它用来管理多个动画对象，使这些动画按照一定的顺序逐个执行。也就是说，在一个动画完成后，下一个动画才会开始。
- 这个类是“QAnimationGroup”的一部分，它允许你将多个动画组合在一起，按照顺序播放，从而创建复杂的动画序列。
- “QSequentialAnimationGroup”可以添加暂停，通过“addPause()”或“insertPause()”方法在动画序列中插入暂停。
- 你可以将“QPropertyAnimation”或其他“QAbstractAnimation”对象添加到“QSequentialAnimationGroup”中，以控制动画的顺序执行。

总结来说，“QPropertyAnimation”专注于对单个对象的属性进行动画化，而“QSequentialAnimationGroup”用于管理和顺序执行多个动画。如果你需要创建一个涉及多个对象或多个属性变化的复杂动画序列，你可能需要将“QPropertyAnimation”对象添加到“QSequentialAnimationGroup”中来实现。

## QAnimation的基本用法

以下是QPropertyAnimation的基本用法，作为QAnimation类的一个常用实现示例：

1. **创建动画对象**：选择要动画化的属性和目标对象。
2. **设置动画的持续时间**：定义动画从开始到结束所需的时间。
3. **定义起始值和结束值**：设置属性的初始值和最终值。
4. **启动动画**：调用动画对象的start()方法来开始动画。

### 示例代码：基本动画用法

```
from PyQt5.QtCore import QPropertyAnimation, QObject

# 假设有一个名为widget的QWidget对象
# 创建动画对象，指定动画的属性和目标对象
anim = QPropertyAnimation(widget, b"opacity")

# 设置动画持续时间为1000毫秒
anim.setDuration(1000)

# 设置动画的起始值和结束值
anim.setStartValue(0) # 开始时不透明度为0
anim.setEndValue(1)   # 结束时不透明度为1

# 启动动画
anim.start()
```

### 讨论不同类型的动画

- **平移动画**：通过改变控件的位置（通常是geometry或pos属性）来实现。
- **旋转动画**：如果是自定义的图形项，可以通过改变其旋转属性来实现。
- **缩放动画**：类似于旋转动画，可以通过改变控件或图形项的缩放属性来实现。

在PyQt5中，您可以通过结合这些不同类型的动画，创造出丰富且动态的用户界面效果。例如，您可以同时改变一个控件的位置、大小和颜色，以创建一个复杂的动画效果。通过使用QSequentialAnimationGroup和QParallelAnimationGroup，您甚至可以组合多个动画，以顺序或并行的方式播放。

## 8.5 动画控制和交互

在PyQt5中，动画的控制和用户交互是通过动画类的方法和信号来实现的。这些功能允许动画根据用户的输入或程序的需求进行灵活的控制

### 动画播放控制

- **播放**：通过调用start()方法开始动画。
- **暂停**：使用pause()方法可以暂停动画。这对于长时间运行的动画特别有用，用户可以在需要时暂停动画。
- **恢复**：暂停后，可以通过resume()方法恢复动画。
- **停止**：stop()方法会停止动画。停止动画后，可以重新调整参数后再次启动。
- **循环播放**：设置loopCount属性可以使动画循环播放指定次数，或者无限循环。

### 示例代码：动画控制

```
from PyQt5.QtCore import QPropertyAnimation, QObject
```

```
# 创建一个QPropertyAnimation对象
anim = QPropertyAnimation(widget, b"geometry")

# 设置动画参数
anim.setDuration(2000)
anim.setStartValue(widget.geometry())
anim.setEndValue(target_geometry)

# 启动动画
anim.start()

# 在需要时，可以调用以下方法来控制动画
anim.pause()    # 暂停动画
anim.resume()   # 恢复动画
anim.stop()     # 停止动画
```

## 用户交互的动画

用户交互的动画是指根据用户的操作来控制动画进程的动画。例如，通过拖动滑块来控制动画的进度。

```
from PyQt5.QtWidgets import QSlider, QApplication, QWidget, QVBoxLayout
from PyQt5.QtCore import QPropertyAnimation, QRect
import sys

class AnimationWidget(QWidget):
    def __init__(self):
        super().__init__()

        self.slider = QSlider(self)
        self.slider.setOrientation(Qt.Horizontal)
        self.slider.valueChanged.connect(self.update_animation)

        self.anim = QPropertyAnimation(self, b"geometry")
        self.anim.setDuration(1000)
        self.anim.setStartValue(QRect(0, 0, 100, 30))
        self.anim.setEndValue(QRect(250, 250, 100, 30))

        layout = QVBoxLayout(self)
        layout.addWidget(self.slider)

    def update_animation(self, value):
        # 根据滑块位置更新动画进度
        progress = value / self.slider.maximum()
        self.anim.setCurrentTime(self.anim.duration() * progress)

app = QApplication(sys.argv)
widget = AnimationWidget()
widget.show()
sys.exit(app.exec_())
```

在这个示例中，创建了一个包含滑块和动画的小部件。滑块的值改变时会调用`update_animation`函数，该函数根据滑块的位置来更新动画的当前时间，从而控制动画的进度。这样的交互方式使用户能够直观地控制动画，增强了应用程序的交互性。

## 自定义缓动函数

自定义缓动函数允许开发者创造独特的动画效果。这通常涉及到创建一个自定义的QEasingCurve对象，并设置一个自定义的缓动函数。

### 示例代码：自定义过渡效果

```
from PyQt5.QtCore import QPropertyAnimation, QEasingCurve, QRect

def custom_easing_curve(t):
    # 自定义一个缓动函数
    return t * t

# 创建动画对象
anim = QPropertyAnimation(widget, b"geometry")

# 设置动画参数
anim.setDuration(2000)
anim.setStartValue(QRect(0, 0, 100, 30))
anim.setEndValue(QRect(250, 250, 100, 30))

# 创建自定义的缓动曲线
curve = QEasingCurve()
curve.setCustomType(custom_easing_curve)
anim.setEasingCurve(curve)

# 启动动画
anim.start()
```

在这个示例中，我们定义了一个简单的自定义缓动函数`custom_easing_curve`，它返回的值随时间的平方变化。然后将这个函数应用到QEasingCurve上，并设置给动画对象。通过这种方式，您可以创造出任意复杂的动画节奏，以符合应用程序的具体需求。

## 8.6 缓动函数和自定义过渡效果

缓动函数在动画中扮演着至关重要的角色，它们定义了动画属性值随时间变化的方式，从而影响动画的感觉和流畅度。PyQt5提供了多种预定义的缓动函数，同时也支持自定义缓动函数。

### 缓动函数的作用

- **控制动画节奏**：缓动函数确定动画的加速和减速方式，使动画看起来更自然。
- **增强视觉效果**：不同的缓动函数可以创造出不同的动画效果，例如平滑的过渡、弹跳效果等。

### 使用预定义缓动函数

PyQt5内置了多种缓动函数，如QEasingCurve.Linear、QEasingCurve.InBounce、QEasingCurve.OutElastic等，可以直接应用于动画对象。

 Important

#### QEasingCurve.Linear:

线性缓动曲线（QEasingCurve::Linear）表示动画以恒定的速度进行，从开始到结束速度不变。这意味着动画的进度是均匀的，没有加速或减速的过程。

这种缓动曲线适用于需要平滑且可预测动画效果的场景，例如，简单地移动或缩放对象。

#### QEasingCurve.InBounce:

InBounce 是一种“弹性”效果的缓动曲线，它模拟了物体受到弹性力作用时的运动。在动画的开始阶段，动画对象会快速移动，然后在接近结束值时出现几次快速的“弹跳”效果，好像它在尝试停靠但被弹回。

这种缓动曲线常用于创建有趣且引人注目的动画效果，例如，按钮按下后的反弹效果。

#### QEasingCurve.OutElastic:

OutElastic 也是一种“弹性”效果的缓动曲线，但它与 InBounce 相反。在动画的大部分时间内，对象似乎以正常速度移动，但在动画结束前，它会突然“弹回”并超出结束值，然后才缓慢地回到最终位置。

这种缓动曲线可以用于创建一种“释放”的效果，例如，当一个对象被快速抛出并最终停下来时。

### 示例代码：使用预定义缓动函数

```
from PyQt5.QtCore import QPropertyAnimation, QEasingCurve, QRect

# 创建动画对象
anim = QPropertyAnimation(widget, b"geometry")

# 设置动画参数
anim.setDuration(2000)
anim.setStartValue(QRect(0, 0, 100, 30))
anim.setEndValue(QRect(250, 250, 100, 30))

# 应用预定义的缓动函数
anim.setEasingCurve(QEasingCurve.OutBounce)

# 启动动画
anim.start()
```

### 自定义缓动函数

自定义缓动函数允许开发者创造独特的动画效果。这通常涉及到创建一个自定义的QEasingCurve对象，并设置一个自定义的缓动函数。

### 示例代码：自定义过渡效果

```
from PyQt5.QtCore import QPropertyAnimation, QEasingCurve, QRect

def custom_easing_curve(t):
    # 自定义一个缓动函数
    return t * t

# 创建动画对象
anim = QPropertyAnimation(widget, b"geometry")

# 设置动画参数
anim.setDuration(2000)
anim.setStartValue(QRect(0, 0, 100, 30))
anim.setEndValue(QRect(250, 250, 100, 30))

# 创建自定义的缓动曲线
```



```
curve = QEasingCurve()
curve.setCustomType(custom_easing_curve)
anim.setEasingCurve(curve)

# 启动动画
anim.start()
```

在这个示例中，我们定义了一个简单的自定义缓动函数`custom_easing_curve`，它返回的值随时间的平方变化。然后将这个函数应用到`QEasingCurve`上，并设置给动画对象。通过这种方式，您可以创造出任意复杂的动画节奏，以符合应用程序的具体需求。

## 8.7 图形效果和滤镜

PyQt5不仅提供了强大的动画功能，还支持各种图形效果和滤镜，这些可以用来改变图像或控件的外观。通过这些功能，可以为用户界面添加视觉上的吸引力和更复杂的效果。

### 使用图形效果和滤镜

PyQt5中的图形效果通常是通过`QGraphicsEffect`类及其子类来实现的。这些效果可以直接应用于任何继承自`QWidget`的对象。

主要的图形效果包括：

- **模糊效果** (`QGraphicsBlurEffect`)：为图像或控件添加模糊效果。
- **阴影效果** (`QGraphicsDropShadowEffect`)：为控件添加阴影。
- **颜色调整** (`QGraphicsColorizeEffect`)：改变控件的颜色。

### 示例代码：模糊和颜色调整效果

以下示例展示了如何将模糊效果和颜色调整效果应用于一个控件。

```
from PyQt5.QtWidgets import QApplication, QLabel, QGraphicsBlurEffect, QGraphicsColorizeEffect
import sys

app = QApplication(sys.argv)

# 创建一个标签
label = QLabel("Hello, PyQt5!")
label.show()

# 创建并应用模糊效果
blur_effect = QGraphicsBlurEffect()
blur_effect.setBlurRadius(5) # 设置模糊半径
label.setGraphicsEffect(blur_effect)

# 创建并应用颜色调整效果
colorize_effect = QGraphicsColorizeEffect()
colorize_effect.setColor(Qt.blue) # 设置颜色
label.setGraphicsEffect(colorize_effect)

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

在这个例子中，我们首先创建了一个`QLabel`控件。接着，我们创建了一个`QGraphicsBlurEffect`对象来添加模糊效果，并设置了模糊半径。然后，我们创建了一个`QGraphicsColorizeEffect`对象来改变标签的颜色。这些效果可以单独使用，也可以结合使用以创造更多样化的视觉效果。

通过这些工具，您可以为PyQt5应用程序添加丰富和吸引人的视觉效果，从而提升用户体验。

## 8.8 逐帧动画

逐帧动画是一种通过连续显示一系列静态图像（每个图像是动画的一个“帧”）来创造运动效果的动画形式。这种类型的动画非常适合创建复杂和详细的动画序列，例如人物运动、复杂的变形效果等。

### 创建逐帧动画

在PyQt5中，创建逐帧动画通常涉及以下步骤：

1. **准备帧序列**：这可以是一系列图像文件，或者是在程序中动态生成的图像序列。
2. **定时更换帧**：使用定时器（例如QTimer）按一定的间隔更换显示的帧。
3. **显示帧**：在每个定时器事件中，更新显示的图像。

### 示例代码：逐帧动画

以下是一个简单的逐帧动画示例，演示了如何在PyQt5应用程序中实现这种动画。

