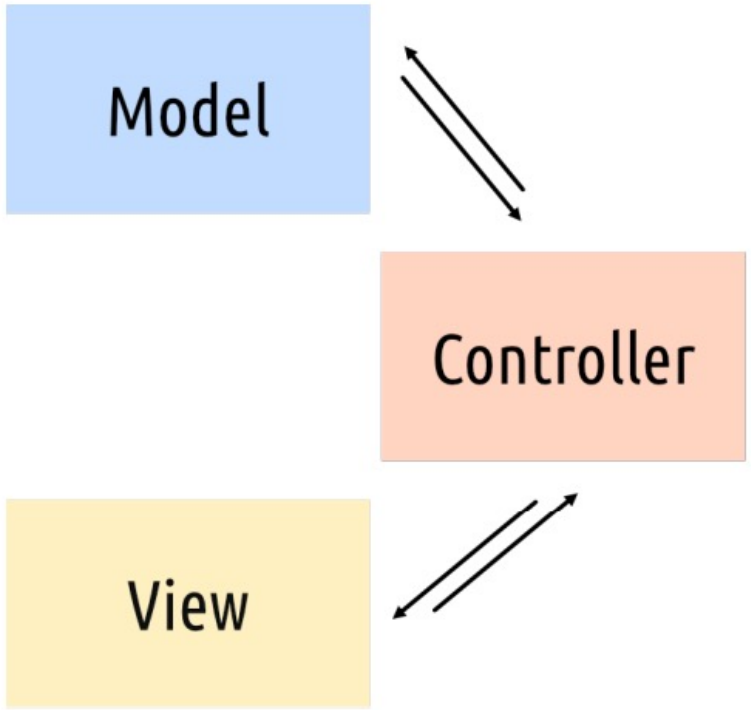
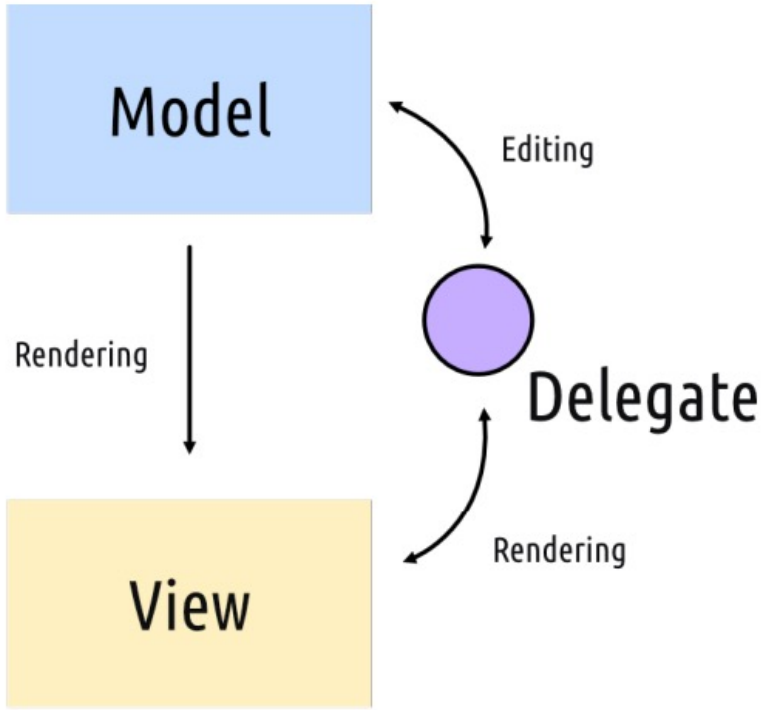
Lecture 15: PyQT (3)

The Model/View Programming



Model View Controller architecture



Qt's Model/Views architecture

The Model/View Programming (2)

- Qt contains a set of item view classes that use a model/view architecture to manage the relationship between data and the way it is presented to the user.
- The separation of functionality introduced by this architecture gives developers greater flexibility to customize the presentation of items, and provides a standard model interface to allow a wide range of data sources to be used with existing item views.

The Model/View Programming (3)

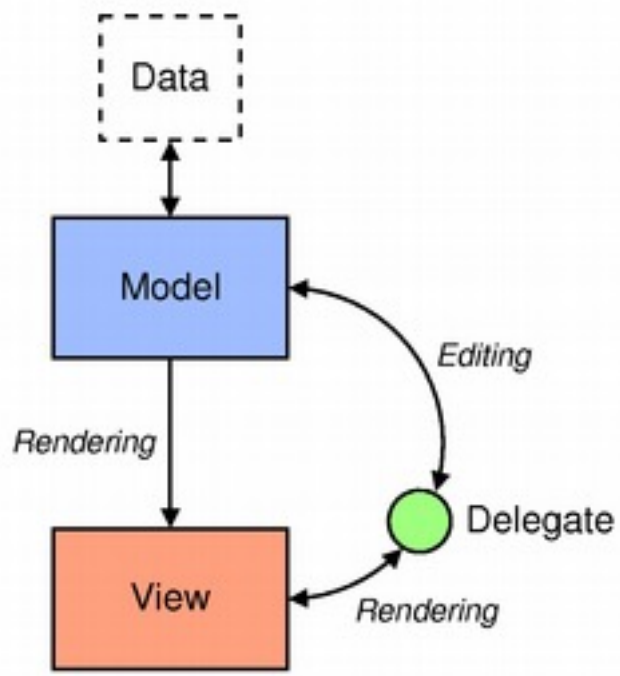
- The model/view architecture
- Model-View-Controller (MVC) is a design pattern originating from Smalltalk that is often used when building user interfaces.
- MVC consists of three kinds of objects. The Model is the application object,
  - the View is its screen presentation, and
  - the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

The Model/View Programming (4)

- If the view and the controller objects are combined, the result is the model/view architecture. This still separates the way that data is stored from the way that it is presented to the user, but provides a simpler framework based on the same principles.
- This separation makes it possible to display the same data in several different views, and to implement new types of views, without changing the underlying data structures.
- To allow flexible handling of user input, we introduce the concept of the *delegate*. The advantage of having a delegate in this framework is that it allows the way items of data are rendered and edited to be customized.

