**Step 1: Creating the Contact Book’s Skeleton App With PyQt**

In this first step, you’ll create a minimal but functional PyQt GUI application to provide the foundation on which you’ll start building the contact book. You’ll also create the minimal required project structure, including the project’s main package and an entry-point script to run the application.

All the code and files you’ll add to the contact book project in this section are collected under the source\_code\_step\_1/ directory. You can download them by clicking the link below:

**Get the Source Code:** [**Click here to get the source code you’ll use**](https://realpython.com/bonus/python-contact-book-code/) to build a contact book with Python, PyQt, and SQLite in this tutorial.

By the end of the this section, you’ll be able to run the skeleton GUI application for your contact book for the first time.

Structuring the Contact Book Project

To start coding the application, go ahead and create a new directory called rpcontacts\_project/. This will be the project’s root directory. Now create a new subdirectory called rpcontacts/ inside rpcontacts\_project/. This subdirectory will hold the application’s main [package](https://realpython.com/python-modules-packages/). Finally, fire up your [code editor or IDE](https://realpython.com/python-ides-code-editors-guide/) within the root directory.

To turn a directory into a package, Python needs an \_\_init\_\_.py module to initialize the package. Create this file within rpcontacts/ and add the following code to it:

# -\*- coding: utf-8 -\*-

"""This module provides the rpcontacts package."""

\_\_version\_\_ = "0.1.0"

This file tells Python that rpcontacts is a package. The code in the file runs when you import the package or some of its modules.

You don’t need to put any code in an \_\_init\_\_.py file to initialize the package. An empty \_\_init\_\_.py file will do the job. However, in this case, you define a module-level constant called \_\_version\_\_ to hold the version number of your application.

Creating the Application’s Main Window

Now it’s time to create your contact book’s main window. To do that, create a module called views.py in your rpcontacts package. Then add the following code to the module and save it:

# -\*- coding: utf-8 -\*-

"""This module provides views to manage the contacts table."""

from PyQt5.QtWidgets import (

QHBoxLayout,

QMainWindow,

QWidget,

)

class Window(QMainWindow):

"""Main Window."""

def \_\_init\_\_(self, parent=None):

"""Initializer."""

super().\_\_init\_\_(parent)

self.setWindowTitle("RP Contacts")

self.resize(550, 250)

self.centralWidget = QWidget()

self.setCentralWidget(self.centralWidget)

self.layout = QHBoxLayout()

self.centralWidget.setLayout(self.layout)

First, you import the required classes from PyQt5.QtWidgets. Then you create Window. This class inherits from [QMainWindow](https://doc.qt.io/qt-5.15/qmainwindow.html) and provides the code to generate the application’s main window. In the initializer method, you set the window’s title to "RP Contacts", resize the window to 550 by 250 pixels, define and set the central widget using [QWidget](https://doc.qt.io/qt-5.15/qwidget.html), and finally define a [layout](https://realpython.com/python-pyqt-layout/) for the central widget using a horizontal box layout.

Coding and Running the Application

Since you already have a main window for the contact book, it’s time to write the code for creating a functional PyQt application using [QApplication](https://doc.qt.io/qt-5.15/qapplication.html). To do that, create a new module called main.py in your rpcontacts package and add the following code to it:

# -\*- coding: utf-8 -\*-

# rpcontacts/main.py

"""This module provides RP Contacts application."""

import sys

from PyQt5.QtWidgets import QApplication

from .views import Window

def main():

"""RP Contacts main function."""

# Create the application

app = QApplication(sys.argv)

# Create the main window

win = Window()

win.show()

# Run the event loop

sys.exit(app.exec())

In this module, you import [sys](https://docs.python.org/3/library/sys.html#module-sys) to get access to [exit()](https://docs.python.org/3/library/sys.html#sys.exit), which allows you to cleanly exit the application when the user closes the main window. Then you import QApplication from PyQt5.QtWidgets and Window from views. The final step is to define [main()](https://realpython.com/python-main-function/) as your application’s main function.

Inside main(), you instantiate QApplication and Window. Then you call .show() on Window, and finally you run the application’s **main loop**, or **event loop**, using [.exec()](https://doc.qt.io/qt-5.15/qapplication.html#exec).

Now move up to the project root directory rpcontacts\_project/ and create a file called rpcontacts.py. This file provides the entry-point script to run the application. Add the following code to the file and save it:

#!/usr/bin/env python3

# -\*- coding: utf-8 -\*-

# rpcontacts\_project/rpcontacts.py

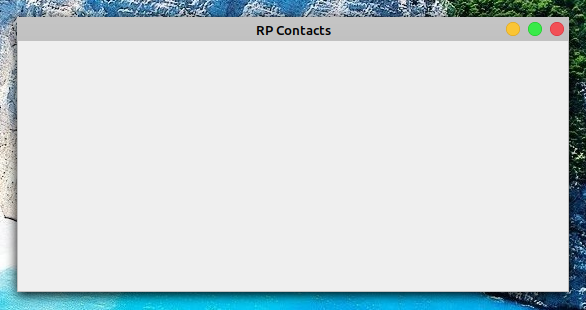
"""This module provides RP Contacts entry point script."""

from rpcontacts.main import main

if \_\_name\_\_ == "\_\_main\_\_":

main()

This file imports main() from your main.py module. Then you implement the traditional [conditional statement](https://realpython.com/python-conditional-statements/) that calls main() if the user [runs](https://realpython.com/run-python-scripts/) this module as a Python script. Now launch the application by running the command python rpcontacts.py in your Python environment. You’ll get the following window on your screen:

