
Skriptsprache Python

Arbeiten mit Qt

Was ist ein GUI (**G**raphical **U**ser **I**nterface)?

- Strukturierte **g**rafische Darstellung eines Ablaufs
- Benutzer (**U**ser) kann durch Aktionen den Verlauf beeinflussen
- Die grafische Darstellung ist die Schnittstelle (**I**nterface) zwischen Benutzer und Programm

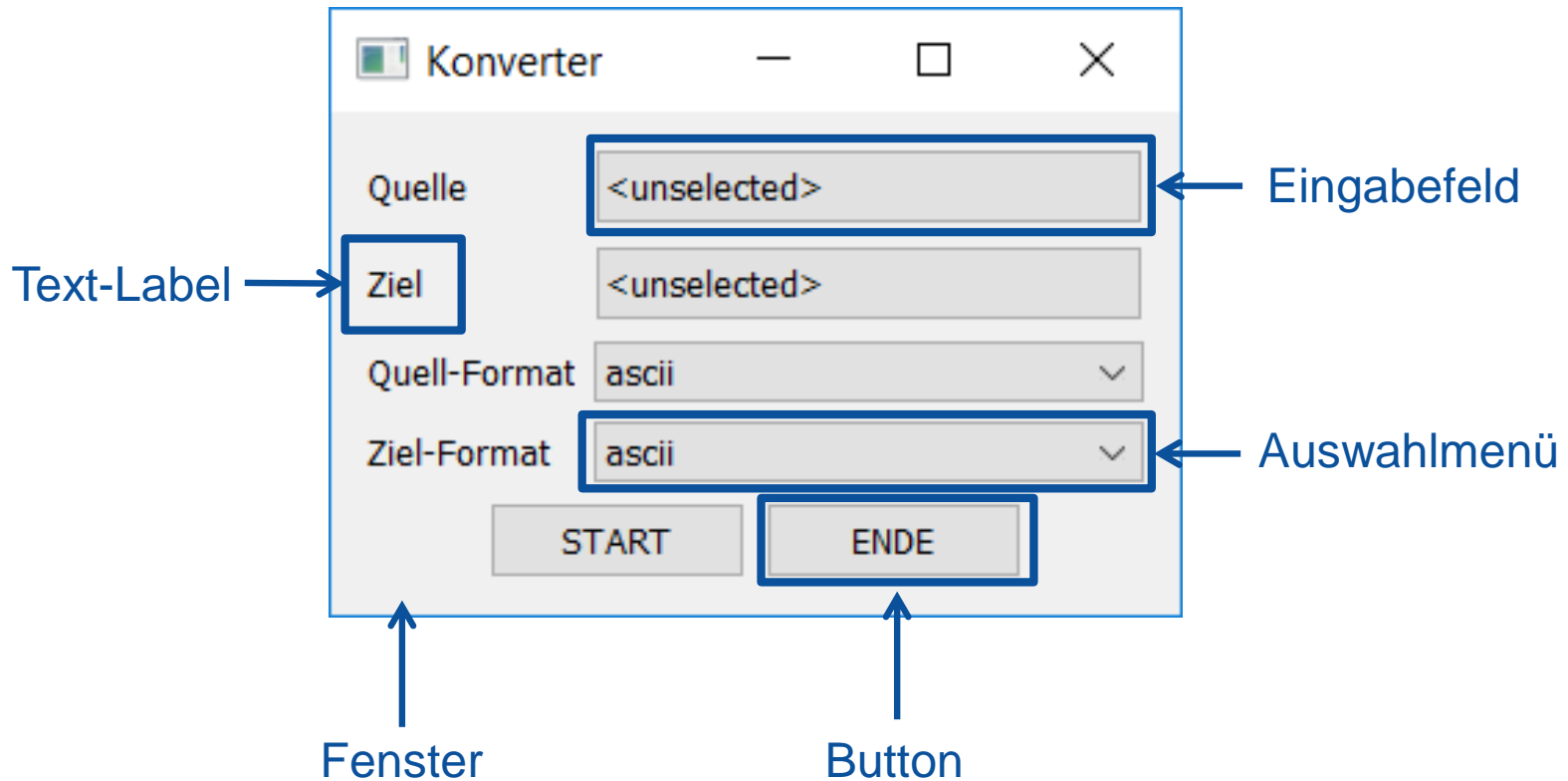
Wie funktioniert ein GUI?