The Model/View Programming (5)

- **The model/view architecture**
- The model communicates with a source of data, providing an *interface* for the other components in the architecture. The nature of the communication depends on the type of data source, and the way the model is implemented.
- The view obtains *model indexes* from the model; these are references to items of data. By supplying model indexes to the model, the view can retrieve items of data from the data source.
- In standard views, a *delegate* renders the items of data. When an item is edited, the delegate communicates with the model directly using model indexes.



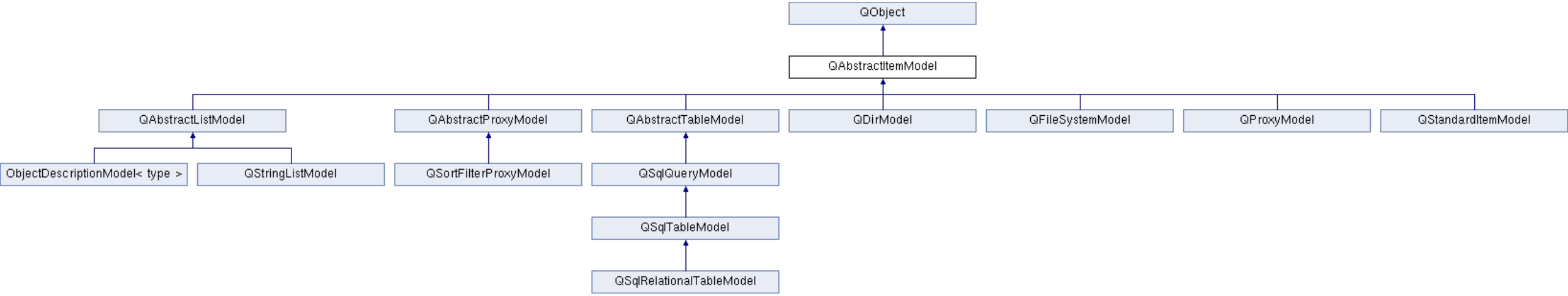
The Model/View Programming (6)

- Models, views, and delegates communicate with each other using *signals and slots*:
  - Signals from the model inform the view about changes to the data held by the data source.
  - Signals from the view provide information about the user's interaction with the items being displayed.
  - Signals from the delegate are used during editing to tell the model and view about the state of the editor.

Models

- All item models are based on the QAbstractItemModel class. This class defines an interface that is used by views and delegates to access data.
- The data itself does not have to be stored in the model; it can be held in a data structure or repository provided by a separate class, a file, a database, or some other application component.

Model (2)


QObject
QAbstractItemModel
QAbstractListModel
QAbstractProxyModel
QAbstractTableModel
QDirModel
QFileSystemModel
QProxyModel
QStandardItemModel
ObjectDescriptionModel< type >
QStringListModel
QSortFilterProxyModel
QSqlQueryModel
QSqlTableModel
QSqlRelationalTableModel

Model (3)

- QAbstractItemModel provides an interface to data that is flexible enough to handle views that represent data in the form of tables, lists, and trees.
- However, when implementing new models for list and table-like data structures,
  the QAbstractListModel and QAbstractTableModel classes are better starting points because they provide appropriate default implementations of common functions.
- Each of these classes can be subclassed to provide models that support specialized kinds of lists and tables.

