Table of Contents

Einleitung

Copyright

Motivation

Nach dem ich bereits seit über 30 Jahren Software entwickle, kam ich von C, Assembler, Clipper, Powerbuilder, Java, C#, Objetive-C, C++ zu Python. Und rate mal...
I liebe Python
Es wäre einfach für mich native Applikationen in Java, C++ oder Objective-C zu schreiben und ausserdem wäre ich ich in der Lage, Kotlin, Dart oder Swift zu lernen, aber die Dinge sind einfacher, wenn man sie mit Python programmiert.
Ich habe mal ein Django Tutorial gemacht, Django ist ein Web-Framework für Python, und dabei hat mich fasziniert, wie einfach es ist, Daten-Modelle zu entwickeln. Einfach eine simple Datenklasse schreiben, ein Scaffolding job starten und schwups werden alle nötigen Tabellen auf dem SQL-Server für dich angelegt. Dann nur noch eben den Python-Interpreter starten, das Modell importieren, eine Instanz erstellen, sie mit Daten füllen und die Save-Methode aufrufen und schon wird das Ganze auf den SQL-Server geschrieben.
Ich musste nicht eine Zeile SQL-Code schreiben. Python hatte mich eingefangen :-)
Dann habe ich noch etwas über Generators, Comprehensions und Meta-Programmierung erfahren und Python hatte mich komplett überzeugt.

Wo wir grad dabei sind...um dieses Buch schreiben zu können, habe ich eine Applikation geschrieben, die ich EbookCreator genannt habe. Diese Anwendung nutz auch PyQt5, QtWidgets und QML nutze ich, um Daten zu serialisieren. Diese App ist Open Source und du kannst sie benutzen, um dich inspirieren zu lassen. Du findest den Source Code auf github.com.

Bevor ich mein erstes Buch über Python GUI geschrieben habe, habe ich ein Video mit "Uncle" Bob Martin über das Programmieren in der Zukunft gesehen. Er sagt, das sich alle 5 Jahre die Zahl der Softwareentwickler

verdoppelt. Das bedeutet, das um die 50% aller Entwickler unerfahren sind. Er sagte auch, das die erfahrenen Entwickler dafür verantwortlich sind, wenn z.B. ein selbst fahrendes Auto einen Menschen umfährt und tötet, nur weil ein Softwarefehler vorlag. Als ich das Heute gehörte habe, habe ich mich entschlossen andere Entwickler auszubilden und habe mit diesem Buch begonnen.

Für Wen Ist Dieses Buch

Wenn du in der Lage bist, einfache Programme in Python zu schreiben und interessiert bist Anwendungen mit einem grafischem Benutzer-Interface für Android zu schreiben, dann ist dieses Buch genau das richtige für dich.
Du musst dich nicht unbedingt mit Qt auskennen.
Wenn du willst, probiere alle Beispiele aus diesem Buch selber aus. Von Vorteil wäre es, wenn du auch, wie ich, auf Linux arbeitest. Die Beispiele sollten aber auch mühelos auf MacOS und Windows laufen. Lediglich für die Installation der benötigten Software solltest du dich selber im Internet einlesen, da ich nur die nötigen Schritte für Linux erkläre.

Für Wen Ist Dieses Buch Nicht

Solltest du noch ein Anfänger in Python sein, dann schlage ich dir vor, erst einmal einen geeigneten Grundkurs für Python zu machen, bevor du in diesem Buch weiterliest. Es gibt hierfür tolle Bücher und eine Menge Videos auf Youtube.
In diesem Buch gehe ich nicht in Python spezifische Details ein.

Wie Dieses Buch Organisiert Ist

Hier findest du die Übersicht über die Teile dieses Buches.
Zuerst werden wir in ***Teil I*** alle nötigen Werkzeuge zum Erstellen der Software mit PyQt5 und Python installieren.
In ***Teil II*** lernst du etwas über die verschiedenen Ansätze um Anwendungen zu bauen.
In ***Teil III*** lernst du Anwendungen mit Python und QML zu schreiben.
In ***Teil IV*** lernst du, wie man eine Line Of Business (LOB) applikation erstellt, die eine Datenbank benutzt, um Daten zu speichern. Und in ***Teil V*** wirst du lernen, wie man ein Setup Programm erstellt, um Python Software auf Linux zu installieren.

Konventionen In Diesem Buch

In diesem Buch nutze ich folgende typografischen Konventionen.

Kursiv

Wird benutzt, um Dateinamen und Pfade zu markieren

```
Feste Schriftbreite
```

Wird für Programm-Beispiele benutzt

Beipiel Source Code Benutzen

Die gesamten Beispiele sind auf github.com verfügbar.

Wie Du Mich Kontaktieren Kannst

Solltest du eine Frage oder ein Kommentar zu diesem Buch haben, scheue dich nicht, mir eine Email zu schreiben. Sende deine Fragen und Kommentare einfach an: japp.olaf@gmail.com

Danksagungen

Zu allererst bin ich meinem Körper dankbar, weil er mich zur richtigen Zeit auf die richtigen Wege geführt hat. Ich weiß, das klingt bestimmt ein bisschen verrückt, aber da ich Maschinenschlosser gelernt habe und bereits nach wenigen Jahren, Rückenschwerzen bekam und über ein halbes Jahr krank war, habe ich angefangen Maschinenbau zu studieren und während des Studiums habe ich dann mit dem Programmieren angefangen. Zu der Zeit habe ich mich entschieden, mein Studium abzubrechen und als Softwareentwickler zu arbeiten.
Dann hat mir mein Körper vor 5 Jahren mit gleich zwei Burnouts zu verstehen gegeben, mich aus dem Arbeitsleben zurückzuziehen.
Nun habe ich viel Zeit, um Open Source Software zu schreiben und neue Dinge auszuprobieren, wie zum Beispiel Bücher wie dieses hier zu schreiben.

Ich bin Guido Rossum dankbar, weil es 1991 Python erfunden und veröffentlicht hat.

Und ausserdem bin ich allen Pythonistas da draussen dankbar, weil sie so tolle Tutorials und Videos gemacht haben, damit ich Python lernen konnte.

Teil I - Prolog

Installation

Wir werden nun anfangen und alle nötigen Programme installieren, um in der Lage zu sein, die gewünschte Software zu schreiben und sie auszuprobieren.
Um Anwendungen auch auf anderen Platformen installieren zu können, werden wir zusätzliche Software im Kapitel über die Installation, später in diesem Buch, installieren.

Ich gehe davon aus, das du bereits Python3 und Pip auf deinem Rechner installiert hast. Wenn nicht, findest du alle notwendigen Informationen auf der Python Webseite. Wir benötigen Python in der Version 3.7.
Ich gehe ausserdem davon aus, dass du in der Lage bist Pakete mittels Pip zu installieren.

Zuerst werden wir PyQt5 welches zusammen mit Qt5 kommt installieren, damit wir Desktop-GUI-Applikationen entwickeln können.

```
1  user@machine:/path$ pip3 install PyQt5
2  user@machine:/path$ pip3 install PyQtWebEngine
```

Dann werden wir Visual Studio Code installieren. VS-Code ist kostenlos und Open Source und hat viele nützliche Erweiterungen um Python Code schreiben zu können.
Du kannst VS-Code hier runterladen. Ich gehe davon aus, dass du in der Lage bist VS-Code selbständig zu installieren, ansonsten findest auf deren Webseite wunderbare Anleitungen.
Du kannst auch **apt** nutzen, wenn du auf Linux bist.

```
1  user@machine:/path$sudo add-apt-repository "deb [arch=amd64] https://
packages.microsoft.com/repos/vscode stable main"
2  user@machine:/path$ sudo apt update
3  user@machine:/path$ sudo apt install code
```

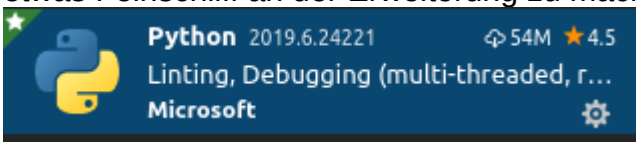
Im Folgenden werden wir ein paar Erweiterungen installieren, um unsere erste Anwendung zu erstellen.

Entwicklungsumgebung Aufsetzen

Nachdem wir nun VS-Code installiert haben, installieren wir noch nachfolgende Erweiterungen.

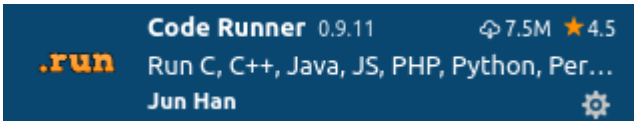
Python

Python ist eine Erweiterung um Syntax farblich hervorzuheben, Python-Code zu debuggen und es enthält einen Linter.
Manchmal zeigt es sogar die korrekte Intellisense an. Da ist wohl noch etwas Feinschliff an der Erweiterung zu machen.

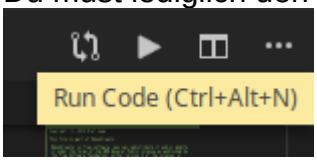


Coderunner

Mit Coderunner bist du in der Lage, deine Anwendung mit nur einem Klick zu starten.

