

PyQt 6.7

Your First QtWidgets Application

As with any other programming framework, you start with the traditional "Hello World" program.

Here is a simple example of a Hello World application in PySide6:

```
import sys
from PySide6.QtWidgets import QApplication, QLabel

app = QApplication(sys.argv)
label = QLabel("Hello World!")
label.show()
app.exec()
```

When you execute it the code, the application will look like:

```
import sys
from PySide6.QtWidgets import QApplication, QLabel

app = QApplication(sys.argv)
label = QLabel("Hello World!")
label.show()
app.exec()
```



For a widget application using PySide6, you must always start by importing the appropriate class from the *PySide6.QtWidgets* module.

After the imports, you create a *QApplication* instance. As Qt can receive arguments from command line, you may pass any argument to the *QApplication* object. Usually, you don't need to pass any arguments so you can leave it as is, or use the following approach:

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

After the creation of the application object, we have created a *QLabel* object. A *QLabel* is a widget that can present text (simple or rich, like html), and images:

```
# This HTML approach will be valid too!
label = QLabel("<font color=red size=40>Hello World!</font>")
```

Note

After creating the label, we call *show()* on it.

Finally, we call *app.exec()* to enter the Qt main loop and start to execute the Qt code. In reality, it is only here where the label is shown, but this can be ignored for now.

Using a Simple Button

In this tutorial, we'll show you how to handle **signals and slots** using Qt for Python. **Signals and slots** is a Qt feature that lets your graphical widgets communicate with other graphical widgets or your python code. Our application creates a button that logs the *Button clicked, Hello!* message to the python console each time you click it.

Let's start by importing the necessary PySide6 classes and python *sys* module:

```
import sys
from PySide6.QtWidgets import QApplication, QPushButton
from PySide6.QtCore import Slot
```

Let's also create a python function that logs the message to the console:

```
# Greetings
@Slot()
def say_hello():
    print("Button clicked, Hello!")
```

Note

The *@Slot()* is a decorator that identifies a function as a slot. It is not important to understand why for now, but use it always to avoid unexpected behavior.

Now, as mentioned in previous examples you must create the *QApplication* to run your PySide6 code:

```
# Create the Qt Application
app = QApplication(sys.argv)
```

Let's create the clickable button, which is a *QPushButton* instance. To label the button, we pass a python string to the constructor:

```
# Create a button
button = QPushButton("Click me")
```

Before we show the button, we must connect it to the `say_hello()` function that we defined earlier. There are two ways of doing this; using the old style or the new style, which is more pythonic. Let's use the new style in this case. You can find more information about both these styles in the [Signals and Slots in PySide6](#) wiki page.

The `QPushButton` has a predefined signal called **clicked**, which is triggered every time the button is clicked. We'll connect this signal to the `say_hello()` function:

```
# Connect the button to the function
button.clicked.connect(say_hello)
```

Finally, we show the button and start the Qt main loop:

```
# Show the button
button.show()
# Run the main Qt loop
app.exec()
```

Here is the complete code for this example:

```
import sys
from PySide6.QtWidgets import QApplication, QPushButton
from PySide6.QtCore import Slot

@Slot()
def say_hello():
    print("Button clicked, Hello!")

# Create the Qt Application
app = QApplication(sys.argv)
# Create a button, connect it and show it
button = QPushButton("Click me")
button.clicked.connect(say_hello)
button.show()
# Run the main Qt loop
app.exec()
```

After a few clicks, you will get something like this on your terminal:

```
~/dev/pyside-setup/sources/pyside6/doc/tutorials/basic/tutorial
```

```
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!  
Button clicked, Hello!
```

```
□
```

Click me

Signals and Slots

Due to the nature of Qt, QObjects require a way to communicate, and that's the reason for this mechanism to be a **central feature of Qt**.

In simple terms, you can understand **Signal and Slots** in the same way you interact with the lights in your house. When you move the light switch (signal) you get a result which may be that your light bulbs are switched on/off (slot).

While developing interfaces, you can get a real example by the effect of clicking a button: the 'click' will be the signal, and the slot will be what happens when that button is clicked, like closing a window, saving a document, etc.

Note

If you have experience with other frameworks or toolkits, it's likely that you read a concept called 'callback'. Leaving the implementation details aside, a callback will be related to a notification function, passing a pointer to a function in case it's required due to the events that happen in your program. This approach might sound similar, but there are essential differences that make it an unintuitive approach, like ensuring the type correctness of callback arguments, and some others.

All classes that inherit from QObject or one of its subclasses, like QWidget can contain signals and slots. **Signals are emitted by objects** when they change their state in a way that may be interesting to other objects. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation, and ensures that the object can be used as a software component.

Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt.

You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you need. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

Qt's widgets have many predefined signals and slots. For example, `QAbstractButton` (base class of buttons in Qt) has a `clicked()` signal and `QLineEdit` (single line input field) has a slot named `clear()`. So, a text input field with a button to clear the text could be implemented by placing a `QToolButton` to the right of the `QLineEdit` and connecting its `clicked()` signal to the slot `clear()`. This is done using the `connect()` method of the signal:

```
button = QToolButton()
line_edit = QLineEdit()
button.clicked.connect(line_edit.clear)
```

`connect()` returns a `QMetaObject.Connection` object, which can be used with the `disconnect()` method to sever the connection.

Signals can also be connected to free functions:

```
import sys
from PySide6.QtWidgets import QApplication, QPushButton

def function():
    print("The 'function' has been called!")

app = QApplication()
button = QPushButton("Call function")
button.clicked.connect(function)
button.show()
sys.exit(app.exec())
```

Connections can be spelled out in code or, for widget forms, designed in the [Signal-Slot Editor](#) of *Qt Widgets Designer*.

The Signal Class

When writing classes in Python, signals are declared as class level variables of the class `QtCore.Signal()`. A `QWidget`-based button that emits a `clicked()` signal could look as follows:

```
from PySide6.QtCore import Qt, Signal
from PySide6.QtWidgets import QWidget

class Button(QWidget):

    clicked = Signal(Qt.MouseButton)

    ...

    def mousePressEvent(self, event):
```

```
self.clicked.emit(event.button())
```

The constructor of Signal takes a tuple or a list of Python types and C types:

```
signal1 = Signal(int) # Python types
signal2 = Signal(QUrl) # Qt Types
signal3 = Signal(int, str, int) # more than one type
signal4 = Signal((float,), (QDate,)) # optional types
```

In addition to that, it can receive also a named argument name that defines the signal name. If nothing is passed, the new signal will have the same name as the variable that it is being assigned to.

```
# TODO
signal5 = Signal(int, name='rangeChanged')
# ...
rangeChanged.emit(...)
```

Another useful option of Signal is the arguments name, useful for QML applications to refer to the emitted values by name:

```
sumResult = Signal(int, arguments=['sum'])

Connections {
    target: ...
    function onSumResult(sum) {
        // do something with 'sum'
    }
}
```

The Slot Class

Slots in QObject-derived classes should be indicated by the decorator @QtCore.Slot(). Again, to define a signature just pass the types similar to the QtCore.Signal() class.

```
@Slot(str)
def slot_function(self, s):
    ...
```

Slot() also accepts a name and a result keyword. The result keyword defines the type that will be returned and can be a C or Python type. The name keyword behaves the same way as in Signal(). If nothing is passed as name then the new slot will have the same name as the function that is being decorated.

We recommend marking all methods used by signal connections with a @QtCore.Slot() decorator. Not doing causes run-time overhead due to the method being added to the QMetaObject when creating the connection. This is particularly important for QObject classes registered with QML, where missing decorators can introduce bugs.

Missing decorators can be diagnosed by setting activating warnings of the logging category qt.pyqtlib.pyqtlib; for example by setting the environment variable:

```
export QT_LOGGING_RULES="qt.pyqtlib.pyqtlib.warning=true"
```

Overloading Signals and Slots with Different Types

It is actually possible to use signals and slots of the same name with different parameter type lists. This is legacy from Qt 5 and not recommended for new code. In Qt 6, signals have distinct names for different types.

The following example uses two handlers for a Signal and a Slot to showcase the different functionality.

```
import sys
from PySide6.QtWidgets import QApplication, QPushButton
from PySide6.QtCore import QObject, Signal, Slot

class Communicate(QObject):
    # create two new signals on the fly: one will handle
    # int type, the other will handle strings
    speak = Signal((int,), (str,))

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

        self.speak[int].connect(self.say_something)
        self.speak[str].connect(self.say_something)

    # define a new slot that receives a C 'int' or a 'str'
    # and has 'say_something' as its name
    @Slot(int)
    @Slot(str)
    def say_something(self, arg):
        if isinstance(arg, int):
            print("This is a number:", arg)
        elif isinstance(arg, str):
            print("This is a string:", arg)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    someone = Communicate()

    # emit 'speak' signal with different arguments.
    # we have to specify the str as int is the default
    someone.speak.emit(10)
    someone.speak[str].emit("Hello everybody!")
```

Specifying Signals and Slots by Method Signature Strings

Signals and slots can also be specified as C++ method signature strings passed through the SIGNAL() and/or SLOT() functions:

```
from PySide6.QtCore import SIGNAL, SLOT

button.connect(SIGNAL("clicked(Qt::MouseButton)"),
               action_handler, SLOT("action1(Qt::MouseButton)"))
```

This is not normally recommended; it is only needed for a few cases where signals are only accessible

via QMetaObject (QAxObject, QAxWidget, QDBusInterface or QWizardPage::registerField()):

```
wizard.registerField("text", line_edit, "text",
                    SIGNAL("textChanged(QString)"))
```

The signature strings can be found by querying QMetaMethod.methodSignature() when introspecting QMetaObject:

```
mo = widget.metaObject()
for m in range(mo.methodOffset(), mo.methodCount()):
    print(mo.method(m).methodSignature())
```

Slots should be decorated using [@Slot](#).

Signals and Slots

Due to the nature of Qt, QObjects require a way to communicate, and that's the reason for this mechanism to be a **central feature of Qt**.

In simple terms, you can understand **Signal and Slots** in the same way you interact with the lights in your house. When you move the light switch (signal) you get a result which may be that your light bulbs are switched on/off (slot).

While developing interfaces, you can get a real example by the effect of clicking a button: the 'click' will be the signal, and the slot will be what happens when that button is clicked, like closing a window, saving a document, etc.

Note

If you have experience with other frameworks or toolkits, it's likely that you read a concept called 'callback'. Leaving the implementation details aside, a callback will be related to a notification function, passing a pointer to a function in case it's required due to the events that happen in your program. This approach might sound similar, but there are essential differences that make it an unintuitive approach, like ensuring the type correctness of callback arguments, and some others.

All classes that inherit from QObject or one of its subclasses, like QWidget can contain signals and slots. **Signals are emitted by objects** when they change their state in a way that may be interesting to other objects. This is all the object does to communicate. It does not know or care whether anything is receiving the signals it emits. This is true information encapsulation, and ensures that the object can be used as a software component.

Slots can be used for receiving signals, but they are also normal member functions. Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it. This ensures that truly independent components can be created with Qt.

You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you need. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.)

Qt's widgets have many predefined signals and slots. For example, `QAbstractButton` (base class of buttons in Qt) has a `clicked()` signal and `QLineEdit` (single line input field) has a slot named `clear()`. So, a text input field with a button to clear the text could be implemented by placing a `QToolButton` to the right of the `QLineEdit` and connecting its `clicked()` signal to the slot `clear()`. This is done using the `connect()` method of the signal:

```
button = QToolButton()
line_edit = QLineEdit()
button.clicked.connect(line_edit.clear)
```

`connect()` returns a `QMetaObject.Connection` object, which can be used with the `disconnect()` method to sever the connection.

Signals can also be connected to free functions:

```
import sys
from PySide6.QtWidgets import QApplication, QPushButton

def function():
    print("The 'function' has been called!")

app = QApplication()
button = QPushButton("Call function")
button.clicked.connect(function)
button.show()
sys.exit(app.exec())
```

Connections can be spelled out in code or, for widget forms, designed in the [Signal-Slot Editor](#) of *Qt Widgets Designer*.

The Signal Class

When writing classes in Python, signals are declared as class level variables of the class `QtCore.Signal()`. A `QWidget`-based button that emits a `clicked()` signal could look as follows:

```
from PySide6.QtCore import Qt, Signal
from PySide6.QtWidgets import QWidget

class Button(QWidget):

    clicked = Signal(Qt.MouseButton)

    ...

    def mousePressEvent(self, event):
```

```
self.clicked.emit(event.button())
```

The constructor of Signal takes a tuple or a list of Python types and C types:

```
signal1 = Signal(int) # Python types
signal2 = Signal(QUrl) # Qt Types
signal3 = Signal(int, str, int) # more than one type
signal4 = Signal((float,), (QDate,)) # optional types
```

In addition to that, it can receive also a named argument name that defines the signal name. If nothing is passed, the new signal will have the same name as the variable that it is being assigned to.

```
# TODO
signal5 = Signal(int, name='rangeChanged')
# ...
rangeChanged.emit(...)
```

Another useful option of Signal is the arguments name, useful for QML applications to refer to the emitted values by name:

```
sumResult = Signal(int, arguments=['sum'])

Connections {
    target: ...
    function onSumResult(sum) {
        // do something with 'sum'
    }
}
```

The Slot Class

Slots in QObject-derived classes should be indicated by the decorator @QtCore.Slot(). Again, to define a signature just pass the types similar to the QtCore.Signal() class.

```
@Slot(str)
def slot_function(self, s):
    ...
```

Slot() also accepts a name and a result keyword. The result keyword defines the type that will be returned and can be a C or Python type. The name keyword behaves the same way as in Signal(). If nothing is passed as name then the new slot will have the same name as the function that is being decorated.

We recommend marking all methods used by signal connections with a @QtCore.Slot() decorator. Not doing causes run-time overhead due to the method being added to the QMetaObject when creating the connection. This is particularly important for QObject classes registered with QML, where missing decorators can introduce bugs.

Missing decorators can be diagnosed by setting activating warnings of the logging category qt.pyqt5.libpyqt5; for example by setting the environment variable:

```
export QT_LOGGING_RULES="qt.pyqt5.libpyqt5.warning=true"
```

Overloading Signals and Slots with Different Types

It is actually possible to use signals and slots of the same name with different parameter type lists. This is legacy from Qt 5 and not recommended for new code. In Qt 6, signals have distinct names for different types.

The following example uses two handlers for a Signal and a Slot to showcase the different functionality.

```
import sys
from PySide6.QtWidgets import QApplication, QPushButton
from PySide6.QtCore import QObject, Signal, Slot

class Communicate(QObject):
    # create two new signals on the fly: one will handle
    # int type, the other will handle strings
    speak = Signal((int,), (str,))

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

        self.speak[int].connect(self.say_something)
        self.speak[str].connect(self.say_something)

    # define a new slot that receives a C 'int' or a 'str'
    # and has 'say_something' as its name
    @Slot(int)
    @Slot(str)
    def say_something(self, arg):
        if isinstance(arg, int):
            print("This is a number:", arg)
        elif isinstance(arg, str):
            print("This is a string:", arg)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    someone = Communicate()

    # emit 'speak' signal with different arguments.
    # we have to specify the str as int is the default
    someone.speak.emit(10)
    someone.speak[str].emit("Hello everybody!")
```

Specifying Signals and Slots by Method Signature Strings

Signals and slots can also be specified as C++ method signature strings passed through the SIGNAL() and/or SLOT() functions:

```
from PySide6.QtCore import SIGNAL, SLOT

button.connect(SIGNAL("clicked(Qt::MouseButton)"),
               action_handler, SLOT("action1(Qt::MouseButton)"))
```

This is not normally recommended; it is only needed for a few cases where signals are only accessible

via QMetaObject (QAxObject, QAxWidget, QDBusInterface or QWizardPage::registerField()):

```
wizard.registerField("text", line_edit, "text",
                    SIGNAL("textChanged(QString)"))
```

The signature strings can be found by querying QMetaMethod.methodSignature() when introspecting QMetaObject:

```
mo = widget.metaObject()
for m in range(mo.methodOffset(), mo.methodCount()):
    print(mo.method(m).methodSignature())
```

Slots should be decorated using [@Slot](#).

Displaying Data Using a Table Widget

If you want to display data arranged in a table, use a QTableWidget to do so, without dealing with much configuration.

Notice that using a QTableWidget is not the only path to display information in tables. You can also create a data model and display it using a QTableView, but that is not in the scope of this tutorial.

Note

This Widget is a ready-to-use version of something you can customize further on. To know more about the Model/View architecture in Qt, refer to its [official documentation](#).