[](https://files.realpython.com/media/Contact-Book-App-Skeleton.8c91f7c7406a.png)

That’s it! You’ve created a minimal but functional PyQt GUI application that you can use as a starting point for building your contact book. At this point, your project should have the following structure:

./rpcontacts\_project/

│

├── rpcontacts/

│ ├── \_\_init\_\_.py

│ ├── views.py

│ └── main.py

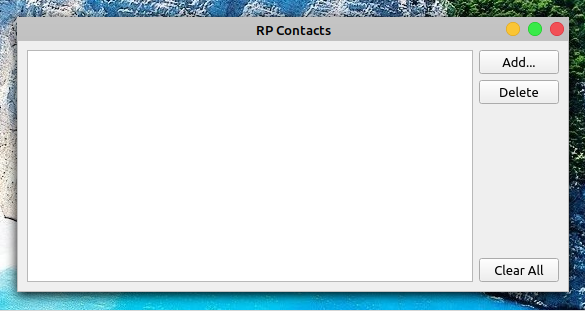
│

└── rpcontacts.py

In this section, you’ve created the minimal required structure for your contact book project using Python modules and packages. You’ve built the application’s main window and put together the boilerplate code to create a PyQt GUI application. You’ve also run the application for the first time. Next, you’ll start adding features to your GUI.

**Step 2: Building the Contact Book’s GUI With Python**

Now that you’ve built the skeleton of your contact book application, you can start coding the main window’s GUI. At the end of this section, you’ll have completed the required steps to create the GUI of your contact book using Python and PyQt. The GUI will look like this:

[](https://files.realpython.com/media/Contact-Book-Main-Window.190ed869068e.png)

In the center of the window, you have a table view to display your contacts list. At the right side of the form, you have three buttons:

1. ***Add*** to add a new contact to the list
2. ***Delete*** to remove a selected contact from the list
3. ***Clear All*** to remove all the contacts from the list

All the code and files you’ll add or modify in this section are collected under the source\_code\_step\_2/ directory. You can download them by clicking the link below:

**Get the Source Code:** [**Click here to get the source code you’ll use**](https://realpython.com/bonus/python-contact-book-code/) to build a contact book with Python, PyQt, and SQLite in this tutorial.

Get back to the views.py module and update the code of Window to generate the above GUI:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/views.py

3

4"""This module provides views to manage the contacts table."""

5

6from PyQt5.QtWidgets import (

7 QAbstractItemView,

8 QHBoxLayout,

9 QMainWindow,

10 QPushButton,

11 QTableView,

12 QVBoxLayout,

13 QWidget,

14)

15

16class Window(QMainWindow):

17 """Main Window."""

18 def \_\_init\_\_(self, parent=None):

19 """Initializer."""

20 # Snip...

21

22 self.setupUI()

23

24 def setupUI(self):

25 """Setup the main window's GUI."""

26 # Create the table view widget

27 self.table = QTableView()

28 self.table.setSelectionBehavior(QAbstractItemView.SelectRows)

29 self.table.resizeColumnsToContents()

30 # Create buttons

31 self.addButton = QPushButton("Add...")

32 self.deleteButton = QPushButton("Delete")

33 self.clearAllButton = QPushButton("Clear All")

34 # Lay out the GUI

35 layout = QVBoxLayout()

36 layout.addWidget(self.addButton)

37 layout.addWidget(self.deleteButton)

38 layout.addStretch()

39 layout.addWidget(self.clearAllButton)

40 self.layout.addWidget(self.table)

41 self.layout.addLayout(layout)

You first import some extra PyQt classes to use in the GUI. Here are some of the more relevant ones:

* [QPushButton](https://doc.qt.io/qt-5.15/qpushbutton.html) to create the *Add*, *Delete*, and *Clear All* buttons
* [QTableView](https://doc.qt.io/qt-5.15/qtableview.html) to provide the table-like view that displays the contacts list
* [QAbstractItemView](https://doc.qt.io/qt-5.15/qabstractitemview.html) to provide access to the table view selection behavior policy

In this code, the first addition to Window is a call to .setupUI() at the end of \_\_init\_\_(). This call generates the main window’s GUI when you run the application.

Here’s what the code inside .setupUI() does:

* **Line 27** creates a QTableView instance to display the contacts list.
* **Line 28** sets the [.selectionBehavior](https://doc.qt.io/qt-5.15/qabstractitemview.html#selectionBehavior-prop) property to [QAbstractItemView.SelectRows](https://doc.qt.io/qt-5.15/qabstractitemview.html" \l "SelectionBehavior-enum). This ensures that when a user clicks on any cell of the table view, the complete row will be selected. The rows in the table view hold all the information related to a single contact in the list of contacts.
* **Lines 31 to 33** add the three buttons to the GUI: *Add*, *Delete*, and *Clear All*. These buttons don’t perform any actions yet.
* **Lines 35 to 41** create and set a coherent layout for all the widgets in the GUI.

With these additions to Window, you can run the application again. The window on your screen will look like the window you saw at the beginning of the section.

**Note:** The line numbers in the above code and in the rest of the code samples in this tutorial are intended to facilitate the explanation. They don’t match the order of lines in the final module or script.

In this section, you’ve run all the required steps to create the GUI of your contact book’s main window. You’re now ready to start working on how your application will manage and store your contact data.

**Step 3: Setting Up the Contact Book’s Database**

At this point, you’ve created a PyQt application and its main window’s GUI to build your contact book project. In this section, you’ll write code to define how the application connects to the contact database. To complete this step, you’ll use SQLite to handle the database and [PyQt’s SQL support](https://realpython.com/python-pyqt-database/) to connect the application to the database and to work with your contact data.

