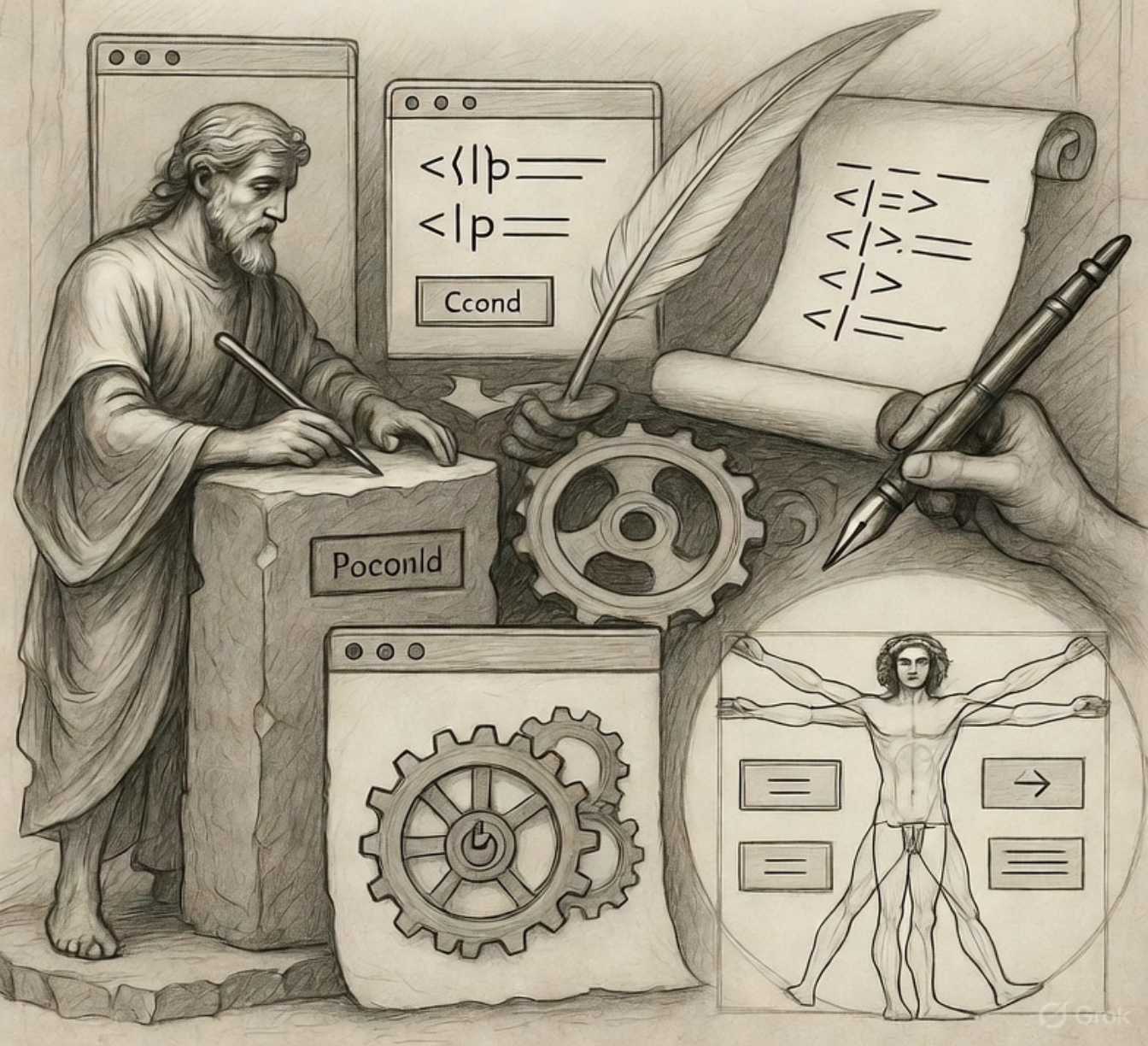
PySide6 Blueprints



PySide6 Blueprints

Step-by-Step Desktop Application Development with Python and Qt6

Uros Calakovic

This book is available at https://leanpub.com/pyside6blueprints

This version was published on 2025-12-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1. Getting Started

Qt is a cross-platform development framework used primarily for creating graphical user interfaces. It is written in C++, which is also its main development language; however, bindings exist for several other programming languages, including Python.

Qt provides two separate GUI toolkits: the traditional, desktop-oriented Qt Widgets, and the modern, declarative Qt Quick for building customized and fluid user interfaces. Qt applications run on Windows, Linux, macOS, iOS, Android, or embedded systems. It also supports deployment to the web via WebAssembly.

This book covers PySide6, Qt's official Python binding. It focuses on beginner-level Qt Widgets while tackling a few intermediate-to-advanced topics, including multithreading and model-view programming. Each chapter presents a series of self-contained PySide6 examples that showcase a single key concept or idea using step-by-step code walkthroughs.

1.1 Installation

You can easily start writing small PySide6 applications by

1. Creating a Python virtual environment from the terminal:

```
python -m venv pyside6-env
```

2. Activating the `pyside6-env` virtual environment:

```
pyside6-env\Scripts\activate
```

on Windows, or

```
source pyside6-env/bin/activate
```

on Linux or MacOS,

3. Installing the PySide6 Python package

```
pip install PySide6
```

Now, with the `pyside6-env` virtual environment active, you can run a PySide6 application from the terminal:

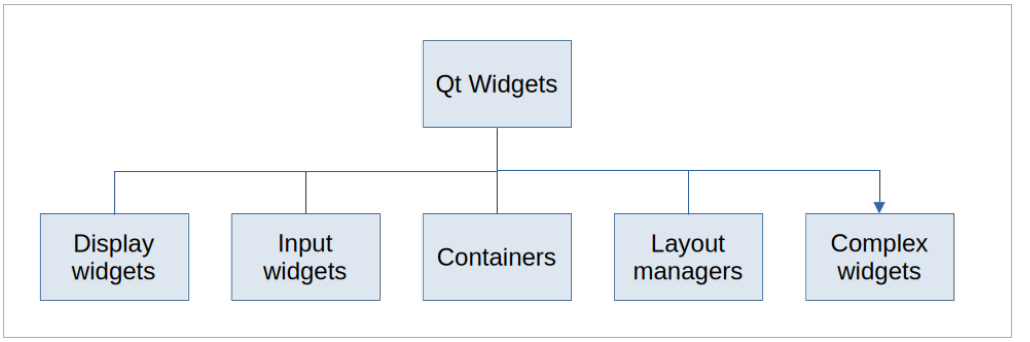
```
(pyside6-env) $ python helloworld.py
```

To improve your workflow, consider using an IDE or text editor to edit and run the examples in this book. Several of them, such as PyCharm, VS Code, or Qt Creator, let you create a virtual environment from their user interface.

1.2 Qt Widgets

Qt Widgets is the original, imperative Qt GUI toolkit, designed for building traditional desktop applications with native-looking windows, buttons, menus, tables, and dialogs. It relies on a hierarchy of objects that you compose programmatically or visually with Qt Designer, using layouts to manage their position and size, and signals & slots for event handling.

Qt Widgets fall into five main groups:



1. Display widgets. These widgets present information to the user - text, images, or visual indicators - without allowing direct editing,
2. Input widgets. Widgets that allow user interaction and data entry, covering data types including binary (on/off), numeric, textual, list-based, tabular, and hierarchical (tree) formats,
3. Containers. Visual organizers for grouping and managing child widgets,
4. Layout managers. Non-visual arrangement tools used to manage widget geometry (size and position),
5. Complex widgets. Self-contained, specialized interaction components composing display, input, and validation components, like dialogs and date/time widgets.

In the rest of this chapter we work through two "Hello, World" applications that you can use to test your PySide6 setup and get familiar with the basic structure of a PySide6 application.

1.3 Hello World

```
# A script that just displays
# an empty Qt6 window

import sys
from PySide6.QtWidgets import QApplication, QWidget


# 1 - Create a class inherited from QWidget

class Window(QWidget):

    def __init__(self):

        # If you don't init the superclass
        # you get a run-time error
        super().__init__()


if __name__ == '__main__':

    # 2 - Create an instance of the QApplication class
    #     Make sure there isn't one already running.

    if not QApplication.instance():
        app = QApplication(sys.argv)
```

```
26      else:
27          app = QApplication.instance()
28
29      # 3 - Create an instance of the Window class
30      #     and show() it
31
32      main_window = Window()
33      main_window.show()
34
35      # 4 - Start receiving events
36
37      sys.exit(app.exec())
```

The first example provides a reusable starting point for a PySide6 application, displaying nothing but an empty window while still showing the necessary PySide6 application boilerplate code.

Your first task: show an empty Qt window on the screen.

To to that:

1. Create a Python class that is a subclass of `QWidget`. `QWidget` is the base class of all Qt widget classes, often used as a top-level widget. For more involved applications that need toolbars, menu bars, etc., you can use `QMainWindow`, which provides them out of the box. In your class `__init__()` method, you must call `__init__()` on the superclass (i.e., `QWidget`) or you'll get a runtime error[1].

In the main script code

2. Create a `QApplication` instance. This should be the first thing you do in your application. For non-Qt-Widgets GUI applications, use `QGuiApplication`; for terminal/console applications, there is `QCoreApplication`. `QApplication` is a singleton, and if you try to create it more than once, you'll get

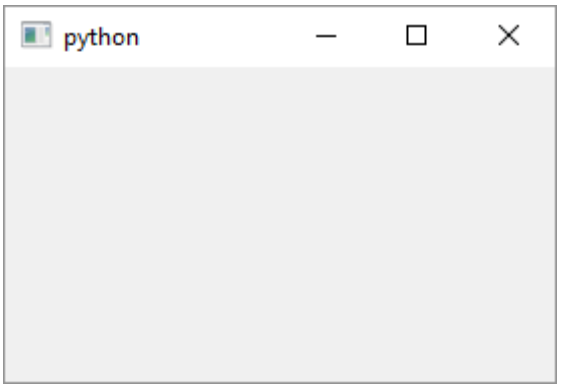
> RuntimeError: Please destroy the QApplication singleton before creating a new QApplication instance.

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

As all the examples in this book are quite simple, we won't be checking if the `QApplication` instance already exists—but it's good to be aware of this pattern for more complex applications.

3. Create an object of your class. We create a `Window` object and use the `QWidget.show()` method to show it on the screen.
4. Use `QApplication.exec()` to start the application event loop. With this, your application's main window starts processing events. When you exit the application, for instance by pressing the window's close button, the `app.exec()` method returns, and your application exits.

After you execute the code, you should see an empty Qt Widgets window like this:



Objects are instances, classes are types, widgets are controls.

In the book, we use the terms **object** and **instance** interchangeably. The Qt documentation mentions that QML **types** are analogous to **classes** and that QML **signal handlers** are **slots**. The Qt Quick equivalent to **widgets** are **controls**.

1.4 Hello World Again

Your task: using the first example as the template, add widgets to that window

```
# Displays a Hello, World! message in QLabel

# Import QLabel and QVBoxLayout

import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QLabel, QVBoxLayout)

# 1 - Create a class that inherits from QWidget

class Window(QWidget):

    def __init__(self):

        super().__init__()

        # 2 - Set window layout

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 3 - Create a QLabel instance
        #     and add it to the window layout

        label = QLabel('Hello, World!')
        layout.addWidget(label)

# The boilerplate code is similar to that from helloworld.py

if __name__ == '__main__':

    # The qApp variable points to the application
    # if it was already created so this is equivalent
    # to checking for QApplication.instance()
    app = qApp or QApplication(sys.argv)

    main_window = Window()
    main_window.show()

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

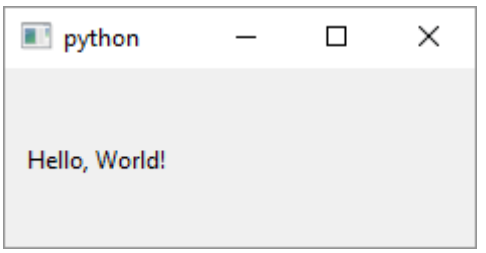
To do that:

1. Create a class that inherits from `QWidget`. This class is used as the application's main window. In the class' `__init__()` method
2. Create a `QVBoxLayout` instance and set it as the window layout. `QVBoxLayout` lays out its child widgets vertically. There are also `QHBoxLayout`, `QFormLayout`, `QGridLayout`, and `QStackedLayout` to

choose from. You don't have to use layouts - you can position widgets manually, but layouts are more flexible and are typically used in Qt/PySide6 applications.
3. Create an instance of the `QLabel` class and add it to the layout. QLabel is a widget that is used to display read-only text or images.

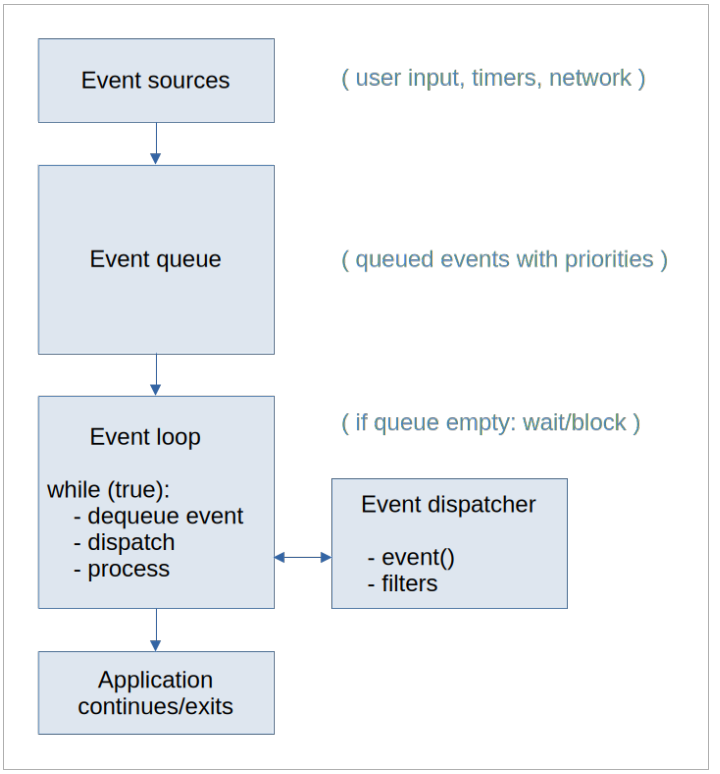
The rest of the code is the same as in the script that shows an empty window, except that we use `qApp` - a variable available after importing `PySide6` - which is equivalent to `QApplication.instance()`.

The result should look like this:



Qt Event Loop

The Qt framework uses event-driven programming - the application flow is determined by events. Mouse clicks, key presses, windowing system, and network events, are posted to the event queue. While the queue is empty the event loop waits for events. Events that arrive are enqueued, then dequeued, and dispatched to the designated `QObject` receiver, which in turn handles them using event handler methods.



In the next several chapters we'll provide an overview of a number of commonly used Qt widgets, but not before we introduce the Qt's signals and slots mechanism.

2. Signals & Slots

The signals and slots mechanism is a core Qt feature used for inter-object communication. A signal is emitted when an event (e.g., a mouse click) occurs and a slot is a Python method or function that gets executed in response. To establish a connection, you must `connect()` a signal to a slot.

2.1 Basic Signals & Slots Mechanism

Your task is to create a button that displays a message in the terminal when clicked.

Here's how to do that:

```
# Signals are event notifications.
# Slots are Python methods/functions.
# A QPushButton is a button that you can push.

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


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the signal sender, ie. the QPushButton.
        #     When you click it the button emits a signal.

        button = QPushButton('Click me!')

```

```
25          # 3 - Connect the signal and the slot.
26          #     Notice there's no parentheses after self.on_button_clicked
27          #     This means you pass a Python function object
28          #     to the connect() method
29
30          button.clicked.connect(self.on_button_clicked)
31
32          layout.addWidget(button)
33
34      # 2 - Create the slot. It's a simple method
35      #     that belongs to our Window class.
36      #     What ever code you put here is
37      #     executed when the button is clicked.
38
39      @Slot(bool)
40      def on_button_clicked(self, checked):
41          print('Button clicked,', 'checked:', checked)
42
43
44  if __name__ == '__main__':
45
46      app = QApplication(sys.argv)
47
48      main_window = Window()
49      main_window.show()
50
51      sys.exit(app.exec())
```

1. Create a `QPushButton` instance and add it to the layout. `QPushButton` is a widget that emits a signal when you click on it. It also supports `pressed()`, `released()` and `toggled()` signals. The `clicked()` signal combines `pressed()` and `released()`.
2. Add a method to the `Window` class that will act as the slot when the button is clicked. We have decorated it with the `Slot()` decorator. This optional - omitting it still works, but the Qt documentation recommends it to avoid runtime overhead.

   Notice the `on_button_clicked()` method's signature includes a `checked` parameter, which matches the signature of `QPushButton.clicked()` signal, which, in turn, maches the parameter passed to the `@Slot()` decorator. In this example, `checked` will always be false because the button's `checkable` property defaults to `False` (and we haven't changed it).You could simplify the slot declaration and use

```
def on_button_clicked(self)
```

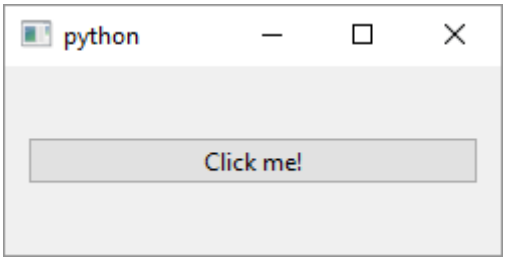
if you don't need the button's `checked` value - it would simply be ignored.[1] The slot just prints a message to the terminal, but you can add any code to it (e.g. to update other widgets or save data).

3. Connect the signal to the slot using

```
object.signal_name.connect(slot_name)
```

Note that there are no parentheses after `self.on_button_clicked` - `connect()` expects a function **object**, not a function call.

Run the script and you'll see a window with a button that prints a message when you click on it.



2.2 Using Python Lambda Functions

Your task is to display a message in the terminal when a button is clicked. `QPushButton.clicked()` doesn't support passing additional parameters so you decide to use a lambda function.

To do this:

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

```
# Demonstrate using a Python lambda function
# as a slot. Lambdas are anonymous functions,
# ie. they have no name.

import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()
        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the widget

        button = QPushButton('Click me!')

        # 2 - In this case the slot is a Python lambda

        button.clicked.connect(
            lambda : self.log('My log message'))

        layout.addWidget(button)

    def log(self, message):
        print('Button clicked')
        print(message)


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Create the widget. It's a `QPushButton` like in the previous example.
2. Connect the `QPushButton.clicked()` signal to the slot. There are two things to note here:

    - You can connect a signal to a Python lambda function (ie. an anonymous function), instead of a named method,

- Lambdas let you pass extra arguments to your slot. In the lambda we call `self.log()` passing it an extra string argument (`My log message`).

When you run the code, clicking the button executes the lambda, which in turn invokes `log()` that prints two lines to the terminal:

```
Button clicked
My log message
```

3. Qt Widgets Layouts

When you create a Qt widget like `QPushButton` you need a way to position it inside a window or a container widget. It is possible, but quite uncommon, to position Qt widgets by setting their x and y coordinates - Most Qt applications use layouts to automatically arrange widgets in rows, columns grids, forms, or stacks, to achieve the desired layout[1].

3.1 Laying out Widgets Vertically - `QVBoxLayout`

`QVBoxLayout` arranges widgets in a column. If you look up the `QVBoxLyout` parents in the Qt documentation, you'll notice that `QWidget` is **not** a `QVBoxLayout`'s parent class. This means you cannot show a layout on the screen - it is not a visual element but a geometry manager.

To add a vertical layout to a `QWidget`:

```
1  # Demonstrate how to stack widgets vertically
2  # using QVBoxLayout.
3  # QVBoxLayout stands for 'vertical box layout object'
4
5  import sys
6
7  from PySide6.QtCore import Slot
8  from PySide6.QtWidgets import (QApplication,
9      QWidget, QPushButton, QVBoxLayout, QSizePolicy)
10
11
12 class Window(QWidget):
13
14     def __init__(self):
15
16         super().__init__()
```

[1]If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

```
17
18          # 1 - Create the layout object
19          #     and set it as our window layout
20
21          layout = QVBoxLayout()
22          self.setLayout(layout)
23
24          # 2 - Create widgets
25
26          button_1 = QPushButton('Button 1')
27          # button_1.setSizePolicy( QSizePolicy.Expanding, QSizePolicy.Expanding)
28          button_2 = QPushButton('Button 2')
29          button_3 = QPushButton('Button 3')
30
31          # 3 - Add widgets to the layout
32
33          layout.addWidget(button_1)
34          layout.addWidget(button_2)
35          layout.addWidget(button_3)
36
37          layout.addStretch()
38
39          # Handle button events for demonstration purposes
40          # Note how I reused the same method (on_button_clicked)
41          # as the slot for all three button widgets.
42
43          button_1.clicked.connect(self.on_button_clicked)
44          button_2.clicked.connect(self.on_button_clicked)
45          button_3.clicked.connect(self.on_button_clicked)
46
47      @Slot()
48      def on_button_clicked(self):
49
50          # Inside the slot use self.sender()
51          # to check which button was clicked:
52
53          print(self.sender().text() + ' clicked.')
54
55
56  if __name__ == '__main__':
57
58      app = QApplication(sys.argv)
59
60      main_window = Window()
61      main_window.show()
62
63      sys.exit(app.exec())
```

1. Create a `QVBoxLayout` instance and set it as your window layout with

`setLayout()`. This also works for any `QWidget` subclass. You can nest layouts inside other layouts to achieve complex layouts.
2. Create widgets you want to lay out. Here we create three `QPushButton` instances.
3. Add them to the layout using `addWidget()`. For nested layouts you would use `addLayout()` instead.

Run the script and resize the window: The buttons stack vertically and expand horizontally to fill the width, but their height stays fixed, creating gaps between them. There are two ways you can change this:

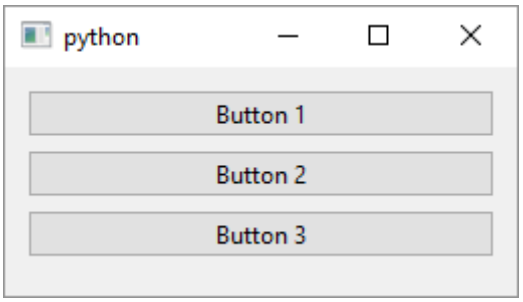
- Add stretchable space with `addStretch()` at the end of the layout - it pushes the widgets to the top while allowing empty space to grow.
- Set a widget's size `Expanding` for both directions, e.g. `button_1.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)`. This lets it resize in both axes.

We've also connected all three buttons' `clicked` signals to a single `@Slot()` method. Inside the slot, you can use `QObject.sender()` to identify the sender: `sender = self.sender()`

as we need a way to tell which button emitted the `clicked` signal. This lets us reuse a single slot to handle signals from different widgets - but keep in mind this warning from the Qt docs:

> Warning: This function violates the object-oriented principle of modularity. However, getting access to the sender might be useful when many signals are connected to a single slot.

When executed, the script shows a window like this:



3.2 Horizontal Layout - `QHBoxLayout`

`QHBoxLayout` arranges widgets horizontally. The setup is the same as for `QVBoxLayout`:

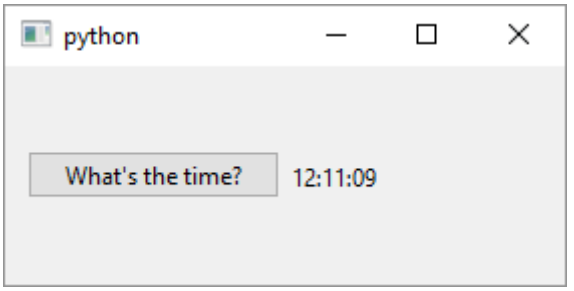
```
# QHBoxLayout is the horizontal box layout
# It lines out widgets horizontally

import sys

from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QLabel, QHBoxLayout)
from PySide6.QtCore import QTime



class Window(QWidget):

    def __init__(self):

        super().__init__()

        # The steps are the same as for the vertical box layout:
        # 1 - Create the layout and set it as the window layout

        layout = QHBoxLayout()
        self.setLayout(layout)

        # 2 - Create some widgets
        #     I add a button and a label for variety purposes

        button = QPushButton("What's the time?")
        self.label = QLabel('')
        self.label.setMinimumWidth(60)

        # I need to access the label from the
        # slot method (ie. on_button_clicked)
        # so I make it a Window instance variable.
        # I also set the label minimum width


        # 3 - Add widgets to the layout

        layout.addWidget(button)
```

```
41          layout.addWidget(self.label)
42
43          # connect button.clicked to a slot
44          # to check if everything works
45
46          button.clicked.connect(self.on_button_clicked)
47
48
49      def on_button_clicked(self):
50
51          # You can use the QtCore.QTime class
52          # to get the current time
53
54          self.label.setText(QTime.currentTime().toString('hh:mm:ss'))
55
56
57  if __name__ == '__main__':
58
59      app = QApplication(sys.argv)
60
61      main_window = Window()
62      main_window.show()
63
64      sys.exit(app.exec())
```

1. Create the layout and assign it to a `QWidget` with `setLayout()`
2. Create child widgets. Here, a `QPushButton` and `QLabel`. We set the label's minimum width to fit its text.
3. Add them to the layout using `addWidget()` in the desired order.

For interactivity, we've connected the button's `clicked` signal to update the label with the current time using `QTime.currentTime()`. This shows an important point: Qt is more than a GUI toolkit and offers tools that overlap with Python's - for instance the developer needs to decide whether to use `time` from the Python standard library or `QTime` from `PySide6,QtCore`.

After execution, you'll see a window like this:



Both layout classes arrange widgets in the order you add them with `addWidget()`, `QVBoxLayout` top to bottom, and `QHBoxLayout` left to right. You can change this their `setDirection()` method with the `QBoxLayout.BottomToTop` or `QBoxLayout.RightToLeft`. You can achieve dynamic layouts by using `removeWidget()` to remove a widget from a layout or `insertWidget()` to insert a widget out of order.

If you create a widget without setting its parent, as we do in the above examples, adding them to a widget's layout reparents them to that widget.

3.3 Grid Layout - `QGridLayout`

QGridLayout positions widgets in a grid. Unlike linear layouts, you specify each widget's row and column when adding it.

To use a grid layout:

```
# QGridLayout lays out widgets, well, in a grid

import sys

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


class Window(QWidget):

    def __init__(self):

```

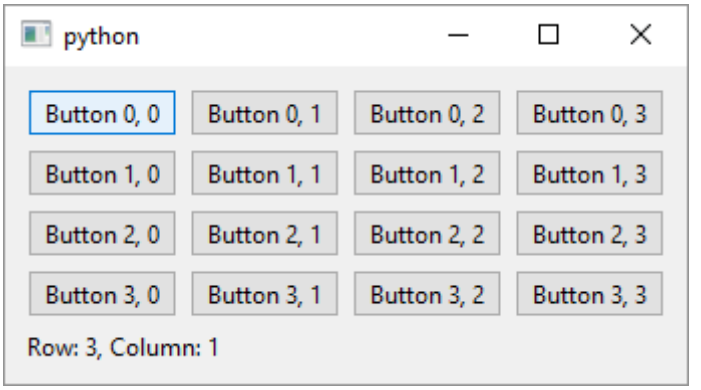
```
14          super().__init__()
15
16          # 1 - Create the layout:
17
18          layout = QGridLayout()
19          self.setLayout(layout)
20
21          # 2 - Create widgets and add them to the grid
22          # We create sixteen buttons in a 4x4 grid
23          # (0, 0) is the top left cell
24          # The first integer is the row number
25          # The second integer is the column number
26
27          for r in range(4):
28              for c in range(4):
29
30                  button = QPushButton(f'Button {r}, {c}')
31                  button.clicked.connect(self.on_button_clicked)
32                  self.layout().addWidget(button, r, c)
33
34          # The label is put in row 4, column 0
35          # and spans one row and four columns.
36
37          self.label = QLabel()
38
39          layout.addWidget(self.label, 4, 0, 1, 4)
40
41      # The slot method
42
43      @Slot()
44      def on_button_clicked(self):
45
46          # Get a reference to the clicked button
47          # and its position in the grid view
48
49          clicked_button = self.sender()
50
51          index = self.layout().indexOf(clicked_button)
52          position = self.layout().getItemPosition(index)
53          r, c, _, _ = position
54          self.label.setText(f'Row: {r}, Column: {c}')
55
56
57  if __name__ == '__main__':
58
59      app = QApplication(sys.argv)
60
61      main_window = Window()
62      main_window.show()
63
```

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

1. Create a `QGridLayout` instance and set it as a `QWidget` layout.
2. Create widgets and add them with `addWidget(widget, row, col)`.Leave cells empty for gaps. For spanning, use `addWidget(widget, row, col, row_span, col_span)` - e.g., `addWidget(self.label, 4, 0, 1, 4)` for a label spanning one row over four columns.

Here, we populate a 4x4 grid with 16 `QPushButtons` using two nested loops. Using `sender()` in the slot lets us identify the clicked button.

After execution, the window looks like this:



Using the documentation

The PySide6 Documentation [2] is, for the most part, auto-generated, as indicated by this warning:

This documentation may contain snippets that were automatically translated from C++ to Python. We always welcome contributions to the snippet translation. If you see an issue with the translation, you can also let us know by creating a ticket on https:/bugreports.qt.io/projects/PYSIDE

The Qt documentation[3] itself, on the other hand, is very good - each class is documented in detail (`QObject`, for instance [4]), having details like:

- Parent and child classes
- Properties

- Functions
- Slots
- Signals
- Detailed description

Each function is documented with its C++ signature, description, and often with a short C++ code snipped demonstrating its use. Most of the time you can refer to it, consulting the PySide6 docs for Python-specific details.

3.4 Form Layout - `QFormLayout`

`QFormLayout` creates a two-column form layout: labels on the left, fields on the right, similar to HTML forms.

To implement it:

```
# QFormLayout lets you create forms

import sys
from PySide6.QtCore import Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QFormLayout, QLineEdit, QLabel)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        # 1 - Create the layout and set it as the window layout

        layout = QFormLayout()
        self.setLayout(layout)

```

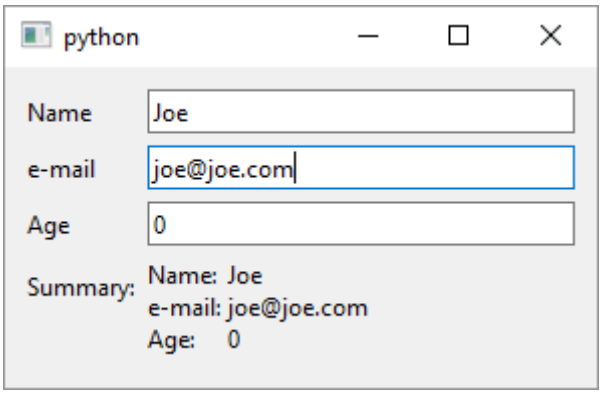
[2]https://doc.qt.io/qtforpython-6/
[3]https://doc.qt.io/
[4]https://doc.qt.io/qt-6/qobject.html

```
20          # 2 - Create widgets
21          #     Will need to access all widgets from the slot
22
23          self.name_edit = QLineEdit()
24          self.email_edit = QLineEdit()
25          self.age_edit = QLineEdit()
26          # The label will display entered data
27          self.summary_label = QLabel()
28
29          # 3 - Add widgets to the form
30          #     QFormLayout will automagically add a label
31          #     for each widget based on the text you pass to addRow()
32
33          layout.addRow('Name', self.name_edit)
34          layout.addRow('e-mail', self.email_edit)
35          layout.addRow('Age', self.age_edit)
36          layout.addRow('Summary:', self.summary_label)
37
38          # self.on_editing_finished() will handle events for all widgets.
39          # editingFinished() is fired when Return is pressed
40          # or the line edit loses focus.
41
42          self.name_edit.editingFinished.connect(self.on_editing_finished)
43          self.email_edit.editingFinished.connect(self.on_editing_finished)
44          self.age_edit.editingFinished.connect(self.on_editing_finished)
45
46      @Slot()
47      def on_editing_finished(self):
48
49          self.summary_label.setText(
50              f'Name:\t{self.name_edit.text()}\n' +
51              f'e-mail:\t{self.email_edit.text()}\n' +
52              f'Age:\t{self.age_edit.text()}')
53
54
55  if __name__ == '__main__':
56
57      app = QApplication(sys.argv)
58
59      main_window = Window()
60      main_window.show()
61
62      sys.exit(app.exec())
```

1. Create a `QFormLayout` instance and set it as the `QWidget` layout.
2. Create child widgets. Since we need to access the widgets from the slot method, we make the them `Windows` instance members.