1. Import QTableWidget, QTableWidgetItem, and QColor to display background colors:
2. **import sys**
3. **from PySide6.QtGui import QColor**
4. **from PySide6.QtWidgets import (QApplication, QTableWidget,**
5. **QTableWidgetItem)**
6. Create a simple data model containing the list of names and hex codes for different colors:
7. `colors = [("Red", "#FF0000"),`
8. `("Green", "#00FF00"),`
9. `("Blue", "#0000FF"),`
10. `("Black", "#000000"),`
11. `("White", "#FFFFFF"),`
12. `("Electric Green", "#41CD52"),`
13. `("Dark Blue", "#222840"),`

```

14. ("Yellow", "#F9E56d")
15. Define a function to translate the hex code into an RGB equivalent:
16. def get_rgb_from_hex(code):
17.     code_hex = code.replace("#", "")
18.     rgb = tuple(int(code_hex[i:i+2], 16) for i in (0, 2, 4))
19.     return QColor.fromRgb(rgb[0], rgb[1], rgb[2])
20. Initialize the QApplication singleton:
21. app = QApplication()
22. Configure the QTableWidget to have a number of rows equivalent to the amount of items from
    the colors structure, and a number of columns with the members of one color entry, plus one. You can set the
    column name using the setHorizontalHeaderLabels as described below:
23. table = QTableWidget()
24. table.setRowCount(len(colors))
25. table.setColumnCount(len(colors[0]) + 1)
26. table.setHorizontalHeaderLabels(["Name", "Hex Code", "Color"])

```

Note



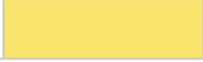
the reason of using + 1 is to include a new column where we can display the color.

```

27. Iterate the data structure, create the QTableWidgetItem instances, and add them into the table using
    a x, y coordinate. Here the data is being assigned row-per-row:
28. for i, (name, code) in enumerate(colors):
29.     item_name = QTableWidgetItem(name)
30.     item_code = QTableWidgetItem(code)
31.     item_color = QTableWidgetItem()
32.     item_color.setBackground(get_rgb_from_hex(code))
33.     table.setItem(i, 0, item_name)
34.     table.setItem(i, 1, item_code)
35.     table.setItem(i, 2, item_color)
36. Show the table and execute the QApplication.
37. table.show()
38. sys.exit(app.exec())

```

The final application will look like this:

	Name	Hex Code	Color
1	Red	#FF0000	
2	Green	#00FF00	
3	Blue	#0000FF	
4	Black	#000000	
5	White	#FFFFFF	
6	Electric Green	#41CD52	
7	Dark Blue	#222840	
8	Yellow	#F9E56d	

Displaying Data Using a Tree Widget

If you want to display data arranged in a tree, use a `QTreeWidget` to do so.

Notice that using a `QTreeWidget` is not the only path to display information in trees. You can also create a data model and display it using a `QTreeView`, but that is not in the scope of this tutorial.

Note

This Widget is a ready-to-use version of something you can customize further on. To know more about the Model/View architecture in Qt, refer to its [official documentation](#).

```

1. Import QTreeWidget and QTreeWidgetItem for this application:
2. import sys
3. from PySide6.QtWidgets import QApplication, QTreeWidget, QTreeWidgetItem

4. Define a dictionary with project structures to display the information as a tree, with files belonging to each project:
5. data = {"Project A": ["file_a.py", "file_a.txt", "something.xls"],
6.        "Project B": ["file_b.csv", "photo.jpg"],
7.        "Project C": []}

8. Initialize the QApplication singleton:
9. app = QApplication()
```

```

10. Configure the QTreeWidget to have two columns, one for the item name, and the other for item type
    information of the files in the project directories. You can set the column name with
    the setHeaderLabels as described below:
11. tree = QTreeWidget()
12. tree.setColumnCount(2)
13. tree.setHeaderLabels(["Name", "Type"])

14. Iterate the data structure, create the QTreeWidgetItem elements, and add the corresponding children
    to each parent. We also extract the extension name for only the files and add them into the second column. In
    the constructor, you can see that each element (QTreeWidgetItem) is added to different columns of the
    tree (QTreeWidget).
15. items = []
16. for key, values in data.items():
17.     item = QTreeWidgetItem([key])
18.     for value in values:
19.         ext = value.split(".")[-1].upper()
20.         child = QTreeWidgetItem([value, ext])
21.         item.addChild(child)
22.     items.append(item)
23.
24. tree.insertTopLevelItems(0, items)

25. Show the tree and execute the QApplication.
26. tree.show()
27. sys.exit(app.exec())

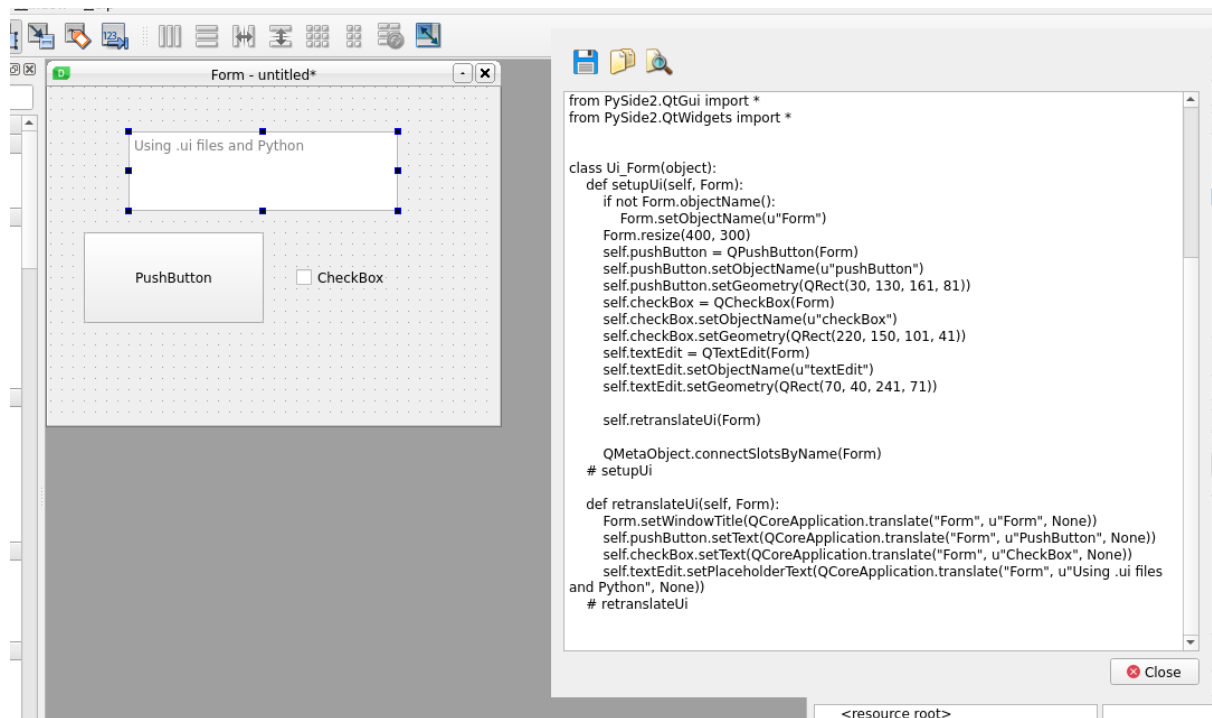
```

The final application will look like this:

Name	Type
▼ Project A	
file_a.py	PY
file_a.txt	TXT
something.xls	XLS
▼ Project B	
file_b.csv	CSV
photo.jpg	JPG
Project C	

Using .ui files from Designer or QtCreator with `QUiLoader` and `pyside6-uic`

This page describes the use of [Qt Widgets Designer](#) to create graphical interfaces based on Qt Widgets for your Qt for Python project. *Qt Widgets Designer* is a graphical UI design tool which is available as a standalone binary (pyside6-designer) or embedded into the [Qt Creator IDE](#). Its use within *Qt Creator* is described at [Using Qt Widgets Designer](#).



The designs are stored in .ui files, which is an XML-based format. It will be converted to Python or C++ code populating a widget instance at project build time by the [pyside6-uic](#) tool.

To create a new Qt Design Form in *Qt Creator*, choose File/New File Or Project and "Main Window" for template. Save it as mainwindow.ui. Add a QPushButton to the center of the centralwidget.

Your file mainwindow.ui should look something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>MainWindow</class>
  <widget class="QMainWindow" name="MainWindow">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>400</width>
        <height>300</height>
      </rect>
    </property>
    <property name="windowTitle">
      <string>MainWindow</string>
    </property>
    <widget class="QWidget" name="centralWidget">
      <widget class="QPushButton" name="pushButton">
        <property name="geometry">
          <rect>
            <x>110</x>
            <y>80</y>
```



```

    <width>201</width>
    <height>81</height>
  </rect>
</property>
<property name="text">
  <string>PushButton</string>
</property>
</widget>
</widget>
<widget class="QMenuBar" name="menuBar">
  <property name="geometry">
    <rect>
      <x>0</x>
      <y>0</y>
      <width>400</width>
      <height>20</height>
    </rect>
  </property>
</widget>
<widget class="QToolBar" name="mainToolBar">
  <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
  </attribute>
  <attribute name="toolBarBreak">
    <bool>>false</bool>
  </attribute>
</widget>
<widget class="QStatusBar" name="statusBar"/>
</widget>
<layoutdefault spacing="6" margin="11"/>
<resources/>
<connections/>
</ui>

```

Now we are ready to decide how to use the **UI file** from Python.

Option A: Generating a Python class

The standard way to interact with a **UI file** is to generate a Python class from it. This is possible thanks to the `pyside6-uis` tool. To use this tool, you need to run the following command on a console:

```
pyside6-uis mainwindow.ui -o ui_mainwindow.py
```

We redirect all the output of the command to a file called `ui_mainwindow.py`, which will be imported directly:

```
from ui_mainwindow import Ui_MainWindow
```

Now to use it, we should create a personalized class for our widget to **setup** this generated design.

To understand the idea, let's take a look at the whole code:

```

import sys
from PySide6.QtWidgets import QApplication, QMainWindow
from PySide6.QtCore import QFile
from ui_mainwindow import Ui_MainWindow

class MainWindow(QMainWindow):
    def __init__(self):
        super(MainWindow, self).__init__()
        self.ui = Ui_MainWindow()
        self.ui.setupUi(self)

if __name__ == "__main__":
    app = QApplication(sys.argv)

    window = MainWindow()
    window.show()

    sys.exit(app.exec())

```

What is inside the *if* statement is already known from the previous examples, and our new basic class contains only two new lines that are in charge of loading the generated python class from the UI file:

```

self.ui = Ui_MainWindow()
self.ui.setupUi(self)

```

Note

You must run `pyside6-uc` again every time you make changes to the **UI file**.

Option B: Loading it directly

To load the UI file directly, we will need a class from the **QtUiTools** module:

```

from PySide6.QtUiTools import QUiLoader

```

The `QUiLoader` lets us load the **ui file** dynamically and use it right away:

```

ui_file = QFile("mainwindow.ui")
ui_file.open(QFile.ReadOnly)

loader = QUiLoader()
window = loader.load(ui_file)
window.show()

```

The complete code of this example looks like this:

```

# File: main.py
import sys
from PySide6.QtUiTools import QUiLoader
from PySide6.QtWidgets import QApplication
from PySide6.QtCore import QFile, QIODevice

if __name__ == "__main__":

```

```

app = QApplication(sys.argv)

ui_file_name = "mainwindow.ui"
ui_file = QFile(ui_file_name)
if not ui_file.open(QIODevice.ReadOnly):
    print(f"Cannot open {ui_file_name}: {ui_file.errorString()}")
    sys.exit(-1)
loader = QUiLoader()
window = loader.load(ui_file)
ui_file.close()
if not window:
    print(loader.errorString())
    sys.exit(-1)
window.show()

sys.exit(app.exec())

```

Then to execute it we just need to run the following on a command prompt:

```
python main.py
```

Note

QUiLoader uses `connect()` calls taking the function signatures as string arguments for signal/slot connections. It is thus unable to handle Python types like `str` or `list` from custom widgets written in Python since these types are internally mapped to different C++ types.

Custom Widgets in Qt Widgets Designer

Qt Widgets Designer is able to use user-provided (custom) widgets. They are shown in the widget box and can be dragged onto the form just like Qt's widgets (see [Using Custom Widgets with Qt Widgets Designer](#)). Normally, this requires implementing the widget as a plugin to *Qt Widgets Designer* written in C++ implementing its [QDesignerCustomWidgetInterface](#).

Qt for Python provides a simple interface for this which is similar to [registerCustomWidget\(\)](#).

The widget needs to be provided as a Python module, as shown by the [WigglyWidget Example](#) (file `wigglywidget.py`) or the [Task Menu Extension Example](#) (file `tictactoe.py`).

Registering this with *Qt Widgets Designer* is done by providing a registration script named `register*.py` and pointing the path-type environment variable `PYSIDE_DESIGNER_PLUGINS` to the directory.

The code of the registration script looks as follows:

```

# File: registerwigglywidget.py
from wigglywidget import WigglyWidget

import QtDesigner

TOOLTIP = "A cool wiggly widget (Python)"

```

```

DOM_XML = """
<ui language='c++'>
  <widget class='WigglyWidget' name='wigglyWidget'>
    <property name='geometry'>
      <rect>
        <x>0</x>
        <y>0</y>
        <width>400</width>
        <height>200</height>
      </rect>
    </property>
    <property name='text'>
      <string>Hello, world</string>
    </property>
  </widget>
</ui>
"""

QPyDesignerCustomWidgetCollection.registerCustomWidget(WigglyWidget, module="wigglywidget",
                                                         tool_tip=TOOLTIP, xml=DOM_XML)

```

QPyDesignerCustomWidgetCollection provides an implementation of [QDesignerCustomWidgetCollectionInterface](#) exposing custom widgets to *Qt Widgets Designer* with static convenience functions for registering types or adding instances of [QDesignerCustomWidgetInterface](#) .

The function [registerCustomWidget\(\)](#) is used to register a widget type with *Qt Widgets Designer*. In the simple case, it can be used like `QUiLoader.registerCustomWidget()`. It takes the custom widget type and some optional keyword arguments passing values that correspond to the getters of [QDesignerCustomWidgetInterface](#) :

When launching *Qt Widgets Designer* via its launcher `pyside6-designer`, the custom widget should be visible in the widget box.

For advanced usage, it is also possible to pass the function an implementation of the class `QDesignerCustomWidgetInterface` instead of the type to [addCustomWidget\(\)](#). This is shown in `taskmenuextension` example, where a custom context menu is registered for the custom widget. The example is a port of the corresponding C++ [Task Menu Extension Example](#) .

Troubleshooting the Qt Widgets Designer Plugin

- The launcher `pyside6-designer` must be used. The standalone *Qt Widgets Designer* will not load the plugin.
- The menu item **Help/About Plugin** brings up a dialog showing the plugins found and potential load error messages.
- Check the console or Windows Debug view for further error messages.
- Due to the buffering of output by Python, error messages may appear only after *Qt Widgets Designer* has terminated.
- When building Qt for Python, be sure to set the `--standalone` option for the plugin to be properly installed.

Using `.qrc` Files (`pyside6-rcc`)

The [Qt Resource System](#) is a mechanism for storing binary files in an application.

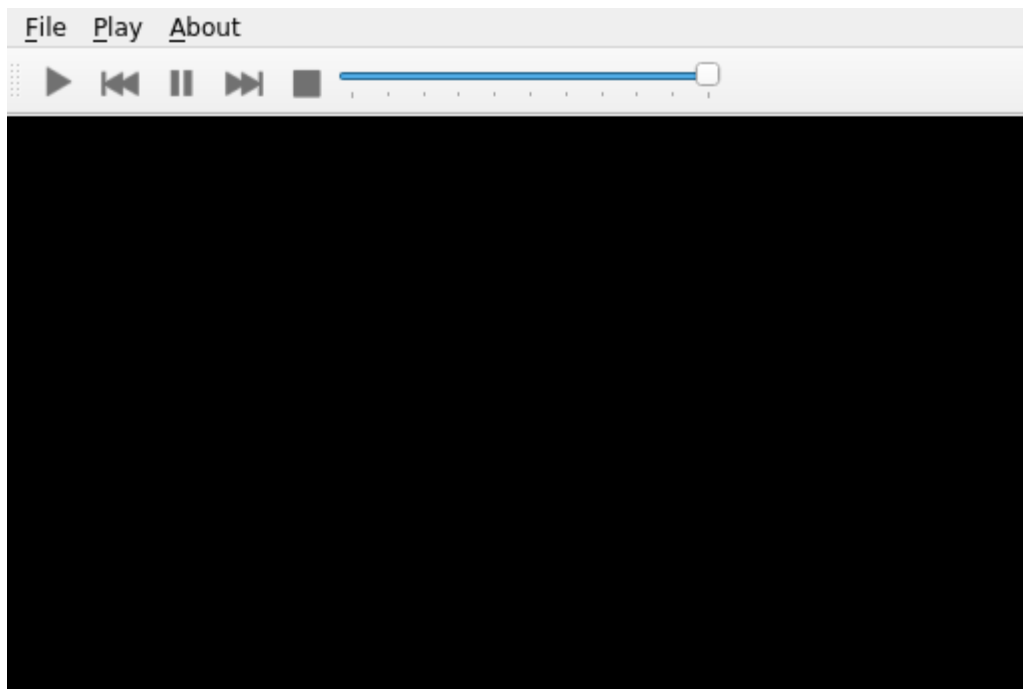
The files will be embedded into the application and be accessible for the `QFile` class and the constructors of the `QIcon` and `QPixmap` classes taking a file name by using a special file name starting with `:/`.

The most common uses are for custom images, icons, fonts, among others.

In this tutorial you will learn how to load custom images as button icons.

For inspiration, we will try to adapt the multimedia player example from Qt.

As you can see on the following image, the `QPushButton` that are used for the media actions (play, pause, stop, and so on) are using the default icons meant for such actions.



You could make the application more attractive by designing the icons, but in case you don't want to design them, you can download and use them.

[Download icons](#)



You can find more information about the `rcc` command, and `.qrc` file format, and the resource system in general in the [Qt Resource System](#) site.

The .qrc file

Before running any command, add information about the resources to a .qrc file. In the following example, notice how the resources are listed in icons.qrc