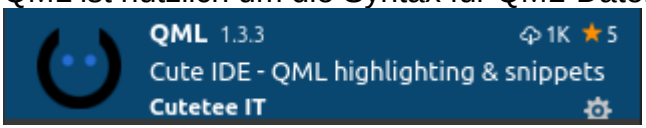


Du must lediglich den "Play" Knopf klicken.



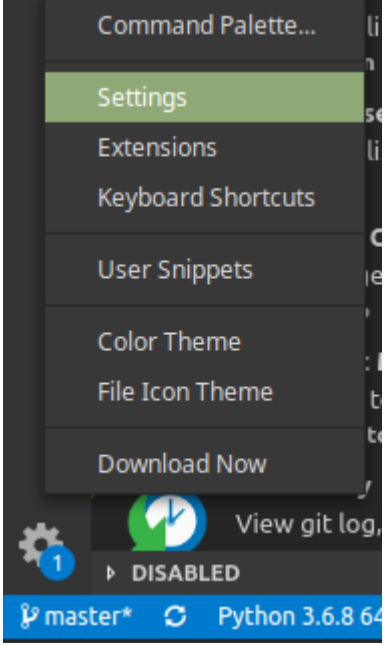
QML

QML ist nützlich um die Syntax für QML-Dateien einzufärben.

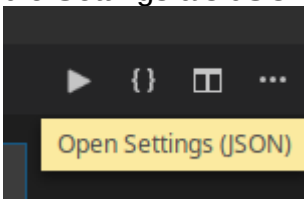


VS-Code Settings

Du kannst die Settings mit einem Klick auf das Icon mit dem Zahnrad öffnen.



Und dann klickst du **"{}"** auf der oberen rechten Seite des Bildschirms um die Settings als JSON-Datei zu öffnen.



Hier sind ein paar nützliche Einstellungen, welche ich in die *settings.json* eingefügt habe.
Das colorTheme ist natürlich Geschackssache.
Die Einstellung vom CoderRunner führt Python mit der *main.py* aus, egal welche Python-Datei gerade im Editor offen ist. Das bedingt natürlich, dass das Projekt eine Datei mit dem Namen *main.py* besitzt und das dies die Startdatei ist.

```
{
    "workbench.colorTheme": "Visual Studio Dark",
    "code-runner.executorMap": {
        "python": "python3 $workspaceRoot/main.py",
    },
    "code-runner.clearPreviousOutput": true,
    "code-runner.saveAllFilesBeforeRun": true,
    "git.autofetch": true,
}
```

Erste Anwendung

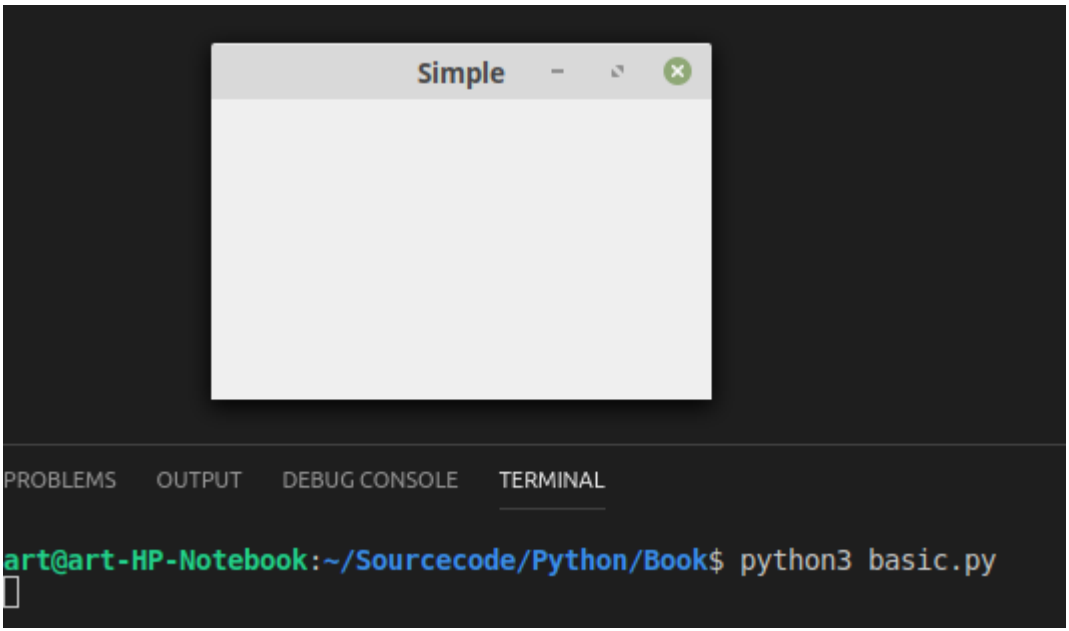
Wir schreiben nun eine simple Anwendung um unsere Umgebung einmal auszuprobieren.
Ich gehe hier nicht weiter ins Detail, versuche aber später im Buch auf die Einzelheiten einzugehen.

basic.py

```
1  import sys
2  from PyQt5.QtWidgets import QApplication, QWidget
3
4
5  app = QApplication(sys.argv)
6  w = QWidget()
7  w.resize(250, 150)
8  w.setWindowTitle('Simple')
9  w.show()
10 app.exec()
```

In diesem Fall, da wir die Python Datei nicht *main.py* genannt haben, führen wir Python in einem Terminal innerhalb von VS-Code aus..

```
user@machine:/path$ python3 basic.py
```



In diesem Beispiel instanziieren wir eine Application, zusätzlich noch ein Widget, welches hier als Fenster dient,setzen die Grösse des Fensters, setzen den Titel des Fensters, machen das Fenster sichtbar und starten den MainLoop.
Im MainLoop wird in einer Schleife auf Ereignisse wie Mausklicks, Tastatureingabe abgefragt und sie dann dem Fenster zur Auswertung übergeben. In diesem Fall wird das Fenster lediglich auf Veränderung der Fenstergrösse, auf das Verschieben des Fensters und das Schliessen des Fensters reagieren, sollten wir auf den grünen Schliessen-Knopf, recht oben in der Ecke, klicken. Hierbei wird dann die MainLoop verlassen und die Anwendung beendet.

Zusammenfassung

Nachdem wir die Entwicklungsumgebung aufgebaut haben, konnten wir die erste PyQt Anwendung erstellen und ausführen.

Teil II - Basis

Viele Unterschiedliche Ansätze

Da Python schon seit 1991 existiert, wurden bereits mehrere Frameworks für die GUI-Programmierung in Python erstellt.
Da gibt es Tkinter, welches eine Brücke zu TK ist. Tkinter wird standardmässig mit Python ausgeliefert.
Dann gibt es noch Kivy, welches eigentlich gute Ansätze, wie zum Beispiel die GUI-Beschreibungssprache kvlang hat, mit der es auf einfache pythonische Weise möglicht ist, das Userinterface zu deklarieren.
Dann gibt es noch BeeWare mit dem u.a. Python in Java-Byte-Code compiliert wird, um auf Android ausgeführt werden zu können.
Da gibt es auch noch Enaml Native, welches man als Pythons Antwort zu React Native sehen kann und dann gibt es noch ein Paar Möglichkeiten um eine Brücke zu Qt zu schlagen, als da wären, PySide welches eine Brücke zu Qt4 baut, PyQt welches ebenfalls eine Brücke zu Qt4 ist und PyQt5, was eine Brücke zu Qt5 darstellt und um das es hier in diesem Buch geht.

Ich werde hier nicht über die Vor- und Nachteile der einzelnen Frameworks schreiben, sondern mich voll auf PyQt5 konzentrieren.

PyQt5 zu benutzen war eine persönliche Entscheidung von mir, da ich bereits ein paar Jahre mit Qt5 und C++ gearbeitet habe.

Qt5 und PyQt5 sind als Open Source Lizenz verfügbar und man kann beide kostenlos nutzen, solange man damit Open Source Software erstellt. Solltest du vorhaben kommerzielle Software zu erstellen, musst du Lizenzen für beide Frameworks erwerben.

Selbst wenn wir Qt5 nutzen, haben wir zwei Optionen Anwendungen damit zu erstellen.

Die erste Option ist es eine Anwendung mit QtWidgets zu erstellen, welches den Desktop als Zielplatform hat und QtQuick, welches einen deklarativen Ansatz wählt um Userinterfaces mittels QML(**Q**t **M**arkup **L**anguage) zu beschreiben und eher für mobile Endgeräte konzipiert wurde.
Da QtQuick derzeit kein Control für einen TreeView und einen TableView besitzt, würde ich nicht empfehlen damit eine Anwendung für den Desktop zu erstellen, ausser man kann auf diese beiden Controls verzichten.

Mit QtQuick kann man ausserdem Behaviours und Transitions deklarieren, welches man heutzutage eher auf mobilen Endgeräten antrifft.

