

Skript zur Vorlesung GeoProgrammierung I: HS2023

Einstieg in die Geo-Programmierung mit Python



Prof. Martin Christen
martin.christen@fhnw.ch
Hochschule für Architektur, Bau und Geomatik
Institut Geomatik

1 Objektorientierung

1.1 Was ist Objektorientierung ?

Objektorientierung ist ein Programmierparadigma, welches vor allem die Wiederverwendbarkeit und Wartung von Programmcode erleichtern soll. Gerade bei grösseren Programmierprojekten ist die Objektorientierung sehr hilfreich.

Bei der objektorientierten Programmierung werden **Klassen** erzeugt, welche die Eigenschaften von Objekten festlegt. Die Eigenschaften sind Werte (in Form von Variablen). Werte einer Klasse werden **Attribute** genannt. Eine Klasse kann auch Funktionen enthalten - diese werden **Methoden** genannt.

Beispiele:

- Was für Attribute und Methoden hat eine Türe ?
- Ein **Dreieck** besteht aus 3 Punkten. Ein **Punkt** besteht (in 2D) aus x- und y-Koordinate. Hier haben wir die beiden Objekte "Dreieck" und "Punkt" beschrieben.

1.2 Klassen definieren

Eine Klasse beschreibt die Struktur eines Objektes, welche aus **Attributen** und **Methoden** besteht.

Achtung: **Eine Klasse ist die Beschreibung eines Objekttyps**, aber noch kein Objekt. Das eigentliche Objekt ist eine sogenannte **Instanz** der Klasse. Aus einer Klasse können zahlreiche Instanzen angelegt werden.

In Python wird die Klasse mit dem Schlüsselwort "class" und dahinter der Klassename definiert. Danach folgt ein Doppelpunkt und auf den nächsten Zeilen (eingerückt) werden Methoden und Attribute definiert. Dies geschieht innerhalb des Konstruktors (init),

Beispiel: Definition der Klasse "Punkt":

```
class Punkt: # Klassendefinition (Beschreibung Klasse)
    def __init__(self):
        self.x = 0      # Attribut
        self.y = 0      # Attribut
```

Methoden (d.h. Funktionen innerhalb einer Klasse) werden mit dem def-Schlüsselwort angelegt. Alle Methoden haben **immer** mindestens einen Parameter, nämlich das Objekt selbst. In der Python-Community gibt es die Konvention diesen Parameter "**self**" zu nennen. Es wäre möglich diesen Parameter beliebig zu benennen, aber wir halten uns immer an diese Konvention!

Eine weitere Konvention ist, dass Methodennamen immer mit Kleinbuchstaben geschrieben werden. Wir halten uns wenn immer möglich auch an diese Konvention. Und wenn wir schon bei Konventionen sind: Klassennamen werden mit der "CapWords" Konvention definiert, das heisst Wörter beginnen immer mit Grossbuchstaben, auch wenn diese zusammengesetzt sind, z.B. Vector, VectorLayer, GeometryStyle.
(mehr zu Coding Styles: siehe PEP 8: <http://legacy.python.org/dev/peps/pep-0008/>, dies ist der defacto Style Guide für die Python Programmierung)

Beispiel: Methoden für die Klasse "Punkt":

```
class Punkt:
    def __init__(self):
        self.x = 0
        self.y = 0

    def ausgabe(self):
        print("Der Punkt hat die folgenden Komponenten:")
        print(" x = ", self.x)
        print(" y = ", self.y)

    def distance(self, other):
        return ((self.x-other.x)**2 + (self.y-other.y)**2)**0.5
```

1.3 Instanz anlegen: Die Instanziierung

Nachdem die Klasse beschrieben wurde, können wir nun **Instanzen** der Klasse anlegen. Um eine Klasse zu instanziieren wird die Klasse mit dessen Namen und dahinter runde Klammern aufgerufen. Der Rückgabewert ist eine Instanz der Klasse und wird normalerweise in einer Variablen gespeichert.

```
P = Punkt() # Instanz anlegen
P.x = 5      # Attribut x setzen
P.y = 10     # Attribut y setzen
P.ausgabe()  # Methode aufrufen

P2 = Punkt() # zweite Instanz anlegen
P2.x = 8     # Attribut x setzen
P2.y = 4     # Attribut y setzen
P2.ausgabe() # Methode aufrufen

d = P2.distance(P) # Methode "distance" aufrufen.
print(d)
```

1.4 Der Konstruktor

Der Lebenszyklus jeder Instanz ist immer gleich:

1. Die Instanz wird erzeugt
2. Die Instanz wird benutzt
3. Die Instanz wird gelöscht

Die Klasse ist dafür verantwortlich, dass sich die Instanz immer in einem sinnvollen Zustand befindet. Es gibt eine spezielle Methode, welche **automatisch** beim instanziieren eines Objektes aufgerufen wird, um das Objekt in einen sinnvollen, gültigen Zustand zu bringen. Diese Methode wird **Konstruktor** genannt.

Um einer Klasse einen Konstruktor zu geben wird eine Methode mit dem Namen `__init__` definiert. Die Methode heisst init und wird links und rechts mit je zwei

Unterstrichen ergänzt. Solche haben in Python eine besondere Bedeutung. Wir werden diese "Magic Methods" später noch im Detail ansehen.

1.5 Der Destruktor

Analog zum Konstruktor gibt es einen Destruktor, welcher aufgerufen wird, wenn die Klasse gelöscht wird. Der Destruktor wird in Python eher selten benötigt. Der Destruktor ist eine Methode mit dem Namen `__del__`.

```
class TestKlasse:  
    def __init__(self):  
        print("Hallo vom Konstruktor")  
    def __del__(self):  
        print("Hallo vom Destruktor")
```

Vorsicht: Python garantiert nicht, dass alle sich noch im Speicher befindlichen Instanzen gelöscht werden. Es kann unter Umständen vorkommen, dass diese erst zu einem späteren Zeitpunkt freigegeben werden. Der Destruktor sollte deshalb nicht für komplexere Aufräumarbeiten wie Dateien schliessen oder Netzwerkverbindungen beenden verwendet werden.

```
A = TestKlasse()  
del A
```

1.6 Objektorientierte Beziehungen

Eine Klasse kann von einer anderen Klasse abhängig sein.

Beispiel:

Das Objekt Punkt hat die Attribute x- und y-Koordinate.

Das Objekt Dreieck hat die **Attribute** A,B,C (Punkt).

Das Objekt Dreieck hat die **Methode** "Fläche", um die Fläche des Dreiecks zu berechnen.

1.7 Mehrere Konstruktoren

In Python gibt es nur einen Konstruktor, nämlich die Methode `__init__()`. In anderen Computersprachen kann es durchaus mehrere Konstruktoren mit Unterschiedlicher Anzahl Parametern oder unterschiedlicher Typen geben.

In Python kann das nur durch Standardparameter, Keyword-Parameter, Argumentliste oder mit einer Keyword-Argumentliste realisiert werden.

In der Regel sollten optionale Attribute immer mit Standard-Parameter-Werten im Konstruktor verwendet werden. Argumentlisten dürfen nur im absoluten Notfall verwendet werden, da es die Lesbarkeit des Python-Codes stark einschränkt.

1.8 Setter- und Getter-Methoden

Wir haben bereits eine sehr einfache Klasse "Punkt" kennengelernt welche die beiden Attribute `x` und `y` besitzt. Auf diese Attribute kann direkt zugegriffen werden. Dasselbe gilt für die Klasse Temperatur, bei der wir auf den Wert („value“) direkt zugreifen können.

```
class Temperature:  
    def __init__(self):  
        self.value = 0 # Temperatur in Celsius  
  
T = Temperature()  
T.value = 15      # Setzen der Temperatur (Celsius)
```

Viel besser ist es, wenn sogenannte Setter- und Getter-Methoden zu verwenden. Setter-Methoden sind dazu da einen Wert zu setzen ohne direkt auf das Attribut zuzugreifen. Das Attribut darf von aussen nur über diese Setter Methode verändert werden.

Dasselbe gilt für die Getter Methode. Der Wert eines Attributes kann von aussen nur über eine Methode ausgelesen werden.

Solche Attribute sind "private" Attribute und **nach Konvention** (Coding-Style) beginnen diese mit einem Unterstrich.

```
class Temperature:  
    def __init__(self):  
        self._value = 0  
  
    def setValue(self, c):  
        self._value = c  
  
    def getValue(self):  
        return self._value
```

Der Vorteil von Setter-/Getter-Methoden ist es, dass bei der Zuweisung überprüft werden kann, ob die Werte sinnvoll sind, oder ob gegebenenfalls etwas korrigiert werden muss. Es kann sogar eine Fehlermeldung (Exception) erstellt werden, wenn die Daten komplett falsch sind:

```
class Temperature:  
    def __init__(self):  
        self._value = 0  
  
    def setValue(self, c):  
        if c<-273.15:  
            raise ValueError("Darf nicht kleiner als 273.15 Grad sein.")  
        self._value = c  
  
    def getValue(self):  
        return self._value
```

Wenn direkt über das Attribut „_value“ zugegriffen würde, so könnte der Wertebereich nicht überprüft werden.

1.9 Property Attribute

Mit den Getter- und Setter-Methoden ist der Zugriff auf die Daten möglich. Mit der Setter-Methode können wir die Daten verändern und mit der Getter-Methode können wir Werte zuweisen.

Auf Ebene der Instanz ist das manchmal eher mühsam, da für jedes Attribut über die Methoden darauf zugegriffen wird.

Abhilfe schafft dabei die property definition, welche die Getter und Setter Methode automatisch aufruft.

```
class Temperature:
    def __init__(self):
        self._value = 0

    def setValue(self, c):
        if c<-273.15:
            raise ValueError("Darf nicht kleiner als 273.15 Grad sein.")
        self._value = c

    def getValue(self):
        return self._value

value = property(getValue, setValue)
```

```
sensor1 = Temperature()
sensor1.value = 10      # indirekter Aufruf des Setters
print(sensor1.value)   # indirekter Aufruf des Getters
```

1.10 Mehrere Properties

Wie könnte die Temperatur Klasse aussehen, wenn neben Celsius auch Fahrenheit und Kelvin unterstützt werden soll ?

Hinweis: Formeln zur Umrechnung Celsius/Kelvin/Fahrenheit:

$$C = K - 273.15$$

$$K = C + 273.15$$

$$F = C * 1.8 + 32$$

$$C = (F - 32) / 1.8$$

C: Wert in Celsius

K: Wert in Kelvin

F: Wert in Fahrenheit

Es gibt noch weitere klassische Temperaturskalen, siehe z.B: <http://de.wikipedia.org/wiki/Newton-Skala>

2 Magic Methods

Es gibt in Python zahlreiche spezielle Methoden, welche einer Klasse besondere Fähigkeiten geben. Die Namen dieser Methoden beginnen und enden immer mit zwei Unterstrichen. Wir haben bereits die Methode `__init__` kennengelernt, d.h. den Konstruktor.

Magisch dabei ist, dass die Funktionen nicht direkt aufgerufen wird sondern bei Bedarf implizit aufgerufen wird. Im Falle des Kontruktors wird `__init__` automatisch beim Erzeugen des Objektes aufgerufen.

Wir werden nun die wichtigsten Magischen Methoden und Attribute kennenlernen. Eine vollständige Liste gibt es in der Python Dokumentation unter:
<https://docs.python.org/3/reference/datamodel.html#special-method-names>

2.1 Typenumwandlung

2.1.1 `__str__`

Wenn die Klasse in eine Zeichenkette umgewandelt werden soll, wird die Methode `__str__` implementiert. Diese gibt eine Zeichenkette zurück. Ist es möglich eine Zeichenkette zu generieren so kann die instanz direkt mit der `print` Funktion ausgegeben werden.