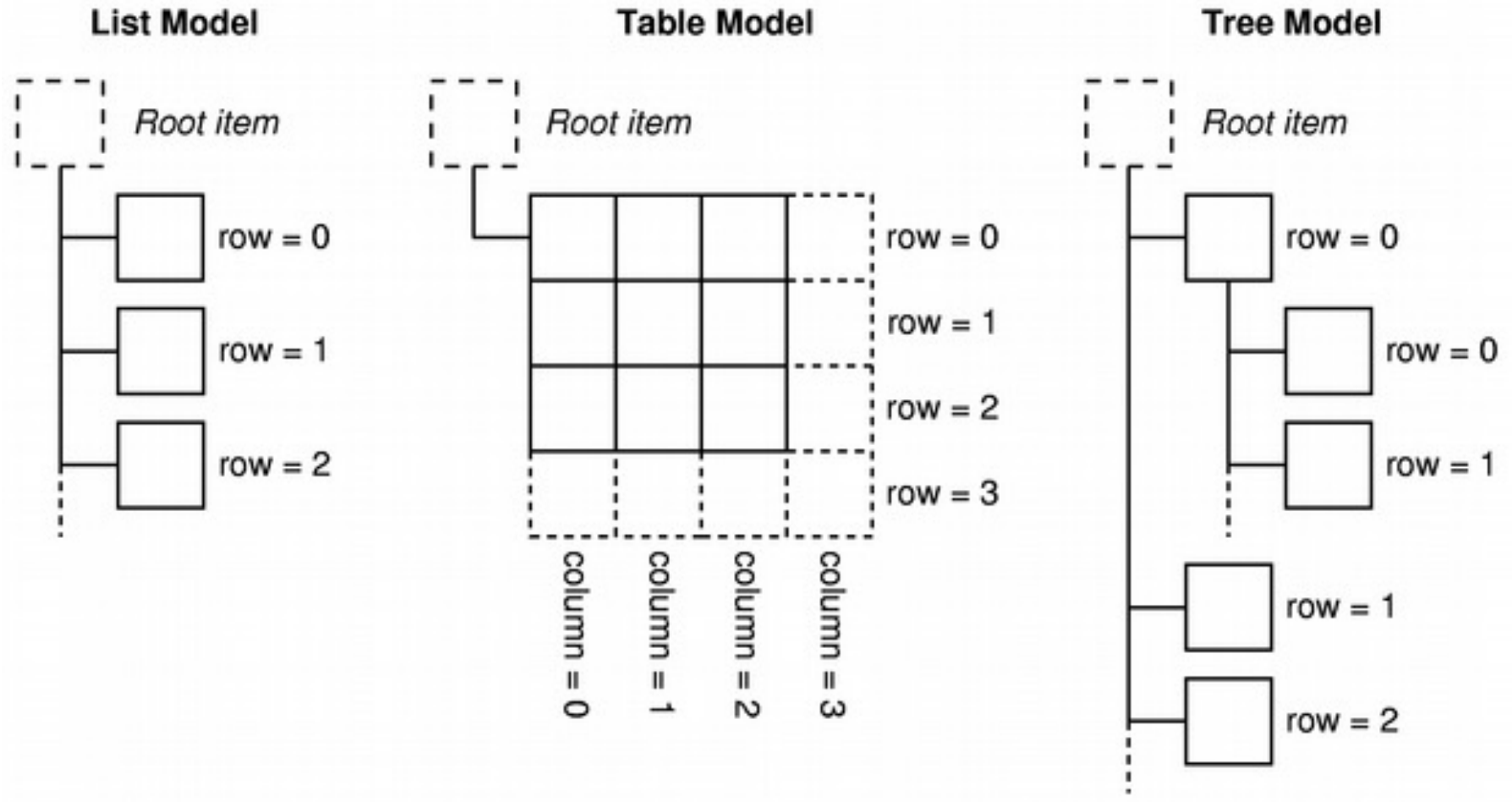
Model (4)

- Qt provides some ready-made models that can be used to handle items of data:
- QStringListModel is used to store a simple list of QString items.
- QStandardItemModel manages more complex tree structures of items, each of which can contain arbitrary data.
- QFileSystemModel provides information about files and directories in the local filing system.
- QSqlQueryModel , QSqlTableModel , and QSqlRelationalTableModel are used to access databases using model/view conventions.

Model & View: Concepts

- In the model/view architecture, the model provides a standard interface that views and delegates use to access data.
- In Qt, the standard interface is defined by the QAbstractItemModel class. No matter how the items of data are stored in any underlying data structure, all subclasses of QAbstractItemModel represent the data as a hierarchical structure containing tables of items.
- Views use this *convention* to access items of data in the model, but they are not restricted in the way that they present this information to the user.

Model & View: Concepts (2)



Model Indexes

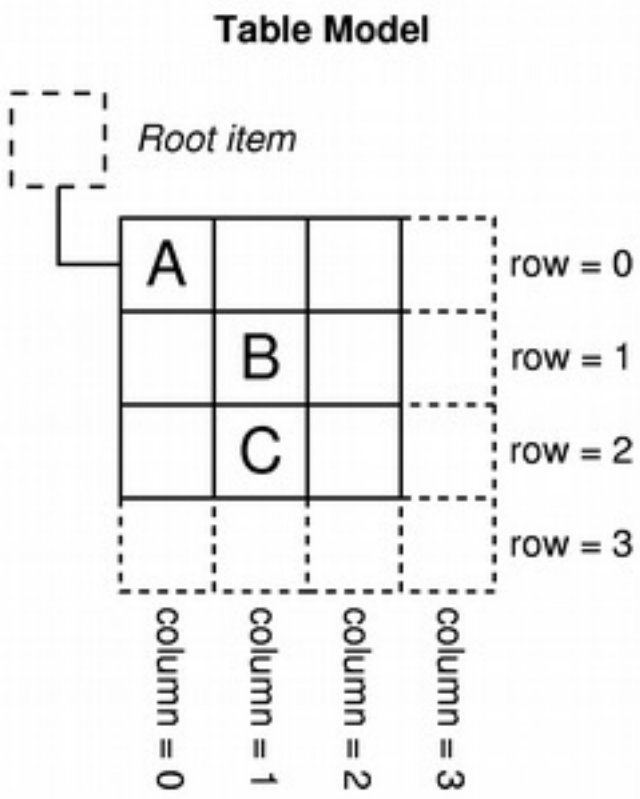
- To ensure that the representation of the data is kept separate from the way it is accessed, the concept of a *model index* is introduced.
- Each piece of information that can be obtained via a model is represented by a model index. Views and delegates use these indexes to request items of data to display.
- As a result, only the model needs to know how to obtain data, and the type of data managed by the model can be defined fairly generally. Model indexes contain a pointer to the model that created them, and this prevents confusion when working with more than one model.