```
</ui>
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
  <file>icons/play.png</file>
  <file>icons/pause.png</file>
  <file>icons/stop.png</file>
  <file>icons/previous.png</file>
  <file>icons/forward.png</file>
</qresource>
</RCC>
```

Generating a Python file

Now that the icons.qrc file is ready, use the pyside6-rcc tool to generate a Python class containing the binary information about the resources

To do this, we need to run:

```
pyside6-rcc icons.qrc -o rc_icons.py
```

The -o option lets you specify the output filename, which is rc_icons.py in this case.

To use the generated file, add the following import at the top of your main Python file:

```
import rc_icons
```

Changes in the code

As you are modifying an existing example, you need to modify the following lines:

```
from PySide6.QtGui import QIcon, QKeySequence
playIcon = self.style().standardIcon(QStyle.SP_MediaPlay)
previousIcon = self.style().standardIcon(QStyle.SP_MediaSkipBackward)
pauseIcon = self.style().standardIcon(QStyle.SP_MediaPause)
nextIcon = self.style().standardIcon(QStyle.SP_MediaSkipForward)
stopIcon = self.style().standardIcon(QStyle.SP_MediaStop)
```

and replace them with the following:

```
from PySide6.QtGui import QIcon, QKeySequence, QPixmap
playIcon = QIcon(QPixmap(":/icons/play.png"))
previousIcon = QIcon(QPixmap(":/icons/previous.png"))
pauseIcon = QIcon(QPixmap(":/icons/pause.png"))
nextIcon = QIcon(QPixmap(":/icons/forward.png"))
stopIcon = QIcon(QPixmap(":/icons/stop.png"))
```

This ensures that the new icons are used instead of the default ones provided by the application theme. Notice that the lines are not consecutive, but are in different parts of the file.

After all your imports, add the following

```
import rc_icons
```

Now, the constructor of your class should look like this:

```
def __init__(self):
    super(MainWindow, self).__init__()

    self.playlist = QMediaPlaylist()
    self.player = QMediaPlayer()

    toolBar = QToolBar()
    self.addToolBar(toolBar)

    fileMenu = self.menuBar().addMenu("&File")
    openAction = QAction(QIcon.fromTheme("document-open"),
        "&Open...", self, shortcut=QKeySequence.Open,
        triggered=self.open)
    fileMenu.addAction(openAction)
    exitAction = QAction(QIcon.fromTheme("application-exit"), "E&xit",
        self, shortcut="Ctrl+Q", triggered=self.close)
    fileMenu.addAction(exitAction)

    playMenu = self.menuBar().addMenu("&Play")
    playIcon = QIcon(QPixmap(":/icons/play.png"))
    self.playAction = toolBar.addAction(playIcon, "Play")
    self.playAction.triggered.connect(self.player.play)
    playMenu.addAction(self.playAction)

    previousIcon = QIcon(QPixmap(":/icons/previous.png"))
    self.previousAction = toolBar.addAction(previousIcon, "Previous")
    self.previousAction.triggered.connect(self.previousClicked)
    playMenu.addAction(self.previousAction)

    pauseIcon = QIcon(QPixmap(":/icons/pause.png"))
    self.pauseAction = toolBar.addAction(pauseIcon, "Pause")
    self.pauseAction.triggered.connect(self.player.pause)
    playMenu.addAction(self.pauseAction)

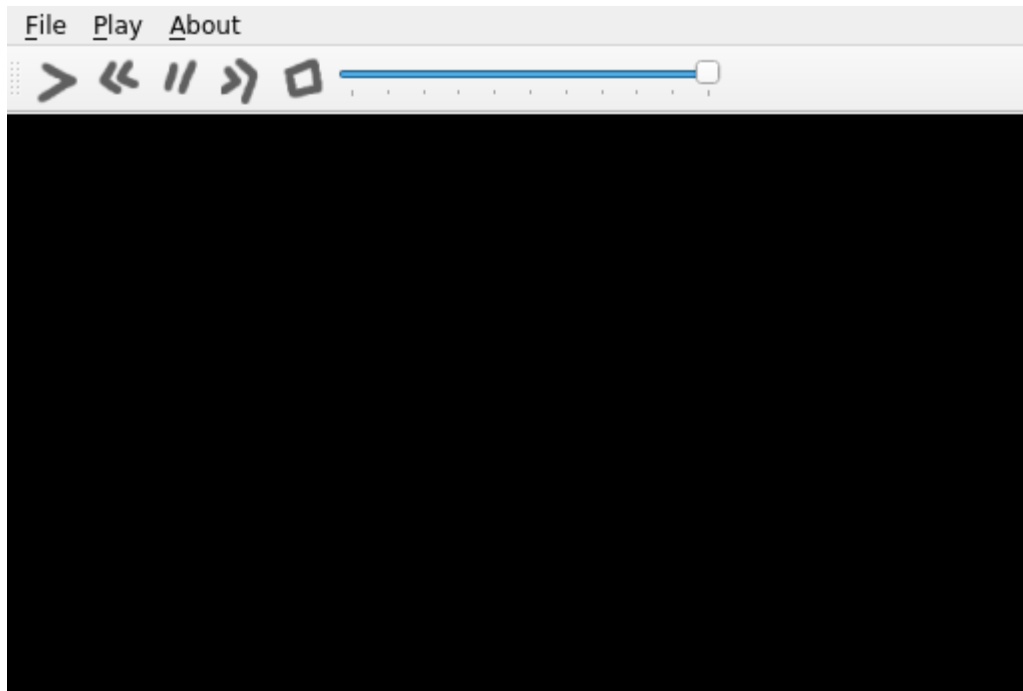
    nextIcon = QIcon(QPixmap(":/icons/forward.png"))
    self.nextAction = toolBar.addAction(nextIcon, "Next")
    self.nextAction.triggered.connect(self.playlist.next)
    playMenu.addAction(self.nextAction)

    stopIcon = QIcon(QPixmap(":/icons/stop.png"))
    self.stopAction = toolBar.addAction(stopIcon, "Stop")
    self.stopAction.triggered.connect(self.player.stop)
    playMenu.addAction(self.stopAction)

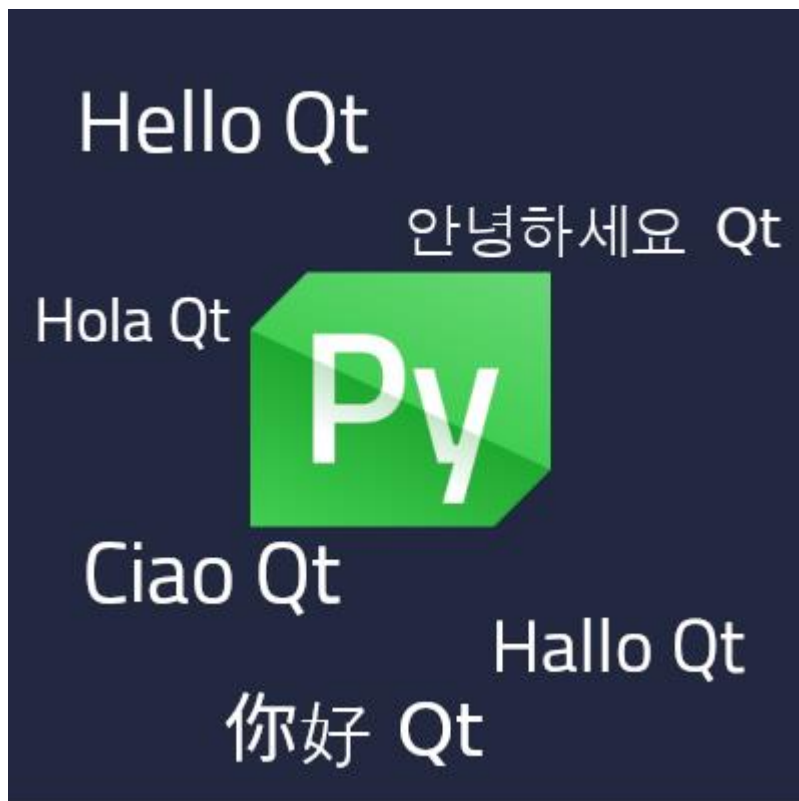
# many lines were omitted
```

Executing the example

Run the application by calling `python main.py` to checkout the new icon-set:



Translating Applications



Qt Linguist

[Qt Linguist](#) and its related tools can be used to provide translations for applications.

The [Qt Linguist Example](#) example illustrates this. The example is very simple, it has a menu and shows a list of programming languages with multiselection.

Translation works by passing the message strings through function calls that look up the translation. Each QObject instance provides a tr() function for that purpose. There is also QApplication.translate() for adding translated texts to non-QObject classes.

Qt ships its own translations containing the error messages and standard dialog captions.

The linguist example has a number of messages enclosed in self.tr(). The status bar message shown in response to a selection change uses a plural form depending on a count:

```
count = len(self.list_widget.selectionModel().selectedRows())
message = self.tr("%n language(s) selected", "", count)
```

The translation workflow for the example is as follows: The translated messages are extracted using the lupdate tool, producing XML-based .ts files:

```
pyside6-lupdate main.py -ts example_de.ts
```

If example_de.ts already exists, it will be updated with the new messages added to the code in-between.

If there are form files (.ui) and/or QML files (.qml) in the project, they should be passed to the pyside6-lupdate tool as well:

```
pyside6-lupdate main.py main.qml form.ui -ts example_de.ts
```

The source files generated by pyside6-uic from the form files should **not** be passed.

The lupdate mode of pyside6-project can also be used for this. It collects all source files and runs pyside6-lupdate when .ts file(s) are given in the .pyproject file:

```
pyside6-project lupdate .
```

.ts files are translated using *Qt Linguist*. Once this is complete, the files are converted to a binary form (.qm files):

```
pyside6-lrelease example_de.ts -qm example_de.qm
```

pyside6-project will build the .qm file automatically when .ts file(s) are given in the .pyproject file:

```
pyside6-project build .
```

To avoid having to ship the .qm files, it is recommend to put them into a Qt resource file along with icons and other applications resources (see [Using .qrc Files \(pyside6-rcc\)](#)). The resource file linguist.qrc provides the example_de.qm under :/translations:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource prefix="translations">
  <file>example_de.qm</file>
</qresource>
</RCC>
```

At runtime, the translations need to be loaded using the QTranslator class:

```
path = QLibraryInfo.location(QLibraryInfo.TranslationsPath)
translator = QTranslator(app)
if translator.load(QLocale.system(), 'qtbase', '_', path):
    app.installTranslator(translator)
translator = QTranslator(app)
path = './translations'
if translator.load(QLocale.system(), 'example', '_', path):
    app.installTranslator(translator)
```

The code first loads the translations shipped for Qt and then the translations of the applications loaded from resources.

The example can then be run in German:

```
LANG=de python main.py
```

GNU gettext

The [GNU gettext](#) module can be used to provide translations for applications.

The [GNU gettext Example](#) example illustrates this. The example is very simple, it has a menu and shows a list of programming languages with multiselection.

Translation works by passing the message strings through function calls that look up the translation. It is common to alias the main translation function to `_`. There is a special translation function for sentences that contain a plural form depending on a count (“{0} items(s) selected”). It is commonly aliased to `ngettext`.

Those functions are defined at the top:

```
import gettext
# ...
_ = None
ngettext = None
```

and later assigned as follows:

```
src_dir = Path(__file__).resolve().parent
try:
    translation = gettext.translation('example', localedir=src_dir / 'locales')
    if translation:
        translation.install()
        _ = translation.gettext
        ngettext = translation.ngettext
except FileNotFoundError:
    pass
if not _:
    _ = gettext.gettext
    ngettext = gettext.ngettext
```

This specifies that our translation file has the base name `example` and will be found in the source tree under `locales`. The code will try to load a translation matching the current language.

Messages to be translated look like:

```
file_menu = self.menuBar().addMenu(_("&File"))
```

The status bar message shown in response to a selection change uses a plural form depending on a count:

```
count = len(self._list_widget.selectionModel().selectedRows())
message = ngettext("{0} language selected",
                   "{0} languages selected", count).format(count)
```

The `ngettext()` function takes the singular form, plural form and the count. The returned string still contains the formatting placeholder, so it needs to be passed through `format()`.

In order to translate the messages to say German, a template file (`.pot`) is first created:

```
mkdir -p locales/de_DE/LC_MESSAGES
```

```
xgettext -L Python -o locales/example.pot main.py
```

This file has a few generic placeholders which can be replaced by the appropriate values. It is then copied to the de_DE/LC_MESSAGES directory.

```
cd locales/de_DE/LC_MESSAGES/  
cp ../example.pot .
```

Further adaptations need to be made to account for the German plural form and encoding:

```
"Project-Id-Version: PySide6 gettext example\n"  
"POT-Creation-Date: 2021-07-05 14:16+0200\n"  
"Language: de_DE\n"  
"MIME-Version: 1.0\n"  
"Content-Type: text/plain; charset=UTF-8\n"  
"Content-Transfer-Encoding: 8bit\n"  
"Plural-Forms: nplurals=2; plural=n != 1;\n"
```

Below, the translated messages can be given:

```
#: main.py:57  
msgid "&File"  
msgstr "&Datei"
```

Finally, the .pot is converted to its binary form (machine object file, .mo), which needs to be deployed:

```
msgfmt -o example.mo example.pot
```

The example can then be run in German:

```
LANG=de python main.py
```

Styling the Widgets Application


Qt Widgets application use a default theme depending on the platform. In some cases, there are system-wide configurations that modify the Qt theme, and applications are displayed differently.

However, you can take care of your own widgets and provide a custom style to each component. As an example, look at the following simple snippet:

```
import sys  
from PySide6.QtCore import Qt  
from PySide6.QtWidgets import QApplication, QLabel  
  
if __name__ == "__main__":  
    app = QApplication()  
    w = QLabel("This is a placeholder text")  
    w.setAlignment(Qt.AlignCenter)
```

```
w.show()
sys.exit(app.exec())
```

When you execute this code, you will see a simple QLabel aligned at the center, and with a placeholder text.



This is a placeholder text

You can style your application using the CSS-like syntax. For more information, see [Qt Style Sheets Reference](#).

A QLabel can be styled differently by setting some of its CSS properties, such as background-color and font-family, so let's see how does the code look like with these changes:

```
import sys
from PySide6.QtCore import Qt
from PySide6.QtWidgets import QApplication, QLabel

if __name__ == "__main__":
    app = QApplication()
    w = QLabel("This is a placeholder text")
    w.setAlignment(Qt.AlignCenter)
    w.setStyleSheet("""
        background-color: #262626;
        color: #FFFFFF;
        font-family: Titillium;
        font-size: 18px;
    """)
    w.show()
    sys.exit(app.exec())
```

Now when you run the code, notice that the QLabel looks different with your custom style:



This is a placeholder text

Note

If you don't have the font Titillium installed, you can try with any other you prefer. Remember you can list your installed fonts using `QFontDatabase`, specifically the `families()` method.

Styling each UI element separately like you did in the previous snippet is a lot of work. The easier alternative for this is to use Qt Style Sheets, which is one or more `.qss` files defining the style for the UI elements in your application.

More examples can be found in the [Qt Style Sheet Examples](#) documentation page.

Qt Style Sheets

Warning

Before starting modifying your application, keep in mind that you will be responsible for all the graphical details of the application. Altering margins, and sizes might end up looking strange or incorrect, so you need to be careful when altering the style. It's recommended to create a full new Qt style to cover all the possible corner cases.

A qss file is quite similar to a CSS file, but you need to specify the Widget component and optionally the name of the object:

```
QLabel {  
    background-color: red;  
}  
  
QLabel#title {  
    font-size: 20px;  
}
```

The first style defines a background-color for all QLabel objects in your application, whereas the later one styles the title object only.