Wenn du einen Design Hintergrund hast, ist wahrscheinlich der QML-Ansatz interessanter für dich, da du hier nicht wirklich Code erzeugen musst.
Bist du eher der CoderTyp, dann ist evtl. QtWidgets die richtige Wahl für dich.

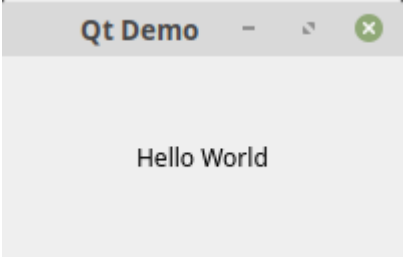
Im folgenden zeige ich dir allerdings gleich vier Varianten Qt Anwendungen zu schreiben.

Hello World (QtWidgets)

Zuerst erstellen wir eine sehr simple QtWidget Anwendung, mit der wir die Worte *Hello World* auf dem Bildschirm ausgeben. Im Gegensatz zu dem ersten Beispiel benutzen wir hier die Klasse QMainWindow anstelle von QWidget. QMainWindow hat zusätzlich noch die Möglichkeit ein Menu, eine Toolbar und eine Statusbar zu nutzen.

QWidget/Basics/main.py

```
import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel
from PyQt5.QtCore import Qt


class MainWindow(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.setWindowTitle("Qt Demo")
        label = QLabel("Hello World")
        label.setAlignment(Qt.AlignCenter)
        self.setCentralWidget(label)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    win = MainWindow()
    win.show()
    sys.exit(app.exec())
```



Das Beispiel ist fast selbsterklärend. Wir instanziieren das Applikations-Objekt indem wir ihm die Argumentenliste der Anwendung übergeben, erzeugen eine Instanz von MainWindow, unserer Fensterklasse. Machen das Fenster mit Aufruf der Methode *show()* auf dem Bildschirm sichtbar und rufen die Hauptschleife der Applikation auf.
In der Init-Methode des Fensters rufen wir zuerst einmal die Init-Methode der VaterKlasse auf, setzen den Titel des Fensters und erzeugen eine Instanz eines Labels, welches hier als zentrales Widget gesetzt wird.
QMainWindow hat neben dem Menu, der Menubar und der Statusbar das zentrale Widget, welches den inneren Bereich des Fensters ausfüllt.
In grösseren Projekte wäre es sinnvoller jede Klasse in einer eigenen Python-Datei zu speichern. Das macht das Projekt übersichtlicher. Unser Fenster würden wir dann in der Datei *mainwindow.py* speichern.

Hello World (QtQuick)

Die Hello World Anwendung für QtQuick besteht aus zwei Dateien. Zuerst haben wir da wieder die *main.py* in der wir die Applikation instanziieren und den MainLoop starten und dann haben wir eine zweite Datei mit dem Namen *view.qml* in der wir das Userinterface definieren werden.

QtQuick/Basics/main.py

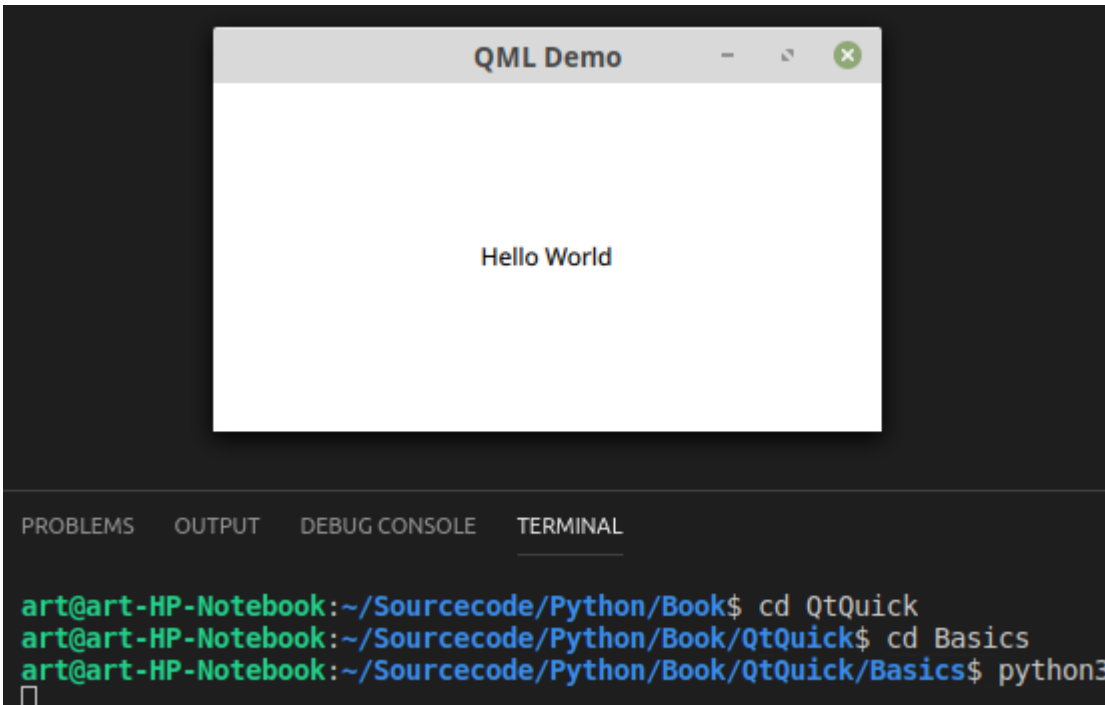
```
1  import sys
2  from PyQt5.QtGui import QGuiApplication
3  from PyQt5.QtQml import QQmlApplicationEngine
4
5
6  if __name__ == "__main__":
7      app = QGuiApplication(sys.argv)
8      engine = QQmlApplicationEngine("view.qml")
9      if not engine.rootObjects():
10         sys.exit(-1)
11     sys.exit(app.exec())
```

QtQuick/Basics/view.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.5

ApplicationWindow {
    visible: true

    Text {
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: "Hello World"
    }
}
```

Um die Anwendung zu starten müssen wir in das Verzeichnis QtQuick/Basics wechseln und die Anwendung wie folgt starten:

```
user@machine:/path$ python3 main.py
```



Diese Anwendung ist ähnlich der QtWidget-Variante bis auf die Tatsache, das wir QGuiApplication anstelle von QApplication benutzen. Und ausserdem nutzen wir QQmlApplicationEngine, um die QML-Datei zu laden. Des Weiteren deklarieren wir wie bereits gesagt das Userinterface mit Hilfe von QML. Auf QML werde ich in diesem Buch nicht näher eingehen, da es

hierfür bereits ausreichend Literatur gibt und es den Rahmen diese Buches sprengen würde.

QWidget und QML Kombinieren

Eine dritte Möglichkeit, Qt Anwendungen zu schreiben ist die Kombination aus QtWidget und QML in dem man ein QQuickView verwendet, um QML innerhalb einer QtWidget Anwendung zu benutzen.

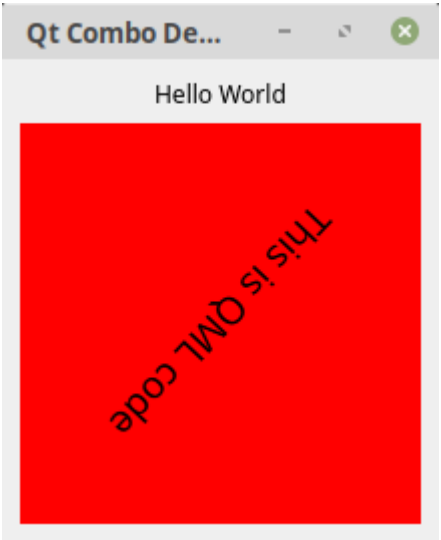
Combo/main.py

```
  import sys
  from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QVBoxLayo
ut, QWidget
  from PyQt5.QtCore import Qt, QUrl
  from PyQt5.QtQuick import QQuickView


  class MainWindow(QMainWindow):
      def __init__(self):
          QMainWindow.__init__(self)
          self.setWindowTitle("Qt Combo Demo")
          widget= QWidget()
          layout = QVBoxLayout()
          view = QQuickView()
          container = QWidget.createWindowContainer(view, self)
          container.setMinimumSize(200, 200)
          container.setMaximumSize(200, 200)
          view.setSource(QUrl("view.qml"))
          label = QLabel("Hello World")
          label.setAlignment(Qt.AlignCenter)
          layout.addWidget(label)
          layout.addWidget(container)
          widget.setLayout(layout)
          self.setCentralWidget(widget)


  if __name__ == "__main__":
      app = QApplication(sys.argv)
      win = MainWindow()
      win.show()
      sys.exit(app.exec())
```

Combo/view.qml

```

```

```
import QtQuick 2.1

   Rectangle {
       id: rectangle
       color: "red"
       width: 200
       height: 200