3. Add child widgets to the form using `QFormLayout.addRow(label_text, widget)`. `addRow()` is a convenience method that automatically creates a `QLabel` instance for you, setting its text to `label_text`.

In this example, three `QlineEdit` fields handle user input. Their `editingFinished` is emitted on pressing Enter/Return or when focus leaves the changed field. A single slot updates a summary `QLabel` with the values. (No `sender()` needed here since we reference widgets directly.)



4. Display Widgets

Display widgets are read-only widgets used to provide information to the user. They include

- `Qlabel` for text, images, and animations,
- `QProgressBar` for showing task progress,
- `QLCDNumber` for displaying number in an LCD-style,
- `QTextBrowser` for read-only rich text with hyperlink support.

4.1 Displaying Text with `QLabel`

`QLabel` supports three text formats: plain text, Markdown, and rich text (rendered using a subset of HTML). The format is controlled via `setTextFormat()`. Available constants are:

Value	Enum Constant	Description
0	`Qt::PlainText`	Plain text
1	`Qt::RichText`	Qt-supported HTML subset
2	`Qt::AutoText`	Format automatically detected (default)
3	`Qt::MarkdownText`	Markdown formatting

To display text using `QLabel`:

```
import sys
from PySide6.QtCore import Qt
from PySide6.QtWidgets import (QApplication,
    QWidget, QLabel, QVBoxLayout)

class Window(QWidget):

    def __init__(self):

        super().__init__()
        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create QLabel objects and set their text

        plain_text_label = QLabel('Plain text label')
        markdown_label = QLabel('**Markdown** label')
        rich_text_label = QLabel('<b>Rich text</b> label')

        # 2. Optionally, set their text format

        markdown_label.setTextFormat(Qt.TextFormat.MarkdownText)
        rich_text_label.setTextFormat(Qt.TextFormat.RichText)

        # 3. Add the objects to the layout

        layout.addWidget(plain_text_label)
        layout.addWidget(markdown_label)
        layout.addWidget(rich_text_label)

if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Create a `QLabel` object, setting its text. We create three labels:

    - one with plain text (no formatting),
    - one with Markdown syntax,
    - one with HTML syntax.

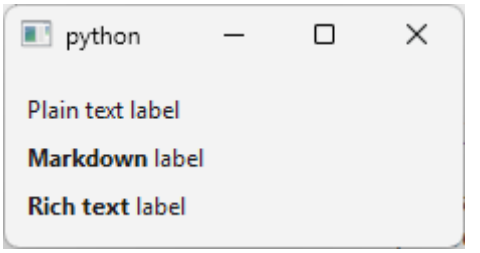
2. Set the text format for the Markdown and rich-text labels using `setTextFormat()`:

    - `Qt.TextFormat.MarkdownText` for the Markdown text,

- `Qt.TextFormat.RichText` for the rich-text label.

3. Add the label(s) to the layout and display the window. When the window is shown, the Markdown and rich-text labels will display bold text.

When you run the application you should see a window like this:



4.2 Displaying Images with `QLabel`

`QLabel` can display static images, vector graphics, and animated GIFs using the following methods:

Method	Description
`setPixmap()`	Displays a static image from a QPixmap
`setPicture()`	Displays a vector graphic from a QPicture
`setMovie()`	Displays an animated GIF or movie from a QMovie

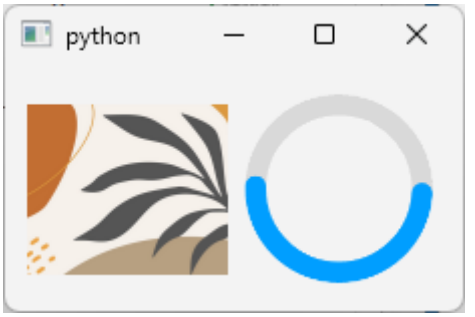
To display an image in a label:

```
import sys
from PySide6.QtCore import QSize, Qt
from PySide6.QtGui import QPixmap, QMovie
from PySide6.QtWidgets import (QApplication,
    QWidget, QLabel, QHBoxLayout)

class Window(QWidget):

    def __init__(self):

        super().__init__()
        layout = QHBoxLayout()
        self.setLayout(layout)

        # 1. Create a QPixmap object
        #    Optionally, resize it.

        pixmap = QPixmap('image.png')
        resized_pixmap = pixmap.scaledToWidth(
            100, Qt.TransformationMode.SmoothTransformation)

        # 2. Create a QLabel object

        png_label = QLabel()

        # 3. Set the label's pixmap

        # Same steps for animated gif

        png_label.setPixmap(resized_pixmap)
        layout.addWidget(png_label)

        animated_gif = QMovie('spinner.gif')
        animated_gif.setScaledSize(QSize(100, 100))
        animated_gif.start()

        gif_label = QLabel()
        gif_label.setMovie(animated_gif)
        layout.addWidget(gif_label)

if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Create an image object. We create two pictures: a png image using `QPixmap` and an animated gif image using `QMovie`,

2. Create label objects that will display the images.
3. Set the labels' contents to the png and the gif images.

The running application should look like this:



4.3 Displaying LCD-like Numbers with `QLCDNumber`

As the documentation states, `QLCDNumber` is the very oldest part of Qt, tracing its roots back to a BASIC program on the Sinclair Spectrum.

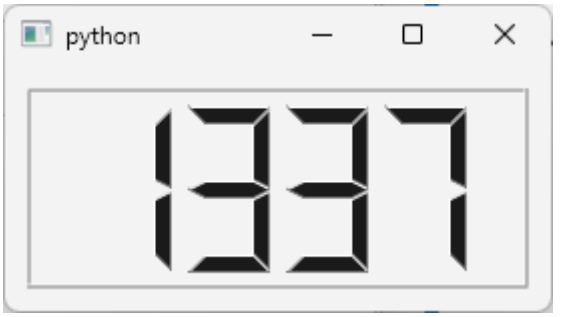
To use it in your application:

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QLCDNumber, QVBoxLayout)

class Window(QWidget):

    # Displays an LCD-like number

    def __init__(self):

        super().__init__()
        layout = QVBoxLayout()
        self.setLayout(layout)

        self.lcd_number = QLCDNumber()
        self.lcd_number.setFixedSize(250, 100)
```

```
17          self.lcd_number.setDigitCount(4)
18          self.lcd_number.display(1337)
19          layout.addWidget(self.lcd_number)
20
21
22  if __name__ == '__main__':
23
24      app = QApplication(sys.argv)
25      main_window = Window()
26      main_window.show()
27      sys.exit(app.exec())
```

1. Create a `QLCDNumber` instance,
2. Set its digit count,
3. Set the number to display using `QLCDNumber.display()`

When you start the application, you should see this:

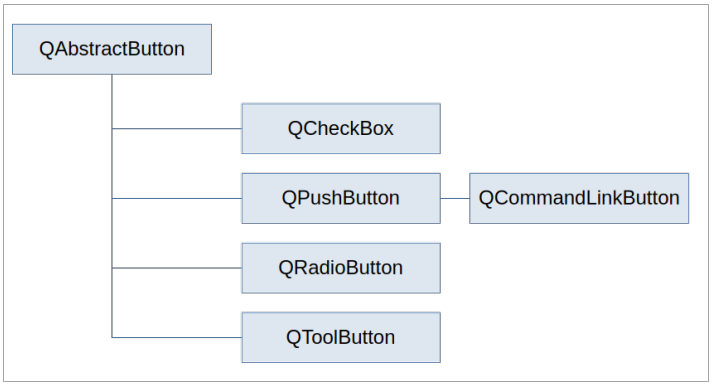


You are not restricted to `QLabel` for images - any widget can draw images by reimplementing its `paintEvent()` method, giving you full control over rendering.

5. Qt Widgets Buttons

Non-interactive display widgets like `QLabel` show text or images to the user but cannot accept input. Buttons, on the other hand, can accept user input, which is typically binary (checked or unchecked, on or off). Not all buttons are checkable by default (e.g., `QPushButton` requires explicit configuration), while `QCheckBox` supports an optional tri-state mode with three states: checked, unchecked, and partially checked.

Qt provides several button classes, all inheriting from `QAbstractButton`:



Qt Button Class	Purpose
`QPushButton`	Lets the user send a command to the application. Supports text, icons, and optional toggling.
`QCheckBox`	Used for non-mutually exclusive options (e.g., preferences). Supports binary or tri-state.
`QRadioButton`	Mutually exclusive choices within a group. Automatically deselects other radio buttons in the group.

Qt Button Class	Purpose
`QToolButton`	For use in toolbars (e.g., “Save” icon). Supports popups for sub-menus.
`QCommandLinkButton`	Mimics Windows Vista command link style.

All button classes share these signals:

Signal	Emitted
`clicked(bool checked=False)`	When the button is clicked.
`pressed()`	When the button is pressed down.
`released()`	When the button is released.
`toggled(bool checked)`	When a checkable button changes its check state.

`toggled()` is only emitted for buttons where `setCheckable(True)` has been called. `clicked()` is emitted for both checkable and non-checkable buttons.

You can optionally add text, an icon, a shortcut, and a tooltip to a button using these methods:

Method
`setText(text)`
`setIcon(icon)`
`setShortcut(shortcut)`
`setToolTip(text)`

5.1 QPushButton

QPushButton is a common GUI element that lets the user send a command to the application. `toggled()` is useful only if you set the `QPushButton.checkable` property to `True`, which effectively turns it into a toggle button: once pressed, it stays pressed until pressed again.

You are given a task to show a random integer in a label when a button is clicked.

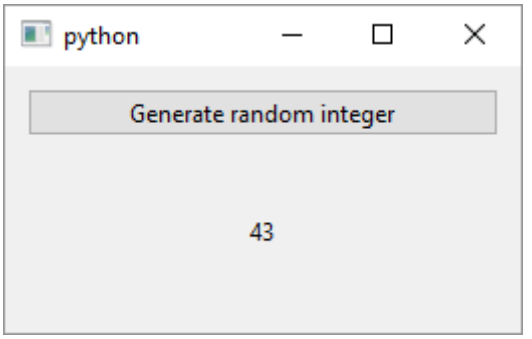
To do this:

```
# QPushButton generates 'clicked' signals

from random import randint
import sys

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


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the button and add it to the window layout

        button = QPushButton('Generate random integer')

        # 3 - Connect the button clicked event with the slot

        button.clicked.connect(self.on_button_clicked)

        self.label = QLabel()
        self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        layout.addWidget(button)
        layout.addWidget(self.label)
```

```
34
35      # 2 - Create the method/function to use as the slot
36
37      @Slot()
38      def on_button_clicked(self):
39          self.label.setNum(randint(0, 100))
40
41
42  if __name__ == '__main__':
43
44      app = QApplication(sys.argv)
45
46      main_window = Window()
47      main_window.show()
48
49      sys.exit(app.exec())
```

1. Create a `QPushButton` object, setting the text to be displayed on the button.
2. Create the method to be used as the slot - this is the method that will be called when the button is clicked. In the example, the method is named `on_button_clicked()` and sets a label's text to a random integer. Note that the label is added to `Window` as an instance variable so we can access it from the slot.
3. Connect the button's `clicked()` signal with the slot.

After running the application and clicking the button you should see a window like this:



Qt Widgets Constructors

If you look at `QPushButton` documentation[1] you'll notice that it has three constructors (listed as the methods with the same name as the class):

- The one that optionally accepts only the button's parent. If you use it the button will initially be shown without any text,
- The one that, in addition to parent, accepts the text to be displayed on the button. This is the one we use in the example,
- The one that, in addition to parent and text, accepts an icon so you can add an image to a button when creating it.

It is common for Qt classes to have more than one constructor - consult the documentation to choose the right one for your use case.

5.2 QCheckBox

`QCheckBox` is a widget that consists of a graphical check box and a label. it can be checked, unchecked and - if you set its `tristate` property to `True` - partially checked. Checkboxes are used to represent options that are **not** mutually exclusive - more than one checkbox in a group can be checked at the same time.

Your task is to allow the user to select one or more operating systems.

To do this:

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

```
# A QCheckBox is an option button that can be
# checked, unchecked or partially checked.

import sys

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


class Window(QWidget):

    def __init__(self):

        super().__init__()

        self.resize(300, 200)
        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the checkboxes and add them to the layout

        self.windows_checkbox = QCheckBox('Windows')
        self.windows_checkbox.setObjectName('Windows')

        self.linux_checkbox = QCheckBox('Linux')
        self.linux_checkbox.setObjectName('Linux')

        self.macos_checkbox = QCheckBox('macOS')
        self.macos_checkbox.setObjectName('macOS')
        self.macos_checkbox.setTristate(True)

        self.label = QLabel()

        layout.addWidget(self.windows_checkbox)
        layout.addWidget(self.linux_checkbox)
        layout.addWidget(self.macos_checkbox)

        layout.addWidget(self.label)

        # 3 - For each checkbox connect QCheckBox.
        #     checkStateChanged signal with the slot.
        #     checkStateChanged is emitted when a checkbox
        #     state changes.

        self.windows_checkbox.checkStateChanged.connect(
            self.on_state_changed)
        self.linux_checkbox.checkStateChanged.connect(
            self.on_state_changed)
        self.macos_checkbox.checkStateChanged.connect(
```

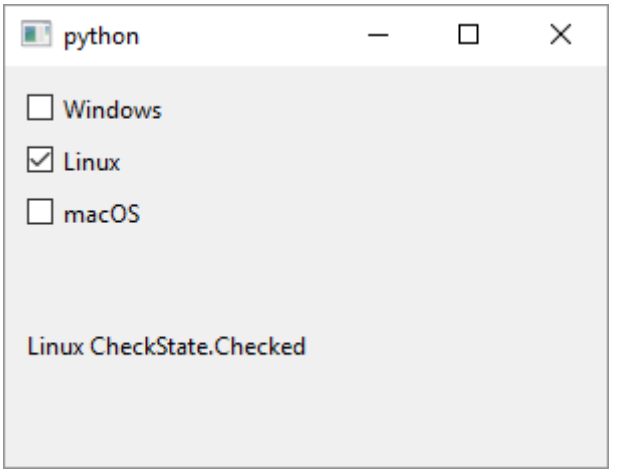
```
51              self.on_state_changed)
52
53      # 2 - Create the slot. Mark it with the @Slot() decorator
54      #     If you don't import PySide6.QtCore.Qt above, you will
55      #     get an error: TypeError: Cannot call meta function ...
56
57      @Slot(Qt.CheckState)
58      def on_state_changed(self, state):
59          sender_name = self.sender().objectName()
60          self.label.setText(f'{sender_name} {state}')
61
62
63  if __name__ == '__main__':
64
65      app = QApplication(sys.argv)
66      main_window = Window()
67      main_window.show()
68      sys.exit(app.exec())
```

1. Create a `QCheckBox` object and set its display text. We create three checkboxes and add them to the window's layout. We also set each checkbox `objectName` property. Setting `objectName` can be useful in debugging or when you need to find a widget using `QObject.findChild()`.
2. Create the slot to process checkboxes' `checkStateChanged` signals. The documentation suggests that you mark it with the `Slot()` decorator but if you do you also need to import the `Qt` class from `PySide6.QtCore`. If you don't, you get the following error:

   > TypeError: Cannot call meta function "void on_state_changed(Qt::CheckState)" because parameter 0 of type "Qt::CheckState" cannot be converted.

   This is because the `state` argument is of type `Qt.CheckState` and `Qt.CheckState`, like many other Qt enums, resides in `PySide6.QtCore.Qt` so you'll often need to import it. So, in this case, either import `Qt` or leave out the `Slot()` decorator.

   The slot sets the label's text to the signal sender's object name.
3. Finally, connect each checkbox `checkStateChanged` signal to the created slot.



5.3 QRadioButton

`QRadioButton` is a widget that presents the user with a set of mutually exclusive options. When you select (check) a radio button all other radio buttons that have the same parent widget are automatically unselected.

You want to let the user select a label's background color.

To do this:

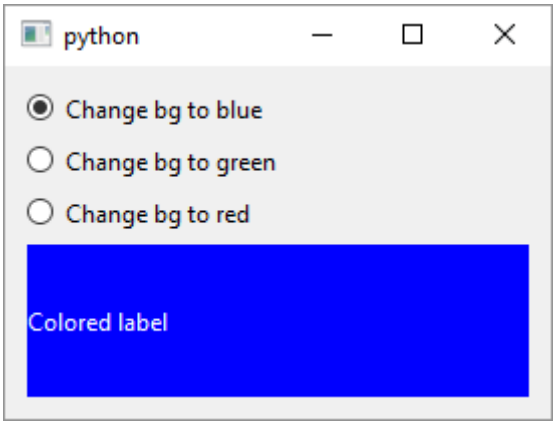
```
# Radio buttons typically present the user
# with a "one of many" choice. In a group of radio buttons,
# only one radio button at a time can be checked

from PySide6.QtCore import Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QRadioButton, QLabel)
import sys


class Window(QWidget):

    def __init__(self):

        super().__init__()

```

```
17          layout = QVBoxLayout()
18          self.setLayout(layout)
19
20          # 1 - Create the radio buttons and add them to the layout
21
22          self.blue_radiobutton = QRadioButton('Change bg to blue')
23          self.green_radiobutton = QRadioButton('Change bg to green')
24          self.red_radiobutton = QRadioButton('Change bg to red')
25
26          self.label = QLabel('Colored label')
27
28          layout.addWidget(self.blue_radiobutton)
29          layout.addWidget(self.green_radiobutton)
30          layout.addWidget(self.red_radiobutton)
31          layout.addWidget(self.label)
32
33          # 3. Connect the radio buttons' toggled signal
34          #    to the slot
35
36          self.blue_radiobutton.toggled.connect(
37              lambda: self.on_button_toggled('blue'))
38          self.green_radiobutton.toggled.connect(
39              lambda: self.on_button_toggled('green'))
40          self.red_radiobutton.toggled.connect(
41              lambda: self.on_button_toggled('red'))
42
43          self.blue_radiobutton.setChecked(True)
44
45      # 2 - Create the slot to handle radio button toggled() signal
46
47      @Slot(str)
48      def on_button_toggled(self, color):
49
50          self.label.setStyleSheet(
51              f'background-color:{color}; color: white;')
52
53
54  if __name__ == '__main__':
55
56      app = QApplication(sys.argv)
57
58      main_window = Window()
59      main_window.show()
60
61      sys.exit(app.exec())
```

1. Create the radio button objects and add them to the parent widget. In this example all three radio buttons have the same parent widget - the `Window` object which is also the top level widget.

2. Create the slot to handle the radio buttons `toggled()` signal. In this case the slot has an argument that `toggled` does not provide named `color`. `color` holds a string containing a color name and each radio button provides an appropriate color to the slot using a lambda. Qt offers its own version of CSS (cascading style sheets), called QSS[2] which you can apply to widgets using `QWidget.setStyleSheet()`. This is how we change the label's background and foreground colors when a raio button is toggled.
3. Connect the signals and the slot. In this case lambdas are needed to pass the extra argument to the slot.

Radio buttons form a group of mutually exclusive states based on their common parent widget but you can customize this behavior using `QButtonGroup`[3].



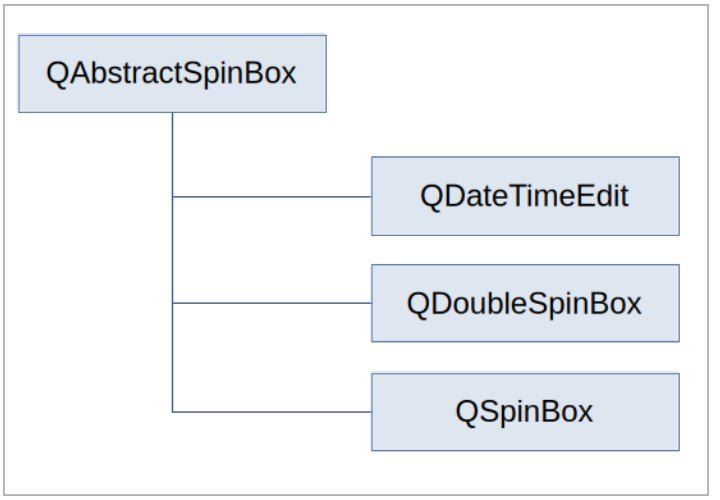
[2]https://doc.qt.io/qtforpython-6/
[3]https://doc.qt.io/

6. Numeric Widgets

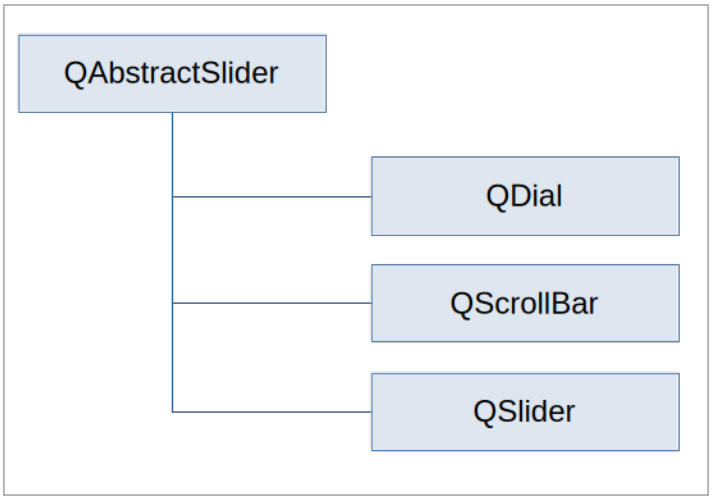
Qt offers several widgets designed for numeric input:

- `QSpinBox` - integer values
- `QDoubleSpinBox` - floating-point (decimal) values
- `QSlider` - linear selection with a sliding handle
- `QDial` - selection with a circular dial

These widgets form two distinct inheritance trees. `QDoubleSpinBox` and `QSpinBox` are inherited from `QAbstractSpinBox`



While `QSlider` and `QDial` are inherited from `QAbstractSlider`:



All of them are bounded (they have configurable minimum and maximum values) and emit the `valueChanged()` when the user modifies the value.

6.1 QSpinBox

A `QSpinBox` allows the user select an integer either by clicking the up/down arrows, pressing the keyboard arrows or typing directly.

You want to enable the user to select the amount of RAM.

To do this:

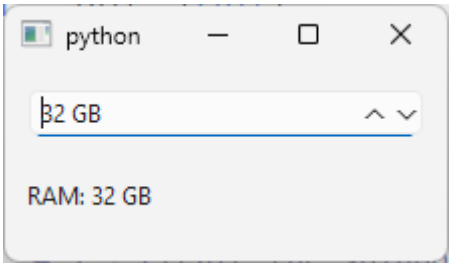
```
# The QSpinBox class provides a spin box widget.
# Access the current value using its value property

import sys
from PySide6.QtCore import Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QSpinBox, QLabel)


class Window(QWidget):

    def __init__(self):

```

```
14          super().__init__()
15
16          layout = QVBoxLayout()
17          self.setLayout(layout)
18
19          # 1 - Create the spinbox and set its properties
20
21          self.spinbox = QSpinBox()
22          self.spinbox.setSuffix(' GB')    # visual unit
23          self.spinbox.setValue(32)        # initial value
24
25          # Set valid RAM sizes: from 16 GB
26          # to 256 GB in steps of 2 GB.
27
28          self.spinbox.setRange(16, 256)
29          self.spinbox.setSingleStep(2)
30          layout.addWidget(self.spinbox)
31
32          self.label = QLabel()
33          self.label.setText('RAM: 32 GB')
34          layout.addWidget(self.label)
35
36          # 3. Connect the valueChanged signal with the slot
37
38          self.spinbox.valueChanged.connect(self.set_ram)
39
40      # 2. Create the slot. The value passed
41      #    from the signal is an integer.
42
43      @Slot(int)
44      def set_ram(self, value):
45          self.label.setText(f'RAM: {value} GB')
46
47
48  if __name__ == '__main__':
49
50      app = QApplication(sys.argv)
51      main_window = Window()
52      main_window.show()
53      sys.exit(app.exec())
```

1. Create the `QSpinBox` instance and configure it. The default spinbox range is 0–99, but in this example we restrict it to realistic RAM sizes (16–256 GB), set the step size 2 GB, the initial value to 32 GB, and append the unit with `setSuffix(' GB')`.
2. Define a slot that receives the new value and decorate it with `@Slot(int)`. The slot name is `set_ram()` and it sets a label's text to the spinbox value.

3. Connect the `valueChanged` signal to the slot. The slot receives the spinbox value as its argument.

The application looks like this:



6.2 QDoubleSpinBox

`QDoubleSpinBox` lets the user select a floating-point (`float`) value. In this example we use it to control the opacity of another widget (a green label).

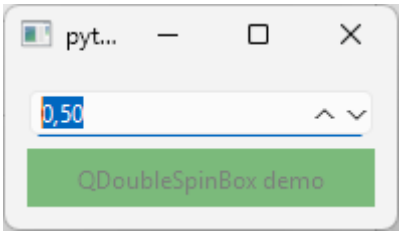
You need to let the user set a label's opacity.

To do this:

```
# QDoubleSpinBox provides a
# spin box widget that takes doubles.

import sys
from PySide6.QtCore import Slot, Qt
from PySide6.QtWidgets import (QApplication, QWidget,
    QVBoxLayout, QDoubleSpinBox, QLabel, QGraphicsOpacityEffect)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
```

```
17          self.setLayout(layout)
18
19          # 1. Create the spinbox and set its properties
20
21          self.spinbox = QDoubleSpinBox()
22          self.spinbox.setRange(0, 1)
23          self.spinbox.setDecimals(2)
24          self.spinbox.setSingleStep(0.05)
25          self.spinbox.setValue(1)
26
27          # 3. Connect the valueChanged signal with the slot
28
29          self.spinbox.valueChanged.connect(self.change_opacity)
30
31          self.label = QLabel('QDoubleSpinBox demo')
32          self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
33          self.label.setStyleSheet('background-color: #008000;')
34
35          layout.addWidget(self.spinbox)
36          layout.addWidget(self.label)
37
38          self.effect = QGraphicsOpacityEffect()
39          self.effect.setOpacity(1)
40          self.label.setGraphicsEffect(self.effect)
41
42      # 2. Create a slot to handle its valueChanged signals.
43
44      @Slot(float)
45      def change_opacity(self, value):
46          self.effect.setOpacity(value)
47
48
49  if __name__ == '__main__':
50
51      app = QApplication(sys.argv)
52      main_window = Window()
53      main_window.show()
54      sys.exit(app.exec())
```

1. Create the spinbox and configure its properties. We limit the range to 0.0-1.0, and choose a single step of 0.05.
2. Create a slot to react to the `valueChanged()` signal. The slot `change_opacity()` receives the new value as a `float`. We appl it to `QGraphicsOpacityEffect` attached to the target widget. The effect's `setOpacity()` method accepts values between 0.0 (fully transparent) and 1.0 (fully opaque).
3. Connect the signal to the slot.



6.3 `QSlider`

`QSlider` provides a vertical or horizontal slider that lets the user select an integer value by dragging a handle. Its main signals are:

Signal	Description
`valueChanged(int)`	Emitted when the slider value changes (by user or programmatically)
`sliderPressed()`	Emitted when the user starts dragging the handle
`sliderMoved(int)`	Emitted when the slider is dragged by the user (not on `setVBalue()`
`sliderReleased()`	Emitted when the user releases the handle

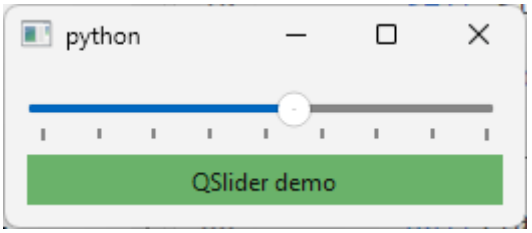
You are given the task to let the user set a label's opacity.

To do this:

```python
# QSlider provides a vertical or horizontal slider.

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


class Window(QWidget):

    # # A class-level constant for the stylesheet template
    LABEL_STYLE = "background-color: rgba(0, 128, 0, {});"

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create a slider and set its properties

        self.slider = QSlider()
        self.slider.setRange(0, 255)
        self.slider.setValue(255)
        self.slider.setPageStep(10)
        self.slider.setTickPosition(QSlider.TickPosition.TicksBelow)
        self.slider.setTickInterval(32)
        self.slider.setOrientation(Qt.Orientation.Horizontal)

        # 3. Connect signal

        self.slider.valueChanged.connect(self.change_opacity)

        self.label = QLabel('QSlider demo')
        self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
        self.label.setStyleSheet(Window.LABEL_STYLE.format(255))

        layout.addWidget(self.slider)
        layout.addWidget(self.label)

    # 2. Slot that receives the new opacity value.

    @Slot(int)
    def change_opacity(self, value):
        self.label.setStyleSheet(Window.LABEL_STYLE.format(value))


if __name__ == '__main__':

```

```
    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Create and configure the slider. Set the range with `setRange(min, max)` In this example we use 0-255 because we want to control the alpha channel of a color in a stylesheet (`rgba(0, 128, 0, alpha)`). We start at 255 (fully opaque) and make the slider horizontal.
2. Create a slot to react to value changes. The slot receives an `int`. Here qe dynamically update a `QLabels` stylesheet, inserting the slider value as the alpha component of an `rgba()` color.
3. Connect the signal to the slot.



6.4 `QDial`

The QDial class provides a rounded range control. Unlike the `QSlider`, which is linear, `QDial` lets the user "turn" a value by dragging around a circular widget.

You need to create a sound volume control in your app.