The source code and files you’ll add or modify in this section are stored under the source\_code\_step\_3/ directory. You can download them by clicking the link below:

**Get the Source Code:** [**Click here to get the source code you’ll use**](https://realpython.com/bonus/python-contact-book-code/) to build a contact book with Python, PyQt, and SQLite in this tutorial.

First, get back to main.py in the rpcontacts/ directory and update the code to create the connection to the database:

# -\*- coding: utf-8 -\*-

# rpcontacts/main.py

"""This module provides RP Contacts application."""

import sys

from PyQt5.QtWidgets import QApplication

from .database import createConnection

from .views import Window

def main():

"""RP Contacts main function."""

# Create the application

app = QApplication(sys.argv)

# Connect to the database before creating any window

if not createConnection("contacts.sqlite"):

sys.exit(1)

# Create the main window if the connection succeeded

win = Window()

win.show()

# Run the event loop

sys.exit(app.exec\_())

In this case, you first import createConnection() from database.py. This function will contain code to create and open a connection to the contact database. You’ll create database.py and write createConnection() in the next section.

Inside main(), the first highlighted line is an attempt to create a connection to the database using createConnection(). If for some reason the application isn’t able to create a connection, then the call to sys.exit(1) will close the application without creating a graphical element and will indicate that an error has occurred.

You have to handle the connection this way because the application depends on the database to work properly. If you don’t have a functional connection, then your application won’t work at all.

This practice allows you to handle errors and cleanly close the application if a problem occurs. You’ll also be able to present the user with relevant information about the error that the application ran into when trying to connect to the database.

With these additions in place, it’s time to dive into the code of createConnection().

Connecting to the Database With PyQt and SQLite

Connecting your contact book application to its associated database is a fundamental step in developing the application. To do this, you’ll code a function called createConnection(), which will create and open a connection to the database. If the connection is successful, then the function will [return](https://realpython.com/python-return-statement/) True. Otherwise, it will provide information about the cause of the connection failure.

Get back to the rpcontacts/ directory and create a new module called database.py within it. Then add the following code to that module:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/database.py

3

4"""This module provides a database connection."""

5

6from PyQt5.QtWidgets import QMessageBox

7from PyQt5.QtSql import QSqlDatabase

8

9def createConnection(databaseName):

10 """Create and open a database connection."""

11 connection = QSqlDatabase.addDatabase("QSQLITE")

12 connection.setDatabaseName(databaseName)

13

14 if not connection.open():

15 QMessageBox.warning(

16 None,

17 "RP Contact",

18 f"Database Error: {connection.lastError().text()}",

19 )

20 return False

21

22 return True

Here, you first import some required PyQt classes. Then you define createConnection(). This function takes one argument: **databaseName** holds the name or path to the physical SQLite database file in your file system.

Here’s what the code inside createConnection() does:

* **Line 11** creates the database connection using the [QSQLITE](https://doc.qt.io/qt-5/sql-driver.html#qsqlite) driver.
* **Line 12** sets the filename or the path to the database.
* **Line 14** attempts to open the connection. If a problem occurs during the call to .open(), then the if code block shows an error message and then returns False to indicate that the connection attempt failed.
* **Line 22** returns True if the connection attempt is successful.

You already coded createConnection(). Now you can write the code to create the contacts tables in the database.

Creating the contacts Table

With the function that creates and opens the connection to the database in place, you can proceed to code a helper function to create the contacts table. You’ll use this table to store the information about your contacts.

Here’s the code that implements \_createContactsTable():

# -\*- coding: utf-8 -\*-

# rpcontacts/database.py

# Snip...

from PyQt5.QtSql import QSqlDatabase, QSqlQuery

def \_createContactsTable():

"""Create the contacts table in the database."""

createTableQuery = QSqlQuery()

return createTableQuery.exec(

"""

CREATE TABLE IF NOT EXISTS contacts (

id INTEGER PRIMARY KEY AUTOINCREMENT UNIQUE NOT NULL,

name VARCHAR(40) NOT NULL,

job VARCHAR(50),

email VARCHAR(40) NOT NULL

)

"""

)

def createConnection(databaseName):

# Snip...

\_createContactsTable()

return True

Here, you first add a new import. You import [QSqlQuery](https://doc.qt.io/qt-5.15/qsqlquery.html) to execute and manipulate SQL statements.

Inside \_createContactsTable(), you create a QSqlQuery instance. Then you call .exec() on the query object with a string-based SQL [CREATE TABLE statement](https://en.wikipedia.org/wiki/Data_definition_language#CREATE_TABLE_statement) as an argument. This statement creates a new table called contacts in your database. The table has the following columns:

| **Column** | **Content** |
| --- | --- |
| id | An [integer](https://realpython.com/python-numbers/#integers) with the table’s [primary key](https://en.wikipedia.org/wiki/Primary_key) |
| name | A [string](https://realpython.com/python-strings/) with the name of a contact |
| job | A string with the job title of a contact |
| email | A string with the email of a contact |

The contacts table in your database will store relevant information about your contacts.

The final step to finish coding database.py is to add a call to \_createContactsTable() from inside createConnection(), right before the last [return statement](https://realpython.com/python-return-statement/). This ensures that the application creates the contacts table before doing any operations on the database.

Once you’ve created the contacts table, you can run some tests on the database and also add some sample data for further testing.

Testing the Contact Book’s Database

So far, you’ve finished writing the required code to handle the connection to the contact book’s database. In this section, you’ll perform some tests to make sure that this code and the database itself work properly. You’ll also add some sample data to the database to perform further testing later in this tutorial.