       Text {
           id: text
           text: "This is QML code"
           font.pointSize: 14
           anchors.centerIn: parent
           PropertyAnimation {
               id: animation
               target: text
               property: "rotation"
               from: 0; to: 360; duration: 5000
               loops: Animation.Infinite
           }
       }
       MouseArea {
           anchors.fill: parent
           onClicked: animation.paused ? animation.resume() : animation.pau
se()
       }
       Component.onCompleted: animation.start()
   }
```



Sei dir bitte bewusst, das man nicht mehrere QQuickViews in einem QWidget Window verwenden sollte, um die Performance nicht negativ zu beeinflussen.

Zusammenfassung

Wir haben drei Möglichkeiten gesehen, um Anwendungen mit PyQt5 zu erstellen.
Der QWidgets-Ansatz wird meist bei Desktop-Anwendungen genutzt. Der QML-Ansatz wird meist genutzt, um Anwendungen für mobile Geräte zu erstellen. Und die Kombination kann genutzt werden um QML innerhalb von Desktop-Anwendungen darzustellen.

Teil III - QML

QML Syntax

Die QML Syntax wurde entwickelt, um Benutzerinterfaces mit einer leicht les- und leicht schreibbaren Syntax, basierend auf Ideen von XAML, Json und Javascript zu nutzen. QML kombiniert alle drei Sprachen in einer.

Das Benutzerinterface wird in QML als Hierarchie deklariert. Jedes Element wird innerhalb eines anderen positioniert. Ereignisse, die von einem Control getriggert werden, wenn der Benutzer zum Beispiel einen Button klickt, können in Javascript kodiert werden.

Um einen Element zu deklarieren, schreiben wir den Namen des Elementes gefolgt von geschweiften Klammern, ähnlich wie in Javascript.

```
Rectangle
{
}
```

Um den Wert einer Eigenschaft zu deklarieren, schreiben wir den Namen der Eigenschaft gefolgt von einem Doppelpunkt und dem Wert.

```
name: value
```

Um einen numerischen Wert zu deklarieren, schreiben wir den numerischen Wert einfach hinter den Doppelpunkt.

```
width: 200
```

Einen Text schreiben wir in Anführungszeichen **"** or **'**.

```
text: "The quick brown fox"
message: 'Hello "Brother"'
```

Farben werden genau so zwischen Anführungszeichen gesetzt und mit dem Hash "#" Zeichen gestartet.

```
```

```
color: "#FF00FF"
```

Hier im Beispiel wurde der Rot-Anteil auf hexadezimal FF (255) gesetzt, der Grün-Anteil ist auf 0 und der Blau-Anteil auch auf FF (255). Man spricht hier auch von RGB. (Rot Grün Blau)

Du kannst die Opacity auch mit einem hexadezimalen Wert setzen. Einfach an das Ende des Farbstrings anhängen.

```
color: "#FF00FFEE"
```

Hier im Beispiel wurde die Opacity auf den Wert EE (238) gesetzt. 0 bedeutet transparent und FF (255) voll sichtbar.

The Id

Ein spezieller Wert in QML ist die **id**.

```
1  ApplicationWindow
2  {
3      id: root
4      Text
5      {
6          text: "The parent is " + root.width + " pixels wide"
7      }
8  }
```

Die **id** is schreibgeschütz und wird einmalig in der Deklaration gesetzt. Sie wird benutzt, um items zu referenzieren. Eine gute Praxis is es, das oberste Element in der Hierarchie mit **root** zu benennen, so das es in allen QML-Objekten gleich heisst.

Kind-Elemente erben das Koordinatensystem vom Vater-Element (parent), dem übergeordnetem Element. Somit sind die x und y Koordinaten immer relativ zum Vater (parent).

Jedes QML-Objekt benötigt genau ein Wurzel-Element (root). Somit kann es zum Beispiel nur ein einziges ApplicationWindow geben.

Kind Elemente

Jedes Element kann Kind-Elemente innerhalb der geschweiften Klammern deklarieren. Auf diese Weise kann ein ganzer Baum an hierarchisch angeordneten Elementen entstehen.

```
Rectangle
{
    color: "red"

    MouseArea
    {
        anchors.fill: parent
    }
}
```

Kind-Elemente können dann wiederum weitere Kind-Elemente enthalten.

Kommentare

Kommentare können mit **//** für eine Zeile oder mit **/* */** für mehrere Zeilen deklariert werden. Genau wie in C, C++ und Javascript.

```
ApplicationWindow
{
    /* This
    ** is a
    ** comment.
    */
    visible: true

    // This is also a comment
}
```

Imports

```
import Namespace Major.Minor
import Namespace Major.Minor as SingletonIdentifier
import "directory"
import "file.js" as ScriptIdentifier
```

Mit einer Import Deklaration werden QML-Objekte mit einer spezifischen Version dazugeladen.

```
import QtQuick 2.5
```

Die Hauptversion deutet auf die QtQuick-Version hin. Die Unterversion deutet auf die Qt-Version hin.
Somit wird im obigen Beispiel QtQuick 2 und Qt mit der Version 5.* importiert.
Die Version 2.12 importiert in diesem Fall QtQuick 2 und Qt 5.12.
Ja, die Versionierung ist etwas verwirrend. Das liegt aber daran, das QML auch abwärtskompatibel sein soll, nehme ich an.

Eigenschaften

Du kannst Eigenschaften mit dem Schlüsselwort **property**, gefolgt von dem Typ, dem Namen und optional mit einem Initialwert deklarieren.
Die Syntax lautet `property <type> <name> : <value>`

```
property int clickCount: 0
```

Eine weitere Möglichkeit ist es das Schlüsselwort **alias** zu nutzen, um die Eigenschaft eines untergeordneten Objektes weiterzureichen.
Die Syntax lautet `property alias <name>: <reference>`

```
property alias text: lable.text
```

Eine alias Eigenschaft benötigt keinen Typ, da der Typ des referenzierten Objektes benutzt wird.

Basic Types

Typ	Beschreibung
bool	Binary true/false value
double	Number with a decimal point, stored in double precision
enumeration	Named enumeration value
int	Whole number, e.g. 0, 11, or -22
list	List of QML objects
real	Number with a decimal point
string	Free form text string
url	Resource locator
var	Generic property type

Property Binding

Eines der wichtigsten Merkmale von QML ist die Eigenschaftsbindung. Ein Wert einer Eigenschaft kann über eine Konstante, einen Ausdruck oder über die Bindung an eine andere Eigenschaft festgelegt werden.

```
Rectangle
{
    width: 200; height: 200

    Rectangle
    {
        width: 100
        height: parent.height
        color: "red"
    }
}
```

Im obigen Fall wurde die Höhe an die Höhe des übergeordneten Objekts gebunden. Wenn wir die Höhe des Elternteils ändern, wird die Höhe des inneren Rectangle automatisch angepasst.

Signals

Empfangen von Signalen und Signalhandlern

Um eine Benachrichtigung zu erhalten, wenn ein Signal für ein Objekt ausgegeben wird, sollte die Objektdefinition einen Signalhandler mit dem Namen `on<Signal>` deklarieren, wobei Signal der Name des Signals ist und der erste Buchstabe groß geschrieben wird. Der Signalhandler sollte den JavaScript-Code enthalten, der ausgeführt werden soll, wenn der Signalhandler aufgerufen wird.

```
        Button
        {
            id: button
            anchors.bottom: parent.bottom
            anchors.horizontalCenter: parent.horizontalCenter
            text: "Change color!"

            onClicked:
            {
                root.color = Qt.rgba(Math.random(), Math.random(), Math.rand
om(), 1);
            }
        }
```

Im obigen Fall hat der Button ein Signal mit dem Namen **clickt ** und wir erstellen einen Signalhandler mit dem entsprechenden Namen **onClicked **.

Signalhandler für Eigenschaftsänderungen

Ein Signal wird ausgegeben, wenn sich der Wert einer QML-Eigenschaft ändert. Diese Art von Signal ist ein Eigenschaftsänderungssignal, und Signalhandler für diese Signale werden in der Form `on<PropertyName>Changed` geschrieben, wobei PropertyName der Name der Eigenschaft ist und der erste Buchstabe groß geschrieben wird.

```
        Rectangle
        {
            id: rect
            width: 100; height: 100
            color: "#C0C0C0"

            TapHandler
            {
```

```
                onPressedChanged: console.log("pressed changed to ",
pressed)
            }
        }
```

Zum Beispiel hat der TapHandler eines Rechtecks eine **pressed** Eigenschaft. Um eine Benachrichtigung zu erhalten, wenn sich diese Eigenschaft ändert, schreiben Sie einen Signalhandler mit dem Namen `onPressedChanged` wie im obigen Beispiel.

Connections

In einigen Fällen möchten wir auf ein Signal außerhalb eines Objekts zugreifen. Daher verwenden wir den Connections-typ.

```
        Button
        {
            id: button
        }

        Connections
        {
            target: button
            onClicked:
            {
                rect.color = Qt.rgba(Math.random(), Math.random(), Math.rand
om(), 1);
            }
        }
```

Attached signal handlers

Ein angeschlossener Signalhandler empfängt ein Signal von einem angeschlossenen Typ.

