Table of Contents

1. Table of Contents
2. Preface
    1. Copyright
    2. Motivation
    3. Who This Book Is For
    4. Who This Book Is Not For
    5. How This Book Is Organized
    6. Conventions Used In This Book
    7. Using Code Examples
    8. How To Contact Me
    9. Acknowledgements
3. Part I - Prolog
    1. Installation
    2. Development Environment Setup
        1. Python
        2. Coderunner
        3. QML
        4. VSCode Settings
    3. First App
    4. Summary
4. Part II - Basics
    1. Many Different Approaches
    2. Hello World (QtWidgets)
    3. Hello World (QtQuick)
    4. Combining QWidget and QML
    5. Summary
5. Part III - QML
    1. QML Syntax
        1. The Id
        2. Child Objects
        3. Comments
        4. Import Statements
    2. Properties
    3. Basic Types
    4. Property Binding
    5. Signals
        1. Receiving signals and signal handlers
        2. Property change signal handlers
        3. Connections
        4. Attached signal handlers

        5. Signals for a custom QML type
    6. Functions
    7. Interaction between QML and Python
    8. Screen size
    9. Positioning Items
    10. Controls
    11. Summary
6. Part IV - Deploment
    1. Deploying a dynamic app to Android
    2. Install pyqtdeploy 2.4
    3. Install Java JDK 8
    4. Install Android SDK
    5. Install Android NDK
    6. Install Qt
    7. Download Source packages
    8. Create the build script
    9. Create the build.py script
    10. Create the resource file
    11. Create a project file
    12. Create sysroot.json
    13. Build the APK
    14. Install APK to device
    15. Summary
7. Afterwords
8. About The Author

Preface

Copyright

Motivation

After coding for over 30 years now, I came from C, Assembler, Clipper, Powerbuilder, Java, C#, Objetive-C, C++ to Python. And guess what...
I love Python
It would be easy for me to develop native apps using Java, C++ or Objective-C and I am also able to learn Kotlin, Dart or Swift, but things are much easier when you just use Python. I have done a Django tutorial which is a web framework and the ease they have build a model class to be stored in an SQL server has stunned me. I just had to write a model class with a few line of code, then I had to run a scaffolding job which creates corresponding tables on the SQL server and after that I could run the Python interpreter, import my model class, write some properties and save the instance. No SQL coding at all. I have been hooked :-) Then I learned about generators, comprehension and meta programming and Python got me.

Btw...to be able to write this book I wrote an application called EbookCreator. This app is also using PyQt5, QtWigets and I am using QML to serialize book projects, it is open source and you may have a look at the source code for inspiration. You are able to find the sources on github.

When I wrote my first book about Python, I have seen a video with "Uncle" Bob Martin about programming in the future. He says that every 5 years the count of software developers on this planet double. That means that there are at least 50% of all developers that are inexperienced. He also said that we as a mature developer are responsible when for example a self driving car is killing other people because of software errors. That heard today I started to teach other developers and I am starting with this book.

Who This Book Is For

If you are able to write basic Python code and you are interested in developing apps with user interfaces for Android in Python this book is right for you. This book does not assume that you are familiar with Qt. If you want, try out all of the examples in this book it would be a great benefit if you are also working on a Linux machine like me, but MacOS and Windows should also be ok if your are able to search for help in the internet for OS specific differences.

Who This Book Is Not For

If you are new to Python and programming you should consider reading beginner books for Python and watch tutorials on Youtube before continuing to read this book. I am not going into Python specific details in this book.
This is a NO-BUDGET project and I am not a native english speaker.
So if you like to complain about type errors or grammar you will like my book.

```
// irony off
```

How This Book Is Organized

Here are the main topics in each part of the book. First in ***Part I*** you are going to install all necessary software to be able to build applications with PyQt5 and Python. In ***Part II*** you will learn about two different approaches to build apps. In ***Part III*** you will learn to code apps using QML and Python. In ***Part IV*** you will learn to to create an APK which you can deploy to an Android device.

Conventions Used In This Book

The following typographical conventions are used in this book.

Italic

Used for filenames and paths

```
Constant width
```

Used for code samples

Also I am using the Pascal convention to put brackets so that the code is more readable. This is a habit from my C/C++ and C# programming experience.

```
Item
{
    SubItem
    {

    }
}
```

Using Code Examples

All code examples are available on github .