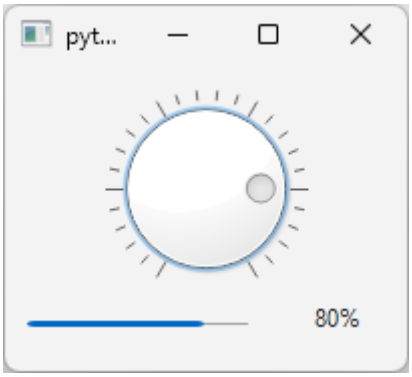
To do this:

```python
# The QDial class provides a rounded range control

import sys
from PySide6.QtCore import Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QDial, QProgressBar)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the dial

        self.dial = QDial()
        self.dial.setRange(0, 100)
        self.dial.setValue(50)
        self.dial.setNotchesVisible(True)

        self.volume_bar = QProgressBar()
        self.volume_bar.setFormat('%v%')
        self.volume_bar.setValue(self.dial.value())

        layout.addWidget(self.dial)
        layout.addWidget(self.volume_bar)

        # 3. Connect the signal the slot

        self.dial.valueChanged.connect(self.set_volume)

    # 2.Create the slot.

    @Slot(int)
    def set_volume(self, value):
        self.volume_bar.setValue(value)


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Create and configure the dial.

2. Create a slot to respond to the dial value changes.
3. Connect the signal to the slot.

In the example, turning the `QDial` updates a `QProgressBar` in real time.



7. Text Widgets

While Qt buttons and the numeric widgets let the user enter binary and numerical data, text widgets support plain text and rich text input. Qt offers three text widgets:

- `QLineEdit` - single-line plain-text editing,
- `QTextEdit` - provides multi-line richtext with Markdown and HTML[1] support,
- `QPlainTextEdit` - multi-line plain-text with optional support for syntax coloring.

7.1 `QLineEdit`

`QLineEdit` provides a single-line plain-text editor. It supports the usual edit operations (copy, cut, paste, undo, redo) via standard keyboard shortcuts and a built-in context menu. You can restrict input length with `setMaxLength()`, validate input using a `QValidator` or input mask, and turn the widget into a password field by setting `echoMode` to `EchoMode.Password`.

Here are the `QLineEdit` signals you are likely to use:

Signal	When Emitted
`textChanged(text)`	Any change to the text (user or programmatic)
`textEdited(text)`	Only when the user types, pastes, or deletes
`returnPressed()`	Enter/Return key is pressed
`editingFinished()`	Enter pressed or the line edit loses focus
`selectionChanged()`	Selected text changes

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

Signal	When Emitted
`inputRejected()`	A validator rejects the input

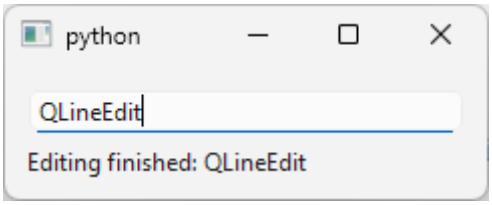
Suppose you need a text field that accepts only letters and warns the user about invalid input.

Here's how to do it:

```
# The QLineEdit widget is a one-line text editor.

import sys

from PySide6.QtCore import Slot
from PySide6.QtGui import QRegularExpressionValidator
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QLineEdit, QLabel)


class Window(QWidget):

    def __init__(self):

        super().__init__()
        self.resize(240, 20)

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create a line edit widget instance

        self.line_edit = QLineEdit()
        validator = QRegularExpressionValidator('^[a-zA-Z]*$')
        self.line_edit.setValidator(validator)

        self.label = QLabel()

        # 3 - Connect the signals with the slots

        self.line_edit.editingFinished.connect(self.on_editing_finished)
        self.line_edit.inputRejected.connect(self.on_input_rejected)

        layout.addWidget(self.line_edit)
        layout.addWidget(self.label)

```

```
37      # 2 - Create methods to handle line edit signals.
38
39      @Slot()
40      def on_editing_finished(self):
41          self.label.setText(
42              f'Editing finished: {self.line_edit.text()}')
43
44      @Slot()
45      def on_input_rejected(self):
46          self.label.setText('Only letters allowed')
47
48
49  if __name__ == '__main__':
50
51      app = QApplication(sys.argv)
52      main_window = Window()
53      main_window.show()
54      sys.exit(app.exec())
```

1. Create a `QLineEdit` object, attach a regular expression validator that allows only letters, and add a `Qlabel` to display feedback.
2. Implement two slots: one to show the final text when editing finishes, and another to warn the user when an invalid character is rejected.
3. Connect the `editingFinished()` and `inputRejected()` signal to the corresponding slots.



7.2 `QTextEdit`

`QTextEdit` is a widget for displaying and editing both plain text and rich text. It can handle large documents efficiently and can be used as an advanced WYSIWYG editor that support rich text formatting via HTML or Markdown. Internally, `QtextEdit` uses a `QTextDocument` object, which organizes content in a hierarchy of frames, blocks and fragments.

Commonly used `QTextEdit` signals:

Signal	Description
`textChanged()`	Emitted when document's content changes
`cursorPositionChanged()`	Emitted when cursor position changes
`copyAvailable(yes)`	Emitted when the availability of copy changes
`redoAvailable(available)`	Emitted when redo becomes available/unavailable
`undoAvailable(available)`	Emitted when undo becomes available/unavailable
`selectionChanged()`	Emitted when current selection changes

Note: `textChanged()` is emitted for both user edits and programmatic changes (e.g., calling `setPlainText()`, `setHtml()`, or `setMarkdown()`).

Let's say you need to create a basic Markdown editor with side-by-side live preview.

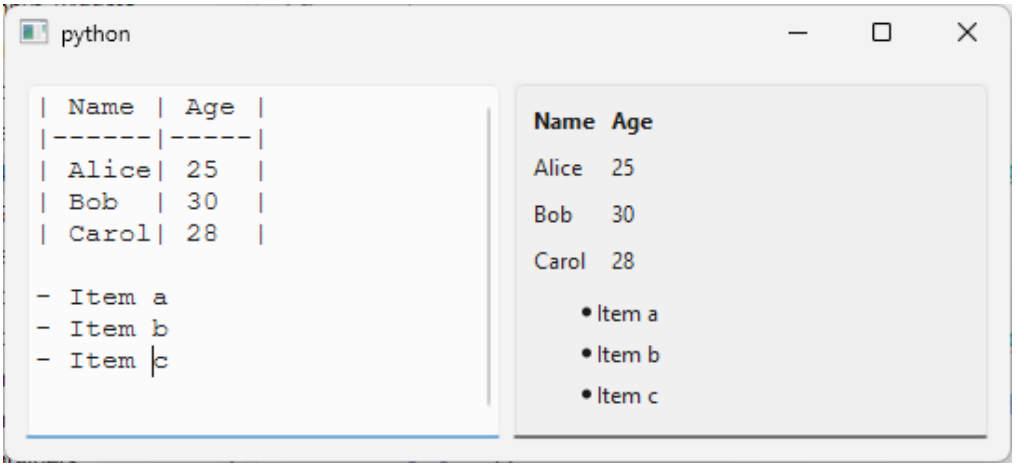
To do it:

```
# The QTextEdit class provides a widget that is used
# to edit and display both plain and rich text.

import sys
from PySide6.QtCore import Slot
from PySide6.QtGui import QFontDatabase
from PySide6.QtWidgets import (QApplication,
    QWidget, QHBoxLayout, QTextEdit)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QHBoxLayout()
        self.setLayout(layout)

```

```
20          # 1. Create the source textedit instance
21
22          self.src = QTextEdit()
23          self.src.textChanged.connect(self.update_preview)
24
25          mono = QFontDatabase.systemFont(
26              QFontDatabase.SystemFont.FixedFont)
27          mono.setPointSize(11)
28          self.src.setFont(mono)
29
30          # 2. Create the preview textedit instance
31
32          self.preview = QTextEdit()
33          self.preview.setReadOnly(True)
34          self.preview.setStyleSheet('background-color: #f0f0f0;')
35
36          layout.addWidget(self.src)
37          layout.addWidget(self.preview)
38
39          self.src.setText('### Enter Markdown Text\n\n')
40
41      # 3. Implement the slot to preview the entered text
42
43      @Slot()
44      def update_preview(self):
45          markdown_text = self.src.toPlainText()
46          self.preview.setMarkdown(markdown_text)
47
48
49  if __name__ == '__main__':
50
51      app = QApplication(sys.argv)
52      main_window = Window()
53      main_window.show()
54      sys.exit(app.exec())
```

1. Create the source editor. Instantiate `QTextEdit` for editing. Use monospace font for better allignment.
2. Create the preview widget. Instantiate a second `QTextEdit` that will display the rendered Markdown. Make it read-only and give it a distinct appearance.
3. Implement the preview slot to and connect the signal. Get the source textedit contents as plain text and use `setMarkdown()` to render it as Markdown in the preview textedit. By default, `setMarkdown()` uses the GitHub-flavored Markdown dialect.

Note that we perform the initial `setPlainText()` only **after** connecting the `textChanged()` signal. This ensures the slot runs immediately and the preview is rendered with the initial text.



7.3 `QPlainTextEdit`

`QPlainTextEdit` provides a widget optimized for editing and displaying plain text. Unlike `QTextEdit`, it does not support HTML or Markdown.

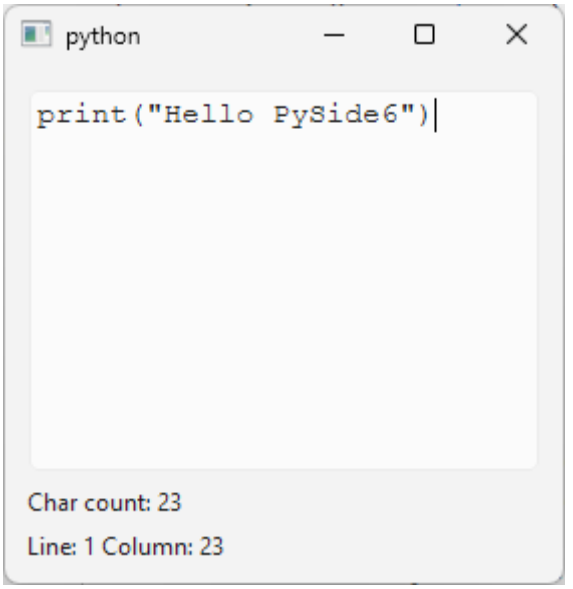
A `QPlainTextEdit` document is composed of characters and blocks (paragraphs) separated by newline characters. Each block contains a sequence of characters with optional formatting.

You are tasked with creating a basic plain-text editor that uses a monospace font and shows the current cursor line and column, as well as the total text length. To do this:

```
# The QPlainTextEdit class provides a widget
# that is used to edit and display plain text.

import sys
from PySide6.QtCore import Slot
from PySide6.QtGui import QFontDatabase
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QPlainTextEdit, QLabel)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create the plain text edit widget
        #     and set its font and wrap mode.

        self.editor = QPlainTextEdit()
        self.editor.setLineWrapMode(
            QPlainTextEdit.LineWrapMode.NoWrap)
        mono = QFontDatabase.systemFont(
            QFontDatabase.SystemFont.FixedFont)
        mono.setPointSize(11)
        self.editor.setFont(mono)

        # Labels displaying document stats.

        self.charcount_label = QLabel()
        self.position_label = QLabel()

        layout.addWidget(self.editor)
        layout.addWidget(self.charcount_label)
        layout.addWidget(self.position_label)

        # 3. Connect the signals to the slots
        self.editor.textChanged.connect(self.update_char_count)
        self.editor.cursorPositionChanged.connect(self.update_position)

        self.editor.setPlainText('print("Hello PySide6")')

    # 2. Get the underlying QTextDocument properties
    #    Character count is directly available.
    #    Cursor position needs to be calculated.

    @Slot()
```

```
51      def update_char_count(self):
52          char_count = self.editor.document().characterCount()
53          self.charcount_label.setText(f'Char count: {char_count}')
54
55      @Slot()
56      def update_position(self):
57
58          cursor = self.editor.textCursor()
59          x = str(cursor.block().blockNumber() + 1)
60          y = str(cursor.positionInBlock() + 1)
61
62          self.position_label.setText(f'Line: {x} Column: {y}')
63
64
65  if __name__ == '__main__':
66
67      app = QApplication(sys.argv)
68      main_window = Window()
69      main_window.show()
70      sys.exit(app.exec())
```

1. Create a `QPlainTextEdit` object, use `QFontDatabase` to set its font to a generic monospace font, and disable line wrapping. Also add two labels to the window: one for showing character count, and the other for showing cursor position.
2. Implement the slots to display text stats. In `update_char_count()`, retrieve the character count directly from the editor's underlying `QTextDocument` and display it in the label. In `update_position()`, use the editor's `QTextCursor` to get the current block number and the current cursor position within the block, then use these to calculate the cursor's current line (block number + 1) and column (position within the block + 1).
3. Connect the signals to the slots. Update the character count when `textChanged()` emits, update cursor position when `cursorPosition-Changed()` emits.



Qt Rich Text Processing Framework

`QTextDocument` is the central part of the Scribe framework[2] designed for representing and editing structured rich text documents. Representing is supported withe document elements such as `QTextBlock`, `QTextFrame`, `QTextTable` and `QTextList` and editing via the `QTextCursor` class. The framework also supports syntax highlighting.

[2]https://doc.qt.io/qtforpython-6/

8. List Widgets

Qt provides several widgets for displaying and managing lists of items, including:

- `QComboBox` - for showing a compact dropdown list,
- `QTableWidget`- for built-in list item management,
- `QTableView` - for model-based list mnagement.

8.1 `QComboBox`

`QComboBox` combines a button and a line edit with a pop-up list, which makes it useful for displaying list in a constrained space. It can be populated using its insert methods:

- `addItem()` - appends an item,
- `addItems()` - appends a list of items,
- `insertItem()` - insert an item at the given index,
- `insertItems()` - inserts a list of items at the given index,
- `removeItem()` - removes the item at the given index.

Combobox items can have text, icons, and additional user data. As `QCombobox` is a part of the Qt's model/view framework, you can provide data for it using one of the Qt model classes instead of managing items manually.

`QComboBox` provides several signals:

Signal	Description
`activated(index)`	User chooses an item from the list
`currentIndexChanged(index)`	Current index changes (user or programmatic)
`currentTextChanged(text)`	Current text changes
`editTextChanged(text)`	Text edited in editable combobox
`highlighted(index)`	Item in the popup list highlighted
`textActivated(text)`	User chooses an item
`textHighlighted(text)`	Item in the popup list highlighted

Combobox items are indexed, starting from zero. Its line edit widget can be accessed with `QComboBox.lineEdit()` and it can be made editable with `setEditable()`.

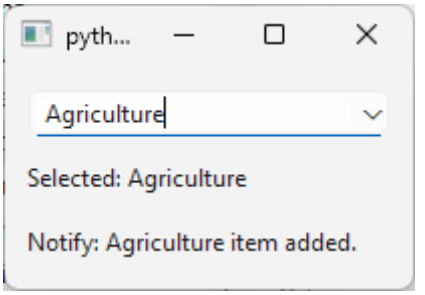
You need to enable the user to select an economic sector from the list and also add new sectors to the list. When a sector is selected, all other users need to be notified. To do this:

```
# The QComboBox widget is a combined button and popup list.

import sys
from PySide6.QtCore import Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QComboBox, QLabel)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create the combo box and add items to it
```

```
19
20          self.combo_box = QComboBox()
21
22          self.combo_box.addItems([
23              'Technology',
24              'Healthcare',
25              'Finance',
26              'Energy',
27              'Real Estate'])
28
29          # 2. Enable the user to add items
30
31          self.combo_box.model().sort(0)
32          self.combo_box.setCurrentIndex(1)
33          self.combo_box.setEditable(True)
34          self.combo_box.setInsertPolicy(
35              QComboBox.InsertPolicy.InsertAlphabetically)
36
37          self.label_activated = QLabel()
38
39          layout.addWidget(self.combo_box)
40          layout.addWidget(self.label_activated)
41
42          self.combo_box.activated.connect(self.on_activated)
43
44      # 3. Create the slot
45
46      @Slot(int)
47      def on_activated(self, index):
48          item_text = self.combo_box.currentText()
49          self.label_activated.setText(f'Selected: {item_text}')
50
51
52  if __name__ == '__main__':
53
54      app = QApplication(sys.argv)
55      main_window = Window()
56      main_window.show()
57      sys.exit(app.exec())
```

1. Create a `QComboBox` object and add items to it. Sort the combobox items and set the first item as the current item.
2. Enable the user to add items. `QComboBox`es are read-only by default so set the `editable()` property to `True`. Set the added items to be inserted alphabetically.
3. Create a slot to notify users when an item is selected. Connect to the list widget's `activated()` signal. When the user selects an item or adds a

new one to the list, the slot is executed.



8.2 `QListWidget`

The `QlistWidget` class is an item-based list widget. This means its items are instances of the `QListWidgetItem` class, and you add them to the list widget using methods such as:

- `addItem(item)`,
- `addItem(label)` or
- `addItems(labels)`.

`QListWidget` also provides methods to insert items at specific positions, retrieve an item at a given row, and remove an item using the `takeItem(row)` method.

`QListWidgetItem`, the accompanying class, can hold several pieces of information, such as text, icons, or tooltips. It can also store custom user-defined data.

Common `QListWidgets` include:

Signal	Description
`currentItemChanged(current, previous)`	Emitted when the current item changes.
`currentRowChanged(currentRow)`	Emitted when the current row changes.
`currentTextChanged(currentText)`	Emitted when the text of the current item changes.
`itemActivated(item)`	Emitted when an item is activated (double-click or Enter/Space while selected).
`itemChanged(item)`	Emitted when an item's data is modified by the user.
`itemClicked(item)`	Emitted on any mouse click on an item.
`itemDoubleClicked(item)`	Emitted specifically on double-click.
`itemSelectionChanged()`	Emitted when the selection changes.

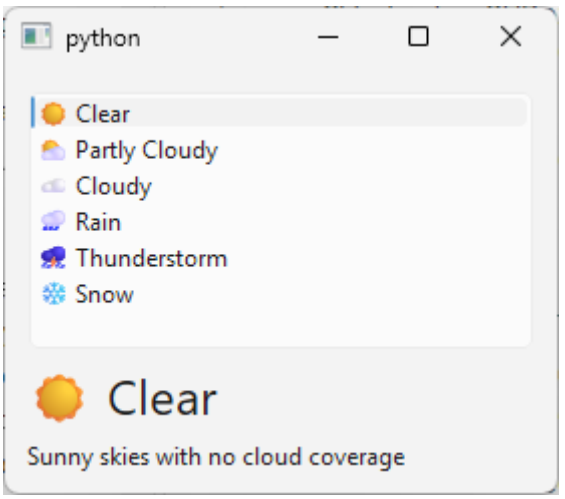
You need to create a list of common weather conditions. When the user selects one, you need provide additional information about it. To do this:

```
# The QListWidget class provides
# an item-based list widget

import sys
from PySide6.QtCore import Slot, Qt
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QListWidget, QLabel,
    QListWidgetItem)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
```

```
        self.setLayout(layout)

        weather_conditions = [
            ('☀️ Clear', 'Sunny skies with no cloud coverage'),
            ('⛅ Partly Cloudy', 'Sun and clouds throughout the day'),
            ('☁️ Cloudy', 'Overcast skies with full cloud coverage'),
            ('🌧️ Rain', 'Precipitation with steady rainfall'),
            ('⛈️ Thunderstorm', 'Heavy rain with lightning and thunder'),
            ('❄️ Snow', 'Frozen precipitation and cold temperatures')
        ]

        # 1. Create a list widget and add items to it.

        self.weather_list = QListWidget()

        for name, description in weather_conditions:
            item = QListWidgetItem(name)
            item.setData(Qt.ItemDataRole.UserRole, description)
            self.weather_list.addItem(item)

        self.selected_label = QLabel()
        self.description_label = QLabel()
        self.selected_label.setStyleSheet("font-size: 24px;")

        layout.addWidget(self.weather_list)
        layout.addWidget(self.selected_label)
        layout.addWidget(self.description_label)

        # 3. Connect the signal to the slot.

        self.weather_list.currentItemChanged.connect(
            self.select_weather)
        self.weather_list.setCurrentRow(0)

    # 3. Create the slot.

    @Slot(QListWidgetItem, QListWidgetItem)
    def select_weather(self, current, previous):

        weather = current.data(Qt.ItemDataRole.DisplayRole)
        description = current.data(Qt.ItemDataRole.UserRole)
        self.selected_label.setText(weather)
        self.description_label.setText(description)


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
```

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

1. Create a `QListWidget` object and add items to it. After creating the list widget, create a Python list of tuples, where each tuple's first element contains the weather condition name and the second element contains its description. Iterate over the list, and for each tuple, create a `QListWidget` object, setting the weather condition description as custom user data. Add each widget item to the list widget. Also create two labels for displaying the information.
2. Implement a slot to display the weather condition name and description when the current item is changed. You get the name of a weather condition using the `DisplayRole` item data role, and the description using the `UserRole` item data role.
3. Connect the signal to the slot. The `currentItemChanged()` signal is emitted whenever the current item changes. It provides both the current and the previous `QListWidgetItem` objects to the slot (though we need only the current item in this example).



8.3 `QListView`

A `QListView` presents items stored in a model, either as a list or a collection of icons. It is part of Qt's model/view framework, which separates data (models)

from its visual representation (views). This allows one model to be shared across multiple views.

Provide a read-only list view of the user's home directory, decorating each item with an appropriate icon. When the user hovers over a file, display a tooltip showing its size in KB. To do this:

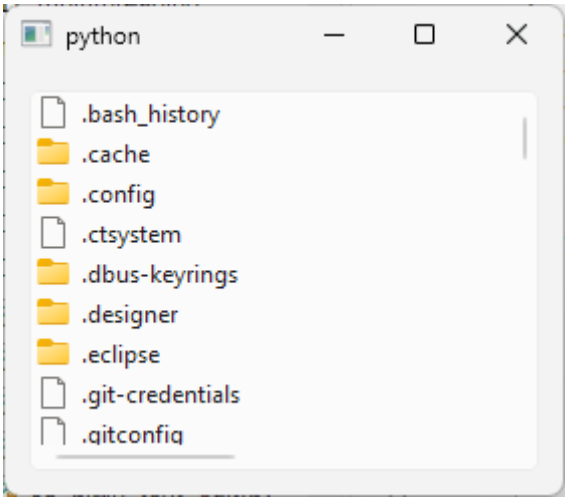
```
# QListView presents items stored in a model.
# ie. it uses the QT model/view architecture
# Models contain data and views display it
# so one model can be used with multiple views.

import sys
from pathlib import Path
from PySide6.QtCore import Qt
from PySide6.QtGui import QStandardItemModel, QStandardItem
from PySide6.QtWidgets import (QApplication, QWidget,
    QVBoxLayout, QListView, QAbstractItemView, QStyle)


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create the view

        self.list_view = QListView()
        self.list_view.setEditTriggers(
            QAbstractItemView.EditTrigger.NoEditTriggers)

        # 2. Create the model and populate it with data

        self.model = QStandardItemModel()

        home = Path.home()
        fs_entries = home.iterdir()

        for entry in fs_entries:
            item = QStandardItem(entry.name)
            icon = self.get_icon(entry)
            item.setIcon(icon)
            tooltip = self.get_tooltip_data(entry)
```

```
            item.setData(tooltip, Qt.ItemDataRole.ToolTipRole)
            self.model.appendRow(item)

        # 3. Set the view's model

        self.list_view.setModel(self.model)

        layout.addWidget(self.list_view)

    def get_icon(self, path):
        if path.is_dir():
            return self.style().standardIcon(
                QStyle.StandardPixmap.SP_DirIcon)
        else:
            return self.style().standardIcon(
                QStyle.StandardPixmap.SP_FileIcon)

    def get_tooltip_data(self, path):
        if path.is_dir():
            return 'Directory'
        elif path.is_file():
            size = round(path.stat().st_size / 1024, 2)
            return f'File: {size} KB'
        else:
            return 'File System Entry'


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Create the view. Instantiate a `QListView` object and disable its edit triggers using `NoEditTriggers` - this prevents actions like double-click editing.
2. Create the model and populate it with data. Use a `QStandardItemModel` object to store filesystem data. For each filesystem entry in the user's home directory create a `QStandardItem` object, initialize it with the entry name, and set its icon based on the type (file or directory). If the entry is a file, set its tooltip text to the file size in a human-readable format. Finally, append the item to the model.
3. Set the view's model to the `QStandardItemModel` object.

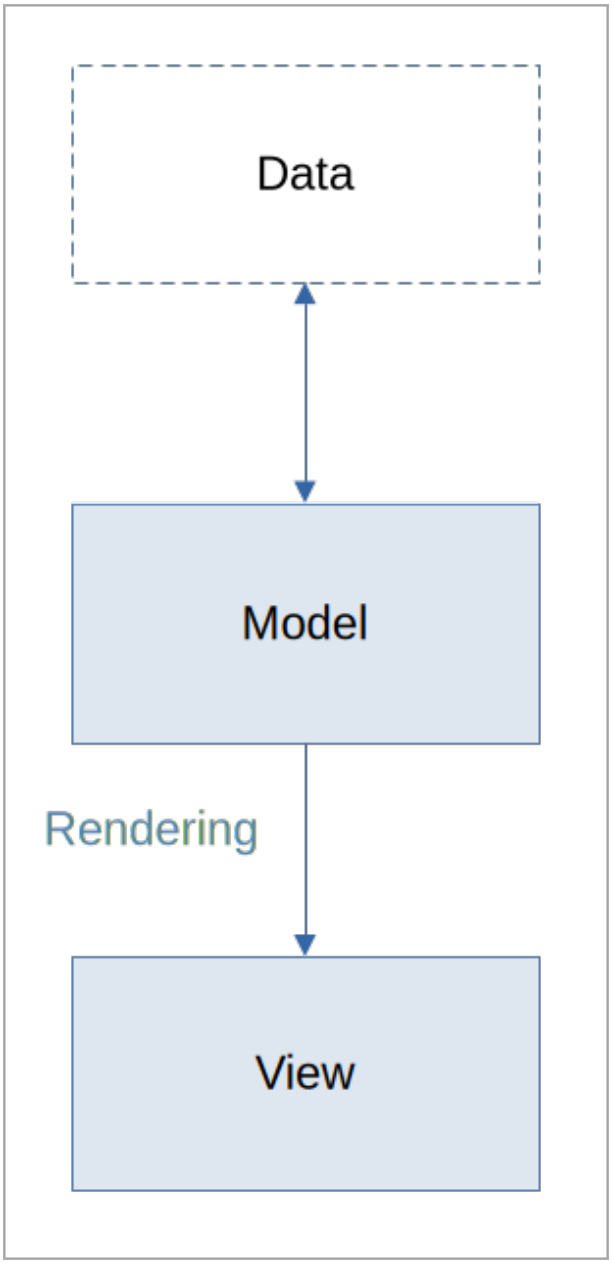
We use Python's `pathlib.Path` to access filesystem data for the model. In

the `get_icon()` method, we return the Qt's standard icons for files and directories to avoid using external resources. When initializing a `QStandardItem` with a filesystem entry name, the name is assigned to the `DisplayRole` by default. To assign data to its tooltip, use the `ToolTipRole` when setting the data.



Qt's model/view framework

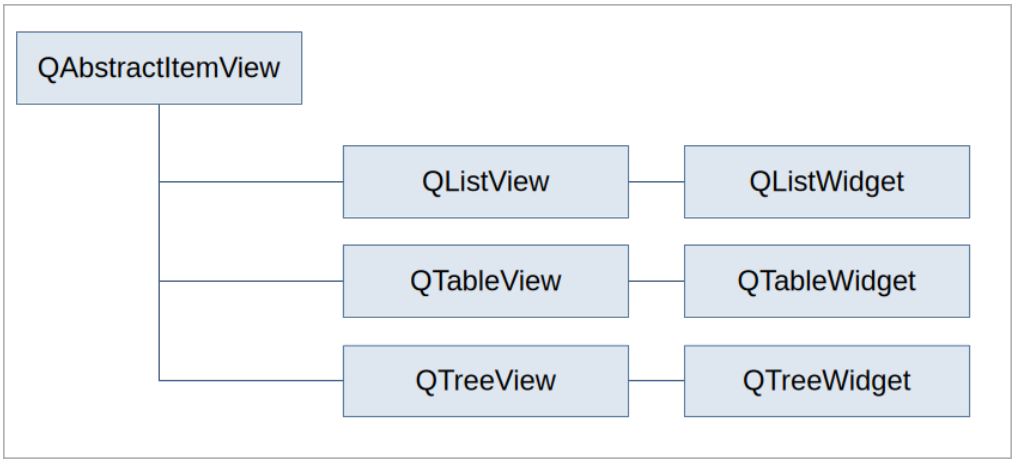
Qt's model/view framework decouples data from its GUI representation using standardized interfaces. Models, inheriting from `QAbstractItemModel`,communicate with data sources and implement a set of methods for its manipulation. Views use these methods to display the data, enabling one model to support multiple views.



List classes inheritance

The `QListWidget` class is a convenience subclass of `QListView` that pro-

vides an item-based interface for adding and managing list items.Similarly, `QTableWidget` and `QTreeWidget` are convenience subclasses of `QTableView` and `QTreeView`, respectively. These widget classes are designed for ease of use, while the view classes (`QListView`, `QTableView`, `QTreeView`) are intended for use with the Model/View framework when greater flexibility is needed.



9. Table Widgets

Table widgets in Qt provide a structured way do display and interact with tabular data.

- `QTableWidget` is a self-contained widget that includes its own data storage, allowing developers to easily manipulate items directly within the widget.
- `QTableView` is part of the Qt model-view framework, requiring an external model to supply data, promoting separation of concerns and reusability in larger applications.

Both support sorting, resizing columns and rows, and selecting multiple cells. These widgets are commonly used in applications requiring grid-based input or visualization, such as data entry forms or result displays.

9.1 `QTableWidget`

[TODO: INHERITANCE TREE]

The `QTableWidget` class provides an item-based table view with a default model. Simply put, it's a table. The items in a `QTableWidget` are provided by `QTableWidgetItem` objects.

You need to enable your boss to create and edit a service comparison matrix for your company.

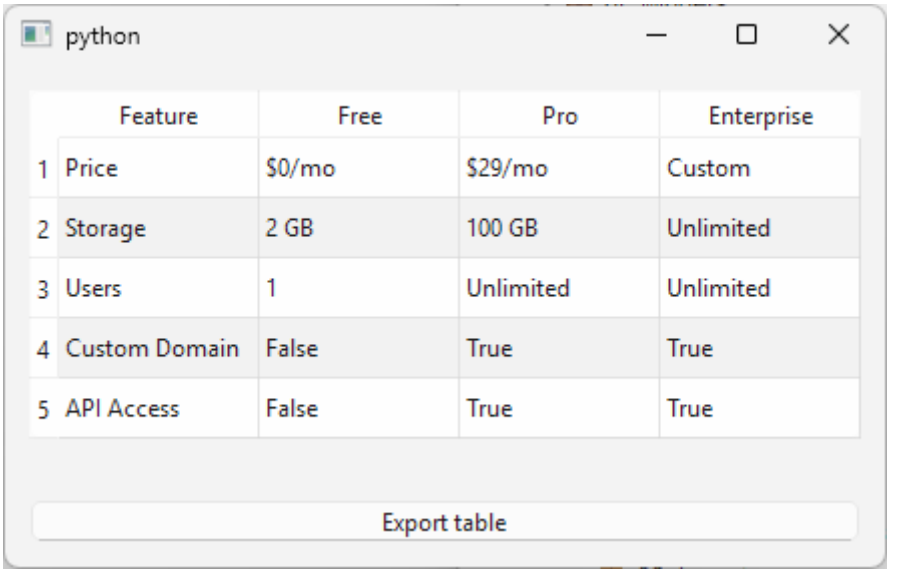
To use a table widget in your application:

```python
# The QTableWidget class provides an
# item-based table view with a default model.

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


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.headers = ["Feature", "Free", "Pro", "Enterprise"]

        self.data = [
            ["Price", "$0/mo", "$29/mo", "Custom"],
            ["Storage", "2 GB", "100 GB", "Unlimited"],
            ["Users", "1", "Unlimited", "Unlimited"],
            ["Custom Domain", False, True, True],
            ["API Access", False, True, True]
        ]

        rows = len(self.data)
        columns = len(self.data[0])

        # 1. Create a table widget.

        self.table_widget = QTableWidget(rows, columns)
        self.table_widget.setHorizontalHeaderLabels(self.headers)
        self.table_widget.setAlternatingRowColors(True)
        self.table_widget.setSizeAdjustPolicy(
            QTableWidget.SizeAdjustPolicy.AdjustToContents)

        # 2. Populate the widget.

        for row in range(rows):
            for col in range(columns):
                item = QTableWidgetItem()
                item.setData(
                    Qt.ItemDataRole.DisplayRole,
                    str(self.data[row][col]))
                self.table_widget.setItem(row, col, item)

        self.table_widget.itemChanged.connect(self.on_item_changed)
```

```
51
52          self.label = QLabel()
53          self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
54
55          self.export_button = QPushButton('Export table')
56          self.export_button.clicked.connect(self.export_data)
57
58          layout.addWidget(self.table_widget)
59          layout.addWidget(self.label)
60          layout.addWidget(self.export_button)
61
62      # 3. Create the required slots.
63
64      @Slot(QTableWidgetItem)
65      def on_item_changed(self, item):
66          row = item.row()
67          col = item.column()
68          text = item.text()
69          self.label.setText(
70              f'Cell[{row}, {col}]: new value is {text}')
71
72      @Slot()
73      def export_data(self):
74          print(', '.join(self.headers))
75          for r in range(self.table_widget.rowCount()):
76              row_data = []
77              for c in range(self.table_widget.columnCount()):
78                  row_data.append(self.table_widget.item(r, c).text())
79              print(', '.join(row_data))
80
81
82  if __name__ == '__main__':
83
84      app = QApplication(sys.argv)
85      main_window = Window()
86      main_window.show()
87      sys.exit(app.exec())
```

1. Create a `QTableWidget` instance and set the table dimensions. The table in the example has five rows and four columns. We also add a label to the window and a push button for exporting the comparison matrix.
2. Fill the table cells with data. Each table cell is represented with a `QTableWidgetItem` instance so we use a nested loop to create the items. We set each item's data from a two-dimensional list using `QTableWidgetItem.setData()`. After that, each `QTableWidgetItem` is assigned to a table cell using `QTableWidget.setItem(row, column, item)`.

3. Create two slot methods: one connected to `QTableWidget`'s `itemChanged()` signal and the other connected to the push button's `clicked()` signal. The first slot updates the label text when a table item is changed, and the second one 'exports' the matrix by printing it. The table is editable; if the user double-clicks a cell, changes its value, and presses Enter, the label's text is updated accordingly.



A key distinction between `QTableWidget` and `QTableView` is in their handling of data. `QTableWidget` is well-suited for semi-structured or heterogeneous data, where individual cells can vary in type or format without column consistency - for instance, allowing a user to enter 'Maybe' in a boolean-like field instead of strictly `True` or `False`. In contrast, `QTableView`, paired with a model, is suitable for use with structured data, as model typically enforce uniform data types and validation rules per column. This flexibility makes `QTableWidget` ideal for quick prototypes and ad-hoc tables, while `QTableView` supports more scalable, data-driven applications.