```
    ApplicationWindow
    {
        Component.onCompleted:
        {
            console.log("The windows title is", title)
        }
    }
```

Im obigen Fall wird das Ereignis onCompleted ausgelöst, wenn das ApplicationWindow vollständig geladen wurde.

Signale für einen benutzerdefinierten QML-Typ

Die Syntax zum Definieren eines neuen Signals lautet: `signal <name>[([<type> <parameter name>[, ...]])`

Ein Signal wird durch Aufrufen des Signals als Methode ausgegeben.

```
// SquareButton.qml
import QtQuick 2.12

Rectangle
{
    id: root
    signal activated(real xPosition, real yPosition)
    property point mouseXY
    property int side: 100
    width: side; height: side

    TapHandler
    {
        id: handler
        onTapped: root.activated(mouseXY.x, mouseXY.y)
        onPressedChanged: mouseXY = handler.point.position
    }
}
```

Das Rechteckobjekt verfügt über ein aktiviertes Signal, das bei jedem Antippen des TapHandlers ausgegeben wird.

Um dieses Signal zu verwenden, deklarieren Sie einen onActivated-Handler im Code, der den SqareButton verwendet.

```
    SqareButton
    {
        color: "#345462"
        x: 200
        y: 0
        onActivated: console.log("Activated at " + xPosition + "," + yPo
sition)
    }
```

Funktionen

Um eine JavaScript-Funktion zu deklarieren, verwenden Sie die folgende Syntax.

```
function <name> ( <parameters> ) { ... }
```

Im nächsten Beispiel haben wir den Text mit der Id txt. Im selben übergeordneten Element haben wir eine MouseArea, die nur zum Abrufen eines Klickereignisses verwendet wird, wenn der Benutzer mit der Maus darauf klickt oder wenn der Benutzer diesen Bereich auf einem Mobiltelefon berührt.

```
property int clickCount: 0

Text
{
    id: txt
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.verticalCenter: parent.verticalCenter
    text: "Right - Bottom "
}

MouseArea
{
    anchors.fill: parent
    onClicked:
    {
        increment()
        txt.text = "clicked " + clickCount + " times"
    }

    function increment()
    {
        clickCount++
    }
}
```

Die MouseArea hat ein clicked-Ereignis, das wir mit dem Präfix "on" codieren. Wenn Sie mehr als eine Codezeile haben, müssen Sie Klammern verwenden. In diesem Fall rufen wir das Funktionsinkrement auf. Die Eigenschaft clickCount deklarieren wir auf oberster Ebene. Es ist eine gute Angewohnheit, alle Variablen oben im Stammelement zu deklarieren.

Interaktion zwischen QML und Python

Sie können fast alles in QML (Javascript) codieren, aber manchmal müssen Sie Geschäftslogik in Python ausführen. Deshalb werde ich Ihnen eine mögliche Brücke zeigen. Wir erstellen eine von QObject abgeleitete Klasse und geben QML über die Kontexteigenschaft eine Instanz dieser Klasse.

```python
import sys
import os
from PyQt5.QtGui import QGuiApplication
from PyQt5.QtQml import QQmlApplicationEngine
from PyQt5.QtCore import QObject, pyqtProperty, pyqtSignal, pyqtSlot


class Bridge(QObject):
    textChanged = pyqtSignal()

    def __init__(self, parent=None):
        QObject.__init__(self, parent)
        self._root = None
        self._cwd = os.getcwd()

    def setRoot(self, root):
        self._root = root

    @pyqtProperty(str, notify=textChanged)
    def cwd(self):
        return self._cwd

    @pyqtSlot("QString")
    def message(self, value):
        print(value + " from QML")
        self._root.updateMessage(value + " from Python")


if __name__ == "__main__":
    bridge = Bridge()
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()
    engine.rootContext().setContextProperty("bridge", bridge)
    engine.load("view.qml")
    roots = engine.rootObjects()
    bridge.setRoot(roots[0])
    if not roots:
        sys.exit(-1)
    sys.exit(app.exec())
```

In QML können Sie die das bridge Objekt verwenden, um mit Python zu kommunizieren.

```
import QtQuick 2.0
import QtQuick.Controls 2.5

ApplicationWindow {
    width: 640
    height: 480
    visible: true
    title: "QML Demo"

    function updateMessage(text)
    {
        txt.text = text
    }

    Text {
        id: txt
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
        text: "cwd: " + bridge.cwd
    }
    MouseArea {
        anchors.fill: parent
        onClicked:
        {
            bridge.message("click")
        }
    }
}
```

`bridge.cwd` verwendet die in Python deklarierte Eigenschaft, um das aktuelle Arbeitsverzeichnis abzurufen.
Und mit `bridge.message` senden wir Daten an den Slot in Python.
Auf der Python-Seite können wir eine Javascript-Funktion mit dem Root-Objekt aufrufen.

```
self._root.updateMessage(value + " from Python")
```

Bildschirmgrößen

Um Elemente auf der Oberfläche zu platzieren, müssen wir sie positionieren. Wenn du aus einer Windows-Umgebung wie WinForms kommst, weißt du, wie du Elemente anhand ihrer x- und y-Koordinaten positionierst. Dies war

in Ordnung für Fenster, bei denen die Bildschirmgröße in den letzten Jahren bekannt war. Wir hatten nur Bildschirmgrößen wie 640 x 480 oder 1024 x 768. Hier hast du versucht, beim Entwerfen eines Dialogfelds die kleinste Bildschirmauflösung zu unterstützen. Auf einem Mobiltelefon haben wir hunderte verschiedener Bildschirmauflösungen und Formfaktoren, daher müssen wir unsere Überlegungen zum Entwerfen eines Dialogs ändern. Außerdem erstellen wir keine Dialoge mehr. Normalerweise erstellen wir Vollbildseiten.

Wenn du auf einem Desktop-Computer entwickelst, musst du den Formfaktor eines Mobiltelefons simulieren. Aufgrund der Tatsache, dass mein Mobiltelefon eine Auflösung von 700 * 1200 und mein Desktop eine Auflösung von 1366 * 768 hat, muss ich eine Bildschirmgröße finden, um den Porträtmodus meines Telefons auf meinem Desktop zu simulieren. Daher stelle ich das ApplicationWindow auf die Größe 350 * 600 ein, damit das Fenster auf meinen Bildschirm passt und den Porträtmodus eines Mobiltelefons nachahmt.

```
ApplicationWindow #
{
    id: applicationWindow
    width: 350
    height: 600
    visible: true
    title: "Anchors Demo"
```

Du solltest eine andere, aber ähnliche Lösung für deine Umgebung finden.

Wenn wir also die direkte Positionierung wie in Windows verwenden würden, in der wir die x- und y-Koordinaten zum Positionieren eines Elements verwendet haben, sollten wir eine andere Lösung für das Telefon verwenden, da wir die Bildschirmauflösung der Zielplattform nicht kennen.

Elemente positionieren

Um ein Element in QML zu positionieren, verwenden wir Anker. Du kannst weiterhin die x- und y-Koordinaten verwenden, dies ist jedoch nur dann sinnvoll, wenn das Element oben links verankert ist. Du kannst ein Element oben, unten, links und rechts verankern. Darüber hinaus kannst du ein Element auch im horizontalen Zentrum und im vertikalen Zentrum verankern.

```
```

```
    Rectangle
        {
            id: rectangle
            width: 100
            height: 100
            color: "#75507b"
            anchors.horizontalCenter: parent.horizontalCenter
            anchors.verticalCenter: parent.verticalCenter
        }
```

Im obigen Beispiel ist die Mitte des Rechtecks an die Mitte des übergeordneten Elements gebunden, das das Anwendungsfenster ist, oder es könnte ein anderes Element sein.

Wenn du ein Objekt unten rechts verankern möchtest, positionierst du es wie folgt.

```
        Rectangle
        {
            width: 100
            height: 100
            color: "#73d216"
            anchors.bottom: parent.bottom
            anchors.bottomMargin: 20
            anchors.right: parent.right
            anchors.rightMargin: 20
        }
```

Die Margin ist der Abstand zwischen dem Element und der Seite der Eltern.

Im Quellcode dieses Buches findest du ein Beispiel mit allen Ankern unter *QtQuick/Anchors*.

Controls

Das am häufigsten verwendete Steuerelement auf einem Mobiltelefon ist möglicherweise die Listbox. Hier ist ein Beispiel, das zeigt, wie es mit einem statischen Modell verwendet wird.

