

4 Einführung in die GUI-Programmierung mit PyQt5

4.1 Einleitung

Bisher haben wir Python nur in der Konsole resp. im Jupyter Notebook kennengelernt. Dies war notwendig um die Grundkonzepte der Sprache zu lernen. Ab diesem Kapitel lernen wir «richtige» Applikationen mit GUI ("Graphical User Interface"), der grafischen Benutzeroberfläche zu entwickeln. Grundsätzlich ist die GUI Programmierung nicht schwierig, jedoch werden die Programme sehr schnell viel grösser.

Für Python gibt es verschiedene Möglichkeiten eine GUI zu erstellen. Eine Möglichkeit ist die Verwendung des Modules "PyQt5".

PyQt5 ist die aktuelle Python Version von Qt. Qt (Englisch ausgesprochen als "cute") ist eine C++ Klassenbibliothek für die plattformunabhängige Programmierung grafischer Benutzeroberflächen. Die Entwicklung von Qt begann im Jahr 1991.

PyQt5 ist ein ausgereiftes GUI-Framework.

Wie die meisten GUI Systeme ist auch Qt/PyQt objektorientiert aufgebaut. Es ist daher essentiell die Prinzipien der Objektorientierung verstanden zu haben.

4.2 Ein erstes PyQt5 Programm

Nachdem nun alle Vorbereitungen getroffen wurden, können wir die erste PyQt5 Applikation entwickeln und auch testen, ob alles funktioniert. Das unterstehende Programm öffnet ein leeres Fenster. Wir werden später besser verstehen, was genau abläuft.

```
import sys
from PyQt5.QtWidgets import *

# Fenster-Klasse: wird von QMainWindow vererbt
class MyWindow(QMainWindow):
    def __init__(self):      # Konstruktor
        super().__init__() # Konstruktor Basis-Klasse
        self.setWindowTitle("Hello World") # Fenster-Titel setzen
        self.show()        # Fenster anzeigen/sichtbar machen

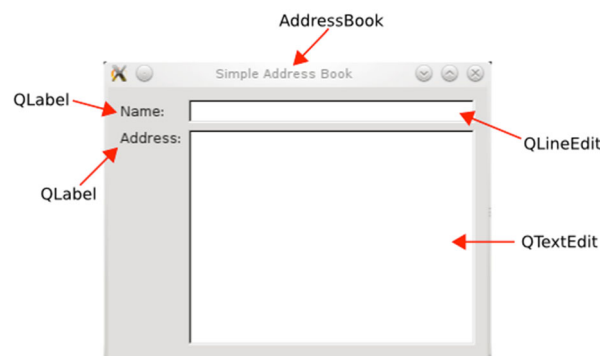
def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                   # Applikations-Loop starten

if __name__ == '__main__':
    main()
```

Programm: 00_window.py


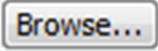
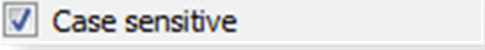
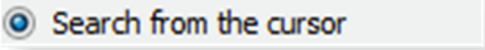
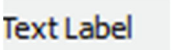
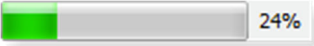

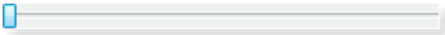
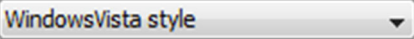
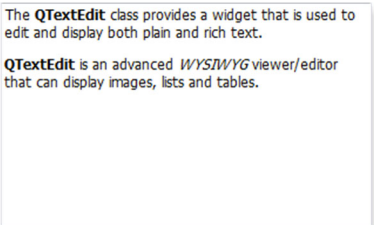

4.3 Widgets

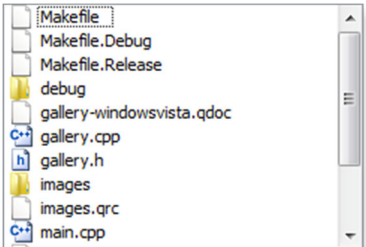
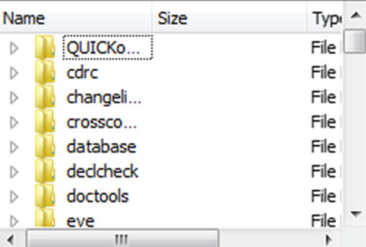
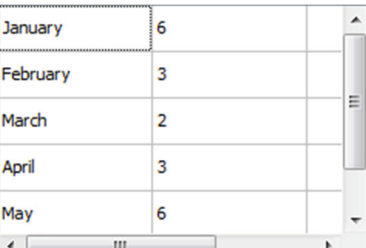
Alle GUI-Elemente - inklusive Fenster wie z.B. QPushButton - werden von der Klasse QWidget vererbt. Zu den Widgets gehören auch Buttons (QPushButton, QCheckBox, QRadioButton), Sliders (QSlider), Menus (QMenu, QMenuBar), Texteingabefelder (QLineEdit) und vieles mehr.