```
from PyQt5.QtWidgets import QApplication, QLabel
from PyQt5.QtGui import QPixmap
from PyQt5.QtCore import QTimer
import sys

class FrameAnimation(QLabel):
    def __init__(self, image_files, interval=100):
        super().__init__()

        self.pixmap_list = [QPixmap(img) for img in image_files]
        self.current_frame = 0
        self.interval = interval

        self.timer = QTimer(self)
        self.timer.timeout.connect(self.next_frame)
        self.timer.start(self.interval)

    def next_frame(self):
        self.setPixmap(self.pixmap_list[self.current_frame])
        self.current_frame = (self.current_frame + 1) % len(self.pixmap_list)

app = QApplication(sys.argv)

# 假设有一系列图像文件名
image_files = ["frame1.png", "frame2.png", "frame3.png", ...]

animation_label = FrameAnimation(image_files, 100)
animation_label.show()

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

在这个例子中，我们定义了一个FrameAnimation类，它继承自QLabel。这个类初始化时会加载一系列图像，并设置一个定时器来每隔一定时间切换显示的图像。next\_frame方法用于更新标签显示的图像，从而创建动画效果。

逐帧动画非常适合创造复杂和详细的视觉效果，但需要注意的是，大量的帧和高频率的更新可能会对性能产生影响。因此，在设计逐帧动画时，应当考虑到动画的性能和资源占用。

## 总结

以下是一些关键点的总结：

### Important

- 1.动画基础：了解动画的基本概念，如帧、时间轴和缓动，是创建流畅动画的起点。
- 2.属性动画：通过逐步改变对象的属性（如位置、大小、颜色）来创造动画，是提升用户体验的常用方法。
- 3.QGraphicsView和QGraphicsItem：这些类提供了一个强大的框架，用于创建复杂的2D图形和动画。
- 4.动画控制和交互：动画的播放、暂停、停止和循环功能，以及用户交互的整合，增强了应用的灵活性和用户体验。
- 5.缓动函数和自定义过渡效果：使用预定义或自定义的缓动函数，可以创造出独特且自然的动画效果。
- 6.图形效果和滤镜：PyQt5的图形效果和滤镜功能，如模糊、阴影和颜色调整，可以显著提升视觉吸引力。
- 7.逐帧动画：用于创建更复杂和详细的视觉效果，尤其是在需要精细控制每个动画阶段时。

## 9. 高级自定义控件

PyQt5通过提供强大的自定义控件功能，使开发者能够设计和实现具有独特视觉和交互特性的用户界面。通过自定义控件，开发者可以不仅遵循最佳的用户体验设计原则，还能确保应用程序在不同平台上保持一致性和功能性。

### 9.1 自定义控件的基本概念

在深入探讨自定义控件之前，让我们简要回顾一下PyQt5控件和继承机制的基本概念

控件(Widgets)：在PyQt5中，控件是构成用户界面的基本元素，如按钮、文本框、标签等。每个控件都是QWidget类的实例或其子类的实例。

继承机制：PyQt5中的控件继承体系允许开发者通过继承现有控件类来创建新的控件。这种继承机制提供了极大的灵活性，使得自定义控件可以扩展和改进现有控件的功能。

#### 控件的用途

自定义控件在PyQt5应用程序开发中发挥着关键作用。以下是自定义控件的一些主要用途和它们可以解决的常见问题：

- 1.实现特定的用户界面需求：标准控件可能不总能满足特定的应用需求。自定义控件允许开发者设计符合特定需求的界面元素，如具有特殊行为的按钮或具有独特布局的表单。
- 2.增强应用的视觉吸引力：通过自定义控件，开发者可以设计独特的视觉元素，从而提升应用的整体外观和用户体验。
- 3.提升应用的交互性：自定义控件可以包含复杂的交互逻辑，如拖放功能、动画效果等，这些在标准控件中可能难以实现。
- 4.重用和模块化：创建可重用的自定义控件可以提高代码的模块化和维护性。这样的控件可以轻松地在不同的项目中复用，节省开发时间和资源。

### 9.2 实现自定义绘图

#### 步骤说明

在PyQt5中，自定义绘图主要通过重写控件的paintEvent方法来实现。这个方法是在控件需要重新绘制时调用的，例如当控件首次显示或大小改变时。以下是实现自定义绘图的基本步骤：

- 1.创建控件类：首先，创建一个新的控件类，该类继承自QWidget或其它基本控件类。

2.重写paintEvent方法：在控件类中重写paintEvent方法。这个方法中包含了绘图的逻辑。

3.使用绘图工具：在paintEvent方法内部，使用QPainter对象来进行绘制。QPainter提供了丰富的API来绘制各种图形，如线条、形状和文本。

4.更新控件：在需要的时候，调用控件的update()方法来请求重绘控件，这将导致paintEvent的执行。

## 代码示例

以下是一个简单的自定义绘图示例：

```
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QColor
import sys

class CustomWidget(QWidget):
    def paintEvent(self, event):
        painter = QPainter(self)
        painter.setPen(QColor(255, 0, 0)) # 设置画笔颜色为红色
        painter.drawRect(10, 10, 100, 100) # 绘制一个矩形

app = QApplication(sys.argv)
widget = CustomWidget()
widget.show()
sys.exit(app.exec_())
```



在这个示例中，CustomWidget类重写了paintEvent方法，使用QPainter绘制了一个红色的矩形。

### 💡 Tip

#### 技巧和最佳实践:

1.**高效绘图**：只在必要时重绘，避免不必要的更新。可以通过使用update()而非repaint()来请求重绘，因为update()是在Qt的事件循环中优化处理的。

2.**资源管理**：确保正确管理绘图资源，如QPainter对象。一般来说，应该在paintEvent开始时创建QPainter对象，并在绘制完成后释放。

**3.使用双缓冲：**为了减少闪烁和提高绘图性能，可以使用双缓冲技术。在Qt中，这通常是默认启用的。

**4.合理利用坐标系：**熟悉并合理使用Qt的坐标系，包括坐标转换和视图变换，可以更容易地实现复杂的绘图需求。

## 9.3 事件处理

### 事件处理机制

在PyQt5中，事件处理机制是与用户界面交互的核心。事件是用户或系统由控件触发的各种动作，如鼠标点击、键盘输入或定时器到期。每个PyQt5应用都有一个事件循环，它不断检测并分派事件到相应的控件上。

**1.事件对象：**每个事件都是QEvent类的一个实例，包含了事件的相关信息，如事件类型和事件源。

**2.事件分派：**当事件发生时，Qt框架将事件对象传递给相应控件的事件处理函数。

**3.事件处理函数：**控件可以通过重写特定的事件处理函数来响应不同类型的事件。例如，mousePressEvent用于处理鼠标按下事件，keyPressEvent用于处理键盘按下事件。

### 自定义事件处理

为自定义控件添加交互性通常涉及重写相应的事件处理函数。以下是一个示例，展示了如何为自定义控件处理鼠标点击和键盘事件

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

class InteractiveWidget(QWidget):
    def mousePressEvent(self, event):
        # 处理鼠标点击事件
        if event.button() == Qt.LeftButton:
            print("鼠标左键点击")

    def keyPressEvent(self, event):
        # 处理键盘按下事件
        if event.key() == Qt.Key_Space:
            print("空格键按下")

app = QApplication(sys.argv)
widget = InteractiveWidget()
widget.show()
sys.exit(app.exec_())
```

在这个示例中，InteractiveWidget类重写了mousePressEvent和keyPressEvent方法来分别处理鼠标点击和键盘按下事件。当用户点击鼠标左键或按下空格键时，控件将打印相应的消息。

通过这种方式，你可以为自定义控件添加各种交互功能，从而提升用户体验和应用程序的互动性。

## 9.4 自定义属性

### 属性的重要性

在PyQt5的控件开发中，属性（Properties）扮演着重要的角色。属性不仅是对象状态的表现，也是对象行为的关键。在PyQt5中，通过使用属性，可以：

1. **保存和更新状态**：属性允许存储关于控件的重要信息，比如颜色、大小或自定义数据。
2. **实现数据绑定**：属性可以与应用程序的其它部分进行数据绑定，使得当数据变化时，界面能够自动更新。
3. **动画和样式**：在动画和样式表中，属性是关键元素，允许动态改变控件的外观和行为。

## 创建和使用属性

在PyQt5中，创建自定义属性通常涉及使用pyqtProperty装饰器。以下是创建和使用自定义属性的步骤：

1. **定义属性的getter和setter方法**：这些是普通的Python方法，用于获取和设置属性值。
2. **使用pyqtProperty装饰器**：将这些方法转换为Qt属性。
3. **在控件中使用属性**：一旦定义了属性，就可以在控件的其它部分，如事件处理或绘图方法中使用这些属性。

## 示例代码

以下示例展示了如何为自定义控件创建和使用一个简单的属性：

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

class CustomWidget(QWidget):
    def __init__(self, parent=None):
        super(CustomWidget, self).__init__(parent)
        self._my_property = 0

    def getMyProperty(self):
        return self._my_property

    def setMyProperty(self, value):
        self._my_property = value
        self.update() # 更新控件以反映属性变化

    myProperty = pyqtProperty(int, getMyProperty, setMyProperty)

app = QApplication(sys.argv)
widget = CustomWidget()
widget.myProperty = 10 # 设置属性值
print(widget.myProperty) # 获取并打印属性值
widget.show()
sys.exit(app.exec_())
```

在这个示例中，CustomWidget类中定义了一个名为myProperty的自定义属性。通过为这个属性提供getter和setter方法，并使用pyqtProperty装饰器，使其成为一个可用于Qt框架内部的完整属性。

通过这种方式，你可以为自定义控件添加丰富的功能性和灵活性，使其更适合复杂的应用场景。

## 9.5 进阶功能

### 动画和复杂逻辑

在自定义控件中加入动画效果和处理复杂逻辑可以显著提升应用的用户体验和视觉吸引力。以下是实现这些高级功能的要点：

#### 1. 动画效果：

- 使用QPropertyAnimation和QAnimationGroup类来创建平滑的动画效果。这些类允许对控件属性进行动画处理，如位置、大小、颜色等。
- 结合QTimer实现逐帧动画，适用于更复杂的动画效果，如精灵动画或复杂的动态变换。
- 考虑使用QGraphicsEffect来添加特殊效果，如阴影、模糊等。

## 2.处理复杂逻辑：

- 在自定义控件中实现复杂逻辑，如数据处理、状态管理等。这可能涉及与应用程序的其它部分（如数据库、文件系统）的交互。
- 使用事件驱动的方法处理用户输入和响应，如通过重写mousePressEvent、keyPressEvent等来响应用户操作。

## 10. 绘制 1 8 种图

### 10.1 折线图

```
import sys
import pyqtgraph as pg
from PyQt5 import QtWidgets
import numpy as np

class MainWindow(QtWidgets.QMainWindow):
    def __init__(self, *args, **kwargs):
        super(MainWindow, self).__init__(*args, **kwargs)

        # 创建一个 PlotWidget 对象
        self.graphWidget = pg.PlotWidget()

        # 设置图表标题和颜色
        self.graphWidget.setBackground('w')
        self.graphWidget.setTitle("Title", color="b", size="30pt")

        # 添加网格
        styles = {"color": "#f00", "font-size": "20px"}
        self.graphWidget.setLabel("left", "Y Axis", **styles)
        self.graphWidget.setLabel("bottom", "X Axis", **styles)

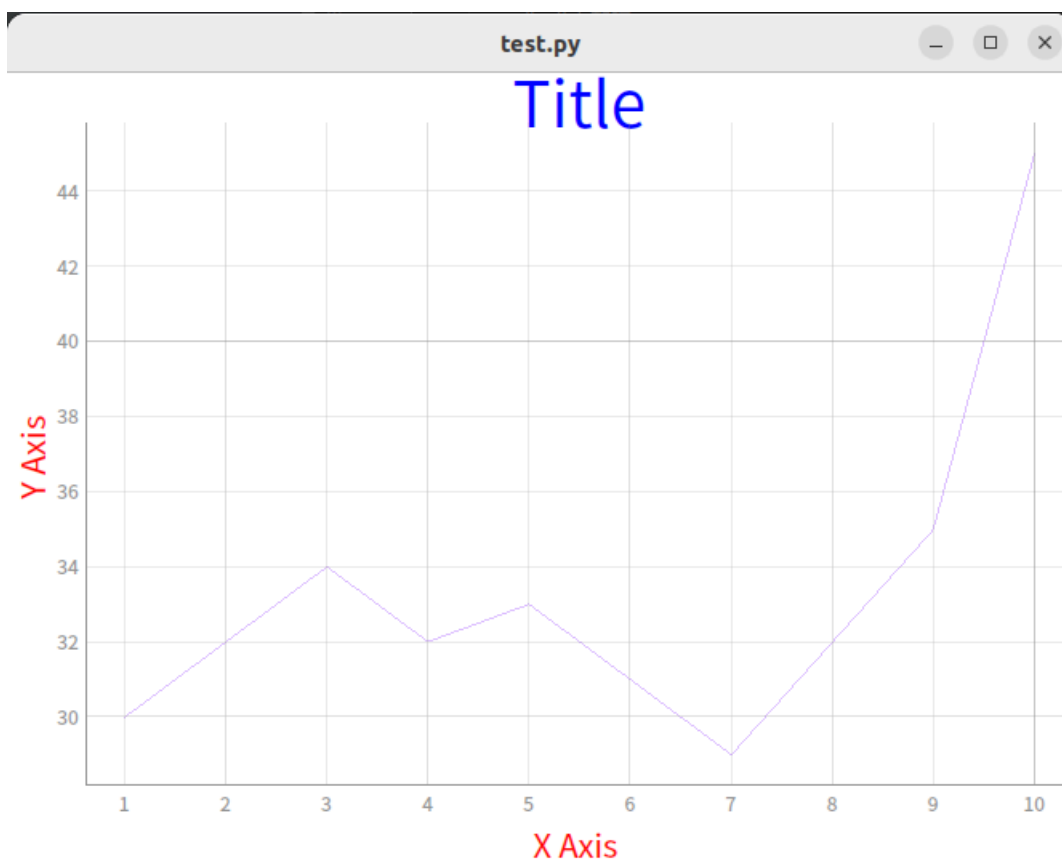
        self.graphWidget.showGrid(x=True, y=True)

        # 设置数据
        x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
        y = np.array([30, 32, 34, 32, 33, 31, 29, 32, 35, 45])

        # 在 PlotWidget 上绘制数据
        pen = pg.mkPen(color="#dcbfff")
        self.graphWidget.plot(x, y, pen=pen)

        # 设置主窗口的中心小部件为图表
        self.setCentralWidget(self.graphWidget)

app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())
```