```
    import QtQuick 2.0
    import QtQuick.Controls 2.5
    import QtQuick.Layouts 1.1

    ApplicationWindow {
        width: 640
```

```
        height: 480
        visible: true
        title: "QML Listview Demo"

        ListView
        {
            clip: true
            anchors.fill: parent
            anchors.margins: 5
            spacing: 5

            delegate: listDelegate

            Component
            {
                id: listDelegate

                RowLayout
                {
                    anchors.left: parent.left
                    anchors.right: parent.right
                    anchors.margins: 10
                    spacing: 10

                    CheckBox {}
                    Label { text: itemType; color: "#888"; font.italic:
true }
                    Label { text: itemName; Layout.fillWidth: true }
                    Label { text: itemPath }
                    ComboBox { model: itemVersions; Layout.preferredWidth: 9
0 }
                }
            }

            model: ListModel
            {
                ListElement
                {
                    itemType: "asset"
                    itemName: "First entry"
                    itemPath: "/documents/fe.md"
                    itemVersions: []
                }
                ListElement
                {
                    itemType: "asset"
                    itemName: "Second entry"
                    itemPath: "/documents/se.md"
                    itemVersions: []
                }
```

```
                ListElement
                {
                    itemType: "asset"
                    itemName: "Third entry"
                    itemPath: "/documents/te.md"
                    itemVersions: []
                }
            }
        }
    }
```

Um die Liste zu rendern, entwerfen wir eine Komponente, die als Delegat verwendet wird. Das Modell, das die Daten liefert, die wir hier in diesem Beispiel deklarativ verwenden, um das Verständnis zu erleichtern.

Als nächstes werden wir ein in Python erstelltes Modell verwenden.

```
import sys
from PyQt5.QtGui import QGuiApplication
from PyQt5.QtQml import QQmlApplicationEngine
from PyQt5.QtCore import Qt, QObject, pyqtProperty, pyqtSignal,
pyqtSlot, QAbstractListModel, QModelIndex


class Model(QAbstractListModel):
    def __init__(self, items, parent=None):
        super(Model, self).__init__(parent)
        self._items = items

    def rowCount(self, parent=None):
        return len(self._items)

    def data(self, index, role=None):
        role = role or QModelIndex()

        if role == Qt.UserRole + 0:
            return self._items[index.row()]["type"]

        if role == Qt.UserRole + 1:
            return self._items[index.row()]["name"]

        if role == Qt.UserRole + 2:
            return self._items[index.row()]["path"]

        if role == Qt.UserRole + 3:
            return self._items[index.row()]["versions"]

    def roleNames(self):
```

```
        return {
            Qt.UserRole + 0: b"itemType",
            Qt.UserRole + 1: b"itemName",
            Qt.UserRole + 2: b"itemPath",
            Qt.UserRole + 3: b"itemVersions",
        }


if __name__ == "__main__":
    items = [
        {
            "type": "asset",
            "name": "shapes",
            "path": "c:/users/Roy/Desktop/shapes.ma",
            "versions": ["v001", "v002", "v003"]
        },
        {
            "type": "asset",
            "name": "shapes1",
            "path": "c:/users/Roy/Desktop/shapes.ma",
            "versions": ["v001", "v002", "v003", "v004"]
        },
        {
            "type": "asset",
            "name": "shapes2",
            "path": "c:/users/Roy/Desktop/shapes.ma",
            "versions": ["v001", "v002", "v003"]
        },
    ]
    model = Model(items)

    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine()
    engine.rootContext().setContextProperty("mymodel", model)
    engine.load("view.qml")
    roots = engine.rootObjects()
    if not roots:
        sys.exit(-1)
    sys.exit(app.exec())
```

In der Ansicht müssen wir nur das Modell ändern.

```
model: mymodel
```

Das Modellobjekt wurde jetzt als Kontexteigenschaft festgelegt.

Weitere Informationen finden Sie in den Qt-Dokumentationen und Beispielen. All QML Types

Zusammenfassung

Wir haben die Grundlagen von QML gesehen. Wir können jetzt einfache Apps mit QML und Python erstellen.

Teil IV - Installation auf Android

Nun werden wir ein APK (**A**ndroid **P**ac**k**age) erstellen und es auf einem Android Gerät installieren.
Hierfür müssen wir ein paar zusätzliche Komponenten installieren.

Installation von pyqtdeploy 2.4

pyqtdeploy is eine Anwendung die von der selben Firma wie PyQt5 zur Verfügung gestellt wird. Du kannst pyqtdeploy mit pip installieren.

```
user@machine:/path$ pip3 install pyqtdeploy
```

Installation von Java JDK 8

Du solltest alles, was du zum Installieren von Java JDK 8 brauchst, hier finden: https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

Nur um es zu erwähnen, ich hatte Probleme ein Paket zu bilden, weil ich Java JDK 10 installiert hatte. Ich habe es deinstallieren müssen.

Installation von Android SDK

Um das Android SDK zu installieren gehe bitte zu https://developer.android.com/studio scrolle runter zu *Command line tools only* und lade die spezifische Version für deine Platform runter.
Dann entpacke diese Datei irgendwo auf deinem Computer.
Nachdem du alles entpackt hast, wechsle bitte in das Verzeichnis *../Sdk/tools/bin* und starte den sdkmanager um zusätzliche Werkzeuge zu installieren:

```
user@machine:/path$ ./sdkmanager "platform-tools" "platforms;android-28"
```

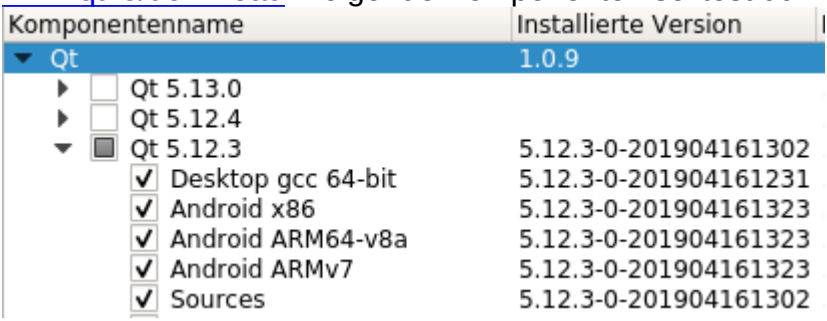
Installation von Android NDK

Um das Android NDK zu installieren, gehe bitte auf diese Seite https://developer.android.com/ndk/downloads scrolle runter zu *Older Versions*, und klicke den Link *NDK Archives*, akzeptiere die Terms und lade *Android NDK, Revision 19c* für deine Platform runter (Unglücklicherweise funktioniert Version 20 nicht für unsere Zwecke). Nachdem du alles geladen hast, entpacke die Datei bitte in das SDK-Verzeichnis in das Unterverzeichnis *ndk-bundle*.
Deine Android-Verzeichnisstruktur sollte dann wie folgt aussehen:

```
└── Sdk
    ├── build-tools
    ├── emulator
    ├── licenses
    ├── ndk-bundle
    │   ├── build
    │   ├── meta
    │   ├── platforms
    │   ├── prebuild
    │   └── ...
    ├── patcher
    └── ...
```

Installation von Qt

Nun benötigen wir auch Qt selber. Du kannst es hier runterladen: https://www.qt.io/download. Folgende Komponenten solltest du installieren.



Du benötigst Desktop gcc 64-bit um die App auf deinem Rechner zu testen.
Du benötigst Android x86 um die App im Simulator testen zu können.
Du benötigst Android ARM64 um die App auf einem 64 bit Gerät zu testen.
Du benötigst Android ARMv7 um die App auf einem 32 bit Gerät zu testen.

Source Pakete runterladen

Nachfolgende Pakete musst du lediglich runterladen aber nicht auspacken.

- openssl-1.0.2s.tar.gz from https://www.openssl.org/source/
- PyQt3D_gpl-5.12.tar.gz from https://www.riverbankcomputing.com/software/pyqt3d/download
- PyQt5_gpl-5.12.1.tar.gz from https://www.riverbankcomputing.com/software/pyqt/download5
- PyQtChart_gpl-5.12.tar.gz from https://www.riverbankcomputing.com/software/pyqtchart/download
- PyQtDataVisualization_gpl-5.12.tar.gz from https://www.riverbankcomputing.com/software/pyqtdatavisualization/download
- PyQtPurchasing_gpl-5.12.tar.gz from https://www.riverbankcomputing.com/software/pyqtpurchasing/download
- Python-3.7.2.tar.xz from https://www.python.org/downloads/source/
- QScintilla_gpl-2.11.1.tar.gz from https://www.riverbankcomputing.com/software/qscintilla/download
- qt-everywhere-src-5.12.2.tar.xz from http://download.qt.io/official_releases/qt/5.12/5.12.2/single/qt-everywhere-src-5.12.2.tar.xz.mirrorlist
- sip-4.19.15.tar.gz from https://www.riverbankcomputing.com/software/sip/download
- zlib-1.2.11.tar.gz from https://zlib.net/

Den Build-Script erstellen

Passe bitte die Pfade für deine Zwecke an.

Deploy/build.sh