Note

You can set object names with the `setObjectName(str)` function to any Qt object, for example: for a `label = QLabel("Test")`, you can write `label.setObjectName("title")`

Once you have a qss file for your application, you can apply it by reading the file and using the `QApplication.setStyleSheet(str)` function:

```
if __name__ == "__main__":
    app = QApplication()

    w = QWidget()
    w.show()

    with open("style.qss", "r") as f:
        _style = f.read()
        app.setStyleSheet(_style)

    sys.exit(app.exec())
```

Having a general qss file allows you to decouple the styling aspects of the code, without mixing it in the middle of the general functionality, and you can simply enable it or disable it.

Look at this new example, with more widgets components:

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

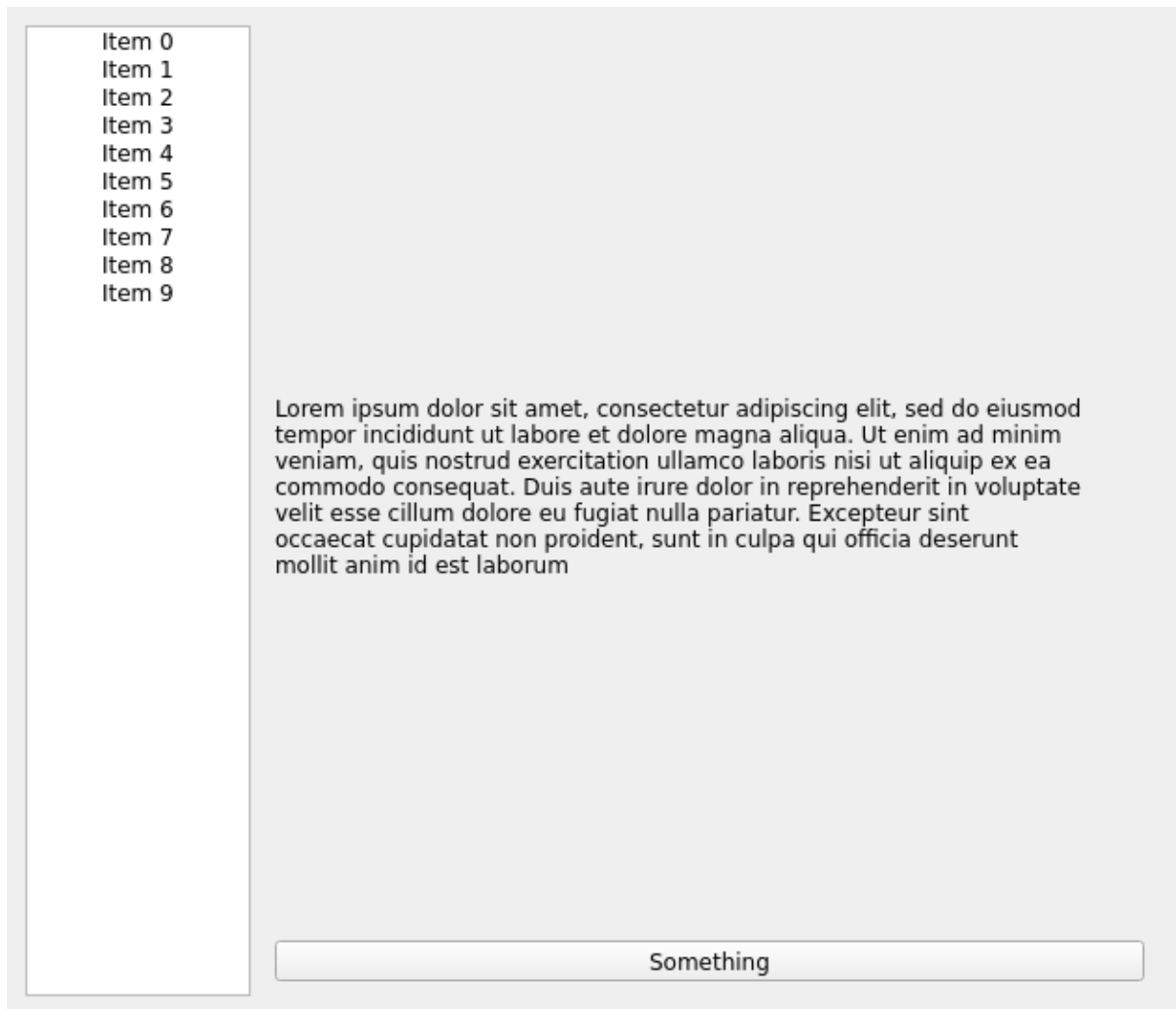
        menu_widget = QListWidget()
        for i in range(10):
            item = QListWidgetItem(f"Item {i}")
            item.setTextAlignment(Qt.AlignCenter)
            menu_widget.addItem(item)

        text_widget = QLabel(_placeholder)
        button = QPushButton("Something")

        content_layout = QVBoxLayout()
        content_layout.addWidget(text_widget)
        content_layout.addWidget(button)
        main_widget = QWidget()
        main_widget.setLayout(content_layout)

        layout = QHBoxLayout()
        layout.addWidget(menu_widget, 1)
        layout.addWidget(main_widget, 4)
        self.setLayout(layout)
```

This displays a two column widget, with a `QListWidget` on the left and a `QLabel` and a `QPushButton` on the right. It looks like this when you run the code:



If you add content to the previously described style.qss file, you can modify the look-n-feel of the previous example:

```
QListWidget {
    color: #FFFFFF;
    background-color: #33373B;
}

QListWidget::item {
    height: 50px;
}

QListWidget::item:selected {
    background-color: #2ABf9E;
}

QLabel {
    background-color: #FFFFFF;
    qproperty-alignment: AlignCenter;
}

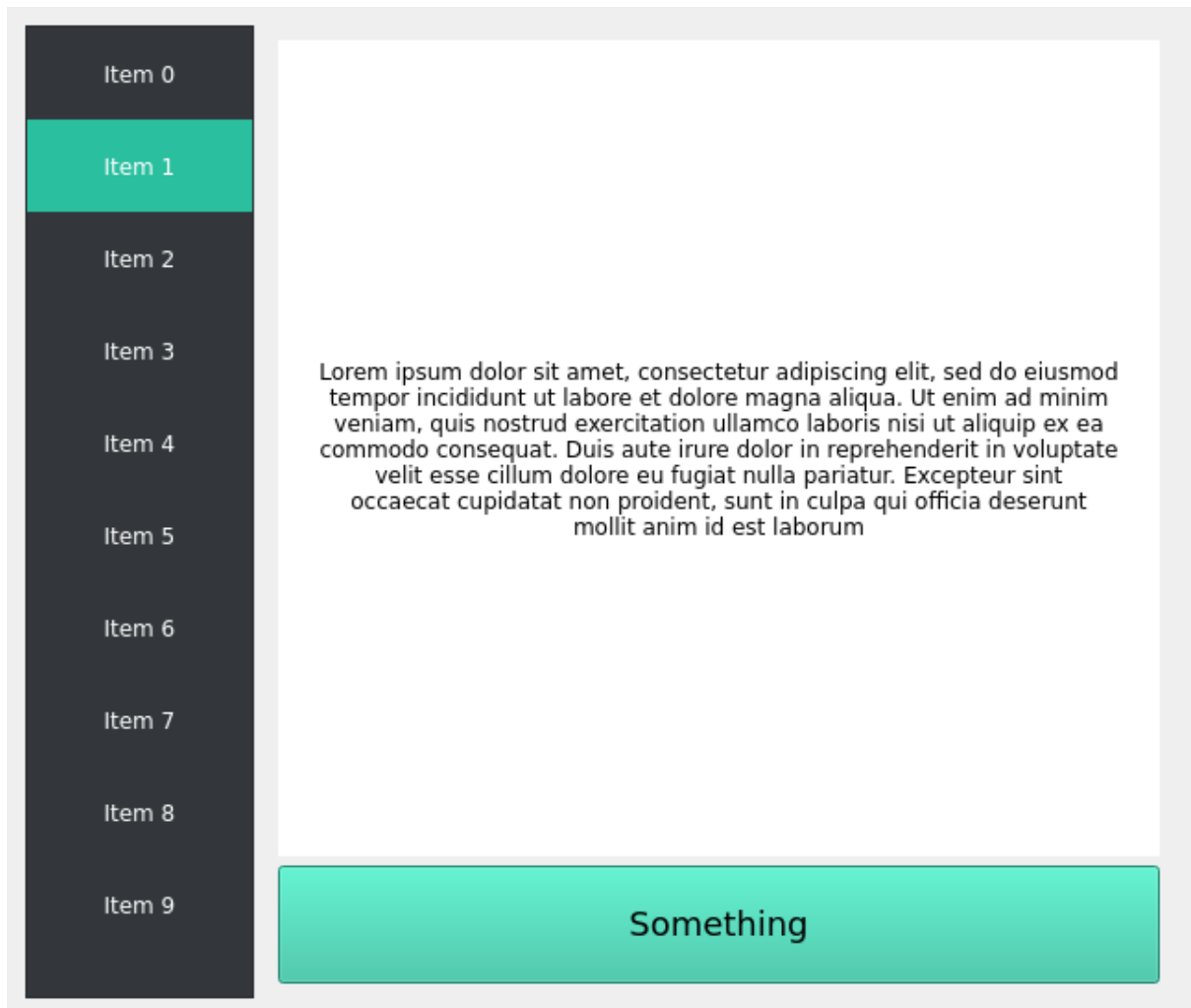
QPushButton {
    background-color: #2ABf9E;
    padding: 20px;
}
```



```
font-size: 18px;  
}
```

The style changes mainly the color of the different widgets, alter the alignment, and includes some spacing. You can also use state-based styling on the `QListWidget` *items* for example, to style them differently depending on whether they are *selected* or not.

After applying all the styling alternatives you explored in this topic, notice that the `QLabel` example looks a lot different now. Try running the code to check its new look:



You have the freedom to tune your style sheets and provide a really nice look-n-feel to all your applications.

Your First QtQuick/QML Application

[QML](#) is a declarative language that lets you develop applications faster than with traditional languages. It is ideal for designing the UI of your application because of its declarative nature. In QML, a user interface is specified as a tree of objects with properties. In this tutorial, we will show how to make a simple “Hello World” application with PySide6 and QML.

A PySide6/QML application consists, at least, of two different files - a file with the QML description of the user interface, and a python file that loads the QML file. To make things easier, let’s save both files in the same directory.

Here is a simple QML file called **view.qml**:

```
import QtQuick

Rectangle {
    id: main
    width: 200
    height: 200
    color: "green"

    Text {
        text: "Hello World"
        anchors.centerIn: main
    }
}
```

We start by importing **QtQuick**, which is a QML module.

The rest of the QML code is pretty straightforward for those who have previously used HTML or XML files. Basically, we are creating a green rectangle with the size *200*200*, and adding a **Text** element that reads “Hello World”. The code **anchors.centerIn: main** makes the text appear centered within the object with **id: main**, which is the Rectangle in this case.

Now, let’s see how the code looks on the PySide6. Let’s call it **main.py**:

```
import sys
from PySide6.QtWidgets import QApplication
from PySide6.QtQuick import QQuickView

if __name__ == "__main__":
    app = QApplication()
    view = QQuickView()

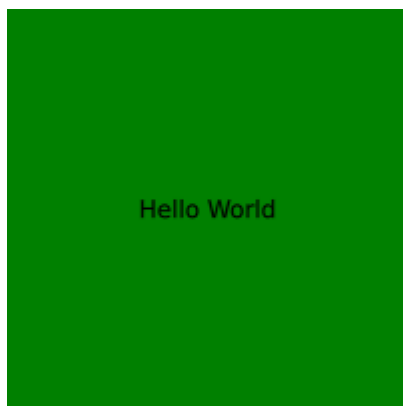
    view.setSource("view.qml")
    view.show()
    sys.exit(app.exec())
```

If you are already familiar with PySide6 and have followed our tutorials, you have already seen much of this code. The only novelties are that you must **import QtQuick** and set the source of the **QQuickView** object to the URL of your QML file. Then, similar to what you do with any Qt widget, you call **QQuickView.show()**.

Note

If you are programming for desktop, you should consider adding `view.setResizeMode(QQuickView.SizeRootObjectToView)` before showing the view.

When you execute the **main.py** script, you will see the following application:



Python-QML integration

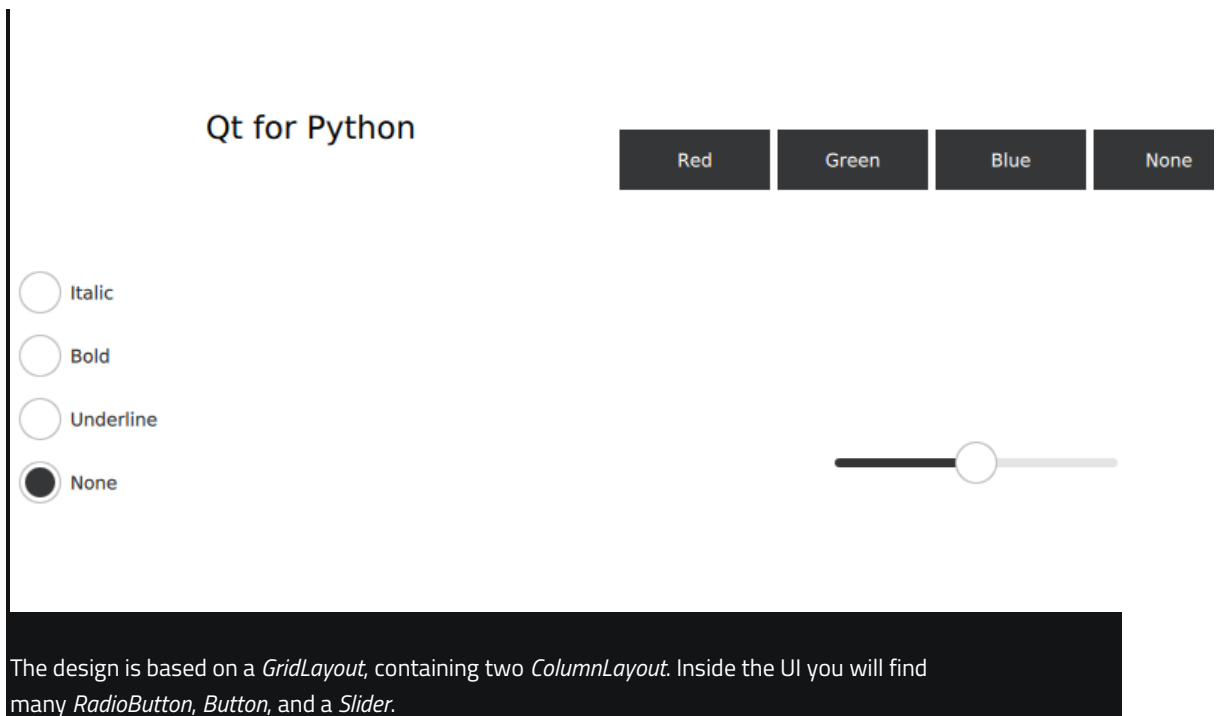
This tutorial provides a quick walk-through of a python application that loads, and interacts with a QML file. QML is a declarative language that lets you design UIs faster than a traditional language, such as C++. The QtQml and QtQuick modules provides the necessary infrastructure for QML-based UIs.

In this tutorial, you will learn how to integrate Python with a QML application. This mechanism will help us to understand how to use Python as a backend for certain signals from the UI elements in the QML interface. Additionally, you will learn how to provide a modern look to your QML application using one of the features from Qt Quick Controls 2.

The tutorial is based on an application that allow you to set many text properties, like increasing the font size, changing the color, changing the style, and so on. Before you begin, install the [PySide6](#) Python packages.

The following step-by-step process will guide you through the key elements of the QML based application and PySide6 integration:

1. First, let's start with the following QML-based UI:



2. With the QML file in place, you can load it from Python:

```
if __name__ == '__main__':
    app = QtGuiApplication(sys.argv)
    QQuickStyle.setStyle("Material")
    engine = QQmlApplicationEngine()

    # Get the path of the current directory, and then add the name
    # of the QML file, to load it.
    qml_file = Path(__file__).parent / 'view.qml'
    engine.load(qml_file)

    if not engine.rootObjects():
        sys.exit(-1)
```

Notice that we only need a `QQmlApplicationEngine` to load the QML file.

3. Define the Bridge class, containing all the logic for the element that will be register in QML:

```
# To be used on the @QmlElement decorator
# (QML_IMPORT_MINOR_VERSION is optional)
QML_IMPORT_NAME = "io.qt.textproperties"
QML_IMPORT_MAJOR_VERSION = 1

@QmlElement
class Bridge(QObject):

    @Slot(str, result=str)
    def getColor(self, s):
        if s.lower() == "red":
            return "#ef9a9a"
```

```

elif s.lower() == "green":
    return "#a5d6a7"
elif s.lower() == "blue":
    return "#90caf9"
else:
    return "white"

@Slot(float, result=int)
def getSize(self, s):
    size = int(s * 34)
    if size <= 0:
        return 1
    else:
        return size

@Slot(str, result=bool)
def getItalic(self, s):
    if s.lower() == "italic":
        return True
    else:
        return False

@Slot(str, result=bool)
def getBold(self, s):
    if s.lower() == "bold":
        return True
    else:
        return False

```

Notice that the registration happens thanks to the `QmlElement` decorator, that underneath uses the reference to the `Bridge` class and the variables `QML_IMPORT_NAME` and `QML_IMPORT_MAJOR_VERSION`.

- Now, go back to the QML file and connect the signals to the slots defined in the `Bridge` class:

```

Bridge {
    id: bridge
}

```

- Inside the `ApplicationWindow` we declare a component with the same name as the Python class, and provide an `id`:. This `id` will help you to get a reference to the element that was registered from Python.