9.2 `QTableView`

[TODO: INHERITANCE TREE]

`QTableView` is the default model-view class for displaying tabular data, and just as `QListWidget` is a subclass of `QListView`, `QTableWidget` is a subclass of `QTableView`.

You users need to edit a table of aggregate economic indicators and select which ones to include in the report.

To use `QTableView` in your application:

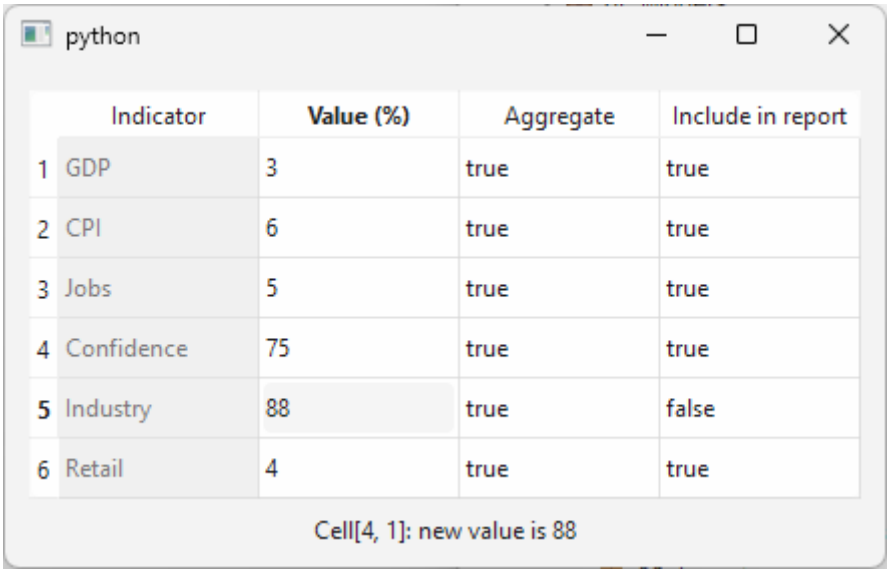
```
# QTableView implements a table view
# that displays items from a model.

import sys
from PySide6.QtCore import Slot, Qt
from PySide6.QtGui import QStandardItemModel, QStandardItem
from PySide6.QtWidgets import (QApplication, QWidget,
    QVBoxLayout, QTableView, QLabel)


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.header = ['Indicator', 'Value (%)',
            'Aggregate', 'Include in report']
        self.data = [
            ['GDP', 3, True, True],
            ['CPI', 6, True, True],
            ['Jobs', 5, True, True],
            ['Confidence', 75, True, True],
            ['Industry', 92, True, True],
            ['Retail', 4, True, True],
        ]

        # 1. Create the view

        self.table_view = QTableView()
        self.table_view.setSizeAdjustPolicy(
            QTableView.SizeAdjustPolicy.AdjustToContents)

        rows = len(self.data)
        columns = len(self.data[0])
```

```
39
40          # 2. Create the model.
41
42          self.model = QStandardItemModel(rows, columns)
43          self.model.setHorizontalHeaderLabels(self.header)
44
45          for row in range(rows):
46              for col in range(columns):
47                  item = QStandardItem()
48                  if col == 0:
49                      item.setFlags(Qt.ItemFlag.NoItemFlags)
50                  data = self.data[row][col]
51                  item.setData(data, Qt.ItemDataRole.DisplayRole)
52                  self.model.setItem(row, col, item)
53
54          # 3. Connect the model with the view.
55
56          self.table_view.setModel(self.model)
57
58          self.notify_label = QLabel()
59          self.notify_label.setAlignment(Qt.AlignmentFlag.AlignCenter)
60          self.model.itemChanged.connect(self.notify_users)
61
62          layout.addWidget(self.table_view)
63          layout.addWidget(self.notify_label)
64
65      @Slot(QStandardItem)
66      def notify_users(self, item):
67          row = item.row()
68          col = item.column()
69          text = item.text()
70          self.notify_label.setText(
71              f'Cell[{row}, {col}]: new value is {text}')
72
73
74  if __name__ == '__main__':
75
76      app = QApplication(sys.argv)
77      main_window = Window()
78      main_window.show()
79      sys.exit(app.exec())
```

1. Create a `QTableView` instance. We also add a label to the window to notify users of changes.
2. Create a model instance. In the example, we use `QStandardItemModel`, populating it with data from the two- dimensional `data` list. Each item is represented with a `QStandardItem` object. We set the data using

`setData()` for `DisplayRole` and adjust flags to make the first column read-only.

3. Set the view's model using `QTableView.setModel()`. Connect the model's `itemChanged()` signal to a slot to handle updates.



This separates the data model from the view, allowing the same model to be used with multiple views, and allowing a view's model to be changed dynamically.

10. Tree Widgets

11. Containers

Qt container widgets are visual elements that group other widgets together to form structured interfaces - they organize layout, provide borders, and create logical sections of the UI.

11.1 `QGroupBox`

`QGroupBox` is a widget that has a frame and a title on top and allows you to display other widgets inside itself but is commonly used to group checkboxes or radiobuttons. `QGroupBox` does not lay out its child widgets automatically - you need to do it yourself using one of the Qt layout classes. You can set a `QGroupBox` to be `checkable`, which lets the user enable or disable all its child widgets simultaneously.

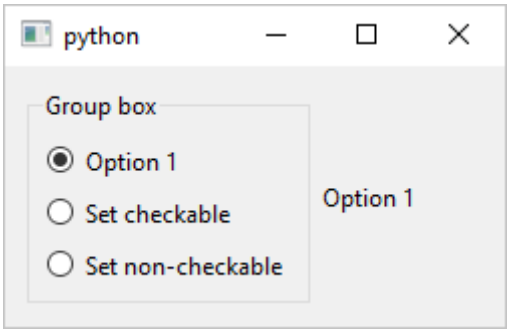
Your task is to create a group of mutually exclusive options. You also need to be able to enable or disable the whole group.

```
# A group box provides a frame, a title on top
# and displays various other widgets inside itself.

import sys

from PySide6.QtWidgets import (QApplication,
    QWidget, QHBoxLayout, QVBoxLayout,
    QGroupBox, QRadioButton, QLabel)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QHBoxLayout()

        self.label = QLabel()
```

```
20          self.label.setFixedWidth(80)
21
22          # 1 - Create the group box
23          #     and add a layout to it. You can't
24          #     add widgets directly to the group box.
25
26          self.groupbox = QGroupBox()
27          self.groupbox.setTitle('Group box')
28
29          groupbox_layout = QVBoxLayout()
30          self.groupbox.setLayout(groupbox_layout)
31
32          # 2 - Add widgets to the layout.
33
34          self.radiobutton_1 = QRadioButton('Option 1')
35          self.radiobutton_2 = QRadioButton('Set checkable')
36          self.radiobutton_3 = QRadioButton('Set non-checkable')
37
38          groupbox_layout.addWidget(self.radiobutton_1)
39          groupbox_layout.addWidget(self.radiobutton_2)
40          groupbox_layout.addWidget(self.radiobutton_3)
41
42          # 4- Connect child widget signals with the slot
43
44          self.radiobutton_1.toggled.connect(self.on_toggled)
45          self.radiobutton_2.toggled.connect(self.on_toggled)
46          self.radiobutton_3.toggled.connect(self.on_toggled)
47
48          self.radiobutton_1.setChecked(True)
49
50          layout.addWidget(self.groupbox)
51          layout.addWidget(self.label)
52
53          self.setLayout(layout)
54
55      # 3 - Add slot method. If there are
56      #     multiple radio buttons in a group box
57      #     only one can be checked, unlike checkboxes.
58
59      def on_toggled(self):
60
61          if self.radiobutton_1.isChecked():
62              self.label.setText(self.radiobutton_1.text())
63          elif self.radiobutton_2.isChecked():
64              self.label.setText(self.radiobutton_2.text())
65              self.groupbox.setCheckable(True)
66          else:
67              self.label.setText(self.radiobutton_3.text())
68              self.groupbox.setCheckable(False)
69
```

```
70
71  if __name__ == '__main__':
72
73      app = QApplication(sys.argv)
74
75      main_window = Window()
76      main_window.show()
77
78      sys.exit(app.exec())
```

1. Create a `QGroupBox` object and add a layout to it as you can't add widgets directly. In the example, we use a `QVBoxLayout`.
2. Create widgets and add them to the layout. In the example, we add three `QRadioButtons`. We also set one of the radio buttons to `checked`.
3. Create the slot method to handle `QRadioButton.toggled()` signals. The slot sets a label's text to the text of the checked radiobutton and the last two radiobuttons also toggle the group box `togglable` property.
4. Connect `QRadioButton.toggled` signals with the slot.



11.2 `QScrollArea`

If a `QScrollArea`' child widget exceeds its size it provides scrollbars so that the whole child widget can be viewed.

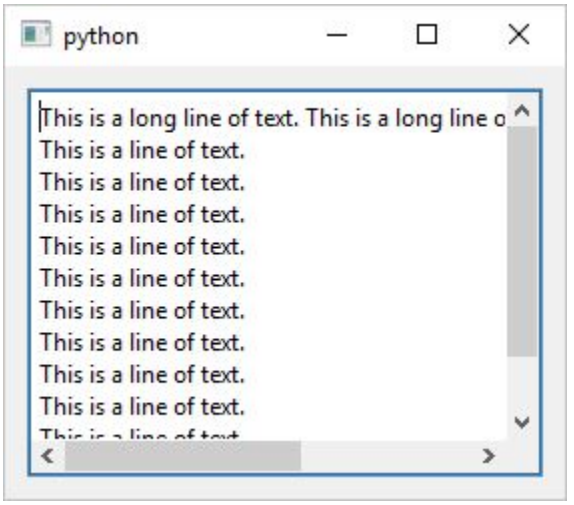
You need to create a simple text editor in a constrained space.

```
# The QScrollArea class provides a
# scrolling view onto another widget

import sys

from PySide6.QtGui import QTextOption
from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QScrollArea, QPlainTextEdit)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the scroll area

        scroll_area = QScrollArea()

        # 2 - Create the widget that needs to be scrolled

        text_edit = QPlainTextEdit()
        text_edit.setWordWrapMode(QTextOption.WrapMode.NoWrap)

        # 3 - Add the widget to the scroll area

        scroll_area.setWidget(text_edit)

        # Use this if you want the inner widget to
        # get resized together with the scroll area.

        scroll_area.setWidgetResizable(True)

        layout.addWidget(scroll_area)


if __name__ == '__main__':

    app = QApplication(sys.argv)

    main_window = Window()
    main_window.show()

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

1. Create a `QScrollArea` object

2. Create the child widget. In the example we use a `QPlainTextEdit` object and set its wrap mode to `WrapMode.NoWrap` so that it expands when text is entered
3. Add the `QPlainTextEdit` to the scroll area using `QScrollArea.addWidget()`. We also set the `QScrollArea.widgetResizable` property to `True` so that the text box resizes along with the scroll area.

Now if you enter a long line of text in the text box the scroll area shows the horizontal scrollbar and if you enter several lines the scroll area shows the vertical scrollbar.



11.3 `QToolBox`

`QToolBox` provides a column of tabbed widget items. This doesn't really tell you much - it is a Qt container widget pretty similar to the ubiquitous accordion widget that lets you pack multiple widgets within a relatively small space and expand or collapse them as needed. `QToolBox` pages are called items in the documentation.

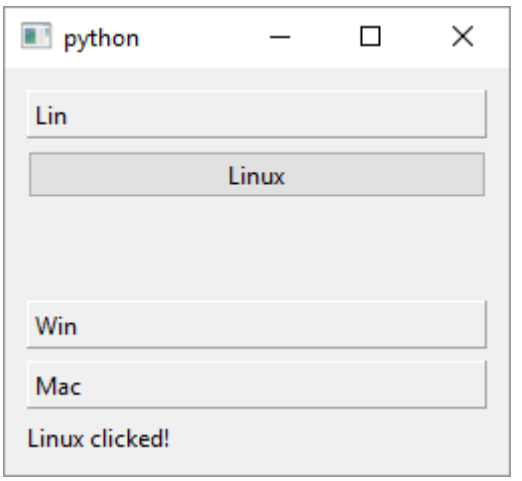
You need to add a collapsible set of options within a small space.

```python
# The QToolBox class provides a column of tabbed widget items

import sys

from PySide6.QtWidgets import (QApplication,
    QWidget, QVBoxLayout, QToolBox, QPushButton, QLabel)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the toolbox

        toolbox = QToolBox()

        # 2 - Create the widgets (buttons in this case)

        windows_button = QPushButton('Windows')
        mac_button = QPushButton('Mac')
        linux_widget = QWidget()

        linux_widget.setLayout(QVBoxLayout())
        debian_button = QPushButton('Debian')
        arch_button = QPushButton('Arch')
        linux_widget.layout().addWidget(debian_button)
        linux_widget.layout().addWidget(arch_button)

        # 3 - Add widgets to the toolbox

        toolbox.addItem(windows_button, 'Win')
        toolbox.addItem(mac_button, 'Mac')
        toolbox.addItem(linux_widget, 'Lin')

        windows_button.clicked.connect(self.on_clicked)
        mac_button.clicked.connect(self.on_clicked)
        debian_button.clicked.connect(self.on_clicked)
        arch_button.clicked.connect(self.on_clicked)

        self.label = QLabel()

        layout.addWidget(toolbox)
        layout.addWidget(self.label)

    def on_clicked(self):
```

```
51          self.label.setText(self.sender().text() + ' clicked!')
52
53
54  if __name__ == '__main__':
55
56      app = QApplication(sys.argv)
57
58      main_window = Window()
59      main_window.show()
60
61      sys.exit(app.exec())
```

1. Create a `QToolBox` object
2. Create the child widgets. In the example we simply create three push buttons each representing a popular operating system. You can also add multiple child widgets to a page by adding them to a layout as demonstrated on the 'Linux' page.
3. Add the widgets to the toolbox using `QToolBox.addItem()`

In the example we also handle the child push buttons `clicked` signals for demonstration purposes, setting a label's text to the text of the clicked button.



11.4 `QTabWidget`

`QTabWidget` is a tabbed container widget - when you click on a tab its associated page is shown.

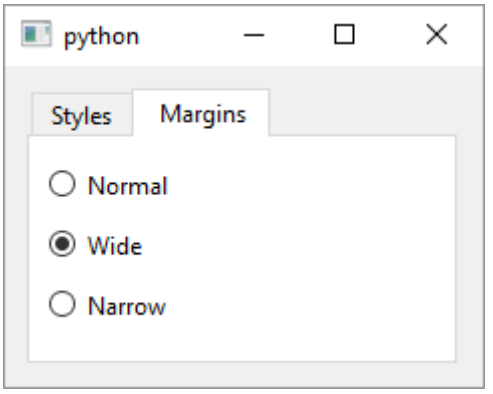
Your space is limited and you need to create options for your editor styles and margins.

```
# The QTabWidget class provides a stack of tabbed widgets.  :S
# ie. tabs

import sys

from PySide6.QtWidgets import (QApplication, QWidget,
    QVBoxLayout, QTabWidget, QRadioButton, QCheckBox)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create the tab widget

        self.tab_widget = QTabWidget()

        # 2 - Create some widgets
        #     A tab contains only one widget
        #     but that widget can be a QWidget instance
        #     or some other container.

        # Tab 0 widgets:

        styles = QWidget()
        styles_layout = QVBoxLayout()
        styles_layout.addWidget(QCheckBox('Heading'))
        styles_layout.addWidget(QCheckBox('Paragraph'))
        styles_layout.addWidget(QCheckBox('List'))
        styles.setLayout(styles_layout)

        # Tab 1 widgets

        margins = QWidget()
        margins_layout = QVBoxLayout()
        margins_layout.addWidget(QRadioButton('Normal'))
        margins_layout.addWidget(QRadioButton('Wide'))
        margins_layout.addWidget(QRadioButton('Narrow'))
        margins.setLayout(margins_layout)

```

```
46          # 3 - Add tabs to the widget
47
48          self.tab_widget.addTab(styles, 'Styles')
49          self.tab_widget.addTab(margins, 'Margins')
50
51          layout.addWidget(self.tab_widget)
52
53
54  if __name__ == '__main__':
55
56      app = QApplication(sys.argv)
57
58      main_window = Window()
59      main_window.show()
60
61      sys.exit(app.exec())
```

1. Create a `QTabWidget` object.
2. Create its intended child widgets. Each page contains a single widget but that widget, in turn, can be a `QWidget` or a container class which allows you to pack multiple children into a `QTabWidget` page.
3. Use `QTabWidget.addTab()` to add the child widgets to the tab widget object. In the example we have two `QWidgets` and add each to the tab widget. The first `QWidget` has three check boxes as its children and the second `QWidgets` children are three radio buttons.

The tab indexes start with zero so the first tab has the index of `0` and the second has the index of `1`. Each tab has a label (`Styles` and `Margins` in the example). You can change the tab position (`North`, `South`, `West` and `East`) and shape (`Rounded` and `Triangular`).



11.5 QSplitter

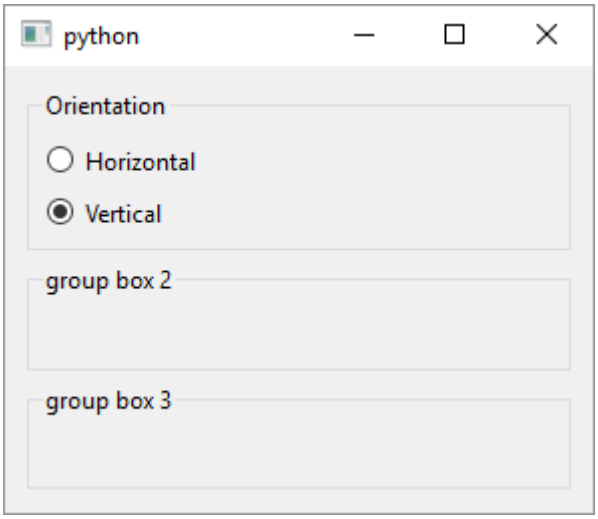
The `QSplitter` class lets the user resize its child widgets using the mouse.

Your task is to create an application with three resizable panes and you want the user to be able to toggle their orientation.

```
# The QSplitter class implements a splitter widget.
# A splitter lets the user control the size of
# child widgets by dragging the boundary between them.
# Any number of widgets may be controlled by a single splitter.

import sys

from PySide6.QtCore import Qt
from PySide6.QtWidgets import (QApplication, QWidget,
    QVBoxLayout, QSplitter, QGroupBox, QRadioButton)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1 - Create a splitter

        self.splitter = QSplitter()

        # 2 - Create widgets
        #     In this case it's three groupboxes

        groupbox_1 = QGroupBox('Orientation')
        groupbox_1.setLayout(QVBoxLayout())

        # The radio buttons are just to demonstrate
        # splitter orientation. I don't think changing
        # splitter orientation at run time is that common.

        self.button_horizontal = QRadioButton('Horizontal')
        self.button_horizontal.setChecked(True)
        self.button_vertical = QRadioButton('Vertical')
```

```
39
40          self.button_horizontal.toggled.connect(self.on_toggled)
41          self.button_vertical.toggled.connect(self.on_toggled)
42
43          groupbox_1.layout().addWidget(self.button_horizontal)
44          groupbox_1.layout().addWidget(self.button_vertical)
45
46          groupbox_2 = QGroupBox('group box 2')
47          groupbox_3 = QGroupBox('group box 3')
48
49          # 3 - Add widgets to the splitter
50
51          self.splitter.addWidget(groupbox_1)
52          self.splitter.addWidget(groupbox_2)
53          self.splitter.addWidget(groupbox_3)
54
55          layout.addWidget(self.splitter)
56
57      def on_toggled(self):
58
59          if self.button_horizontal.isChecked():
60              self.splitter.setOrientation(
61                  Qt.Orientation.Horizontal)
62          else:
63              self.splitter.setOrientation(
64                  Qt.Orientation.Vertical)
65
66
67  if __name__ == '__main__':
68
69      app = QApplication(sys.argv)
70
71      main_window = Window()
72      main_window.show()
73
74      sys.exit(app.exec())
```

1. Create the `QSplitter` object. It lays its child widgets horizontally by default but you can change that using the `QSplitter.setOrientation()` method.
2. Create the child widgets. In the example we create three `QGroupBox` objects. We also add two radio buttons to the first group box - selecting the buttons changes the `QSplitter` orientation dynamically.
3. Add the child widgets to the splitter.


python
Orientation
Horizontal
Vertical
group box 2
group box 3

12 QMainWindow

13. Dialogs

14. Complex Widgets

15. Further Topics

16. Further Topics

17. Object Trees and Ownership

`QObject`s are organized in trees: when you create an object with another object as a parent, the child is added to the parent's `children()` list and automatically deleted when the parent is[1]. There are several ways to set an object's parent:

- Creating an object by passing the parent to its `__init__()` method.
- Using `QObject.setParent()`. This methods allows changing an object's parent after creation.
- Adding a widget to a layout. When adding a widget to a `QLayout`, the layout automatically reparents it to the layout's owning widget
- Setting a layout on a widget. Assigning a layout using `setLayout()` makes the widget the parent of any widgets in the layout.

Note that Python objects also form tree hierarchies, but a Python parent-child relationship doesn't imply Qt a Qt one.

17.1 Parent-Child Relationships

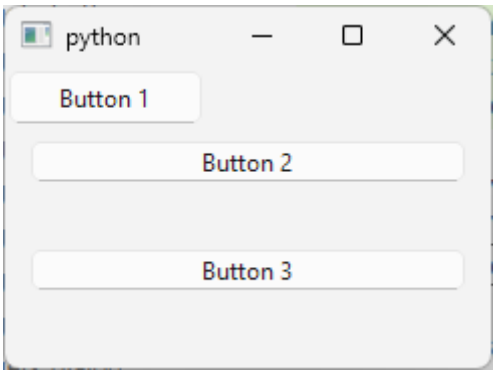
You are tasked with creating five Qt push buttons and displaying them in a window.

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)

        self.setObjectName('Main Window')
        self.setMinimumHeight(150)

        layout = QVBoxLayout()
        self.setLayout(layout)

        button1 = QPushButton('Button 1', self)
        button2 = QPushButton('Button 2')
        button3 = QPushButton('Button 3', self)
        button4 = QPushButton('Button 4')
        self.button5 = QPushButton('Button 5')

        button3.clicked.connect(self.on_button3_clicked)

        layout.addWidget(button2)
        layout.addWidget(button3)

        print('Button 1 parent: ', button1.parent().objectName())
        print('Button 2 parent: ', button2.parent().objectName())
        print('Button 3 parent: ', button3.parent().objectName())
        print('Button 4 parent: ', button4.parent())
        print('Button 5 parent: ', self.button5.parent())

    def on_button3_clicked(self):
        print('\nWindow members:')
        for member in dir(self):
            if member.startswith('button'):
                print(member)


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

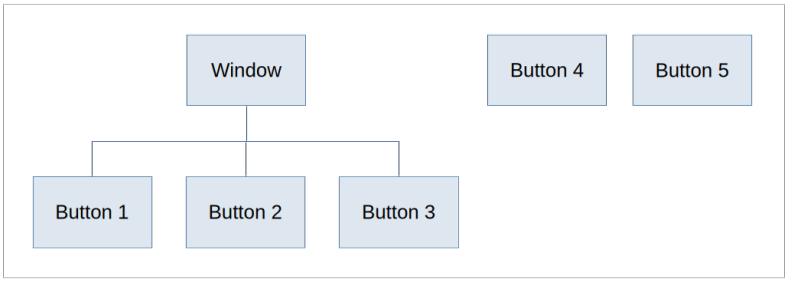
You create five `QPushButton` instances in the window's `__init__()` method:

1. Create the first button and set the window as its parent. Omit adding it to the layout or as an instance member, leaving it as a local variable.
2. Create the second button without setting a parent. Add it to the layout (which reparents it to the window), but keep it as a local variable.
3. Create the third button, setting the window as its parent. Add it to the layout, but as local variable.
4. Create the fourth button as a local variable without setting a parent.
5. Create the fifth button as an instance member of the window class but without setting a Qt parent.



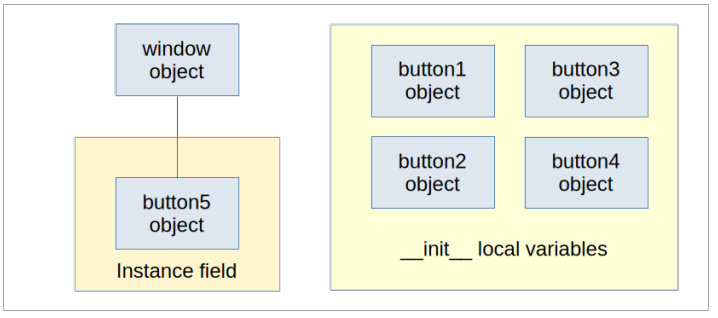
The program output is:

```
Button 1 parent:  Main Window
Button 2 parent:  Main Window
Button 3 parent:  Main Window
Button 4 parent:  None
Button 5 parent:  None

Window members:
button5
```



The first three buttons become the windows child widgets - directly or via the layout - and appear on the screen. The first button displays in the top-left corner since it is not in the layout. None of these three buttons are instance members of the `Window` Python object.

The fourth button is neither part of the Qt object hierarchy nor a Python object member, so it goes out of scope (and is garbage-collected) when `__init__()` returns, and is never shown.



The fifth button is an instance member of the window object (preventing garbage collection) but not part of the Qt object hierarchy, so it is not shown in the window.

17.2 Reparenting Qt Objects

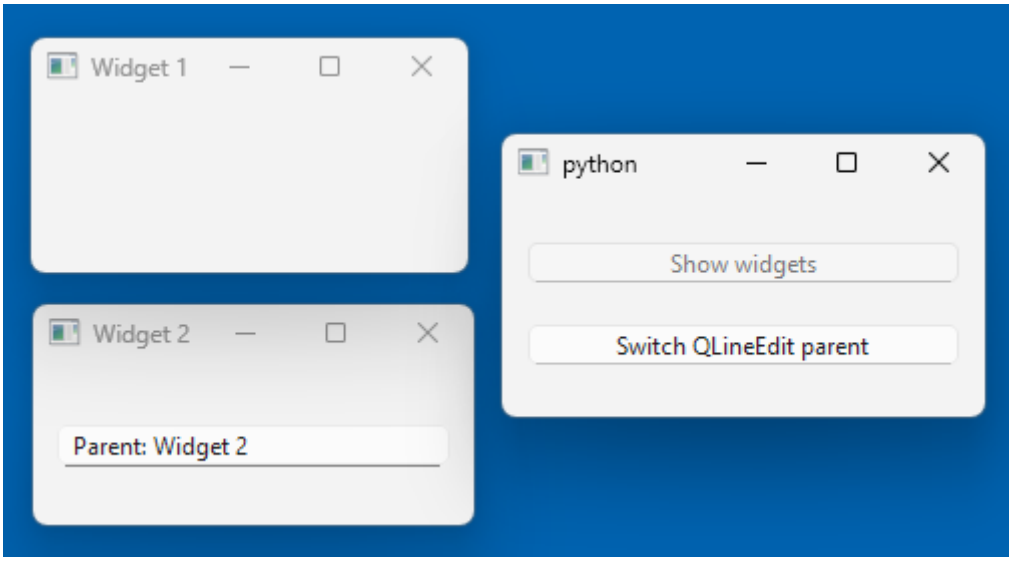
You need to create two top-level Qt windows and switch a child widget on demand.

To do this:

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QLineEdit, QPushButton, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.show_button = QPushButton('Show widgets')
        self.show_button.clicked.connect(self.show_widgets)
        layout.addWidget(self.show_button)

        self.switch_button = QPushButton('Switch QLineEdit parent')
        self.switch_button.clicked.connect(self.switch_parent)
        self.switch_button.setDisabled(True)
        layout.addWidget(self.switch_button)

        # 1. Create two top-level windows

        self.widget1 = QWidget()
        self.widget1.setObjectName('Widget 1')
        self.widget1.setLayout(QVBoxLayout())
        self.widget1.setWindowTitle('Widget 1')
        self.widget1.resize(200, 80)

        self.widget2 = QWidget()
        self.widget2.setObjectName('Widget 2')
        self.widget2.setLayout(QVBoxLayout())
        self.widget2.setWindowTitle('Widget 2')
        self.widget2.resize(200, 80)

        # 2. Create the child widget

```

```
40          self.edit = QLineEdit()
41          self.widget1.layout().addWidget(self.edit)
42
43      def show_widgets(self):
44          self.show_button.setDisabled(True)
45          self.switch_button.setEnabled(True)
46          self.widget1.show()
47          self.widget2.show()
48
49      # 3. Switch child widget parent on button click.
50
51      def switch_parent(self):
52
53          self.edit.parent().layout().removeWidget(self.edit)
54          if self.edit.parent() == self.widget1:
55              self.edit.setParent(self.widget2)
56          else:
57              self.edit.setParent(self.widget1)
58
59          self.edit.parent().layout().addWidget(self.edit)
60          self.edit.setText(f' Parent: {self.edit.parent().objectName()}')
61
62
63  if __name__ == '__main__':
64
65      app = QApplication(sys.argv)
66      main_window = Window()
67      main_window.show()
68      sys.exit(app.exec())
```

1. Create two top-level Qt Windows. On clicking the main window's 'Show widgets' button show two `QWidget` objects. Don't set their parents on creation - this makes them top-level windows (if you close the main window, they remain visible and active).
2. Create the child widget. Add a `QLineEdit` object to the layout of `widget1`.
3. On clicking the 'Switch parent' button reparent the line edit. Remove it from the current widget's layout, use `setParent()` to assign the other widget as its parent, and add it to its layout for visibility. The line edit displays its current parent's object name.



17.3 Finding Qt Object Children

Qt lets you find an object's children using these `QObject` methods:

- `findChild(type[, name[, options]])`: Finds a single matching child.
- `findChildren(type, pattern[, options])`: Finds multiple children by exact name.
- `findChildren(type[, name[, options]])`: Finds multiple children by regex pattern,

with these parameters:

- `type`: A PySide6 class such as `QWidget`, `QLabel`, or `QCheckBox`,
- `name`: The child's object name, set via `setObjectName()`. Defaults to an empty string so it must be set it explicitly for name-based searches (optional).
- `pattern`: A `QRegularExpression` instance for pattern matching.
- `options`: Either `Qt.FindChildOptions.FindDirectChildrenOnly` or `Qt.FindChildOptions.FindDirectChildrenRecursively` (optional).

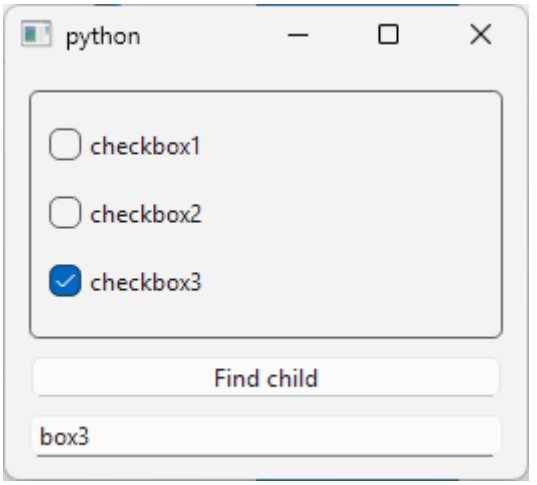
You have a set of checkboxes in a group box and need to check them programmatically based on user input.