- Computer erzeugt eine grafische Darstellung auf dem Bildschirm ...
- und reagiert auf Benutzeraktionen wie z.B. Tastatureingaben, Mausbewegung, usw.
- Aufgrund der Benutzeraktionen und der Ablaufsteuerung wird die Bildschirmdarstellung aktualisiert, usw.

Qt (für engl. „cute“):

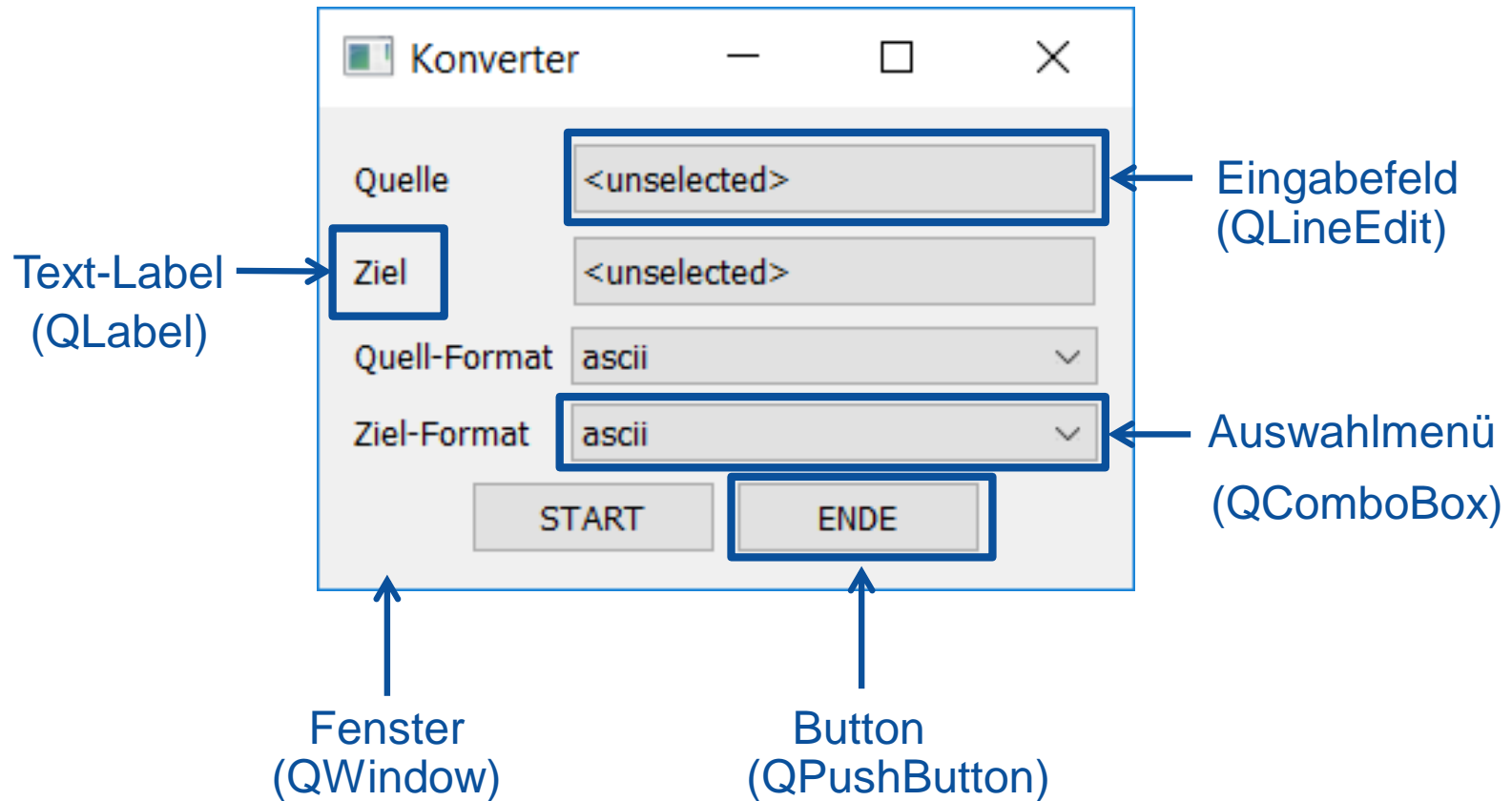
- Programmier-Framework für Objekt-orientierte Programmierung mit C++
- Häufig für Betriebssystem-unabhängige GUIs genutzt
- Implementierungen für verschiedene Betriebssysteme (Windows, Linux,...)
- Als Open-Source-Software unter der GNU LGPL verfügbar oder mit einer kommerziellen Lizenz
- Programmierbibliotheken für GUI, Multimedia, Netzwerk, ...
- Aktuell neueste Qt-Version 6 → Qt6
- Qt unterstützt Anbindung vieler Skript- bzw. Programmiersprachen, darunter auch Python