How To Contact Me

If you have any questions or comments don't hesitate to send me an email with your questions or comments to the following address: artanidos@gmail.com

Acknowledgements

First of all I am very thankful to my body which points me to the right ways in life. I know this sounds a bit weird but after learning to be a mechanic many years ago my back hurts after I have worked way too hard and after a half year being ill I decided to study to build machines and then I have learned to do programming just to solve a mathematical problem. Since that time I decided to be a software developer. Then 5 years ago my body told me to stop working for profit after I had my second burnout. Now I have got much time to develop open source software, try out new stuff and write books like this.

I am very thankful to Guido van Rossum for inventing Python in 1991.

I am also very thankful to all Pythonistas out there who are creating tutorials and videos on YouTube where one is able to learn Python for free.

Part I - Prolog

Installation

Here we are going to install all necessary software to be able to write code and test it at least on the desktop. To be able to also deploy the app to other devices we will install additional software in the deployment chapter later in this book.

I assume that you already have installed Python3 and Pip. If not you should find all necessary information on the Python website. We need version 3.7. I also assume that you able to install software using pip.

First we will install PyQt5 which comes together with Qt5 to be able to code applications for the desktop.

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

Then we will install Visual Studio Code which is free and open source and has many good extensions to develop Python code. You can download VSCode here. I assume that you are able to install VSCode on your own. Otherwise you will find installation instructions on this website. You can also use apt if you are on a Linux system. Use google search if there are some prerequisites missing.

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

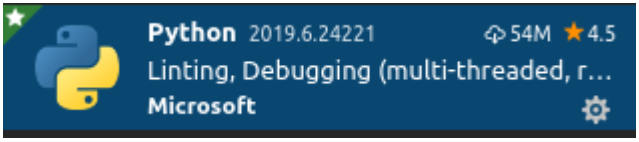
In the next chapter we will setup the environment to be able to write our first app.

Development Environment Setup

After installing VSCode you should install some nifty extensions like these.

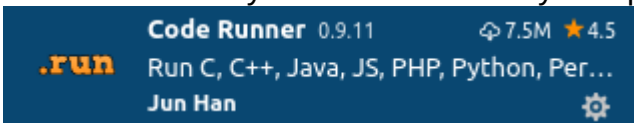
Python

Python is a syntax hiliter, debugger and linter. Sometimes it also shows you the correct intellisense.

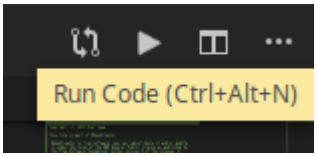


Coderunner

With coderunner you are able to start your app with just on click.

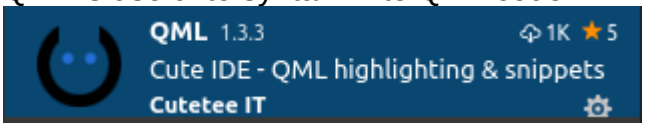


You just have to click the "Play" button.



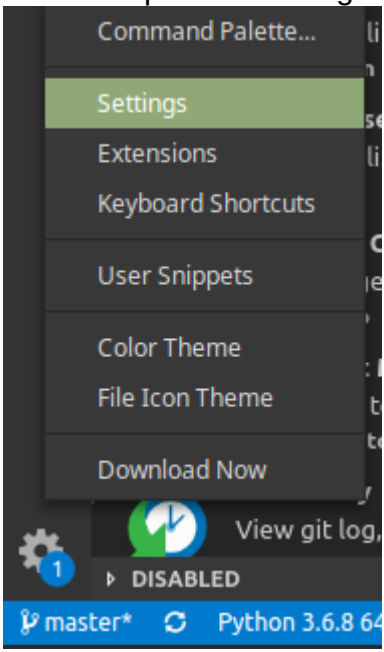
QML

QML is useful to syntax hilite QML code.

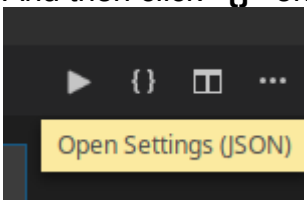


VSCode Settings

You can open the settings file using this menu.



And then click **"{}"** on upper right site of the screen to open the json file.



Here are some useful setting which I have entered into the *settings.json*.