```

RadioButton {
    id: italic
    Layout.alignment: Qt.AlignLeft
    text: "Italic"
    onToggled: {
        leftlabel.font.italic = bridge.getItalic(italic.text)
        leftlabel.font.bold = bridge.getBold(italic.text)
        leftlabel.font.underline = bridge.getUnderline(italic.text)
    }
}

```

The properties *Italic*, *Bold*, and *Underline* are mutually exclusive, this means only one can be active at any time. To achieve this each time we select one of these options, we check the three properties via the QML element

property as you can see in the above snippet. Only one of the three will return *True*, while the other two will return *False*, that is how we make sure only one is being applied to the text.

6. Each slot verifies if the selected option contains the text associated to the property:

```
@Slot(str, result=bool)
def getItalic(self, s):
    if s.lower() == "italic":
        return True
    else:
        return False
```

Returning *True* or *False* allows you to activate and deactivate the properties of the QML UI elements.

It is also possible to return other values that are not *Boolean*, like the slot in charge of returning the font size:

```
@Slot(float, result=int)
def getSize(self, s):
    size = int(s * 34)
    if size <= 0:
        return 1
    else:
```

7. Now, for changing the look of our application, you have two options:

1. Use the command line: execute the python file adding the option, `--style`:
2. `python main.py --style material`
3. Use a `qtquickcontrols2.conf` file:

```
[Controls]
Style=Material

[Universal]
Theme=System
Accent=Red

[Material]
Theme=Dark
Accent=Red
```

Then add it to your `.qrc` file:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource prefix="/">
    <file>qtquickcontrols2.conf</file>
</qresource>
</RCC>
```

Generate the `rc` file running, `pyside6-rcc style.qrc -o style_rc.py` And finally import it from your `main.py` script.

```
import sys
from pathlib import Path

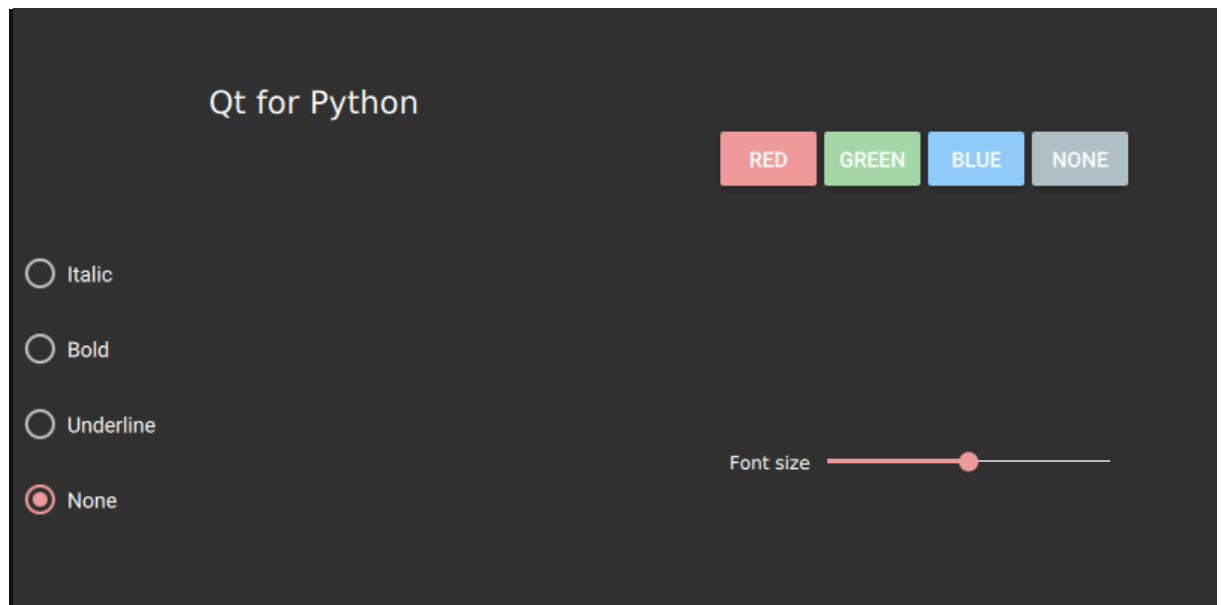
from PySide6.QtCore import QObject, Slot
```

```
from PySide6.QtGui import QApplication
from PySide6.QtQml import QQmlApplicationEngine, QmlElement
from PySide6.QtQuickControls2 import QQuickStyle

import style_rc
```

You can read more about this configuration file [here](#).

The final look of your application will be:



QML Application Tutorial

This tutorial provides a quick walk-through of a python application that loads a QML file. QML is a declarative language that lets you design UIs faster than a traditional language, such as C++. The QtQml and QtQuick modules provides the necessary infrastructure for QML-based UIs.

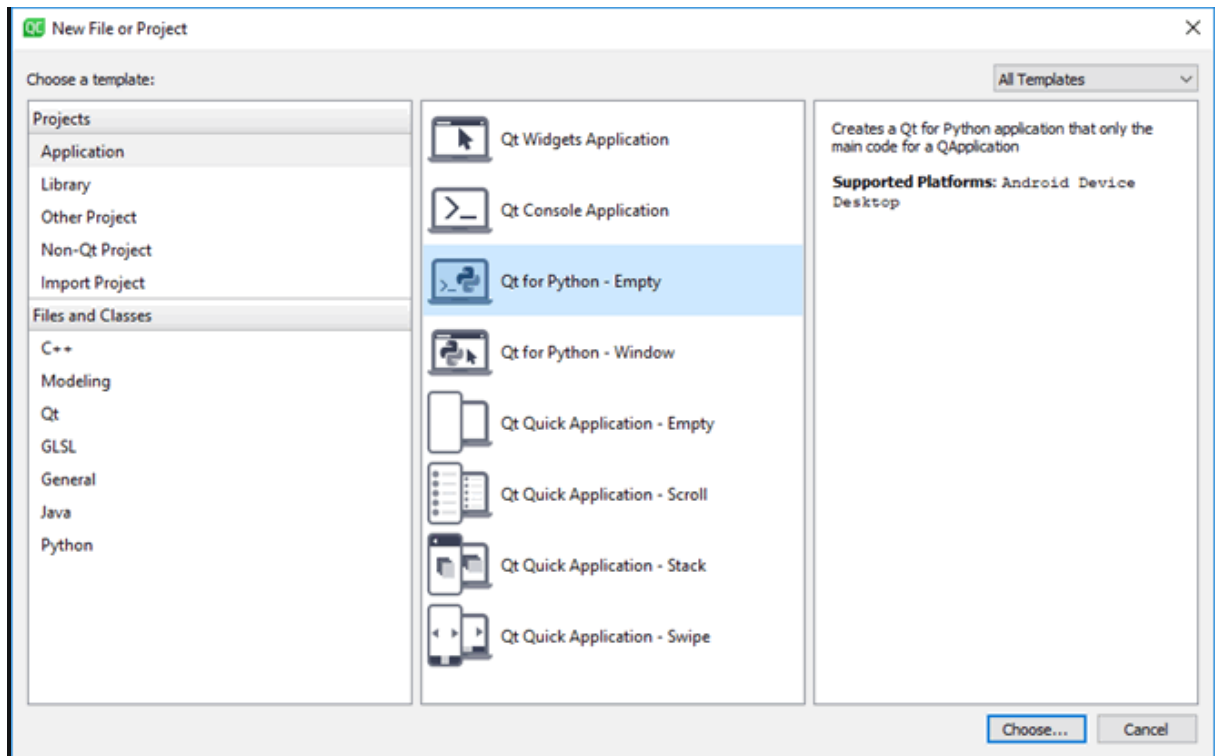
In this tutorial, you'll also learn how to provide data from Python as a QML initial property, which is then consumed by the ListView defined in the QML file.

Before you begin, install the following prerequisites:

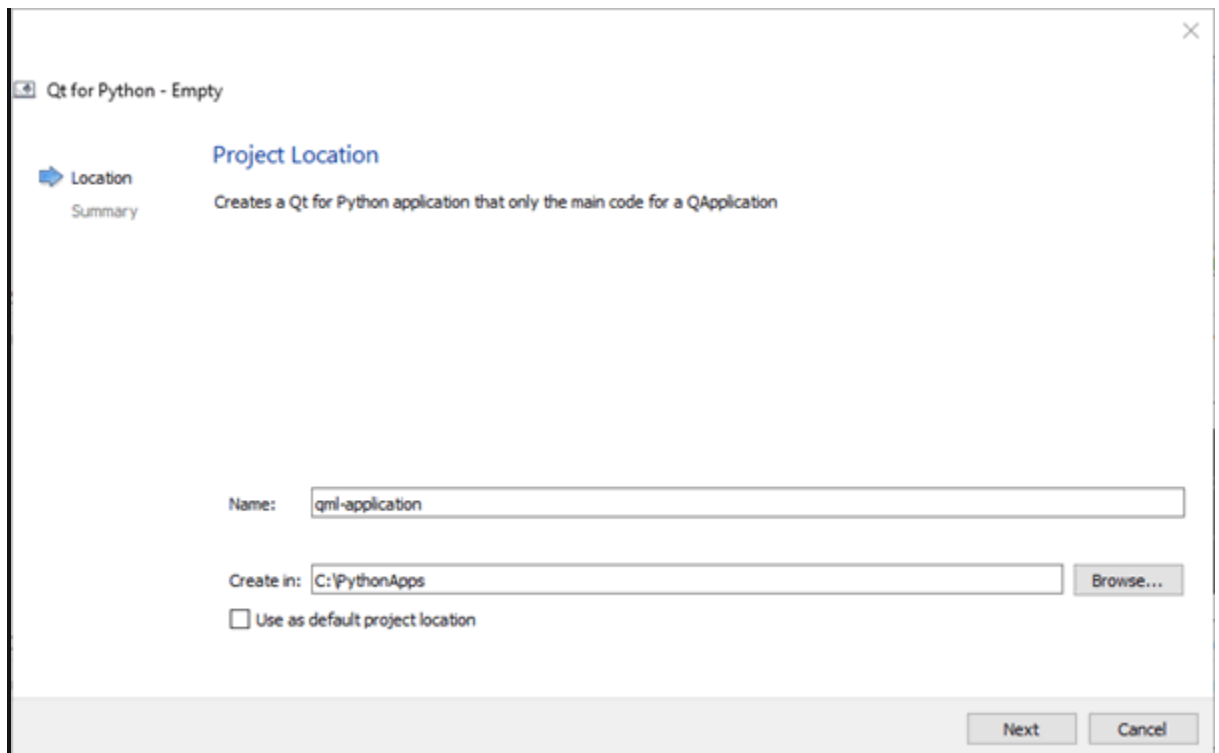
- The [PySide6](#) Python packages.
- *Qt Creator* from <https://download.qt.io>.

The following step-by-step instructions guide you through application development process using *Qt Creator*.

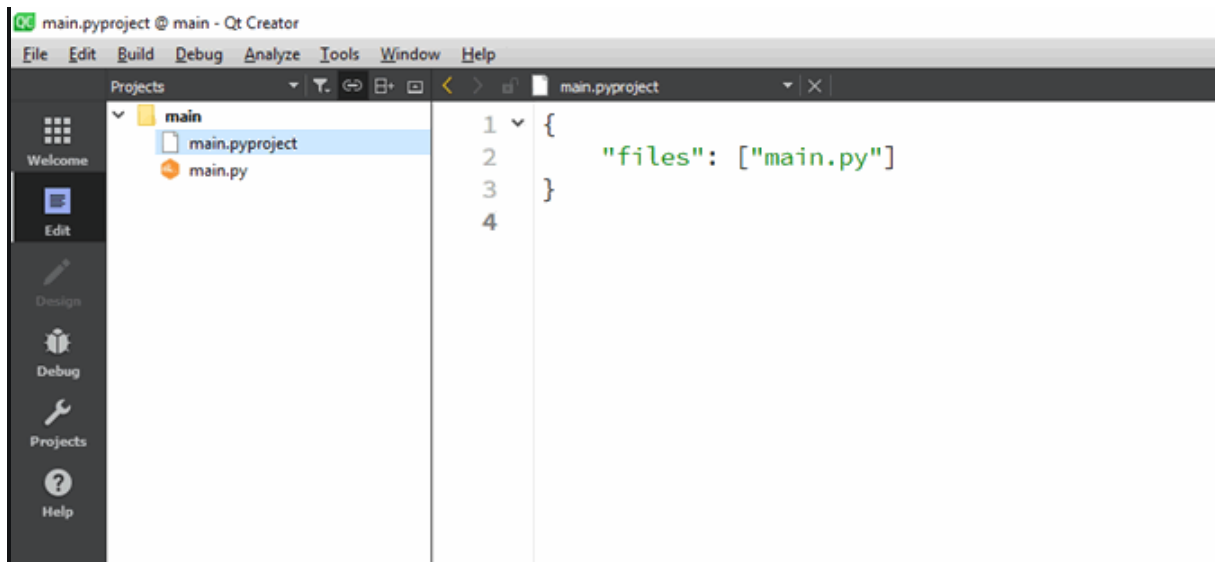
1. Open *Qt Creator* and select **File > New File or Project..** menu item to open following dialog:



2. Select **Qt for Python - Empty** from the list of application templates and select **Choose**.



3. Give a **Name** to your project, choose its location in the filesystem, and select **Finish** to create an empty `main.py` and `main.pyproject`.



This should create a main.py and `main.pyproject` files for the project.

4. Download [view.qml](#) and [logo.png](#) and move them to your project folder.
5. Double-click on main.pyproject to open it in edit mode, and append view.qml and logo.png to the **files** list. This is how your project file should look after this change:

```
{
  "files": ["main.py", "view.qml", "logo.png"]
}
```

6. Now that you have the necessary bits for the application, import the Python modules in your main.py, and download country data and format it:

```
import sys
import urllib.request
import json
from pathlib import Path

from PySide6.QtQuick import QQuickView
from PySide6.QtCore import QStringListModel, QUrl
from PySide6.QtGui import QApplication

if __name__ == '__main__':

    #get our data
    url = "http://country.io/names.json"
    response = urllib.request.urlopen(url)
    data = json.loads(response.read().decode('utf-8'))

    #Format and sort the data
    data_list = list(data.values())
    data_list.sort()
```

7. Now, set up the application window using PySide6.QtGui.QGuiApplication, which manages the application-wide settings.

```

import sys
import urllib.request
import json
from pathlib import Path

from PySide6.QtQuick import QQuickView
from PySide6.QtCore import QStringListModel, QUrl
from PySide6.QtGui import QApplication

if __name__ == '__main__':

    #get our data
    url = "http://country.io/names.json"
    response = urllib.request.urlopen(url)
    data = json.loads(response.read().decode('utf-8'))

    #Format and sort the data
    data_list = list(data.values())
    data_list.sort()

    #Set up the application window
    app = QApplication(sys.argv)
    view = QQuickView()
    view.setResizeMode(QQuickView.SizeRootObjectToView)

```

Note

Setting the resize policy is important if you want the root item to resize itself to fit the window or vice-a-versa. Otherwise, the root item will retain its original size on resizing the window.