To do that:

```
import sys
from PySide6.QtCore import QRegularExpression, Qt
from PySide6.QtWidgets import (QApplication, QWidget,
    QGroupBox, QCheckBox, QLineEdit, QPushButton, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        self.setObjectName('Main Window')

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.groupbox = QGroupBox()
        gb_layout = QVBoxLayout()
        self.groupbox.setLayout(gb_layout)
        layout.addWidget(self.groupbox)

        # 1. Create the checkboxes

        self.checkbox1 = QCheckBox('checkbox1')
        self.checkbox1.setObjectName('checkbox1')
        gb_layout.addWidget(self.checkbox1)

        self.checkbox2 = QCheckBox('checkbox2')
        self.checkbox2.setObjectName('checkbox2')
        gb_layout.addWidget(self.checkbox2)

        self.checkbox3 = QCheckBox('checkbox3')
        self.checkbox3.setObjectName('checkbox3')
        gb_layout.addWidget(self.checkbox3)

        self.find_button = QPushButton('Find child')
        self.find_button.clicked.connect(self.find_widget)
        layout.addWidget(self.find_button)

        self.edit = QLineEdit()
        layout.addWidget(self.edit)

```

```
44      def find_widget(self):
45
46          # 2. Find the checkboxes.
47
48          pattern = self.edit.text()
49          regexp = QRegularExpression(pattern)
50          children = self.groupbox.findChildren(QCheckBox, regexp)
51
52          # 3. Set the checkboxes state to checked
53
54          if len(children) > 0:
55              for child in children:
56                  child.setCheckState(Qt.CheckState.Checked)
57
58
59  if __name__ == '__main__':
60
61      app = QApplication(sys.argv)
62      main_window = Window()
63      main_window.show()
64      sys.exit(app.exec())
```

1. Create the checkboxes. Instantiate three `QCheckBox` objects in a `QGroupBox` and set an object name for each. Also, create a `QLineEdit` for user input and a button to start the search.
2. Find the checkboxes. When the button is pressed, use the `QLineEdit` text to create a `QRegularExpression` and pass it to `findChildren()`.
3. Update the checkboxes. Iterate over the found checkboxes and set each one's check state to `Checked`.



17.4 Manual Ownership Transfer

Sometimes you want to dynamically reconfigure a UI by moving a widget from one container to another – complete with correct behavior in its new location.

A "Clear" button lives inside a group box and clears its sibling `QLineEdit`. You want to move that button to another group box so it clears a different lineedit.

We have two groupboxes, each containing a `QlineEdit`. A single "Clear" button starts in Group 1 and clear its sibling lineedit. Clicking "Move the Button" transfers the button to the other groupbox, and you must set it to clear its new sibling.

```
import sys
from PySide6.QtWidgets import (QApplication, QWidget,
    QGroupBox, QVBoxLayout, QPushButton, QLineEdit)


class MainWindow(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout(self)

        self.group1 = QGroupBox('Group 1')
        self.group1.setObjectName('Groupbox 1')
        self.group1.setLayout(QVBoxLayout())
        self.line_edit1 = QLineEdit()
        self.group1.layout().addWidget(self.line_edit1)
        layout.addWidget(self.group1)

        self.clear_button = QPushButton('Clear text')
        self.clear_button.clicked.connect(self.line_edit1.clear)
        self.group1.layout().addWidget(self.clear_button)

        self.group2 = QGroupBox('Group 2')
        self.group2.setObjectName('Groupbox 2')
        self.group2.setLayout(QVBoxLayout())
        self.line_edit2 = QLineEdit()
        self.line_edit2.setDisabled(True)
        self.group2.layout().addWidget(self.line_edit2)
        layout.addWidget(self.group2)
```

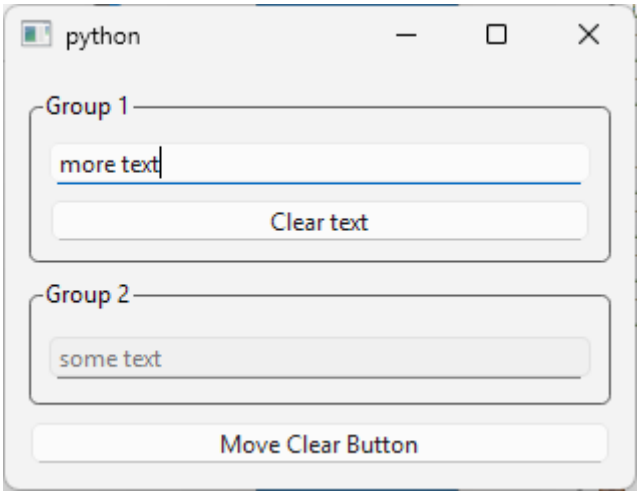
```
32
33          self.move_button = QPushButton('Move Clear Button')
34          layout.addWidget(self.move_button)
35          self.move_button.clicked.connect(self.move_widget)
36
37      def move_widget(self):
38
39          # 1. Adjust any state/side-effects
40
41          old_parent = self.clear_button.parent()
42
43          if old_parent == self.group1:
44              self.line_edit1.setDisabled(True)
45              self.line_edit2.setEnabled(True)
46              self.clear_button.clicked.disconnect(self.line_edit1.clear)
47              self.clear_button.clicked.connect(self.line_edit2.clear)
48              new_parent = self.group2
49          else:
50              self.line_edit2.setDisabled(True)
51              self.line_edit1.setEnabled(True)
52              self.clear_button.clicked.disconnect(self.line_edit2.clear)
53              self.clear_button.clicked.connect(self.line_edit1.clear)
54              new_parent = self.group1
55
56          # 2. Transfer ownership
57
58          new_parent.layout().addWidget(self.clear_button)
59
60
61  if __name__ == '__main__':
62
63      app = QApplication(sys.argv)
64      window = MainWindow()
65      window.show()
66      sys.exit(app.exec())
```

1. Update state and connections. Moving a widget does **not** affect existing signal-slot connections. Our button is still connected to the old lineedit's `clear()` slot. We have to:

    - Disconnect the old connection
    - Connect to the new lineedit
    - Update state (enable/disable the appropriate lineedit)

2. Transfer ownership. There is no need remove the button from its original layout or use `setObject()` to change its parent, as adding it to the new layout takes care of the both.

Now if you move the Clear button from one groupbox to another, it will clear its sibling lineedit text correctly.



18. More Signals & Slots

18.1 A Common Pitfall

We have already seen that a Python lambda can be used to pass additional arguments to a slot, however, there is a pitfall that makes lambdas a bit tricky. The value of a variable used in a lambda is looked up at the time it is called[1]. Let's say you have five lambda functions and you want to pass them a loop index as the parameter:

```
functions = []

for i in range(5):
    functions.append(lambda: print(i))

print("Calling the functions after the loop:")
for func in functions:
    func()
```

The output is:

```
Calling the functions after the loop:
4
4
4
4
4
```

When the functions are executed (in the second loop) the `i` variable value is 4 so all four print 4 instead of 0, 1, 2, 3 and 4 as one would expect. You can change this by assigning the current index to a lambda argument:

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

```
functions = []

for i in range(5):
    functions.append(lambda x=i: print(x))

print("Calling the fixed functions:")
for func in functions:
    func()
```

Now, the output is:

```
Calling the fixed functions:
0
1
2
3
4
```

You want to log each `QCheckBox` checked state change into multiple log files.

```
import sys
from PySide6.QtCore import Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()
        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = QPushButton('Click me!')

        '''
        for i in range(5):
            button.clicked.connect(
                lambda : self.log_to_file(i))
        '''

        for i in range(5):
```

```
            self.button.clicked.connect(
                lambda checked=self.button.isChecked(),
                x=i: self.log_to_file(checked, x))

        layout.addWidget(self.button)

    def log_to_file(self, checked, log_id):
        print('Button clicked')
        print(f'Logging to file no: {log_id}')
        print(f'Checked: {checked}')


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Create the checkbox.
2. Use a loop to connect its `checkStateChanged()` signal to the slots. In each loop iteration, capture the index and checkbox `checkState()` current values
3. Log the checkbox state changes.

Now, if you check the checkbox, the output is:

```
Logging to file no: 0
State: CheckState.Checked
Logging to file no: 1
State: CheckState.Checked
Logging to file no: 2
State: CheckState.Checked
Logging to file no: 3
State: CheckState.Checked
Logging to file no: 4
State: CheckState.Checked
```

18.2 Custom Signals

Most of the Qt classes provide a set of predefined signals. However, when creating your own `QObject` inherited class, you may want to provide custom signals to accompany it.

Suppose we want to create a custom button class that that keeps track of the number of times the user has clicked on it. We also want the class to be able to notify other classes when the counter changes.

PySide6 provides the means to do this in a pythonic way:

```
import sys

# 1. Import the Signal class

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

# 2. Declare the signal

class CounterButton(QPushButton):

    counterChanged = Signal(int)

    def __init__(self, parent=None):
        super().__init__(parent)
        self.counter = 0
        self.clicked.connect(self.on_clicked)

    def on_clicked(self):
        self.counter += 1
        self.counterChanged.emit(self.counter)


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = CounterButton('Click me!')

        # 3. Use the signal

        self.button.clicked.connect(self.on_button_clicked)
        self.button.counterChanged.connect(self.on_counter_changed)

        layout.addWidget(self.button)

```

```
43      @Slot()
44      def on_button_clicked(self):
45          print('Button clicked')
46
47      @Slot(int)
48      def on_counter_changed(self, counter):
49          print('Counter: ', counter)
50
51
52  if __name__ == '__main__':
53
54      app = QApplication(sys.argv)
55      main_window = Window()
56      main_window.show()
57      sys.exit(app.exec())
```

1. Import the `Signal` class from the `PySide6.QtCore` namespace. `Signal` provides the `connect()`, `disconnect()` and `emit()` methods.
2. Create a `QObject` inherited class and add a signal to it. We inherit the class from `QPushButton` and add a signal named `counterChanged()` to it. The signal is declared as a class-level variable of the class and takes a list of Python types as argument. `counterChanged()` takes a single `int` argument. Each time the button is clicked we increment the counter and emit the signal, passing it the `self.counter` variable.
3. In the main window use the custom signal the same way as the predefined Qt signals. We create an instance of the `CounterButton`, create a slot named `on_counter_changed()`, and connect the button's `counterChanged` signal with it. Now, each time the button is clicked, the counter is incremented and the `counterChanged()` signal is emitted:

```
Counter:  1
Button clicked
Counter:  2
Button clicked
Counter:  3
Button clicked
```

18.3 Signal Blocking

At times, you may want to prevent signal emission, for instance during class initialization, or when making programmatic changes to a widget's values that would trigger a signal.

Your task is to allow the user to temporarily block a button's `clicked()` signals

To temporarily block a signal, use the `QObject.blockSignals()` passing it a boolean value. If the value is `True` all signals emitted by the object are blocked. If the value is `False`, signals are not blocked.

```
import sys
from PySide6.QtCore  import Slot, Qt
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QCheckBox, QVBoxLayout)

class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)
        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = QPushButton('Click me!')
        self.button.clicked.connect(self.on_button_clicked)
        layout.addWidget(self.button)

        block_checkbox = QCheckBox('Block signals')
        block_checkbox.checkStateChanged.connect(
            self.on_check_state_changed)
        layout.addWidget(block_checkbox)

    @Slot(bool)
    def on_button_clicked(self, checked):
        print('Button clicked,', 'checked:', checked)

    # Use QObject.blockSignals() to block/unblock a signal

    @Slot(bool)
    def on_check_state_changed(self, state):
        if state == Qt.CheckState.Checked:
            self.button.blockSignals(True)
            print('Signals blocked!')
        else:
            self.button.blockSignals(False)
            print('Signals unblocked!')


if __name__ == '__main__':

```

```
41      app = QApplication(sys.argv)
42      main_window = Window()
43      main_window.show()
44      sys.exit(app.exec())
```

In the example we have a button's `clicked()` signal connected to a slot named `on_button_clicked()`. We use a `QCheckBox()` to block/unblock the button's signals:

```
if state == Qt.CheckState.Checked:
    self.button.blockSignals(True)
    print('Signals blocked!')
else:
    self.button.blockSignals(False)
    print('Signals unblocked!')
```

When the checkbox is checked the button's signals are blocked:

```
Button clicked, checked: False
Signals blocked!
Signals unblocked!
Button clicked, checked: False
```

18.4 Connection Objects

The `Signal.connect()` method has a return value of type `QMetaObject.Connection`. It is a handle to the signal-slot connection the `Signal.connection()` call established.

You need to enable the user to connect or disconnect a button's signals.

To use a connection object in your application:

```python
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = QPushButton('Click me!')
        layout.addWidget(self.button)

        # 1. Get a reference to the Connection object

        self.conn = self.button.clicked.connect(
            self.on_button_clicked)

        self.disconnect_button = QPushButton('Disconnect')
        layout.addWidget(self.disconnect_button)
        self.disconnect_button.clicked.connect(
            self.on_disconnect_button_clicked)

    def on_button_clicked(self):
        print('Button clicked')

    # 2. Use the reference to disconnect signal from slot

    def on_disconnect_button_clicked(self):

        print(type(self.conn))

        if self.conn:
            print('Connection is valid')
            self.button.clicked.disconnect(self.conn)
        else:
            print('Connection is invalid')
            print('Already disconnected')


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Store a reference to the `Connection` object that `Signal.connect()` returned. We store the reference in the instance variable named `conn`.
2. Use the reference to disconnect the signal from the slot. Aside from `connect()` PuSide6 `Signal` object also have a method named `disconnect()` that we use to disconnect the signal from the slot.

The output is:

```
<class 'PySide6.QtCore.QMetaObject.Connection'>
Connection is valid
<class 'PySide6.QtCore.QMetaObject.Connection'>
Connection is invalid
Already disconnected
```

The first time we click the `Disconnect` button the connection is valid and succesfully disconnected. The second time we click it the connection is not valid so we don't call `disconnect()`

If we didn't check the connection validity we would have got a

```
RuntimeWarning: Failed to disconnect (<PySide6.QtCore.QMetaObject.Connection
↪  object at 0x7fee46154040>) from signal "clicked()".
```

18.5 Connecting Multiple Slots with a Signal

You can connect a slot with more than one signals. In the example we create three slots and connect them with the `button.clicked` signal.

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)
```

```
14
15          self.button = QPushButton('Click me!')
16          layout.addWidget(self.button)
17
18          self.button.clicked.connect(self.on_button_clicked_1)
19          self.button.clicked.connect(self.on_button_clicked_2)
20          self.button.clicked.connect(self.on_button_clicked_3)
21
22      def on_button_clicked_1(self):
23          print('Executed first')
24
25      def on_button_clicked_2(self):
26          print('Executed second')
27
28      def on_button_clicked_3(self):
29          print('Executed third')
30
31
32  if __name__ == '__main__':
33
34      app = QApplication(sys.argv)
35
36      main_window = Window()
37      main_window.show()
38
39      sys.exit(app.exec())
```

The output is

```
1  Executed first
2  Executed second
3  Executed third
```

Notice that the slots are executed in the order in which they were connected to the signal. This not necessarily true for signals and slots across different threads.

18.6 Disconnecting

In the `Qt.UniqueConnection` example we saw that you can break a signal-slot connection using the `Signal.disconnect(receiver)` method.

```
import sys
from PySide6.QtCore import Slot, QObject, SIGNAL
from PySide6.QtWidgets import (QApplication, QWidget,
    QHBoxLayout, QVBoxLayout, QLabel, QPushButton)


class Window(QWidget):

    def __init__(self):
        super().__init__()

        self.resize(300, 200)
        layout = QVBoxLayout()
        self.setLayout(layout)

        self.connections = []

        self.button = QPushButton('Click Me')
        self.button.setCheckable(True)
        self.label = QLabel('Button: 0 receivers')

        layout.addWidget(self.button)
        layout.addWidget(self.label)

        h_layout = QHBoxLayout()
        self.connect_btn = QPushButton('Connect signals')
        self.connect_btn.clicked.connect(self.on_connect_btn_clicked)

        self.disconnect_1 = QPushButton('Disconnect 1')
        self.disconnect_1.clicked.connect(self.on_disconnect_1)

        self.disconnect_2 = QPushButton('Disconnect 2')
        self.disconnect_2.clicked.connect(self.on_disconnect_2)

        self.disconnect_3 = QPushButton('Disconnect 3')
        self.disconnect_3.clicked.connect(self.on_disconnect_3)

        self.disconnect_4 = QPushButton('Disconnect 4')
        self.disconnect_4.clicked.connect(self.on_disconnect_4)

        h_layout.addWidget(self.connect_btn)

        v_layout = QVBoxLayout()
        v_layout.addWidget(self.disconnect_1)
        v_layout.addWidget(self.disconnect_2)
        v_layout.addWidget(self.disconnect_3)
        v_layout.addWidget(self.disconnect_4)
        h_layout.addLayout(v_layout)
        layout.addLayout(h_layout)

```

```
51      def on_connect_btn_clicked(self):
52          c1 = self.button.clicked.connect(self.slot_1)
53          c2 = self.button.clicked.connect(self.slot_2)
54          c3 = self.button.toggled.connect(self.slot_3)
55          self.connections.extend([c1, c2, c3])
56          self.update_label()
57
58      def on_disconnect_1(self):
59          for c in self.connections:
60              QObject.disconnect(c)
61          self.update_label()
62
63      def on_disconnect_2(self):
64          QObject.disconnect(self.button, None, None, None)
65          self.update_label()
66
67      def on_disconnect_3(self):
68          QObject.disconnect(self.button, SIGNAL('clicked(bool)'), None, None)
69          self.update_label()
70
71      def on_disconnect_4(self):
72          QObject.disconnect(self.button, None, self, None)
73          self.update_label()
74
75      @Slot(bool)
76      def slot_1(self, checked):
77          print('Slot 1 executing')
78          self.update_label()
79
80      @Slot()
81      def slot_2(self):
82          print('Slot 2 executing')
83          self.update_label()
84
85      @Slot(bool)
86      def slot_3(self, checked):
87          print('Slot 3 executing')
88          self.update_label()
89
90      def update_label(self):
91          count_1 = self.button.receivers(SIGNAL('clicked(bool)'))
92          count_2 = self.button.receivers(SIGNAL('toggled(bool)'))
93          count = count_1 + count_2
94          self.label.setText(f'Button: {count} receivers')
95
96
97  if __name__ == '__main__':
98
99      app = QApplication(sys.argv)
100     main_window = Window()
```

```
101     main_window.show()
102     sys.exit(app.exec())
```

```
1  self.button.clicked.disconnect(self.on_clicked)
```

This breaks up the connection between `button.clicked` signal and the `on_clicked()` slot. However, `QObject` also has a `disconnect()` method with several overloads that give you more control over which signal-slot connections are removed.

- `static disconnect(connection)` lets you pass a connection object to it. In the example we store all connection objects in the `Window.connections` list. On clicking the `Disconnect 1` button all connections are disconnected in a loop:

```
1  def on_disconnect_1(self):
2      for c in self.connections:
3          QObject.disconnect(c)
4      self.update_label()
```

- `static disconnect(sender, signal, receiver, member)` lets you specify both the `sender` and `the receiverobjects as well as thesignaland thememberie the slot. On clicking theDisconnect 2button we setself.buttonas thesenderand the other three arguments toNone. Noneacts as a wildcard so this disconnects **all** signal-slot connections whereself.button‘ is the signal sender.

```
1  def on_disconnect_2(self):
2      QObject.disconnect(self.button, None, None, None)
3      self.update_label()
```

- On clicking the `Disconnect 3` button we set `self.button` as `sender` and `clicked(bool)` as `signal.` receiverandmemberare still set toNone‘.

```
def on_disconnect_3(self):
    QObject.disconnect(self.button, SIGNAL('clicked(bool)'), None, None)
    self.update_label()
```

This will disconnect only the slots with the signature `clicked(bool)` ie. `Slot1` and `Slot3`. The connection between `button.clicked` and `Slot2` remains.

- On clicking the `Disconnect 4` button we set `sender` to `button` and `reciver` to `self` (ie to the `Window` instance). `signal` and `member` are set to `None`. This breaks up all connections since all the slots are `Window` members.

```
def on_disconnect_4(self):
    QObject.disconnect(self.button, None, self, None)
    self.update_label()
```

19. Events

In the Qt framework, events are objects derived from the `QEvent` class that represent occurences, such as user input or system notifications, that require handling. Events are especially relevant for Qt widgets, as they allow widgets to react to changes like mouse movements, keyboard presses, or window resizes. The event system relies on Qt's event loop, which processes queued events and dispatches them to the appropriate `QObject` receivers. Developers can customize event handling by overriding virtual event methods in subclasses, providing fine-grained control over application behavior. Common event types include `QMouseEvent` for pointer interactions, `QKeyEvent` for keyboard actions, and `QPaintEvent` for rendering updates[1].

There are five ways events can be processed:

1. Reimplementing event type-specific event handler methods like `mousePressEvent()`.
2. Reimplementing the `QObject.event()` method.
3. Installing an event filter on the object.
4. Installing an event filter on `QCoreApplication` instance.
5. Reimplementing the `QCoreApplication.notify()` method[2].

19.1 Event Handlers

A Qt class main event handler is the `event()` method. When an event occurs, this method receives a `QEvent` object representing it. Typically, it does not handle the event itself but instead calls the specific event handler method for that event type. You can override the base class's event handler completely, in which case you must implement all handling yourself, or handle only certain event types and call the base class method for others.

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

[2] https://doc.qt.io/qtforpython-6/

Whenever your application's main window becomes visible on the screen, add it to the list of visible windows. When it becomes hidden, remove it from the list. Your additional task is to record and suppress key press events when the user presses the letter 'q'.

```
import sys
from PySide6.QtCore import Qt, QEvent
from PySide6.QtWidgets import (QApplication,
    QWidget, QLabel, QVBoxLayout)

class Window(QWidget):

    def __init__(self):

        super().__init__()

        self.resize(300, 200)
        layout = QVBoxLayout()
        self.setLayout(layout)

        self.label = QLabel('Hello, events!')
        self.label.setAlignment(Qt.AlignCenter)
        layout.addWidget(self.label)

    # Centralized event handler

    def event(self, event):
        if event.type() == QEvent.Type.KeyPress:
            print(f'Intercepted key pressed: {event.text()}')
            #return True
        if event.type() == QEvent.Type.Show:
            print('Intercepted show event')
            #return True
        return super().event(event)

    # Specialized event handlers

    # 1. Extend the keyPressEvent() method

    def keyPressEvent(self, event):
        if event.text() == 'q':
            print(f'isAccepted: {event.isAccepted()}')
            self.label.setText('q pressed')
        else:
            self.label.clear()
            super().keyPressEvent(event)

```

```
43      # 2. Extend the showEvent() method
44
45      def showEvent(self, event):
46          self.label.setText('Added to visible windows list')
47          super().showEvent(event)
48
49      # 3. Extend the hideEvent() method.
50
51      def hideEvent(self, event):
52          self.label.clear()
53          print('Removed from visible windows list')
54          super().hideEvent(event)
55
56  if __name__ == '__main__':
57
58      app = QApplication(sys.argv)
59      main_window = Window()
60      main_window.show()
61      sys.exit(app.exec())
```

1. Extend the `keyPressEvent()` method. To suppress the 'q' key presses, check if `event.text()` equals 'q'. If it does, record it in a label. Since `QKeyEvent.isAccepted()` is `True` by default, this suppresses the 'q' key presses. For all other key presses, call the base class `keyPressEvent()`.
2. Extend the `showEvent()` method. Simulate adding the window to a visible windows list by recording the event in the label, then call the base class `showEvent()`.
3. Extend the `hideEvent()` method. When the window is hidden, the label is also hidden, so simulate removing the window from the visible windows list by printing a message. Then call the base class `hideEvent()` for further processing.

The `event()` method dispatches events to type-specific event handlers like `showEvent()` or `keyPressEvent()`. While you can use it to handle events directly, the documentation recommends using the event type-specific handlers instead. In the example, we intercept key press and show events in `event()` and print a message for each. Then we call the base class `event()` for further processing. You can also use `event()` to suppress events - if you uncomment the two `return True` statements, events never reach `keyPressEvent()` and `showEvent()`.

19.2 Object Event Filters

In the previous section we saw how to intercept events by overriding a `QObject`'s `event()` method before the events reach the specific event handlers. Sometimes, however, you need to monitor or suppress events destined for *other* objects. This is achieved by installing an **event filter**.

Your task is to log mouse press events for a group of buttons while completely suppressing clicks on one particular button.

To use an event filter in your application:

```
import sys

from PySide6.QtCore import QObject, QEvent
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)

# 1. Create a class and implement its eventFilter() method.

class MousePressFilter(QObject):

    def eventFilter(self, watched, event):
        if event.type() == QEvent.Type.MouseButtonPress:
            print('Filter mouse press for: ', watched.objectName())
            if watched.objectName() == 'button2':
                return True
        return super().eventFilter(watched, event)


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)

        layout = QVBoxLayout()
        self.setLayout(layout)

        button1 = QPushButton('Button 1')
        button1.setObjectName('button1')
        button2 = QPushButton('Button 2')
        button2.setObjectName('button2')
        button3 = QPushButton('Button 3')
        button3.setObjectName('button3')
```

```
34
35          # 2. Create the event filter object.
36
37          event_filter = MousePressFilter(self)
38
39          # 3. Install the event filter for the target objects.
40
41          button1.installEventFilter(event_filter)
42          button2.installEventFilter(event_filter)
43          button3.installEventFilter(event_filter)
44
45          button1.clicked.connect(self.on_button_clicked)
46          button2.clicked.connect(self.on_button_clicked)
47          button3.clicked.connect(self.on_button_clicked)
48
49          layout.addWidget(button1)
50          layout.addWidget(button2)
51          layout.addWidget(button3)
52          layout.addStretch()
53
54      def on_button_clicked(self):
55          print('Button clicked: ', self.sender().objectName())
56
57
58  if __name__ == '__main__':
59
60      app = QApplication(sys.argv)
61      main_window = Window()
62      main_window.show()
63      sys.exit(app.exec())
```

1. Create a class that inherits from `QObject` and implement its `eventFilter()` method. The signature is:

```
1  def eventFilter(self, watched: QObject, event: QEvent) -> bool
```

The `watched` parameter refers to the object receiving the event. In this example, we check for `MouseButtonPress` events, log them to the console, and return `True` for `button2` to consume the event (preventing further processing, including the `clicked()` signal). For all other cases, we return the result of the base class implementation.

2. Instantiate the event filter object (in this case a `MousePressFilter` instance named `event_filter`.

3. Install the event filter on each target widget using `installEventFilter()`.

If you delete the watched object inside `eventFilter()`, you need to return `True`, otherwise the application might crash.

The output is:

```
Filter mouse press for:  button1
Button clicked:  button1
Filter mouse press for:  button2
Filter mouse press for:  button3
Button clicked:  button3
```

Consuming an event means stopping it from being processed further. The `clicked()` signal requires Qt to process both a mouse press and a mouse release on the button. By consuming the `mouseButtonPress` event in the filter, the button never receives the press event, so it can't recognize a complete click action.

Event filters can be chained, meaning you can install multiple filters on the same object. When event occurs, Qt calls the filters in reverse order of installation.

19.3 Application-Wide Event Filters

In the previous section, we installed an event filter on individual buttons. An event filter installed on the `QApplication` object, however, can intercept all events for all widgets in the application - including mouse events that would normally be ignored by disabled widgets.

You want users to be able to "click" disabled buttons (i.e., trigger their clicked signal) without re-enabling them visually or functionally.

To implement a global event filter:

```python
import sys

from PySide6.QtCore import QObject, QEvent
from PySide6.QtWidgets import (QApplication, QWidget,
    QPushButton, QVBoxLayout)


# 1. Create the event filter class.

class MousePressFilter(QObject):

    def eventFilter(self, watched, event):
        if event.type() == QEvent.Type.MouseButtonPress:
            print('Filter mouse press for: ', watched.objectName())
            if isinstance(watched, QPushButton):
                if not watched.isEnabled():
                    watched.clicked.emit()
                    return True
        return super().eventFilter(watched, event)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        self.setObjectName('Main Window')

        layout = QVBoxLayout()
        self.setLayout(layout)

        button1 = QPushButton('Button 1')
        button1.setObjectName('button1')
        button1.clicked.connect(self.on_button_clicked)

        button2 = QPushButton('Button 2')
        button2.setObjectName('button2')
        button2.clicked.connect(self.on_button_clicked)
        button2.setDisabled(True)

        button3 = QPushButton('Button 3')
        button3.setObjectName('button3')
        button3.clicked.connect(self.on_button_clicked)

        layout.addWidget(button1)
        layout.addWidget(button2)
        layout.addWidget(button3)
        layout.addStretch()

```

```
51      def on_button_clicked(self):
52          print('Button clicked: ', self.sender().objectName())
53
54
55  if __name__ == '__main__':
56
57      app = QApplication(sys.argv)
58
59      # 2. Instantiate the event filter object.
60
61      filter_obj = MousePressFilter(app)
62
63      # 3. Install the event filter on the application.
64
65      app.installEventFilter(filter_obj)
66
67      main_window = Window()
68      main_window.show()
69      sys.exit(app.exec())
```

1. Create the event filter class. Override `eventFilter()` to detect `MouseButtonPress` events If the target widget (the `watched` parameter) is a disabled `QPushButton`, manually emit its `clicked()` signal and consume the event by returning `True`.
2. Instantiate the filter object, passing the `QApplication` instance as its parent.
3. Install the filter on the `QApplication` instance using `QApplication.installEventFilter()`.

Sample console output when clicking each button in order:

```
1  Filter mouse press for:  Main WindowWindow
2  Filter mouse press for:  button1
3  Button clicked:  button1
4  Filter mouse press for:  Main WindowWindow
5  Filter mouse press for:  button2
6  Button clicked:  button2
7  Filter mouse press for:  Main WindowWindow
8  Filter mouse press for:  button3
9  Button clicked:  button3
```

Notice that filter receives each mouse press event twice. Qt delivers them first to the top-level window, then to the specific button under the mouse.

Since the filter is installed at the application level, it intercepts the event at both stages.

This approach bypasses Qt's normal disabled-state behavior. Disabled buttons will emit `clicked()` signals but won't show visual press feedback. Application-wide filters see every event in your application, so keep the `eventFilter()` method efficient to avoid perdormance issues.

19.4 Event Propagation

The `QCoreApplication.notify()` method delivers all events to their receiver objects. Certain event types, such as mouse and key events, propagate up the parent widget chain if the receiver ignores them (i.e., does not accept the event).

Your application requires users to create an account. To better understand how they interact with the "Create account" dialog - for usability testing or debugging - you want to log every mouse click and keyboard press, including which widget received the event and how events propagate when not handled.

To implement a global activity logger:

```
import sys
from PySide6.QtCore import QObject, QEvent, Qt
from PySide6.QtWidgets import (QApplication, QWidget,
    QGroupBox, QVBoxLayout, QLineEdit, QLabel, QCheckBox)


# 1. Create the event filter class.

class EventFilter(QObject):

    def eventFilter(self, watched, event):
        if event.type() == QEvent.Type.MouseButtonPress:
            print('Log: mouse pressed in ', watched.objectName())
        if event.type() == QEvent.Type.KeyPress:
            print('Log: key pressed in ', watched.objectName())
        return super().eventFilter(watched, event)

class Window(QWidget):

```

```
    def __init__(self):
        super().__init__()
        self.setObjectName('MainWindow')

        layout = QVBoxLayout(self)

        self.outer_groupbox = QGroupBox('Outer GroupBox')
        self.outer_groupbox.setObjectName('outer_groupbox')
        self.outer_groupbox.setLayout(QVBoxLayout())

        self.inner_groupbox = QGroupBox('Inner GroupBox')
        self.inner_groupbox.setObjectName('inner_groupbox')
        self.inner_groupbox.setLayout(QVBoxLayout())
        self.outer_groupbox.layout().addWidget(self.inner_groupbox)

        # 3. Create the user registration form.

        self.username_edit = QLineEdit()
        self.username_edit.setPlaceholderText('User name')
        self.username_edit.setObjectName('line_edit')
        self.username_edit.textChanged.connect(self.update_email)
        self.inner_groupbox.layout().addWidget(self.username_edit)

        self.email_label = QLabel()
        self.email_label.setObjectName('label')
        self.email_label.setFocusPolicy(Qt.FocusPolicy.StrongFocus)
        self.email_label.setTextInteractionFlags(
            Qt.TextInteractionFlag.TextSelectableByKeyboard |
            Qt.TextInteractionFlag.TextSelectableByMouse)
        self.inner_groupbox.layout().addWidget(self.email_label)

        self.email_checkbox = QCheckBox('Enable email notifications')
        self.email_checkbox.setObjectName('checkbox')
        self.inner_groupbox.layout().addWidget(self.email_checkbox)

        layout.addWidget(self.outer_groupbox)
        layout.addStretch()

    def update_email(self, text):
        if text:
            self.email_label.setText(f'{text}@domain.com')
        else:
            self.email_label.clear()


if __name__ == '__main__':

    app = QApplication(sys.argv)