这个脚本首先创建了一个 PyQt 应用程序和一个主窗口。然后，它创建了一个 PlotWidget 对象，这是 PyQtGraph 提供的用于绘图的控件。接下来，使用 plot 方法在绘图控件上绘制了一条折线图，其中 x 和 y 是数据点的列表。最后，显示主窗口并启动应用程序的事件循环。

## 10.2 二维填充图

在 PyQtGraph 中创建二维填充图可以通过使用 PlotWidget 和 fillLevel 参数来实现。二维填充图是指在折线图的基础上，将线下方的区域填充以颜色或图案。

下面是一个示例代码，展示如何使用 PyQtGraph 创建一个简单的二维面图：

```
import sys
import numpy as np
from PyQt5 import QtWidgets
import pyqtgraph as pg

class MainWindow(QtWidgets.QMainWindow):
    def __init__(self):
        super(MainWindow, self).__init__()

        # 创建一个 PlotWidget 对象
        self.graphWidget = pg.PlotWidget()

        # 设置图表的标题和颜色
        self.graphWidget.setBackground('w')
        self.graphWidget.setTitle("2D Surface Plot", color="b", size="15pt")

        # 添加网格
        self.graphWidget.showGrid(x=True, y=True)
```



```

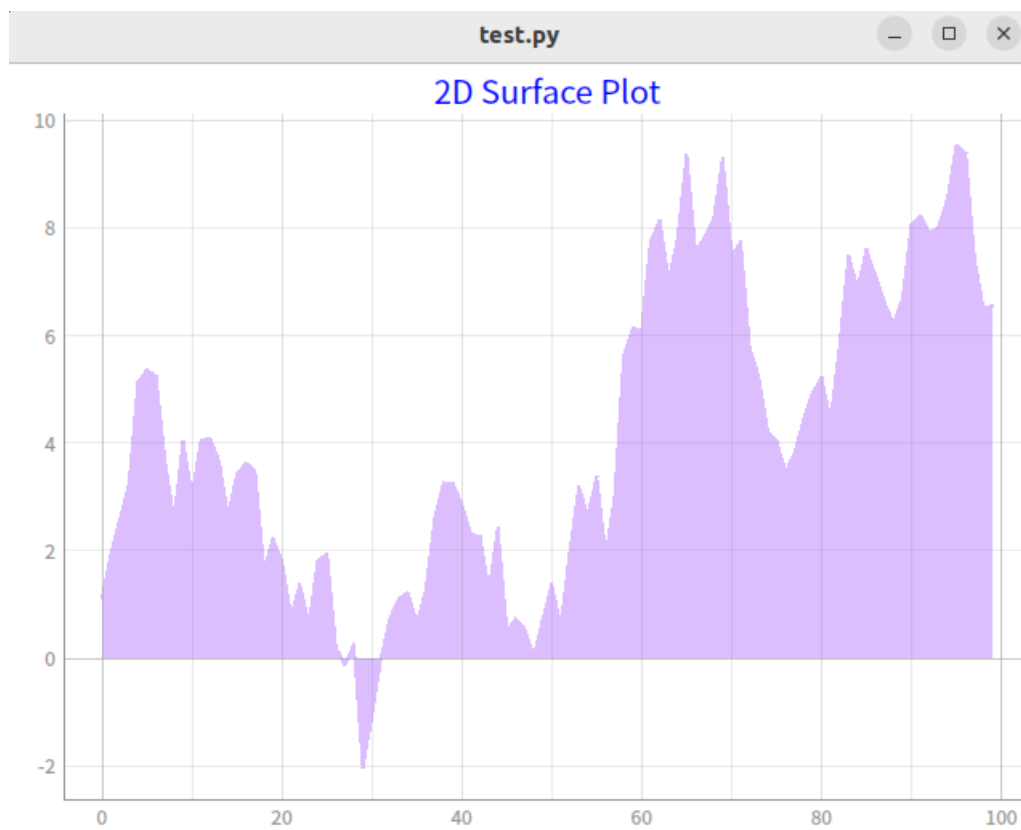
# 设置数据
x = np.arange(100) # X轴数据
y = np.random.normal(size=100).cumsum() # Y轴数据, 这里使用累积和来模拟一个变化的数据集

# 绘制面图
plotItem = self.graphWidget.plot(x, y, pen=pg.mkPen(color="#dcbfff", width=2))
plotItem.setFillBrush(pg.mkBrush(color="#dcbfff", alpha=90))
plotItem.setFillLevel(0)

# 设置主窗口的中心小部件为图表
self.setCentralWidget(self.graphWidget)

app = QtWidgets.QApplication(sys.argv)
main = MainWindow()
main.show()
sys.exit(app.exec_())

```



在这个例子中, 我们创建了一个 `PlotWidget`, 然后使用 `plot` 方法绘制了一条蓝色的线, 并使用 `setFillBrush` 方法设置了填充颜色和 `setFillLevel` 方法指定了填充的底部水平线。alpha 参数设置了填充颜色的透明度。

### 10.3 三维曲面图

绘制三维曲面图可以通过在 PyQt5 结合 Matplotlib 来进行绘制, 通过使用 Matplotlib 的 `FigureCanvasQTAgg` 将 Matplotlib 图嵌入到 PyQt5 界面中。以下是一个简单的示例, 展示了如何使用 Matplotlib 在 PyQt5 应用程序中创建一个三维图:

```

import sys
from PyQt5.QtWidgets import QApplication, QVBoxLayout, QMainWindow, QWidget
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import numpy as np

```

```
class ThreeDPlotWidget(QWidget):
    def __init__(self, parent=None):
        super(ThreeDPlotWidget, self).__init__(parent)
        self.setupUi()

    def setupUi(self):
        # 创建一个 Matplotlib 图和轴
        self.figure = Figure()
        self.canvas = FigureCanvas(self.figure)
        self.ax = self.figure.add_subplot(111, projection='3d')

        # 在布局中添加 canvas
        layout = QVBoxLayout()
        layout.addWidget(self.canvas)
        self.setLayout(layout)

        # 添加三维数据
        self.add3dPlot()

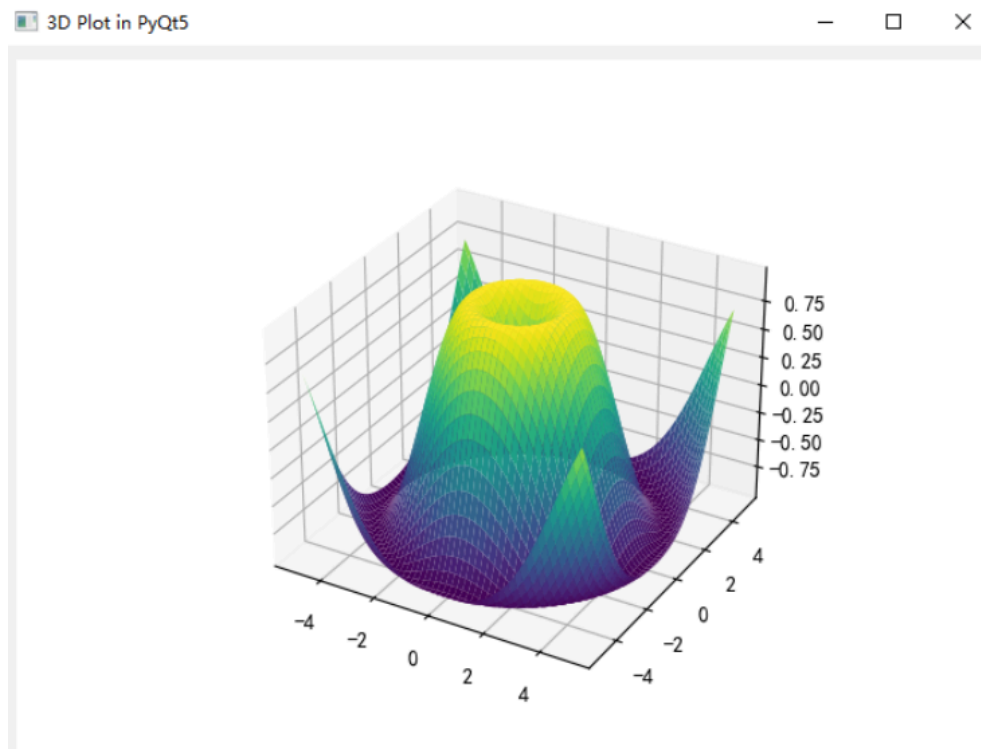
    def add3dPlot(self):
        # 示例数据
        x = np.linspace(-5, 5, 100)
        y = np.linspace(-5, 5, 100)
        x, y = np.meshgrid(x, y)
        z = np.sin(np.sqrt(x**2 + y**2))

        # 绘制三维图
        self.ax.plot_surface(x, y, z, cmap='viridis')
        self.canvas.draw()

class MainWindow(QMainWindow):
    def __init__(self):
        super(MainWindow, self).__init__()

        # 设置主窗口的中心小部件
        self.threeDPlotWidget = ThreeDPlotWidget()
        self.setCentralWidget(self.threeDPlotWidget)
        self.setWindowTitle("3D Plot in PyQt5")

app = QApplication(sys.argv)
mainWin = MainWindow()
mainWin.show()
sys.exit(app.exec_())
```



这个程序创建了一个 QMainWindow，其中包含一个自定义的 ThreeDPlotWidget。这个小部件包含一个用于绘制三维图表的 Matplotlib FigureCanvas。在 add3dPlot 方法中，我们使用 Matplotlib 绘制一个三维曲面图，并将其显示在 PyQt5 窗口中。