Now open a terminal or command line and move to the project’s root directory, rpcontacts\_project/. Once there, launch a Python [interactive session](https://realpython.com/interacting-with-python/) and type in the following code:

>>>

>>> from rpcontacts.database import createConnection

>>> # Create a connection

>>> createConnection("contacts.sqlite")

True

>>> # Confirm that contacts table exists

>>> from PyQt5.QtSql import QSqlDatabase

>>> db = QSqlDatabase.database()

>>> db.tables()

['contacts', 'sqlite\_sequence']

Here, you first import createConnection() from the database.py module. Then you call this function to create and open a connection to the contact database. The database filename is contacts.sqlite. Since this file doesn’t exist in the project’s root directory, SQLite creates it for you. You can check this by taking a look at your current directory.

Next, you confirm that the database contains a table called contacts. To do that, you call .database() on QSqlDatabase. This class method returns a [pointer](https://realpython.com/pointers-in-python/) to the current database connection. With this reference to the connection, you can call .tables() to get the list of tables in the database. Note that the first table in the list is contacts, so now you’re sure that everything is working well.

Now you can prepare an SQL query to insert sample data into the contacts table:

>>>

>>> # Prepare a query to insert sample data

>>> from PyQt5.QtSql import QSqlQuery

>>> insertDataQuery = QSqlQuery()

>>> insertDataQuery.prepare(

... """

... INSERT INTO contacts (

... name,

... job,

... email

... )

... VALUES (?, ?, ?)

... """

... )

True

The above query allows you to insert specific values into the name, job, and email attributes and to save those values to the database. Below is an example of how to do this:

>>>

>>> # Sample data

>>> data = [

... ("Linda", "Technical Lead", "linda@example.com"),

... ("Joe", "Senior Web Developer", "joe@example.com"),

... ("Lara", "Project Manager", "lara@example.com"),

... ("David", "Data Analyst", "david@example.com"),

... ("Jane", "Senior Python Developer", "jane@example.com"),

... ]

>>> # Insert sample data

>>> for name, job, email in data:

... insertDataQuery.addBindValue(name)

... insertDataQuery.addBindValue(job)

... insertDataQuery.addBindValue(email)

... insertDataQuery.exec()

...

True

True

True

True

True

In this piece of code, you first define data to hold the contact information of a list of people. Next, you use a [for loop](https://realpython.com/python-for-loop/) to insert the data by calling .addBindValue(). Then you call .exec() on the query object to effectively run the SQL query on the database.

Since all the calls to .exec() return True, you can conclude that the data was successfully inserted into the database. If you want to confirm this, then run the following code:

>>>

>>> query = QSqlQuery()

>>> query.exec("SELECT name, job, email FROM contacts")

True

>>> while query.next():

... print(query.value(0), query.value(1), query.value(2))

...

Linda Technical Lead linda@example.com

Joe Senior Web Developer joe@example.com

Lara Project Manager lara@example.com

David Data Analyst david@example.com

Jane Senior Python Developer jane@example.com

That’s it! Your database works fine! Now you have some sample data to test the application with, and you can focus on how to load and display the contact information in your contact book’s main window.

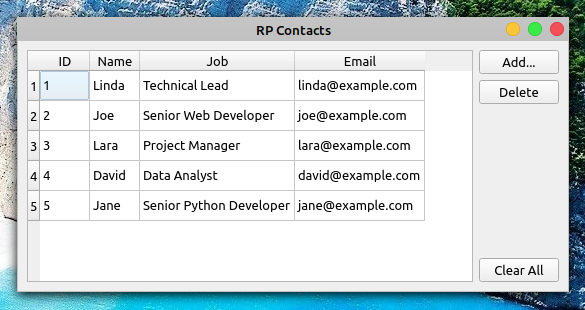
**Step 4: Displaying and Updating Existing Contacts**

To display your contact data in the application’s main window, you can use QTableView. This class is part of PyQt’s [Model-View architecture](https://doc.qt.io/qt-5/model-view-programming.html#the-model-view-architecture) and provides a robust and efficient way of displaying items from a PyQt model object.

The files and the code you’ll add or modify in this section are stored under the source\_code\_step\_4/ directory. To download them, click the link below:

**Get the Source Code:** [**Click here to get the source code you’ll use**](https://realpython.com/bonus/python-contact-book-code/) to build a contact book with Python, PyQt, and SQLite in this tutorial.

Once you’ve finished this step, your contact book will look like this:

[](https://files.realpython.com/media/Contact-Book-Visualize-Data.8ee132b0a2b2.png)

The table view object in the main window provides the required functionality to allow you to modify and update the contact information quickly.

For example, to update the name of a contact, you can double-click the cell containing the name, update the name, and then press Enter to automatically save the changes to the database. But before you can do this, you need to create a model and connect it to the table view.

Creating a Model to Handle the Contact Data

PyQt provides a rich [set of classes](https://doc.qt.io/qt-5/qtsql-module.html) for working with SQL databases. For your contact book application, you’ll use [QSqlTableModel](https://doc.qt.io/qt-5/qsqltablemodel.html), which provides an editable data model for a single database table. It’s perfect for the job since your database has a single table, contacts.

Get back to your code editor and create a new module called model.py inside the rpcontacts/ directory. Add the following code to the file and save it:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/model.py

3

4"""This module provides a model to manage the contacts table."""

5

6from PyQt5.QtCore import Qt

7from PyQt5.QtSql import QSqlTableModel

8

9class ContactsModel:

10 def \_\_init\_\_(self):

11 self.model = self.\_createModel()

12

13 @staticmethod

14 def \_createModel():

15 """Create and set up the model."""

