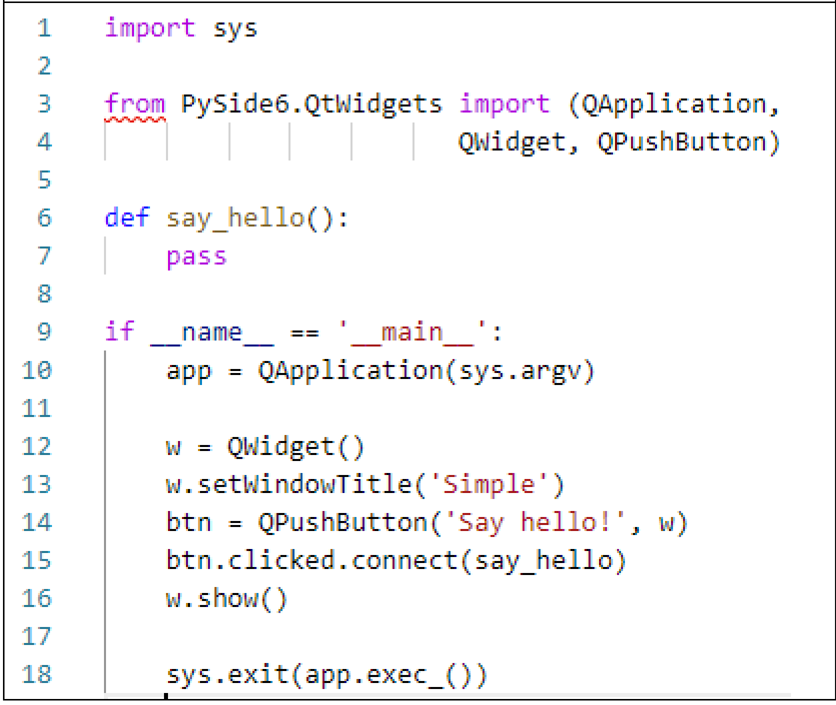
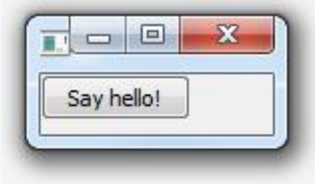
Software Engineering Lab #2
GUI Programming with Qt for Python

Prerequisites

Make sure that PySide6 have already been installed and are ready to use. See a simple PySide6 program below.

Program 2.1: Your first PySide6 program

Source Code	Output
	

```python
import sys

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

def say_hello():
    pass

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

    w = QWidget()
    w.setWindowTitle('Simple')
    btn = QPushButton('Say hello!', w)
    btn.clicked.connect(say_hello)
    w.show()

    sys.exit(app.exec_())
```

How this program works:

- First, we need to import the `sys` module to pass `sys.argv` list to create a `QApplication` object. The main function of a PySide6 program usually begins by creating a `QApplication` object.

```
app = QtGui.QApplication(sys.argv)
```

- After creating the application object, we want to show a button to the screen. To do that, we simply create a push button (`QPushButton`) object.
- To interact with the user when the button is clicked, we need to connect the clicked signal to an action (defines as a function or a method):

```
btn.clicked.connect(say_hello)
```

  This statement connects the `clicked` signal of the `button` to the function named `say_hello`. The program will print "Hello world!" every time the user clicks on the button (as defined in `say_hello`). We will learn more about signals and slots later.
- After completing a setup for a button behavior, we need to show the button to the screen (`w.show()`). Then, we need to run an event loop for our application (`app.exec_()`). This is the usual pattern of every GUI application. Once the event loop finished, we should terminate our program by returning the return code from an `exec_` method.

PySide6 Signals and Slots

Every GUI application is event-driven. An application reacts to different event types. Events are generated mainly by the user who interacts with an application. But they can also be generated by other means such as internet connection, window manager, and timer. In the event model, there are three components:

- event source
- event object
- event target

The **event source** is the object whose state is changed. This generates events. The **event object** encapsulates the state changes in the event source. The **event target** is the object that wants to be notified. Event source object delegates the task of handling an event to the event target.

The mechanisms used for event-driven programming in PySide6 are mainly signals and slots. Signals and slots are used for communication between objects. A **signal** is emitted when a particular event occurs. A **slot** can be any Python callable. A slot is called when a signal connected to it is emitted. The general syntax for signal connection has many forms (assuming that we import everything from the module `PySide6.QtCore`):

```
source.signalName[signature].connect(functionName)
source.signalName[signature].connect(target.methodName)
```

The `[signature]` part is optional and only needed when an ambiguity arises.