```
export ANDROID_NDK_ROOT=/home/art/Android/Sdk/ndk-bundle
export ANDROID_NDK_PLATFORM=android-28
export ANDROID_SDK_ROOT=/home/art/Android/Sdk
python3.7 build.py --target android-32 --installed-qt-dir /home/art/Qt/5.12.3 --no-sysroot --verbose --source-dir ./external-sources
```

external-sources zeigt auf ein Verzeichnis, in dem alle Pakete aus dem vorherigen Schritt gespeichert wurden.
Installierst du auf einem Android Gerät mit 64 bit, dann ändere bitte das target zu android-64.
--installed-qt-dir zeigt auf das Verzeichnis in dem Qt installiert wurde.

Erstelle den build.py Script

Den build.py habe ich von hier: https://pypi.org/project/pyqtdeploy/#files aus dem Demo-Projekt. Ich habe dort folgende Zeilen hinzugefügt:

Deploy/build.py

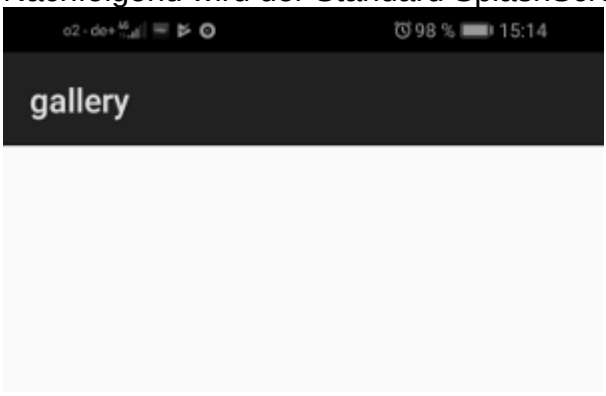
```
    ...
    run(['pyqtdeploy-build', '--target', target, '--sysroot', sysroot_dir, '--build-dir', build_dir, 'demo.pdy'])

    # copy the main.qml to a directory where androiddeployqt will find it to add required libraries based on the import statements
    cp = "cp " + os.path.join(dir_path, "view.qml") + " " + os.path.join(dir_path, build_dir)
    run([cp])
    # append the ANDROID_PACKAGE to the .pro file
    with open(os.path.join(dir_path, build_dir, "main.pro"), "a") as fp:
        fp.write("\ncontains(ANDROID_TARGET_ARCH, armeabi-v7a) {\nANDROID_PACKAGE_SOURCE_DIR = " + os.path.join(dir_path, "android") + "\n}")

    os.chdir(build_dir)
    ...
```

Es ist wichtig, das das Programm androiddeployqt, welches zu Qt gehört und von pyqtdeploy aufgerufen wird, eine QML-Datei in diesem Verzeichnis findet. Das Programm scannt alle nötigen QML-Dateien nach import statements, um nötige shared libraries mit in das APK zu packen.

Du findest das komplette build.py und alle anderen Source-Dateien in diesem Github Repo: https://github.com/Artanidos/DevAndroidPythonBook

Mit dem ANDROID_PACKAGE_SOURCE_DIR deklarieren wir, das wir Android spezifische Dateien wie zum Beispiel *AndroidManifest.xml* und die Icons in einem bestimmten Verzeichnis abgelegt haben.
Ausserdem haben wir ein Theme dort abgelegt. Dieses Theme hat nur einen wichtige Eintrag: *Theme.DeviceDefault.Light.NoActionBar*. Hiermit verhindern wir, das der Standard SplashScreen von Qt angezeigt wird, wenn die App gestartet wird.

Nachfolgend wird der Standard SplashScreen von Qt gezeigt.



Wir ändern diesen Screen in einen schlichten weissen Hintergrund, da wir ansonsten drei verschiedene Screens beim Starten der Anwendung erhalten würden. Einmal den oberen, dann wird kurz ein weisser Screen gezeigt und dann das eigentlich Programm. Wir reduzieren die Anzahl auf lediglich zwei unterschiedliche Screens, damit es beim Starten nicht so flackert.

Resource Datei erstellen

Die Resource Datei enthält unsere QML-Datei als Python Resource.
Um eine Resource Datei zu erstellen muss man folgenden Befehl ausführen.

```
user@machine:/path$ pyrcc5 main.qrc -o lib/main_rc.py
```

Dieser Schritt ist nötig, damit die QML Datei später auf dem Gerät gefunden wird. Nach dem wir nun eine Resource-Datei haben, müssen wir die main.py wie folgt anpassen.

DynPy/main.py

```
1  import sys
2  import os
3  import lib.main_rc
4  from PyQt5.QtGui import QGuiApplication
5  from PyQt5.QtQml import QQmlApplicationEngine
6
7  if __name__ == "__main__":
8      sys_argv = sys.argv
9      sys_argv += ['--style', 'material']
10     app = QGuiApplication(sys.argv)
11     view =  "/storage/emulated/0/view.qml"
12     if os.path.exists(view):
```

```
            # we are trying to load the view dynamically from the root of
the storage
            engine = QQmlApplicationEngine(view)
            if not engine.rootObjects():
                sys.exit(-1)
        else:
            # if the attempt to load the local file fails, we load the
fallback
            engine = QQmlApplicationEngine(":/view.qml")
            if not engine.rootObjects():
                sys.exit(-1)
        sys.exit(app.exec())
```

Mit `import lib.main_rc` fügen wir die Resource hinzu.
Und mit den Doppelpunkt vor **:**/view.qml teilen wir Qt mit, die QML aus der Resource zu laden.

Das Material muss für Android über die args in der Methode QGuiApplication() gesetzt werden. (wird allerdings in der App noch nicht benutzt) Damit du die App auf deinem Handy weiterentwickeln kannst, kann du die Datei view.qml auf deinem Handy speichern bzw. anlegen und dort editieren.
Du legst sie einfach in das Wurzelverzeichnis auf deinem Handy. Das sollte dann der Pfad "/storage/emulated/0/" sein, ansonsten einfach anpassen.

Die App schaut beim Starten, ob an dem besagten Pfad die Datei view.qml liegt und lädt sie, wenn vorhanden. Ansonsten läft die App die view.qml aus der Resource.

DynPy/viel.qml

```
import QtQuick 2.5
import QtQuick.Controls 2.0
import QtQuick.Controls.Material 2.1
import QtQuick.Layouts 1.0
import QtMultimedia 5.4

ApplicationWindow
{
    visible: true

    Text
    {
        anchors.centerIn: parent
        text: "DynPy"
    }
}
```

Eine Projekt-Datei erstellen

Die Projektdatei hat den Namen demo.pdy und wird von pyqtdeploy benutzt, um alle nötigen Python Pakete einzufrieren.

Deploy/demo.pdy

```
<?xml version='1.0' encoding='utf-8'?>
<Project usingdefaultlocations="1" version="7">
<Python major="3" minor="7" patch="2" platformpython="" />
<Application entrypoint="" isbundle="0" isconsole="0" ispyqt5="1" name="
" script="main.py" syspath="">
<Package name="lib">
<PackageContent included="1" isdirectory="0" name="__init__.py" />
<PackageContent included="1" isdirectory="0" name="main_rc.py" />
<Exclude name="*.py" />
<Exclude name="*.qml" />
<Exclude name="*.sh" />
<Exclude name="*.pdy" />
<Exclude name="*.json" />
<Exclude name="*.qrc" />
<Exclude name="build-android-32" />
<Exclude name="sysroot-android-32" />
</Package>
</Application>
<PyQtModule name="QtWidgets" />
<PyQtModule name="QtNetwork" />
<PyQtModule name="QtAndroidExtras" />
<PyQtModule name="QtSvg" />
<PyQtModule name="QtQuick" />
```

```
    <PyQtModule name="QtQml" />
    <PyQtModule name="Qt" />
    <PyQtModule name="QtQuickWidgets" />
    <PyQtModule name="QtSensors" />
    <PyQtModule name="QtBluetooth" />
    <StdlibModule name="http.server" />
    <StdlibModule name="http" />
    <StdlibModule name="ssl" />
    <StdlibModule name="sysconfig" />
    <StdlibModule name="zlib" />
    <StdlibModule name="importlib.resources" />
    <StdlibModule name="os" />
    <StdlibModule name="marshal" />
    <StdlibModule name="imp" />
    <StdlibModule name="logging" />
    <StdlibModule name="logging.config" />
    <StdlibModule name="logging.handlers" />
    <StdlibModule name="contextlib" />
    <StdlibModule name="urllib" />
    <StdlibModule name="urllib.request" />
    <StdlibModule name="traceback" />
    <ExternalLib defines="" includepath="" libs="-lz" name="zlib" target="an
droid" />
    </Project>
```

Eine wichtige Sache hier ist die Tatsache, das wir ein Verzeichnis benötigen, in dem alle Python-Dateien unserer Anwendung zu finden sind. Wir benutzen hier das Verzeichnis *lib*. In diesem Verzeichnis sollte sich auch eine leere Datei mit dem Namen *__init__.py* befinden.
Alle benötigten PyQt Module sollten in dieser Projektdatei aufgelistet werden. (Wir benötigen nicht alle, aber evtl. später)
Ausserdem werden hier alle benötigten Python Pakete gelistet.
Die Datei kann mit dem Programm pyqtdeploy erstellt werden:

```
user@machine:/path$ pyqtdeploy
```

sysroot.json erstellen

Diese Datei enthält alle Einstellungen um die das sysroot Verzeichnis zu erstellen. Die sysroot enthält ein paar Werkzeuge um das APK zu erstellen.

Deploy/sysroot.json

```
{
    "Description": "The sysroot for the DynPy application.",
```

```
    "android|macos|win#openssl": {
        "android#source":   "openssl-1.0.2s.tar.gz",
        "macos|win#source": "openssl-1.1.0j.tar.gz",
        "win#no_asm":       true
    },