Beispiel Adressbuch: Einige QWidgets (Quelle: qt-project.org)

In der folgenden Tabelle werden einige Widgets gezeigt. Dies ist nur eine kleine Auswahl der wichtigsten GUI-Elemente von Qt.

Screenshot	Klassenname	Beschreibung
	QPushButton	Ein Kommando-Button
	QToolButton	Button, welcher normalerweise in einer Toolbar verwendet wird
	QCheckBox	Eine Checkbox mit einem Text.
	QRadioButton	Ein Radio-Button mit einem Text (oder auch Bild)
	QLabel	Mit QLabel kann Text (oder auch ein Bild) angezeigt werden. In der Regel wird dies als beschreibendes Element für andere Widgets angezeigt.
	QProgressBar	Eine horizontale Statusanzeige, welche einen Fortschritt einer Operation anzeigen soll.
	QLineEdit QDateEdit QTimeEdit QDateTimeEdit	Eingabefelder für Text, Datum, Zeit, Datum&Zeit. Diese sind genau eine Zeile lang.
	QSlider	Ein horizontaler oder vertikaler Slider
	QComboBox	Ein Auswahl-Menü.
	QTextEdit	Ein (in der Regel) mehrzeiliges Texteingabefeld, welches normalen oder auch formatierten "rich Text" darstellen kann.
	QCalendarWidget	Für die Auswahl eines Datums kann auch das komplexere Widget verwendet werden.

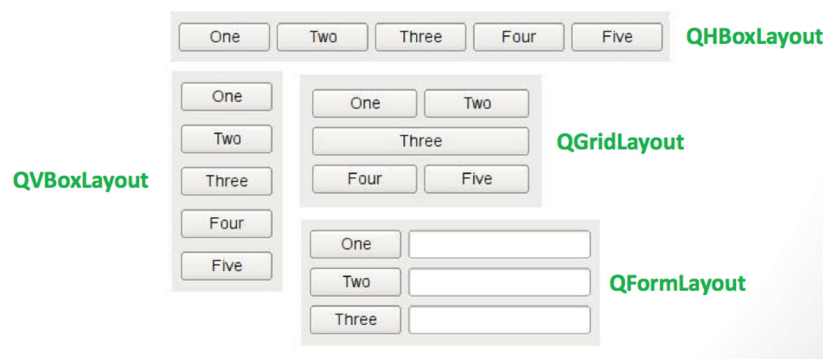
	QListView	Mit der List View kann eine Liste von Elementen angezeigt werden.
	QTreeView	mit dem Tree View kann ein Baum von Elementen angezeigt werden.
	QTableView	Mit der Table View kann eine Tabelle angezeigt werden.

4.4 Layout

Um GUI-Elemente anzuordnen wird ein Layout benötigt. Das Layout beschreibt die Anordnung der GUI-Elemente, also der Widgets. In Qt gibt es verschiedene Layout-Möglichkeiten.

Die wichtigsten Layout-Typen sind:

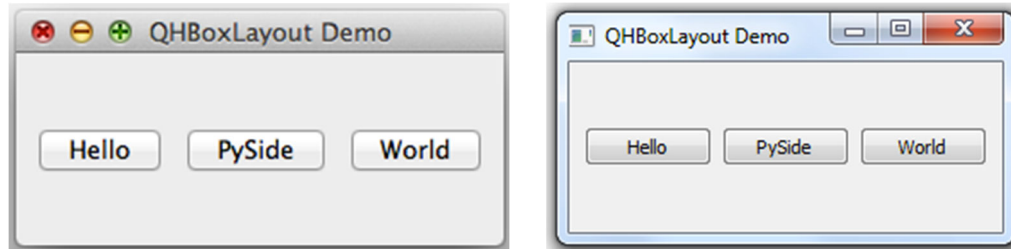
QVBoxLayout, QHBoxLayout, QGridLayout und QFormLayout.



Beispiele von Layouts (Quelle: Yung-Yu Chen, 2013)

4.5 Widgets und Layouts erstellen

4.5.1 BoxLayout: Erstellen von QHBoxLayout und QVBoxLayout



Beispiel eines QHBoxLayout

Nun erstellen wir ein `QHBoxLayout`, welches Widgets horizontal anordnet. Dazu müssen wir eine Instanz von `QHBoxLayout` erstellen. Diese wird in der Variablen "layout" gespeichert. Danach erstellen wir 3 `QPushButton`s und speichern diese in den Variablen "button1", "button2" und "button3".