Example:
The example below illustrates how to define and connect signals and slots in PySide6:

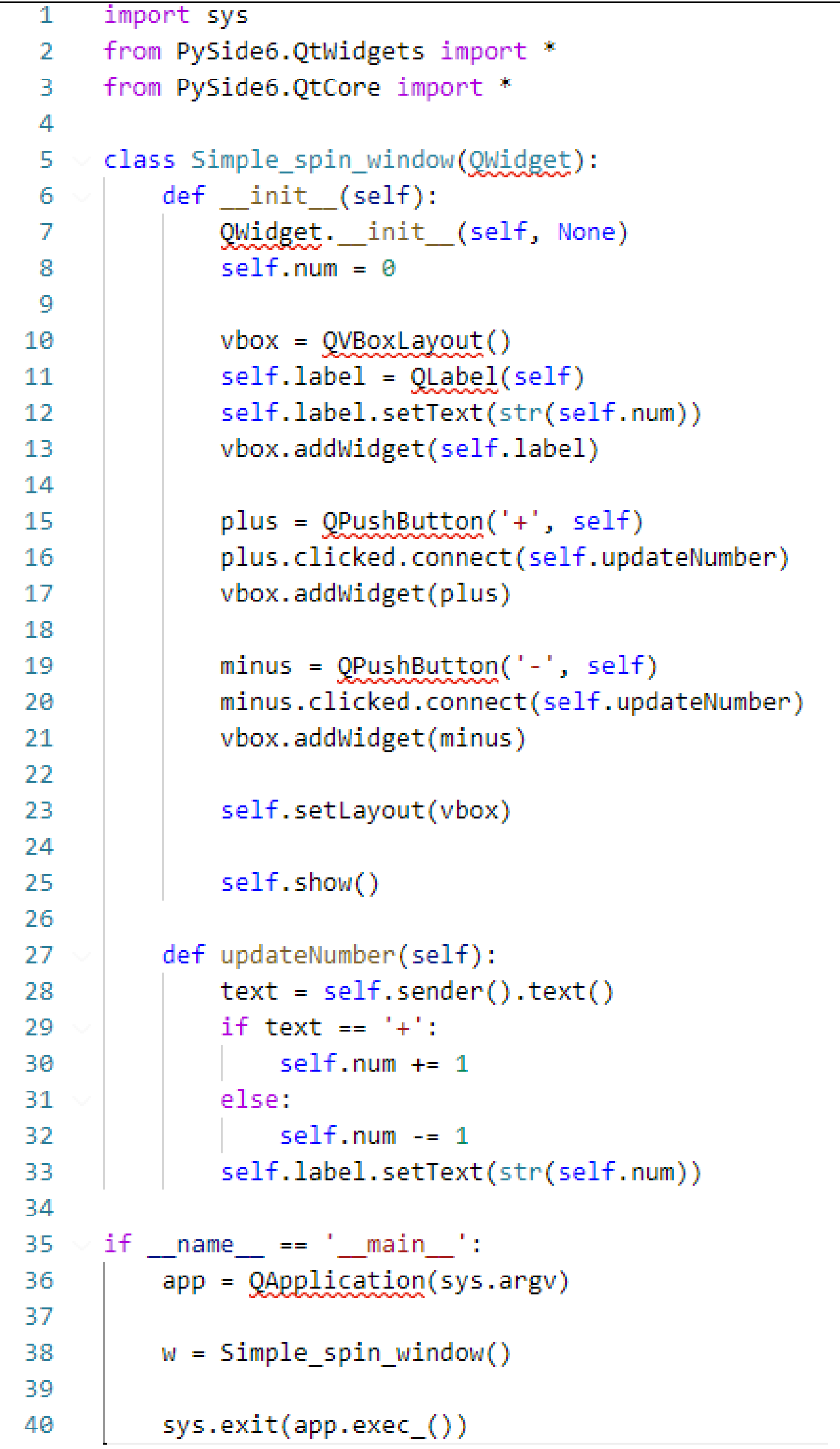
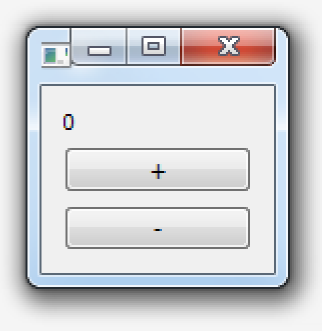
```
#!/usr/bin/env python

from PySide6.QtCore import QObject, Signal, Slot

# define a new slot that receives a string and has 'saySomeWords' as its name
@Slot
def saySomeWords(words):
    print(words)

class Communicate(QObject):
    # create a new signal on the fly and name it 'speak'
    speak = Signal(str)

someone = Communicate()
someone.speak.connect(saySomeWords)    # connect signal and slot
someone.speak.emit("Hello everybody!") # emit 'speak' signal
```

Note: See https://wiki.qt.io/Qt_for_Python_Signals_and_Slots for more information about signals and slots in PySide6.