Model Indexes (2)

- Model indexes provide *temporary* references to pieces of information, and can be used to retrieve or modify data via the model

Model Indexes: Row and Columns

- We obtain a model index that refers to an item of data by passing the relevant row and column numbers to the model.
  - indexA = model.index(0, 0, QModelIndex())
  - indexB = model.index(1, 1, QModelIndex())
  - indexC = model.index(2, 1, QModelIndex())

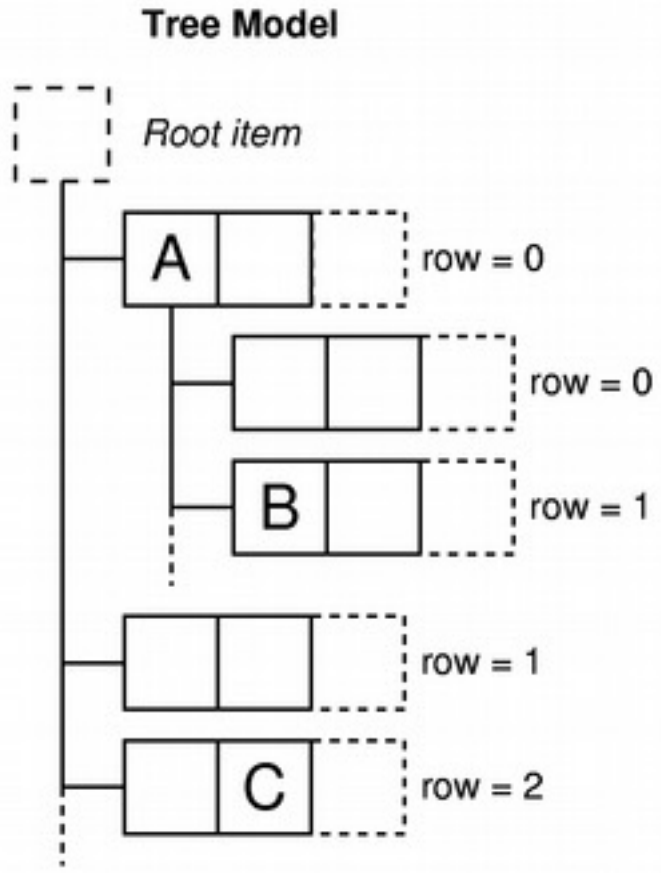


- Top level items in a model are always referenced by specifying QModelIndex() as their parent item.

Parents of items

- The diagram shows a representation of a tree model in which each item is referred to by a parent, a row number, and a column number.

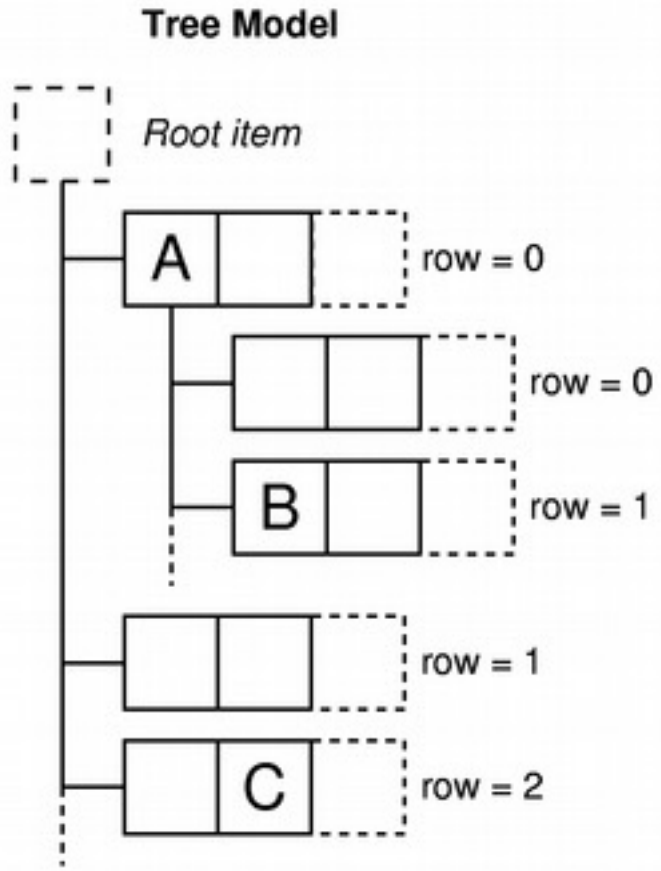
- Items “A” and “C” are represented as top-level siblings in the model:
- indexA = model.index(0, 0, QModelIndex())
- indexC = model.index(2, 1, QModelIndex())



Parents of items (2)

- Item “A” has a number of children. A model index for item “B” is obtained with the following code:

- indexB = model.index(1,0,indexA)



Views

- Complete implementations are provided for different kinds of views:
- QListView displays a list of items, QTableView displays data from a model in a table, and QTreeView shows model items of data in a hierarchical list. Each of these classes is based on the QAbstractItemView abstract base class.
- Although these classes are ready-to-use implementations, they can also be subclassed to provide customized views.

QFileSystemModel

- The QFileSystemModel class provides a data model for the local filesystem.
- This class provides access to the local filesystem, providing functions for renaming and removing files and directories, and for creating new directories.
- In the simplest case, it can be used with a suitable display widget as part of a browser or filter.

QFileSystemModel (2)

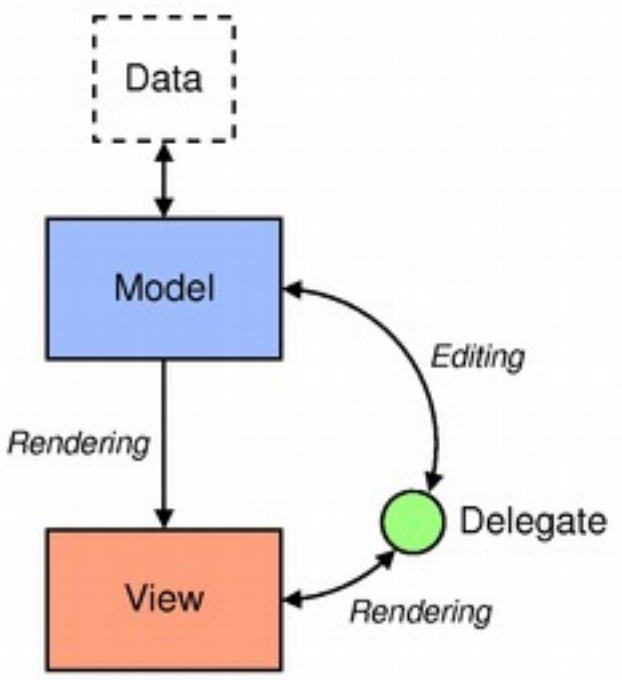
- QFileSystemModel can be accessed using the standard interface provided by QAbstractItemModel, but it also provides some convenience functions that are specific to a directory model. The fileInfo(), isDir(), fileName() and filePath() functions provide information about the underlying files and directories related to items in the model. Directories can be created and removed using mkdir(), rmdir().