16 tableModel = QSqlTableModel()

17 tableModel.setTable("contacts")

18 tableModel.setEditStrategy(QSqlTableModel.OnFieldChange)

19 tableModel.select()

20 headers = ("ID", "Name", "Job", "Email")

21 for columnIndex, header in enumerate(headers):

22 tableModel.setHeaderData(columnIndex, Qt.Horizontal, header)

23 return tableModel

In this code, you first do some required imports, then you create ContactsModel. In the class initializer, you define an instance attribute called .model to hold the data model.

Next, you add a [static method](https://realpython.com/instance-class-and-static-methods-demystified/) to create and set up the model object. Here’s what the code in .\_createModel() does:

* **Line 16** creates an instance of QSqlTableModel() called tableModel.
* **Line 17** associates the model object with the contacts table in your database.
* **Line 18** sets the [.editStrategy](https://doc.qt.io/qt-5/qsqltablemodel.html#editStrategy) property of the model to [QSqlTableModel.OnFieldChange](https://doc.qt.io/qt-5/qsqltablemodel.html" \l "EditStrategy-enum). With this, you ensure that the changes on the model get saved into the database immediately.
* **Line 19** loads the table into the model by calling .select().
* **Lines 20 to 22** define and set user-friendly headers for the contacts table’s columns.
* **Line 23** returns the newly created model.

At this point, you have your data model ready to use. Now you need to connect the table view widget to the model so you can present your users with the contact information.

Connecting the Model to the View

To display contact data in your contact book’s main window, you need to connect the table view with the data model. To perform this connection, you need to call .setModel() on the table view object and pass the model as an argument:

# -\*- coding: utf-8 -\*-

# rpcontacts/views.py

# Snip...

from .model import ContactsModel

class Window(QMainWindow):

"""Main Window."""

def \_\_init\_\_(self, parent=None):

# Snip...

self.contactsModel = ContactsModel()

self.setupUI()

def setupUI(self):

"""Setup the main window's GUI."""

# Create the table view widget

self.table = QTableView()

self.table.setModel(self.contactsModel.model)

self.table.setSelectionBehavior(QAbstractItemView.SelectRows)

# Snip...

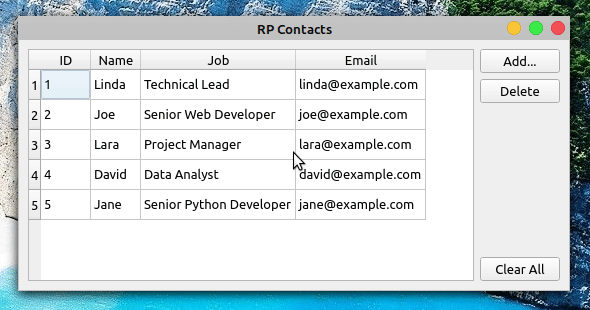
In this code, you first import ContactsModel from model.py. This class provides the model that manages the data in your contact database.

In the initializer of Window, you create an instance of ContactsModel. Then inside .setupUI(), you call .setModel() on .table to connect the model with the table view. If you run the application after this update, then you’ll get the window you saw at the beginning of [step 4](https://realpython.com/python-contact-book/#step-4-displaying-and-updating-existing-contacts).

Displaying and Updating Contacts

PyQt’s Model-View architecture provides a robust and user-friendly way to create GUI applications that manage databases. **Models** communicate with and access the data in the database. Any change in a model updates the database immediately. **Views** are responsible for displaying the data to the user and also for providing editable widgets to allow the user to modify the data directly in the view.

If the user modifies the data through the view, then the view internally communicates with and updates the model, which saves the changes to the physical database:

[](https://files.realpython.com/media/Contact-Book-Visualize-Update-Data.b4d076c6d2e1.gif)

In this example, you double-click Joe’s *Job* field. This gives you access to an editable widget that allows you to modify the value in the cell. Then you update the job description from Senior Web Developer to Web Developer. When you hit Enter, the table view communicates the change to the model, and the model saves the change to the database immediately.

To confirm that the changes were successfully saved into the database, you can close the application and run it again. The table view should reflect your updates.

**Step 5: Creating New Contacts**

At this step, your contact book application provides functionality to load, display, and update the information about your contacts. Even though you’re able to modify and update the contact information, you can neither add nor remove contacts from the list.

All the files and the code you’ll add or modify in this section are collected in the source\_code\_step\_5/ directory. To download them, click the link below:

**Get the Source Code:** [**Click here to get the source code you’ll use**](https://realpython.com/bonus/python-contact-book-code/) to build a contact book with Python, PyQt, and SQLite in this tutorial.

In this section, you’ll provide the required functionality to add new contacts to the database, using a pop-up dialog to enter the new information. The first step is to create the *Add Contact* dialog.

Creating the Add Contact Dialog

Dialogs are small windows that you can use to communicate with your users. In this section, you’ll code the contact book’s *Add Contact* dialog to allow your users add new contacts to their current list of contacts.

To code the *Add Contact* dialog, you’ll subclass [QDialog](https://doc.qt.io/qt-5.15/qdialog.html). This class provides a blueprint to build dialogs for your GUI applications.