#### 10.4 柱状图

要在 PyQtGraph 中绘制柱状图，需要使用 BarGraphItem 类。以下是一个如何使用 PyQtGraph 创建并显示柱状图的示例：

以下是一个简单的示例，展示了如何创建一个包含柱状图的 PyQt 应用程序。

```
import sys
from PyQt5 import QtWidgets
import pyqtgraph as pg

# 创建一个应用程序实例
app = QtWidgets.QApplication(sys.argv)

# 创建一个主窗口
mainWindow = QtWidgets.QMainWindow()

# 创建一个绘图控件
plotWidget = pg.PlotWidget()
plotWidget.setBackground("w")
mainWindow.setCentralWidget(plotWidget)

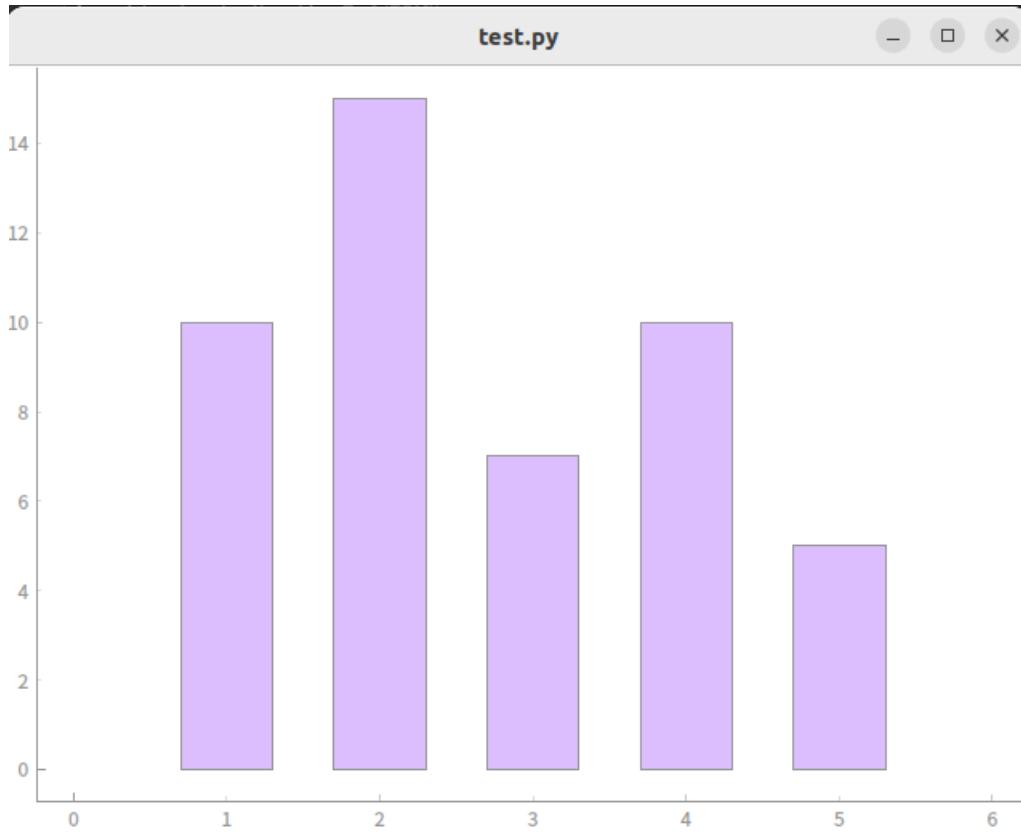
# 准备数据
x = [1, 2, 3, 4, 5] # X轴数据
height = [10, 15, 7, 10, 5] # 柱状图的高度
width = 0.6 # 柱子的宽度

# 创建柱状图项目
barGraphItem = pg.BarGraphItem(x=x, height=height, width=width, brush='#dcbeff')
plotWidget.addItem(barGraphItem)

# 显示主窗口
```

```
mainWindow.show()

# 启动应用程序的事件循环
sys.exit(app.exec_())
```



在这个例子中，我们创建了一个包含柱状图的窗口。x 是柱子的位置，height 是每个柱子的高度，而 width 是柱子的宽度。通过使用 `BarGraphItem` 类来创建柱状图，并添加到 `PlotWidget` 控件中。

## 10.5 饼图

在 `PyQtGraph` 中绘制饼图需要使用不同的方法，因为 `PyQtGraph` 主要是为了绘制二维和三维图形而设计的，它并不直接支持饼图。但是，可以通过使用 `PyQt` 的标准组件，如 `QGraphicsScene` 和 `QGraphicsView`，结合 `QPainterPath` 来绘制饼图。

以下是一个使用 `PyQt5` 绘制饼图的示例代码：

```
import sys
import numpy as np
from PyQt5.QtWidgets import QApplication, QMainWindow, QGraphicsScene, QGraphicsView,
    QGraphicsTextItem
from PyQt5.QtGui import QPainterPath, QBrush, QColor, QFont
from PyQt5.QtCore import QRectF, QPointF

class PieChartView(QGraphicsView):
    def __init__(self, data):
        super().__init__()

        # 创建一个 QGraphicsScene 来容纳饼图
        self.scene = QGraphicsScene()
        self.setScene(self.scene)

        # 绘制饼图
        self.draw_pie_chart(data)
```

```

def draw_pie_chart(self, data):
    # 饼图的起始角度
    start_angle = 0
    total_value = sum(value for value, _ in data)

    for value, color in data:
        # 计算每个扇形的角度
        span_angle = value * 360

        # 创建一个 QPainterPath 对象来绘制这个扇形
        path = QPainterPath()
        path.moveTo(0, 0)
        rect = QRectF(-200, -200, 400, 400) # 设置饼图尺寸
        path.arcTo(rect, start_angle, span_angle)
        path.closeSubpath()

        # 创建一个画刷并设置颜色
        brush = QBrush(QColor(color))

        # 将扇形添加到场景中
        slice_item = self.scene.addPath(path, brush=brush)

        # 添加标签
        label = QGraphicsTextItem(f"{value/total_value:.1%}", parent=slice_item)
        label1 = QGraphicsTextItem(f"{value / total_value:.1%}", parent=slice_item)

        label.setFont(QFont("Arial", 10))
        label.setPos(-label.boundingRect().width() / 2, -label.boundingRect().height() / 2)
# Center the label
        label1.setFont(QFont("Arial", 10))

        # 更新起始角度
        start_angle += span_angle

        # 计算标签的位置
        angle = np.radians(start_angle - span_angle / 2) # 中间的角度
        radius = 100 # 标签放置的半径
        label_x = radius * np.cos(angle)
        label_y = radius * np.sin(angle)

        # 考虑标签的尺寸，对其位置进行微调
        label.setPos(label_x - label.boundingRect().width() / 2,
                     label_y - label.boundingRect().height() / 2)

app = QApplication(sys.argv)

# 饼图数据: (值, 颜色)
data = [
    (0.3, '#dcbeff'),
    (0.4, '#fffac8'),
    (0.3, '#aaffc3')
]

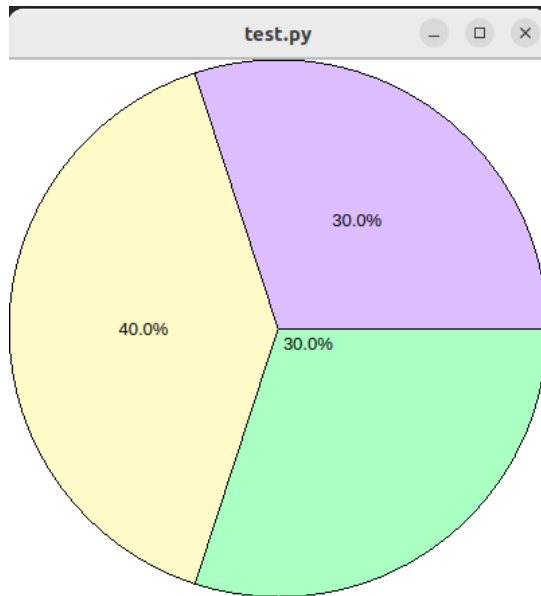
# 创建主窗口
mainWindow = QMainWindow()

```

```
# 创建饼图视图并设置为主窗口的中心控件
pieChartView = PieChartView(data)
mainWindow.setCentralWidget(pieChartView)

# 显示主窗口
mainWindow.show()

# 启动应用程序的事件循环
sys.exit(app.exec_())
```



在这个例子中，我们创建了一个继承自 `QGraphicsView` 的 `PieChartView` 类来显示饼图。数据以（值，颜色）对的形式传递给 `PieChartView`，每个数据对表示饼图中一个扇区的大小和颜色。

## 10.6 散点图

编写代码以绘制散点图:

```
import sys
from PyQt5 import QtWidgets
import pyqtgraph as pg
import numpy as np

# 创建一个应用程序实例
app = QtWidgets.QApplication(sys.argv)

# 创建一个主窗口
mainWindow = QtWidgets.QMainWindow()

# 创建一个绘图控件
plotWidget = pg.PlotWidget()
plotWidget.setBackground("w")
mainWindow.setCentralWidget(plotWidget)

# 准备数据
x = np.random.normal(size=1000)
y = np.random.normal(size=1000)

# 使用散点图绘制数据
```

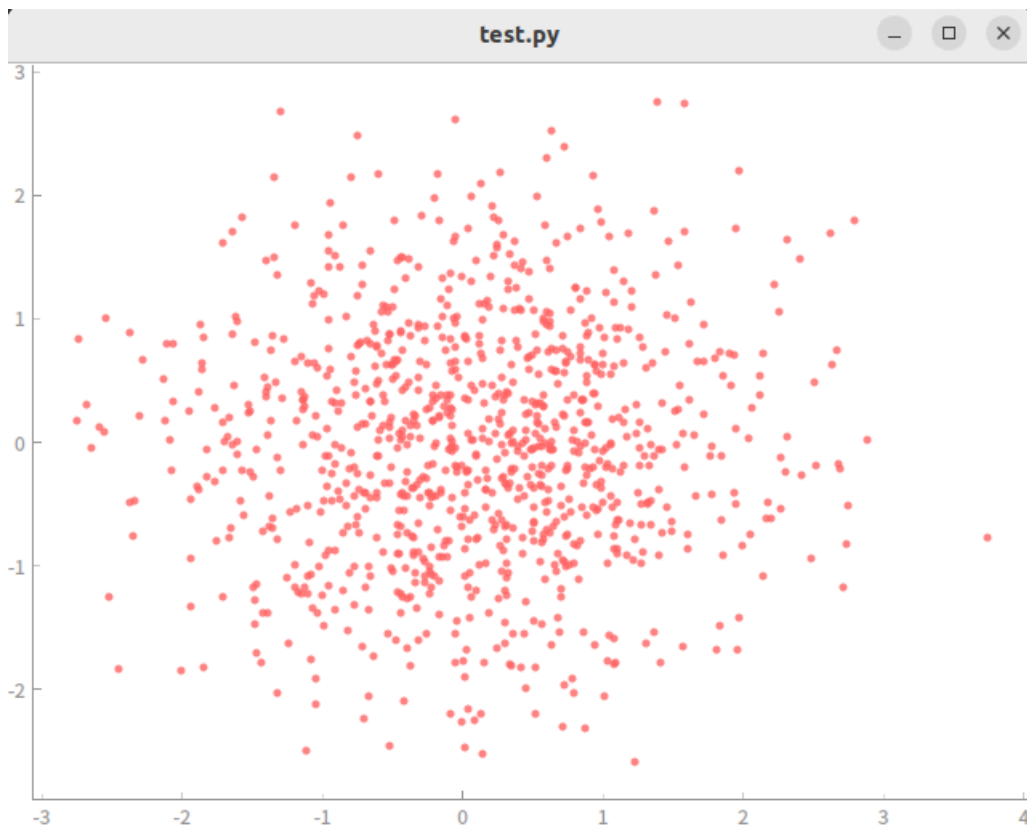
```

scatterPlotItem = pg.ScatterPlotItem(pen=pg.mkPen(None), brush=pg.mkBrush(255, 100, 100, 200),
size=5)
scatterPlotItem.setData(x, y)
plotWidget.addItem(scatterPlotItem)

# 显示主窗口
mainWindow.show()

# 启动应用程序的事件循环
sys.exit(app.exec_())

```



在这个例子中，我们创建了一个包含散点图的窗口。x 和 y 是数据点的坐标数组，这里我们使用 numpy 生成了一些随机数据。我们使用 ScatterPlotItem 类来创建散点图，并添加到 PlotWidget 控件中。

## 10.7 热力图

在 PyQtGraph 中绘制热力图可以通过使用 ImageItem 来完成。热力图通常用于表示二维数据集的强度或频率。以下是如何使用 PyQtGraph 创建热力图的步骤和示例代码：

```

import sys
import numpy as np
from PyQt5 import QtWidgets
import pyqtgraph as pg

# 创建一个应用程序实例
app = QtWidgets.QApplication(sys.argv)

# 创建一个主窗口
mainWindow = QtWidgets.QMainWindow()

# 创建一个绘图控件
plotWidget = pg.PlotWidget()
plotWidget.setBackground("w")