Beispiel:

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __str__(self):
        return "POINT (" + str(self.x) + " " + str(self.y) + ")"

P = Point(10,20)
print(P)

S = str(P)
print(S)
```

2.1.2 `__bytes__`, `__bool__`, und `__complex__`

Gewisse Klassen können in eine Byte-Repräsentation, in ein bool und in eine komplexe Zahl umgewandelt werden. Diese Methoden werden jedoch eher selten benötigt.

2.1.3 __int__ und __float__

Klassen welche zum Beispiel eine Zahl repräsentieren können mit den speziellen Methodennamen `__int__` und `__float__` in eine ganze Zahl oder in eine Gleitkommazahl umgewandelt werden. Auch diese Methoden benötigen wir auch eher selten.

2.2 Operatoren überladen

2.2.1 Vergleichsoperatoren

Wir definieren eine einfachere Form der Klasse "Temperatur" welche nur die Temperatur in Celsius speichert. Diese sehr einfache Klasse sieht folgendermassen aus:

```
class Temperature:
    def __init__(self, c=0):
        self.celsius = c
```

Nun können wir mehrere Instanzen dieser Klasse anlegen, beispielsweise eine Zeitreihe verschiedener Temperaturen T0, T1, T2, gemessen am Tag X zu bestimmten Zeiten, an einem Tag wurde folgendes gemessen:

```
T0 = Temperature(15)
T1 = Temperature(18)
T2 = Temperature(17)
```

Die Fragen lauten nun:

- Ist T1 grösser als T0 ?
- Ist T2 grösser als T1 ?

Der entsprechende Python Code ist:

```
if T1.celsius > T0.celsius:
    print("T1 ist grösser als T0")
if T2.celsius > T1.celsius:
    print("T2 ist grösser als T1")
```

Dieser Code ist jedoch nicht intuitiv, da bei der Anwendung plötzlich das Attribut "celsius" ins Spiel kommt. Schöner wäre es doch, wenn folgendes geschrieben werden könnte:

```
if T1 > T0:
    print("T1 ist grösser als T0")
if T2 > T1:
    print("T2 ist grösser als T1")
```

Python bietet die Möglichkeit Operatoren zu defininieren. Dies geschieht über Magische Methoden. Im oberen Beispiel muss der "grösser als" Operator definiert werden, was mit der Magischen Funktion `__gt__` (`gt` = greater than) realisiert werden kann.

Die Magische Funktion hat als erster Parameter eine Referenz auf sich selbst ("self") und als zweiter Parameter eine Referenz auf die zu Vergleichende Instanz ("other"). Als Rückgabewert wird ein Boolean erwartet, welcher True ist, falls "self" > "other" ist.

```
class Temperature:  
    def __init__(self, c=0):  
        self.celsius = c  
    def __gt__(self, other):  
        return self.celsius > other.celsius
```

Für andere Vergleichsoperationen ist das Prinzip identisch. In der Klasse sollten immer alle Vergleichsoperationen implementiert werden. In der folgenden Tabelle werden sämtliche Vergleichsoperatoren aufgelistet, welche in einer Python Klasse als Magische Methode implementiert werden können.

Operator	Methode	Beschreibung
<	<code>__lt__(self, other)</code>	less than (kleiner als)
<code>==</code>	<code>__le__(self, other)</code>	less or equal (kleiner oder gleich)
<code>!=</code>	<code>__eq__(self, other)</code>	equal (gleich)
<code>></code>	<code>__gt__(self, other)</code>	greater than (grösser als)
<code>>=</code>	<code>__ge__(self, other)</code>	greater or equal (grösser oder gleich)

2.2.2 Binäre arithmetische Operatoren

Ein binärer arithmetischer Operator verarbeitet zwei Operanden. Ein Beispiel wäre die Addition:

```
a + b
```

In Python können diese auch mit Magischen Methoden implementiert werden. Dabei ist - ähnlich wie den Vergleichsoperatoren - der erste Operand "self" und der zweite Operand "other". Der Rückgabewert ist immer eine neue Instanz welche das Resultat der binären Operation enthält, also in unserem Beispiel die Summe $a+b$, oder allgemeiner:

Linker-Operand Operator Rechter-Operand

Als Beispiel erstellen wir eine Klasse `Vector2` welche einen zweidimensionalen Vektor repräsentiert. Die Klasse soll die Attribute `x` und `y` besitzen:

```
class Vector2:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Vector2(self.x + other.x, self.y + other.y)
```

Die Magische Methode `__add__` bildet die Summe der beiden Operanden. Mit dieser Klasse kann nun folgendes implementiert werden:

```
v0 = Vector2(0,3)
v1 = Vector2(2,3)

v2 = v0 + v1

print(v2.x, v2.y)
```

In der nachfolgenden Tabelle sind sämtliche binären arithmetischen Operatoren und die dazugehörige Magischen Methoden aufgelistet.

Operator	Magische Methode
+	<code>__add__(self, other)</code>
-	<code>__sub__(self, other)</code>
*	<code>__mul__(self, other)</code>
/	<code>__truediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>
divmod	<code>__divmod__(self, other)</code>
**	<code>__pow__(self, other)</code>
%	<code>__mod__(self, other)</code>
>>	<code>__lshift__(self, other)</code>
<<	<code>__rshift__(self, other)</code>
&	<code>__and__(self, other)</code>
	<code>__or__(self, other)</code>
^	<code>__xor__(self, other)</code>

Nun gibt es noch eine Besonderheit bei binären arithmetischen Operatoren. Wir müssen beachten, dass es auch Operationen mit unterschiedlichen Datentypen gibt, wie beispielsweise `2 * "Hello"`. Existiert beim ersten Datentypen keine magische Funktion für den Operator, wird beim zweiten Datentypen gesucht. In folgender Liste werden diese aufgeführt.

Operator	Magische Methode
+	<code>__radd__(self, other)</code>
-	<code>__rsub__(self, other)</code>
*	<code>__rmul__(self, other)</code>
/	<code>__rtruediv__(self, other)</code>
//	<code>__rfloordiv__(self, other)</code>
divmod	<code>__rdivmod__(self, other)</code>
**	<code>__rpow__(self, other)</code>
%	<code>__rmod__(self, other)</code>
>>	<code>__rlshift__(self, other)</code>
<<	<code>__rrshift__(self, other)</code>
&	<code>__rand__(self, other)</code>
	<code>__ror__(self, other)</code>
^	<code>__rxor__(self, other)</code>

2.2.3 Erweiterte Zuweisungen

Die erweiterten Zuweisungen (Beispiel: `+=`) können auch überladen werden.

Standardmäßig verwendet Python für solche Zuweisungen den Operator selbst, so dass `a+=5` intern wie `a = a + 5` ausgeführt wird. Diese Vorgehensweise hat für gewisse Datentypen, wie zum Beispiel Listen den Nachteil, dass jedesmal eine neue Liste erzeugt werden muss. Aus Gründen der Optimierung macht es mehr Sinn den `+=` Operator zu benutzen und diesen gezielt so anzupassen, dass die Effizienz gesteigert wird.

In der nachfolgenden Tabelle sind alle Operatoren und dessen Magische Methoden aufgelistet.

Operator	Magische Methode
<code>+=</code>	<code>__iadd__(self, other)</code>
<code>-=</code>	<code>__isub__(self, other)</code>
<code>*=</code>	<code>__imul__(self, other)</code>
<code>/=</code>	<code>__itruediv__(self, other)</code>
<code>//=</code>	<code>__ifloordiv__(self, other)</code>
<code>**=</code>	<code>__ipow__(self, other)</code>
<code>%=</code>	<code>__imod__(self, other)</code>
<code>>>=</code>	<code>__ilshift__(self, other)</code>
<code><<=</code>	<code>__ishift__(self, other)</code>
<code>&=</code>	<code>__iand__(self, other)</code>
<code> =</code>	<code>__ior__(self, other)</code>
<code>^=</code>	<code>__ixor__(self, other)</code>

Die Methoden müssen alle die Instanz `self` zurückgeben.

2.2.4 Unäre Operatoren

Zu den unären Operatoren gehören vor allem die Vorzeichen `+` und `-`. Auch die eingegebene Funktion `abs` (Absolutwert) und der Operator `~` (Komplement) gehören dazu. Auch diese können in einer Python Klasse überladen werden, die Methoden sind in der folgenden Tabelle aufgelistet:

Operator	Magische Methode
<code>+</code>	<code>__pos__(self)</code>
<code>-</code>	<code>__neg__(self)</code>
<code>abs</code>	<code>__abs__(self)</code>
<code>~</code>	<code>__invert__(self)</code>

Die Methoden geben in der Regel eine neue Instanz mit dem veränderten Wert zurück.

2.2.5 Container Methoden

Auch Operatoren für Container können als Magische Methoden implementiert werden:

Magische Methode	Beschreibung
<code>__len__(self)</code>	Liefert die Anzahl Elemente in dem Container zurück (als Integer)
<code>__getitem__(self, key)</code>	Liefert das Resultat für den Operator [].
<code>__setitem__(self, key, value)</code>	Verändert das Element über den Operatoren [].
<code>__delitem__(self, key)</code>	Entfernt das Element des Containers welches dem Schlüssel key zugeordnet ist
<code>__iter__(self)</code>	Gibt einen Iterator über die Werte des Containers zurück.
<code>__contains__(self, item)</code>	Prüft, ob ein Element in dem Container enthalten ist.

3 Vererbung und Klassendiagramme

3.1 Objektorientierte Modellierung und UML

In der objektorientierten Programmierung haben Klassen eine Vielzahl von Aufgaben. Das primäre Ziel jedoch ist die Gruppierung respektive Kapselung von Attributen und dazugehörigen Methoden, vor allem aus dem Grund der Wiederverwendbarkeit. Die Gruppierung entspricht einer konzeptuellen Einheit, welche durch den Klassennamen repräsentiert wird.

Andere wichtige Gründe für die Anwendung der Objektorientierung sind die Verkürzung der Entwicklungszeit, die Senkung der Fehlerrate und eine verbesserte Erweiterbarkeit und Anpassungsfähigkeit.

Ein wichtiger Aspekt der Objektorientierung ist die **objektorientierte Modellierung**. Bei der objektorientierten Modellierung geht es darum, Anforderungen zu erfassen und zu beschreiben, welches das endgültige Produkt haben soll. Dies geschieht zum Beispiel in der Form eines Pflichtenheftes.

Sind die Anforderungen bekannt, so kann das werdende Softwareprodukt in Form eines Diagrammes beschrieben werden. Wir unterscheiden zwei wichtige Typen von Modellen:

- Dynamische Modelle, z.B. Flussdiagramme, Aktivitätsdiagramme
- Statische Modelle, z.B. Klassendiagramme

Es gibt Regeln, wie solche Diagramme erstellt werden. Die **Unified Modelling Language** (kurz UML) ist die grafische Modellierungssprache, welche am häufigsten benutzt wird um Software-Projekte zu modellieren/realisieren. UML besteht aus Sprache, Beziehungen, grafischer Notation und Austauschformat. Wir werden vor allem die grafische Notation betrachten.

Es existieren verschiedene Typen von Diagrammen, auf die wir nicht alle detailliert eingehen können. Wir beschränken uns hier auf das "**UML Klassendiagramm**". Weitere wichtige Diagramme von UML sind das Objektdiagramm, Aktivitätsdiagramm, Sequenzdiagramm und das Anwendungsfalldiagramm.

3.2 Darstellung von Klassen in UML

Das UML Klassendiagramm dient dazu, Klassen **unabhängig von der Programmiersprache abzubilden**. Es können die Klassen selbst und deren Beziehungen zu anderen Klassen dargestellt werden.

Klassennamen	Namensfeld - enthält den Namen der Klasse
Attribut	Attributliste - enthält alle Attribute
Attribut	
Operation	Operationsliste - enthält alle Methoden
Operation	

Die Darstellung einer Klasse in UML ist einfach. Es wird ein Rechteck genommen und darin steht der Klassennamen. Besitzt die Klasse Attribute und Methoden, so werden diese durch weitere Rechtecke erweitert, wie in der Abbildung oben.

UML Diagramme werden oft mit Softwaretools wie Visio (Windows) oder OmniGraffle (Mac) erstellt. Es gibt auch komplexe Softwarepakete, welche auf UML spezialisiert sind, welche allerdings für kleinere bis mittlere Projekt oft ungeeignet sind (z.B. Borland Together, ArgoUML, ...) Ein sehr "einfaches" Vektorprogramm, oder noch besser - eine Skizze von Hand ist als Entwurf oft besser geeignet! In Visual Studio Code gibt es auch die Extension PlantUML.

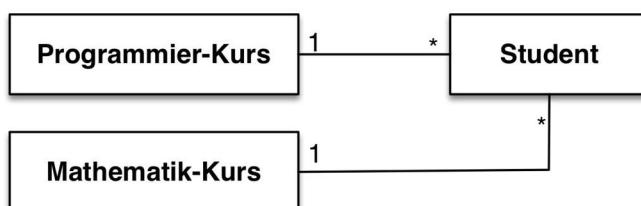
3.3 Objektorientierte Beziehungen

Nachdem wir in der Lage sind Klassen in der UML Notation darzustellen, ist es wichtig auch die Beziehungen zwischen Klassen anzusehen. Es gibt dabei verschiedene Beziehungstypen zwischen Klassen.

3.3.1 Assoziation ("hat-Beziehung")

Eine Assoziation beschreibt eine Beziehung zwischen zwei (oder mehr) Klassen. Ein Objekt besitzt ein oder mehrere Objekte. Bei der Assoziation haben alle Objekte ihren eigenen Lebenszyklus und werden separat erstellt. Die Klassen haben keinen Besitzer. Die assoziierten Objekte können von unterschiedlichen Klassen benutzt werden.

UML Notation: Strich und Angabe der Multiplizität. Ein (offener) Pfeil ist optional.



Beispiel: Assoziation Student und Kurs. "Kurs hat Student"

In Python wird die Assoziation oft so realisiert, dass die Objekte ausserhalb erstellt werden und über eine Methode zugewiesen werden. Hier eine mögliche Implementierung der oben gezeigten Klassendefinition:

```
class Student:
    def __init__(self, ...):
        self.vorname = ...
        self.name = ...

class ProgrammierKurs:
    def __init__(...):
        ...
    def AddStudent(student):
```

...