- **Note:** QFileSystemModel requires an instance of QApplication.

QFileSystemModel (3)

- The use of QFileSystemModel:
- model = QFileSystemModel()
- model.setRootPath(Qdir.currentPath())

- window.treeView.setModel(model) or
- window.listView.setModel(model) or
- window.tableView.setModel(model)

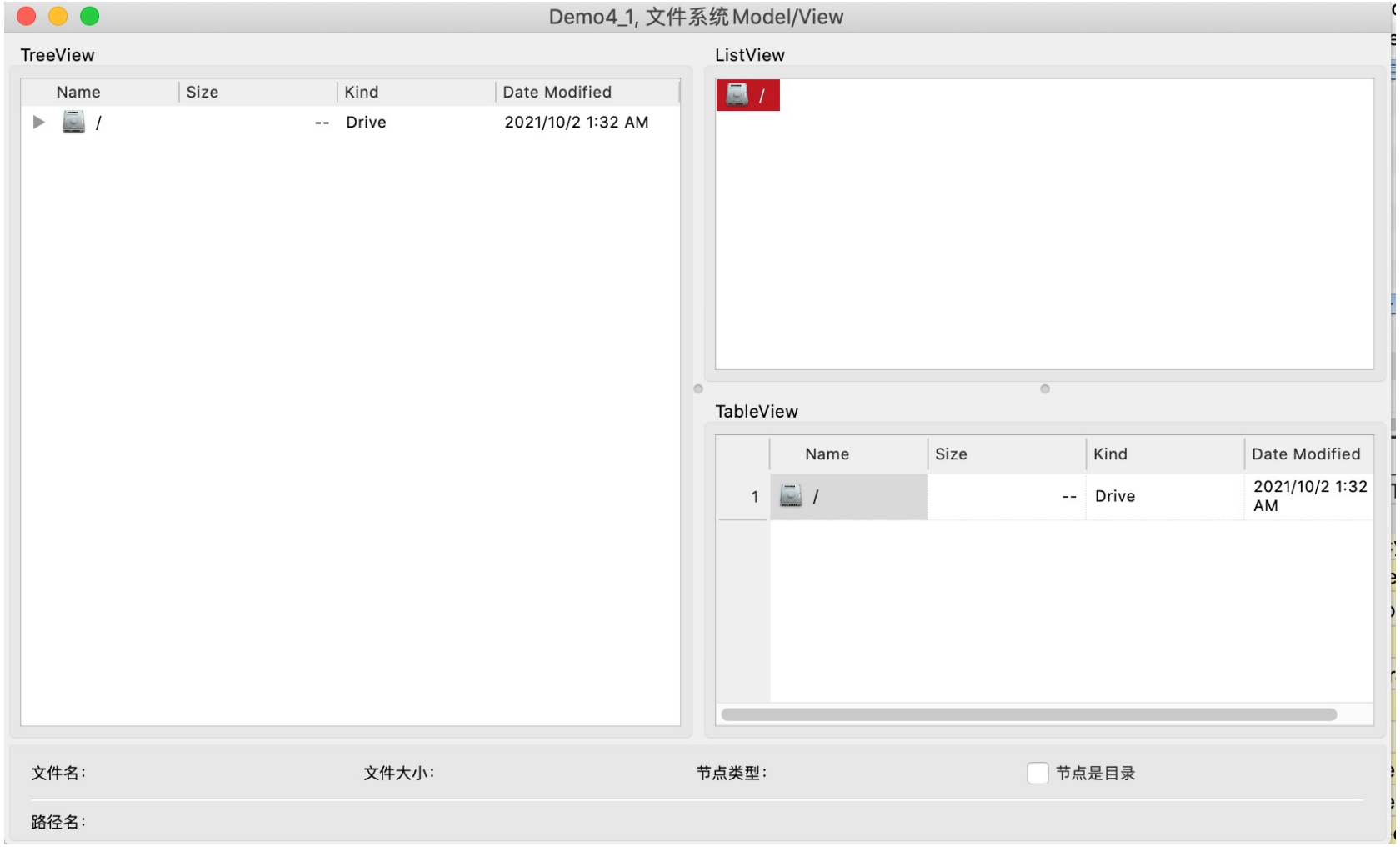


QFileSystemModel (4)