Dann können wir die Instanzen der Buttons dem layout hinzufügen. Dies machen wir für alle Buttons mit den Anweisungen `layout.addWidget(button1)`, `layout.addWidget(button2)` und `layout.addWidget(button3)`.

Dann erstellen wir ein zentrales Widget "center" welches sozusagen das zentrale "Hauptwidget" des Fensters ist. Diesem Widget übergeben wir das Layout mit `center.setLayout(layout)`. Danach müssen wir unserem Fenster noch mitteilen, dass "center" das Zentrale Widget ist. Dies geschieht mit dem Aufruf von `self.setCentralWidget(center)`.

```
import sys
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        # Fenster-Titel definieren:
        self.setWindowTitle("QHBoxLayout Demo")

        # Layout erstellen:
        layout = QHBoxLayout()

        # 3 Push-Buttons erzeugen:
        button1 = QPushButton("Hello")
        button2 = QPushButton("PyQt5")
        button3 = QPushButton("World")

        # Buttons dem Layout hinzufügen
        layout.addWidget(button1)
        layout.addWidget(button2)
        layout.addWidget(button3)

        # Zentrales Widget erstellen und layout hinzufügen
        center = QWidget()
        center.setLayout(layout)

        # Zentrales Widget in diesem Fenster setzen
        self.setCentralWidget(center)

        # Fenster anzeigen
        self.show()

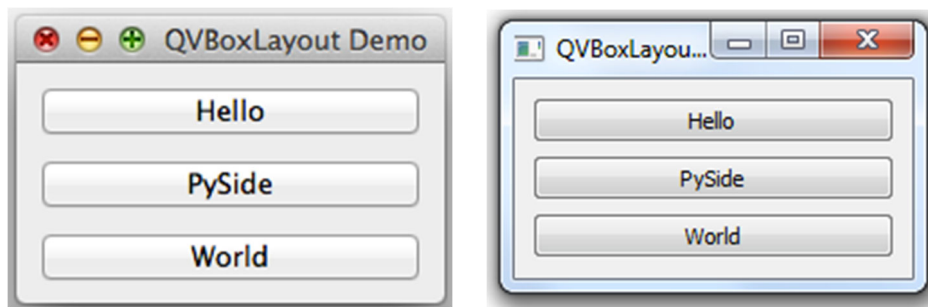
def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                   # Applikations-Loop starten

if __name__ == '__main__':
    main()
```

Programm: 01_hboxlayout.py

Um nun ein `QVBoxLayout` zu erstellen, welche die Widgets vertikal anordnet, kann einfach `"layout = QHBoxLayout()"` ersetzt werden durch:

```
layout = QVBoxLayout()
```



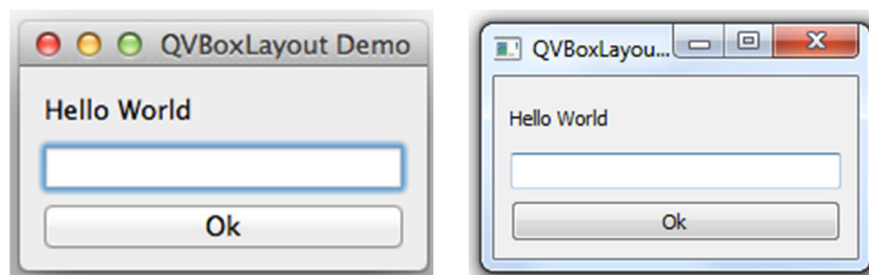
Beispiel eines QVBoxLayout

Um andere Widgets als Buttons zu erstellen kann anstelle der Buttons auch beispielsweise ein `QLabel` oder `QLineEdit` verwendet werden:

```
# Widget-Instanzen erstellen:
label = QLabel("Hello World")
edit = QLineEdit()
button = QPushButton("Ok")

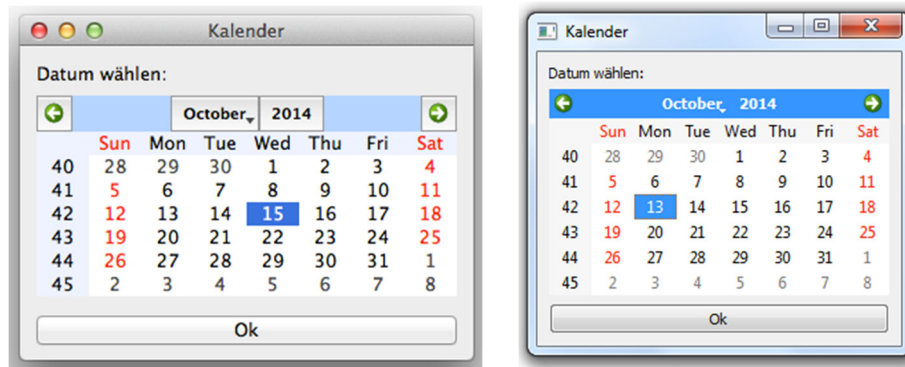
# Widgets dem Layout hinzufügen:
layout.addWidget(label)
layout.addWidget(edit)
layout.addWidget(button)
```