```

```
mainWindow.setCentralWidget(plotWidget)

# 创建一个
ImageItemimgItem = pg.ImageItem()

# 添加 ImageItem 到绘图控件
plotWidget.addItem(ImageItemimgItem)

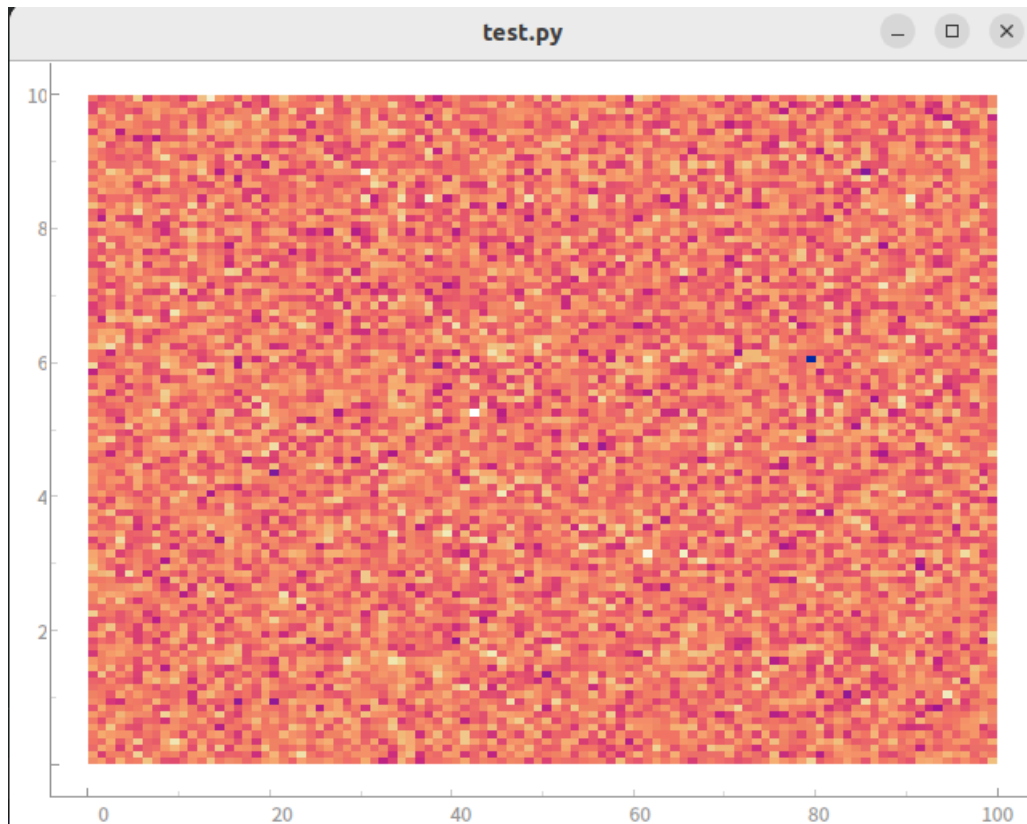
# 准备数据：生成一个二维数组
data = np.random.normal(size=(100, 100))

# 设置 ImageItem 的图像数据
ImageItemimgItem.setImage(data)

# 使用颜色映射来表示不同的强度
colorMap = pg.colormap.get('CET-L17')
ImageItemimgItem.setLookupTable(colorMap.getLookupTable())

# 显示主窗口
mainWindow.show()

# 启动应用程序的事件循环
sys.exit(app.exec_())
```



在这个例子中，我们使用 `ImageItem` 来创建热力图。`data` 是一个二维数组，用于表示热力图中的数值。`setLookupTable` 方法用于设置颜色映射，根据数据的不同值显示不同的颜色。



## 10.8 雷达图

PyQtGraph 并不直接支持雷达图（也称为蜘蛛网图或极坐标图），但您可以使用 PyQt 的绘图能力手动创建一个。要在 PyQt 中绘制雷达图，您需要使用 QPainter 类来在 QWidget 或类似对象上进行自定义绘图。

以下是一个使用 PyQt5 创建简单雷达图的示例：

```
import sys
import math
from PyQt5.QtWidgets import QApplication, QWidget
from PyQt5.QtGui import QPainter, QPolygon, QPen
from PyQt5.QtCore import Qt, QPoint

class RadarWidget(QWidget):
    def __init__(self, data):
        super().__init__()
        self.data = data

    def paintEvent(self, event):
        painter = QPainter(self)
        painter.setRenderHint(QPainter.Antialiasing)

        # 雷达图的中心点
        center = QPoint(int(self.width() / 2), int(self.height() / 2))
        radius = min(int(self.width() / 2), int(self.height() / 2)) - 10

        # 绘制雷达图的轴
        num_axes = len(self.data)
        for i in range(num_axes):
            angle = 2 * math.pi * i / num_axes
            x = int(center.x() + radius * math.cos(angle))
            y = int(center.y() + radius * math.sin(angle))
            painter.drawLine(center, QPoint(x, y))

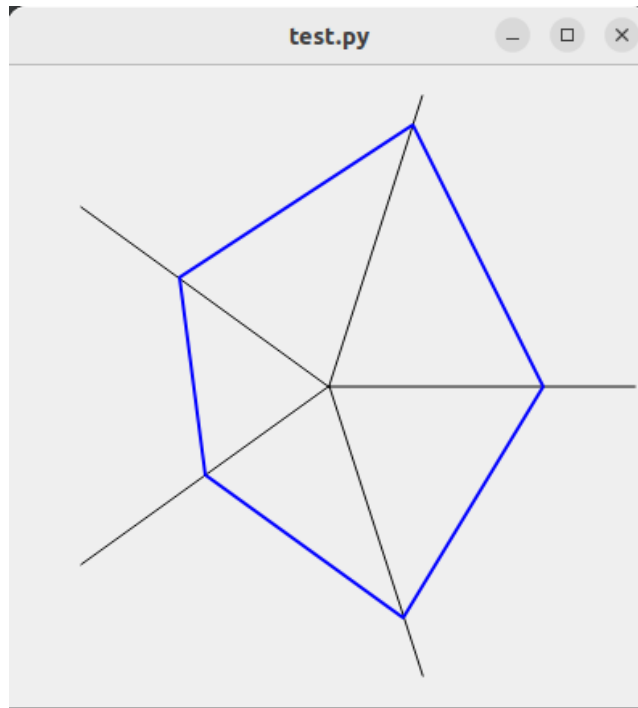
        # 绘制雷达图的数据多边形
        polygon = QPolygon()
        for i in range(num_axes):
            angle = 2 * math.pi * i / num_axes
            value = self.data[i]
            x = int(center.x() + value * radius * math.cos(angle))
            y = int(center.y() + value * radius * math.sin(angle))
            polygon << QPoint(x, y)
        painter.setPen(QPen(Qt.blue, 2))
        painter.drawPolygon(polygon)

# 创建一个 QApplication 实例
app = QApplication(sys.argv)

# 创建雷达图窗口
radar_data = [0.7, 0.8, 0.5, 0.6, 0.9] # 示例数据
radarWidget = RadarWidget(radar_data)
radarWidget.resize(400, 400)
radarWidget.show()

# 启动应用程序的事件循环
```

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



在这个例子中，我们创建了一个继承自 `QWidget` 的 `RadarWidget` 类来显示雷达图。数据以列表形式传递给 `RadarWidget`，每个元素表示雷达图上一个轴的值。

## 10.9 流程图

在 `PyQtGraph` 中绘制流程图需要使用图形和线条来手动创建图形元素。`PyQtGraph` 主要专注于数据可视化，并不直接支持流程图的绘制。但是，您可以使用 `PyQt5` 的图形视图框架（`QGraphicsScene` 和 `QGraphicsView`）来创建流程图。

```
import sys
import math
from PyQt5.QtWidgets import QApplication, QGraphicsScene, QGraphicsView, QGraphicsRectItem,
QGraphicsTextItem, QGraphicsLineItem, QGraphicsPolygonItem
from PyQt5.QtCore import Qt, QRectF, QPointF, QLineF
from PyQt5.QtGui import QPen, QBrush, QColor, QPolygonF

class Arrow(QGraphicsLineItem):
    def __init__(self, start_item, end_item, parent=None):
        super().__init__(parent)

        # 计算起始点和结束点
        start_point = start_item.mapToScene(start_item.rect().bottomLeft() +
        QPointF(start_item.rect().width() / 2, 0))
        end_point = end_item.mapToScene(end_item.rect().topLeft() +
        QPointF(end_item.rect().width() / 2, 0))

        # 设置线条
        line = QLineF(start_point, end_point)
        self.setLine(line)
        self.setPen(QPen(Qt.black, 2))

        # 箭头头部尺寸
        arrow_size = 9.0
```

```

# 计算箭头角度
angle = math.atan2(-line.dy(), line.dx())

# 创建箭头头部
arrow_head = QPolygonF()
arrow_head.append(line.p2())
arrow_head.append(line.p2() - QPointF(math.sin(angle + math.pi / 3) * arrow_size,
                                         math.cos(angle + math.pi / 3) * arrow_size))
arrow_head.append(line.p2() - QPointF(math.sin(angle + math.pi - math.pi / 3) *
arrow_size,
                                         math.cos(angle + math.pi - math.pi / 3) *
arrow_size))

# 添加箭头头部到场景
arrow_head_item = QGraphicsPolygonItem(arrow_head, self)
arrow_head_item.setBrush(Qt.black)
arrow_head_item.setPen(QPen(Qt.black, 2))

class FlowChartNode(QGraphicsRectItem):
    def __init__(self, text, x, y, width, height, parent=None):
        super().__init__(x, y, width, height, parent)

        self.setBrush(QBrush(QColor(200, 200, 255)))
        self.setPen(QPen(Qt.black, 2))

# 添加文本标签
text_item = QGraphicsTextItem(text, self)
text_item.setPos(x + 10, y + 10)
text_item.setTextWidth(width - 20)

app = QApplication(sys.argv)
scene = QGraphicsScene()

node1 = FlowChartNode("开始", 50, 50, 100, 50)
node2 = FlowChartNode("步骤1", 50, 150, 100, 50)
node3 = FlowChartNode("结束", 50, 250, 100, 50)

scene.addItem(node1)
scene.addItem(node2)
scene.addItem(node3)

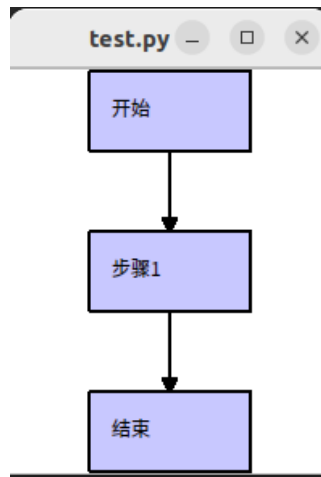
arrow1 = Arrow(node1, node2)
arrow2 = Arrow(node2, node3)

scene.addItem(arrow1)
scene.addItem(arrow2)

view = QGraphicsView(scene)
view.show()

sys.exit(app.exec_())

```



在这个例子中，我们定义了一个 `FlowChartNode` 类来表示流程图中的节点，该类继承自 `QGraphicsRectItem`。然后，我们创建了一个 `QGraphicsScene` 来容纳这些节点，并使用 `QGraphicsView` 显示它们。

### 10.10 树状图

在 `PyQtGraph` 中创建树状图（或树形结构图）通常不是直接支持的，因为 `PyQtGraph` 主要用于数据可视化，如绘制折线图、散点图等。但是，您可以使用 `PyQt5` 的 `QTreeView` 和 `QStandardItemModel` 来创建一个树状图界面。

以下是使用 `PyQt5` 创建树状图的示例代码：

```
import sys
from PyQt5.QtWidgets import QApplication, QTreeView
from PyQt5.QtGui import QStandardItem, QStandardItemModel

app = QApplication(sys.argv)

# 创建标准项模型
model = QStandardItemModel()
rootNode = model.invisibleRootItem()

# 创建节点
parent1 = QStandardItem('父节点1')
child1 = QStandardItem('子节点1')
child2 = QStandardItem('子节点2')
parent1.appendRow(child1)
parent1.appendRow(child2)

parent2 = QStandardItem('父节点2')
child3 = QStandardItem('子节点3')
parent2.appendRow(child3)

rootNode.appendRow(parent1)
rootNode.appendRow(parent2)

# 创建树视图并设置模型
treeView = QTreeView()
treeView.setHeaderHidden(True)

treeView.setModel(model)
treeView.expandAll() # 展开所有节点

# 显示树视图
treeView.show()
```

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



在这个例子中，我们创建了一个 `QStandardItemModel` 来存储树形数据，然后向模型中添加了几个 `QStandardItem` 作为节点。`QTreeView` 用于展示这个模型。每个节点都可以有子节点，形成树形结构。

### 10.11 网络图

在 `PyQtGraph` 中直接绘制网络图（Graph Visualization）并不直接支持，因为 `PyQtGraph` 主要用于绘制二维和三维的数据可视化图形，如折线图、散点图等。但是，您可以结合使用 `PyQt` 和其他 Python 图形库，如 `NetworkX`，来创建和显示网络图。