Qt-GUI als Zusammenwirken verschiedener GUI-Elemente



- GUI-Elemente in Qt → Objekte, die zu bestimmten Klassen gehören
- Basisklasse der GUI-Elemente ist **QWidget**:
 - rechteckiger Bereich,
 - der sich selbst darstellen kann und
 - auf Tastatur- und Mauseingaben reagiert
- Andere GUI-Elemente sind von dieser Basisklasse abgeleitet
- GUI-Element **QWindow** enthält andere Widgets wie Label, Button, ...

GUI-Elemente mit Ihren Klassennamen



- Für Python existieren zwei unabhängige Qt-Anbindungen:
 - Für Qt5: PySide2 und PyQt5
 - Für Qt6: PySide6 und PyQt6
- PySide bzw. PyQt kein Standardbestandteil von Python → muss nachträglich installiert werden
- Qt-Anbindung enthält mehrere Module mit über 600 Klassen und 6000 Funktionen/Methoden
- wichtige Qt-Module, die im Folgenden verwendet werden
 - QtCore → GUI-unabhängige Kernkomponenten
 - QtGui → Kernkomponenten für grafische Oberfläche
 - QtWidgets → Komplexere GUI-Elemente

- Speicherung von Daten:
 - QByteArray, QString, QPoint, QSize, ...
- Zugriff auf Dateisystem
 - QFile, QDir, QTextStream, ...
- Grundkomponenten für Ereignisbehandlung
 - QEvent, QTimer, ...
- Weitere, in Summe etwa 240 Komponenten

- Grundlegende Werttypen für Grafikdarstellung:
 - QColor, QFont, QBrush, QPen, ...
- Grafiksystem
 - QPainter, QPrinter , QImage, QPixmap, QWidget, ...
- GUI-bezogene Grundkomponenten für Ereignisbehandlung
 - QCloseEvent, QPaintEvent, ...
- Weitere, in Summe etwa 190 Komponenten

- Applikation:
 - QApplication
- Widget-Komponenten
 - Basis-Komponente für GUI: QWidget
 - Darstellung: QLabel, QListWidget,
 - Eingabe: QTextEdit, QDateEdit, QTimeEdit
 - usw.
- Weitere, in Summe etwa 210 Komponenten

Einfache Qt-Anwendung

prozedural

```
# tb_qt1.py

import sys

from PySide6.QtWidgets import \
    QApplication, QWidget

app = QApplication(sys.argv)
w=QWidget()
w.resize(300, 200)
w.setWindowTitle("tb_qt1")
w.show()
app.exec()
```

→ prozedural im Kurs nicht weiter thematisiert

Objekt-orientiert

```
# tb_qt2.py

import sys

from PySide6.QtWidgets import \
    QApplication, QWidget

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

    def initUI(self):
        self.resize(300, 200)
        self.setWindowTitle("tb_qt2")
        self.show()

app = QApplication(sys.argv)
w=MyWidget()
app.exec()
```