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

First App

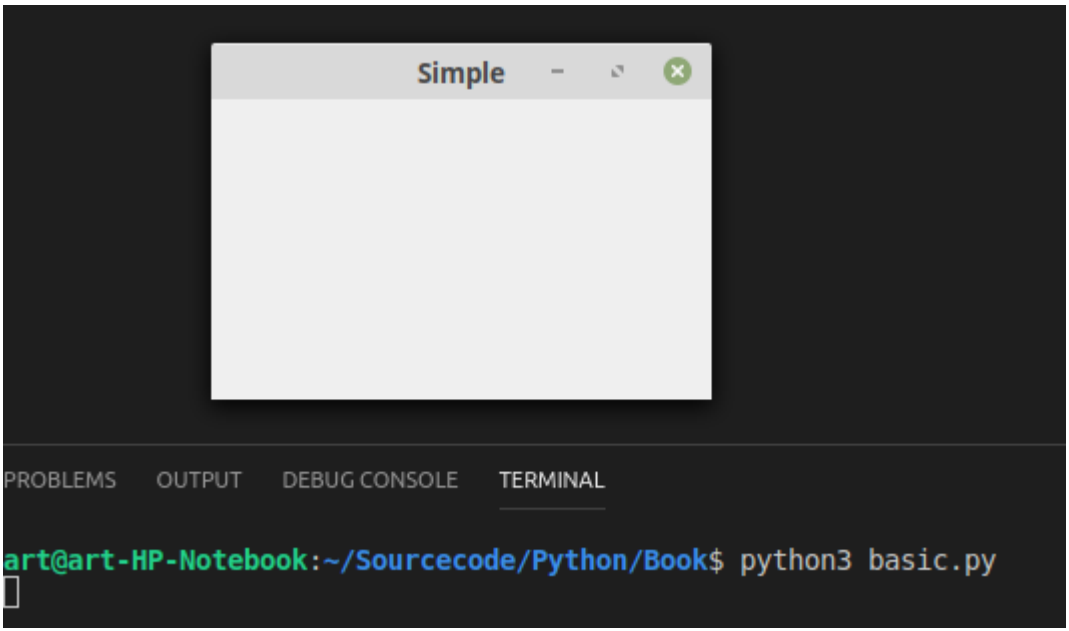
To test the environment we now create a simple app.

basic.py

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget


app = QApplication(sys.argv)
w = QWidget()
w.resize(250, 150)
w.setWindowTitle('Simple')
w.show()
app.exec()
```

In this case, because the python file is not named *main.py* we will run it using the terminal inside of VSCode.

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



Here we create an app, instantiate a widget, set the widgets size, set a window title, make the widget visible and execute the main loop.
In this main loop the app is polling for events like mouse and keyboard send them to the window and the window is reacting on them. In this case it will only react on resize, move and the close event when the user clicks the green close button on the upper right corner of the window where the main loop is exited.

Summary

After setting up the development environment we now have created our first Qt app at least for the desktop.

Part II - Basics

Many Different Approaches

Because of the fact that Python has been introduced in 1991, meanwhile there are a few user interface approaches to develop Python applications. There is Tkinter which is a bridge to TK. Tkinter is been delivered with Python. There is Kivy which has got a nice approach using a special pythonic language to declare the user interface. There is BeeWare which compiles Python to Java-ByteCode which then can be deployed on an Android device. There is Enaml Native which act like Pythons answer to React Native and there are a few approaches to bridge to Qt. Qt is pronounced 'Cute'. These approaches are PySide, which bridges to Qt4. There is PyQt which also bridges to Qt4, PySide2 which bridges to Qt5 and last but not least PyQt5 which we are talking about in this book. I am not going to talk about the pros and cons of all these possibilities in this book. Instead I am focusing on solutions. To use PyQt5 is a personal decision after working a few years with Qt5. Qt5 and PyQt5 are available under an open source license so you might use these frameworks for free as long as you plan to create open source programs. If you are going to create commercial software you have to purchase licenses for both frameworks. Even when we are using Qt5 we have got two options to develop applications. First option is QtWidgets which has been introduced to create platform independent desktops applications and QtQuick which uses a declarative way to implement user interfaces using QML (Qt Markup Language). Because at the moment QtQuick lacks of a tree view and a table view implementation I guess it has been primarily developed to satisfy mobile app development. QtQuick has also implemented behaviours and transitions which you normally only see on mobile platforms. If you have got design background then the QML approach could be your best choice because you don't really have to write code that much and when you like to write imperative code then QtWidgets might be your way.

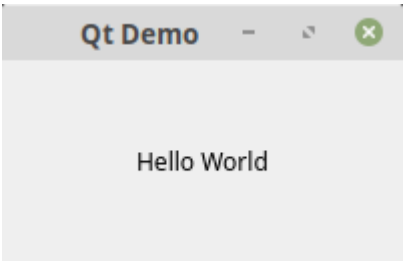
Hello World (QtWidgets)

Now I will show you how a very basic QtWidgets app will look like.

QWidget/Basics/main.py