以下是一个使用 `PyQt5` 和 `NetworkX` 创建网络图的示例：

```
import sys
import networkx as nx
import matplotlib.pyplot as plt
from PyQt5.QtWidgets import QApplication, QMainWindow, QSizePolicy, QWidget, QVBoxLayout
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgn as FigureCanvas

class NetworkGraph(QWidget):
    def __init__(self, parent=None):
        super(NetworkGraph, self).__init__(parent)

        self.figure, self.ax = plt.subplots()
        self.canvas = FigureCanvas(self.figure)
        self.canvas.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
        self.canvas.updateGeometry()

        layout = QVBoxLayout()
        layout.addWidget(self.canvas)
        self.setLayout(layout)

        self.draw_graph()

    def draw_graph(self):
        # 创建一个网络图
        G = nx.karate_club_graph()

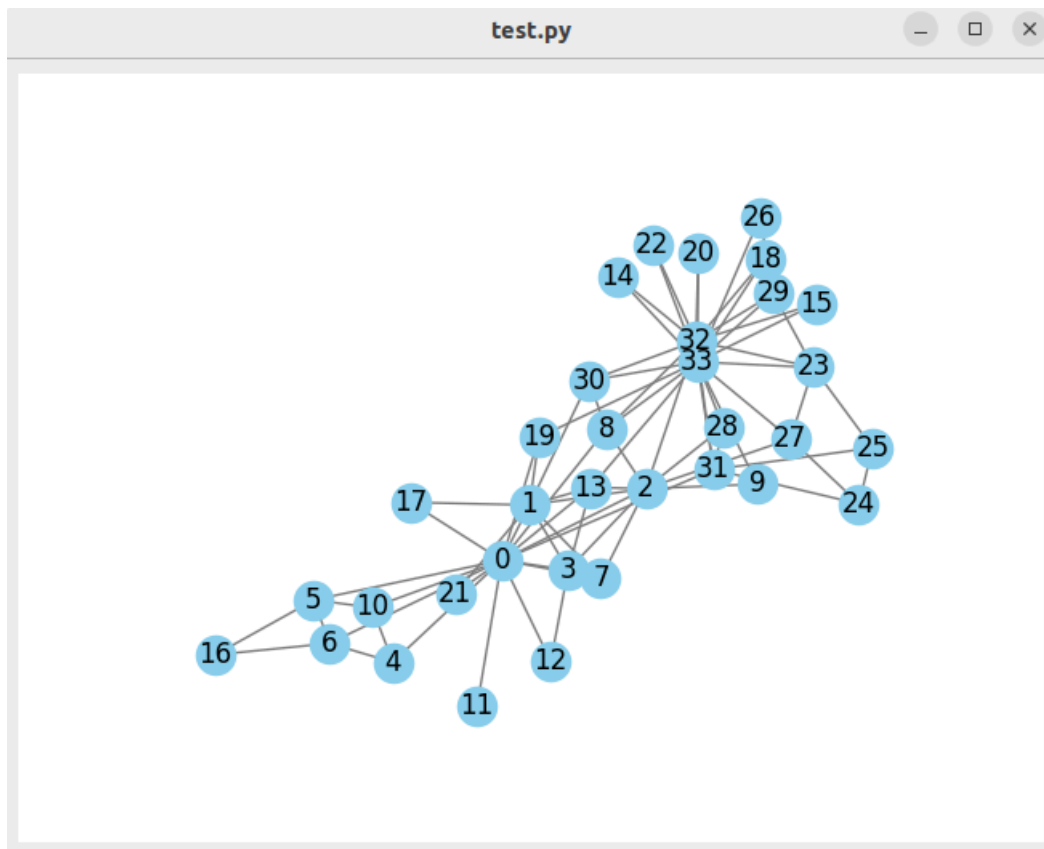
        # 绘制网络图
        pos = nx.spring_layout(G)
        nx.draw(G, pos, ax=self.ax, with_labels=True, node_color='skyblue',
edge_color='gray')

        # 更新画布
        self.canvas.draw()
```

```

app = QApplication(sys.argv)
mainWindow = QMainWindow()
mainWindow.setCentralWidget(NetworkGraph())
mainWindow.show()
sys.exit(app.exec_())

```



在这个示例中，我们创建了一个名为 `NetworkGraph` 的自定义 `QWidget`，用于显示由 `NetworkX` 生成的网络图。使用 `NetworkX` 创建并配置图形，然后使用 `Matplotlib` 进行绘图。最后，将 `Matplotlib` 图形嵌入到 `PyQt5` 应用程序窗口中。

## 10.12 气泡图

在 `PyQtGraph` 中绘制气泡图可以通过使用 `ScatterPlotItem` 类并调整点的大小来实现。气泡图是散点图的一种变体，其中每个点的大小可以代表数据的一个额外维度。

以下是使用 `PyQtGraph` 创建气泡图的步骤和示例代码：

```

import sys
import numpy as np
from PyQt5 import QtWidgets
import pyqtgraph as pg

# 创建一个应用程序实例
app = QtWidgets.QApplication(sys.argv)

# 创建一个主窗口
mainWindow = QtWidgets.QMainWindow()

# 创建一个绘图控件
plotWidget = pg.PlotWidget()
plotWidget.setBackground("w")

```

```

mainWindow.setCentralWidget(plotWidget)

# 准备数据
x = np.random.rand(100) * 10
y = np.random.rand(100) * 10
sizes = np.random.rand(100) * 50 # 气泡的大小

# 创建散点图项目
scatterPlotItem = pg.ScatterPlotItem(pen=pg.mkPen(None), brush="#dcbfff")

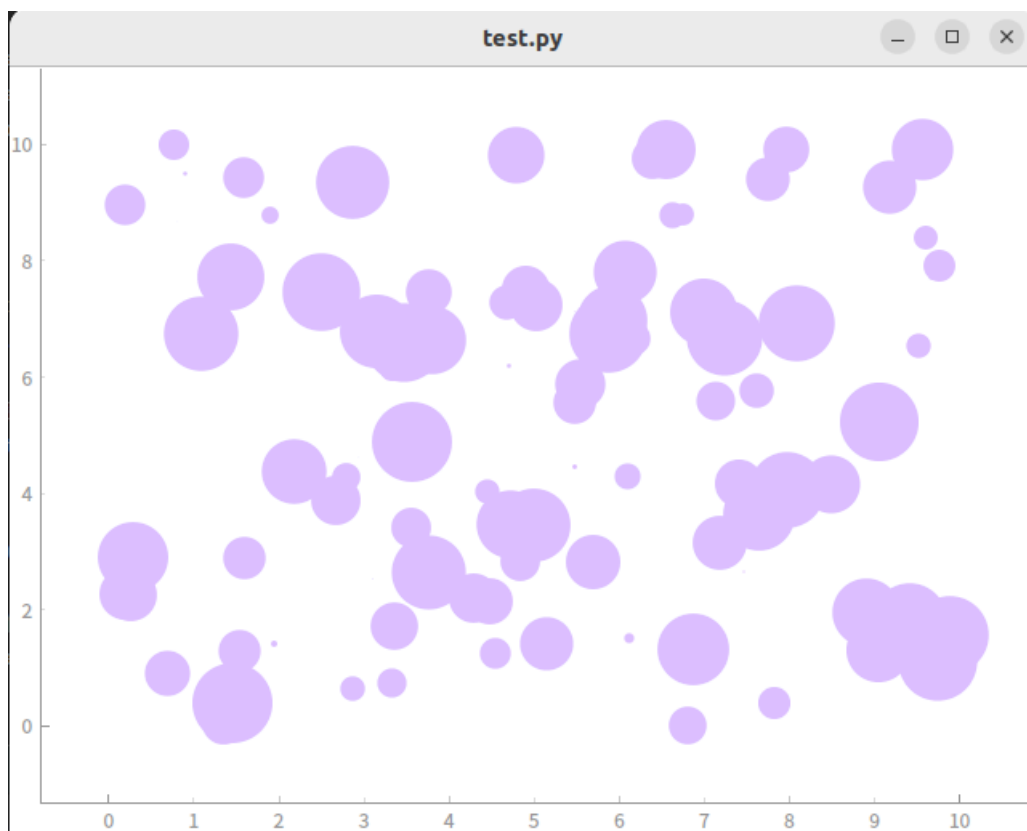
# 现在使用 setData 设置点的坐标和大小
scatterPlotItem.setData(x, y, size=sizes)

# 将散点图添加到绘图控件
plotWidget.addItem(scatterPlotItem)

# 显示主窗口
mainWindow.show()

# 启动应用程序的事件循环
sys.exit(app.exec_())

```



在这个例子中，我们使用 `ScatterPlotItem` 创建了一个散点图，并通过改变每个点的大小（`size` 参数）来创建一个气泡图效果。`x` 和 `y` 是数据点的坐标，`sizes` 数组决定了每个点的大小。

### 10.13 地图图表

在 PyQt 中绘制地图图表通常需要结合使用地理信息系统（GIS）相关的库，例如 `GeoPandas`、`Folium` 或 `PyQGIS`。这些库可以处理地理数据并生成地图，但是它们不是 `PyQtGraph` 的一部分。另一种选择是使用 `QGIS`，这是一个完整的地理信息系统平台，它可以与 PyQt 结合使用。

以下是一个使用 Python、PyQt5 和 Folium 创建交互式地图图表的基本示例：

```

import sys
import os
import folium
from PyQt5.QtWidgets import QApplication, QMainWindow, QWidget
from PyQt5.QtWebEngineWidgets import QWebEngineView

from PyQt5.QtCore import QUrl

class MapWidget(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)
        self.view = QWebEngineView(self)
        self.init_ui()

    def init_ui(self):
        # 创建一个 Folium 地图，您可以调整 zoom_start 来设置初始缩放级别
        m = folium.Map(location=[45.5236, -122.6750], zoom_start=13)

        # 获取当前文件夹的绝对路径
        dirname = os.path.abspath(os.path.dirname('image/test1.png'))
        # 保存地图为 HTML 文件
        map_path = os.path.join(dirname, 'map.html')
        m.save(map_path)

        # 在 QWebEngineView 中加载 HTML
        self.view.load(QUrl.fromLocalFile(map_path))

        # 设置 QWebEngineView 控件的最小尺寸
        self.view.setMinimumSize(800, 600) # 例如，设置为宽800像素，高600像素
        self.view.show()

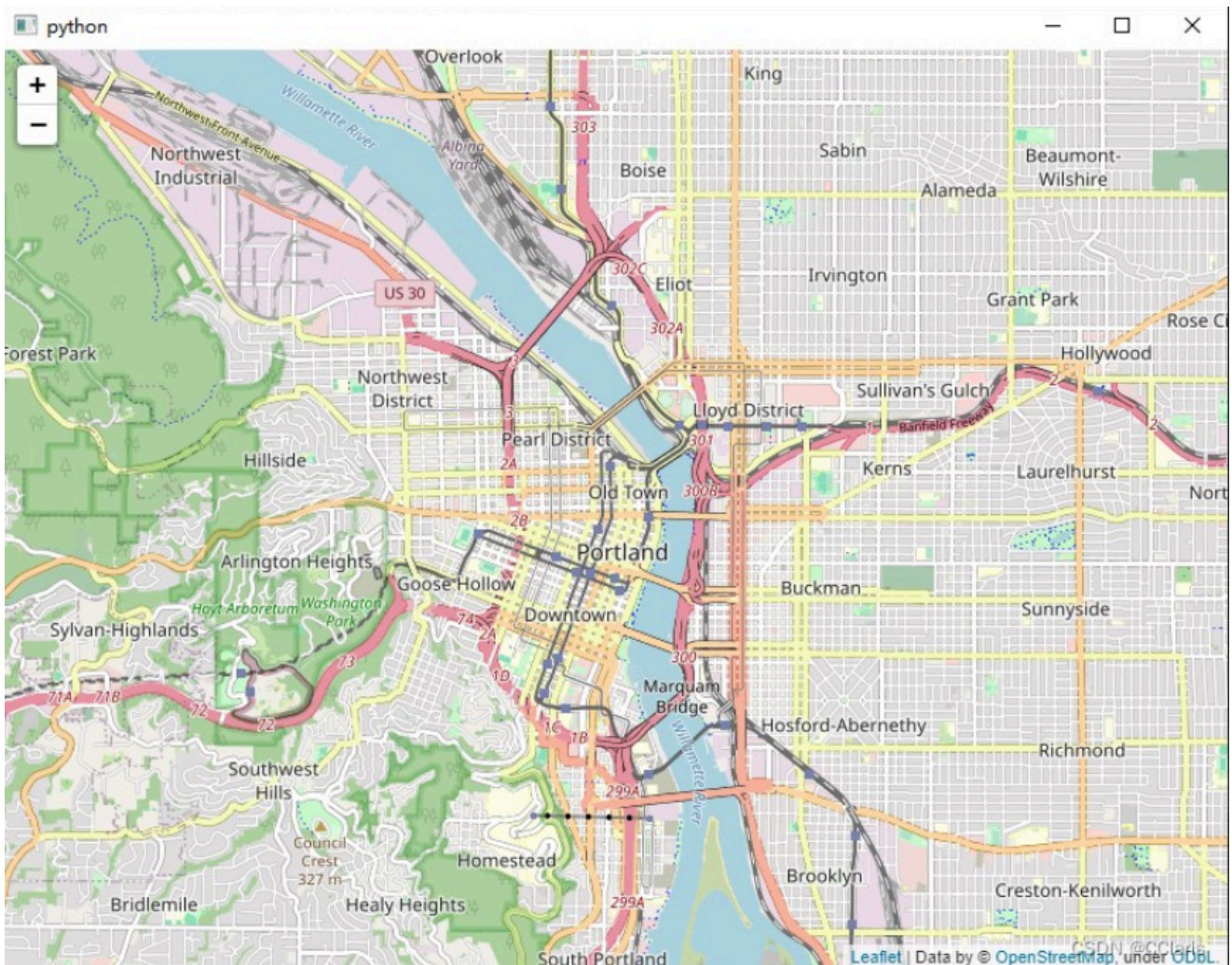
app = QApplication(sys.argv)
mainWindow = QMainWindow()
mapWidget = MapWidget()

# 设置主窗口的初始大小
mainWindow.resize(800, 600) # 可以根据需要调整这个尺寸

mainWindow.setCentralWidget(mapWidget)
mainWindow.show()
sys.exit(app.exec_())