GUI layout management in PySide6

PySide6 provides three layout managers: one for vertical layouts (QVBoxLayout), one for horizontal layouts (QHBoxLayout), and one for grid layouts (QGridLayout). Layouts can be nested, so you can create complex layouts from simple layouts. You can use `layout.addWidget(widget)` to add a widget to the layout. After you complete the layout setup, you have to set the layout to the parent widget by using `parent.setLayout(layout)` to make all widgets added to the layout appeared inside the parent widget.

Program 2.2: Spin a number up and down

Source Code	Output
	

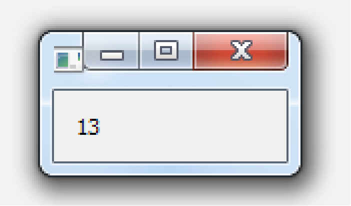
```python
import sys
from PySide6.QtWidgets import *
from PySide6.QtCore import *

class Simple_spin_window(QWidget):
    def __init__(self):
        QWidget.__init__(self, None)
        self.num = 0

        vbox = QVBoxLayout()
        self.label = QLabel(self)
        self.label.setText(str(self.num))
        vbox.addWidget(self.label)

        plus = QPushButton('+', self)
        plus.clicked.connect(self.updateNumber)
        vbox.addWidget(plus)

        minus = QPushButton('-', self)
        minus.clicked.connect(self.updateNumber)
        vbox.addWidget(minus)

        self.setLayout(vbox)

        self.show()

    def updateNumber(self):
        text = self.sender().text()
        if text == '+':
            self.num += 1
        else:
            self.num -= 1
        self.label.setText(str(self.num))

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

    w = Simple_spin_window()

    sys.exit(app.exec_())
```

Program 2.3: Using periodic timer

Source Code

```python
import sys
from PySide6.QtWidgets import *
from PySide6.QtCore import *

class Simple_timer_window(QWidget):
    def __init__(self):
        QWidget.__init__(self, None)
        self.num = 0

        vbox = QVBoxLayout()
        self.label = QLabel(self)
        self.label.setText(str(self.num))
        vbox.addWidget(self.label)

        timer = QTimer(self)
        timer.timeout.connect(self.updateValue)
        timer.start(500)

        self.setLayout(vbox)

        self.show()

    def updateValue(self):
        self.num += 1
        if self.num >= 100:
            self.num = 0
        self.label.setText(str(self.num))

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

    w = Simple_timer_window()

    sys.exit(app.exec_())
```

Output



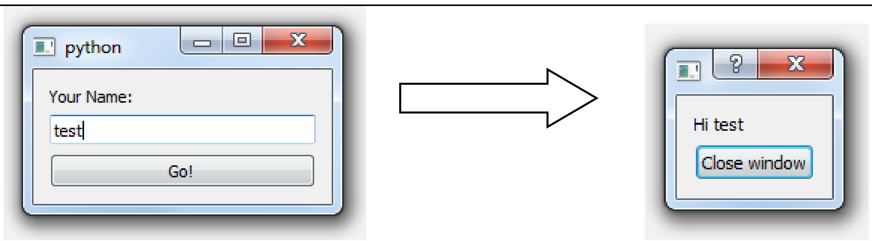
13

Program 2.4: Using Qt LineEdit and Dialog

Source Code

```
import sys
from PySide6.QtWidgets import *
from PySide6.QtCore import *

class Greeting_window(QWidget):
    def __init__(self):
        QWidget.__init__(self, None)
        vbox = QVBoxLayout()
        self.label = QLabel(self)
        self.label.setText("Your Name:")
        vbox.addWidget(self.label)

        self.name_entry = QLineEdit(self)
        vbox.addWidget(self.name_entry)

        pbt = QPushButton('Go!', self)
        pbt.clicked.connect(self.greeting)
        vbox.addWidget(pbt)

        self.setLayout(vbox)

        self.show()

    def greeting(self):
        dialog = QDialog(self)
        layout = QVBoxLayout()

        label = QLabel(self)
        label.setText("Hi " + self.name_entry.text())
        layout.addWidget(label)

        close_button = QPushButton('Close window', self)
        close_button.clicked.connect(dialog.close)
        layout.addWidget(close_button)
        dialog.setLayout(layout)

        dialog.show()

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

    w = Greeting_window()

    sys.exit(app.exec_())
```

Output



python

Your Name:

test

Hi test