```
1
```

```python
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())
```



The sample is simply self explanatory. We are creating an application object. Instantiate a window. Make the window visible and execute the app main loop.
In the window there is just a label widget which will be set as the central widget.
In bigger projects it would be better to create a python file for each class so that the class MainWindow will become an own file called *mainwindow.py*.

Hello World (QtQuick)

The hello world app for the QtQuick part will be made out of two file. First the main.py where the qml file will be loaded and the main loop will be started and a second file named view.qml where the user interface is been declared.

QtQuick/Basics/main.py

```python
import sys
from PyQt5.QtGui import QGuiApplication
```

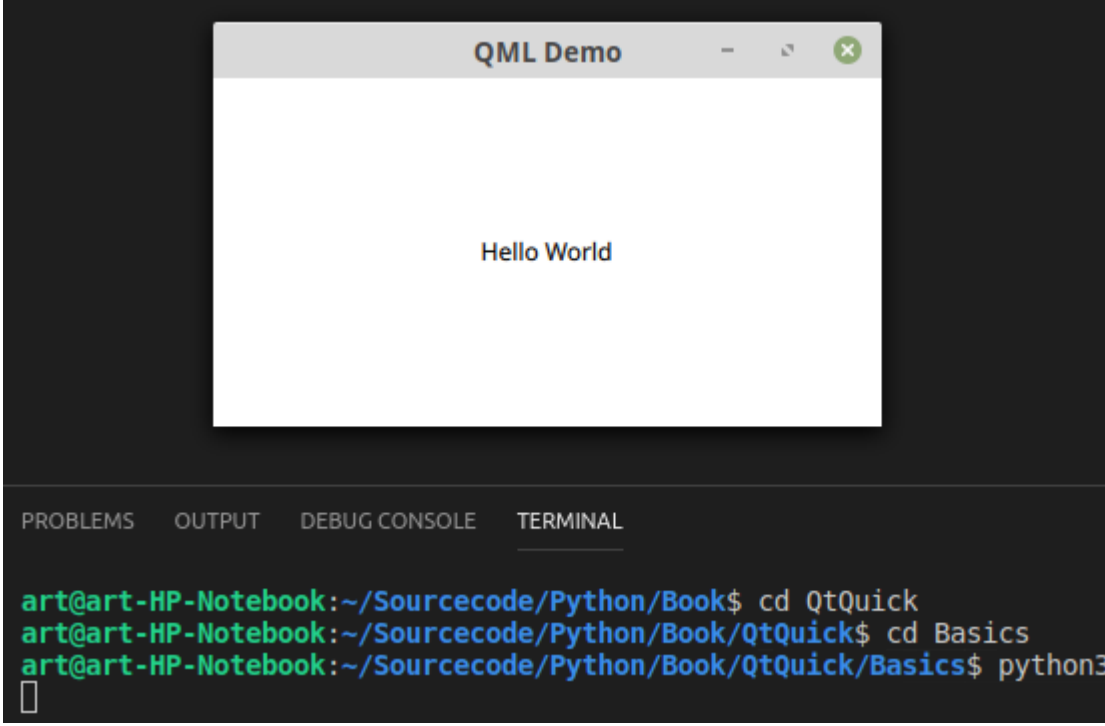
```
from PyQt5.QtQml import QQmlApplicationEngine


if __name__ == "__main__":
    app = QGuiApplication(sys.argv)
    engine = QQmlApplicationEngine("view.qml")
    if not engine.rootObjects():
        sys.exit(-1)
    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"
    }
}
```

In order to start the app we have to cd into the Basics directory and start it like this:

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



This app is similar to the QtWidgets version except the fact that we use a QGuiApplication instead of a QApplication. And also a QQmlApplicationEngine is responsible to load the user interface.

Combining QWidget and QML

A third possibility is to combine both QWidget and QML using the QQuickView inside a QWidget application.

Combo/main.py

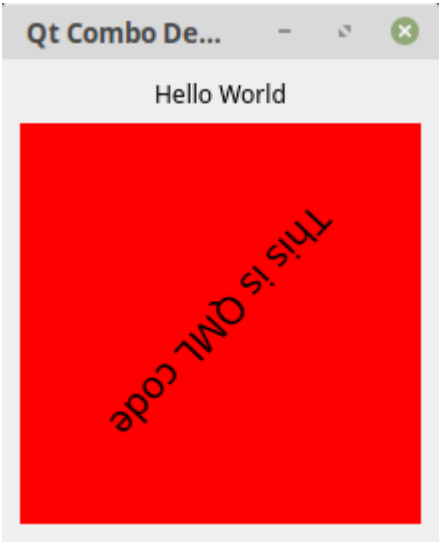
```
  import sys
  from PyQt5.QtWidgets import QApplication, QMainWindow, QLabel, QVBoxLayo
ut, QWidget
  from PyQt5.QtCore import Qt, QUrl
  from PyQt5.QtQuick import QQuickView