```





在这个示例中，我们使用 Folium 创建了一个简单的地图，并将其保存为一个 HTML 文件。然后我们使用 PyQt5 的 QWebEngineView 小部件加载并显示这个 HTML 文件，从而在 PyQt 应用程序中嵌入了地图。

## 10.14 桑基图

桑基图是一种特殊的流程图，它用于表示数据流和传输。在 PyQtGraph 中直接绘制桑基图并不支持，因为它主要用于绘制二维和三维的数据可视化图形，如折线图、散点图等。然而，您可以使用其他的 Python 库，如 matplotlib，来创建桑基图，并将其嵌入到 PyQt 应用程序中。

以下是一个使用 Python、PyQt5 和 Matplotlib 创建桑基图的基本示例：

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QVBoxLayout, QWidget
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
import matplotlib.pyplot as plt
from matplotlib.sankey import Sankey

class SankeyWidget(QWidget):
    def __init__(self, parent=None):
        super(SankeyWidget, self).__init__(parent)
        self.figure, self.ax = plt.subplots()
        self.canvas = FigureCanvas(self.figure)

        layout = QVBoxLayout()
        layout.addWidget(self.canvas)
        self.setLayout(layout)

        self.draw_sankey()
```

```

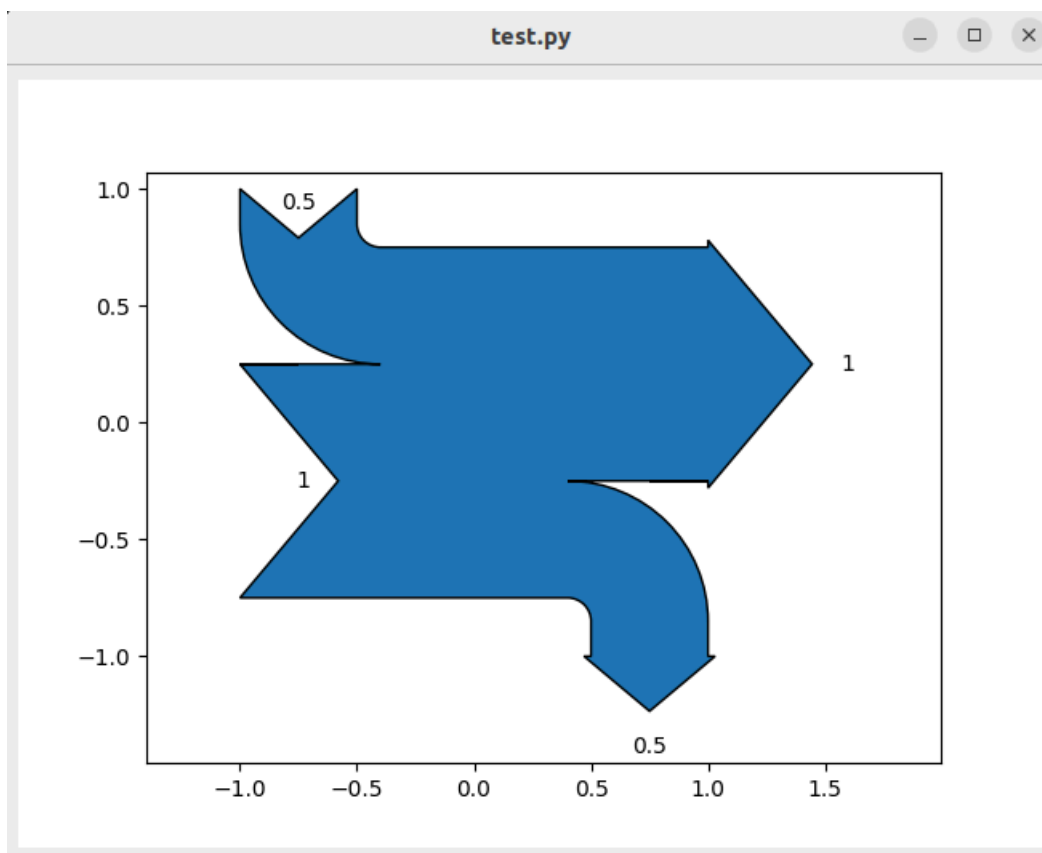
def draw_sankey(self):
    # 创建桑基图
    sankey = Sankey(ax=self.ax)
    sankey.add(flows=[1, -1, 0.5, -0.5], labels=['', '', '', ''], orientations=[0, 0, 1,
-1])

    sankey.finish()

    # 更新画布
    self.canvas.draw()

app = QApplication(sys.argv)
mainWindow = QMainWindow()
mainWindow.setCentralWidget(SankeyWidget())
mainWindow.show()
sys.exit(app.exec_())

```



在这个示例中，我们定义了一个 `SankeyWidget` 类来展示桑基图。使用 `Matplotlib` 创建桑基图，并将其嵌入到 `PyQt5` 应用程序中

### 10.15 组合图

在 `PyQtGraph` 中创建组合图（例如，将折线图和柱状图组合在一起）可以通过在同一个绘图控件（`PlotWidget`）中添加多个图表元素来实现。这样，您可以在同一个坐标轴上或在相互关联的坐标轴上展示不同类型的数据。

以下是一个示例，展示如何在 `PyQtGraph` 中创建一个组合了折线图和柱状图的组合图：

```

import sys
import numpy as np
from PyQt5 import QtWidgets
import pyqtgraph as pg

# 创建一个应用程序实例

```

```

app = QtWidgets.QApplication(sys.argv)

# 创建一个主窗口
mainWindow = QtWidgets.QMainWindow()

# 创建一个绘图控件
plotWidget = pg.PlotWidget()
plotWidget.setBackground('w')
mainWindow.setCentralWidget(plotWidget)

# 准备数据
x = np.arange(10)
y1 = np.random.randint(1, 20, size=10) # 柱状图的数据
y2 = np.random.randint(1, 20, size=10) # 折线图的数据

# 创建柱状图项目
barGraphItem = pg.BarGraphItem(x=x, height=y1, width=0.6, brush='#ffffac8')
plotWidget.addItem(barGraphItem)

# 创建折线图项目
plotWidget.plot(x, y2, pen=pg.mkPen(color='#dcbfff', width=2))

# 显示主窗口
mainWindow.show()

# 启动应用程序的事件循环
sys.exit(app.exec_())

```



在这个示例中，我们首先创建了一个包含柱状图和折线图的 `PlotWidget`。`BarGraphItem` 用于柱状图，而 `plot` 方法用于折线图。通过将它们添加到同一个 `PlotWidget` 中，我们可以将两种图表组合在一起显示。

## 10.16 股票图

在 PyQtGraph 中绘制股票图（通常指蜡烛图或OHLC图）需要一些额外的步骤，因为这需要特殊的图表来表示股票的开盘价、最高价、最低价和收盘价。PyQtGraph 提供了 `CandlestickItem` 类来创建这种类型的图表。

以下是使用 PyQtGraph 创建股票图的步骤和示例代码：

```
import sys
import numpy as np
from PyQt5 import QtWidgets
import pyqtgraph as pg

class CandlestickItem(pg.GraphicsObject):
    def __init__(self, data):
        pg.GraphicsObject.__init__(self)
        self.data = data # data应该是一个包含(open, high, low, close)的列表
        self.generatePicture()

    def generatePicture(self):
        self.picture = pg.QtGui.QPicture()
        p = pg.QtGui.QPainter(self.picture)
        p.setPen(pg.mkPen('b')) # 设置蜡烛图的颜色
        w = 0.3
        for (t, open, high, low, close) in self.data:
            p.drawLine(pg.QtCore.QPointF(t, low), pg.QtCore.QPointF(t, high))
            if open > close:
                p.setBrush(pg.mkBrush('g'))
            else:
                p.setBrush(pg.mkBrush('r'))
            p.drawRect(pg.QtCore.QRectF(t - w, open, w * 2, close - open))
        p.end()

    def paint(self, p, *args):
        p.drawPicture(0, 0, self.picture)

    def boundingRect(self):
        return pg.QtCore.QRectF(self.picture.boundingRect())

# 创建一个应用程序实例
app = QtWidgets.QApplication(sys.argv)

# 创建一个主窗口
mainWindow = QtWidgets.QMainWindow()

# 创建一个绘图控件
plotWidget = pg.PlotWidget()
plotWidget.setBackground("w")
mainWindow.setCentralWidget(plotWidget)

# 准备数据
data = [ # 示例数据: (时间, 开盘价, 最高价, 最低价, 收盘价)
        (1, 10, 15, 5, 13),
        (2, 13, 20, 10, 15),
        (3, 15, 17, 13, 16),
        (4, 16, 19, 14, 10),
        # ...
```

```

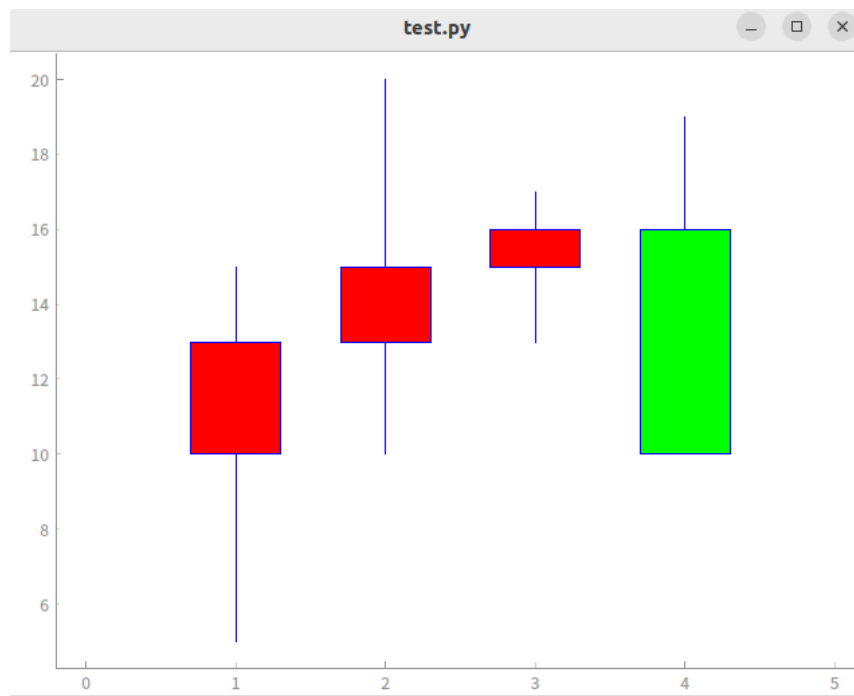
]

# 创建蜡烛图项目
item = CandlestickItem(data)
plotWidget.addItem(item)

# 显示主窗口
mainWindow.show()

# 启动应用程序的事件循环
sys.exit(app.exec_())

```



在这个示例中，我们首先定义了一个 `CandlestickItem` 类来创建股票图。该类继承自 `GraphicsObject`，并且重写了绘图方法来绘制蜡烛图。数据格式是一个元组列表，每个元组代表一个时间点上的开盘价、最高价、最低价和收盘价。

### 10.17 甘特图

在 `PyQtGraph` 中创建甘特图需要一些额外的工作，因为它主要用于绘制二维和三维的数据可视化图形，如折线图、散点图等，而不直接支持甘特图。但是，您可以使用 `PyQt` 的基本图形功能来手动绘制甘特图。

以下是使用 `PyQt5` 创建简单甘特图的示例代码：