    # 2. Instantiate and install the filter.
```

```
70
71      event_filter = EventFilter(app)
72      app.installEventFilter(event_filter)
73
74      main_window = Window()
75      main_window.show()
76      sys.exit(app.exec())
```

1. Create an `EventFilter` class inheriting from `QObject`. In its `eventFilter()` method, detect `MouseButtonPress` and `KeyPress` events and log them to the console.
2. In the main section, after creating the `QApplication`, instantiate the filter and install it globally using `QApplication.installEventFilter()`.
3. Create the main window as a `QWidget` containing:

- A `QLineEdit` for entering the user name,
- A `Qlabel` below it displaying the generated e-mail address. Make the text selectable so users can copy it.
- A checkbox to enable email notifications.

The widgets are nested inside two `QGroupBox` containers to create a clear parent hierarchy.

Here is a sample output from typical interactions:

```
Log: key pressed in  MainWindowWindow
Log: key pressed in  line_edit
Log: mouse pressed in  MainWindowWindow
Log: mouse pressed in  checkbox
Log: mouse pressed in  MainWindowWindow
Log: mouse pressed in  label
Log: key pressed in  MainWindowWindow
Log: key pressed in  label
Log: key pressed in  inner_groupbox
Log: key pressed in  outer_groupbox
Log: key pressed in  MainWindow
```

Because the filter is installed at the application level, it sees events at every step of delivery and propagation.Interactive widgets like the line edit and checkbox accept most events they receive, stopping further propagation. The label behaves differently: it accepts mouse presses but ignores most key presses. As a result, key events on the label propagate up through the inner group box, outer group box, and finally the main window.

19.5 Custom Events

Qt allows you to define and post your own event types, extending the event system for application-specific needs. Custom events are processed in the event loop like built-in events and can be handled in the `customEvent()` handler, `event()` or via event filters.

To create a custom event:

- Register a unique type with `QEvent.registerEventType()`.
- Subclass `QEvent` and add data attributes.
- Post instances to receivers using `QApplication.postEvent(receiver, event)`.

Your task is to create a custom Qt event whenever the current CPU usage percent exceeds a threshold. The event should hold the CPU usage percent as the data.

To create custom Qt events:

```
import sys
import psutil
from PySide6.QtCore import Qt, QEvent, QTimer, QObject
from PySide6.QtWidgets import (QApplication,
    QWidget, QLabel, QVBoxLayout)

# 1. Create a custom event class.

class CpuUsageEvent(QEvent):

    event_type = QEvent.Type(QEvent.registerEventType())

    def __init__(self, cpu_percent):
        super().__init__(CpuUsageEvent.event_type)
        self.cpu_percent = cpu_percent

class CpuUsageEventFilter(QObject):

    def eventFilter(self, watched, event):
        if event.type() == CpuUsageEvent.event_type:
            print('In eventFilter(): ', event.cpu_percent)
            watched.setText(f'{event.cpu_percent} % CPU used')
```

```
        return super().eventFilter(watched, event)

class Window(QWidget):

    THRESHOLD = 10 # CPU percent

    def __init__(self):

        super().__init__()
        self.resize(300, 200)

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.label = QLabel()
        self.label.setAlignment(Qt.AlignCenter)

        self.filter = CpuUsageEventFilter()
        self.label.installEventFilter(self.filter)
        layout.addWidget(self.label)

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.on_timeout)
        self.timer.start()

    def on_timeout(self):
        cpu_percent = psutil.cpu_percent(interval=None)
        if cpu_percent > self.THRESHOLD:
            # 3. Create an event object and post it.
            QApplication.postEvent(self.label, CpuUsageEvent(cpu_percent))
            QApplication.postEvent(self, CpuUsageEvent(cpu_percent))
        else:
            self.label.clear()

    def event(self, event):
        if event.type() == CpuUsageEvent.event_type:
            print('In event(): ', event.cpu_percent)
        return super().event(event)

    def customEvent(self, event):
        if event.type() == CpuUsageEvent.event_type:
            print('In customEvent(): ', event.cpu_percent)


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
```

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

1. Define a custom event class `CpuUsageEvent` that inherits from `QEvent` and carries the used CPU percent value. Use `QEvent.registerEventType()` to assign a unique type to the class.
2. Use a `QTimer` with an interval of 1000 ms to periodically poll the CPU usage percent with `psutil.cpu_percent`.
3. If the CPU percent exceeds the threshold, create and post a new event object to each receiver.

Note that custom events do not automatically propagate up the widget hierarchy like mouse or key events.

20. Timers

In programming, timers are used to measure time intervals and trigger actions or events after a speecified delay (single-shot) or at regular intervals. They enable task scheduling, such as updating GUI elements, or polling resources, without blocking the main execution flow.[1]

In Qt, timers are integrated with the application's event loop. They allow non-blocking time-based operations, ensuring responsive applications, ensuring responsive GUI applications. Qt provides several timer facilites:

- `QObject.startTimer()`: A low-level method on any QObject to start a timer with a millisecond interval.
- `QBasicTimer`: A lightweight wrapper around a timer Id from `startTimer()`
- `QTimer`: A high-level `QObject` subclass with millisecond precision (up to 24 days range) which emits a `timeout()` signal.
- `QChronoTimer`: A drop-in replacement for `QTimer` but with nanosecond precision and larger ranges (up to 292 years).

The following two timer classes do not integrate with the Qt event loop:

- `QElapsedTimer`: A stopwatch for measuring elapsed time.
- `QDeadlineTimer`: Represents a deadline for timeouts in blocking functions.

The one you will be using the most of the time is `QTimer` - here are its most important methods:

Here's a table of QTimer's most important methods:

[1] If you don't implement `__init__()` in your class at all, the parent class gets initalized automatically.

Method	Description
`start(int msec)`	Starts or restarts the timer with a timeout of msec milliseconds
`start()`	Starts or restarts the timer with the timeout specified in `setInterval()`
`stop()`	Stops the timer
`setInterval(int msec)`	Sets the timeout interval in milliseconds
`interval()`	Returns the current interval in milliseconds
`isActive()`	Returns true if the timer is running, false otherwise
`setSingleShot(bool)`	Sets whether the timer fires only once (true) or repeatedly (false)
`isSingleShot()`	Returns true if the timer is single-shot
`timerId()`	Returns the timer's ID, or -1 if the timer is not running
`remainingTime()`	Returns the remaining time in milliseconds before the timer times out
`setTimerType(Qt::TimerType)`	Sets the timer's precision (Precise, Coarse, or VeryCoarse)
`timerType()`	Returns the timer's type
`timeout`	Signal emitted when the timer times out
`singleShot(int msec, receiver, slot)`	Static method that creates a single-shot timer that calls a slot after msec milliseconds

20.1 Single-Shot

Qt's single-shot timer is a feature of the `QTimer` class that allows developers to schedule a one-time event to occur after a specified delay,without the need for

repeated intervals. It can be useful for tasks like delaying UI updates, handling asynchronous operations, or implementing timeouts in applications.

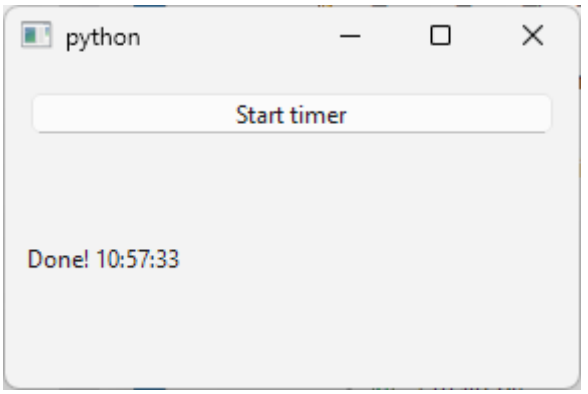
You need to updata a label's text one second after a button is clicked.

To do this:

```
import sys
from PySide6.QtCore import QTime, QTimer, Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QLabel, QVBoxLayout)

class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create the button object

        self.button = QPushButton('Start timer')
        self.button.clicked.connect(self.on_button_clicked)
        layout.addWidget(self.button)

        self.label = QLabel('Start the timer')
        layout.addWidget(self.label)

    # 2. Call the singleShot() class method

    @Slot()
    def on_button_clicked(self):
        self.button.setDisabled(True)
        self.label.setText('Waiting...')
        QTimer.singleShot(1000, self.on_single_shot)

    # 3. Set the label text after 1000ms

    @Slot()
    def on_single_shot(self):
        current_time = QTime.currentTime()
        text = f'Done! {current_time.toString("hh:mm:ss")}'
        self.label.setText(text)
        self.button.setEnabled(True)
```

```
40
41
42  if __name__ == '__main__':
43
44      app = QApplication(sys.argv)
45      main_window = Window()
46      main_window.show()
47      sys.exit(app.exec())
```

1. Create `QPushButton` and `QLabel` instances, add them to the window layout, and connect the button's `clicked` signal to a slot.
2. In the slot, call `QTimer.singleShot()`, passing a 1000ms delay interval and the `on_single_shot()` method to execute after the delay.
3. In `on_single_shot()`, get the current time and display it in the label. This executes with a one-second delay while keeping the GUI responsive.



As `singleSHot()` is a class (static) method, you don't create an instance of the `QTimer` class.

20.2 Starting and Stopping a Timer

Calling a timer's `start()` method begins firing `timeout()` signals at specified intervals. Stopping a timer with `stop()` halts its operation, preventing further timeouts until it is restarted.

You are given the task of creating a simple application that counts elapsed seconds, allowing the user to start and stop the tracking as needed.

To do this:

```
import sys
from PySide6.QtCore import QTimer, Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QLabel, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        self.counter = 0

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create the start and stop buttons.

        self.start_button = QPushButton('Start timer')
        self.start_button.clicked.connect(self.start_timer)
        layout.addWidget(self.start_button)

        self.stop_button = QPushButton('Stop timer')
        self.stop_button.clicked.connect(self.stop_timer)
        self.stop_button.setEnabled(False)
        layout.addWidget(self.stop_button)

        self.label = QLabel('0')
        layout.addWidget(self.label)

        # 2. Create the timer and set its interval

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.on_timeout)

    # 3. Start/stop the timer on buttons click

    @Slot()
    def start_timer(self):
        self.start_button.setDisabled(True)
        self.stop_button.setEnabled(True)
        self.timer.start()

    @Slot()
    def stop_timer(self):
        self.start_button.setEnabled(True)
```

```
49          self.stop_button.setDisabled(True)
50          self.timer.stop()
51
52      # 4. Update the display on timer timeout.
53
54      @Slot()
55      def on_timeout(self):
56          self.counter += 1
57          self.label.setText(str(self.counter))
58
59
60  if __name__ == '__main__':
61
62      app = QApplication(sys.argv)
63      main_window = Window()
64      main_window.show()
65      sys.exit(app.exec())
```

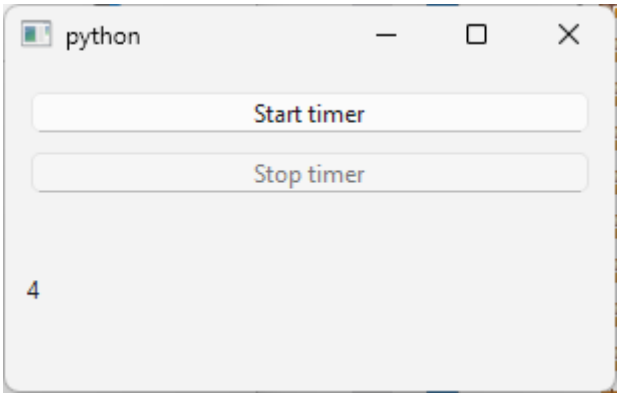
1. Create a Start timer and a Stop timer `QPushButtons` along with a `QLabel` that displays the current number of elapsed seconds. Keep the count in an instance field named `counter` initialized to 0.
2. Create a `QTimer` object and set its interval to 1000 milliseconds.
3. When the Start timer button is clicked:

   - Disable the Start button and enable the Stop button,
   - Start the timer.

   When the Stop timer button is clicked:

   - Enable the Start button and disable the Stop button,
   - Stop the timer.

4. Each time the timer's timeout signal is emitted, increment `counter` by one and update the label to show the new value.



20.3 Adjusting a Timer Interval

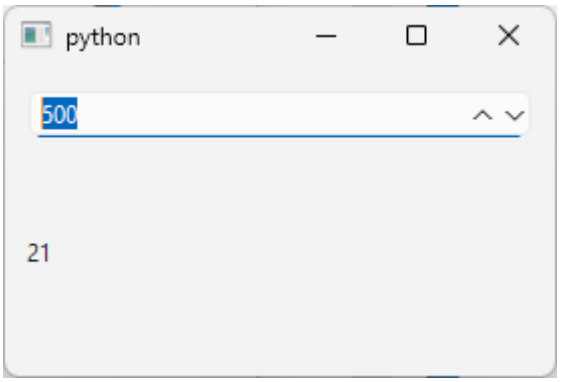
You need a timer that remains active through your application's runtime, while allowing the user to adjust its interval without restarting the countdown.

```
import sys
from PySide6.QtCore import QTimer, Slot
from PySide6.QtWidgets import (QApplication, QWidget,
    QLabel, QSpinBox, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        self.counter = 0
        self.pending_interval = None

        layout = QVBoxLayout()
        self.setLayout(layout)

        # 1. Create the spinbox and timer objects

        self.spinbox = QSpinBox()
        self.spinbox.setRange(200, 2000)
        self.spinbox.setValue(1000)
```

```
24          self.spinbox.setSingleStep(100)
25          self.spinbox.valueChanged.connect(self.adjust_interval)
26          layout.addWidget(self.spinbox)
27
28          self.label = QLabel(str(self.counter))
29          layout.addWidget(self.label)
30
31          self.timer = QTimer()
32          self.timer.timeout.connect(self.on_timeout)
33
34          self.timer.start(self.spinbox.value())
35
36      # 3. On timeout, check if the user changed the spinbox value
37      #    and adjust the timer interval to sync them.
38      #    Also, perform timed tasks.
39
40      @Slot()
41      def on_timeout(self):
42
43          if self.pending_interval is not None:
44              self.timer.setInterval(self.pending_interval)
45              self.pending_interval = None
46
47          self.counter += 1
48          self.label.setText(str(self.counter))
49
50      # 2. When the spinbox value changes store it in a
51      #    member variable.
52
53      @Slot(int)
54      def adjust_interval(self, value):
55          self.pending_interval = value
56
57
58  if __name__ == '__main__':
59
60      app = QApplication(sys.argv)
61      main_window = Window()
62      main_window.show()
63      sys.exit(app.exec())
```

1. Create the spinbox and timer. In the main window's `__init__()`, create a `QSpinBox` with an initial value of 1000ms, a range of 200-2000 ms, and a step of 100ms. COnnect its `valueChanged()` signal to a slot for handling value changes. Then, create a `QTimer`, connect its `timeout()` signal to a slot, and start it with the initial interval.
2. When the user changes the spinbox value, store that value in the `pending_interval` instance variable.

3. On the timer's `timeout()` signal, check if `pending_interval` is set. If it is, adjust the timer interval and reset `pending_interval` to `None`. Also perform your application timer-related tasks (in this case, update a label).



The intermediate `pending_interval` variable is necessary to avoid restarting the timer's current countdown cycle when the spinbox value changes. Calling `setInterval()` directly on a running `QTimer` stops and restarts it, resetting the countdown to the new interval from that moment, which could delay or advance the next `timeout()` signal. By using `pending_interval`, we defer the change until the start of the `on_timeout()` slot - after the current interval has completed - ensuring the new interval only applies to future cycles without interrupting the ongoing one.

20.4 Countdown Timer

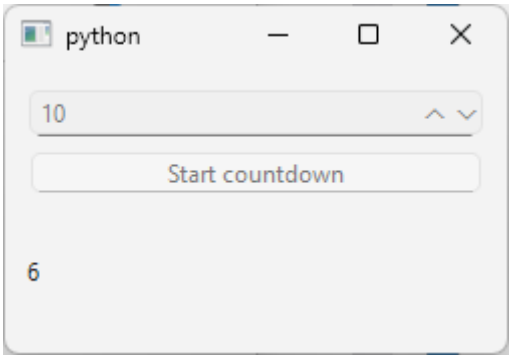
Countdown timers are designed to count down from a specified time interval to zero, signaling the end of a period or event. They operate by decrementing the time at regular intervals, often in seconds, and can trigger actions like notifications or program executions upon reaching zero.

You need to implement a simple countdown in your application.

```python
import sys
from PySide6.QtCore import QTimer, Slot
from PySide6.QtWidgets import (QApplication, QWidget,
    QPushButton, QLabel, QSpinBox, QVBoxLayout)


class Window(QWidget):

    def __init__(self):

        super().__init__()

        # 1. Set the countdown variable with an initial value.
        #    Add a spinbox to let the user adjust it.
        #    Add a button for starting the countdown.
        #    Create a timer.

        self.counter = 5

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.spinbox = QSpinBox()
        self.spinbox.setMinimum(1)
        self.spinbox.setMaximum(10)
        self.spinbox.setValue(self.counter)
        self.spinbox.setSingleStep(1)
        self.spinbox.valueChanged.connect(self.on_value_changed)
        layout.addWidget(self.spinbox)

        self.start_button = QPushButton('Start countdown')
        self.start_button.clicked.connect(self.start_countdown)
        layout.addWidget(self.start_button)

        self.label = QLabel(str(self.counter))
        layout.addWidget(self.label)

        self.timer = QTimer()
        self.timer.setInterval(1000)
        self.timer.timeout.connect(self.on_timeout)

    # 2. On the Start button click:
    #    Set the counter value.
    #    Disable the Start button.
    #    Disable the spinbox.
    #    Start the timer.

    @Slot()
    def start_countdown(self):
        self.counter = self.spinbox.value()
```

```
51          self.label.setText(str(self.spinbox.value()))
52          self.start_button.setDisabled(True)
53          self.spinbox.setDisabled(True)
54          self.timer.start()
55
56      # 3. On timeout:
57      #    Decrement the counter
58      #    Set the label text
59      #    If the counter reaches zero, stop the timer
60
61      @Slot()
62      def on_timeout(self):
63          self.counter -= 1
64          self.label.setText(str(self.counter))
65          if self.counter == 0:
66              self.timer.stop()
67              self.start_button.setEnabled(True)
68              self.spinbox.setEnabled(True)
69
70      @Slot(int)
71      def on_value_changed(self, value):
72          self.counter = value
73          self.label.setText(str(self.counter))
74
75
76  if __name__ == '__main__':
77
78      app = QApplication(sys.argv)
79      main_window = Window()
80      main_window.show()
81      sys.exit(app.exec())
```

1. Add a `counter` variable to your main class and set its initial value. Add a spinbox to let the user set the starting countdown value and a button to start the countdown. Also, create a `QTimer` and set its interval.
2. On the Start button click, set `counter` value, disable the Start button and the spinbox, and start the timer.
3. On the timer `timeout()`, decrement `counter`, update the label text, and if `counter` reaches zero, stop the timer and re-anable the button and the spinbox.



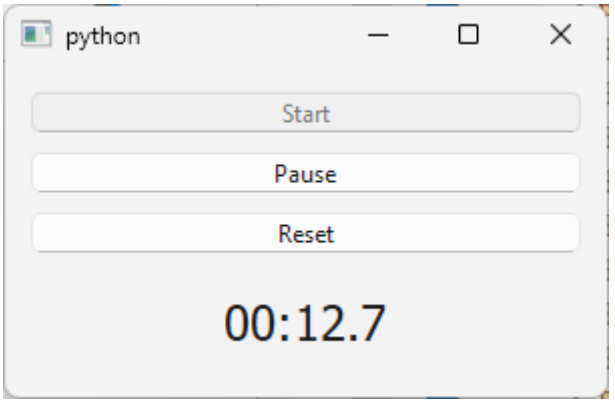
This ensures the countdown runs uninterrupted by disabling controls during operation, providing a basic countdown mechanism.

20.5 Stopwatch

The previous example showed how to decrease a counter value at regular intervals, stopping the timer when the counter reaches zero. Conversely, you can also use a timer to increase a value periodically.

Your task is to create a simple stopwatch application with one-hundredth of a second (100ms) resolution and 'Start', 'Pause' and 'Reset' functionality.

1. Add an `elapsed_time` instance variable to the main window. This is a `QTime` object initialized to zero hours, minutes, seconds and milliseconds. `QTime` is convenient here because it provides `addMSecs()` for easy time increments.
2. Create the buttons. When the aplication starts, the 'Start' button is enabled, the 'Pause' button is disabled and the 'Reset' button is always enabled (allowing reset at any time).
    - Clicking 'Start' disables it, enables 'Pause' and starts the timer.
    - Clicking 'Pause' disables it, enables 'Start' and stops the timer.
3. Create a `QTimer`, set its interval to 100 milliseconds, and connect its `timeout()` signal to a slot.
4. Update the time. In the `on_timeout()` slot, add the 100 ms to `elapsed_time` using `addMSecs()`, then update a label to display the current time in the format `mm:ss.z` (minutes:seconds.tenths).


python
Start
Pause
Reset
00:12.7

21. Properties

22. Model-View Programming with `QAbstractListModel`

This chapter introduces custom Qt models using `QAbstractListModel` through a series of simple demonstrations manipulating a list of company clients.

The Qt model-view architecture is Qt's variant of the model-view-controller (MVC) design pattern where, per the documentation, the controller and view are combined. The model and view are decoupled, allowing the same model to be used with multiple views.

Qt provides several abstract classes you can subclass for custom models, such as:

- `QAbstractItemModel`
- `QAbstractTableModel`
- `QAbstractListModel`

Qt also offers ready-to-use concrete models, including:

- `QStandardItemModel`
- `QFileSystemModel`
- `QSqlQueryModel`

Earlier examples demonstrated some of these with Qt view classes. Here, we focus on creating custom models.

22.1 Read-only List Model

[TODO: QAbstractListModel inheritance tree]

Of the three abstract model classes above, `QAbstractListModel` is the easiest to subclass. It provides default implementations for several `QAbstractItemModel` methods and offers a specialized interface for simple, non-hierarchical sequences of items (lists).

Like other model classes, a `QAbstractListModel` subclass acts as an intermediary between a data source and a view:

[TODO: DATA-MODEL-VIEW DIAGRAM]

A data source can range from a simple Python list or structured text file, relational database data, and anything in between. The model's role is to supply data in a format that a Qt view can read and display.

To create a read-only list model you must implement at least two methods:

- `rowCount()`: Returns the number of rows in the model. (e.g., `len(lst)` for a Python list, line count for a file, or `count(*)` for a SQL query). The `parent` parameter is for hierarchical model and unused here.
- `data(index, role)`: Returns the data for the given role and item referenced by the index. For list models, `index` is a `QModelIndex` object with `column()` as zero and `row()` indicating the position in the underlying data. The `role` is one of the `Qt.ItemDataRole` enumeration values(default: `DisplayRole`).

You have a text file listing company clients ('data.txt') and you need to display them in a list view.

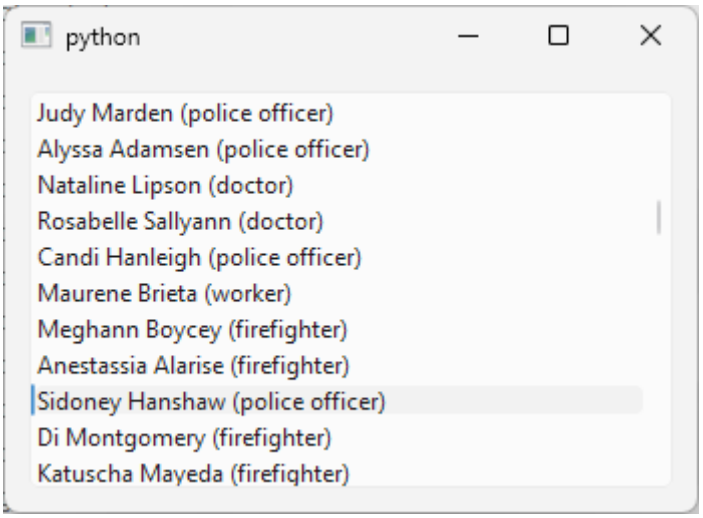
To achieve this:

```
from PySide6.QtCore import QAbstractListModel, Qt

# 1. Create a QAbstractListModel subclass.
#    The data is read from a text file
#    and stored in a Python list.

class TxtFileModel(QAbstractListModel):

    def __init__(self, source, parent=None):

        super().__init__(parent)

        self.txt_data = []
        self.header = 'Clients'
        with open(source) as txt_file:
            for line in txt_file:
                self.txt_data.append(line.strip())

```

```
    # 2. Implement the rowCount() method

    def rowCount(self, parent) -> int:
        return len(self.txt_data)

    # 3. Implement the data() method

    def data(self, index, role=Qt.ItemDataRole.DisplayRole) -> object|None:
        if role == Qt.ItemDataRole.DisplayRole:
            return self.txt_data[index.row()]
        return None

    # 4. Optionally, implement the headerData() method
    #    QListView does not have a header
    #    so this is never executed!

    def headerData(self, section, orientation, role) -> object | None:
        if orientation == Qt.Orientation.Horizontal:
            if role == Qt.ItemDataRole.DisplayRole:
                return self.header
```

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QListView, QVBoxLayout)
from PySide6.QtTest import QAbstractItemModelTester
from txtfilemodel import TxtFileModel


class Window(QWidget):

    def __init__(self):

        super().__init__()
        layout = QVBoxLayout()
        self.setLayout(layout)

        model = TxtFileModel('data.txt')
        QAbstractItemModelTester(model)
        view = QListView()
        view.setModel(model)
        layout.addWidget(view)


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```

1. Subclass `QAbstractListModel` and provide access to the data source. Here, read the text file entirely in `__init__()` and store lines in a Python list (`txt_data`), where each element is a single string (a client's name and profession). For other types of data sources you can retrieve data dynamically instead.
2. Implement `rowCount()`. View classes call this method to determine the model's length. Here, return `len(self.txt_data)`. Qt list models default to one column.
3. Implement `data()`. This returns model data for a given index and role. Use `index.row()` to access `txt_data`. Return data only for `DisplayRole` (the text for display); return `None` otherwise.
4. Optionally, implement `headerData()` for row or column headers. Here, the single column header is 'Clients'. `QListView` does not display headers but `QTableView` would.



`QModelIndex`:

This class helps views locate model items. Qt models are table-based - think of the model in this example as a one-column table with `n` rows. Use `row()` and `column()` to reference cells.

`QAbstractItemModelTester`

This class aids model development. Pass your model instance to its constructor - it logs implementation errors to the console, helping catch issues early.

This demo is read-only; next sections cover editable and resizable models.

22.2 Editable List Model

As shown in the read-only list model example, creating a basic list model requires implementing at least `rowCount()` and `data()`. To make it editable, add two more methods:

- `setData(index, value, role)`: Sets the data for the given role (typically `EditRole`) and index, and returns `True` if successful; otherwise returns `False`. If set successfully, emit the `dataChanged()` signal.
- `flags(index)`: Returns the item flags for the index. The base implementation enables and selects items. To allow editing, add `Qt.ItemFlags.ItemIsEditable`.

You have implemented a read-only model of a company clients backed by a text file ('data.txt') and you need to make it editable.

To create an editable list model:

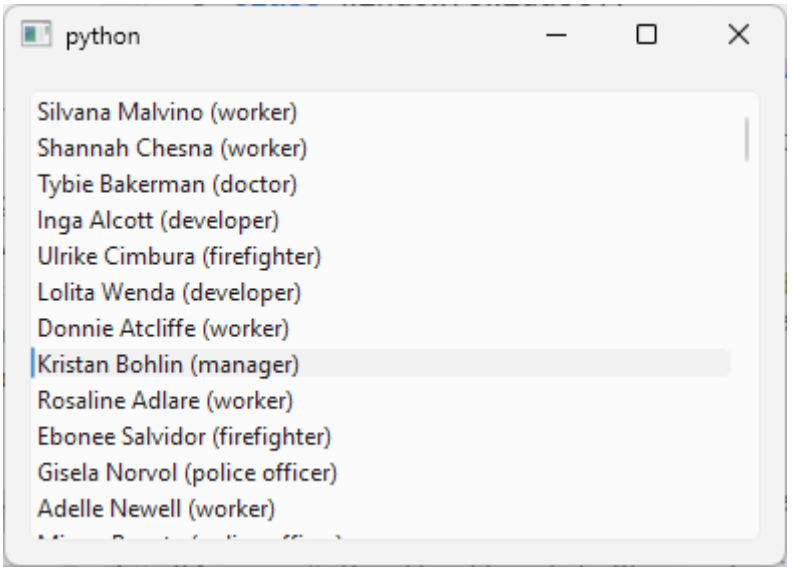
```
from PySide6.QtCore import QAbstractListModel, Qt

# 1. Create a QAbstractListModel subclass
#    and make the data available to it.

class TxtFileModel(QAbstractListModel):

    def __init__(self, source, parent=None):

        super().__init__(parent)

        self.txt_data = []
        self.header = 'Clients'
        with open(source) as txt_file:
            for line in txt_file:
                self.txt_data.append(line.strip())
```

```
17
18      # 2. Implement the rowCount() and data() methods
19
20      def rowCount(self, parent) -> int:
21          return len(self.txt_data)
22
23      def data(self, index, role=Qt.ItemDataRole.DisplayRole) -> object|None:
24          if role in [Qt.ItemDataRole.DisplayRole, Qt.ItemDataRole.EditRole]:
25              return self.txt_data[index.row()]
26          return None
27
28      # 3. Implement the setData() method
29
30      def setData(self, index, value, role) -> bool:
31          if not index.isValid():
32              return False
33          if role == Qt.ItemDataRole.EditRole:
34              if self.txt_data[index.row()] != value:
35                  self.txt_data[index.row()] = value
36                  self.dataChanged.emit(index, index, [role])
37                  return True
38              return False
39          return False
40
41      # 4. Implement the flags() method
42
43      def flags(self, index) -> Qt.ItemFlags:
44          return super().flags(index) | Qt.ItemFlags.ItemIsEditable
45
46      def headerData(self, section, orientation, role) -> object | None:
47          if orientation == Qt.Orientation.Horizontal:
48              if role == Qt.ItemDataRole.DisplayRole:
49                  return self.header
```

1. Create a `QAbstractListModel` subclass and the access to the data source.
2. Implement the `rowCount()` and `data()` methods just as you did for the read-only model.
3. Implement the `setData()` method. `setData()` accepts three arguments: `index`, `value` and `role`. In the method we check if the role is equal to `Qt.ItemDataRole.EditRole` and, if it is, we set the model data for the `index.row()` to `value` using `self.txt_data[index.row()] = value`. If the data is set successfully we emit the `dataChanged` signal and the method returns `True`. Otherwise it returns `False`.
4. Implement the `flags()` method. In this method we signal to views that the model data is editable by adding `Qt.ItemFlags.ItemIsEditable` to

the flags. Note that we can make the model items editable selectively by using the `index` parameter. In the example we ignore `index` which means that all the model items are editable.

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QListView, QVBoxLayout)
from PySide6.QtTest import QAbstractItemModelTester
from txtfilemodel import TxtFileModel


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)
        layout = QVBoxLayout()
        self.setLayout(layout)

        model = TxtFileModel('data.txt')
        QAbstractItemModelTester(model)
        view = QListView()
        view.setModel(model)
        layout.addWidget(view)

        model.dataChanged.connect(self.on_data_changed)

    # Handle the dataChanged signals

    def on_data_changed(self, topLeft, bottomRight, roles):
        print('Model changed, row:', topLeft.row())


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()
    sys.exit(app.exec())
```



Now if you double-click any of the list view lines you are able to edit it and the changes are saved in the model (ie. to the `TxtFileModel.txt_data` list and signaled by the `dataChanged` signal. You can also implement the logic to update the text file from the `txt_data` values which we omit in this example.

22.3 Editable List Model with Data-Widget Mapping

The `QDataWidgetMapper` class lets you map a data model row (or column) to a set of widgets, making them data-aware. When the model's current index changes, mapped widgets are updated with data from the model. This is useful for creating forms enhancing user experience in viewing and editing data.

You have an editable model of a list of company clients. Editing is enabled by double-clicking a client row in the list view. Your task is to make editing more user-friendly.