Now open the views.py module and update the import section like this:

# -\*- coding: utf-8 -\*-

# rpcontacts/views.py

# Snip...

from PyQt5.QtCore import Qt

from PyQt5.QtWidgets import (

QAbstractItemView,

QDialog,

QDialogButtonBox,

QFormLayout,

QHBoxLayout,

QLineEdit,

QMainWindow,

QMessageBox,

QPushButton,

QTableView,

QVBoxLayout,

QWidget,

)

The highlighted lines in the above code import the required classes to build the *Add Contact* dialog. With these classes in your [namespace](https://realpython.com/python-namespaces-scope/), add the following class at the end of views.py:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/views.py

3

4# Snip...

5class AddDialog(QDialog):

6 """Add Contact dialog."""

7 def \_\_init\_\_(self, parent=None):

8 """Initializer."""

9 super().\_\_init\_\_(parent=parent)

10 self.setWindowTitle("Add Contact")

11 self.layout = QVBoxLayout()

12 self.setLayout(self.layout)

13 self.data = None

14

15 self.setupUI()

16

17 def setupUI(self):

18 """Setup the Add Contact dialog's GUI."""

19 # Create line edits for data fields

20 self.nameField = QLineEdit()

21 self.nameField.setObjectName("Name")

22 self.jobField = QLineEdit()

23 self.jobField.setObjectName("Job")

24 self.emailField = QLineEdit()

25 self.emailField.setObjectName("Email")

26 # Lay out the data fields

27 layout = QFormLayout()

28 layout.addRow("Name:", self.nameField)

29 layout.addRow("Job:", self.jobField)

30 layout.addRow("Email:", self.emailField)

31 self.layout.addLayout(layout)

32 # Add standard buttons to the dialog and connect them

33 self.buttonsBox = QDialogButtonBox(self)

34 self.buttonsBox.setOrientation(Qt.Horizontal)

35 self.buttonsBox.setStandardButtons(

36 QDialogButtonBox.Ok | QDialogButtonBox.Cancel

37 )

38 self.buttonsBox.accepted.connect(self.accept)

39 self.buttonsBox.rejected.connect(self.reject)

40 self.layout.addWidget(self.buttonsBox)

There are a lot of things happening in this code. Here’s a summary:

* **Line 5** defines a new class that [inherits](https://realpython.com/inheritance-composition-python/) from QDialog.
* **Lines 7 to 15** define the class initializer. In this case, the most relevant addition is .data, which is an [instance attribute](https://realpython.com/python3-object-oriented-programming/#class-and-instance-attributes) that you’ll use to hold the data your users provide.

In .setupUI(), you define the dialog’s GUI:

* **Lines 20 to 25** add three [QLineEdit](https://doc.qt.io/qt-5.15/qlineedit.html) objects: name, job, and email. You’ll use these line edits to take the user’s input for the name, job description, and email of the contact to add. They represent the corresponding fields in the database.
* **Lines 27 to 30** create a [QFormLayout](https://doc.qt.io/qt-5.15/qformlayout.html) instance that arranges the line edits in a form. This **layout manager** also provides user-friendly labels for each line edit or field.
* **Lines 33 to 37** add a [QDialogButtonBox](https://doc.qt.io/qt-5.15/qdialogbuttonbox.html) object that provides two standard buttons: *OK* and *Cancel*. The *OK* button accepts the user’s input and the *Cancel* button rejects it.
* **Lines 38 and 39** connect the dialog’s built-in .accepted() and .rejected() signals with the .accept() and reject() slots, respectively. In this case, you’ll rely on the dialog’s built-in .reject() slot, which closes the dialog without processing the input. Other than that, you just need to code the .accept() slot.

To code the dialog’s .accept() slot, you need to consider that any user input needs **validation** to make sure that it’s correct and safe. This is especially true when you’re working with SQL databases because of the risk of an [SQL injection attack](https://realpython.com/prevent-python-sql-injection/).

In this example, you’ll add a minimal validation rule just to make sure that the user provides data for each input field in the dialog. However, adding your own, more robust validation rules would be a good exercise.

Without further ado, get back to AddDialog and add the following code for its .accept() slot:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/views.py

3

4# Snip...

5class AddDialog(QDialog):

6 def \_\_init\_\_(self, parent=None):

7 # Snip...

8

9 def setupUI(self):

10 # Snip...

11

12 def accept(self):

13 """Accept the data provided through the dialog."""

14 self.data = []

15 for field in (self.nameField, self.jobField, self.emailField):

16 if not field.text():

17 QMessageBox.critical(

18 self,

19 "Error!",

20 f"You must provide a contact's {field.objectName()}",

21 )

22 self.data = None # Reset .data

23 return

24

25 self.data.append(field.text())

26

27 if not self.data:

28 return

29

30 super().accept()

The code within .accept() does the following:

* **Line 14** initializes .data to an empty list ([]). This list will store the user’s input data.
* **Line 15** defines a for loop that iterates over the three line edits, or fields, in the dialog.
* **Lines 16 to 23** define a conditional statement that checks if the user has provided data for each field in the dialog. If not, then the dialog shows an error message that warns the user about the missing data.
* **Line 25** adds the user’s input for each field to .data.
* **Line 30** calls the superclass’s .accept() slot to provide the standard behavior that closes the dialog after the user clicks *OK*.

With this code, you’re ready to add a new slot to the contact book’s main window. This slot will launch the dialog, and if the user provides valid input, then the slot will use the model to save the newly added contact to the database.

Launching the Add Contact Dialog

Now that you’ve coded the *Add Contact* dialog, it’s time to add a new slot to Window so you can launch the dialog by clicking *Add* and process the user’s input once they click *OK*.

Go to the definition of Window and add the following code:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/views.py

3

4# Snip...

5class Window(QMainWindow):

6 # Snip...

7

8 def setupUI(self):

9 # Snip...