Programm: 02_widgets_vbox.py



Beispiel eines QVBoxLayout mit verschiedenen Widgets

Es können natürlich auch andere Widget Typen verwendet werden, wie beispielsweise das etwas komplexere `QCalendarWidget`, bei welchem ein Datum ausgewählt werden kann.



Das QCalendarWidget()

```
# Layout erstellen:
layout = QVBoxLayout()

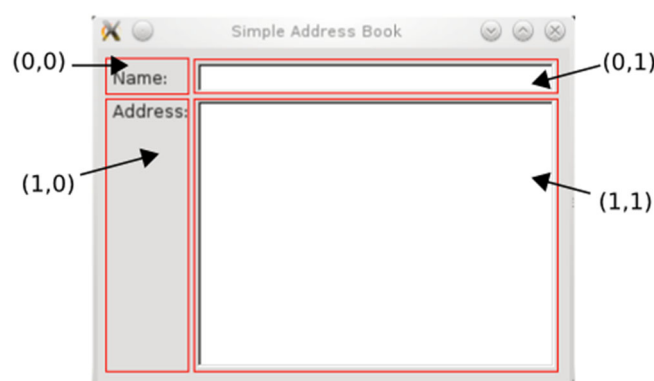
# Widget-Instanzen erstellen:
label = QLabel("Datum wählen:")
calendar = QCalendarWidget()
button = QPushButton("Ok")

# Widgets dem Layout hinzufügen:
layout.addWidget(label)
layout.addWidget(calendar)
layout.addWidget(button)
```

Programm: 03_calendar.py

4.5.2 Erstellen eines QGridLayout

Widgets können mit dem `QGridLayout` in einem Raster angeordnet werden. Das zuvor schon betrachtete Adressbuch im Kapitel 4.3 wurde mit Hilfe eines solchen Rasters erstellt:



QGridLayout mit Raster-Koordinaten (Quelle: qt-project.org)


```
# Layout erstellen:
layout = QGridLayout()

# Widget-Instanzen erstellen:
nameLabel = QLabel("Name:")
nameLine = QLineEdit()
addressLabel = QLabel("Address:")
addressText = QTextEdit()

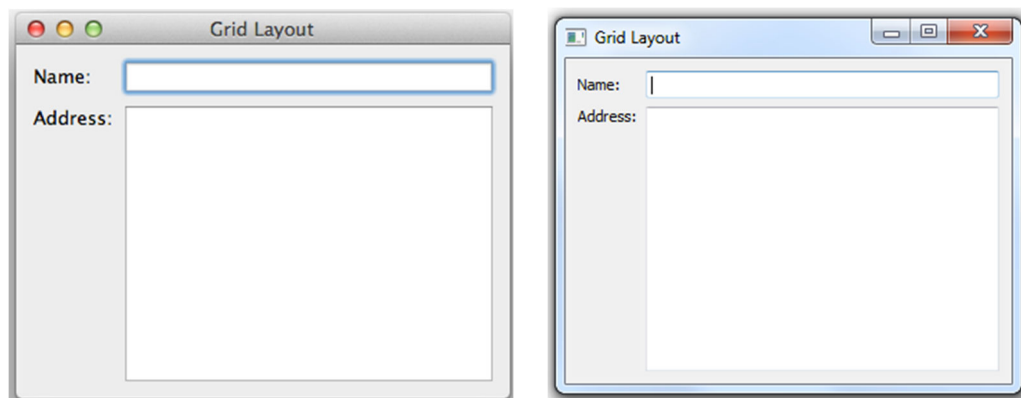
# Widgets mit Grid-Koordinaten dem Layout hinzufügen
layout.addWidget(nameLabel, 0, 0)
layout.addWidget(nameLine, 0, 1)
layout.addWidget(addressLabel, 1, 0, Qt.AlignTop)
layout.addWidget(addressText, 1, 1)
```

Programm: 04_gridlayout.py

Für das `Qt.AlignTop` (um den Text des Labels ganz oben zu sehen) benötigen wir auch noch folgenden Import:

```
from PyQt5.QtCore import *
```

Im `QtCore` Untermodul befinden sich sehr viele solche Definitionen.