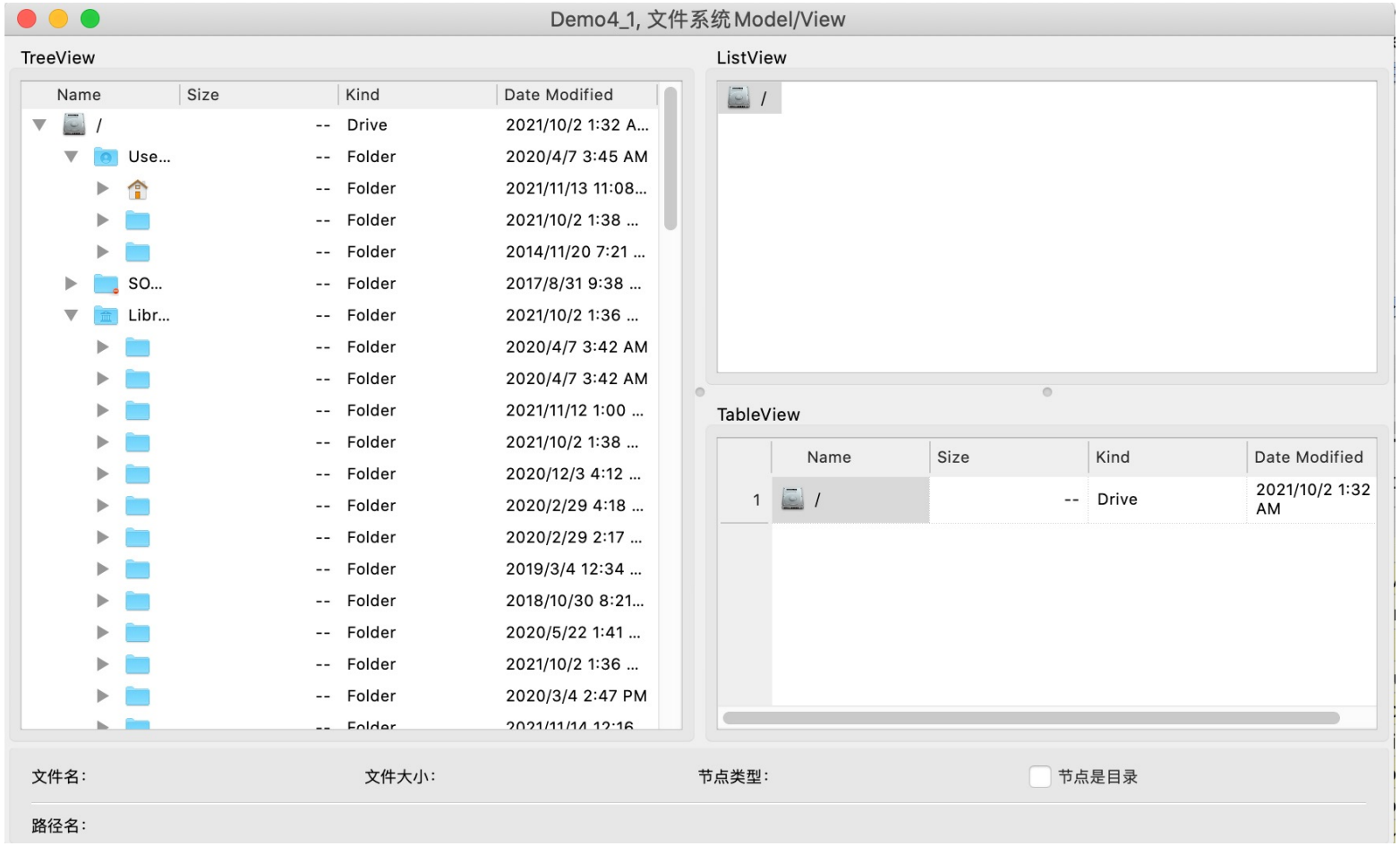
- **if** __name__ == '**__main__**':

```
    app = QApplication(sys.argv)

    window = uic.loadUi("MainWindow.ui")

    model = QFileSystemModel()
    model.setRootPath(QDir.currentPath())

    window.treeView.setModel(model)  # 设置数据模型
    window.listView.setModel(model)
    window.tableView.setModel(model)

    window.show()

    app.exec()
```

QFileSystemModel (5)



QFileSystemModel (6)



QFileSystemModel (7)

- A tree view can be used to display the contents of the model:
  - treeView.setModel (model)

QStringListModel

- QStringListModel is an editable model that can be used for simple cases where you need to display a number of strings in a view widget, such as a QListView or a QComboBox.

- The model provides all the standard functions of an editable model, representing the data in the string list as a model with one column and a number of rows equal to the number of items in the list.

QStringListModel (2)

- Model indexes corresponding to items are obtained with the index() function, and item flags are obtained with flags(). Item data is read with the data() function and written with setData(). The number of rows (and number of items in the string list) can be found with the rowCount() function.

- The model can be constructed with an existing string list, or strings can be set later with the setStringList() convenience function. Strings can also be inserted in the usual way with the insertRows() function, and removed with removeRows(). The contents of the string list can be retrieved with the stringList() convenience function.

QStringListModel (3)

- **class** QmyWidget(QWidget):
  def __init__(self, parent=**None**):
  super().__init__(parent) *# 调用父类构造函数，创建窗体*
  self.ui = Ui_Widget() *# 创建UI对象*
  self.ui.setupUi(self) *# 构造UI界面*
  self.__provinces = ["北京", "上海", "天津", "河北",
  "山东", "四川", "重庆", "广东", "河南"]
-
  self.model = QStringListModel() # 创建ListModel
  self.model.setStringList(self.__provinces) # 设置内容
  self.ui.listView.setModel(self.model)

  self.ui.listView.setEditTriggers(QAbstractItemView.EditTrigger.DoubleClicked
  | QAbstractItemView.EditTrigger.SelectedClicked) # 当双击或选择后单击可以修改

QStringListModel (4)

QStringListModel (5)

- Setup
  - **def** __init__(self, parent=**None**):

```
super().__init__(parent)  # 调用父类构造函数，创建窗体
self.ui = Ui_Widget()  # 创建UI对象
self.ui.setupUi(self)  # 构造UI界面
self.__provinces = ["北京", "上海", "天津", "河北",
                    "山东", "四川", "重庆", "广东", "河南"]
self.model = QStringListModel() # 创建ListModel
self.model.setStringList(self.__provinces) # 设置内容
self.ui.listView.setModel(self.model)
```