8. You can now expose the `data_list` variable as a QML initial property, which will be consumed by the QML `ListView` item in `view.qml`.

```

import sys
import urllib.request
import json
from pathlib import Path

from PySide6.QtQuick import QQuickView
from PySide6.QtCore import QStringListModel, QUrl
from PySide6.QtGui import QApplication

if __name__ == '__main__':

    #get our data
    url = "http://country.io/names.json"
    response = urllib.request.urlopen(url)
    data = json.loads(response.read().decode('utf-8'))

    #Format and sort the data
    data_list = list(data.values())
    data_list.sort()

```

```
#Set up the application window
app = QtGuiApplication(sys.argv)
view = QQuickView()
view.setResizeMode(QQuickView.SizeRootObjectToView)
```

```
#Expose the list to the Qml code
my_model = QStringListModel()
my_model.setStringList(data_list)
view.setInitialProperties({"myModel": my_model})
```

9. Load the view.qml to the QQuickView and call show() to display the application window.

```
import sys
import urllib.request
import json
from pathlib import Path

from PySide6.QtQuick import QQuickView
from PySide6.QtCore import QStringListModel, QUrl
from PySide6.QtGui import QtGuiApplication

if __name__ == '__main__':

    #get our data
    url = "http://country.io/names.json"
    response = urllib.request.urlopen(url)
    data = json.loads(response.read().decode('utf-8'))

    #Format and sort the data
    data_list = list(data.values())
    data_list.sort()

    #Set up the application window
    app = QtGuiApplication(sys.argv)
    view = QQuickView()
    view.setResizeMode(QQuickView.SizeRootObjectToView)

    #Expose the list to the Qml code
    my_model = QStringListModel()
    my_model.setStringList(data_list)
    view.setInitialProperties({"myModel": my_model})

    #Load the QML file
    qml_file = Path(__file__).parent / "view.qml"
    view.setSource(QUrl.fromLocalFile(qml_file.resolve()))

    #Show the window
    if view.status() == QQuickView.Error:
        sys.exit(-1)
    view.show()
```

10. Finally, execute the application to start the event loop and clean up.

```
import sys
import urllib.request
```

```

import json
from pathlib import Path

from PySide6.QtQuick import QQuickView
from PySide6.QtCore import QStringListModel, QUrl
from PySide6.QtGui import QApplication

if __name__ == '__main__':

    #get our data
    url = "http://country.io/names.json"
    response = urllib.request.urlopen(url)
    data = json.loads(response.read().decode('utf-8'))

    #Format and sort the data
    data_list = list(data.values())
    data_list.sort()

    #Set up the application window
    app = QApplication(sys.argv)
    view = QQuickView()
    view.setResizeMode(QQuickView.SizeRootObjectToView)

    #Expose the list to the Qml code
    my_model = QStringListModel()
    my_model.setStringList(data_list)
    view.setInitialProperties({"myModel": my_model})

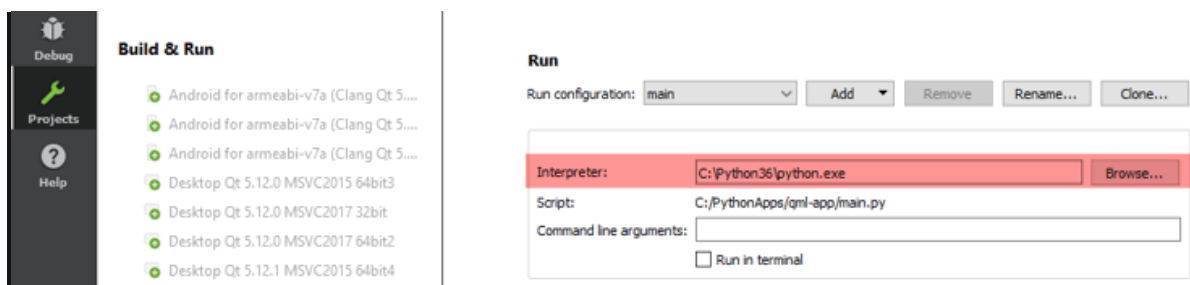
    #Load the QML file
    qml_file = Path(__file__).parent / "view.qml"
    view.setSource(QUrl.fromLocalFile(qml_file.resolve()))

    #Show the window
    if view.status() == QQuickView.Error:
        sys.exit(-1)
    view.show()

    #execute and cleanup
    app.exec()
    del view

```

11. Your application is ready to be run now. Select **Projects** mode to choose the Python version to run it.



Run the application by using the CTRL+R keyboard shortcut to see if it looks like this:



QML, SQL and PySide Integration Tutorial

This tutorial is very similar to the [Qt Chat Tutorial](#) one but it focuses on explaining how to integrate a SQL database into a PySide6 application using QML for its UI.

sqlDialog.py

We import the pertinent libraries to our program, define a global variable that hold the name of our table, and define the global function createTable() that creates a new table if it doesn't already exist. The database contains a single line to mock the beginning of a conversation.

```
import datetime
import logging

from PySide6.QtCore import Qt, Slot
from PySide6.QtSql import QSqlDatabase, QSqlQuery, QSqlRecord, QSqlTableModel
from PySide6.QtQml import QmlElement

table_name = "Conversations"
QML_IMPORT_NAME = "ChatModel"
QML_IMPORT_MAJOR_VERSION = 1
```

```

def createTable():
    if table_name in QSqlDatabase.database().tables():
        return

    query = QSqlQuery()
    if not query.exec_(
        """
        CREATE TABLE IF NOT EXISTS 'Conversations' (
            'author' TEXT NOT NULL,
            'recipient' TEXT NOT NULL,
            'timestamp' TEXT NOT NULL,
            'message' TEXT NOT NULL,
            FOREIGN KEY('author') REFERENCES Contacts ( name ),
            FOREIGN KEY('recipient') REFERENCES Contacts ( name )
        )
        """
    ):
        logging.error("Failed to query database")

    # This adds the first message from the Bot
    # and further development is required to make it interactive.
    query.exec_(
        """
        INSERT INTO Conversations VALUES(
            'machine', 'Me', '2019-01-07T14:36:06', 'Hello!'
        )
        """
    )

```

The `SqlConversationModel` class offers the read-only data model required for the non-editable contacts list. It derives from the `QSqlQueryModel` class, which is the logical choice for this use case. Then, we proceed to create the table, set its name to the one defined previously with the `setTable()` method. We add the necessary attributes to the table, to have a program that reflects the idea of a chat application.

```

@QmlElement
class SqlConversationModel(QSqlTableModel):
    def __init__(self, parent=None):
        super(SqlConversationModel, self).__init__(parent)

        createTable()
        self.setTable(table_name)
        self.setSort(2, Qt.DescendingOrder)
        self.setEditStrategy(QSqlTableModel.OnManualSubmit)
        self.recipient = ""

        self.select()
        logging.debug("Table was loaded successfully.")

```

In `setRecipient()`, you set a filter over the returned results from the database, and emit a signal every time the recipient of the message changes.

```

def setRecipient(self, recipient):
    if recipient == self.recipient:
        pass

    self.recipient = recipient

```

```

filter_str = (f"(recipient = '{self.recipient}' AND author = 'Me') OR "
              f"(recipient = 'Me' AND author='{self.recipient}')"
self.setFilter(filter_str)
self.select()

```

The `data()` function falls back to `QSqlTableModel`'s implementation if the role is not a custom user role. If you get a user role, we can subtract `UserRole()` from it to get the index of that field, and then use that index to find the value to be returned.

```

def data(self, index, role):
    if role < Qt.UserRole:
        return QSqlTableModel.data(self, index, role)

    sql_record = QSqlRecord()
    sql_record = self.record(index.row())

    return sql_record.value(role - Qt.UserRole)

```

In `roleNames()`, we return a Python dictionary with our custom role and role names as key-values pairs, so we can use these roles in QML. Alternatively, it can be useful to declare an Enum to hold all of the role values. Note that names has to be a hash to be used as a dictionary key, and that's why we're using the hash function.

```

def roleNames(self):
    """Converts dict to hash because that's the result expected
    by QSqlTableModel"""
    names = {}
    author = "author".encode()
    recipient = "recipient".encode()
    timestamp = "timestamp".encode()
    message = "message".encode()

    names[hash(Qt.UserRole)] = author
    names[hash(Qt.UserRole + 1)] = recipient
    names[hash(Qt.UserRole + 2)] = timestamp
    names[hash(Qt.UserRole + 3)] = message

    return names

```

The `send_message()` function uses the given recipient and message to insert a new record into the database. Using `OnManualSubmit()` requires you to also call `submitAll()`, since all the changes will be cached in the model until you do so.

```

# This is a workaround because PySide doesn't provide Q_INVOKABLE
# So we declare this as a Slot to be able to call it from QML
@Slot(str, str, str)
def send_message(self, recipient, message, author):
    timestamp = datetime.datetime.now()

    new_record = self.record()
    new_record.setValue("author", author)
    new_record.setValue("recipient", recipient)
    new_record.setValue("timestamp", str(timestamp))
    new_record.setValue("message", message)

```

```
logging.debug(f'Message: "{message}" \n Received by: "{recipient}"')

if not self.insertRecord(self.rowCount(), new_record):
    logging.error("Failed to send message: {self.lastError().text()}")
    return

self.submitAll()
self.select()
```

chat.qml

Let's look at the chat.qml file.

```
import QtQuick
import QtQuick.Layouts
import QtQuick.Controls
```

First, import the Qt Quick module. This gives us access to graphical primitives such as Item, Rectangle, Text, and so on. For a full list of types, see the [Qt Quick QML Types](#) documentation. We then add QtQuick.Layouts import, which we'll cover shortly.

Next, import the Qt Quick Controls module. Among other things, this provides access to ApplicationWindow, which replaces the existing root type, Window:

Let's step through the chat.qml file.

```
ApplicationWindow {
    id: window
    title: qsTr("Chat")
    width: 640
    height: 960
    visible: true
```

ApplicationWindow is a Window with some added convenience for creating a header and a footer. It also provides the foundation for popups and supports some basic styling, such as the background color.

There are three properties that are almost always set when using ApplicationWindow: width, height, and visible. Once we've set these, we have a properly sized, empty window ready to be filled with content.

Because we are exposing the SqlConversationModel class to QML, we will declare a component to access it:

```
SqlConversationModel {
    id: chat_model
}
```

There are two ways of laying out items in QML: [Item Positioners](#) and [Qt Quick Layouts](#).

- Item positioners ([Row](#), [Column](#), and so on) are useful for situations where the size of items is known or fixed, and all that is required is to neatly position them in a certain formation.
- The layouts in Qt Quick Layouts can both position and resize items, making them well suited for resizable user interfaces. Below, we use [ColumnLayout](#) to vertically lay out a [ListView](#) and a [Pane](#).

```
ColumnLayout {
    anchors.fill: window

    ListView {

        Pane {
            id: pane
            Layout.fillWidth: true
        }
    }
}
```

Pane is basically a rectangle whose color comes from the application's style. It's similar to [Frame](#), but it has no stroke around its border.

Items that are direct children of a layout have various [attached properties](#) available to them. We use [Layout.fillWidth](#) and [Layout.fillHeight](#) on the [ListView](#) to ensure that it takes as much space within the [ColumnLayout](#) as it can, and the same is done for the Pane. As [ColumnLayout](#) is a vertical layout, there aren't any items to the left or right of each child, so this results in each item consuming the entire width of the layout.

On the other hand, the [Layout.fillHeight](#) statement in the [ListView](#) enables it to occupy the remaining space that is left after accommodating the Pane.

Let's look at the Listview in detail:

```
ListView {
    id: listView
    Layout.fillWidth: true
    Layout.fillHeight: true
    Layout.margins: pane.leftPadding + messageField.leftPadding
    displayMarginBeginning: 40
    displayMarginEnd: 40
    verticalLayoutDirection: ListView.BottomToTop
    spacing: 12
    model: chat_model
    delegate: Column {
        anchors.right: sentByMe ? listView.contentItem.right : undefined
        spacing: 6

        readonly property bool sentByMe: model.recipient !== "Me"
        Row {
            id: messageRow
            spacing: 6
            anchors.right: sentByMe ? parent.right : undefined

            Rectangle {
                width: Math.min(messageText.implicitWidth + 24,
                    listView.width - (!sentByMe ? messageRow.spacing : 0))
                height: messageText.implicitHeight + 24
                radius: 15
            }
        }
    }
}
```

```

        color: sentByMe ? "lightgrey" : "steelblue"

        Label {
            id: messageText
            text: model.message
            color: sentByMe ? "black" : "white"
            anchors.fill: parent
            anchors.margins: 12
            wrapMode: Label.Wrap
        }
    }
}

Label {
    id: timestampText
    text: Qt.formatDateTime(model.timestamp, "d MMM hh:mm")
    color: "lightgrey"
    anchors.right: sentByMe ? parent.right : undefined
}

ScrollBar.vertical: ScrollBar {}
}

```

After filling the width and height of its parent, we also set some margins on the view.

Next, we set [displayMarginBeginning](#) and [displayMarginEnd](#). These properties ensure that the delegates outside the view don't disappear when you scroll at the edges of the view. To get a better understanding, consider commenting out the properties and then rerun your code. Now watch what happens when you scroll the view.

We then flip the vertical direction of the view, so that first items are at the bottom.

Additionally, messages sent by the contact should be distinguished from those sent by a contact. For now, when a message is sent by you, we set a `sentByMe` property, to alternate between different contacts. Using this property, we distinguish between different contacts in two ways:

- Messages sent by the contact are aligned to the right side of the screen by setting `anchors.right` to `parent.right`.
- We change the color of the rectangle depending on the contact. Since we don't want to display dark text on a dark background, and vice versa, we also set the text color depending on who the contact is.

At the bottom of the screen, we place a [TextArea](#) item to allow multi-line text input, and a button to send the message. We use `Pane` to cover the area under these two items:

```

Pane {
    id: pane
    Layout.fillWidth: true

    RowLayout {
        width: parent.width

        TextArea {

```

```

        id: messageField
        Layout.fillWidth: true
        placeholderText: qsTr("Compose message")
        wrapMode: TextArea.Wrap
    }

    Button {
        id: sendButton
        text: qsTr("Send")
        enabled: messageField.length > 0
        onClicked: {
            listView.model.send_message("machine", messageField.text, "Me");
            messageField.text = "";
        }
    }
}
}
}

```

The [TextArea](#) should fill the available width of the screen. We assign some placeholder text to provide a visual cue to the contact as to where they should begin typing. The text within the input area is wrapped to ensure that it does not go outside of the screen.

Lastly, we have a button that allows us to call the `send_message` method we defined on `sqlDialog.py`, since we're just having a mock up example here and there is only one possible recipient and one possible sender for this conversation we're just using strings here.

main.py

We use logging instead of Python's `print()`, because it provides a better way to control the messages levels that our application will generate (errors, warnings, and information messages).

```

import sys
import logging

from PySide6.QtCore import QDir, QFile, QUrl
from PySide6.QtGui import QApplication
from PySide6.QtQml import QQmlApplicationEngine
from PySide6.QtSql import QSqlDatabase

# We import the file just to trigger the QmlElement type registration.
import sqlDialog

logging.basicConfig(filename="chat.log", level=logging.DEBUG)
logger = logging.getLogger("logger")

```

`connectToDatabase()` creates a connection with the SQLite database, creating the actual file if it doesn't already exist.

```

def connectToDatabase():
    database = QSqlDatabase.database()
    if not database.isValid():
        database = QSqlDatabase.addDatabase("QSQLITE")
    if not database.isValid():

```

```

        logger.error("Cannot add database")

write_dir = QDir("")
if not write_dir.mkpath("."):
    logger.error("Failed to create writable directory")

# Ensure that we have a writable location on all devices.
abs_path = write_dir.absolutePath()
filename = f"{abs_path}/chat-database.sqlite3"

# When using the SQLite driver, open() will create the SQLite
# database if it doesn't exist.
database.setDatabaseName(filename)
if not database.open():
    logger.error("Cannot open database")
    QFile.remove(filename)

```

A few interesting things happen in the main function:

- Declaring a `QGuiApplication`. You should use a `QGuiApplication` instead of `QApplication` because we're not using the **QtWidgets** module.
- Connecting to the database,
- Declaring a `QQmlApplicationEngine`. This allows you to access the QML Elements to connect Python and QML from the conversation model we built on `sqlDialog.py`.
- Loading the `.qml` file that defines the UI.

Finally, the Qt application runs, and your program starts.

```

if __name__ == "__main__":
    app = QGuiApplication()
    connectToDatabase()

    engine = QQmlApplicationEngine()
    engine.load(QUrl("chat.qml"))

    if not engine.rootObjects():
        sys.exit(-1)

app.exec()

```

Hello!