  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
```

```
28              onClicked: animation.paused ? animation.resume() : animation.pau
se()
29          }
30          Component.onCompleted: animation.start()
31      }
```



Be aware that you should not use many QQuickViews inside a QWidget window because of performance issues.

Summary

We have seen three approaches to build GUI applications using Qt5. The QWidgets approach is mostly used to create desktop applications. The QML approach is mostly used to create mobile applications and the combination can be used to create desktop applications where we use QML to render a part of thes desktop experience.

Part III - QML

QML Syntax

The QML syntax has been invented to declare user interfaces with an easy to read and easy to write new syntax, based on the ideas of XAML, Json and Javascript.
It combines the best of all of these languages.

In QML the userinterface is declared as a hirachy tree. Each item is positioned inside another item. The events triggert by a control, when the user clicks a button for example, can be coded in a Javascript like language.

To declare the item tree we use the item name followed by an open bracket and a close bracket.

```
Rectangle
{
}
```

To set the value of a property you write the name of the property followed by a colon ":" and then the value.

```
name: value
```

To declare numeric value you just put the numeric digit behind the colon.

```
width: 200
```

If you have got a text, then the value is put between quotations marks. **"** or **'**.

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

If you are using colors you put them also between quotation marks, leaded by a hash.

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

Here the red part of the color is set to hex FF (255). Green is set to 0 and blue is set to FF (255).

You can also set the opacity using an additional hex value at the end of the string.

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

Here the opacity has been set to EE (238).

The Id

A special value in QML is called id.

```
ApplicationWindow
{
    id: root
    Text
    {
        text: "The parent is " + root.width + " pixels wide"
    }
}
```

The **id** is a readonly value which can only be declared once. It is used to reference items. A good practice is to name the root item with the id: root, so that it's similar to all QML objects.

Child elements inherits the coordinate sytem from the parent. So the x and y coordinates are always relative to the parent.

Every QML needs to have exactly one root element. So there can only be one ApplicationWindow for example.

Child Objects

Any object declaration can define child objects through nested object declarations. In this way, any object declaration implicitly declares an object tree that may contain any number of child objects.

```
Rectangle
{
    color: "red"

```

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

Comments

Comments can be made using // for a single line or /* */ for multiple lines, just like in C, C++ and Javascript.

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

    // This is also a comment
}
```

Elements can be nested. So a parent element can have multiple child elements.

Import Statements

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

With the import statement you import a specific version of a QML module.

```
import QtQuick 2.5
```

The major version points to the QtQuick version you intent to use. And the minor version stands for the Qt version. So the above sample is importing QtQuick 2 and a Qt version 5.*.
So the version 2.12 to import points to QtQuick 2 and Qt 5.12.

Properties

You can declare properties using the property qualifier followed by the type, the name and an optional initial value. The syntax is `property <type> <name> : <value>`

```
property int clickCount: 0
```

Another possibilty is to use the alias keyword to forward a property of an object or the object itself to the outer scope. The syntax is `property alias <name>: <reference>`

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

A property alias does not need a type. It used the type of the referenced property.

Basic Types

Type	Description
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

One of the most importand features of QML is the property binding. A value of a property can be set via a constant, an expression or via binding to another property.

```
1  Rectangle
2  {
3      width: 200; height: 200
```

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

In the above case the height has been bound to the height of the parent object. If we change the height of the parent, then the height of the inner rect will be adjusted automatically.

Signals

Receiving signals and signal handlers

To receive a notification when a signal is emitted for an object, the object definition should declare a signal handler named `on<Signal>` , where Signal is the name of the signal, with the first letter capitalized. The signal handler should contain the JavaScript code to be executed when the signal handler is invoked.

```
        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);
            }
        }
```

In the above case the Button has got a signal with the name **clicked** and we create a signal handler with the corresponding name **onClicked**.

Property change signal handlers

A signal is emitted when the value of a QML property changes. This type of signal is a property change signal and signal handlers for these signals are

written in the form `on<PropertyName>Changed`, where PropertyName is the name of the property, with the first letter capitalized.