```
class MathematikKurs:  
    def __init__(self):  
        ...  
    def AddStudent(student):  
        ...
```

Der Lebenszyklus der Instanzen wird also "von aussen" definiert:

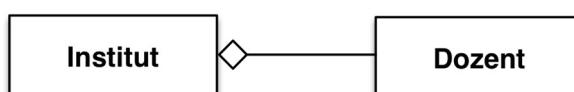
```
student = Student("Martin", "Müller")  
kurs1 = ProgrammierKurs()  
kurs2 = MathematikKurs()  
  
kurs1.AddStudent(student)  
kurs2.AddStudent(student)
```

3.3.2 Aggregation ("besteht-aus-Beziehung")

Die Aggregation ist eine spezielle Form der Assoziation. Ein Objekt besteht aus einem (oder mehreren) anderen Objekten. Die Objekte haben auch ihren eigenen Lebenszyklus, aber es gibt genau einen Besitzer eines Objektes.

Die einzelnen Komponenten des Aggregators sind auch ohne andere Klasse existierend zu benutzen.

UML-Notation: Raute und Strich und Angabe der Multiplizität.

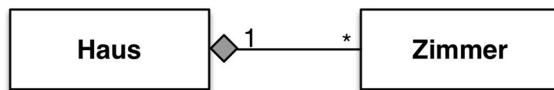


Beispiel: Institut und Dozent: Der Dozent gehört in diesem Beispiel genau einem Institut an. Es wäre aber möglich das Institut zu wechseln.

3.3.3 Komposition ("starke besteht-aus-Beziehung")

Ein Objekt besteht aus einem (oder mehrerer) anderen Objekten. Die einzelnen Komponenten des Aggregators sind ohne die andere Klasse nicht existent und nicht zu benutzen. Die beiden Klassen haben einen gemeinsamen Lebenszyklus und können nicht alleine existieren.

UML-Notation: ausgefüllte Raute und Strich und Angabe der Multiplizität.



Beispiel: Haus und Zimmer: Das Zimmer kann das Haus nie wechseln!

Mögliche Implementierung:

```

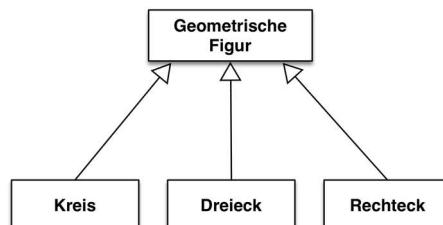
class Zimmer:
    def __init__(self):
        ...
        ...

class Haus:
    def __init__(self):
        self.Zimmer1 = Zimmer()
        self.Zimmer2 = Zimmer()
  
```

3.3.4 Vererbung ("ist-Beziehung")

Die Vererbung ist eine Beziehung bei der alle Attribute und Methoden der Elternklasse übernommen wird und ergänzt werden kann.

UML Notation: Pfeil mit unausgefüllter Pfeilspitze ("Dreieck") auf übergeordnete Klasse.



Beispiel: Geometrische Figur vererbt Kreis, Dreieck, Rechteck

3.3.4.1 Beispiel Code für Vererbung: Figur

```

class Figur:
    def __init__(self, name):
        self.name = name

    def getName(self):
        return self.name

    def flaeche(self):
        return 0
  
```

figur.py

```
from figur import *
```

```
class Kreis(Figur):
    def __init__(self, mittelpunkt, radius):
        super().__init__(self, "Kreis")
        self.mittelpunkt = mittelpunkt
        self.radius = radius

    def flaeche(self):
        return self.radius**2 * math.pi
```

kreis.py

3.3.4.2 Beispiel für Vererbung: Fahrzeug

Wir schreiben eine Simulation. Fahrzeuge sind:

Fahrrad, PKW, LKW, Lieferwagen, Polizeiauto.

Fahrzeuge haben:

- Anzahl Räder
- Klassenspezifische Attribute / Methoden
- Eine maximale Geschwindigkeit (`getMaxVelocity()`)
Eine Methode `move()`, welche aufgrund der Maximalgeschwindigkeit sich innerhalb einer Minute x Meter weit bewegt (Zufallsgenerator)
[km/h / 3.6 -> m/s]

Verschiedene Fahrzeuge sollen in einer Klasse “Flotte” verwaltet werden und in einem Wettrennen bewegt. Sieger ist, wer zuerst 100km zurückgelegt hat.

4 Einführung in die GUI-Programmierung mit PyQt5

4.1 Einleitung

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

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

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

PyQt5 ist ein ausgereiftes GUI-Framework.

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

4.2 Ein erstes PyQt5 Programm

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

```
import sys
from PyQt5.QtWidgets import *

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

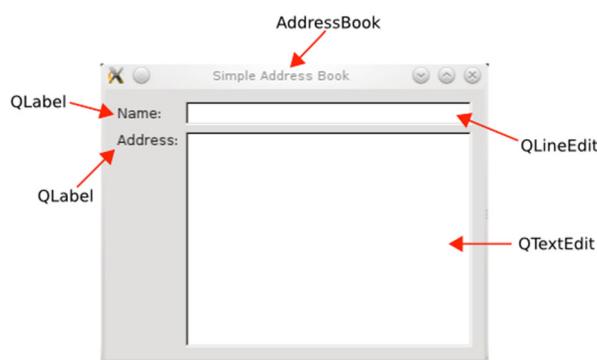
def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                  # Applikations-Loop starten

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

Programm: 00_window.py

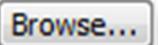
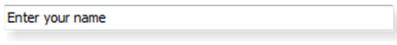
4.3 Widgets

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



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

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

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

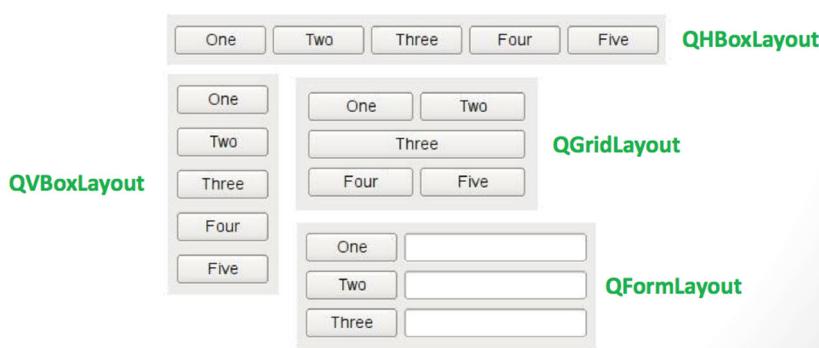
	QListView	Mit der List View kann eine Liste von Elementen angezeigt werden.
	QTreeView	mit dem Tree View kann ein Baum von Elementen angezeigt werden.
	QTableView	Mit der Table View kann eine Tabelle angezeigt werden.

4.4 Layout

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

Die wichtigsten Layout-Typen sind:

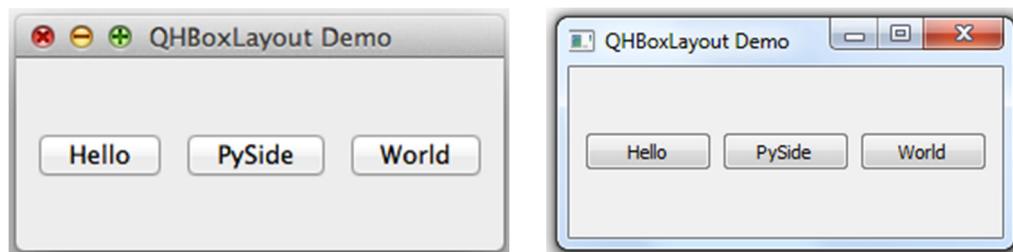
QVBoxLayout, QHBoxLayout, QGridLayout und QFormLayout.



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

4.5 Widgets und Layouts erstellen

4.5.1 BoxLayout: Erstellen von QHBoxLayout und QVBoxLayout



Beispiel eines QHBoxLayout

Nun erstellen wir ein `QHBoxLayout`, welches Widgets horizontal anordnet. Dazu müssen wir eine Instanz von `QHBoxLayout` erstellen. Diese wird in der Variablen "layout" gespeichert. Danach erstellen wir 3 `QPushButton`s und speichern diese in den Variablen "button1", "button2" und "button3". Dann können wir die Instanzen der Buttons dem layout hinzufügen. Dies machen wir für alle Buttons mit den Anweisungen `layout.addWidget(button1)`
`layout.addWidget(button2)` und `layout.addWidget(button3)`.

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

```
import sys
from PyQt5.QtWidgets import *

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

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

        # Layout erstellen:
        layout = QHBoxLayout()

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

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

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

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

        # Fenster anzeigen
        self.show()

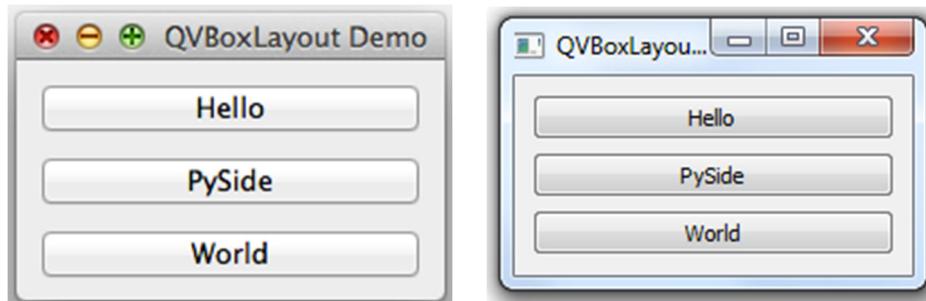
def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                  # Applikations-Loop starten

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

Programm: 01_hboxlayout.py

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