→ objekt-orientiert Fokus Im Kurs

Vergleich zwischen PyQt und PySide

Wenn auf Besonderheiten verzichtet wird, ist die Programmierung weitgehend portabel zwischen PySide und PyQt:

```
# tb_pyside.py

import sys

from PySide6.QtWidgets import \
    QApplication, QWidget

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

    def initUI(self):
        self.show()

app = QApplication(sys.argv)
w=MyWidget()
app.exec()
```

```
# tb_pyqt.py

import sys

from PyQt6.QtWidgets import \
    QApplication, QWidget

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

    def initUI(self):
        self.show()

app = QApplication(sys.argv)
w=MyWidget()
app.exec()
```

Instanziierung von Widgets

GUI-Fenster erzeugen

```
w=QWidget()  
w.resize(300, 200)  
w.show()
```

- Fenster erzeugen
- Größe festlegen und anzeigen

Button im GUI-Fenster erzeugen

```
btn=QPushButton("START", w)  
btn.move(100, 200)  
btn.show()
```

- Button mit Text „START“ erzeugen
- Position festlegen und anzeigen

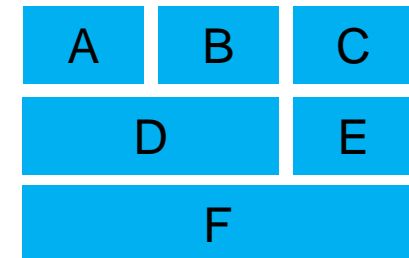
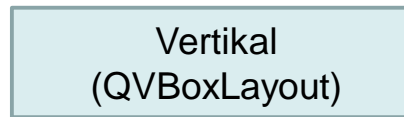
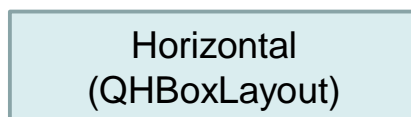
Layout-Manager zum automatischen Platzieren von Buttons erzeugen

```
layout=QHBoxLayout()  
  
btn=QPushButton("&START")  
layout.addWidget(btn)  
btn=QPushButton("&ENDE")  
layout.addWidget(btn)  
w.setLayout(layout)
```

- Layout-Manager erzeugen
- Buttons im Layout platzieren
- Layout für GUI-Fenster verwenden

Layout-Manager

- Größe und Position von Widgets können mit Methoden wie `move()`, `resize()` geändert werden
- Layout-Manager können automatisch Größe und Position von Widgets nach bestimmten Regeln festlegen
- Vorteil: ändert sich Geometrie des beinhaltenden Widgets, werden vom Layout-Manager verwaltete Komponenten automatisch angepasst
- Oft verwendete Layout-Manager: horizontal, vertikal, als Gitter



QHBoxLayout / QVBoxLayout

- Layout-Manager für horizontale / vertikale Aufteilung
- Wichtige Methoden:

Methode	Funktion
<code>addWidget(widget)</code>	Widget hinzufügen
<code>addStretch()</code>	Freiraum hinzufügen, der überzähligen Platz ausfüllt
<code>addLayout(layout)</code>	Von weiterem Layout-Manager verwalteten Bereich hinzufügen

Beispiel zu QVBoxLayout

```
# tb_qt_box.py

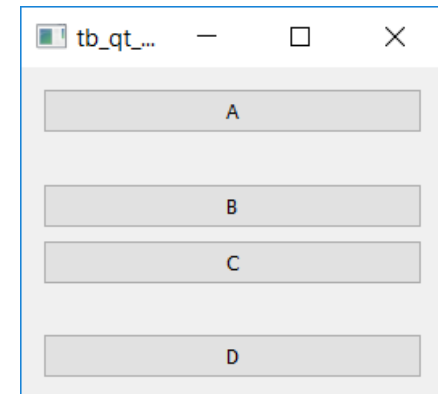
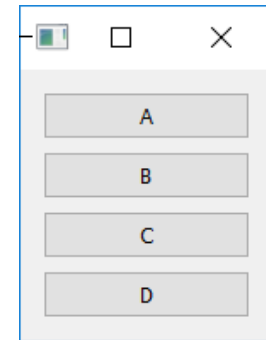
import ...

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

    def initUI(self):
        self.setWindowTitle("tb_qtbox")
        box = QVBoxLayout()

        box.addWidget(QPushButton("A", self))
        box.addStretch()
        box.addWidget(QPushButton("B", self))
        box.addWidget(QPushButton("C", self))
        box.addStretch()
        box.addWidget(QPushButton("D", self))

        self.setLayout(box)
        self.show()
```