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

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

For example, the TapHandler of a Rectangle has got a **pressed** property. To receive a notification whenever this property changes, write a signal handler named `onPressedChanged` like in the above sample.

Connections

In some cases we want to access a signal outside of an object. Therefore we use the Connections type.

```
        Button
        {
            id: button
        }

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

Attached signal handlers

An attached signal handler receives a signal from an attached type.

```

```

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

In the above case the onCompleted event is fired, when the ApplicationWindow has been loaded completely.

Signals for a custom QML type

The syntax for defining a new signal is:
`signal <name>[([<type> <parameter name>[, ...]])`

A signal is emitted by invoking the signal as a method.

```
// 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
    }
}
```

The Rectangle object has an activated signal, which is emitted whenever the TapHandler is tapped.

To use this signal you declare an onActivated handler in the code that uses the SqareButton.

```
    SqareButton
    {
        color: "#345462"
```

```
            x: 200
            y: 0
            onActivated: console.log("Activated at " + xPosition + "," + yPosition)
        }
```

Functions

To declare a JavaScript function you are using the folowing syntax

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

In the next sample we have got the Text with the id txt. In the same parent we have go a MouseArea which is used just a get a click event when the user click in it with the mouse or when the user touches this area on a mobile.

```
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++
    }
}
```

The MouseArea has got a Clicked event, which we are coding with the "on" prefix.
If you have more than one line of code you have to use brackets.

In this event we are calling the function increment.
The property clickCount we declare on top level. It's a god habit to declare all variables in the top of the root element.

Interaction between QML and Python

You may code almost everything in QML (Javascript), but sometimes you need to do business logic in Python. Therefore I will show you a possible bridge.
We are creating a class derived from QObject and give an instance of this class to QML via context property.

```
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 you can use the object bridge to communicate with Python.

```
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` is using the property declared in Python to get the current working directory.
And with `bridge.message` we are sending data to a Python slot.
On the Python side we are able to call a Javascript function using the root object.

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

Screen size

To place items on the surface we have to position them.
When you are comming from a windows environment like WinForms, then you know how to position items using their x and y coordinates.
This was fine for windows, where the screensize was kind of known back in the years.
We only had screen sizes like 640 x 480 or 1024 x 768. Where you tried to support the smallest screen resolution when designing a dialog.
On a mobil phone we have hundreds of different screen resolutions and form factors, so we have to change our thinking how to design a dialog.
Also we do not create dialogs anymore.
Normally we create full screen pages.

When you are developing on a desktop computer you have to simulate the form factor of a mobile phone.
Because of the fact that my mobile phone has a resolution of 700 * 1200 and my desktop has 1366 * 768, I have to find a screen size to simulate the portrait mode of my phone on my desktop. So I am setting the ApplicationWindow to the size 350 * 600, so that the window fits on my screen and mimics the portrait mode of a mobile phone.

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

You should find another but similar solution for your environment.

So when we would use the direct positioning like we did on windows, where we used the x and y coordinates to position an item, we should use a different solution for the phone, because we don't know the screen resolution of the target platform.

Positioning Items

To position an item in QML we are using anchors.
You can still use the x and y coordinates, but it only makes sense, if the item is anchored to the upper left. You can anchor an item to the top, bottom, left

and to the right. Additionally you can also anchor an item to the horizontalCenter and to the verticalCenter.

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

In the above sample the center of the rectangle is bound to the center of the parent item, which is the application window or it could be another item.

If you want to anchor an item to the lower right then you positioning it as follows.

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

The margin is the distance between the item am the parents site.

In the source code to this book you will find a sample with all anchors under *QtQuick/Anchors*.

Controls

The most used control on a mobile phone might be the listbox.
Here is a sample which shows how to use it with a static model.

```
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: []
                }
            }
        }
    }
```

To render the list we design a component which is used as a delegate.
The model which delivers the data we use declarative here in this sample, to make it easier to understand.

So next we will use a model build in Python.

```
    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 the view we just have to change the model.

```
model: mymodel
```

The model object has now been set as a context property.

For additional control please refer the Qt documentations and samples.
All QML Types

Summary

We have seen the basics of QML. We are now able to create simple apps using QML and Python.

Part IV - Deploment

Deploying a dynamic app to Android

Now we are going to build an APK (**A**ndroid **P**ac**k**age). Therefore we first have to install a few other components.

Install pyqtdeploy 2.4

pyqtdeploy is a tool made by the same company who also created PyQt5. You can install pyqtdeploy via pip.