10 self.addButton = QPushButton("Add...")

11 self.addButton.clicked.connect(self.openAddDialog)

12 # Snip...

13

14 def openAddDialog(self):

15 """Open the Add Contact dialog."""

16 dialog = AddDialog(self)

17 if dialog.exec() == QDialog.Accepted:

18 self.contactsModel.addContact(dialog.data)

19 self.table.resizeColumnsToContents()

Here’s a summary of what’s happening in the above code:

* **Line 11** connects the .clicked() signal of the *Add* button to the newly created slot, .openAddDialog(). This way, a click on the button will automatically call the slot.
* **Line 14** defines the .openAddDialog() slot.
* **Line 16** creates an instance of AddDialog.
* **Lines 17 to 19** define a conditional statement to check if the dialog was accepted. If so, then line 14 calls .addContact() on the data model with the dialog’s .data attribute as an argument. The final statement in the if code block resizes the table view to fit the size of its updated content.

Now that you have a way to launch the *Add Contact* dialog and to process its data, you need to provide the code for .addContact() in your data model. That’s a topic for the next section.

Processing the Add Dialog’s Data in the Model

In this section, you’ll add a method called .addContact() to your data model, ContactsModel. Open model.py in your code editor, go to the definition of ContactsModel, and add the following code:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/model.py

3

4# Snip...

5class ContactsModel:

6 # Snip...

7

8 def addContact(self, data):

9 """Add a contact to the database."""

10 rows = self.model.rowCount()

11 self.model.insertRows(rows, 1)

12 for column, field in enumerate(data):

13 self.model.setData(self.model.index(rows, column + 1), field)

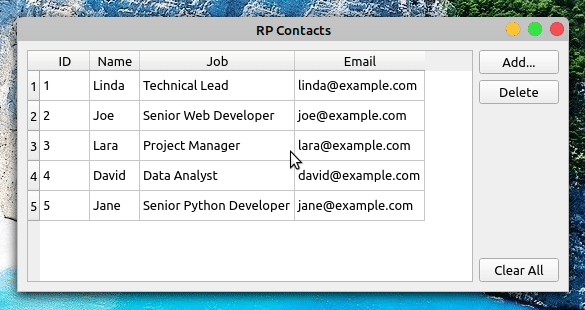
14 self.model.submitAll()

15 self.model.select()

Inside .addContact(), the code does the following:

* **Line 10** gets the current number of rows in the data model.
* **Line 11** inserts a new row at the end of the data model.
* **Lines 12 and 13** run a for loop that inserts every item in data into the corresponding cell in the data model. To do this, line 9 calls .setData() on the model, with the index of the cell and the current data field as arguments.
* **Line 14** submits the changes to the database by calling .submitAll() on the model.
* **Line 15** reloads the data from the database into the model.

If you run the application with these new additions, then you’ll get the following behavior:

[](https://files.realpython.com/media/Contact-Book-Add-Contact.cfe6c106c0d0.gif)

Now when you click *Add*, the *Add Contact* dialog appears on your screen. You can use the dialog to provide the required information for a new contact and to add the contact to the database by clicking *OK*.

**Step 6: Deleting Existing Contacts**

The final feature you’ll add to the contact book application is the ability to remove contacts from the database using the GUI.

Again, you’ll find all the files and the code added or modified in this section under the source\_code\_step\_6/ directory. You can download them by clicking the link below:

**Get the Source Code:** [**Click here to get the source code you’ll use**](https://realpython.com/bonus/python-contact-book-code/) to build a contact book with Python, PyQt, and SQLite in this tutorial.

In this section, you’ll first add the capability to delete a single contact at a time. Then you’ll add code to remove all the contacts from the database.

Deleting Selected Contacts

To remove a single contact from the contact database, you need to select the desired contact in the table view on the contact book’s main window. Once you’ve selected the contact, you can click *Delete* to perform the operation on the database.

Go to the model.py module and add the following code to implement .deleteContact() inside ContactsModel:

# -\*- coding: utf-8 -\*-

# rpcontacts/model.py

# Snip...

class ContactsModel:

# Snip...

def deleteContact(self, row):

"""Remove a contact from the database."""

self.model.removeRow(row)

self.model.submitAll()

self.model.select()

This method has three lines of code. The first line removes the selected row. The second line submits the change to the database. Finally, the third line reloads the data into the model.

Next, get back to the views.py module and add the code behind the *Delete* button in Window:

# -\*- coding: utf-8 -\*-

# rpcontacts/views.py

# Snip...

class Window(QMainWindow):

# Snip...

def setupUI(self):

"""Setup the main window's GUI."""

# Snip...

self.deleteButton = QPushButton("Delete")

self.deleteButton.clicked.connect(self.deleteContact)

# Snip...

def deleteContact(self):

"""Delete the selected contact from the database."""

row = self.table.currentIndex().row()

if row < 0:

return

messageBox = QMessageBox.warning(

self,

"Warning!",

"Do you want to remove the selected contact?",

QMessageBox.Ok | QMessageBox.Cancel,

)

if messageBox == QMessageBox.Ok:

self.contactsModel.deleteContact(row)

In the first highlighted line, you connect the .clicked() signal of the *Delete* button to the .deleteContact() slot. This connection triggers a call to .deleteContact() every time the user clicks the button.

In .deleteContact(), you first get the index of the currently selected row in the table view. The if statement checks if the index is lower than 0, which would mean that there are no contacts in the table view. If so, then the method returns immediately without performing any further actions.

Then the method shows a warning message confirming that the user wants to delete the selected contact. If the user accepts the operation, then .deleteContact(row) gets called. In this case, row represents the index of the currently selected row in the table.