QGridLayout

- Layout-Manager für Aufteilung als Gitter
- Wichtige Methoden:

Methode	Funktion
<code>addWidget(widget, row, col)</code>	Widget in Zeile row und Spalte col hinzufügen
<code>addWidget(w, r, c, rowspan, colspan)</code>	Widget an Pos. Row/col hinzufügen, das rowspan Zeilen und colspan Spalten belegt.
<code>addLayout(layout, row, col)</code>	Von weiterem Layout-Manager verwalteten Bereich an row / col hinzufügen

Beispiel zu QGridLayout

```
# tb_qt_grid.py (gekürzt)
```

```
import ...
```

```
class MyWidget(QWidget):
```

```
    def __init__(self): ...
```

```
    def initUI(self):
```

```
        grid = QGridLayout()
```

```
        grid.addWidget(QPushButton("A", self), 0, 0)
```

```
        grid.addWidget(QPushButton("B", self), 0, 1)
```

```
        grid.addWidget(QPushButton("C", self), 0, 2)
```

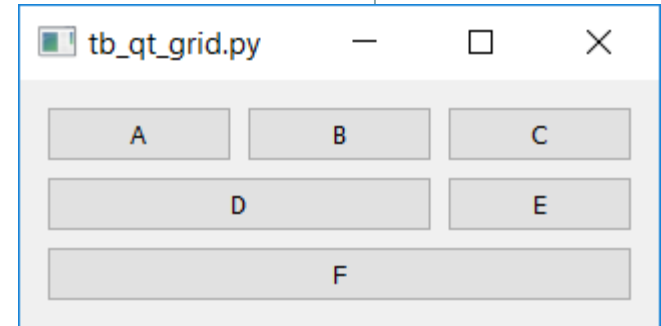
```
        grid.addWidget(QPushButton("D", self), 1, 0, 1, 2)
```

```
        grid.addWidget(QPushButton("E", self), 1, 2)
```

```
        grid.addWidget(QPushButton("F", self), 2, 0, 1, 3)
```

```
        self.setLayout(grid)
```

```
        self.show()
```



Ereignisbehandlung

- Ein Widget kann über **Event-Handler** bestimmte Ereignisse bearbeiten
- Diese Event-Handler sind Methoden mit vorgegebenen Namen
- Wichtige Event-Handler:

Event-Handler	Aufruf bei ...
<code>closeEvent (ev)</code>	Schließen des Widgets
<code>hideEvent ()</code> <code>showEvent ()</code>	Fenster wird minimiert bzw. sichtbar
<code>keyPressEvent (ev)</code> <code>keyReleaseEvent (ev)</code>	Taste wurde gedrückt / losgelassen
<code>mouseMoveEvent (ev)</code> <code>mousePressEvent (ev)</code> <code>mouseReleaseEvent (ev)</code>	Maus wurde bewegt bzw. eine Maustaste gedrückt oder losgelassen
<code>paintEvent (ev)</code>	Widget soll neu gezeichnet werden
<code>resizeEvent (ev)</code>	Größe des Widgets hat sich geändert
Weitere ...	