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

Install Java JDK 8

You should be able to find all you need to install Java JDK 8 here: https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html

Just to mention I had problem to build the package with Java JDK 10 installed.

Install Android SDK

To install the Android SDK go to https://developer.android.com/studio scroll down to *Command line tools only* and download the file for your platform. Then unzip the file somewhere on your computer. After unzipping go to the Sdk/tools/bin directory and run the sdkmanager to install all necessary tools:

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

Install Android NDK

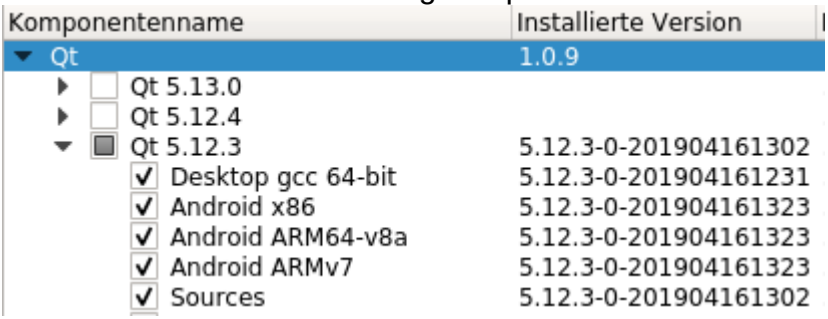
To install the Android NDK go to https://developer.android.com/ndk/downloads scroll down to *Older Versions*, click the link *NDK Archives*, agree to the terms and download *Android NDK, Revision 19c* for your platform (Unfortuneatly version 20 is not working for us yet).

After downloading unzip all the file into the SDK directory as *ndk-bundle*.
Your Android directory should look like the following:

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

Install Qt

Now we also need Qt. You can download it here: https://www.qt.io/download.
You should install the following components.



You need Desktop to test your apps on the desktop.
You need Android x86 to test yours apps on a simulator.
You need Android ARM64 to test on a 64 bit device and
you need Android ARMv7 to test on a 32 bit device.

Download Source packages

The following packages should only be downloaded and not extracted.

- 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/

Create the build script

You should change the paths in the following script to your needs.

Deploy/build.sh

```
  export ANDROID_NDK_ROOT=/media/art/data/Android/Sdk/ndk-bundle
  export ANDROID_NDK_PLATFORM=android-28
  export ANDROID_SDK_ROOT=/media/art/data/Android/Sdk
  pyrcc5 main.qrc -o lib/main_rc.py
  python3.7 build.py --target android-32 --installed-qt-dir /media/art/data/Qt/5.12.2 --no-sysroot --verbose --source-dir ./../../DeployAndroid/external-sources
```

external-sources points to the directory for the just downloaded packages.
If you want to install the app to an Android device with 64 bit, then you should change the target to `android-64` .
--installed-qt-dir points to the directory where Qt has been installed.

Create the build.py script

The build.py I have got from here: https://pypi.org/project/pyqtdeploy/#files from the Demo project. I have just added these lines of code:

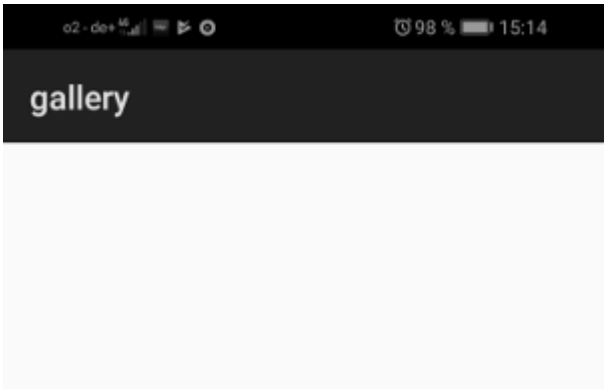
Deploy/build.py

```
    ...
    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_PA
CKAGE_SOURCE_DIR = " + os.path.join(dir_path, "android") + "\n}")

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

It is necessary that the androiddeployqt app will find a QML file in this directory. It scans all needed QML files for import statements to include the needed shared libraries into the APK.
androiddeployqt will be called by pyqtdeploy.

You can find the complete build.py and all other source files in the github repo: https://github.com/Artanidos/PythonAndroidBook

With the ANDROID*PACKAGE*SOURCE_DIR we specify that we have got Android specific files like the *AndroidManifest.xml* and the icons in this specific folder.
Also an app theme is included there. This theme only has one important thing *Theme.DeviceDefault.Light.NoActionBar*. This is needed to make the splash screen look not so ugly.
This is the default splash screen. With this little change the screen is just white.



Create the resource file

The resource file will contain our QML file as a python resource.
To build the resource run the following.

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

This is needed to find the QML later on the device. After these changes we have to change the main.py as follows:

Deploy/main.py

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

  if __name__ == "__main__":
      sys_argv = sys.argv
      sys_argv += ['--style', 'material']
      app = QGuiApplication(sys.argv)
      view =  "/storage/emulated/0/view.qml"
      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())
```

In this app we are looking for the file viem.qml, if it's stored on the mobile phone root. This file we will produce later. It can be edited on the phone, so you don't have to recompile the whole app over and over.