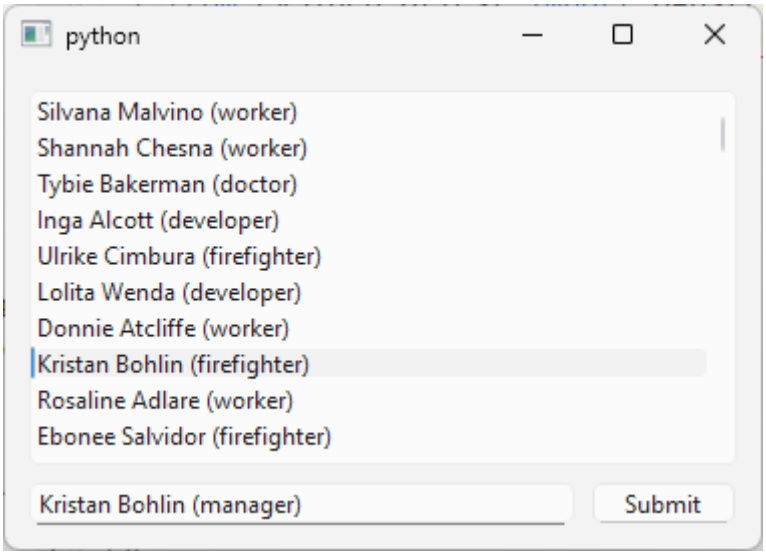
```python
import csv
from PySide6.QtCore import QAbstractListModel, QModelIndex, Qt

# 1. Create the model class

class TxtFileModel(QAbstractListModel):

    def __init__(self, source, parent=None):

        super().__init__(parent)

        self.txt_data = []
        self.header = 'Clients'
        with open(source) as txt_file:
            for line in txt_file:
                self.txt_data.append(line.strip())

    def rowCount(self, parent=QModelIndex()) -> int:
        return len(self.txt_data)

    def data(self, index, role=Qt.ItemDataRole.DisplayRole) -> object|None:
        if role == Qt.ItemDataRole.DisplayRole \
        or role == Qt.ItemDataRole.EditRole:
            return self.txt_data[index.row()]

    def setData(self, index, value, role) -> bool:
        if not index.isValid():
            return False
        if role == Qt.ItemDataRole.EditRole:
            if self.txt_data[index.row()] != value:
                self.txt_data[index.row()] = value
                self.dataChanged.emit(index, index, [role])
                return True
            return False
        return False

    def flags(self, index) -> Qt.ItemFlags:
        return super().flags(index) | Qt.ItemFlags.ItemIsEditable

    def headerData(self, section, orientation, role) -> object|None:
        if orientation == Qt.Orientation.Horizontal:
            if role == Qt.ItemDataRole.DisplayRole:
                return self.header
```

1. Create a `QAbstractListModel` subclass to represent your model. Then, in the main window class:

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QListView, QLineEdit, QPushButton,
    QHBoxLayout, QVBoxLayout, QDataWidgetMapper)
from PySide6.QtTest import QAbstractItemModelTester
from txtfilemodel import TxtFileModel


class Window(QWidget):

    def __init__(self, parent=None):

        super().__init__(parent)

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.model = TxtFileModel('data.txt')
        QAbstractItemModelTester(self.model)

        self.view = QListView()
        self.view.setModel(self.model)
        self.view.selectionModel().currentChanged.connect(
            self.sync_model_with_mapper)

        # 2. Create the widgets for displaying
        #    and editing the data

        self.lineedit = QLineEdit()
        self.lineedit.returnPressed.connect(self.submit_new_value)
        self.view.activated.connect(self.lineedit.setFocus)

        self.submit_button = QPushButton('Submit')
        self.submit_button.clicked.connect(self.submit_new_value)

        # 3. Create the mapper object and set its model

        self.mapper = QDataWidgetMapper()
        self.mapper.setModel(self.model)

        # 4. Add the mappings

        self.mapper.addMapping(self.lineedit, 0)

        self.mapper.setSubmitPolicy(
            QDataWidgetMapper.SubmitPolicy.ManualSubmit)
        self.mapper.toFirst()

        horizontal_layout = QHBoxLayout()
        horizontal_layout.addWidget(self.lineedit)
```

```
51          horizontal_layout.addWidget(self.submit_button)
52
53          layout.addWidget(self.view)
54          layout.addLayout(horizontal_layout)
55
56      # 5. Synchronize the model with the mapper
57
58      def sync_model_with_mapper(self, current, previous):
59          self.mapper.setCurrentIndex(current.row())
60
61      def submit_new_value(self):
62          self.mapper.submit()
63          self.view.setFocus()
64
65
66  if __name__ == '__main__':
67
68      app = QApplication(sys.argv)
69      main_window = Window()
70      main_window.show()
71      sys.exit(app.exec())
```

2. Create your model object and the widgets for displaying and editing your model data.
3. Create the mapper object and use `QDataWidgetMapper.setModel()` to connect your model to it.
4. Use `QDataWidgetMapper.addMapping()` to map your model columns with the widgets. The example model has only one column, which we map to a `QLineEdit` widget.
5. Synchronize the view's current item with the model's current index so both update when the user changes the view's selection.



In the example, we use manual submit policy, updating the model via a 'Submit' button. This syncs the line edit with the current view item, allowing easy updates by editing the line edit value and clicking 'Submit'.

22.4 Resizable List Model

For a basic `QAbstractListModel` subclass, you need to implement at least two methods: `rowCount()` and `data()`. To make the model editable, you need to implement two more: `setData()` and `flags()`. To be able to add or remove rows, you need to implement `insertRows()` and `removeRows()`.

You have an editable model of a list of company clients. You need to enable the user to insert or remove clients from it.

To make a resizable `QAbstractListModel` subclass:

```python
import csv
from PySide6.QtCore import QAbstractListModel, QModelIndex, Qt

# 1. Create a QAbstractListModel subclass

class TxtFileModel(QAbstractListModel):

    def __init__(self, source, parent=None):

        super().__init__(parent)

        self.txt_data = []
        self.header = 'Clients'
        with open(source) as txt_file:
            for line in txt_file:
                self.txt_data.append(line.strip())

    def rowCount(self, parent=QModelIndex()) -> int:
        return len(self.txt_data)

    def data(self, index, role=Qt.ItemDataRole.DisplayRole) -> object|None:
        if role == Qt.ItemDataRole.DisplayRole \
        or role == Qt.ItemDataRole.EditRole:
            return self.txt_data[index.row()]

    def setData(self, index, value, role) -> bool:
        if not index.isValid():
            return False
        if role == Qt.ItemDataRole.EditRole:
            if self.txt_data[index.row()] != value:
                self.txt_data[index.row()] = value
                self.dataChanged.emit(index, index, [role])
                return True
            return False
        return False

    def flags(self, index) -> bool:
        return super().flags(index) | Qt.ItemFlags.ItemIsEditable

    # 2. Implement the insertRows() method

    def insertRows(self, row, count, parent=QModelIndex()):
        if 0 <= row <= self.rowCount():
            self.beginInsertRows(parent, row, row )
            self.txt_data.insert(row, '<insert row data>')
            self.endInsertRows()
            return True
        return False

    # 3. Implement the removeRows() method
```

```
51
52      def removeRows(self, row, count, parent=QModelIndex()):
53          if 0 <= row < len(self.txt_data):
54              self.beginRemoveRows(parent, row, row)
55              self.txt_data[row:row + 1] = []
56              self.endRemoveRows()
57              return True
58          return False
59
60      # QListView does not have a header
61      # so this is never executed!
62
63      def headerData(self, section, orientation, role):
64          if orientation == Qt.Orientation.Horizontal:
65              if role == Qt.ItemDataRole.DisplayRole:
66                  return self.header
```

1. Create a subclass of the `QAbstractListModel` class. As in previous examples, read the data from a text file and store it in a Python list named `self.txt_data`.
2. Implement the `insertRows()` method. For simplicity, the example inserts rows filled with template text ("<insert row data>"). Guard the data insertion with `beginInsertRows()` (to signal connected views that rows are about to be inserted) and `endInsertRows()`. This pair of methods ensures that views remain in a valid state. `beginInsertRows()` takes three arguments: `parent` (an invalid `QModelIndex()` in our case), `first` (the starting row number post-insertion) and `last` (the ending row number post-insertion). The method returns `True` on success or `False` otherwise.
3. Implement the `removeRows()` method. This removes rows from the `self.txt_data`, enclosed by `beginRemoveRows()` and `endRemoveRows()`, and returns `True` on success or `False` otherwise.

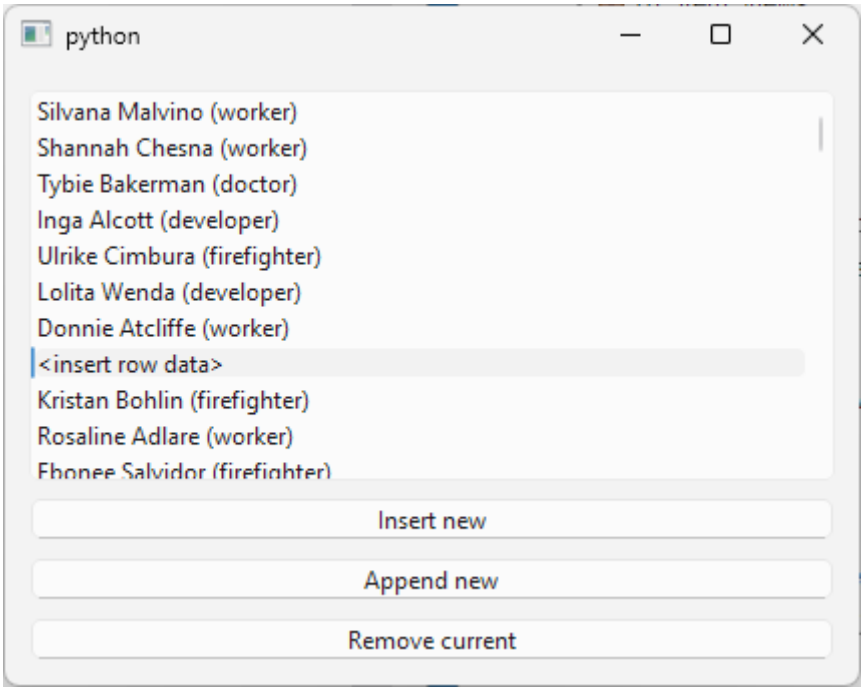
Then, in your main class

```
import sys
from PySide6.QtWidgets import (QApplication,
    QWidget, QListView, QLineEdit, QPushButton,
    QHBoxLayout, QVBoxLayout, QDataWidgetMapper)
from PySide6.QtTest import QAbstractItemModelTester
from txtfilemodel import TxtFileModel


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.model = TxtFileModel('data.txt')
        QAbstractItemModelTester(self.model)
        self.model.rowsInserted.connect(self.on_rows_inserted)

        self.view = QListView()
        self.view.setModel(self.model)
        self.view.selectionModel().currentChanged.connect(
            self.on_current_changed)

        self.lineedit = QLineEdit()
        self.lineedit.returnPressed.connect(self.submit_new_value)
        self.view.activated.connect(self.lineedit.setFocus)

        self.mapper = QDataWidgetMapper()
        self.mapper.setModel(self.model)
        self.mapper.setSubmitPolicy(
            QDataWidgetMapper.SubmitPolicy.ManualSubmit)
        self.mapper.addMapping(self.lineedit, 0)
        self.mapper.toFirst()

        self.submit_button = QPushButton('Submit')
        self.submit_button.clicked.connect(self.submit_new_value)

        self.insert_button = QPushButton('Insert new')
        self.insert_button.clicked.connect(self.on_insert)

        self.append_button = QPushButton('Append new')
        self.append_button.clicked.connect(self.on_append)

        self.remove_button = QPushButton('Remove current')
        self.remove_button.clicked.connect(self.on_remove)

        input_layout = QVBoxLayout()
```

```
51          input_layout.addWidget(self.lineedit)
52          input_layout.addWidget(self.submit_button)
53
54          buttons_layout = QVBoxLayout()
55          buttons_layout.addWidget(self.insert_button)
56          buttons_layout.addWidget(self.append_button)
57          buttons_layout.addWidget(self.remove_button)
58
59          controls_layout = QHBoxLayout()
60          controls_layout.addLayout(input_layout)
61          controls_layout.addLayout(buttons_layout)
62
63          layout.addWidget(self.view)
64          layout.addLayout(controls_layout)
65
66      def on_current_changed(self, current, previous):
67          self.mapper.setCurrentIndex(current.row())
68
69      def submit_new_value(self):
70          self.mapper.submit()
71          self.view.setFocus()
72
73      def on_insert(self):
74          row = self.view.selectionModel().currentIndex().row()
75          if self.model.insertRows(row, 1):
76              self.mapper.setCurrentIndex(row)
77              self.lineedit.setFocus()
78
79      def on_append(self):
80          row = self.model.rowCount()
81          self.model.insertRows(row, 1)
82          index = self.model.index(row, 0)
83          self.view.scrollTo(index)
84          self.lineedit.setFocus()
85
86      def on_remove(self):
87          index = self.view.currentIndex()
88          self.model.removeRows(index.row(), 1)
89
90      def on_rows_inserted(self, parent, first, last):
91          index = self.model.index(first, 0)
92          if index.isValid():
93              self.view.setCurrentIndex(index)
94              self.lineedit.setFocus()
95
96
97  if __name__ == '__main__':
98
99      app = QApplication(sys.argv)
100     main_window = Window()
```

```
101     main_window.show()
102     sys.exit(app.exec())
```

4. Create three `QPushButtons`: `insert_button`, `append_button` and `remove_button`. Handle the insert button's `clicked()` signal with a slot named `on_insert()` calling `insertRow()` to insert a single row by invoking `insertRows()`. Similarly, handle the append button's `clicked()` signal with `on_append()` to insert a row at the end.



23. Model-View Programming with `QAbstractTableModel`

24. Model-View Programming with `QAbstractItemModel`

25. Model-View Programming - Delegates

26. Model-View Programming - Sorting, Filtering and Selection

27. Multithreading - `moveToThread`

Threads of execution let you execute your code concurrently while sharing the program memory and other resources. There are primary two use cases for threads:

- Accelerate processing by utilizing multiple processor cores,
- Maintain GUI responsiveness by offoading long-running tasks to background threads.

In the following examples, we focus on the second use case. First, however,let's demonstrate the issue by showing a non-responsive Qt GUI.

27.1 Blocking the Qt GUI: How Not to Do It

You need to search the filesystem for a file and report progress.

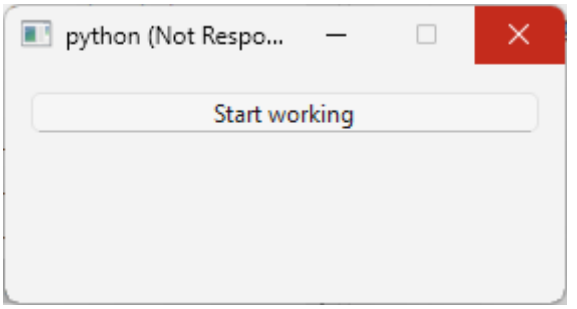
```
import os
import sys
from PySide6.QtCore import QObject, Signal, Slot
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QLabel, QVBoxLayout)


class Task(QObject):

    progress = Signal(str)

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

    # 1. Create a long running task
    #    We search for a file using os.walk()
    #    using a tight (blocking) for loop.
    #    The loop blocks the Qt event loop
```

```
19      #    so no signals are send or events
20      #    processed until the loop exits.
21      #    This effectively freezes the Gui.
22
23      @Slot()
24      def do_work(self):
25          path = os.path.abspath('.').split(os.path.sep)[0] + os.path.sep
26          name = 'bogus'
27          for root, _, files in os.walk(path):
28              self.progress.emit(root)
29              if name in files:
30                  print(os.path.join(root, name))
31
32
33  class Window(QWidget):
34
35      def __init__(self):
36
37          super().__init__()
38
39          layout = QVBoxLayout()
40          self.setLayout(layout)
41
42          # 2. Create a push button
43
44          self.button = QPushButton('Start working')
45          self.label = QLabel()
46
47          self.button.clicked.connect(self.do_work)
48
49          layout.addWidget(self.button)
50          layout.addWidget(self.label)
51
52      # 3. Execute the long running task.
53
54      def do_work(self):
55          self.task = Task()
56          self.task.progress.connect(self.label.setText)
57          self.task.do_work()
58
59
60  if __name__ == '__main__':
61
62      app = QApplication(sys.argv)
63      main_window = Window()
64      main_window.show()
65      sys.exit(app.exec())
```

1. Create the task. In the example, we use `os.walk()` within a `for` loop

to search the file system for a non-existent file. We attempt to report progress after each iteration using a custom Qt signal named `progress`. This fails because the loop blocks the Qt event loop, preventing the signals from being emitted or events from being processed until the loop completes. Since the loop runs in the main application thread (also known as the GUI thread), the entire application becomes unresponsive - it freezes and you cannot even close it.

2. Create a push button.
3. Create a slot that starts the long-running task when the push button is clicked.



27.2 A Minimal Working Example

The previous example illustrates how a PySide6 GUI can become unresponsive. Now let's explore using Qt threads to run long tasks in the background while keeping the GUI responsive.

You need to create a minimal working Qt multithreading application.

```python
from PySide6.QtCore import QObject, Signal, Slot

# 1. Create the worker_obj class

class Worker(QObject):

    finished = Signal()
    error = Signal(str)

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

    # This method to be executed

    @Slot()
    def process(self):
        print('Hello World')
        self.finished.emit()
```

1. Create a `QObject` subclass containing the slot/method to execute in a background thread (i.e., a thread other than the GUI thread). In the example, the slot is named `process()`; it prints a message and emits a custom signal named `finished()` before returning. The `Worker` class declares an `error()` signal, which could be emitted under error conditions during `process()` execution.

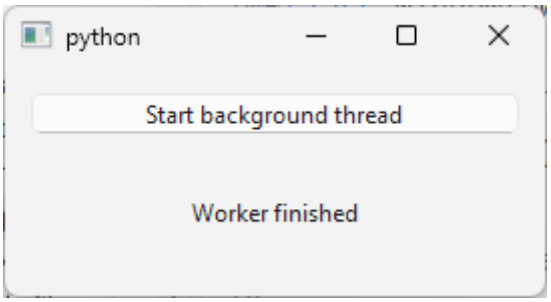
```python
#
↪ https://mayaposch.wordpress.com/2011/11/01/how-to-really-truly-use-qthreads-the-full-exp

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


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

```

```
19          button = QPushButton('Start background thread')
20          button.clicked.connect(self.on_button_clicked)
21
22          self.label = QLabel()
23          self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
24
25          layout.addWidget(button)
26          layout.addWidget(self.label)
27
28      @Slot()
29      def on_button_clicked(self):
30
31          # 2. Create the thread object
32
33          self.background_thread = QThread()
34
35          # 3. Create the worker_obj and move it to the thread
36
37          self.worker_obj = Worker()
38          self.worker_obj.moveToThread(self.background_thread)
39
40          self.worker_obj.finished.connect(self.on_finished)
41
42          # 4. Connect the signals and the slots
43
44          self.worker_obj.error.connect(self.on_error)
45          self.background_thread.started.connect(self.worker_obj.process)
46          self.worker_obj.finished.connect(self.background_thread.quit)
47          self.worker_obj.finished.connect(self.worker_obj.deleteLater)
48          self.background_thread.finished.connect(self.background_thread.
            ↪  deleteLater)
49
50          # 5. Start the thread
51
52          self.background_thread.start()
53
54      @Slot()
55      def on_finished(self):
56          self.label.setText('Worker finished')
57
58      @Slot()
59      def on_error(self, message):
60          print(message)
61
62
63  if __name__ == '__main__':
64
65      app = QApplication(sys.argv)
66
67      main_window = Window()
```

```
68      main_window.show()
69
70      sys.exit(app.exec())
```

Then, in the main window class:

2. Create a `QThread` object named `background_thread` (avoid naming it `thread`, as that name is already used). Use `self` to make it the member of the main window, ensuring it remains in scope while running.
3. Create a `Worker` object and move it to `background_thread` using `QObject.moveToThread()`. Now, `Worker.process()` will execute in `background_thread`.
4. Connect the appropriate signals and slots:
    - Connect `background_thread.started()` to `worker.process()`. This executes `Worker.process()` when the background thread starts.
    - Connect `Worker.finished()` to `background_thread.quit()`. This quits the background thread when `Worker.process()` returns.
    - Connect `Worker.finished()` to `Worker.deleteLater()` method. This deletes the worker object some time after it emits `finished()`.
    - Connect `background_thread.finished()` to `background_thread.deleteLater()`. This deletes the background thread some time after it finishes.
5. Start `background_thread` using `QThread.start()`.



With these steps, `process()` runs upon thread start, the thread quits when the method returns, and both the worker and thread object are destroyed - all while keeping the GUI responsive.

27.3 Walking the Filesystem

The `moveToThread()` template provides a basic structure, but the worker does nothing but print a message. For a more practical example, let's have `Worker.process()` traverse the filesystem using `os.walk()` from the root. This operation can be time-consuming and would block the GUI if run on the main thread.

You need to use Qt multithreading to search the filesystem for a file and report progress.

```
import os
from PySide6.QtCore import QObject, QThread, Signal, Slot

# 1. Create the worker_obj class

class Worker(QObject):

    finished = Signal()
    progress = Signal(str)
    error = Signal(str)

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

    # This is the method we want to execute.
    # We are in a tight loop.

    @Slot()
    def process(self):
        path = os.path.abspath('.').split(os.path.sep)[0] + os.path.sep
        for root, _, _ in os.walk(path):
            if QThread.currentThread().isInterruptionRequested():
                return
            self.progress.emit(os.path.basename(root))
        self.finished.emit()
```

1. Create the worker class. The method, `process()` uses Python's `os.walk()` to traverse the filesystem. For each enumerated filesystem object, we emit a custom `progress` signal (added to `Worker` alongside `finished` and `error`). If the background thread receives an interruption

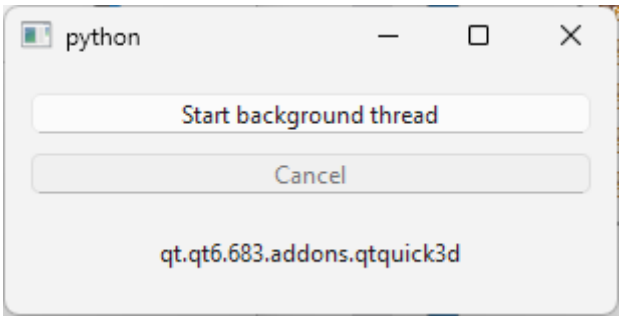
request, we return from `process()`, triggering the signal-slot chain to stop and delete both the worker and the background thread. Note how in `Worker.process()`, we access the current (background) thread via the static `QThread.currentThread()` method.

```
import sys
from PySide6.QtCore import QThread, Slot, Qt
from PySide6.QtWidgets import (QApplication, QPushButton,
    QLabel, QWidget, QVBoxLayout)
from worker import Worker

class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.start_button = QPushButton('Start background thread')
        self.start_button.clicked.connect(self.on_start_button_clicked)

        self.cancel_button = QPushButton('Cancel')
        self.cancel_button.clicked.connect(self.on_cancel_button_clicked)
        self.cancel_button.setDisabled(True)

        self.label = QLabel()
        self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        layout.addWidget(self.start_button)
        layout.addWidget(self.cancel_button)
        layout.addWidget(self.label)

    @Slot()
    def on_start_button_clicked(self):

        self.start_button.setDisabled(True)
        self.cancel_button.setEnabled(True)

        # 2. Create the thread

        self.background_thread = QThread()

        # 3. Create the worker_obj and move it to the thread

        self.worker_obj = Worker()
        self.worker_obj.moveToThread(self.background_thread)
```

```
44
45          self.worker_obj.finished.connect(self.on_finished)
46
47          # 4. Connect the appropriate signals to ensure
48          #    both the worker_obj and the thread are destroyed.
49
50          self.worker_obj.error.connect(self.on_error)
51          self.background_thread.started.connect(self.worker_obj.process)
52          self.worker_obj.finished.connect(self.background_thread.quit)
53          self.worker_obj.finished.connect(self.worker_obj.deleteLater)
54          self.background_thread.finished.connect(self.background_thread.
            ↪  deleteLater)
55
56          self.worker_obj.progress.connect(self.label.setText)
57
58          # 5. Start the thread.
59
60          self.background_thread.start()
61
62      # Stop the work by requesting interruption
63
64      @Slot()
65      def on_cancel_button_clicked(self):
66
67          self.start_button.setEnabled(True)
68          self.cancel_button.setDisabled(True)
69
70          if hasattr(self, 'background_thread'):
71              self.background_thread.requestInterruption()
72              self.background_thread.quit()
73              self.background_thread.wait()
74
75      @Slot()
76      def on_finished(self):
77          self.label.setText('Worker finished')
78
79      @Slot()
80      def on_error(self, message):
81          print(message)
82
83      # Make sure the thread is destroyed
84      # when the main window is closed.
85
86      def closeEvent(self, event):
87          try:
88              self.background_thread.requestInterruption()
89              self.background_thread.quit()
90              self.background_thread.wait()
91          except Exception as e:
92              print(e)
```

```
93
94
95  if __name__ == '__main__':
96
97      app = QApplication(sys.argv)
98
99      main_window = Window()
100     main_window.show()
101
102     sys.exit(app.exec())
```

2. In the main window class, create the thread object.
3. Create a `Worker` object and move it to the `QThread` using `QObject.moveToThread()`.
4. Use signals and slots to ensure proper creation and deletion: Starting the background thread triggers `Worker.process()`; `QObject.finished()` triggers both `QThread.quit()` and `Worker.deleteLater()`; `QThread.finished()` triggers `QThread.deleteLater()`.
5. Start the background thread. This occurs in `Window.on_start_button_clicked()` creating new `QThread` and `Worker` objects each time the start button is clicked.



We override `QWidget.closeEvent()` to interrupt the background thread, ensuring cleanup if the main window closes while the thread runs. The sequence, `QThread.requestInterruption()` + `QThread.quit()` + `QThread.wait()`, is also used in `Window.on_cancel_button_clicked()` for clean interruption.

27.4 Reusing the QThread object

In the prior examples a new background thread is created each time the task runs. However, the official QThread documentation example creates both thread and worker in the main class constructor. Let's try to replicate that.

Your task is to recreate the official Qt multithreading example in PySide6.

```
from PySide6.QtCore import QObject, Signal, Slot

# 1. Create the worker_obj class

class Worker(QObject):

    result_ready = Signal(str)

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

    @Slot()
    def do_work(self, parameter):
        print(parameter)
        self.result_ready.emit(parameter)
```

1. Create the worker class. The background method is `do_work()`, matching the `QThread` documentation.

```
# https://doc.qt.io/qt-6/qthread.html

import sys
from PySide6.QtCore import QThread, Slot, Signal, Qt
from PySide6.QtWidgets import (QApplication,
    QPushButton, QLabel, QWidget, QVBoxLayout)
from worker import Worker


class Controller(QWidget):

    operate = Signal(str)

```

```
14      def __init__(self):
15
16          super().__init__()
17
18          layout = QVBoxLayout()
19          self.setLayout(layout)
20
21          button = QPushButton('Start background thread')
22          button.clicked.connect(self.on_button_clicked)
23
24          self.label = QLabel()
25          self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
26
27          layout.addWidget(button)
28          layout.addWidget(self.label)
29
30          # 2. Create the thread object
31
32          self.worker_thread = QThread()
33
34          # 3. Create the worker_obj and move it to the thread
35
36          self.worker_obj = Worker()
37          self.worker_obj.moveToThread(self.worker_thread)
38
39          # 4. Connect the signals and the slots
40
41          self.worker_thread.finished.connect(
42              self.worker_obj.deleteLater)
43          self.operate.connect(self.worker_obj.do_work)
44          self.worker_obj.result_ready.connect(self.handle_results)
45
46          # 5. Start the thread
47
48          self.worker_thread.start()
49
50      # 6. On the button click emit the operate signal
51
52      @Slot()
53      def on_button_clicked(self):
54
55          self.operate.emit('Hello World')
56
57      @Slot()
58      def handle_results(self):
59          self.label.setText('Worker finished')
60
61      # 7. Quit the thread when the main window is closed
62
63      def closeEvent(self, event):
```

```
64          try:
65              self.worker_thread.quit()
66              self.worker_thread.wait()
67          except Exception as e:
68              print(e)
69          event.accept()
70
71
72  if __name__ == '__main__':
73
74      app = QApplication(sys.argv)
75
76      main_window = Controller()
77      main_window.show()
78
79      sys.exit(app.exec())
```

Then, in the main window class:

2. Create the worker thread.
3. Create the `Worker` object and move it to the the worker thread using `QObject.moveToThread()`.
4. Connect signals and slots: `QThread.finished()` to `QObject.deleteLater()` for worker deletion on thread finish; `Controller.operate()` to `Worker.do_work()` to start work via custom signal; `Worker.result_ready()` to `Controller.handle_result()` for handling results.
5. Start the worker thread. Steps 2-5 occur in `Controller.__init__()`, so the thread and the worker persist until the main window closes or explicit deletion.
6. On button click, emit the `operate()` signal to execute `Worker.do_work()`.
7. Override `QWidget.closeEvent()` to quit the thread using `QThread.quit()` and `QThread.wait()`.

27.5 Walking the Filesystem reusing the `QThread` Object

You need to use Qt multithreading to walk the filesistem but you reuse the same worker thread.

```python
import os
from PySide6.QtCore import (QObject, QMutex,
    QMutexLocker, Signal, Slot)

# 1. Create the worker_obj class

class Worker(QObject):

    result_ready = Signal()
    progress = Signal(str)

    def __init__(self, parent=None):
        super().__init__(parent)
        self.interruption_requested = False
        self.mutex = QMutex()

    @Slot()
    def do_work(self):

        self.interruption_requested = False

        path = os.path.abspath('.').split(os.path.sep)[0] + os.path.sep
        for root, _, _ in os.walk(path):
            with QMutexLocker(self.mutex):
                if self.interruption_requested:
                    self.progress.emit('Canceled')
                    self.result_ready.emit()
                    return
            self.progress.emit(os.path.basename(root))
        self.result_ready.emit()

    @Slot()
    def stop(self):
        with QMutexLocker(self.mutex):
            self.interruption_requested = True

    @Slot()
    def reset(self):
        with QMutexLocker(self.mutex):
            self.interruption_requested = False
```

1. Create the worker class. There are several differences from the first walk filesystem example: We use a boolean flag `Worker.interruption_requested` instead of `QThread.isInterruptionRequested()`; add `Worker.stop()` and `Worker.reset()` to toggle the flag; guard each use with `QMutexLOcker` and `QMutex` for thread safety.