Layout-Beispiel mit dem `QGridLayout`

4.5.4 Erstellen eines QFormLayout

Das Form-Layout ist speziell für Formulare gedacht, wo jedes Element ein Label hat und dahinter etwas ausgewählt werden kann. Der Unterschied zum QGridLayout ist, dass ein QLabel nicht extra definiert werden muss. Sehen wir direkt ein Beispiel an:

```
# Layout erstellen:
layout = QFormLayout()

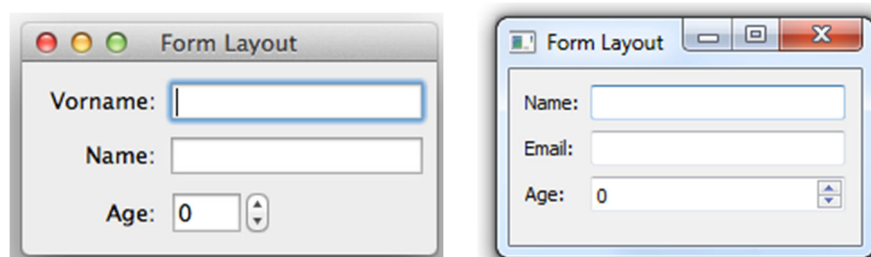
# Widget-Instanzen erstellen:

nameLineEdit = QLineEdit()
emailLineEdit = QLineEdit()
ageSpinBox = QSpinBox()

# Layout füllen:
layout.addRow("Name:", nameLineEdit)
layout.addRow("Email:", emailLineEdit)
layout.addRow("Age:", ageSpinBox)
setLayout(formLayout)
```

Programm: 05_formlayout.py

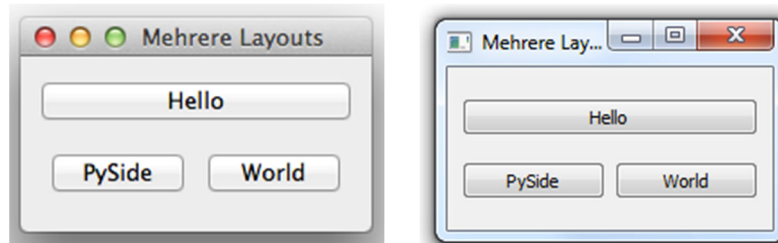
Wie wir im Programm deutlich erkennen können, benötigen wir nur die Hälfte an Widgets gegenüber dem QGridLayout, da wir uns die QLabel sparen konnten. Anstelle von Text-Labels ist es auch möglich andere Widget-Typen zu verwenden.



Beispiel: QFormLayout

4.5.5 Vermischen von Layouts

Layouts können untereinander auch vermischt werden. Dies geschieht über eine Layout-Hierarchie. Zuoberst gibt es immer ein Hauptlayout, welchem mit der Methode "addLayout" ein weiteres Layout hinzugefügt werden kann.



Mehrere Layouts: QVBoxLayout und QHBoxLayout vermischt

```
# Layout erstellen:
layout_top = QVBoxLayout()
layout_bottom = QHBoxLayout()

# 3 Push-Buttons erzeugen:
button1 = QPushButton("Hello")

button2 = QPushButton("PyQt5")
button3 = QPushButton("World")

# Buttons dem ersten (vertikalen) Layout hinzufügen
layout_top.addWidget(button1)

# Restliche Buttons dem zweiten (horizontalen) Layout hinzufügen
layout_bottom.addWidget(button2)
layout_bottom.addWidget(button3)

# Dem ersten Layout das zweite Layout hinzufügen!
layout_top.addLayout(layout_bottom)

# Zentrales Widget erstellen und layout hinzufügen
center = QWidget()
center.setLayout(layout_top)
```

Programm: 06_multilayout.py

4.6 Das Anwendungsmenu

4.6.1 Ein einfaches Menu erstellen

Ein Menu besteht aus einer Menubar (z.B. "File", "Edit") diese haben sogenannte Menueinträge welche wiederum Untermenüs haben können. Die Untermenüs sind in PyQt5 sogenannte "Actions". Diese können mit der Methode `addAction` einem Menu hinzugefügt werden.