```

import sys
from PyQt5.QtWidgets import QApplication, QGraphicsScene, QGraphicsView, QGraphicsRectItem,
    QGraphicsTextItem
from PyQt5.QtGui import QColor, QPen, QFont
from PyQt5.QtCore import QRectF

class GanttChartItem(QGraphicsRectItem):
    def __init__(self, text, start, duration, row, color, parent=None):
        super(GanttChartItem, self).__init__(parent)
        # 设置矩形的尺寸和位置
        self.setRect(QRectF(start, row * 50, duration, 30))
        self.setBrush(QColor(color))
        self.setPen(QPen(QColor('black'), 2)) # 添加边框

```

```

# 添加文本标签
text_item = QGraphicsTextItem(text, self)
text_item.setFont(QFont('Arial', 10))
text_item.setDefaultTextColor(QColor('black'))
text_item.setPos(start + 2, row * 50 + 5)

app = QApplication(sys.argv)

# 创建场景
scene = QGraphicsScene()

# 创建甘特图条目
gantItem1 = GanttChartItem("项目1", 0, 100, 0, '#78C0A8')
gantItem2 = GanttChartItem("项目2", 50, 150, 1, '#F4D35E')
gantItem3 = GanttChartItem("项目3", 120, 100, 2, '#EE964B')

# 将条目添加到场景
scene.addItem(gantItem1)
scene.addItem(gantItem2)
scene.addItem(gantItem3)

# 创建视图
view = QGraphicsView(scene)

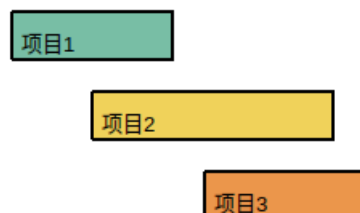
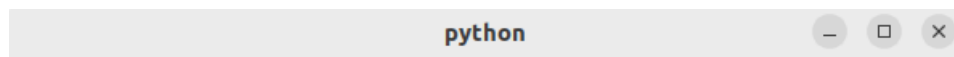
# 设置合适的视图尺寸
view.setSceneRect(0, 0, 400, 200)
view.setMinimumSize(600, 400)

# 设置窗口标题
view.setWindowTitle("python")

# 显示视图窗口
view.show()

sys.exit(app.exec_())

```



在这个示例中，我们定义了一个 `GanttChartItem` 类来表示甘特图中的单个任务。每个任务在水平方向上表示其持续时间，在垂直方向上表示不同的任务行。`QGraphicsScene` 用于容纳这些任务，而 `QGraphicsView` 用于显示它们。

## 10.18 平行坐标图

在 `PyQtGraph` 中直接绘制平行坐标图 (Parallel Coordinates) 并不直接支持，因为 `PyQtGraph` 主要用于绘制二维和三维的数据可视化图形，如折线图、散点图等。然而，您可以使用其他的Python库，如 `matplotlib` 或 `pandas`，来创建平行坐标图，并将其嵌入到PyQt应用程序中。

以下是一个使用 Python、PyQt5 和 Matplotlib 创建平行坐标图的基本示例：

```
import sys
import pandas as pd
import matplotlib.pyplot as plt
from PyQt5.QtWidgets import QApplication, QMainWindow, QVBoxLayout, QWidget
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from pandas.plotting import parallel_coordinates

class ParallelCoordinatesWidget(QWidget):
    def __init__(self, parent=None):
        super(ParallelCoordinatesWidget, self).__init__(parent)
        self.figure, self.ax = plt.subplots()
        self.canvas = FigureCanvas(self.figure)

        layout = QVBoxLayout()
        layout.addWidget(self.canvas)
        self.setLayout(layout)

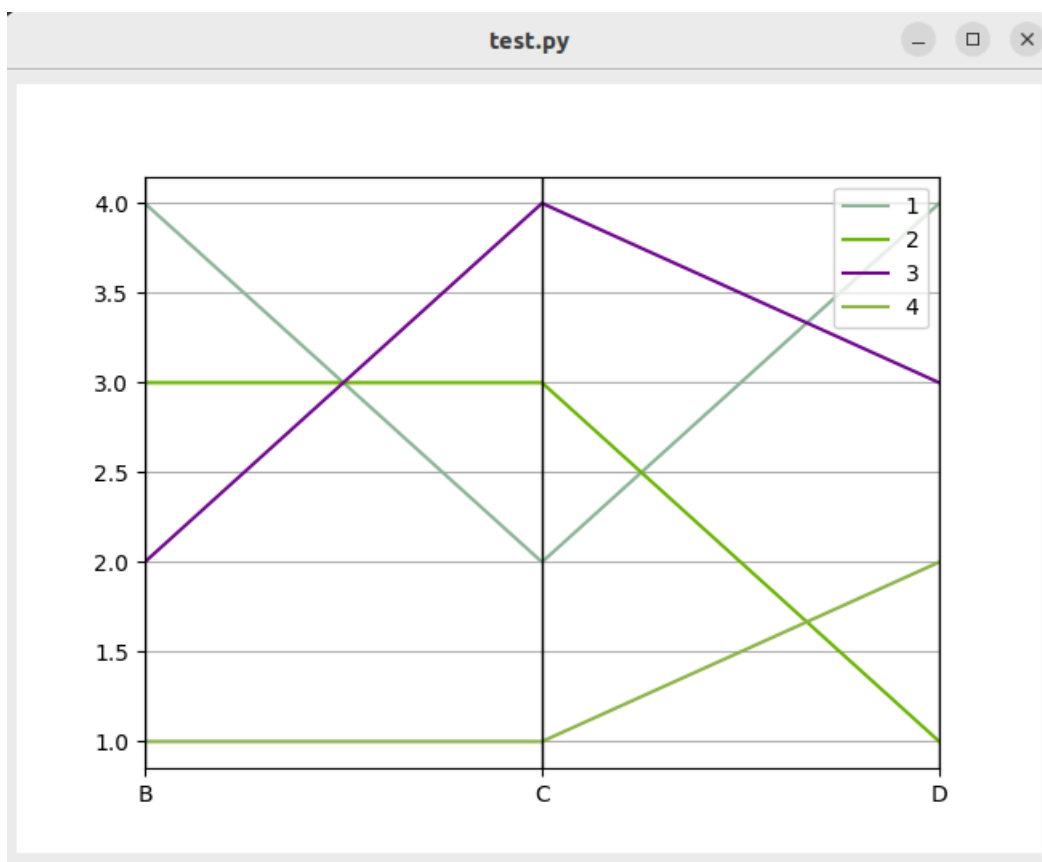
        self.draw_parallel_coordinates()

    def draw_parallel_coordinates(self):
        # 创建数据
        data = pd.DataFrame({
            'A': [1, 2, 3, 4],
            'B': [4, 3, 2, 1],
            'C': [2, 3, 4, 1],
            'D': [4, 1, 3, 2]
        })

        # 绘制平行坐标图
        parallel_coordinates(data, 'A', ax=self.ax)

        # 更新画布
        self.canvas.draw()

app = QApplication(sys.argv)
mainWindow = QMainWindow()
mainWindow.setCentralWidget(ParallelCoordinatesWidget())
mainWindow.show()
sys.exit(app.exec_())
```



在这个示例中，我们定义了一个 `ParallelCoordinatesWidget` 类来展示平行坐标图。使用 `Pandas` 和 `Matplotlib` 创建平行坐标图，并将其嵌入到 `PyQt5` 应用程序中。

## 11.其他

### 11.1 QMessageBox-消息提醒框

`QMessageBox`类提供了一个模态对话框，用于通知用户或向用户提问并接收答案。

函数简介：用于显示消息框、询问框、警告框等用户交互提示框的类。

#### ⚠ Caution

函数说明：

信息消息框 <code>defaultButton()</code>	<code>QMessageBox.information(parent, title, message, buttons,</code>
询问消息框	<code>QMessageBox.question(parent, title, message, buttons, defaultButton)</code>
警告消息框	<code>QMessageBox.warning(parent, title, message, buttons, defaultButton)</code>
严重错误消息框	<code>QMessageBox.critical(parent, title, message, buttons, defaultButton)</code>

输入参数：

<code>parent:</code>	可选参数，父级窗口。
<code>title:</code>	消息框的标题。
<code>message:</code>	消息框中显示的消息文本。
<code>buttons:</code>	消息框中显示的按钮类型，如 <code>QMessageBox.Yes</code> 、 <code>QMessageBox.No</code> 等。
<code>defaultButton:</code>	可选参数，指定默认按钮。

## 案例

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QPushButton, QVBoxLayout, QWidget,
QMessageBox
```



```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        info_button = QPushButton("Information")
        question_button = QPushButton("Question")
        warning_button = QPushButton("Warning")
        critical_button = QPushButton("Critical")
        info_button.clicked.connect(self.show_information)
        question_button.clicked.connect(self.show_question)
        warning_button.clicked.connect(self.show_warning)
        critical_button.clicked.connect(self.show_critical)

        layout = QVBoxLayout()
        layout.addWidget(info_button)
        layout.addWidget(question_button)
        layout.addWidget(warning_button)
        layout.addWidget(critical_button)

        widget = QWidget()
        widget.setLayout(layout)
        self.setCentralWidget(widget)

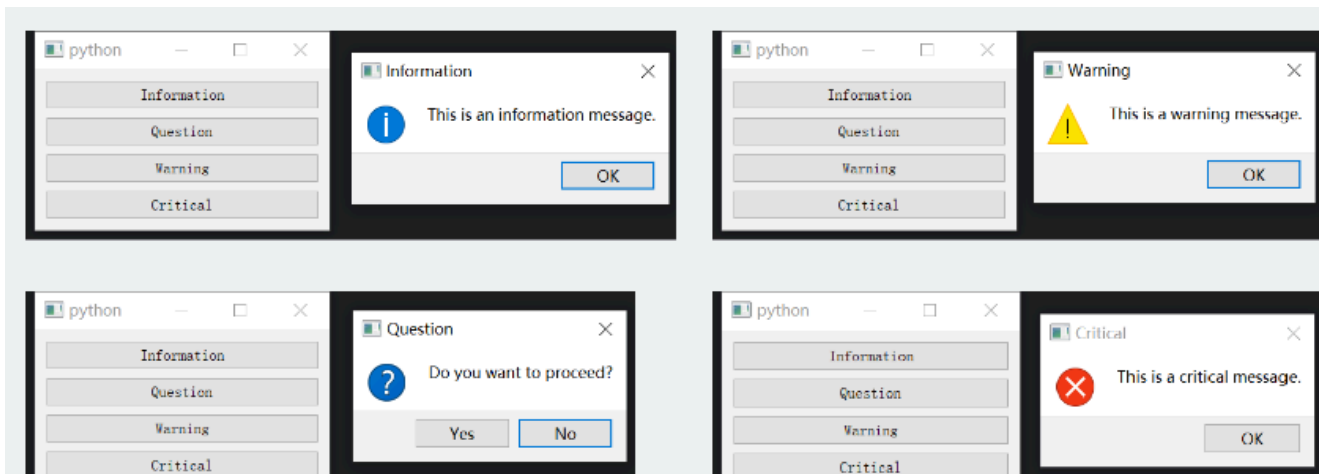
    def show_information(self):
        QMessageBox.information(self, "Information", "This is an information message.",
                                QMessageBox.Ok, QMessageBox.Ok)

    def show_question(self):
        result = QMessageBox.question(self, "Question", "Do you want to proceed?",
                                     QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
        if result == QMessageBox.Yes:
            print("User clicked Yes")
        else:
            print("User clicked No")

    def show_warning(self):
        QMessageBox.warning(self, "Warning", "This is a warning message.", QMessageBox.Ok,
                            QMessageBox.Ok)

    def show_critical(self):
        QMessageBox.critical(self, "Critical", "This is a critical message.", QMessageBox.Ok,
                             QMessageBox.Ok)

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



## 11.2 QWebEngineView-Web浏览器引擎

QWebEngineView是Qt框架中的一个组件，它是基于Chromium内核的Web浏览器引擎，用于在Qt应用程序中嵌入网页内容和实现各种Web应用功能。通过QWebEngineView，开发者可以在本地桌面应用程序中轻松地集成网页浏览功能，支持HTML5、CSS3、JavaScript等现代Web技术

```
# 创建 WebView 并加载页面
self.webview = QWebEngineView(self.centralwidget)
self.webview.setGeometry(QtCore.QRect(0, 0, 1920, 1050))
self.webview.setObjectName("webview")

# 启用远程 URL 访问、JavaScript 支持等

self.webview.settings().setAttribute(QWebEngineSettings.WebAttribute.LocalContentCanAccessRemote
Urls, True)

self.webview.settings().setAttribute(QWebEngineSettings.WebAttribute.LocalStorageEnabled, True)
self.webview.settings().setAttribute(QWebEngineSettings.WebAttribute.JavascriptEnabled,
True)

html_file_path = QFileInfo("Page2.html").absoluteFilePath()
self.webview.setUrl(QUrl.fromLocalFile(html_file_path))
```