QStringListModel (6)

- Set editable:
  - Self.ui.listView.setEditTriggers(QAbstractItemView.EditTrigger.DoubleClicked| QAbstractItemView.EditTrigger.SelectedClicked) # 当双击或选择后单击可以修改
- Add: # 加入list尾部
  - lastRow = self.model.rowCount()
  - self.model.insertRow(lastRow) # 在Model中加入一个空行
  - index = self.model.index(lastRow,0), # 获取最后一行的ModelIndex
  - #这里 index的类型不是int而是modelIndex
  - self.model.setData(index,“新加入的行”)
  - self.ui.listView.setCurrentIndex(index); 设置当前选中的行

QStringListModel (7)

- Insert # 插入（在任意位置插入）

```
def on_btnList_Insert_clicked(self):
    index = self.ui.listView.currentIndex()  # 当前 modelIndex
    self.model.insertRow(index.row())
    self.model.setData(index, "inserted item")  # 设置显示文字
    self.ui.listView.setCurrentIndex(index)  # 设置当前选中的行
```

QStringListModel (5)

- Delete 删除当前项
- **def** on_btnList_Delete_clicked(self):
  index = self.ui.listView.currentIndex() *# 获取当前 modelIndex*
  self.model.removeRow(index.row()) *# 删除当前行*

QStringListModel (6)

- Clear 清空
  - **def** on_btnList_Clear_clicked(self):
    count = self.model.rowCount()
    self.model.removeRows(0, count)

QStandardItemModel

- QStandardItemModel can be used as a repository for standard Qt data types. It is one of the Model/View Classes and is part of Qt's model/view framework.

- QStandardItemModel provides a classic item-based approach to working with the model. The items in a QStandardItemModel are provided by QStandardItem .

QStandardItemModel (2)

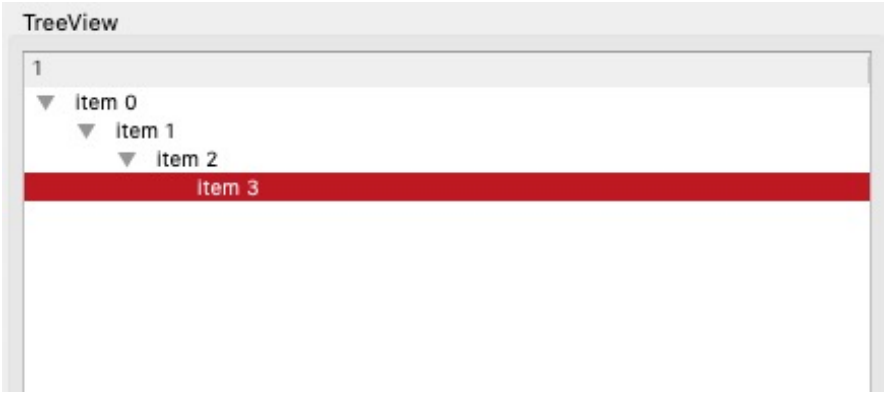
- QStandardItemModel implements the QAbstractItemModel interface, which means that the model can be used to provide data in any view that supports that interface (such as QListView , QTableView and QTreeView , and your own custom views).

QStandardItemModel (3)

- When you want a list or tree, you typically create an empty QStandardItemModeland use appendRow() to add items to the model, and item() to access an item.
- If your model represents a table, you typically pass the dimensions of the table to the QStandardItemModel constructor and use setItem() to position items into the table.
- You can also use setRowCount() and setColumnCount() to alter the dimensions of the model. To insert items, use insertRow() or insertColumn(), and to remove items, use removeRow() or removeColumn().
- You can search for items in the model with findItems(), and sort the model by calling sort().
- Call clear() to remove all items from the model.

QStandardItemModel (4)

- An example usage of QStandardItemModel to create a table:
- model = QStandardItemModel (4, 4)
- for row in range(4):
- for column in range(4):
- item = QStandardItem("row %d, column %d" % (row, column))
- model.setItem(row, column, item)



QStandardItemModel (5)

An example usage of QStandardItemModel to create a tree

- model = QStandardItemModel()
- parentItem = model.invisibleRootItem()
- for i in range(4):
- item = QStandardItem("item %d" % i)
- parentItem.appendRow(item)
- parentItem = item

- window.tableView.setModel(model)

QStandardItemModel (6)

```
class StandardItem:

    def __init__(self):
        self.window = uic.loadUi("MainWindow.ui")

        tablemodel = QStandardItemModel(4, 4)
        for row in range(4):
            for column in range(4):
                item = QStandardItem("row %d, column %d" % (row, column))
                tablemodel.setItem(row, column, item)

        treemodel = QStandardItemModel()
        parentItem = treemodel.invisibleRootItem()
        for i in range(4):
            item = QStandardItem("item %d" % i)
            parentItem.appendRow(item)
            parentItem = item

        self.window.treeView.setModel(treemodel)  # 设置数据模型
        self.window.tableView.setModel(tablemodel)


        self.window.show()
```

QStandardItemModel (7)

- self.window.treeView.clicked.connect(self.on_item_click)

- def on_item_click(self,index): #QModelIndex
  - item = self.treeModel.itemFromIndex (index)
  - # item QStandardItem

  - content = item.text()