With `import lib.main_r` we add the resource file.
And with the double point in **:**/view.qml we will tell Qt to load this file from a resource.

Create a project file

The project file has the name demo.pdy and is needed for pyqtdeploy to be able to freeze all necessary packages.

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="android" />
  </Project>
```

One important thing here is the fact that we need a directory for all other python files to be included into the APK. We are using *lib* here. Inside this directory should be an empty file named *__init__.py*.
Then all needed Qt packages are defined here. (We don't need all of them but maybe later).
And also all needed standard python libraries are listed here. This file can be produced using this command in the terminal:

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

Create sysroot.json

This file hold all properties necessary to create a sysroot directory where some tools will be compiled.

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", "qtwebengine",
                    "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"
    }
}
```

Build the APK

If all went well we are now able to build the APK and deploy it to a device. Therefore we run `./build.s` in the terminal. Make sure that *build.sh* is executable. This first will create a sysroot directory with tools and libraries. If you are going to build the APK a second time you can skip building the sysroot with this option in the *build.sh*:

```
--no-sysroot
```

Install APK to device

When you have successfully build the APK you are now able to install it to your device.
First of all make sure the developer-mode is switched on on your device. If not you have to switch it on first.
Open your device settings app. Look for "About you phone" and search for the "Build Number". Then you have to tap on it seven times.
You should get a message that the developer options are now turned on.
Connect your device to your computer using a USB cable.
Now open the developer option in the settings app and look for USB-Configuration. Normally it is set to "Just load". Change it now to media transfer protocol (MTP).
Then look for USB-Debugging and switch it to on.
This done you are now able to find your device with the adb command:

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

If your device is listed copy the string to the left and paste it into the terminal:

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

Also copy the APK path which will be displayed after the build process has finished.

I have put the whole command into the *deploy.sh* file to use it later.

Now you can try to start the app. It should show up with the text DynPy in the middle.
Now you open the settings app on your phone and enable the storage for DynPy.
Then the app can open file here: */storage/emulated/0*, which emulates the root on the phone.
Now we are creating a new file with the name *view.qml* in this directory on your phone and restart the app.

```
import QtQuick 2.5
import QtQuick.Controls 2.0

ApplicationWindow
{
    visible: true

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

If everything went well, the app shall now display "DynPy".

This was you are able to develop the app further directly on the phone, without the need to recompile everything. You might use PyDroid or even Markdown an editor for MD files.

Summary

Hopefully we have deployed the DynPy app to our device. To find out how to build and deploy to Android took me several days.
If you are running into trouble here don't hesitate to contact me.

Maybe I can help you out and publish our experience into the next version of this book so that other people are not running into trouble.

Afterwords

I am happy that you have read so far. I hope this book could help you to learn some more possibilities as a software developer. I wish you good luck on your way. If you like this book then I would appreciate when you could write a small review to help other people to also find this book.

About The Author

Olaf Art Ananda, born 1963 in Hamburg, Germany has been a software developer for over 30 years now. He started with C, learned Assembler to speed up C programs and after trying out most popular programming languages like Java, C# and Objective-C he came back to C/C++ and started to develop desktop appĺications using Qt5 in 2016. Qt5 was the ideal platform for him to express his skills he has learned studying Human Computer Interaction Design in 2013. After a first try to use Python to develop plug-ins for his programs he needed another two years to really get to know Python. Today he enjoys the simplicity of this language to write apps in a very short time compared to C++. Olaf has worked for several top 500 companies like Dupont, Dresdner Bank, Commerzbank and Zürcher

Kantonalbank to name a few. After his burnout and a near death experience he decided to quit working for profit. Since 2016 he is writing open source software like the AnimationMaker, the FlatSiteBuilder and the EbookCreator. He also has written a book about his life and Tantra. Since 2016 he is living in his mobile home in the middle of Berlin and he is also playing percussion on the streets. To write code he is going to a public library. That's an easy living.