```
layout = QVBoxLayout()
```

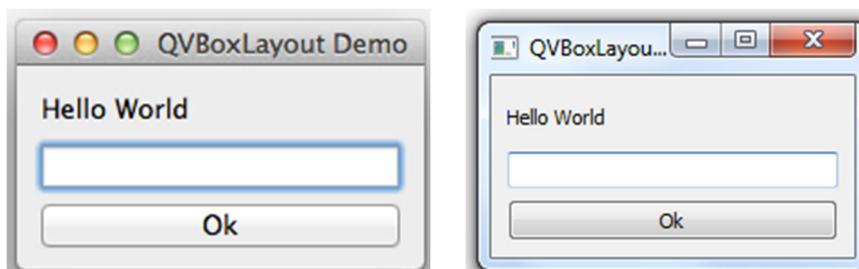


Beispiel eines QVBoxLayout

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

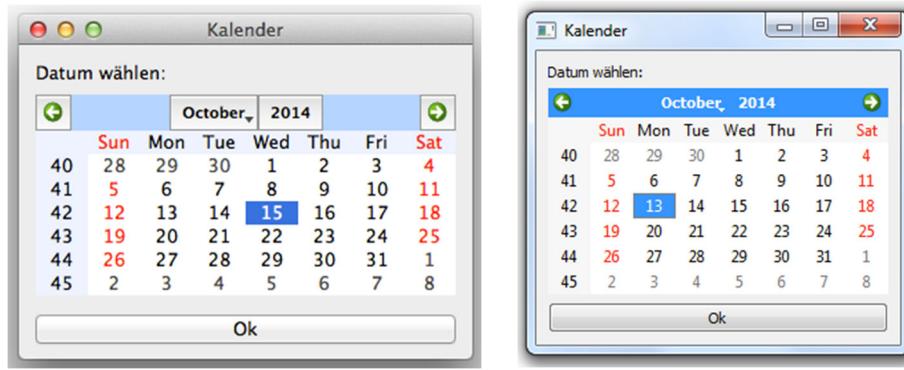
```
# Widget-Instanzen erstellen:  
label = QLabel("Hello World")  
edit = QLineEdit()  
button = QPushButton("Ok")  
  
# Widgets dem Layout hinzufügen:  
layout.addWidget(label)  
layout.addWidget(edit)  
layout.addWidget(button)
```

Programm: 02_widgets_vbox.py



Beispiel eines QVBoxLayout mit verschiedenen Widgets

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



Das QCalendarWidget()

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

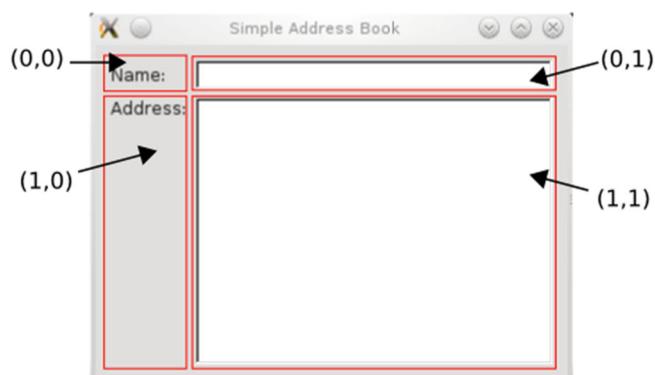
# Widget-Instanzen erstellen:
label = QLabel("Datum wählen:")
calendar = QCalendarWidget()
button = QPushButton("Ok")

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

Programm: 03_calendar.py

4.5.2 Erstellen eines QGridLayout

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



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

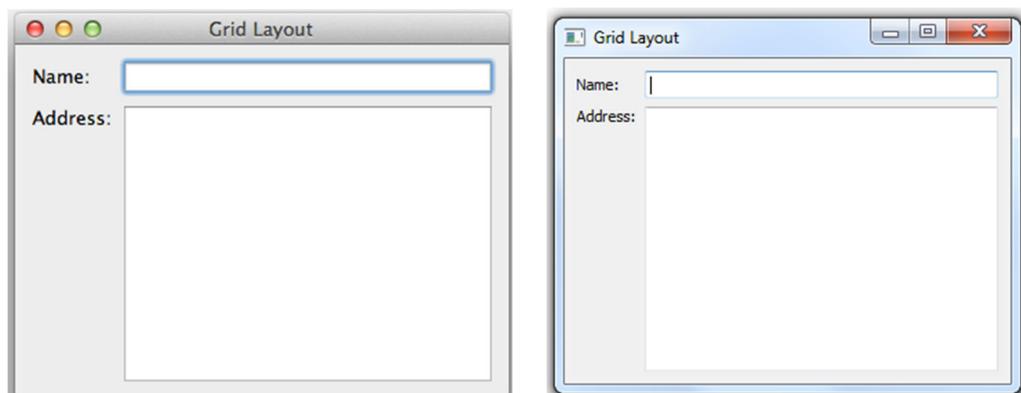
```
# Layout erstellen:  
layout = QGridLayout()  
  
# Widget-Instanzen erstellen:  
nameLabel = QLabel("Name:")  
nameLine = QLineEdit()  
addressLabel = QLabel("Address:")  
addressText = QTextEdit()  
  
# Widgets mit Grid-Koordinaten dem Layout hinzufügen  
layout.addWidget(nameLabel, 0, 0)  
layout.addWidget(nameLine, 0, 1)  
layout.addWidget(addressLabel, 1, 0, Qt.AlignTop)  
layout.addWidget(addressText, 1, 1)
```

Programm: 04_gridlayout.py

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

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

Im QtCore Untermodul befinden sich sehr viele solche Definitionen.



Layout-Beispiel mit dem QGridLayout

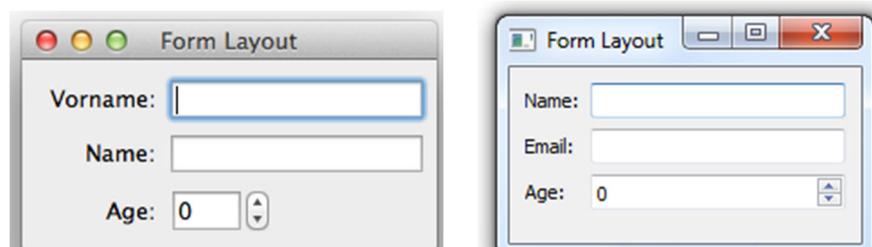
4.5.4 Erstellen eines QFormLayout

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

```
# Layout erstellen:  
layout = QFormLayout()  
  
# Widget-Instanzen erstellen:  
  
nameLineEdit = QLineEdit()  
emailLineEdit = QLineEdit()  
ageSpinBox = QSpinBox()  
  
# Layout füllen:  
layout.addRow("Name:", nameLineEdit)  
layout.addRow("Email:", emailLineEdit)  
layout.addRow("Age:", ageSpinBox)  
setLayout(formLayout)
```

Programm: 05_formlayout.py

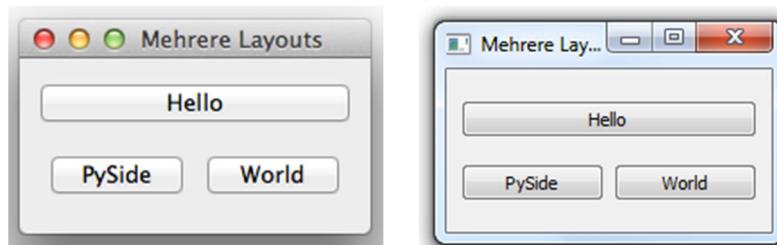
Wie wir im Programm deutlich erkennen können, benötigen wir nur die Hälfte an Widgets gegenüber dem QGridLayout, da wir uns die QLabel sparen konnten. Anstelle von Text-Labels ist es auch möglich andere Widget-Typen zu verwenden.



Beispiel: QFormLayout

4.5.5 Vermischen von Layouts

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



Mehrere Layouts: QVBoxLayout und QHBoxLayout vermischt

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

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

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

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

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

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

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

Programm: 06_multilayout.py

4.6 Das Anwendungsmenu

4.6.1 Ein einfaches Menu erstellen

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

Sehen wir uns folgendes Beispiel an:

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

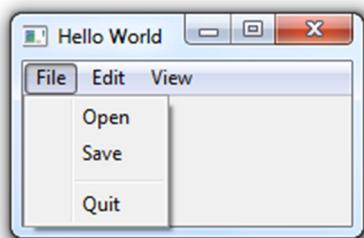
filemenu = menubar.addMenu("File") # "File"-Menu erstellen
editmenu = menubar.addMenu("Edit") # "Edit"-Menu erstellen
viewmenu = menubar.addMenu("View") # "View"-Menu erstellen

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

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

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

Programm: 07_menu.py



Das Menu-Beispiel unter Windows

4.6.2 Menus verschachteln: Untermenüs erstellen

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

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

Danach können wir diesem Untermenü wie zuvor Actions hinzufügen.

4.7 Einführung Signale und Slots

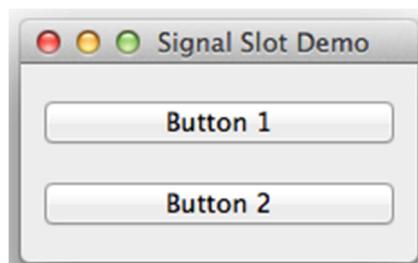
Bisher haben wir GUI Oberflächen erstellt, aber noch nicht auf Eingaben der Benutzenden reagiert. Für die Kommunikation zwischen Objekten benutzt Qt das sogenannte "Signal-Slot-Konzept".

Das Prinzip ist recht einfach:

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

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

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



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

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

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

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

```
# Layout erstellen:  
layout = QVBoxLayout()  
  
# Widget-Instanzen erstellen:  
button1 = QPushButton("Button 1")  
button2 = QPushButton("Button 2")  
  
# Widgets dem Layout hinzufügen:  
layout.addWidget(button1)  
layout.addWidget(button2)  
  
# Zentrales Widget erstellen und layout hinzufügen  
center = QWidget()  
center.setLayout(layout)  
  
# Zentrales Widget in diesem Fenster setzen  
self.setCentralWidget(center)  
  
# Fenster anzeigen  
self.show()  
  
#####  
# CONNECTS #  
#####  
  
# Klick auf Button 1 ruft die Funktion button1_clicked() auf  
button1.clicked.connect(self.button1_clicked)  
  
# Klick auf Button 2 ruft die Funktion button2_clicked() auf  
button2.clicked.connect(self.button2_clicked)  
  
  
def button1_clicked(self):  
    print("Der Button 1 wurde gedrückt")  
  
def button2_clicked(self):  
    print("Der Button 2 wurde gedrückt")  
  
  
def main():  
    app = QApplication(sys.argv) # Qt Applikation erstellen  
    mainwindow = MyWindow()      # Instanz Fenster erstellen  
    app.exec()                  # Applikations-Loop starten  
  
if __name__ == '__main__':  
    main()
```

Programm: 08_signal_slot.py

4.8 Signale und Slots für Widgets

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

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

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

4.9 Signale für Menus

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

```

import sys
from PyQt5.QtWidgets import *

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

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

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

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

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

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

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

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

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

```

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

Programm: 09_signal_slot_menu.py

4.10 Beispiele von Signalen für verschiedene Widgets

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

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

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

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

        # Layout erstellen:
        layout = QVBoxLayout()

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

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

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

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

    # Fenster anzeigen
```

```
self.show()

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

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

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

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

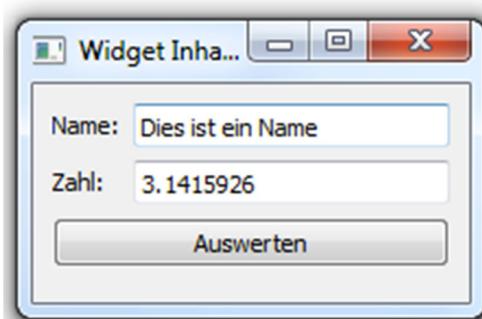
def main():
    app = QApplication(sys.argv) # Qt Applikation erstellen
    mainwindow = MyWindow()      # Instanz Fenster erstellen
    app.exec()                  # Applikations-Loop starten

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

Programm: 10_signal_slot_widgets.py

4.11 Zugriff auf Widget-Inhalte

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



Beispiel: Daten über QLineEdit erfassen und auswerten

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

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

        self.createLayout()
        self.createConnects()

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

        # Layout erstellen:
        layout = QFormLayout()

        # Widget-Instanzen erstellen:

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

        # Layout füllen:
```

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

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

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

# Fenster anzeigen
self.show()

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

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

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

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

Programm: 11_read_widgets.py

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

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

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

5 Dialoge

Dialoge sind eine weitere Form von Fenstern, welche besonders dazu geeignet sind, Daten zu präsentieren oder Eingaben abzufragen. In PyQt gibt es etliche Standarddialoge wie zum Beispiel für das Öffnen von Dateien. Neben den Standarddialogen können auch eigene Dialoge erstellt werden.