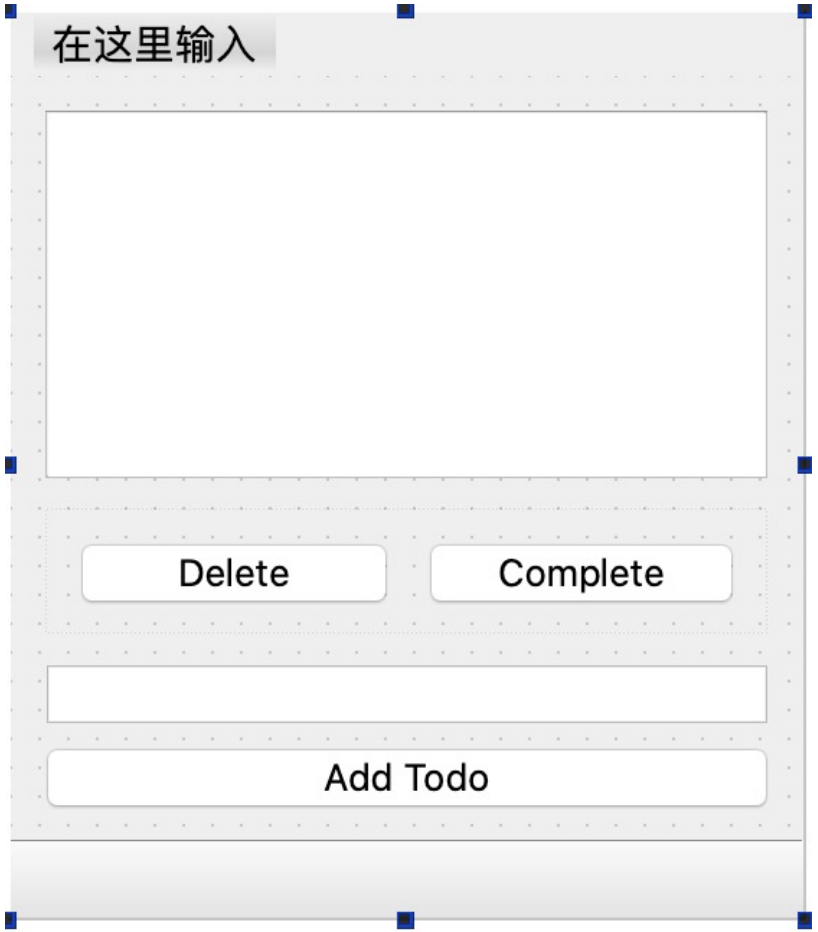
```python
class StandardItem:

    def __init__(self):
        self.window = uic.loadUi("MainWindow.ui")

        self.tablemodel = QStandardItemModel(4, 4)
        for row in range(4):
            for column in range(4):
                item = QStandardItem("row %d, column %d" % (row, column))
                self.tablemodel.setItem(row, column, item)

        self.treemodel = QStandardItemModel()
        parentItem = self.treemodel.invisibleRootItem()
        for i in range(4):
            item = QStandardItem("item %d" % i)
            parentItem.appendRow(item)
            parentItem = item

        self.window.treeView.setModel(self.treemodel)  # 设置数据模型
        self.window.tableView.setModel(self.tablemodel)
        self.window.treeView.clicked.connect(self.on_item_click)

        self.window.show()

    def on_item_click(self,index):
        print(type(index))
        item = self.treemodel.itemFromIndex(index)
        print(type(item))
        print(item.text())


if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = StandardItem()
    app.exec()
```

A Simple Model View --- A TODO list

- To demonstrate how to use the ModelViews in practice, we'll put together a very simple implementation of a desktop Todo list.
- The UI
- The simple UI was laid out using Qt Creator and saved as mainwindow.ui.

A Simple Model View --- A TODO list (2)

- UI



List

Button

LineEdit

Button

A Simple Model View --- A TODO list (3)

- Next, we compile the .ui file into a .py file
- pyuic6 mainwindow.ui –o MainWindow.py
- This generates a MainWindow.py file which contains our custom window class as designed in Qt Designer. This can be imported in our application code as normal—a basic skeleton app to display our UI is shown below

A Simple Model View --- A TODO list (4)

- **class** TODO:
  def __init__(self):
  self.window = QMainWindow()
  ui_form = Ui_MainWindow()
  ui_form.setupUi(self.window)
  self.window.show()

- **if** __name__ == '__**main**__':
  app = QApplication(sys.argv)
  window = TODO()
  app.exec()

A Simple Model View --- A TODO list (5)

- Add

```
def add(self):
    text = self.ui_form.todoEdit.text()
    text = text.strip()

    if text:
        lastRow = self.model.rowCount()
        self.model.insertRow(lastRow) # 在Model中加入一个空行
        index = self.model.index(lastRow, 0)  # 获取最后一行的ModelIndex
        #这里 index的类型不是int而是modelIndex
        self.model.setData(index,text)
```

A Simple Model View --- A TODO list (6)

A Simple Model View --- A TODO list (7)

- Delete
  - self.ui_form.deleteButton.clicked.connect(self.delete)
  - def delete(self):
  - (1) indexes = self.ui_form.todoView.selectedIndexes(); 先获取选定Indexes
  - (2) if indexes:
  - (3) **for** index **in** indexes:
    self.model.removeRow(index.row()) *# 删除当前行*

A Simple Model View --- A TODO list (8)

- Complete
  - self.ui_form.deleteButton.clicked.connect(self.delete)
  - def delete(self):
  - (1) indexes = self.ui_form.todoView.selectedIndexes(); 先获取选定Indexes
  - (2) if indexes:
  - (3) **for** index **in** indexes:

```
data = self.model.data(index)
update_data = "✅"+data
self.model.setData(index,update_data)
```

A Simple Model View --- A TODO list (9)