Beispiel zu Event-Handler closeEvent

```
# tb_qt_close.py (gekürzt)

import ...

from PySide6.QtWidgets import ... QMessageBox as QMB

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

    def initUI(self):
        self.resize(300,100)
        self.show()

    def closeEvent(self, ev):
        r = QMB.question(self, "Abfrage", "Beenden?",
                        QMB.Yes | QMB.No, QMB.No)
        if r == QMB.Yes:
            ev.accept()
        else:
            ev.ignore()
```

Beispiel zu Event-Handler paintEvent

```
# tb_qt_paint.py (gekürzt)
```

```
class MyWidget(QWidget):
```

```
...
```

```
def paintEvent(self, ev):
```

```
    qp = QPainter()
```

```
    w, h = self.width(), self.height()
```

```
    qp.begin(self)
```

```
    qp.setBrush(Qt.black)
```

```
    qp.setPen(Qt.black)
```

```
    qp.drawRect(0, 0, w, h)
```

```
    qp.setFont(QFont("Decorative", 10))
```

```
    qp.setPen(Qt.white)
```

```
    qp.drawText(0, 15, "Ein Demo")
```

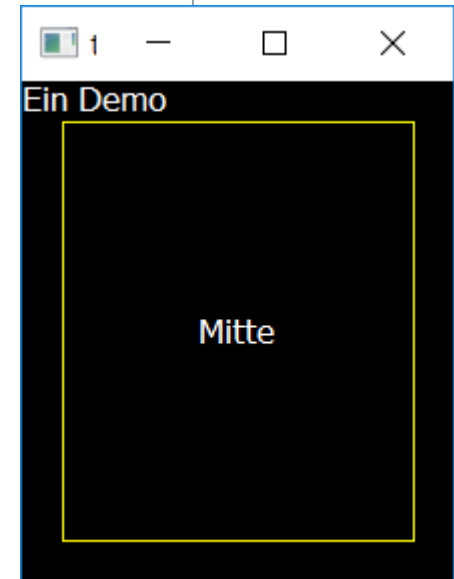
```
    qp.drawText(ev.rect(), Qt.AlignCenter, "Mitte")
```

```
    qp.setBrush(Qt.transparent)
```

```
    qp.setPen(Qt.yellow)
```

```
    qp.drawRect(20, 20, w-40, h-40)
```

```
    qp.end()
```



Beispiel zu Event-Handler keyPressEvent

```
# tb_qt_paint.py (gekürzt)

class MyWidget(QWidget):
    ...

    # Tasten R und Q bearbeiten

    def keyPressEvent(self, ev):
        key = ev.key() # Keycode ermitteln
        if key == Qt.Key_Q: self.close() # Fenster zu
        if key == Qt.Key_R: self.update() # Fenster neu zeichnen
        ev.accept()
```

- Event-Handler haben vorgegebene Namen und müssen für eine bestimmte Funktionalität in eigenen Klassen mit gleichnamigen Methoden überschrieben werden
- Daneben können in Qt auch zur Laufzeit eines Programms beliebige Funktionen / Methoden an bestimmte Ereignisse gebunden werden
- Diese Ereignisse werden „**Signale**“ genannt und die aufgerufenen Funktionen / Methoden „**Slots**“
- Tritt ein bestimmtes Ereignis / Signal auf, werden die damit verbundenen Routinen / Slots aufgerufen
- Vorteil: Es muss keine neue Klasse definiert werden, wenn z.B. auf den Tastendruck eines Buttons reagiert werden soll

Beispiel: Signal wird an mehrere Slots gesendet

```
# tb_qt_slot.py (gekürzt)

import ...

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

    def initUI(self):
        b=QPushButton("Drück mich", self)
        b.clicked.connect(self.onPress)
        b.clicked.connect(self.onAnotherPress)
        self.show()

    def onPress(self):
        print("Button gedrückt 1. Slot")

    def onAnotherPress(self):
        print("Button gedrückt 2. Slot")
```

Signal: clicked

Slot: onPress

Slot: onAnotherPress

- **Autor**

Prof. Dr.-Ing. Jürgen Krumm

- **Impressum**

Prof. Dr.-Ing. J. Krumm, TH Nürnberg Georg Simon Ohm,
Fakultät Elektrotechnik Feinwerktechnik Informationstechnik,
Postfach 210320, 90121 Nürnberg, Germany,
Tel:+49-911-5880-1111,
E-mail: juergen.krumm@th-nuernberg.de

Dieses Skriptum ist nur für den eigenen Gebrauch im Studium gedacht. Eine Weitergabe ist nur mit Zustimmung des Autors gestattet.