Auf Desktop-Systemen wird meist zwischen **modalen** und **nicht-modalen Dialogen** unterschieden: Modale Dialoge erzwingen den Eingabefokus und es kann mit der Anwendung erst weitergearbeitet werden wenn dieser geschlossen wird. Nicht-modale Dialoge erlauben das weiterarbeiten mit der Applikation und sollten auch nur verwendet werden, wenn dies möglich ist.

5.1 Nachrichten und Fragen

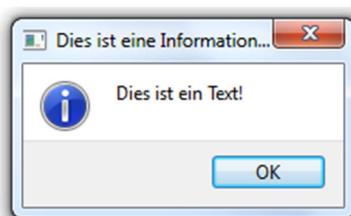
Nachrichtdialoge sind die einfachste Form von Dialogen. Diese Kategorie von Dialogen stellt eine Nachricht dar oder stellt eine Frage, welche verschiedene Antwortmöglichkeiten bietet. Diese Dialoge werden mit `QMessageBox` erstellt.

5.1.1 Informationsbox

Mit `QMessageBox.information(...)` wird das Informationsfenster erstellt. Der erste Parameter ist das parent Widget, also das Widget von dem das Informationsfenster gestartet wird, dies ist meist `self`, wenn das Informationsfenster innerhalb einer `QWidget`-Klasse erstellt wird.

```
titel = "Dies ist eine Informations-Box"
text = "Dies ist ein Text!"
QMessageBox.information(self, titel, text)
```

Der Text kann auch HTML-Tags enthalten und wird dann entsprechend formatiert.



Beispiel der `QMessageBox.information()`

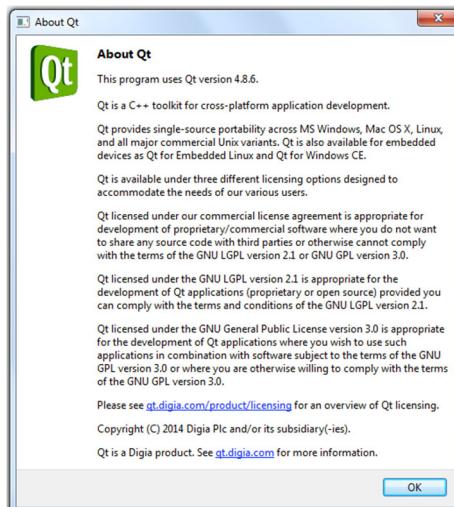
5.1.2 About und AboutQt Dialoge

Die meisten Applikationen haben einen "About" Dialog, welcher Informationen zur Applikation liefert. Unter MacOS kann ein About-Menu immer beim Applikations-Menu gefunden werden. Unter Windows ist dies nicht einheitlich. Um ein About-Fenster anzuzeigen kann `QMessageBox.about(...)` aufgerufen werden.

QMessageBox.about(self, titel, text)

Beispiel des QMessageBox.about()

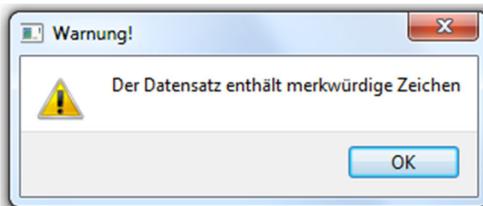
Ein spezielles About-Fenster, welches Informationen zu Qt anzeigt, kann mit `QMessageBox.aboutQt(...)` aufgerufen werden.

QMessageBox.aboutQt(self) # Informationen über Qt

Screenshot aboutQt()-Dialog

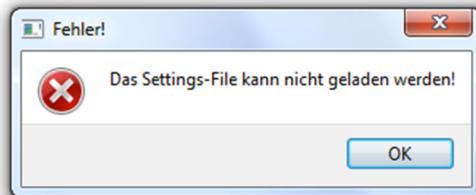
5.1.3 Warnungen und kritische Fehler

Warnungs-Dialoge können mit `QMessageBox.warning(...)` und Dialoge für kritische Fehler mit `QMessageBox.critical(...)` erstellt werden. Diese Dialoge werden in der Regel mit einem Ok Button bestätigt.

QMessageBox.warning(self, "Warnung!", "Der Datensatz enthält merkwürdige Zeichen")

QMessageBox.warning(...)

```
QMessageBox.critical(self, "Fehler!", "Das Settings-File kann nicht geladen werden!")
```

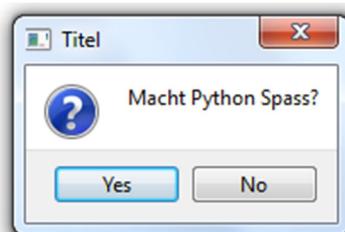


QMessageBox.critical(...)

5.1.4 Fragen

Ein Fragedialog stellt eine Frage, welche in der Regel mit Ja und Nein beantwortet werden kann. Der Fragedialog wird mit `QMessageBox.question(. . .)` realisiert. Die Antwort kann danach weiter verarbeitet werden.

```
antwort = QMessageBox.question(self, "Titel", "Macht Python Spass?",  
                             QMessageBox.Yes, QMessageBox.No)  
if antwort == QMessageBox.Yes:  
    print("Sehr gut!")  
elif antwort == QMessageBox.No:  
    print("Sehr schade!")
```



Dialog mit QMessageBox.question(...)

5.2 Datei-Dialoge

Datei-Dialoge für das Öffnen oder Speichern von Dateien sind häufig verwendete Dialoge. PyQt stellt dafür vier Arten von Datei- und Verzeichnisdialogen bereit. Diese Dialoge liefern eigentlich nur jeweils Filenamen resp. Verzeichnissnamen.

Die einfachste Art einen existierenden Dateinamen zu erhalten ist über die statische Funktion `QFileDialog.getOpenFileName(. . .)`. Der erste Parameter ist das Widget von dem es aufgerufen wird (meistens `self`), der zweite Parameter ist der Titel des Fensters, der dritte Parameter ist das Verzeichnis in dem gestartet wird und im

vierten Parameter werden die Unterstützten Filetypen angegeben. Sehen wir uns folgendes Beispiel an:



Dialog mit `QFileDialog.getOpenFileName(...)`

```
filename, filter = QFileDialog.getOpenFileName(self,
                                              "Datei öffnen",
                                              "C:/data/",
                                              "Bilder (*.jpg *.png)")

if (filename != ""):
    print(filename)
    print(filter)
```

Falls "Abbrechen" gewählt wurde, so ist der Filename leer. Im Filter ist der gewählte Filter. Der Filter wird später noch detaillierter beschrieben.

Der dritte Parameter sollte eigentlich nie ein absoluter Pfad wie im Beispiel sein. Mit `QDir` und den `QStandardPaths` können bestimmte Verzeichnisse ausgegeben werden. In folgender Tabelle sind die wichtigsten Verzeichnisse aufgelistet:

<code>QDir.currentPath()</code>	Das aktuelle Verzeichnis
<code>None</code>	
<code>QStandardPaths.storageLocation(QStandardPaths::DocumentsLocation)</code>	Dokument-Ordner
<code>QStandardPaths.storageLocation(QStandardPaths::FontsLocation)</code>	Font-Ordner
<code>QStandardPaths.storageLocation(QStandardPaths::ApplicationsLocation)</code>	Anwendungen
<code>QStandardPaths.storageLocation(QStandardPaths::MusicLocation)</code>	Musik-Ordner
<code>QStandardPaths.storageLocation(QStandardPaths::MoviesLocation)</code>	Filme-Ordner
<code>QStandardPaths.storageLocation(QStandardPaths::PicturesLocation)</code>	Bilder-Ordner
<code>QStandardPaths.storageLocation(QStandardPaths::TempLocation)</code>	Temporäre Daten
<code>QStandardPaths.storageLocation(QStandardPaths::HomeLocation)</code>	Home Verzeichnis
<code>QStandardPaths.storageLocation(QStandardPaths::DesktopLocation)</code>	Desktop

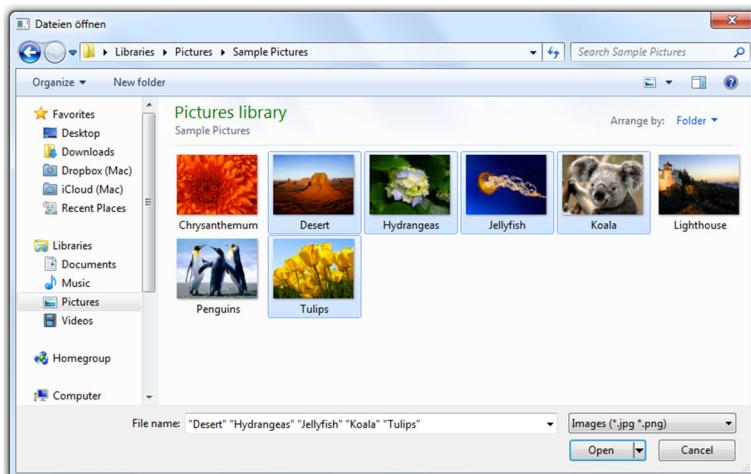
Siehe auch: <https://doc.qt.io/qt-5/qstandardpaths.html>

Der nächste File-Dialog ermöglicht das Wählen von **mehreren Dateien**. Dieser File-Dialog wird mit `QFileDialog.getOpenFileNames(...)` geöffnet. Diese Funktion funktioniert wie die vorherige mit dem Unterschied, dass neben dem Filter eine Liste von Filenamen zurückgegeben wird:

```
Location = QDesktopServices.storageLocation(QDesktopServices.PicturesLocation)
dateien, filter = QFileDialog.getOpenFileNames(self,
                                              "Dateien öffnen",
                                              Location,
                                              "Images (*.jpg *.png)")

if len(dateien)>0:
    for file in dateien:
        print(file)
print(filter)
```

Wenn "Abbrechen" gedrückt wurde, so ist die Liste leer.

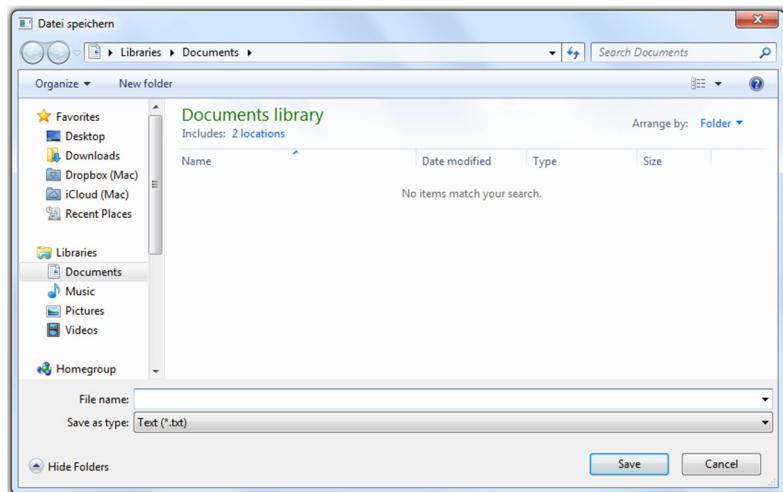


Mehrere Files mit `QFileDialog.getOpenFileNames(...)`

Der nächste Filedialog-Typ ist für das Speichern von Dateien. Es ist dasselbe Prinzip wie beim Laden einer Datei.

Der Speicher-File-Dialog wird mit `QFileDialog.getSaveFileName(...)` aufgerufen.

```
Location = QDesktopServices.storageLocation(QDesktopServices.DocumentsLocation)
filename, filter = QFileDialog.getSaveFileName(self,
                                              "Datei speichern",
                                              Location,
                                              "Text (*.txt)")
```



Dialog für das Speichern einer Datei: QFileDialog.getSaveFileName(...)

Wir haben bei den File-Dialogen die Filter gesehen, z.B. "Text (*.txt)". Wird der Filter weggelassen, so werden alle Dateien "All Files (*.*)" unterstützt. Sollen verschiedene Filter-Typen unterstützt werden können diese durch zwei Semikolon getrennt angegeben werden: "Text (*.txt);;CSV (*.csv)". Es können auch mehrere Einträge pro Filter-Kategorie gemacht werden. Diese werden dann einfach durch ein Leerzeichen getrennt: "Images (*.png *.jpg *.tif *.bmp)"

Beim Speichern-Dialog werden zwei Werte zurückgegeben: Der erste Wert ist der Dateiname und der zweite ist der Filter. Die File-Extension wird automatisch hinzugefügt, sollte diese fehlen - bei mehreren Filtern wird der Erste genommen.