Sehen wir uns folgendes Beispiel an:

```
menubar = self.menuBar() # Die Menubar des Fensters erhalten

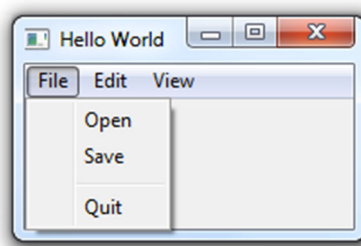
filemenu = menubar.addMenu("File") # "File"-Menu erstellen
editmenu = menubar.addMenu("Edit") # "Edit"-Menu erstellen
viewmenu = menubar.addMenu("View") # "View"-Menu erstellen

open = QAction("Open", self) # Eine Action definieren
save = QAction("Save", self) # Eine weitere Action definieren
quit = QAction("Quit", self) # Eine dritte Action definieren

quit.setMenuRole(QAction.QuitRole) # Rolle "beenden" für MacOS

filemenu.addAction(open) # Die Action wird dem Filemenu hinzugefügt
filemenu.addAction(save) # Eine weitere Action hinzufügen
filemenu.addSeparator() # Eine Trennlinie wird hinzugefügt
filemenu.addAction(quit) # Eine weitere Action hinzufügen
```

Programm: 07_menu.py



Das Menu-Beispiel unter Windows

4.6.2 Menus verschachteln: Untermenüs erstellen

Menus können noch weiter verschachtelt werden. Dies geschieht indem wir dem Menu mit der Methode `addMenu` ein weiteres Menu hinzufügen, also z.B.

```
recent = filemenu.addMenu("Recent")
```

Danach können wir diesem Untermenu wie zuvor Actions hinzufügen.

4.7 Einführung Signale und Slots

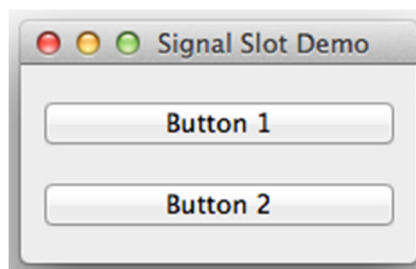
Bisher haben wir GUI Oberflächen erstellt, aber noch nicht auf Eingaben der Benutzenden reagiert. Für die Kommunikation zwischen Objekten benutzt Qt das sogenannte "Signal-Slot-Konzept".

Das Prinzip ist recht einfach:

Sobald ein Objekt ein Signal aussendet werden alle mit ihm verbundenen Slots der Objekte aufgerufen. Wird ein Objekt gelöscht (z.B. ein Fenster geschlossen) so enden alle Verbindungen automatisch.

Damit ein Objekt den Signal-Slot-Mechanismus unterstützt, muss es von der Klasse QObject vererbt werden. Dies ist bereits der Fall für alle QWidgetts.

Für ein QWidget können wir Signale relativ einfach definieren, sehen wir uns doch zunächst folgendes Beispiel an:



Beispiel: Zwei Buttons werden in einem QVBoxLayout definiert. Beim Klicken wird etwas in die Konsole geschrieben.

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        #####
        # LAYOUT #
        #####

        # Fenster-Titel definieren:
        self.setWindowTitle("Signal Slot Demo")
```

```
# Layout erstellen:
layout = QVBoxLayout()

# Widget-Instanzen erstellen:
button1 = QPushButton("Button 1")
button2 = QPushButton("Button 2")

# Widgets dem Layout hinzufügen:
layout.addWidget(button1)
layout.addWidget(button2)

# Zentrales Widget erstellen und layout hinzufügen
center = QWidget()
center.setLayout(layout)

# Zentrales Widget in diesem Fenster setzen
self.setCentralWidget(center)

# Fenster anzeigen
self.show()

#####
# CONNECTS #
#####

# Klick auf Button 1 ruft die Funktion button1_clicked() auf
button1.clicked.connect(self.button1_clicked)

# Klick auf Button 2 ruft die Funktion button1_clicked() auf
button2.clicked.connect(self.button2_clicked)

def button1_clicked(self):
    print("Der Button 1 wurde gedrückt")

def button2_clicked(self):
    print("Der Button 2 wurde gedrückt")

def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                   # Applikations-Loop starten