    "linux|macos|win#zlib": {
        "source":               "zlib-1.2.11.tar.gz",
        "static_msvc_runtime":  true
    },

    "qt5": {
        "android-32#qt_dir":        "android_armv7",
        "android-64#qt_dir":        "android_arm64_v8a",

        "linux|macos|win#source":   "qt-everywhere-src-5.12.2.tar.xz",
        "edition":                  "opensource",

        "android|linux#ssl":        "openssl-runtime",
        "ios#ssl":                  "securetransport",
        "macos|win#ssl":            "openssl-linked",

        "configure_options":        [
                "-opengl", "desktop", "-no-dbus", "-qt-pcre"
        ],
        "skip":                     [
                "qtactiveqt", "qtconnectivity", "qtdoc", "qtgamepad",
                "qtlocation", "qtmultimedia", "qtnetworkauth",
                "qtremoteobjects",
                "qtscript", "qtscxml", "qtserialbus",
                "qtserialport", "qtspeech", "qttools",
                "qttranslations", "qtwayland", "qtwebchannel", "qtwebeng
ine",
                "qtwebsockets", "qtwebview", "qtxmlpatterns"
        ],

        "static_msvc_runtime":      true
    },

    "python": {
        "build_host_from_source":   false,
        "build_target_from_source": true,
        "source":                   "Python-3.7.2.tar.xz"
    },

    "sip": {
        "module_name":  "PyQt5.sip",
        "source":       "sip-4.19.15.tar.gz"
    },
```

```
    "pyqt5": {
        "android#disabled_features":    [
                "PyQt_Desktop_OpenGL", "PyQt_Printer", "PyQt_PrintDialog",
                "PyQt_PrintPreviewDialog", "PyQt_PrintPreviewWidget"
        ],
        "android#modules":              [
                "QtQuick", "QtCore", "QtGui", "QtNetwork", "QtPrintSupport", "QtWidgets",
                "QtAndroidExtras", "QtQuickWidgets", "QtSvg", "QtBluetooth", "QtNetwork", "QtSensors",
                "QtQml"
        ],

        "source":                       "PyQt5_*-5.12.1.tar.gz"
    }
  }
```

APK Erstellen

Wenn alles glatt laufen sollte sind wir nun in der Lage, das APK zu erstellen und es auf dem Gerät zu installieren. Hierfür lassen wir folgenden Befehl im Terminal laufen: `./build.sh` Stelle sicher, das *build.sh* ausführbar ist. Zuerst erzeugt dieser Script die sysroot mit den Werkzeugen und Bibliotheken und danach wird das eigentliche APK erstellt. Wenn du das APK zum zweiten mal bilden möchtest, kannst du den ersten Schritt überspringen, in dem du in der Datei *build.sh* das Flag --no-sysroot einfügst:

```
--no-sysroot
```

Installaliere das APK auf einem Gerät

Wenn du das APK erfolgreich bilden konntest, dann kannst du es auf deinem Android-Gerät installieren.
Stell zuerst den Entwicklermodus auf deinem Gerät ein.
Hierfür öffnest du die Einstellungen-App. Dann öffnest du den Menupunkt "Über das Telefon", welches du wohl unter "System" finden wirst, je nach Version von Android.
Dort tippst du sieben mal auf die "Build-Nummer" und eine Meldung zeigt dir an, das der Entwickler-Modus aktiviert wurde.
Suche nun den Menupunkt, "Entwickleroptionen" und verbinde das Gerät mit deinem Computer.

Suche in den Optionen nach "USB-Konfiguration auswählen" und aktiviere Media Transfer Protocol (MTP). Normal steht diese Option auf "Nur Laden". Dann aktiviere das "USB-Debugging".
Dein Gerät kann nun mit dem Befehl adb gefunden werden:

```
user@machine:/path$ adb devices
List of devices attached
* daemon not running; starting now at tcp:5037
* daemon started successfully
5WH6R19329010194    device
```

Wenn dein Gerät gelistet wird, dann kopiere bitte den linken Teil mit der Nummer und vervollständige den nachfolgen Befehl:

```
user@machine:/path$ adb -s 5WH6R19329010194 install /home/art/Sourcecode/
Python/Book/Deploy/build-android-32/dynpy/build/outputs/apk/debug/demo-
debug.apk
```

Der Pfad muss natürlich noch angepasst werden, er wird vom build.py ausgegeben. Den kompletten Befehl habe ich in die Datei *deploy.sh* kopiert, um ihn wiederzuverwenden.

Nun sollte das Programm in Form eines Icons auf dem Gerät sichtbar sein.

GUI Von WebServer Laden

Du hast bei QML auch die Möglichkeit, die Datei dynamisch von einem WebServer zu laden. Damit kannst du zum Beispiel die Benutzer zwingen, alle die selbe Version zu nutzen. Das funktioniert dann analog zu einer ganz normalen Webseite. Hier der Code:

```
import sys
import os
import lib.main_rc
from PyQt5.QtCore import QUrl
from PyQt5.QtGui import QGuiApplication
from PyQt5.QtQml import QQmlApplicationEngine

def ready(object, url):
    if not object:
        print("Remote view could not be loaded")
        sys.exit(-1)

def warnings(warnings):
```

```
        for warn in warnings:
            print("Warning:", warn.toString())

    if __name__ == "__main__":
        sys_argv = sys.argv
        sys_argv += ['--style', 'material']
        app = QGuiApplication(sys.argv)
        remoteview = "https://raw.githubusercontent.com/Artanidos/
PythonAndroidBook/master/DynPy/remote/view.qml"
        engine = QQmlApplicationEngine()
        engine.objectCreated.connect(ready)
        engine.warnings.connect(warnings)
        engine.load(QUrl(remoteview))
        sys.exit(app.exec())
```

Die print Methode funktioniert natürlich nur auf Linux. Aber zumindest kannst du im Fehlerfall ja lokal testen, ob das QML fehlerfrei ist. Ich habe die geänderte *view.qml* im Verzeichnis remote abgelegt und źu Github hochgeladen. Du kannst die Datei mit der RAW-Adresse dynamisch in die App laden. Hier kannst du natürlich deinen eigenen Webserver nutzen.

Zusätzlich werte ich in der Methode ready() aus, wenn die Datei geladen wurde, ob das object einen Wert hat. Und ausserdem überprüfe ich in der Methode warnings() die Warnungen und Fehler.

Zusammenfassung

Nun haben wir hoffentlich eine Anwendung auf einem Android-Gerät installiert und zum Laufen gebracht. Wie man eine Anwendung unter Android installiert hat mich mehrere Tage gekostet.
Solltest Du auch Probleme damit haben, kannst du mich gerne kontaktieren. Eventuell kann ich dir ja helfen und die Erkenntnisse für andere in dieses Buch packen, damit die anderen Leser nicht die selben Fehler bekommen.

Nachwort

Ich bin froh, dass du bis hierher gelesen hast.
Ich hoffe, das dir dieses Buch helfen konnte, ein paar neue Möglichkeiten der Softwareentwicklung kennen zu lernen.
Ich wünsche dir viel Erfolg auf deinem weiteren Weg.
Wenn du das Buch magst dann würde ich mich sehr freuen, wenn du eine kurze Rezession hinterlassen könntest, damit andere Menschen auch dieses Buch finden können.

Über den Autor

Olaf Art Ananda, ist 1963 in Hamburg, Deutschland geboren und arbeitet schon seit über 30 Jahren als Softwareentwickler. Er hat mit C angefangen und dann noch Assembler gelernt, um die Programme zu beschleunigen. Nachdem er so gängige Programmiersprachen wie Java, C# und Objective-C erlernt hat, kam er dann schließlich 2016 zu C/C++ zurück und startete Applikationen mit Qt5 zu erstellen.
Qt5 war für ihn das ideale Framework um seine Fähigkeiten, die er in seinem Studium 2013 in "Human Computer Interaction Design" gelernt hatte, umzusetzen.

Nachdem er zum ersten Mal mit Python in Form von Plugins für seine Anwendungen in Berührung kam, dauerte es noch weitere zwei Jahre, bis er Python so richtig kennen lernte.
Heute liebt er die Einfachheit dieser Sprache um wesentlich schneller Anwendungen zu erstellen als damals in C++.

Olaf hat für mehrere Top 500 Unternehmen wie Dupont, Dresdner Bank, Commerzbank und Zürcher Kantonalbank gearbeitet, um nur einige zu nennen. Nach seinem Burnout und einer Nahtoderfahrung beschloss er, nicht mehr für Profit zu arbeiten. Seit 2016 schreibt er Open Source Software wie den AnimationMaker, den FlatSiteBuilder und den EbookCreator. Er hat auch die folgenden Bücher geschrieben: Camp Eden - Wie wir unsere Paradies wiedererschafft haben und Step Out - Guideline to step out of the system. Seit 2016 lebt er in seinem Wohnmobil, derzeit in Portugal, und spielt auf der Straße Gitarre für ein paar Münzen. Das ist ein leichtes Leben.

Impressum

Olaf Japp
japp.olaf@gmail.com