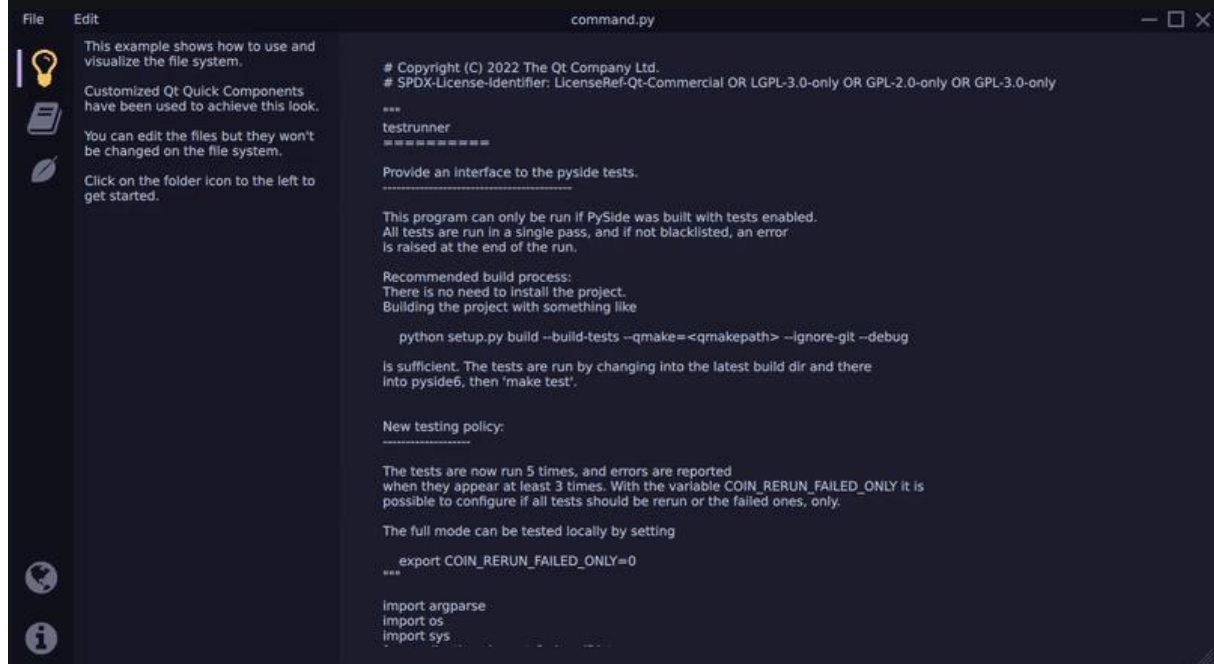
7 Jan 14:36

Howdy! :)

26 Nov 09:55

Extending the file system explorer example

This tutorial shows how to extend the [Filesystem Explorer Example](#) by adding a simple scheme manager. This feature will allow you to switch color schemes during the application's runtime. The color schemes will be declared in JSON format and made available through a custom Python-QML plugin.



Defining the color schemes

To define your color scheme, you can use the same color names as the original example, so you don't have to rename every occurrence. The original colors are defined in the Colors.qml file as follows:

resources/Colors.qml

```
QtObject {
    readonly property color background: "#23272E"
    readonly property color surface1: "#1E2227"
    readonly property color surface2: "#090A0C"
    readonly property color text: "#ABB2BF"
    readonly property color textFile: "#C5CAD3"
    readonly property color disabledText: "#454D5F"
    readonly property color selection: "#2C313A"
    readonly property color active: "#23272E"
    readonly property color inactive: "#3E4452"
    readonly property color folder: "#3D4451"
    readonly property color icon: "#3D4451"
    readonly property color iconIndicator: "#E5C07B"
    readonly property color color1: "#E06B74"
```

```
}
    readonly property color color2: "#62AEEF"
}
```

The schemes.json file holds the color schemes. To start implementing this, you can use the [Catppuccin](#) scheme.

schemes.json

```
"Catppuccin": {
  "background": "#1E1E2E",
  "surface1": "#181825",
  "surface2": "#11111B",
  "text": "#CDD6F4",
  "textFile": "#CDD6F4",
  "disabledText": "#363659",
  "selection": "#45475A",
  "active": "#1E1E2E",
  "inactive": "#6C7086",
  "folder": "#6C7086",
  "icon": "#6C7086",
  "iconIndicator": "#FFCC66",
  "color1": "#CBA6F7",
  "color2": "#89DCEB"
},
```

In addition to the “Catppuccin” color scheme, four other color schemes got implemented: Nordic, One Dark, Gruvbox, and Solarized. However, feel free to get creative and experiment with **your** schemes.

To define a new color scheme, copy the structure from above and provide your color values

Implement the scheme manager

After defining the color schemes, you can implement the actual scheme manager. The manager will read the schemes.json file and provide QML bindings to switch between schemes during runtime.

To implement the scheme manager, create a Python-QML plugin that exposes the SchemeManager object to QML. This object will have methods to load the color schemes from the schemes.json file and switch between them.

Create a new Python file called schememanager.py in your project directory. In this file, define the **SchemeManager** class:

scheme_manager.py

```
QML_IMPORT_NAME = "FileSystemModule"
QML_IMPORT_MAJOR_VERSION = 1

@QmlNamedElement("Colors")
@QmlSingleton
class SchemeManager(QObject):
```

To integrate smoothly into the already existing code, attach the SchemeManager to the same QML module that's already present with `QML_IMPORT_NAME = "FileSystemModule"`. Additionally, use the `@QmlNamedElement` decorator to smoothly transition to using the custom plugin instead of the `Colors.qml` file. With these changes, we can avoid editing all previous assignments like:

```
import FileSystemModule
...
Rectangle {
    color: Colors.background
}
```

The constructor reads the `schemes.json` file once upon application start and then calls the `setTheme` member function.

scheme_manager.py

```
schemeChanged = Signal()

def __init__(self, parent=None):
    super().__init__(parent=parent)
    with open(Path(__file__).parent / "schemes.json", 'r') as f:
        self.m_schemes = json.load(f)
    self.m_activeScheme = {}
```

By adding the SchemeManager as a callable QML element named **Colors** to the `FileSystemModule`, the class is now accessible in the code without the need to import it each time or edit previous assignments. This, in turn, will streamline the workflow.

After defining the schemes in the JSON format and making the SchemeManager class a callable element from QML under the name **Colors**, there are two remaining steps to fully integrate the new scheme manager in the example.

The **first step** is to create a function in the SchemeManager class that loads a color scheme from the JSON file. The **second step** is to make the individual colors available in QML with the *same name* as used before with the syntax `Colors.<previousName>` as assignable properties.

scheme_manager.py

```
self.setScheme(self.m_activeSchemeName)

@Slot(str)
def setScheme(self, theme):
    for k, v in self.m_schemes[theme].items():
        self.m_activeScheme[k] = QColor.fromString(v)
```

The `setScheme` method is responsible for switching between color schemes. To make this method accessible in QML, use the `@Slot(str)` decorator and specify that it takes a string as its input parameter. In this method, we populate a dictionary with the color values from the JSON file.

Note: For simplicity reasons no other error checking is performed. You would probably want to validate the keys contained in the json.

scheme_manager.py

```
@Property(QColor, notify=schemeChanged)
def background(self):
    return self.m_activeScheme["background"]
```

To make the color property assignable in QML, use the `@Property` decorator. We simply return the corresponding color value from the dictionary for each property. This process is repeated for all other colors that are used in the application. At this point the application should start with the colors provided by the active scheme in the constructor.

Add the scheme switching to QML

To visualize the current scheme and enable interactive scheme switching, start by adding a new entry to the `Sidebar.qml` file.

FileSystemModule/qml/Sidebar.qml

```
// Shows the scheme switcher
SidebarEntry {
    icon.source: "../icons/leaf.svg"
    checkable: true

    Layout.alignment: Qt.AlignHCenter
}
```

To update the main content area of the application to display the `ColorScheme`, the logic that checks the active index from the Sidebar buttons needs to be modified. The necessary changes will be made to the `Main.qml` file:

FileSystemModule/Main.qml

```
// The main view that contains the editor or the scheme-manager.
StackLayout {
    currentIndex: sidebar.currentTabIndex > 1 ? 1 : 0

    SplitView.fillWidth: true
    SplitView.fillHeight: true

    Editor {
        id: editor
        showLineNumbers: root.showLineNumbers
        currentFilePath: root.currentFilePath
    }

    ColorScheme {
        Layout.fillWidth: true
        Layout.fillHeight: true
    }
}
```

In addition, change the behavior of the application so that there are two `StackLayouts`: one for the resizable navigation and one for the main content area where we display our color scheme switching functionality. These changes will also be made to the `Main.qml` file.

FileSystemModule/Main.qml

```
// selected buttons inside the sidebar.
StackLayout {
```



```
anchors.fill: parent
currentIndex: sidebar.currentIndex > 1 ? 1 : sidebar.currentIndex
```

To complete our implementation, the ColorScheme.qml file needs to be created. The implementation is straightforward and follows the same principles as in the original example. If anything is unclear, please refer to the documentation provided there. To display all colors and scheme names, use a Repeater. The model for the Repeater is provided by our scheme_manager.pyfile as a QStringList.

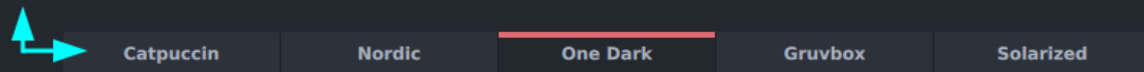
FileSystemModule/qml/ColorScheme.qml

```
// Display all used colors inside a row
Row {
    anchors.centerIn: parent
    spacing: 10

    Repeater {
        model: Colors.currentColors
        Rectangle {
            width: 35
            height: width
            radius: width / 2
            color: modelData
        }
    }
}
```

When examining the code in more detail, you will notice that there are different ways to retrieve the models. The `getKeys()` method is defined as a **Slot** and therefore requires parentheses when called. On the other hand, the `currentColors` model is defined as a **property** and is therefore assigned as a property in QML. The reason for this is to receive notifications when the color scheme is switched so that the colors displayed in the application can be updated. The keys for the color schemes are loaded only once at application startup and do not rely on any notifications.

model: Colors.getKeys() **Select a Theme!**



model: Colors.currentColors

Expenses Tool Tutorial

In this tutorial you will learn the following concepts:

- creating user interfaces programatically,
- layouts and widgets,
- overloading Qt classes,
- connecting signal and slots,
- interacting with QWidgets,
- and building your own application.

The requirements:

- A simple window for the application ([QMainWindow](#)).
- A table to keep track of the expenses ([QTableWidget](#)).
- Two input fields to add expense information ([QLineEdit](#)).
- Buttons to add information to the table, plot data, clear table, and exit the application ([QPushButton](#)).
- A verification step to avoid invalid data entry.
- A chart to visualize the expense data ([QChart](#)) that will be embedded in a chart view ([QChartView](#)).

Empty window

The base structure for a *QApplication* is located inside the `if __name__ == "__main__":` code block.

```
if __name__ == "__main__":
    app = QApplication([])
    # ...
    sys.exit(app.exec())
```

Now, to start the development, create an empty window called *MainWindow*. You could do that by defining a class that inherits from *QMainWindow*.

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Tutorial")

if __name__ == "__main__":
    # Qt Application
    app = QApplication(sys.argv)

    window = MainWindow()
    window.resize(800, 600)
    window.show()

    # Execute application
    sys.exit(app.exec())
```

Now that our class is defined, create an instance of it and call *show()*.

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Tutorial")

if __name__ == "__main__":
    # Qt Application
    app = QApplication(sys.argv)

    window = MainWindow()
    window.resize(800, 600)
    window.show()

    # Execute application
    sys.exit(app.exec())
```

Menu bar

Using a *QMainWindow* gives some features for free, among them a *menu bar*. To use it, you need to call the method *menuBar()* and populate it inside the *MainWindow* class.

```
def __init__(self):
    super().__init__()
    self.setWindowTitle("Tutorial")

    # Menu
    self.menu = self.menuBar()
    self.file_menu = self.menu.addMenu("File")

    # Exit QAction
    exit_action = self.file_menu.addAction("Exit", self.close)
    exit_action.setShortcut("Ctrl+Q")
```

Notice that the code snippet adds a *File* menu with the *Exit* option only.

The *Exit* option must be connected to a slot that triggers the application to exit. We pass *QWidget.close()* here. After the last window has been closed, the application exits.

Empty widget and data

The *QMainWindow* enables us to set a central widget that will be displayed when showing the window ([read more](#)). This central widget could be another class derived from *QWidget*.

Additionally, you will define example data to visualize later.

```
class Widget(QWidget):
    def __init__(self):
        super().__init__()

    # Example data
    self_data = {"Water": 24.5, "Electricity": 55.1, "Rent": 850.0,
```

```
"Supermarket": 230.4, "Internet": 29.99, "Bars": 21.85,  
"Public transportation": 60.0, "Coffee": 22.45, "Restaurants": 120}
```

With the *Widget* class in place, modify *MainWindow*'s initialization code

```
# QWidget  
widget = Widget()  
# QMainWindow using QWidget as central widget  
window = MainWindow(widget)
```

Window layout

Now that the main empty window is in place, you need to start adding widgets to achieve the main goal of creating an expenses application.

After declaring the example data, you can visualize it on a simple *QTableWidget*. To do so, you will add this procedure to the *Widget* constructor.

Warning

Only for the example purpose a *QTableWidget* will be used, but for more performance-critical applications the combination of a model and a *QTableView* is encouraged.

```
def __init__(self):  
    super().__init__()  
    self.items = 0  
  
    # Example data  
    self_data = {"Water": 24.5, "Electricity": 55.1, "Rent": 850.0,  
                 "Supermarket": 230.4, "Internet": 29.99, "Bars": 21.85,  
                 "Public transportation": 60.0, "Coffee": 22.45, "Restaurants": 120}  
  
    # Left  
    self.table = QTableWidget()  
    self.table.setColumnCount(2)  
    self.table.setHorizontalHeaderLabels(["Description", "Price"])  
    self.table.horizontalHeader().setSectionResizeMode(QHeaderView.Stretch)  
  
    # QWidget Layout  
    self.layout = QHBoxLayout(self)  
    self.layout.addWidget(self.table)  
  
    # Fill example data  
    self.fill_table()
```

As you can see, the code also includes a *QHBoxLayout* that provides the container to place widgets horizontally.

Additionally, the *QTableWidget* allows for customizing it, like adding the labels for the two columns that will be used, and to *stretch* the content to use the whole *Widget* space.

The last line of code refers to *filling the table**, and the code to perform that task is displayed below.

```
def fill_table(self, data=None):
    data = self._data if not data else data
    for desc, price in data.items():
        self.table.insertRow(self.items)
        self.table.setItem(self.items, 0, QTableWidgetItem(desc))
        self.table.setItem(self.items, 1, QTableWidgetItem(str(price)))
    self.items += 1
```

Having this process on a separate method is a good practice to leave the constructor more readable, and to split the main functions of the class in independent processes.

Right side layout

Because the data that is being used is just an example, you are required to include a mechanism to input items to the table, and extra buttons to clear the table's content, and also quit the application.

For input lines along with descriptive labels, you will use a *QFormLayout*. Then, you will nest the form layout into a *QVBoxLayout* along with the buttons.

```
# Right
self.description = QLineEdit()
self.description.setClearButtonEnabled(True)
self.price = QLineEdit()
self.price.setClearButtonEnabled(True)

self.add = QPushButton("Add")
self.clear = QPushButton("Clear")

form_layout = QFormLayout()
form_layout.addRow("Description", self.description)
form_layout.addRow("Price", self.price)
self.right = QVBoxLayout()
self.right.addLayout(form_layout)
self.right.addWidget(self.add)
self.right.addStretch()
self.right.addWidget(self.clear)
```

Leaving the table on the left side and these newly included widgets to the right side will be just a matter to add a layout to our main *QHBoxLayout* as you saw in the previous example:

```
# QWidget Layout
self.layout = QHBoxLayout(self)
self.layout.addWidget(self.table)
self.layout.addLayout(self.right)
```

The next step will be connecting those new buttons to slots.

Adding elements

Each *QPushButton* have a signal called *clicked*, that is emitted when you click on the button. This will be more than enough for this example, but you can see other signals in the [official documentation](#).

```
# Signals and Slots
self.add.clicked.connect(self.add_element)
self.clear.clicked.connect(self.clear_table)
```

As you can see on the previous lines, we are connecting each *clicked* signal to different slots. In this example slots are normal class methods in charge of perform a determined task associated with our buttons. It is really important to decorate each method declaration with a *@Slot()*, that way, PySide6 knows internally how to register them into Qt and they will be invocable from *Signals* of QObjects when connected.

```
@Slot()
def add_element(self):
    des = self.description.text()
    price = self.price.text()

    self.table.insertRow(self.items)
    self.table.setItem(self.items, 0, QTableWidgetItem(des))
    self.table.setItem(self.items, 1, QTableWidgetItem(price))

    self.description.clear()
    self.price.clear()

    self.items += 1

def fill_table(self, data=None):
    data = self._data if not data else data
    for desc, price in data.items():
        self.table.insertRow(self.items)
        self.table.setItem(self.items, 0, QTableWidgetItem(desc))
        self.table.setItem(self.items, 1, QTableWidgetItem(str(price)))
        self.items += 1

@Slot()
def clear_table(self):
    self.table.setRowCount(0)
    self.items = 0
```

Since these slots are methods, we can access the class variables, like our *QTableWidget* to interact with it.

The mechanism to add elements into the table is described as the following:

- get the *description* and *price* from the fields,
- insert a new empty row to the table,
- set the values for the empty row in each column,
- clear the input text fields,
- include the global count of table rows.

To exit the application you can use the `quit()` method of the unique *QApplication* instance, and to clear the content of the table you can just set the table *row count*, and the internal count to zero.

Verification step

Adding information to the table needs to be a critical action that require a verification step to avoid adding invalid information, for example, empty information.