if __name__ == '__main__':
    main()
```

Programm: 08_signal_slot.py

4.8 Signale und Slots für Widgets

Die wichtigsten Signale für einige Widgets sind in der untenstehenden Tabelle zu finden.

QPushButton	pressed()	Der Button wurde gedrückt
	released()	Der Button wurde losgelassen
	clicked()	Der Button wurde gedrückt & losgelassen
QLineEdit	textChanged(txt)	Der Text hat sich verändert (auch durch Programmierung). txt enthält den Text.
	textEdited(txt)	Der Text hat sich verändert (nur durch editieren). txt enthält den Text.
	returnPressed()	Die Return-Taste wurde gedrückt
	editingFinished()	Das Editieren ist fertig
	selectionChanged()	Die Text-Selektion hat geändert
	cursorPositionChanged(old,new)	Die Cursor Position hat sich verändert
QTextEdit	textChanged()	Wenn der Text ändert.
QCheckBox	stateChanged(state)	Status-Änderung bei Check-Box state kann folgende Werte haben: Qt.CheckState.Checked Qt.CheckState.Unchecked Qt.CheckState.PartiallyChecked
QRadioButton	toggled(checked)	Der Radio Button hat sich verändert. (auch durch Programmcode)
QCalendarWidget	clicked(date)	Es wurde auf ein Datum geklickt. Date ist vom Typ QtCore.QDate
QSlider	valueChanged(value)	Wird aufgerufen, wenn der Wert des Slides sich verändert hat. Mit tracking() kann herausgefunden werden, ob es während der User-Interaktion geschieht.
	sliderPressed()	Wird aufgerufen, wenn begonnen wird den Slider zu verschieben (klick).
	sliderMoved(value)	Wird aufgerufen, wenn der Slider bewegt wird.
	sliderReleased()	Wird aufgerufen, wenn der Slider losgelassen wird.
QComboBox	currentIndexChanged(index)	Wird aufgerufen wenn ein neues Item in der ComboBox gewählt wurde
	activated(index)	Wird immer aufgerufen wenn eine Auswahl getroffen wurde (auch dieselbe, auch vom Programm)
QAction	triggered()	Wenn die Action ausgelöst wird.

Weitere Signale können in der Qt Dokumentation (<http://qt-project.org/doc/>) oder PyQt5 Dokumentation (<http://pyqt.sourceforge.net/Docs/PyQt5/>) gefunden werden.

4.9 Signale für Menus

Auch für Menus können Signale definiert werden. Dies geschieht auch über `QAction` mit dem Signal `triggered()`.

```
import sys
from PyQt5.QtWidgets import *

# Fenster-Klasse: wird von QMainWindow vererbt
class MyWindow(QMainWindow):
    def __init__(self):      # Konstruktor
        super().__init__() # Konstruktor Basis-Klasse
        self.setWindowTitle("Hello World") # Fenster-Titel setzen
        self.show() # Fenster anzeigen/sichtbar machen

        menubar = self.menuBar()
        filemenu = menubar.addMenu("File")

        open = QAction("Open", self)
        open.triggered.connect(self.menu_open)
        save = QAction("Save", self)
        save.triggered.connect(self.menu_save)
        quit = QAction("Quit", self)
        quit.triggered.connect(self.menu_quit)

        # Rolle "beenden" (für MacOS)
        quit.setMenuRole(QAction.QuitRole)

        filemenu.addAction(open)
        filemenu.addAction(save)
        filemenu.addSeparator()
        filemenu.addAction(quit)

    def menu_open(self):
        print("Menu Open wurde gewählt...")

    def menu_save(self):
        print("Menu Save wurde gewählt...")

    def menu_quit(self):
        print("Menu Quit wurde gewählt...")
        self.close() # Hauptfenster schliessen = beenden!

def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                   # Applikations-Loop starten
```

```
if __name__ == '__main__':
    main()
```

Programm: 09_signal_slot_menu.py

4.10 Beispiele von Signalen für verschiedene Widgets

Im folgenden Beispiel werden mehrere verschiedene Widgets erzeugt und Signale/Slots definiert. Neu ist auch, dass das Erstellen des Layouts and das erstellen der Connections nun in Methoden ausgelagert wird. Die Widgets selbst sind nun auch alle Attribute der Klasse. Dies erhöht die Lesbarkeit und die Wartbarkeit des Codes.

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.createLayout()
        self.createConnects()

    def createLayout(self):
        # Fenster-Titel definieren:
        self.setWindowTitle("Signal Slot Demo")

        # Layout erstellen:
        layout = QVBoxLayout()

        # Widget-Instanzen erstellen:
        self.button = QPushButton("Button 1")
        self.lineedit = QLineEdit()
        self.checkbox = QCheckBox("Bla")
        self.checkbox.setChecked(Qt.CheckState.Checked)

        # Widgets dem Layout hinzufügen:
        layout.addWidget(self.button)
        layout.addWidget(self.lineedit)
        layout.addWidget(self.checkbox)

        # Zentrales Widget erstellen und layout hinzufügen
        center = QWidget()
        center.setLayout(layout)

        # Zentrales Widget in diesem Fenster setzen
        self.setCentralWidget(center)

    # Fenster anzeigen
```

```

self.show()

def createConnects(self):
    self.button.clicked.connect(self.button_clicked)
    self.lineedit.textChanged.connect(self.lineedit_update)
    self.checkbox.stateChanged.connect(self.checkbox_changed)

def button_clicked(self):
    print("Der Button wurde gedrückt")

def lineedit_update(self, txt):
    print("LineEdit Update:", txt)

def checkbox_changed(self, state):
    if state == Qt.CheckState.Checked:
        print("checkbox is checked")
    elif state == Qt.CheckState.Unchecked:
        print("checkbox is unchecked")

def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                   # Applikations-Loop starten

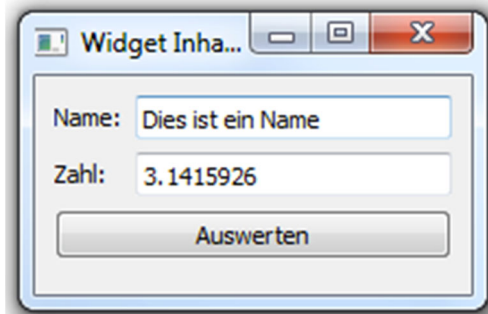
if __name__ == '__main__':
    main()

