DATA

PyQt & Relational Databases — Data Format

PyQt provides a convenient way for relational database data presentation.

Stefan Melo 1 day ago · 4 min read ★

Image by Author, background by Pexels

What have we learned?

We have learned that the `QTableView` widget is a convenient and flexible way to present data to the user. In addition, it can handle relational databases efficiently, and it is needless to emphasize that it fits the Model-View-Controller design pattern perfectly. We already know how to connect to a database and present data from relational tables. The most valued benefit was that we needed no SQL language expertise. Now, let's make our data more appealing for the human eye.

What do we need?

It's essential to start with the previous lesson to follow the code we'll be using:

PyQt & Relational Databases

Easy to use full-featured widget for working with relational database data.

towardsdatascience.com

MVC in PyQt

Model-View-Controller (MVC) is a *software design pattern* usually used to develop user interfaces to separate the logic between internal information representations (*model*) and how they are presented (*view*). The *controller* accepts inputs and translates them into instructions for the model or view. PyQt merges view and controller into one element. When the view and the controller elements are combined into one, the result is *model-view* design architecture. PyQt has different model classes and respective view classes. The following table of views shows matching model bases.

Present your data on your own

To have more control over how the data is presented, we will customize the model. In model-view architecture, to write a *custom model*, we must subclass an appropriate abstract model. For our example, the `QAbstractTableModel` is precisely what we need. However, `QAbstractTableModel` is an *abstract base class* known as an *interface* for those from other programming languages. If you use it directly, it won't work. The minimum required methods to be implemented for a custom table are `data` and `rowCount`. The `data` method returns the table data. While the `rowCount` method must return the dimension of a data source.

Modify the previous code

To return some data from our `data` method, we need to modify the code we already have. The data will come from our relational data model. Otherwise, we would lose the relational table functionality.

So, instead having `view.setModel(model)` we go with a new model `view.setModel(presentation_model)`, which will source data from our existing relational table `model`. The modified code looks like this:

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

        self.setWindowTitle('QTableView Example')

        # Setup the model
        model = QSqlRelationalTableModel(self)
        model.setTable('orders')
        model.setRelation(2, QSqlRelation('customers', 'CustomerID',
'Customer'))
        model.setRelation(3, QSqlRelation('products', 'ProductID',
'Product'))
        model.setRelation(1, QSqlRelation('products', 'ProductID',
'Price'))
        model.select()

        # Create the presentation model, which gets data from
```

```
relational table model
        presentation_model = MyTableModel(model)

        # Setup the view
        view = QTableView(self)
        view.setModel(presentation_model)
        view.resizeColumnsToContents()
        self.setCentralWidget(view)
```

The presentation model

But what is this mysterious `MyTableModel` class? Where it comes from? It's our new `QAbstractTableModel` subclass. Let's create it.

```
class MyTableModel(QAbstractTableModel):
    def __init__(self, model):
        super().__init__(model)

        self._model = model

    # Create the data method
    def data(self, index, role):

        if role == Qt.ItemDataRole.DisplayRole:
            value =
self._model.record(index.row()).value(index.column())
            if isinstance(value, int) and index.column() == 1:
                # Format the currency value
                return "${: ,.2f}".format(value)
            return value

    # Create the headerData method
    def headerData(self, section: int, orientation: Qt.Orientation,
role: int):
        if role == Qt.ItemDataRole.DisplayRole and orientation ==
Qt.Orientation.Horizontal:
            return self._model.headerData(section, orientation,
role=role)

    # Create the rowCount method
    def rowCount(self, parent: QModelIndex) -> int:
        return self._model.rowCount()

    # Create the columnCount method
    def columnCount(self, parent: QModelIndex) -> int:
        return self._model.columnCount()
```

Let's comment on some important details:

- We must not forget to import a few modules:
  `from PyQt6.QtCore import QAbstractTableModel, QModelIndex, Qt`
- The `__init__` constructor accepts only a single parameter `model`.
  We store it in `self._model` to access it in our methods.
- The `data` method first looks for the display `role`. When it's found, we get the `value` and check the column where formatting is required. Finally, the method returns a formatted `value` for the presentation.
- The `headerData` method looks for the `role` and `orientation`. When orientation is `horizontal`, it returns the headers available. As we don't have row headers defined, the row header becomes hidden.
- Although we have created the `rowCount` and `columnCount` methods, they do nothing

at the moment. However, we'll use them later.

Conclusion

With a slight update to our code, we can format the data in any way we like. Here's the result:

Next, we'll look at a few other formats and start to modify the data.

If you'd like to read more content like this, become a member:

Join Medium with my referral link — Stefan Melo

As a Medium member, a portion of your membership fee goes to writers you read, and you get full access to every story…

melo-stefan.medium.com

Or you can subscribe to my newsletter below. Thank you.

Pyqt Python Relational Databases Data Towards Data Science