You can use a signal from *QLineEdit* called *textChanged* which will be emitted every time something inside changes, i.e.: each key stroke.

You can connect two different object's signal to the same slot, and this will be the case for your current application:

```
self.description.textChanged.connect(self.check_disable)
self.price.textChanged.connect(self.check_disable)
```

The content of the *check_disable* slot will be really simple:

```
@Slot()
def check_disable(self, s):
    enabled = bool(self.description.text() and self.price.text())
    self.add.setEnabled(enabled)
```

You have two options, write a verification based on the current value of the string you retrieve, or manually get the whole content of both *QLineEdit*. The second is preferred in this case, so you can verify if the two inputs are not empty to enable the button *Add*.

Note

Qt also provides a special class called [QValidator](#) that you can use to validate any input.

Empty chart view

New items can be added to the table, and the visualization is so far OK, but you can accomplish more by representing the data graphically.

First you will include an empty *QChartView* placeholder into the right side of your application.

```
# Chart
self.chart_view = QChartView()
self.chart_view.setRenderHint(QPainter.Antialiasing)
```

Additionally the order of how you include widgets to the right *QVBoxLayout* will also change.

```
form_layout = QFormLayout()
```

```

form_layout.addRow("Description", self.description)
form_layout.addRow("Price", self.price)
self.right = QVBoxLayout()
self.right.addLayout(form_layout)
self.right.addWidget(self.add)
self.right.addWidget(self.plot)
self.right.addWidget(self.chart_view)
self.right.addWidget(self.clear)

```

Notice that before we had a line with *self.right.addStretch()* to fill up the vertical space between the *Add* and the *Clear* buttons, but now, with the *QChartView* it will not be necessary.

Also, you need include a *Plot* button if you want to do it on-demand.

Full application

For the final step, you will need to connect the *Plot* button to a slot that creates a chart and includes it into your *QChartView*.

```

# Signals and Slots
self.add.clicked.connect(self.add_element)
self.plot.clicked.connect(self.plot_data)
self.clear.clicked.connect(self.clear_table)
self.description.textChanged.connect(self.check_disable)
self.price.textChanged.connect(self.check_disable)

```

That is nothing new, since you already did it for the other buttons, but now take a look at how to create a chart and include it into your *QChartView*.

```

@Slot()
def plot_data(self):
    # Get table information
    series = QPieSeries()
    for i in range(self.table.rowCount()):
        text = self.table.item(i, 0).text()
        number = float(self.table.item(i, 1).text())
        series.append(text, number)

    chart = QChart()
    chart.addSeries(series)
    chart.legend().setAlignment(Qt.AlignLeft)
    self.chart_view.setChart(chart)

```

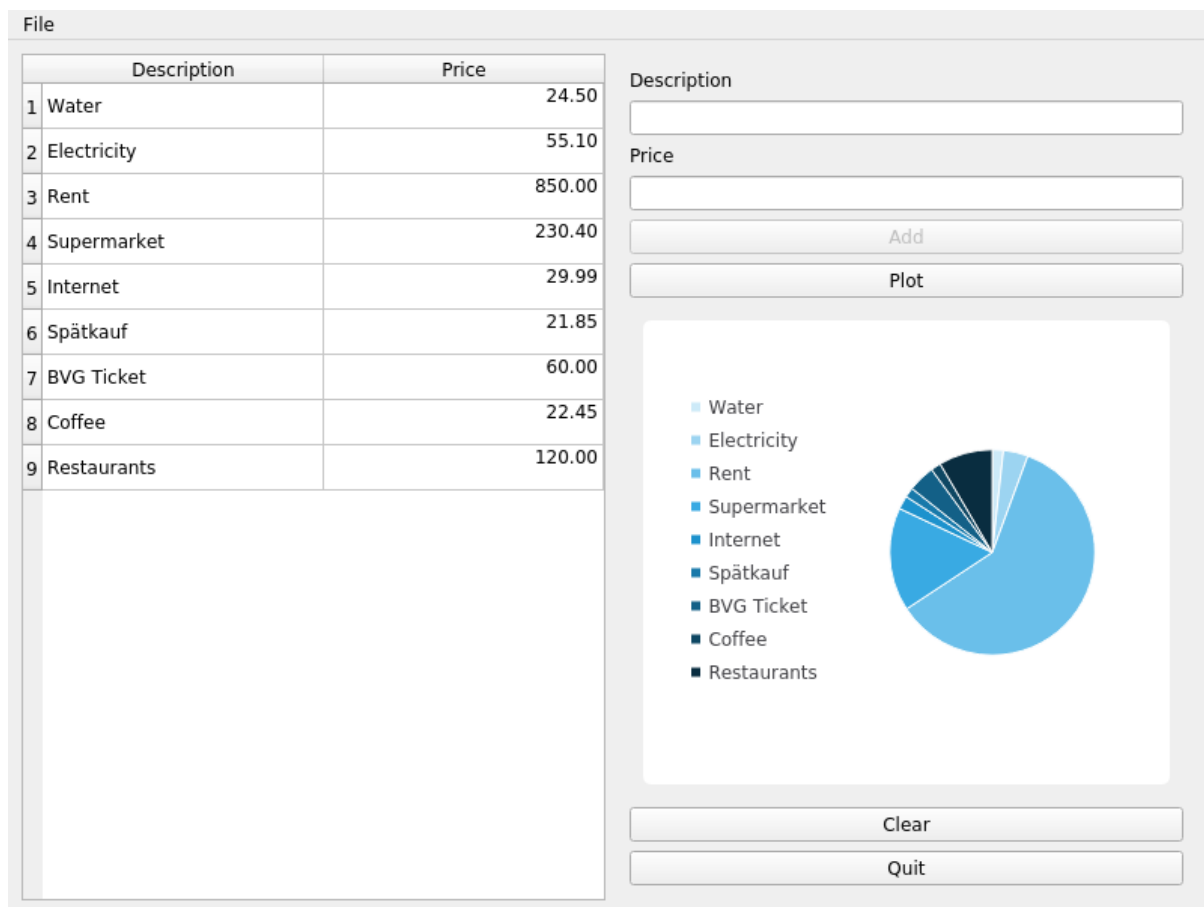
The following steps show how to fill a *QPieSeries*:

- create a *QPieSeries*,
- iterate over the table row IDs,
- get the items at the *i* position,
- add those values to the *series*.

Once the series has been populated with our data, you create a new *QChart*, add the series on it, and optionally set an alignment for the legend.

The final line `self.chart_view.setChart(chart)` is in charge of bringing your newly created chart to the `QChartView`.

The application will look like this:



And now you can see the whole code:

```
# Copyright (C) 2022 The Qt Company Ltd.
# SPDX-License-Identifier: LicenseRef-Qt-Commercial

import sys
from PySide6.QtCore import Qt, Slot
from PySide6.QtGui import QPainter
from PySide6.QtWidgets import (QApplication, QFormLayout, QHeaderView,
                               QHBoxLayout, QLineEdit, QMainWindow,
                               QPushButton, QTableWidget, QTableWidgetItem,
                               QVBoxLayout, QWidget)
from PySide6.QtCharts import QChartView, QPieSeries, QChart

class Widget(QWidget):
    def __init__(self):
        super().__init__()
        self.items = 0

        # Example data
        self._data = {"Water": 24.5, "Electricity": 55.1, "Rent": 850.0,
                     "Supermarket": 230.4, "Internet": 29.99, "Bars": 21.85,
```

```

        "Public transportation": 60.0, "Coffee": 22.45, "Restaurants": 120}

# Left
self.table = QTableWidgetItem()
self.table.setColumnCount(2)
self.table.setHorizontalHeaderLabels(["Description", "Price"])
self.table.horizontalHeader().setSectionResizeMode(QHeaderView.Stretch)

# Chart
self.chart_view = QChartView()
self.chart_view.setRenderHint(QPainter.Antialiasing)

# Right
self.description = QLineEdit()
self.description.setClearButtonEnabled(True)
self.price = QLineEdit()
self.price.setClearButtonEnabled(True)

self.add = QPushButton("Add")
self.clear = QPushButton("Clear")
self.plot = QPushButton("Plot")

# Disabling 'Add' button
self.add.setEnabled(False)

form_layout = QFormLayout()
form_layout.addRow("Description", self.description)
form_layout.addRow("Price", self.price)
self.right = QVBoxLayout()
self.right.addLayout(form_layout)
self.right.addWidget(self.add)
self.right.addWidget(self.plot)
self.right.addWidget(self.chart_view)
self.right.addWidget(self.clear)

# QWidget Layout
self.layout = QHBoxLayout(self)
self.layout.addWidget(self.table)
self.layout.addLayout(self.right)

# Signals and Slots
self.add.clicked.connect(self.add_element)
self.plot.clicked.connect(self.plot_data)
self.clear.clicked.connect(self.clear_table)
self.description.textChanged.connect(self.check_disable)
self.price.textChanged.connect(self.check_disable)

# Fill example data
self.fill_table()

@Slot()
def add_element(self):
    des = self.description.text()
    price = float(self.price.text())

    self.table.insertRow(self.items)
    description_item = QTableWidgetItem(des)
    price_item = QTableWidgetItem(f"{price:.2f}")
    price_item.setTextAlignment(Qt.AlignRight)

```

```

self.table.setItem(self.items, 0, description_item)
self.table.setItem(self.items, 1, price_item)

self.description.clear()
self.price.clear()

self.items += 1

@Slot()
def check_disable(self, s):
    enabled = bool(self.description.text() and self.price.text())
    self.add.setEnabled(enabled)

@Slot()
def plot_data(self):
    # Get table information
    series = QPieSeries()
    for i in range(self.table.rowCount()):
        text = self.table.item(i, 0).text()
        number = float(self.table.item(i, 1).text())
        series.append(text, number)

    chart = QChart()
    chart.addSeries(series)
    chart.legend().setAlignment(Qt.AlignLeft)
    self.chart_view.setChart(chart)

def fill_table(self, data=None):
    data = self._data if not data else data
    for desc, price in data.items():
        description_item = QTableWidgetItem(desc)
        price_item = QTableWidgetItem(f"{price:.2f}")
        price_item.setTextAlignment(Qt.AlignRight)
        self.table.insertRow(self.items)
        self.table.setItem(self.items, 0, description_item)
        self.table.setItem(self.items, 1, price_item)
        self.items += 1

@Slot()
def clear_table(self):
    self.table.setRowCount(0)
    self.items = 0

class MainWindow(QMainWindow):
    def __init__(self, widget):
        super().__init__()
        self.setWindowTitle("Tutorial")

        # Menu
        self.menu = self.menuBar()
        self.file_menu = self.menu.addMenu("File")

        # Exit QAction
        exit_action = self.file_menu.addAction("Exit", self.close)
        exit_action.setShortcut("Ctrl+Q")

        self.setCentralWidget(widget)

```

```
if __name__ == "__main__":  
    # Qt Application  
    app = QApplication(sys.argv)  
    # QWidget  
    widget = QWidget()  
    # QMainWindow using QWidget as central widget  
    window = QMainWindow(widget)  
    window.resize(800, 600)  
    window.show()  
  
    # Execute application  
    sys.exit(app.exec())
```

Porting a C++ Application to Python

Qt for Python lets you use Qt APIs in a Python application. So the next question is: What does it take to port an existing C++ application? Try porting a Qt C++ application to Python to understand this.

Before you start, ensure that all the prerequisites for Qt for Python are met. See [Getting Started](#) for more information. In addition, familiarize yourself with the basic differences between Qt in C++ and in Python.

Basic differences

This section highlights some of the basic differences between C++ and Python, and how Qt differs between these two contexts.

C++ vs Python

- In the interest of code reuse, both C++ and Python provide ways for one file of code to use facilities provided by another. In C++, this is done using the `#include` directive to access the API definition of the reused code. The Python equivalent is an `import` statement.
- The constructor of a C++ class shares the name of its class and automatically calls the constructor of any base-classes (in a predefined order) before it runs. In Python, the `__init__()` method is the constructor of the class, and it can explicitly call base-class constructors in any order.
- C++ uses the keyword, `this`, to implicitly refer to the current object. In python, you need to explicitly mention the current object as the first parameter to each instance method of the class; it is conventionally named `self`.
- And more importantly, forget about curly braces, `{ }`, and semi-colon, `;`.
- Precede variable definitions with the `global` keyword, only if they need global scope.

```

var = None
def func(key, value = None):
    """Does stuff with a key and an optional value.

    If value is omitted or None, the value from func()'s
    last call is reused.
    """
    global var
    if value is None:
        if var is None:
            raise ValueError("Must pass a value on first call", key, value)
        value = var
    else:
        var = value
    doStuff(key, value)

```

In this example, `func()` would treat `var` as a local name without the `global` statement. This would lead to a `NameError` in the `value is None` handling, on accessing `var`. For more information about this, see [Python reference documentation](#).

Tip

Python being an interpreted language, most often the easiest way is to try your idea in the interpreter. You could call the `help()` function in the interpreter on any built-in function or keyword in Python. For example, a call to `help(import)` should provide documentation about the `import` statement

Last but not the least, try out a few examples to familiarize yourself with the Python coding style and follow the guidelines outlined in the [PEP8 - Style Guide](#).

```

import sys

from PySide6.QtWidgets import QApplication, QLabel

app = QApplication(sys.argv)
label = QLabel("Hello World")
label.show()
sys.exit(app.exec())

```

Note

Qt provides classes that are meant to manage the application-specific requirements depending on whether the application is console-only (`QCoreApplication`), GUI with `QtWidgets` (`QApplication`), or GUI without `QtWidgets` (`QGuiApplication`). These classes load necessary plugins, such as the GUI libraries required by an application. In this case, it is `QApplication` that is initialized first as the application has a GUI with `QtWidgets`.

Qt in the C++ and Python context

Qt behaves the same irrespective of whether it is used in a C++ or a Python application. Considering that C++ and Python use different language semantics, some differences between the two variants of Qt are inevitable. Here are a few important ones that you must be aware of:

- **Qt Properties:** Q_PROPERTY macros are used in C++ to add a public member variable with getter and setter functions. Python's alternative for this is the @property decorator before the getter and setter function definitions.
- **Qt Signals and Slots:** Qt offers a unique callback mechanism, where a signal is emitted to notify the occurrence of an event, so that slots connected to this signal can react to it. In C++, the class definition must define the slots under the public Q_SLOTS: and signals under Q_SIGNALS: access specifier. You connect these two using one of the several variants of the QObject::connect() function. Python's equivalent for this is the @Slot` decorator just before the function definition. This is necessary to register the slots with the QtMetaObject.
- **QString, QVariant, and other types**
 - Qt for Python does not provide access to QString and QVariant. You must use Python's native types instead.
 - QChar and QStringRef are represented as Python strings, and QStringList is converted to a list of strings.
 - QDate, QDateTime, QTime, and QUrl's __hash__() methods return a string representation so that identical dates (and identical date/times or times or URLs) have identical hash values.
 - QTextStream's bin(), hex(), and oct() functions are renamed to bin_(), hex_(), and oct_() respectively. This should avoid name conflicts with Python's built-in functions.
- **QByteArray:** A QByteArray is treated as a list of bytes without encoding. Python 3 uses "bytes". QString is represented as an encoded human readable string, which means it is a "str".

Here is the improved version of the Hello World example, demonstrating some of these differences:

```
import sys
import random

from PySide6.QtWidgets import (QApplication, QLabel,
                                QPushButton, QVBoxLayout, QWidget)
from PySide6.QtCore import Qt, Slot

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

        self.hello = ["Hallo Welt", "Hei maailma", "Hola Mundo", "Привет мир"]

        self.button = QPushButton("Click me!")
        self.text = QLabel("Hello World")
        self.text.setAlignment(Qt.AlignCenter)

        self.layout = QVBoxLayout()
        self.layout.addWidget(self.text)
        self.layout.addWidget(self.button)
        self.setLayout(self.layout)

        self.button.clicked.connect(self.magic)

    @Slot()
```

```
def magic(self):
    self.text.setText(random.choice(self.hello))

if __name__ == "__main__":
    app = QApplication(sys.argv)

    widget = MyWidget()
    widget.resize(800, 600)
    widget.show()

    sys.exit(app.exec())
```

Note

The if block is just a good practice when developing a Python application. It lets the Python file behave differently depending on whether it is imported as a module in another file or run directly. The `__name__` variable will have different values in these two scenarios. It is `__main__` when the file is run directly, and the module's file name (`hello_world_ex` in this case) when imported as a module. In the later case, everything defined in the module except the if block is available to the importing file.

Notice that the `QPushButton`'s clicked signal is connected to the magic function to randomly change the `QLabel`'s text property. The `@Slot` decorator marks the methods that are slots and informs the `QtMetaObject` about them.

Porting a Qt C++ example

Qt offers several C++ examples to showcase its features and help beginners learn. You can try porting one of these C++ examples to Python. The [books SQL example](#) is a good starting point as it does not require you to write UI-specific code in Python, but can use its `.ui` file instead.