Wird beim Speichern-Dialog auf "Abbrechen" geklickt, so ist der Filename leer ("").

5.3 Eingaben abfragen

Eine spezielle Form für die Eingabe von Werten geht über die Eingabedialoge der QInputDialog Klasse. Diese Eingabemethode kann aber ab und zu ganz nützlich sein. Es gibt verschiedene Typen dieser Eingabedialoge, welche hier nun vorgestellt werden.

5.3.1 Zahlen abfragen

Mit QInputDialog.getInt(...) kann eine ganze Zahl eingelesen werden. Der Funktionsaufruf sieht folgendermassen aus:

```
ergebnis, ok = QInputDialog.getInt(self,
                                      Titel,
                                      Text,
                                      Anfangswert,
                                      Minimalwert,
```

Maximalwert, Schrittweite)

Ein konkretes Beispiel dazu wäre folgendes Ratespiel:

```
ergebnis, ok = QInputDialog.getInt(self,
                                    "InputDialog.getInt()",  

                                    "Was ist 10+7 ? ",  

                                    0, 0, 20, 1)  

if ok:  

    if ergebnis == 17:  

        print("Richtig!")
```

Eine alternative Form für das Einlesen von Zahlen ist `QInputDialog.getDouble(...)`. Die Funktion ist wie `getInt(...)` aufgebaut, mit dem Unterschied, dass der letzte Parameter die Anzahl Nachkommastellen beinhaltet.

```
ergebnis, ok = QInputDialog.getDouble(self,
                                       "InputDialog.getDouble()",  

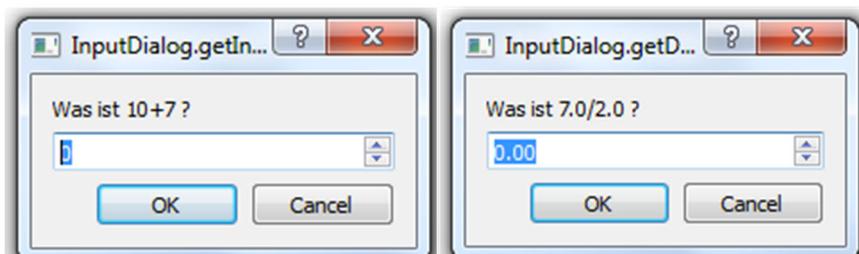
                                       "Was ist 7.0/2.0 ? ",  

                                       0, 0, 10, 2)  

if ok:  

    if ergebnis == 3.5:  

        print("Richtig!")
```



Input-Dialoge für Zahlen

5.3.2 Aus Werten auswählen

Mit der Funktion `QInputDialog.getItem(...)` kann aus einer Liste von Werten ausgewählt werden. Dazu wird einfach eine Liste mit Zeichenketten angelegt.

```
ergebnis, ok = QInputDialog.getItem(self,  

                                     Titel,  

                                     Text,  

                                     Werte,  

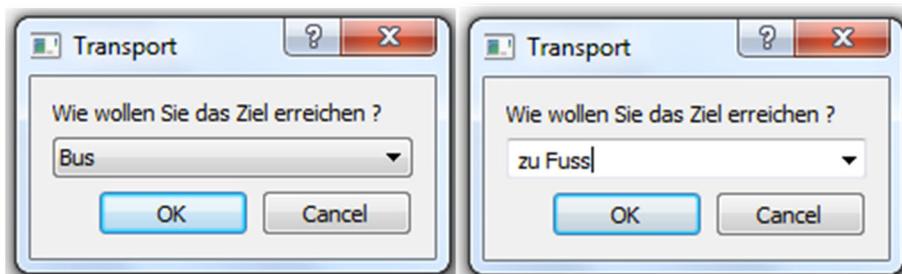
                                     AktuellesElement,
```

Editierbar)

Werte ist eine Liste von Elementen, wie z.B. ["Glas", "Metall", "Holz"].
AktuellesElement ist das zuerst angezeigte Element, beginnend bei 0. *Editierbar* ist True, falls ein eigener Wert hinzugefügt werden kann, ansonsten False.
Ok ist ein bool, welcher True ist, falls "Ok" gedrückt wurde. Das Ergebnis enthält die ausgewählte Zeichenkette (oder auch den eingegebenen Text).

Ein konkretes Beispiel dazu wäre:

```
ergebnis, ok = QInputDialog.getItem(self, "Transport",
                                      "Wie wollen Sie das Ziel erreichen ?",
                                      ["Velo", "Bus", "Eisenbahn", "Auto"],
                                      1, False)
if ok:
    print(ergebnis)
```



QInputDialog.getItem(...) mit Editierbarkeit ein- und ausgeschalten

5.3.3 Eine Zeichenkette einlesen

Mit `QInputDialog.getText(...)` kann eine Zeichenkette eingelesen werden. Dies geschieht mit folgendem Aufruf:

```
eingabe, ok = QInputDialog.getText(self,
                                    Titel,
                                    Text,
                                    Modus,
                                    Inhalt)
```

Titel und *Text* sind wie bei den vorherigen Dialogen.

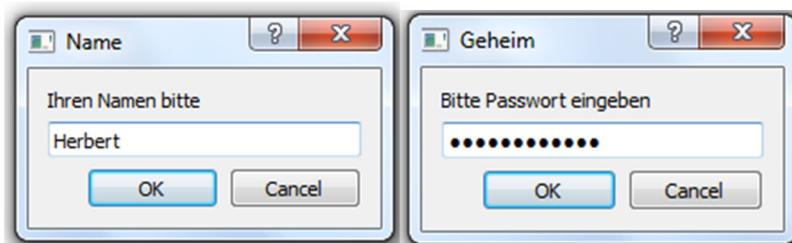
Für den *Modus* gibt es drei Möglichkeiten:

- Text normal einlesen (`QLineEdit.Normal`)
- Text als Passwort einlesen (`QLineEdit.Password`)
- Text als Passwort einlesen ohne sichtbare Zeichen (`QLineEdit.NoEcho`)

Inhalt füllt die Texteingabe mit einem vordefinierten Wert. Normalerweise ist diese leer und hat damit den Wert "".

Sehen wir uns folgendes konkretes Beispiel an:

```
eingabe, ok = QInputDialog.getText(self, "Name",
                                    "Ihren Namen bitte",
                                    QLineEdit.Normal, "")  
  
if ok:  
    print("Eingabe:", eingabe)
```



QInputDialog.getText(...) mit verschiedenen Modi

Der Rückgabewert ist jeweils der eingegebene Text als Zeichenkette, sofern `ok` True ist.

5.4 Farbdialog

Ein weiterer wichtiger Dialog ist das auswählen einer Farbe. Der Aufruf eines Farbdialoges ist sehr einfach:

```
color = QColorDialog.getColor([initial=QColor])
```

color ist vom Typ "QColor" und hat unzählige Attribute und Methoden. Einige davon sind:

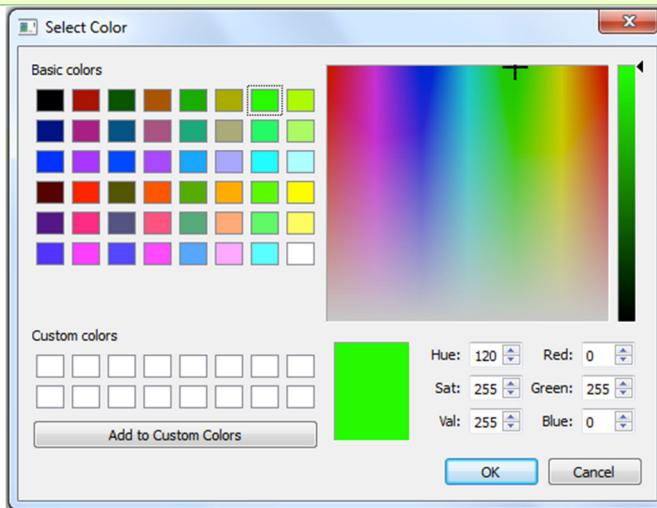
<code>QColor.red()</code>	Gibt den Rot-Wert als ganze Zahl [0,255] zurück
<code>QColor.green()</code>	Gibt den Grün-Wert als ganze Zahl [0,255] zurück
<code>QColor.blue()</code>	Gibt den Blau-Wert als ganze Zahl [0,255] zurück
<code>QColor.redF()</code>	Gibt dem Rot-Wert als floating point im Bereich [0,1] zurück
<code>QColor.greenF()</code>	Gibt dem Grün-Wert als floating point im Bereich [0,1] zurück
<code>QColor.blueF()</code>	Gibt dem Blau-Wert als floating point im Bereich [0,1] zurück

Ein konkretes Beispiel wäre:

```
color = QColorDialog.getColor()  
print(color.red(), color.green(), color.blue())
```

Soll eine Farbe zu Beginn schon im Farbdialog gewählt werden, so geht das mit dem keyword Parameter "initial":

```
color = QColorDialog.getColor(initial=QColor(255,0,0))
```

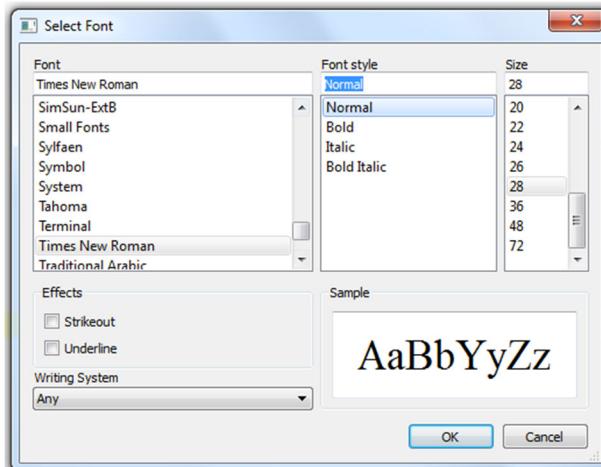


Beispiel eines Farbdialogs

5.5 Fontdialog

Zur Auswahl einer Schriftart kann der Font-Dialog nützlich sein. Mit dem Fontdialog kann eine im System vorhandene Schrift und deren Attribute gewählt werden:

```
font = QFontDialog.getFont([initial=QFont])
```



Der Rückgabewert ist vom Typ `QFont`.

Ein `QWidget` kann eine Font erhalten mit `widget.setFont(font)`.

Weitere Informationen zu Font-Dialogen kann in der PyQt oder Qt Dokumentation nachgeschlagen werden.

5.6 Druck- und Druckvorschau-Dialog

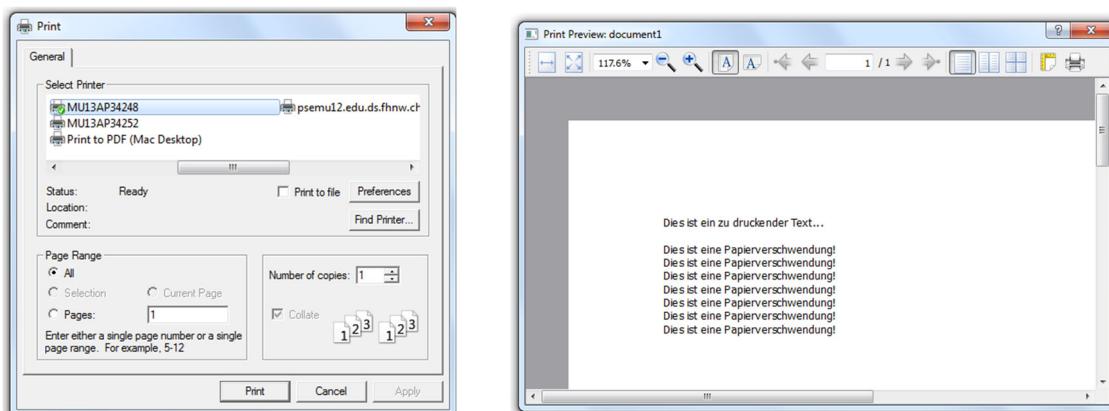
Drucken ist in PyQt denkbar einfach. Es können Dokumente aus Widgets gedruckt werden, wie zum Beispiel der Text aus einem `QTextEdit`. In Kapitel 0 werden wir komplexere Vektor-Grafiken erstellen, die auch ausgedruckt werden können. Dies geschieht über den `QPrintDialog()`.