After these additions, you can run the application again to get the following behavior:

[](https://files.realpython.com/media/Contact-Book-Delete-Contact.94bad3c91752.gif)

Now when you select a contact from the table view and click *Delete*, you’re presented with a warning message. If you click the message dialog’s *OK* button, then the application removes the selected contact from the database, updating the table view accordingly.

Clearing the Contact Database

To remove all the contacts from the database, you’ll start by adding a method called .clearContacts() to ContactsModel. Open your model.py module and add the following method at the end of the class:

1# -\*- coding: utf-8 -\*-

2# rpcontacts/model.py

3

4# Snip...

5class ContactsModel:

6 # Snip...

7

8 def clearContacts(self):

9 """Remove all contacts in the database."""

10 self.model.setEditStrategy(QSqlTableModel.OnManualSubmit)

11 self.model.removeRows(0, self.model.rowCount())

12 self.model.submitAll()

13 self.model.setEditStrategy(QSqlTableModel.OnFieldChange)

14 self.model.select()

Here’s what each line of code does:

* **Line 10** sets the data model’s .editStrategy property to QSqlTableModel.OnManualSubmit. This allows you to cache all the changes until you call .submitAll() later on. You need to do this because you’re changing several rows at the same time.
* **Line 11** removes all the rows from the model.
* **Line 12** saves changes to the database.
* **Line 13** resets the model’s .editStrategy property to its original value, QSqlTableModel.OnFieldChange. If you don’t reset this property to its original value, then you won’t be able to update the contacts directly in the table view.
* **Line 14** reloads the data into the model.

Once you’ve coded .clearContacts(), you can get back to the views.py file and update Window with the following code:

# -\*- coding: utf-8 -\*-

# rpcontacts/views.py

# Snip...

class Window(QMainWindow):

# Snip...

def setupUI(self):

"""Setup the main window's GUI."""

# Snip...

self.clearAllButton = QPushButton("Clear All")

self.clearAllButton.clicked.connect(self.clearContacts)

# Snip...

def clearContacts(self):

"""Remove all contacts from the database."""

messageBox = QMessageBox.warning(

self,

"Warning!",

"Do you want to remove all your contacts?",

QMessageBox.Ok | QMessageBox.Cancel,

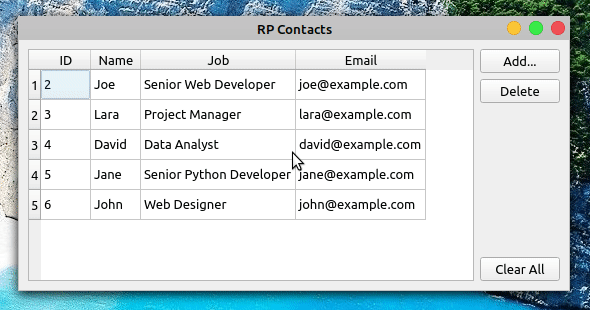
)

if messageBox == QMessageBox.Ok:

self.contactsModel.clearContacts()

The first highlighted line in this code connects the .clicked() signal of the *Clear All* button to the .clearContacts() slot below.

In .clearContacts(), you first create a message dialog, messageBox, to ask the user to confirm the removing operation. If the user confirms the operation by clicking *OK*, then .clearContacts() gets called on the model to remove all the contacts from the database:

[](https://files.realpython.com/media/Contact-Book-Clear-All-Contacts.70cc216d2f46.gif)

That’s it! With this last piece of code, your contact book application is complete. The application provides features that allow your users to display, add, update, and remove contacts from the database.

**Conclusion**

Building a contact book GUI application with Python, PyQt, and SQLite is an excellent exercise for you to expand your skills with these tools and as a developer in general. Coding projects like this allows you to apply the knowledge and skills you already have and also pushes you to research and learn about new topics every time you encounter a new programming problem.

**In this tutorial, you learned how to:**

* Build the **GUI for a contact book application** using PyQt
* Use **PyQt’s SQL support** to connect the application to an **SQLite database**
* Use **PyQt’s Model-View architecture** to work with the application’s database

You can download the complete source code for the contact book application and also the code to complete each step in this tutorial by clicking the link below:

**Get the Source Code:** [**Click here to get the source code you’ll use**](https://realpython.com/bonus/python-contact-book-code/) to build a contact book with Python, PyQt, and SQLite in this tutorial.

**Next Steps**

At this point, you’ve completed a fully functional contact book project. The application provides minimal functionality, but it’s a good starting point to continue adding features and take your Python and PyQt skills to the next level. Here are some next step ideas that you can implement:

* **Add new data fields:** Adding new data fields to store more information about your contacts would be great. For example, you can add the contact’s photo, phone number, web page, Twitter handle, and so on. To do this, you might need to create new tables and set up relations between them. PyQt provides the [QSqlRelationalTableModel](https://doc.qt.io/qt-5.15/qsqlrelationaltablemodel.html), which defines an editable data model for a single table and provides foreign key support.
* **Provide search capability:** Giving your users a way to search for a contact in the database is arguably a must-have feature in this kind of application. To implement it, you can use PyQt’s QSqlQuery and [QSqlQueryModel](https://doc.qt.io/qt-5.15/qsqlquerymodel.html).
* **Add back-up capability:** Providing a way of backing up contact information is another interesting feature. Users might face problems with their computers and lose their data. You can provide options to upload the data to a cloud service or to back it up to an external disk.

These are just a few ideas for how you can continue adding features to your contact book. Take the challenge and build something amazing on top of this!