```python
# https://doc.qt.io/qt-6/qthread.html

import sys

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


class Controller(QWidget):

    operate = Signal()

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.start_button = QPushButton('Start background thread')
        self.start_button.clicked.connect(self.on_start_button_clicked)

        self.cancel_button = QPushButton('Cancel')
        self.cancel_button.clicked.connect(self.on_cancel_button_clicked)
        self.cancel_button.setDisabled(True)

        self.label = QLabel()
        self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        layout.addWidget(self.start_button)
        layout.addWidget(self.cancel_button)
        layout.addWidget(self.label)

        # 2. Create the thread

        self.worker_thread = QThread()

        # 3. Create the worker_obj and move it to the thread

        self.worker_obj = Worker()
        self.worker_obj.moveToThread(self.worker_thread)

        # 4. Connect the signals with the slots

        self.worker_thread.finished.connect(self.worker_obj.deleteLater)
        self.operate.connect(self.worker_obj.do_work)
        self.worker_obj.result_ready.connect(self.handle_results)

```

```
51          self.worker_obj.progress.connect(self.label.setText)
52
53          # 5. Start the thread
54
55          self.worker_thread.start()
56
57      # 6. On the start button click emit the operate signal
58
59      @Slot()
60      def on_start_button_clicked(self):
61
62          self.start_button.setDisabled(True)
63          self.cancel_button.setEnabled(True)
64
65          self.worker_obj.reset()
66          self.operate.emit()
67
68      # 7. On the cancel button click stop the worker_obj
69
70      @Slot()
71      def on_cancel_button_clicked(self):
72
73          self.start_button.setEnabled(True)
74          self.cancel_button.setDisabled(True)
75          self.worker_obj.stop()
76
77      @Slot()
78      def handle_results(self):
79          self.label.setText('Worker finished')
80
81      # 8. Quit the thread when the main window is closed
82
83      def closeEvent(self, event):
84          try:
85              self.worker_obj.stop()
86              self.worker_thread.quit()
87              self.worker_thread.wait()
88          except Exception as e:
89              print(e)
90          event.accept()
91
92
93  if __name__ == '__main__':
94
95      app = QApplication(sys.argv)
96
97      main_window = Controller()
98      main_window.show()
99
100     sys.exit(app.exec())
```

2. In `Controller.__init__()` create the worker thread object.
3. Create the worker object and move it to the worker thread using `QObject.moveToThread()`.
4. Connect signals and slots. Main class `operate()` signal triggers `Worker.do_work()`.
5. Start the worker thread.
6. On start button click, reset the worker with `Worker.reset()` and emit `operate()`.
7. On cancel button click, stop the worker with `Worker.stop()`.
8. Quit the thread on main window close.

But why did we use a mutex-guarded boolean flag? `QThread.requestInterruption()` is one-time: once requested, `QThread.isInterruptionRequested()` stays `True` and it cannot be reset. A signal to toggle a flag would require `QApplication.processEvents()` in the blocking loop. DIrect flag setting is unsafe across threads, so `QMutex` and `QMutexLocker` are used.

27.6 Signals and Slots Across Threads

```
from PySide6.QtCore import QObject, QThread, Signal, Slot

class Worker(QObject):

    auto_signal = Signal()
    direct_signal = Signal()
    queued_signal = Signal()
    blocking_signal = Signal()

    @Slot()
    def auto_slot(self):
        print('Auto connection')
        print('In', QThread.currentThread().objectName(),
            ', Loop level', QThread.currentThread().loopLevel())

    @Slot()
    def direct_slot(self):
        print('Direct connection')
        print('In', QThread.currentThread().objectName(),
            ', Loop level', QThread.currentThread().loopLevel())
    @Slot()
    def queued_slot(self):
        print('Queued connection')
        print('In', QThread.currentThread().objectName(),
```

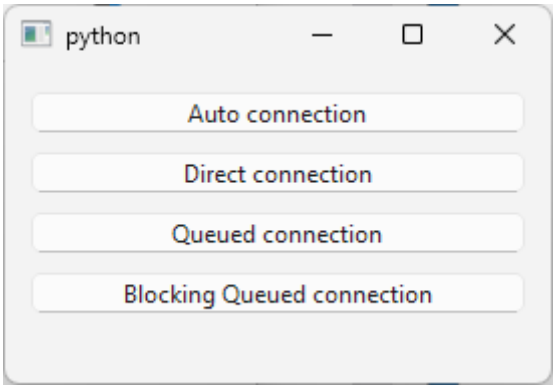
```
            ', Loop level', QThread.currentThread().loopLevel())

    @Slot()
    def blocking_slot(self):
        print('Blocking Queued connection')
        print('In', QThread.currentThread().objectName(),
            ', Loop level', QThread.currentThread().loopLevel())
        QThread.sleep(10)
```

```
# https://doc.qt.io/qt-6/qthread.html

import sys

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


class Window(QWidget):

    def __init__(self):

        super().__init__()

        QThread.currentThread().setObjectName('Main thread')

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button_auto = QPushButton('Auto connection')
        self.button_direct = QPushButton('Direct connection')
        self.button_queued = QPushButton('Queued connection')
        self.button_blocking = QPushButton('Blocking Queued connection')

        self.label = QLabel()
        self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        self.emitting_thread = QThread()
        self.emitting_thread.setObjectName('Emitting Thread')

        self.emitter = Worker()
        self.emitter.moveToThread(self.emitting_thread)

        self.receiving_thread = QThread()
        self.receiving_thread.setObjectName('Receiving Thread')

        self.receiver = Worker()
        self.receiver.moveToThread(self.receiving_thread)
```

```
41
42          self.emitter.auto_signal.connect(self.receiver.auto_slot,
            ↪  Qt.ConnectionType.AutoConnection)
43          self.emitter.direct_signal.connect(self.receiver.direct_slot,
            ↪  Qt.ConnectionType.DirectConnection)
44          self.emitter.queued_signal.connect(self.receiver.queued_slot,
            ↪  Qt.ConnectionType.QueuedConnection)
45          self.emitter.blocking_signal.connect(self.receiver.blocking_slot,
            ↪  Qt.ConnectionType.BlockingQueuedConnection)
46
47          self.button_auto.clicked.connect(self.emitter.auto_signal)
48          self.button_direct.clicked.connect(self.emitter.direct_signal)
49          self.button_queued.clicked.connect(self.emitter.queued_signal)
50          self.button_blocking.clicked.connect(self.emitter.blocking_signal)
51
52          self.emitting_thread.start()
53          self.receiving_thread.start()
54
55          layout.addWidget(self.button_auto)
56          layout.addWidget(self.button_direct)
57          layout.addWidget(self.button_queued)
58          layout.addWidget(self.button_blocking)
59          layout.addWidget(self.label)
60
61      @Slot()
62      def on_result_ready(self, result):
63          self.label.setText(result)
64
65      def closeEvent(self, event):
66          try:
67              self.emitting_thread.quit()
68              self.emitting_thread.wait()
69              self.receiving_thread.quit()
70              self.receiving_thread.wait()
71          except Exception as e:
72              print(e)
73          event.accept()
74
75
76  if __name__ == '__main__':
77
78      app = QApplication(sys.argv)
79      main_window = Window()
80      main_window.show()
81
82      sys.exit(app.exec())
```

Connection Type	Emitter Thread	Receiver Thread	Slot Invoked	Slot Executed In	Blocks	Unique
Auto	A	A	immediately when the signal is emitted	A		No
Auto	A	B	when control returns to the event loop of the receiver's thread	B		No
Direct	A	B	immediately when the signal is emitted	A		No
Queued	A	B	when control returns to the event loop of the receiver's thread	B		No

Connection Type	Emitter Thread	Receiver Thread	Slot Invoked	Slot Executed In	Blocks	Unique
Blocking Queued	A	B	when control returns to the event loop of the receiver's thread	B		No
Unique	A	A	immediately when the signal is emitted	A		Yes
Unique	A	B	when control returns to the event loop of the receiver's thread	B		Yes


python
Auto connection
Direct connection
Queued connection
Blocking Queued connection

28. Using a `QThread` subclass

Perhaps the most important thing to remember about `QThread` is that **a `QThread` object does not represent an operating system thread – it is a thread manager**. In practice, this means only the `QThread.run()` method executes in the new thread. All other `QThread` methods run in the thread that created the `QThread` object.

Another point when subclassing `QThread` is that **a `QThread` subclass does not start its event loop unless you explicitly call `QThread.exec()` in your `QThread.run()` implementation.**

28.1 A Minimal Example

There are two ways to use `QThread` to run a background task:

- Create a worker object and move it to a background thread using `QObject.moveToThread()`.
- Create a `QThread` subclass and override its `run()` method.

The examples from the previous chapter used `QObject.moveToThread()`. Now let's examine a minimal `QThread` subclass.

You need to provide a working multithreading application by subclassing `QThread`.

```python
from PySide6.QtCore import QThread, Signal, Slot

# 1. Create a QThread subclass
#    and override its run() method.
#    Add signals as needed.

class WorkerThread(QThread):

    result_ready = Signal(str)

    def __init__(self, parent=None):
        super().__init__(parent)
        print('Init in', QThread.currentThread().objectName(),
            ', Loop level', QThread.currentThread().loopLevel())

    def run(self):

        print('Running in', QThread.currentThread().objectName(),
            ', Loop level', QThread.currentThread().loopLevel())

        result = 'Hello World'
        print(result)
        self.result_ready.emit(result)
```

1. Create a `QThread` subclass (named `WorkerThread` in the example) and override its `run()` method. Add custom signals to the subclass and emit them from `run()` to communicate with the main thread.

```python
# https://doc.qt.io/qt-6/qthread.html

import sys

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


class Window(QWidget):

    def __init__(self):

        super().__init__()

        QThread.currentThread().setObjectName('Main thread')

```

```
        layout = QVBoxLayout()
        self.setLayout(layout)

        button = QPushButton('Start background thread')
        button.clicked.connect(self.start_work_in_a_thread)

        self.label = QLabel()
        self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        layout.addWidget(button)
        layout.addWidget(self.label)

    @Slot()
    def start_work_in_a_thread(self):

        print('Button click in', QThread.currentThread().objectName(),
            ', Loop level', QThread.currentThread().loopLevel())

        # 2. Create the WorkerThread object

        self.worker_thread = WorkerThread()
        self.worker_thread.setObjectName('Worker thread')

        # 3. Connect the signals with the slots

        self.worker_thread.result_ready.connect(self.on_result_ready)
        self.worker_thread.finished.connect(self.worker_thread.deleteLater)

        # 5. Start the worker thread

        self.worker_thread.start()

    # 4. Handle the worker thread signals

    @Slot()
    def on_result_ready(self, result):
        self.label.setText(result)


if __name__ == '__main__':

    app = QApplication(sys.argv)
    main_window = Window()
    main_window.show()

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

2. In your main window class, create a `WorkerThread` instance.

3. Connect the WorkerThread signals to main window slots. Also connect WorkerThread.finished() to WorkerThread.deleteLater() for clean thread exit.
4. Create slots in the main window to handle WorkerThread signals. Here, we set a QLabels text to "Hello, World" on WorkerThread.result_ready().
5. Start the worker thread.

When creating a plain QThread object (as in moveToThread() examples) the sequence is:

- Call QThread.start() from the main thread.
- The background QThread emits started().
- QThread.start() invokes QThread.run().
- QThread.run() calls QThread.exec().
- QThread.exec() enters the thread-local event loop, waiting for QThread.exit() or QThread.quit().
- Call QThread.exit() or QThread.quit() to stop.
- The event loop stops, emitting QThread.finished()
- QObjects with QObject.deleteLater() are deleted.

(Remember that in moveToThread() examples, we connected QThread.started() to a worker slot for immediate execution; and worker finished() to QThread.quit() for thread completion.)

When subclassing QThread, no local event loop unless QThread.exec() is called in run(). Without it, classes that need an event loop (e.g., QTimer, QTcpSocket, QProcess) won't work. However, the thread object will still be able to emit signals.

In this example, Window.start_work_in_a_thread() runs in the main thread (QThread.loopLevel() = 1, event loop active). WorkerThread.__init__() also runs in the main thread, while only WorkerThread.run() executes in the worker thread (QThread.loopLevel() = 0, no event loop).

28.2 Walking the Filesystem

Now let's walk the filesystem using a QThread subclass.

```python
import os
from PySide6.QtCore import QThread, Signal

# 1. Create a QThread subclass
#    and subclass its run() method.
#    Add signals as needed.

class WorkerThread(QThread):

    progress = Signal(str)

    def __init__(self, parent=None):
        super().__init__(parent)
        print('Init it', QThread.currentThread().objectName())

    def run(self):

        print('Running in: ',
            QThread.currentThread().objectName())
        print('event loop level: ',
            QThread.currentThread().loopLevel())

        path = os.path.abspath('.').split(os.path.sep)[0] + os.path.sep
        for root, _, _ in os.walk(path):
            if QThread.currentThread().isInterruptionRequested():
                return
            self.progress.emit(os.path.basename(root))
```

1. Create a `QThread` subclass named `WorkerThread` and override `run()`. In `run()`, print the current thread object name to confirm execution in the worker thread, then use `os.walk()` for recursive traversal. Emit `progress()` with each object's name as the argument.

```python
import sys
from PySide6.QtCore import QThread, Slot, Qt
from PySide6.QtWidgets import (QApplication, QPushButton,
    QLabel, QWidget, QVBoxLayout)
from worker_thread import WorkerThread


class Window(QWidget):

    def __init__(self):

        super().__init__()

```

```
14          QThread.currentThread().setObjectName('Main thread')
15
16          layout = QVBoxLayout()
17          self.setLayout(layout)
18
19          self.start_button = QPushButton('Start background thread')
20          self.start_button.clicked.connect(self.on_start_button_clicked)
21
22          self.cancel_button = QPushButton('Cancel')
23          self.cancel_button.clicked.connect(self.on_cancel_button_clicked)
24          self.cancel_button.setDisabled(True)
25
26          self.label = QLabel()
27          self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
28
29          layout.addWidget(self.start_button)
30          layout.addWidget(self.cancel_button)
31          layout.addWidget(self.label)
32
33      @Slot()
34      def on_start_button_clicked(self):
35
36          print('Main thread loop level',
37              QThread.currentThread().loopLevel())
38
39          # 2. Create the WorkerThread object
40
41          self.worker_thread = WorkerThread()
42          self.worker_thread.setObjectName('Worker thread')
43
44          # 3. Connect the signals with the slots
45
46          self.worker_thread.progress.connect(self.on_progress)
47          self.worker_thread.finished.connect(
48              self.worker_thread.deleteLater)
49
50          self.start_button.setDisabled(True)
51          self.cancel_button.setEnabled(True)
52
53          # 5. Start the worker thread
54
55          self.worker_thread.start()
56
57      # 4. Handle the signals
58
59      @Slot()
60      def on_cancel_button_clicked(self):
61
62          self.start_button.setEnabled(True)
63          self.cancel_button.setDisabled(True)
```

```
64
65          if hasattr(self, 'worker_thread'):
66              self.worker_thread.requestInterruption()
67              self.worker_thread.wait()
68
69      @Slot()
70      def on_progress(self, msg):
71          self.label.setText(msg)
72
73      # Make sure the thread is destroyed
74      # when the main window is closed.
75
76      def closeEvent(self, event):
77          try:
78              self.worker_thread.requestInterruption()
79              self.worker_thread.wait()
80          except Exception as e:
81              print(e)
82
83
84  if __name__ == '__main__':
85
86      app = QApplication(sys.argv)
87      main_window = Window()
88      main_window.show()
89
90      sys.exit(app.exec())
```

2. On start button click, create a `WorkerThread` object and set its name to "Worker thread".
3. Connect signals to slots: `WorkerThread.progress()` to `Window.on_progress()` for label updates; `WorkerThread.finished()` to `WorkerThread.deleteLater()` for cleanup.
4. Handle signals. On `progress()` update the label; On cancel request interruption with `QThread.requestInterruption()` and wait for finish.
5. Start the worker thread on each start button click.

Override `Qwidget.closeEvent()` to ensure thread deletion on window close. Note that we didn't need to call `QThread.quit()` but only `QThread.wait()` since no event loop runs.

29. Multithreading with `QThreadPool` and `QRunnable`

The `QThreadPool` class manages a collection of `QThreads` and is used with the `QRunnable` class.

29.1 A Minimal Example

You need to provide a working multithreading application using `QThreadPool` with `QRunnable`.

```
from PySide6.QtCore import QObject, QRunnable, Signal


class Signals(QObject):
    progress = Signal(str)
    error = Signal(str)

# 1. Create a QRunnable subclass
#    and implement its run() method.

class Runnable(QRunnable):

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

    # The run method will be executed
    # in the worker thread.

    def run(self):
        self.signals.progress.emit('Progress emitted')
        print('Hello World')
        self.signals.deleteLater()
```

1. Create a `QRunnable` subclass and implement its `run()` method with the code to execute in a background thread. `QRunnable` is not a `QObject` subclass, so it cannot have signals directly. You can easily work around this by creating a separate `QObject` subclass (named `Signals` here) and adding an instance to your `QRunnable`. In this example, `run()` emits `Signals.progress` and prints a message.

```
import sys

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


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        button = QPushButton('Start background thread')
        button.clicked.connect(self.on_button_clicked)

        self.label = QLabel()
        self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)

        layout.addWidget(button)
        layout.addWidget(self.label)

    # When the button is clicked:

    @Slot()
    def on_button_clicked(self):

        # 2. Create a Runnable object

        runnable = Runnable()

        runnable.signals.progress.connect(self.label.setText)
        runnable.signals.error.connect(self.on_error)

        # 3. Access the QThreadPool global instance
        #    and run the task.
```

```
41
42          QThreadPool.globalInstance().start(runnable)
43
44      @Slot()
45      def on_error(self, message):
46          print(message)
47
48
49  if __name__ == '__main__':
50
51      app = QApplication(sys.argv)
52
53      main_window = Window()
54      main_window.show()
55
56      sys.exit(app.exec())
```

2. In the main window, create a `Runnable` instance.
3. Use `QThreadPool.globalInstance()` to access the global thread pool and `QThreadPool.start()` to execute `Runnable.run()` in one of its threads.

29.2 Walking the Filesystem

Your task is to walk the filesystem using `QThreadPool` and `QRunnable`.

```
1   import os
2   from PySide6.QtCore import (QObject, QThread,
3       QRunnable, Signal, Slot)
4
5
6   class Signals(QObject):
7       progress = Signal(str)
8       error = Signal(str)
9
10  # 1. Create a QRunnable subclass
11  #    and implement its run() method
12
13  class Runnable(QRunnable):
```

```
14
15      def __init__(self, parent=None):
16          super().__init__(parent)
17          self.signals = Signals()
18          self.do_work = True
19
20      # Enumerate fs objects while self.do_work flag is True
21
22      def run(self):
23          path = os.path.abspath('.').split(os.path.sep)[0] + os.path.sep
24          for root, _, _ in os.walk(path):
25              if not self.do_work:
26                  return
27              self.signals.progress.emit(os.path.basename(root))
28              print(QThread.currentThread())
29
30      @Slot()
31      def on_cancel_emitted(self):
32          self.do_work = False
```

1. Create a `QRunnable` subclass and implement `run()`. Add a member for signals and a boolean flag `do_work` for interruption. `run()` uses `os.walk()` to enumerate filesystem objects, emitting `progress()` for each while `do_work` is `True`.

```
1  import sys
2  from PySide6.QtCore import QThreadPool, Slot, Signal, Qt
3  from PySide6.QtWidgets import (QApplication,
4      QPushButton, QLabel, QWidget, QVBoxLayout)
5  from runnable import Runnable
6
7
8  class Window(QWidget):
9
10     # 2. add a custom signal to be emitted
11     #    when we want to cancel the task.
12
13     cancel_runnable = Signal()
14
15     def __init__(self):
16
17         super().__init__()
18
19         layout = QVBoxLayout()
20         self.setLayout(layout)
21
```

```
22          self.start_button = QPushButton('Start background thread')
23          self.start_button.clicked.connect(self.on_start_button_clicked)
24
25          self.cancel_button = QPushButton('Cancel')
26          self.cancel_button.clicked.connect(self.on_cancel_button_clicked)
27          self.cancel_button.setDisabled(True)
28
29          self.label = QLabel()
30          self.label.setAlignment(Qt.AlignmentFlag.AlignCenter)
31
32          layout.addWidget(self.start_button)
33          layout.addWidget(self.cancel_button)
34          layout.addWidget(self.label)
35
36      @Slot()
37      def on_start_button_clicked(self):
38
39          # 3. Create a Runnable object
40          #    and connect the signals and the slots
41
42          runnable = Runnable()
43
44          runnable.signals.progress.connect(self.label.setText)
45          runnable.signals.error.connect(self.on_error)
46          self.cancel_runnable.connect(runnable.on_cancel_emitted)
47
48          # 4. Run the task.
49
50          QThreadPool.globalInstance().start(runnable)
51
52          self.start_button.setDisabled(True)
53          self.cancel_button.setEnabled(True)
54
55      @Slot()
56      def on_cancel_button_clicked(self):
57          self.cancel_runnable.emit()
58          self.start_button.setEnabled(True)
59          self.cancel_button.setDisabled(True)
60
61      @Slot()
62      def on_error(self, message):
63          print(message)
64
65      # Emit the cancel_runnable signal to interrupt
66      # the runnable on the main window close
67
68      def closeEvent(self, event):
69          try:
70              self.cancel_runnable.emit()
71          except Exception as e:
```

```
72              print(e)
73
74
75  if __name__ == '__main__':
76
77      app = QApplication(sys.argv)
78
79      main_window = Window()
80      main_window.show()
81
82      sys.exit(app.exec())
```

2. In the main window class, add a custom `cancel_runnable()` signal. Connect it to `Runnable.on_cancel_emitted()`, which sets `Runnable.do_work` to `False`.
3. In `on_start_button_clicked()` create a `Runnable` object and connects signals to slots: `progress` updates the label; on cancel, emit `cancel_runnable()` to set `do_work` `False`.
4. Access the global thread pool and run the task with `QThreadPool.start()`.

30. Thread Synchronization

31 More Timers

32. Signals & Slots Connection Types

32.1 Direct Connection

```
import sys
from PySide6.QtCore import Signal, Slot, QEvent
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)


class Window(QWidget):

    # 1. Create a custom signal

    custom_signal = Signal()

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = QPushButton('Click me!')
        layout.addWidget(self.button)

        self.button.clicked.connect(self.on_button_clicked)

        # 3. Connect the signal with the slots

        self.custom_signal.connect(self.on_custom_signal_1)
        self.custom_signal.connect(self.on_custom_signal_2)
        self.custom_signal.connect(self.on_custom_signal_3)

    # 4. Emit the signal

    @Slot()
    def on_button_clicked(self):
        print('on_button_clicked: Emitting custom signal\n')
        self.custom_signal.emit()
        print('on_button_clicked: Returning')

    # 2. Create the slots
```

```
40
41      @Slot()
42      def on_custom_signal_1(self):
43          print('First custom slot executed')
44
45      @Slot()
46      def on_custom_signal_2(self):
47          print('Second custom slot executed')
48
49      @Slot()
50      def on_custom_signal_3(self):
51          print('Third custom slot executed\n')
52
53      def event(self, event):
54          if event.type() == QEvent.Type.MetaCall:
55              print(f"Queued signal (MetaCallEvent) intercepted")
56          return QWidget.event(self, event)
57
58
59  if __name__ == '__main__':
60
61      app = QApplication(sys.argv)
62
63      main_window = Window()
64      main_window.show()
65
66      sys.exit(app.exec())
```

With `Qt.DirectConnection` the slot is invoked immediately when the signal is emitted. In the example we create a custom signal named `custom_signal`. We also create three slots, `on_custom_signal_1`, `on_custom_signal_2` and `on_custom_signal_3` and connect the signal with all three slots. We add a button to the main window and emit `custom_signal` when it is clicked.

The output is

```
1  on_button_clicked: Emitting custom signal
2
3  First custom slot executed
4  Second custom slot executed
5  Third custom slot executed
6
7  on_button_clicked: Returning
```

From the output we see that when we emit the signal in `on_button_clicked` the three slots are called sequentially and, only after `on_custom_`-

`signal_3` completes and returns, `on_button_clicked` continues execution. The code execution is synchronous.

When establishing the connections we didn't pass the optional `type` to the `Signal.connect()` method. `type` defaults to `Qt.AutoConnection` and since both the signal and the slots live in the main thread the connection becomes `Qt.DirectConnection`.

There is one more thing to note: The slot execution bypasses the Qt event loop. We reimplement `Window.event()` to print a message but it is never executed.

32.2 Queued Connection

We have the same setup as in the direct connection example: a single custom signal connected to three slots.

```
import sys
from PySide6.QtCore import Signal, Slot, QEvent, Qt
from PySide6.QtWidgets import (QApplication,
    QWidget, QPushButton, QVBoxLayout)


class Window(QWidget):

    custom_signal = Signal()

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = QPushButton('Click me!')
        layout.addWidget(self.button)

        self.button.clicked.connect(self.on_button_clicked)

        # 1. Force the queued connection

        self.custom_signal.connect(self.on_custom_signal_1,
            Qt.ConnectionType.QueuedConnection)
        self.custom_signal.connect(self.on_custom_signal_2,
            Qt.ConnectionType.QueuedConnection)
```

```
29          self.custom_signal.connect(self.on_custom_signal_3,
30              Qt.ConnectionType.QueuedConnection)
31
32      @Slot()
33      def on_button_clicked(self):
34          print('on_button_clicked: Emitting custom signal')
35          self.custom_signal.emit()
36          print('on_button_clicked: Returning\n')
37
38      @Slot()
39      def on_custom_signal_1(self):
40          print('First custom slot executed\n')
41
42      @Slot()
43      def on_custom_signal_2(self):
44          print('Second custom slot executed\n')
45
46      @Slot()
47      def on_custom_signal_3(self):
48          print('Third custom slot executed\n')
49
50      # 2. Intercept MetaCall events
51
52      def event(self, event):
53          if event.type() == QEvent.Type.MetaCall:
54              print(f"Queued signal (MetaCall event) intercepted")
55          return QWidget.event(self, event)
56
57
58  if __name__ == '__main__':
59
60      app = QApplication(sys.argv)
61
62      main_window = Window()
63      main_window.show()
64
65      sys.exit(app.exec())
```

But now the output is

```
on_button_clicked: Emitting custom signal
on_button_clicked: Returning

Queued signal (MetaCallEvent) intercepted
First custom slot executed

Queued signal (MetaCallEvent) intercepted
Second custom slot executed

Queued signal (MetaCallEvent) intercepted
Third custom slot executed
```

Now each slot execution is queued in the Qt event loop. After emitting the signal `on_button_clicked` continues execution and returns immediately, returning control to the event loop, allowing the slots to be executed. Each slot execution is wrapped in an asynchronous method invocation via `QMetaObject::invokeMethod()`.

All three slots are still executed in the receiver thread which happens to be the main thread.

32.3 Blocking Queued Connection

`Qt.BlockingQueuedConnection` behaves the same as `Qt::QueuedConnection`, except that the signalling thread blocks until the slot returns.

```python
from PySide6.QtCore import QObject, Signal, Slot, QThread

class Sender(QObject):

    operate = Signal()

    @Slot()
    def handle_results(self):
        print('Sender: handle_results started')
        QThread.sleep(5)
        print('Sender: handle_results finished after delay')
```

```
from PySide6.QtCore import QObject, Signal, Slot

class Receiver(QObject):

    result_ready = Signal()

    @Slot()
    def do_work(self):
        print('Receiver: Starting do_work')
        print('Receiver: Emitting result_ready')
        self.result_ready.emit()
        print('Receiver: After emit (this should delay if blocking)')
```

We have two worker objects: the `Sender` object signals the `Receiver` to `do_work`. `Receiver` in turn signals to the `Sender` that the work results are ready so that the `Sender` can `handle_results`.

```
import sys
from PySide6.QtCore import QThread, Slot, Qt
from PySide6.QtWidgets import (QApplication,
    QPushButton, QWidget, QVBoxLayout)
from sender import Sender
from receiver import Receiver


class Window(QWidget):

    def __init__(self):

        super().__init__()

        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = QPushButton('Start the task')
        self.button.clicked.connect(self.on_button_clicked)
        layout.addWidget(self.button)

        self.sender_thread = QThread()
        self.receiver_thread = QThread()

        self.sender_obj = Sender()
        self.receiver_obj = Receiver()

        self.sender_obj.moveToThread(self.sender_thread)
        self.receiver_obj.moveToThread(self.receiver_thread)

        self.sender_thread.finished.connect(
```

```
32              self.sender_obj.deleteLater)
33          self.receiver_thread.finished.connect(
34              self.receiver_obj.deleteLater)
35
36          self.sender_obj.operate.connect(self.receiver_obj.do_work)
37
38          self.receiver_obj.result_ready.connect(
39              self.sender_obj.handle_results,
40              Qt.ConnectionType.BlockingQueuedConnection)
41          '''
42          self.receiver_obj.result_ready.connect(
43              self.sender_obj.handle_results)
44          '''
45
46          self.sender_thread.start()
47          self.receiver_thread.start()
48
49      @Slot()
50      def on_button_clicked(self):
51          self.sender_obj.operate.emit()
52
53      def closeEvent(self, event):
54          try:
55              self.sender_thread.quit()
56              self.receiver_thread.quit()
57              self.sender_thread.wait()
58              self.receiver_thread.wait()
59          except Exception as e:
60              print(e)
61          event.accept()
62
63
64  if __name__ == '__main__':
65
66      app = QApplication(sys.argv)
67
68      main_window = Window()
69      main_window.show()
70
71      sys.exit(app.exec())
```

In the main window we connect the signals with the slots. If we use `Qt.BlockingQueuedConnection`:

```
self.receiver_obj.result_ready.connect(
    self.sender_obj.handle_results,
    Qt.ConnectionType.BlockingQueuedConnection)
```

the output is

```
Receiver: Starting do_work
Receiver: Emitting result_ready
Sender: handle_results started
Sender: handle_results finished after delay
Receiver: After emit (this should delay if blocking)
```

The `Receiver` thread is blocked for five seconds after it emits `results_ready` and only prints the last message after `Sender` finished handling the results.

If we used `QtQueuedConnection` instead

```
self.receiver_obj.result_ready.connect(
    self.sender_obj.handle_results)
```

the output is

```
Receiver: Starting do_work
Receiver: Emitting result_ready
Receiver: After emit (this should delay if blocking)
Sender: handle_results started
Sender: handle_results finished after delay
```

which means that `do_work` emits the `results_ready` signal and immediately continues its execution.

32.4 Unique Connection

Making your signal-slot connections unique is a good idea most of the time and `Qt.UniqueConnection` lets you make that explicit.

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


class Window(QWidget):

    def __init__(self):
        super().__init__()

        self.resize(300, 200)
        layout = QVBoxLayout()
        self.setLayout(layout)

        self.button = QPushButton('Click Me')
        self.label = QLabel('Button: 0 receivers')

        layout.addWidget(self.button)
        layout.addWidget(self.label)

        h_layout = QHBoxLayout()
        self.connect_btn = QPushButton('Connect signals')
        self.disconnect_btn = QPushButton('Disconnect signals')

        self.connect_btn.clicked.connect(
            self.on_connect_btn_clicked)
        self.disconnect_btn.clicked.connect(
            self.on_disconnect_btn_clicked)

        h_layout.addWidget(self.connect_btn)
        h_layout.addWidget(self.disconnect_btn)
        layout.addLayout(h_layout)

    def on_connect_btn_clicked(self):
        conn = self.button.clicked.connect(self.on_clicked,
            Qt.ConnectionType.UniqueConnection)
        if conn:
            print('Connection is valid')
        else:
            print('Connection is invalid')
        self.update_label()

    def on_disconnect_btn_clicked(self):
        self.button.clicked.disconnect(self.on_clicked)
        self.update_label()

    @Slot(bool)
    def on_clicked(self, checked):
        print('Button clicked')
```

```
51
52      def update_label(self):
53          count = self.button.receivers(SIGNAL('clicked(bool)'))
54          self.label.setText(f'Button: {count} receivers')
55
56
57  if __name__ == '__main__':
58
59      app = QApplication(sys.argv)
60      main_window = Window()
61      main_window.show()
62      sys.exit(app.exec())
```

The example starts without any connections made. We make the connection on clicking the `connect_btn` button but we use `QtUniqueConnection` so the connection is established only once. All subsequent attempts to connect `self.button.clicked` with `self.on_clicked` return invalid connection objects.The `label` text and the console output also reflect this.

```
1  Connection is valid
2  Connection is invalid
3  Connection is invalid
4  Connection is invalid
5  Connection is invalid
```

32.5 Single-Shot Connection

While `Qt.UniqueConnection` makes connections unique, `Qt.SingleShotConnection` makes them disposable: the connection is automatically broken when the signal is emitted.

```
1  import sys
2  from PySide6.QtCore import Slot, Qt, SIGNAL
3  from PySide6.QtWidgets import (QApplication, QWidget,
4      QVBoxLayout, QLabel, QPushButton)
5
6
7  class Window(QWidget):
8
9      def __init__(self):
10         super().__init__()
11
12         self.resize(300, 200)
```

```
13          layout = QVBoxLayout()
14          self.setLayout(layout)
15
16          self.button = QPushButton('Click Me')
17          self.connect_btn = QPushButton('Connect signals')
18          self.label = QLabel('Button: 0 receivers')
19
20          layout.addWidget(self.button)
21          layout.addWidget(self.connect_btn)
22          layout.addWidget(self.label)
23
24          self.connect_btn.clicked.connect(
25              self.on_connect_btn_clicked)
26
27      def on_connect_btn_clicked(self):
28          self.button.clicked.connect(self.on_clicked,
29              Qt.ConnectionType.SingleShotConnection)
30          self.update_label()
31
32      @Slot(bool)
33      def on_clicked(self, checked):
34          print('Button clicked')
35          self.update_label()
36
37      def update_label(self):
38          count = self.button.receivers(SIGNAL('clicked(bool)'))
39          self.label.setText(f'Button: {count} receivers')
40
41
42  if __name__ == '__main__':
43
44      app = QApplication(sys.argv)
45      main_window = Window()
46      main_window.show()
47      sys.exit(app.exec())
```

The example lets you accumulate one-shot connections by clicking the `Connect signals` button and their count is displayed in the label. Once you click the `Click Me` button all connections are broken and the `on_clicked` slot is executed as many times as there were connections.

33. Databases

34. Processes

35. State Machines