Ein Druckdialog und Ausdruck aus einem `QTextEdit` (im Beispiel `self.textEdit`) kann folgendermassen erstellt werden:

```
dialog = QPrintDialog()
if dialog.exec() == QDialog.Accepted:
    doc = self.textEdit.document()
    doc.print_(dialog.printer())
```

Manchmal ist es auch sinnvoll eine Vorschau eines zu druckenden Elements darzustellen, dies kann im Falle unseres Text-Edits folgendermassen mithilfe von `QPrintPreviewDialog()` erstellt werden:

```
dialog = QPrintPreviewDialog()
dialog.paintRequested.connect(self.textEdit.print_)
dialog.exec()
```



`QPrintDialog()` und `QPrintPreviewDialog()`

5.7 Eigene Dialoge erstellen

Einen eigenen Dialog kann über die Klasse `QDialog` erstellt werden. Dies geschieht sehr ähnlich wie wir es bereits vom `QMainWindow` kennen. Dazu wird eine neue Klasse erstellt, welche von `QDialog` vererbt wird. Sehen wir uns folgendes Beispiel an:

```
class MyDialog(QDialog):
    def __init__(self, parent):
        super().__init__(parent)
        label = QLabel("Bitte Button klicken!")
        button = QPushButton("PushButton")
        layout = QHBoxLayout()
        layout.addWidget(label)
        layout.addWidget(button)
        self.setLayout(layout)
        button.clicked.connect(self.close)

# Dialog öffnen (innerhalb einer anderen QWidget-Klasse)
dialog = MyDialog(self)      # Dialog erstellen
dialog.exec()                # Dialog loop-starten
print("Dialog ende!")       # wird aufgerufen, wenn Dialog geschlossen
```

Der Unterschied zum bisher bekannten `QMainWindow` ist, dass ein Dialog über `exec()` ausgeführt werden muss. Das Programm fährt erst im Programm weiter, wenn der Dialog geschlossen ist. Hier sehen wir das Prinzip eines **modalen Dialogs**.

Gibt es in einem Projekt mehrere Fenster und Dialoge, so können diese in verschiedenen Python-Files stehen und mittels `import` dazugeladen werden. So können dieselben Dialoge in mehreren Projekten verwendet werden.

6 Arbeiten mit dem QtDesigner

Eine GUI-Anwendung zu entwickeln ist nicht nur die Arbeit eines Software-Entwicklers. Es gibt GUI-Designer, welche sich auf diese Aufgabe spezialisiert haben. Auch das Verändern vom GUI (z.B. neuanordnen von GUI-Elementen) sollte – gerade bei grösseren Projekten - nicht abhängig von Python-Code sein. Ein weiterer Aspekt ist die Mehrsprachigkeit von Applikationen: Es sollte auch für einen nicht-Programmierer recht einfach sein eine Applikation in eine andere Sprache zu übersetzen, also z.B. von Englisch auf Chinesisch.

PyQt hat zahlreichen Tools, welche diese Aufgaben einfacher lösen können. In diesem Kapitel sehen wir uns diese ein wenig an.

Der **Qt-Designer** ist eine grafische Anwendung zum Erstellen von Benutzeroberflächen. Das Tool erlaubt das „zusammenklicken“ einer GUI. Mit dem **uic Tool** wandeln wir die Benutzeroberfläche, welche wir mit dem Qt-Designer erstellt haben in Python-Code um.

6.1 Starten des Qt-Designer

Der Qt-Designer befindet sich unter Windows im Python Ordner unter Anaconda3/Library/bin

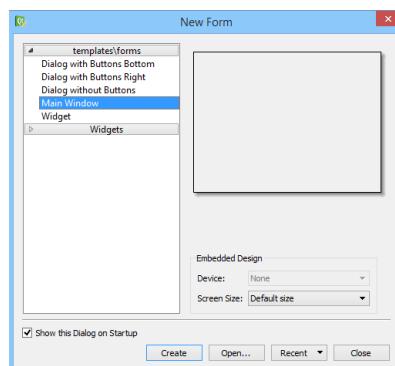
Wo genau designer.exe ist kann mit diesem Programm herausgefunden werden:

```
import sys
import os
s = os.path.split(sys.executable)
os.startfile(s[0] + "/Library/bin")
```

6.2 Das erste Projekt mit Qt-Designer

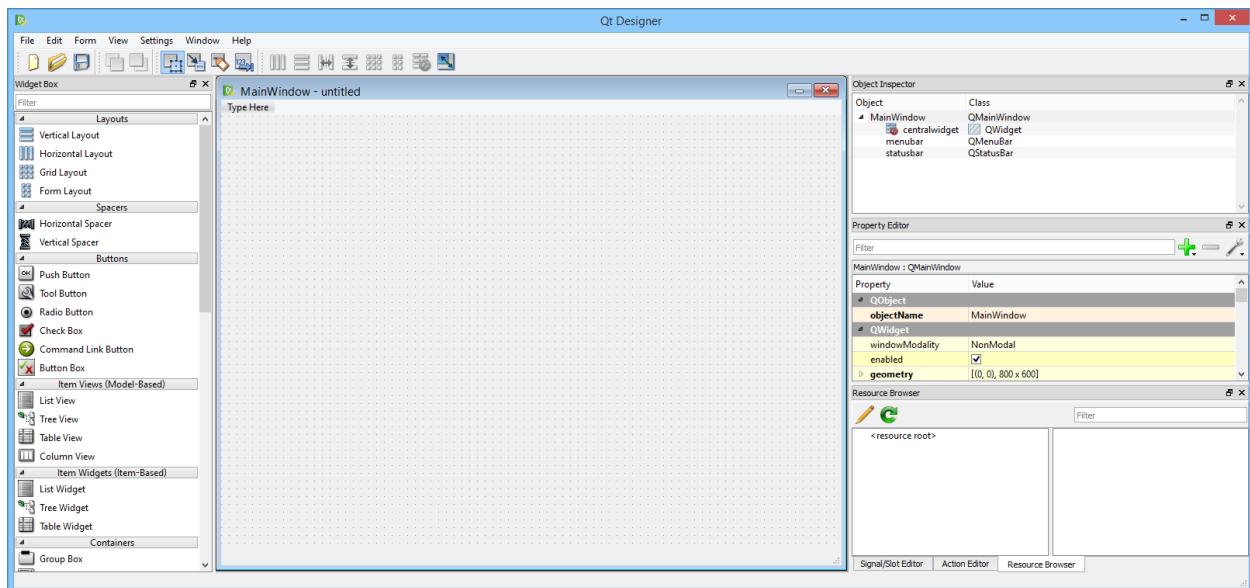
Zunächst erstellen wir mit PyCharm ein neues Projekt. Danach öffnen wir den Qt-Designer.

Nach dem Öffnen des Qt-Designer kann im Startup Dialog ein „Main Window“ gewählt werden. Mit diesem erstellen wir ein Hauptfenster.



Der Startup Dialog von Qt-Designer

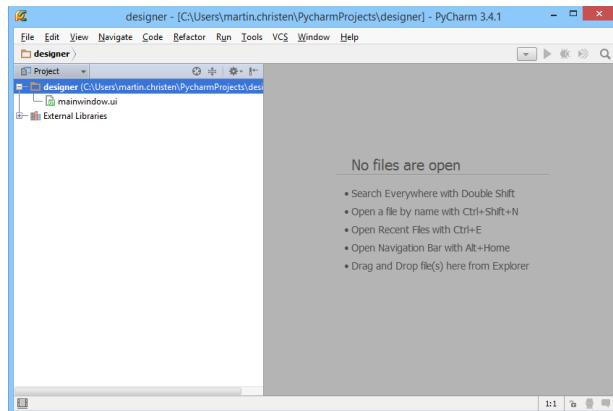
Es erscheint ein leeres Fenster, welches unter Windows so aussehen sollte:



Der Qt-Designer mit einem leeren QMainWindow

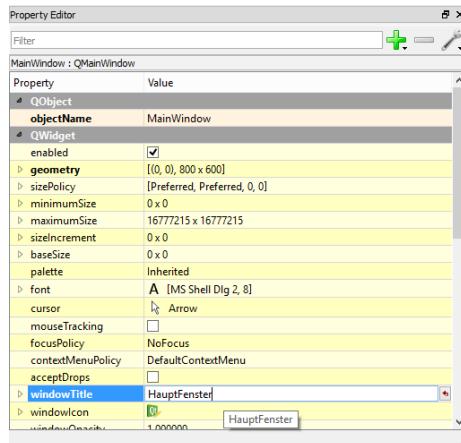
Nun können wir das Projekt bereits abspeichern:

Das *.ui File - wir nennen es `mainwindow.ui` - soll nun beim zuvor erstellen PyCharm Projekt gespeichert werden.



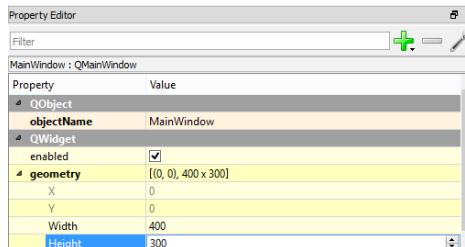
PyCharm mit einem ui-File

Nach Klick auf das „MainWindow“ können wir den Namen des Fensters editieren. Dies geschieht durch Ändern von `windowTitle` im **Property Editor**:



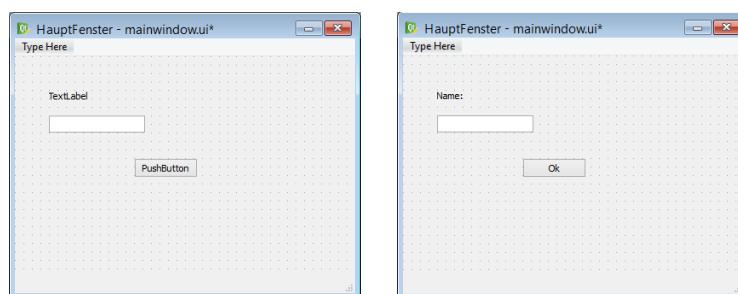
Setzen des Fenster-Titels

Danach Ändern wir die Fenstergrösse auf 400x300. Dies geschieht durch Ändern von `geometry`:



Ändern der Fenstergrösse

Nun können wir im MainWindow einen „Push Button“, ein „Label“ und ein „Text Edit“ hinzufügen. Dies geschieht indem wir die entsprechenden Widgets mit Drag&Drop von der Widget Box in das Hauptfenster ziehen. Das Resultat kann zum Beispiel so aussehen:



Widgets auf dem Main-Window vor und nach der Umbenennung

Durch Doppelklick auf den Label oder auf den PushButton kann der Name geändert werden. Wir ändern den Label auf „Name“ und den Push Button auf „Ok“.

6.3 Ui-File in Code umwandeln

Mit dem Tool pyuic5 können wir aus einem UI-File direkt Code erstelleb.

```
pyuic5 -x uifile.ui -o mypythoncode.py
```

6.4 Ui-Files direkt im Code verwenden

Ui files können auch direkt verwendet werden.

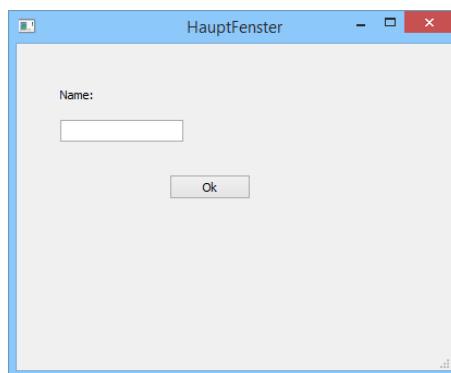
```
import sys
from PyQt5.QtWidgets import *
from PyQt5.uic import *

# Eine Qt-Applikation erstellen:
app = QApplication(sys.argv)

window = loadUi("uifile.ui") # GUI aus ui-File laden:
window.show() # Fenster anzeigen

# Application-Loop starten
app.exec()
```

Und wir sehen, dass unser zuvor erstelltes GUI nun korrekt dargestellt wird:



Das im Qt-Designer erstellte GUI als Python Programm

Nun können wir das GUI mit dem Qt-Designer einfach verändern, und die Änderungen sind im Programm direkt sichtbar!