```

Programm: 10_signal_slot_widgets.py

4.11 Zugriff auf Widget-Inhalte

Wenn wir auf Inhalte von Widgets zugreifen wollen geschieht dies immer über das Widget selbst. Im unteren Beispiel sehen wir, dass ein Name und eine Zahl über ein `QLineEdit` eingelesen werden. Durch einen Klick auf den Button "Auswerten" soll mit diesen zwei Eingaben gearbeitet werden. In unserem einfachen Beispiel soll einfach der Name ausgegeben werden und die Zahl (sofern es eine Zahl ist) verdoppelt werden.



Beispiel: Daten über QLineEdit erfassen und auswerten

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        self.createLayout()
        self.createConnects()

    def createLayout(self):
        # Fenster-Titel definieren:
        self.setWindowTitle("Widget Inhalt lesen")

        # Layout erstellen:
        layout = QFormLayout()

        # Widget-Instanzen erstellen:

        self.nameLineEdit = QLineEdit()
        self.zahlLineEdit = QLineEdit()
        self.button = QPushButton("Auswerten")

        # Layout füllen:
```

```

layout.addRow("Name:", self.nameLineEdit)
layout.addRow("Zahl:", self.zahlLineEdit)
layout.addRow(self.button)

# Zentrales Widget erstellen und layout hinzufügen
center = QWidget()
center.setLayout(layout)

# Zentrales Widget in diesem Fenster setzen
self.setCentralWidget(center)

# Fenster anzeigen
self.show()

def createConnects(self):
    self.button.clicked.connect(self.auswertung)

def auswertung(self):
    name = self.nameLineEdit.text()
    print("Der Name ist", name)
    try:
        zahl = float(self.zahlLineEdit.text())
        print("Die Zahl ist:", zahl)
        print("Zahl * 2 = ", zahl*2)
    except ValueError:
        print("keine gültige Zahl eingegeben!!")

def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                   # Applikations-Loop starten

if __name__ == '__main__':
    main()

```

Programm: 11_read_widgets.py

Im Beispiel haben wir gesehen, dass es mit `lineEdit.text()` möglich ist auf den Inhalt eines `QLineEdit` zuzugreifen. Andere Widgets haben ähnliche Methoden um auf den Inhalt zuzugreifen oder den Inhalt zu verändern.

Die nachfolgende Tabelle zeigt wichtige Methoden zum Abfragen und Setzen des Inhalts für gängige Widgets.

QLineEdit	text()	Zugriff auf den Text innerhalb des QLineEdit
	setText(text)	Setzt den Inhalt des QLineEdit
QLabel	text()	Zugriff auf den Text des Labels
	setText(text)	Setzt den Inhalt des Labels
QCheckBox und QRadioButton	isChecked()	Gibt True zurück falls Checkbox gewählt ist.
	setChecked(bool)	Setzt Checkbox.
QDateEdit	date()	Gibt das Datum als QDate Objekt zurück
	setDate(date)	Setzt das Datum
QTimeEdit	time()	Gibt die Zeit als QTime Objekt zurück
	setTime(time)	Setzt die Zeit
QSlider	value()	Gibt den aktuellen Wert des Sliders
	setValue(value)	Setzt den Wert
	setRange(min, max)	Setzt den Wertebereich des Slides. Es dürfen nur ganze Zahlen verwendet werden.
QComboBox	currentIndex()	gibt den aktuell gewählten Index zurück
	currentText()	gibt den aktuell gewählten Text zurück
	setCurrentIndex()	setzt den aktuellen Index