6.5 Verwenden von Layouts und Signals/Slots

Die geladenen UI-Files agieren wie die in den vorherigen Kapiteln erzeugten Klassen. Mit „connect“ können Interaktionen erstellt werden und mit den im UI-Designer definierten Namen kann direkt auf die Widget-Instanz zugegriffen werden.

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

def buttonClick():
    print("Button Clicked")

# Eine Qt-Applikation erstellen:
app = QApplication(sys.argv)

mainwindow = loadUi("uifile.ui")

# Nun kann auf die im UI-Designer gesetzten Namen zugegriffen werden
mainwindow.MeinLabel.setText("Dies ist ein Label!")
mainwindow.pushButtonOk.clicked.connect(buttonClick)

mainwindow.show() # Fenster anzeigen

# Application-Loop starten
app.exec()
```

6.6 Objektorientiert arbeiten

Wir können auch loadUi innerhalb des Konstruktors ausführen, und so ganz gewohnt objektorientiert arbeiten. Danach können alle connects wie gewohnt gemacht werden.

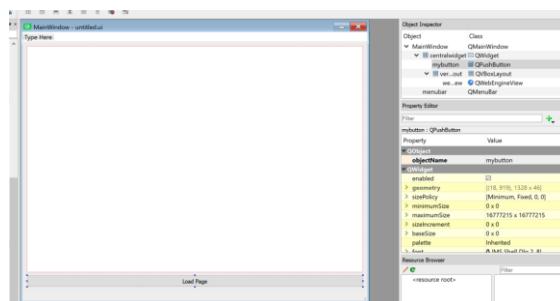
```
from PyQt5 import uic
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
import sys

class UiWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        uic.loadUi('uifile.ui', self)
        self.show()

app = QApplication(sys.argv)
window = UiWindow()
app.exec()
```

6.7 Erstellen eines Webbrowsers

Dies erfordert eventuell die Installation von PyQtWebEngine (danach restart Designer)
conda install pyqtwebengine



```
from PyQt5.uic import *
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *
from PyQt5.QtWebEngineWidgets import QWebEngineView

class UiWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        loadUi('uifile.ui', self)
        self.show()

        self.mybutton.clicked.connect(self.loadPage)

    def loadPage(self):
        #self.webEngineView.setHtml("<h1>Hello World</h1>")
        self.webEngineView.load(QUrl("http://www.fhnw.ch/"))

app = QApplication([])
window = UiWindow()
app.exec()
```

Die QWebEngine View hat einige wichtige Signale, wie z.B.

```
loadFinished(ok : bool)
loadProgress(progress : int)
urlChanged(url: QUrl)
```

Es ist u.a. auch möglich JavaScript auszuführen (am besten nach loadFinished):

```
self.webEngineView.page().runJavaScript("document.getElementsByName('bla')[0].value", self.store_value)
```

7 Numerisches Python I: Grundlagen

Numerisches Programmieren ist auch bekannt unter dem eher irreführenden Namen „wissenschaftliches Programmieren“. Unter numerischem Programmieren versteht man das Gebiet der Informatik und Mathematik, bei dem es um Approximationsalgorithmen geht, d.h. numerische Approximation von mathematischen Problemen oder numerischer Analysis.

Wir werden in dieser Einführung folgende Module kennenlernen:



Matplotlib

Matplotlib ist eine Python-Bibliothek zur Erstellung von Datenvisualisierungen in verschiedenen Formen, darunter Linien- und Flächenplots, Histogramme, Balkendiagramme und Tortendiagramme. Sie bietet eine Vielzahl von Optionen zur Anpassung von Farben, Beschriftungen, Achsen und anderen Aspekten der Grafiken. Matplotlib ist ebenfalls eine Kernbibliothek des wissenschaftlichen Python-Ökosystems und wird häufig mit Numpy und anderen Bibliotheken kombiniert.

NumPy

Numpy ist eine Python-Bibliothek, die für numerische Berechnungen und die Arbeit mit grossen, mehrdimensionalen Arrays und Matrizen optimiert ist. Sie bietet eine Vielzahl von Funktionen und Methoden zur Datenmanipulation und mathematischen Berechnung, einschliesslich lineare Algebra, Fourier-Transformationen und Zufallszahlengenerierung. Numpy ist eine der Kernbibliotheken des wissenschaftlichen Python-Ökosystems und wird von vielen anderen Bibliotheken und Anwendungen verwendet.

NumFOCUS

NumFOCUS ist eine gemeinnützige Organisation, die sich für die Förderung von Open-Source-Software und offenen Standards für wissenschaftliche Berechnungen und Datenanalyse engagiert. NumFOCUS unterstützt eine Vielzahl von Projekten und Gemeinschaften, darunter Numpy, Matplotlib und andere wichtige Bibliotheken im wissenschaftlichen Python-Ökosystem. Die Organisation bietet finanzielle und organisatorische Unterstützung, Schulungen und andere Ressourcen für diese Projekte und hilft dabei, sie für eine breitere Benutzerschaft zugänglich zu machen.

7.1 Einführung in Matplotlib

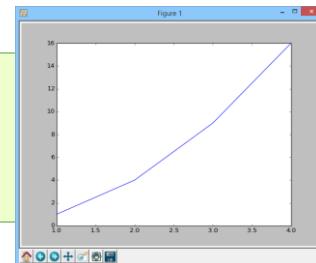
7.1.1 Grundlagen Plotten

Matplotlib (<https://matplotlib.org/>) ist eine sehr umfassende Bibliothek zum Plotten von Daten.

Wenn wir beispielsweise Punkte haben und diese plotten wollen, können wir das ganz einfach mit der `plot` Funktion tun:

```
import matplotlib.pyplot as plt

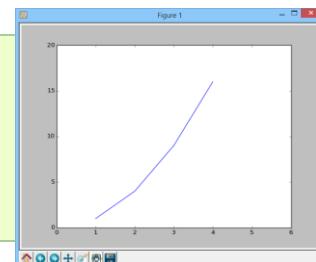
plt.plot([1,2,3,4], [1,4,9,16])
plt.show()
```



Wir können mit `axis()` noch den Bereich welcher dargestellt wird angeben:

```
import matplotlib.pyplot as plt

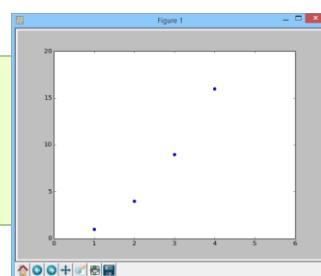
plt.plot([1,2,3,4], [1,4,9,16])
plt.axis([0,6,0,20])
plt.show()
```



Ähnlich wie in Matlab kann auch noch angegeben werden wie das Resultat dargestellt wird. Standardmäßig wird mit einer blauen Linie geplottet (Option "b-")

Sollen nun Punkte dargestellt werden so kann die Option "bo" für blaue Punkte oder "ro" für rote Punkte verwendet werden:

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], "bo")
plt.axis([0,6,0,20])
plt.show()
```



Die folgenden Farbabkürzungen sind möglich, alternativ kann auch der Parameter «color» verwendet werden, um eine Farbe zu definieren.

'b'	blau
'g'	grün
'r'	rot
'c'	cyan
'm'	magenta
'y'	gelb
'k'	schwarz
'w'	weiss

Der Linienstil resp. die Marker können folgendermassen gesteuert werden:

' - '	(Bindestrich) durchgezogene Linie
'---'	(zwei Bindestriche) gestrichelte Linie
'-..'	Strichpunkt-Linie
'.:.'	punktierte Linie
'. .'	Punkt-Marker
',.'	Pixel-Marker
'o'	Kreis-Marker
'v'	Dreiecks-Marker, Spitze nach unten
'^'	Dreiecks-Marker, Spitze nach oben
'<'	Dreiecks-Marker, Spitze nach links
'>'	Dreiecks-Marker, Spitze nach rechts
'1'	tri-runter-Marker
'2'	tri-hoch-Marker
'3'	tri-links Marker
'4'	tri-rechts Marker
's'	quadratischer Marker
'p'	fünfeckiger Marker
'*'	Stern-Marker
'h'	Sechseck-Marker1
'H'	Sechseck-Marker2
'+'	Plus-Marker
'x'	x-Marker
'D'	rautenförmiger Marker
'd'	dünner rautenförmiger Marker
' '	Marker in Form einer vertikalen Linie
'_'	Marker in Form einer horizontalen Linie

Eine Beschriftung der Achse kann mit den Funktionen `xlabel("name")` und `ylabel("name")` eingefügt werden:

Mit `grid(True)` ist es zudem möglich ein Koordinatengitter darzustellen.

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], "b^")
plt.axis([0,6,0,20])
plt.xlabel("Äpfel")
plt.ylabel("Birnen")
plt.grid(True)

plt.show()
```

- Wir können übrigens auch direkt im Jupyter Lab plotten
- Visual Studio kann auch mit jupyter notebooks umgehen!

7.2 Einführung zu Numpy

Dies ist nur ein ganz kleiner Einstieg in NumPy, denn diese ist eine sehr grosse Bibliothek. Die Dokumentation von NumPy ist auf <http://docs.scipy.org/doc/> zu finden. Die Numpy Referenz (PDF) umfasst ca. 1500 Seiten. Ein guter Start in NumPy bietet sich auf: <http://docs.scipy.org/doc/numpy/user/basics.html> an.

Nach der Installation des NumPy Moduls können wir das Modul wie gewohnt importieren:

```
import numpy
```

Da numpy relativ oft geschrieben werden muss, bevorzugen viele Leute folgenden import:

```
import numpy as np
```

Damit können wir anstelle von „numpy“ ganz einfach „np“ schreiben. Wir verwenden auch diese Schreibweise.

Eine der wichtigsten Klassen von numpy ist die Array-Klasse. Diese ist sehr ähnlich wie eine Liste, jedoch darf darin nur genau ein Datentyp pro Array vorkommen. Es gibt in numpy mehr Datentypen als bei Standard-Python, jedoch sind diese Hardware-näher.

<code>np.bool_</code>	Boolean (True oder False)
<code>np.int_</code>	Standard integer Typ (32- oder 64-bit je nach Python)
<code>np.int8</code>	Byte (-128 to 127)
<code>np.int16</code>	Integer (-32768 to 32767)
<code>np.int32</code>	Integer (-2147483648 to 2147483647)
<code>np.int64</code>	Integer (-9223372036854775808 to 9223372036854775807)
<code>np.uint8</code>	Unsigned integer (0 to 255)
<code>np.uint16</code>	Unsigned integer (0 to 65535)
<code>np.uint32</code>	Unsigned integer (0 to 4294967295)
<code>np.uint64</code>	Unsigned integer (0 to 18446744073709551615)
<code>np.float_</code>	Abkürzung für float64.
<code>np.float16</code>	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
<code>np.float32</code>	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
<code>np.float64</code>	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
<code>np.complex_</code>	Abkürzung für complex128.
<code>np.complex64</code>	Komplexe Zahl, bestehend aus zwei 32-bit floats
<code>np.complex128</code>	Komplexe Zahl, bestehend aus zwei 64-bit floats

```
a = np.array([2,4,6,8], dtype=np.float64)
a = np.array([2,4,6,7])
```

Der Datentyp kann mit `dtype` abgefragt werden:

```
a.dtype
```

Ein Array kann auch in ein Array eines anderen Typs konvertiert werden:

```
b = a.astype(np.float64)
```

Auf die Elemente kann wie bei einer Liste zugegriffen werden, z.B.

```
a[0]  
a[2:3]
```

Es ist auch möglich mehrdimensionale Arrays zu erstellen, zum Beispiel ein zweidimensionales Array würde folgendermassen konstruiert:

```
a = np.array([[1,2,3], [4,5,6]], np.float64)
```

Bei mehrdimensionalen Arrays funktioniert das „Slicing“ sehr ähnlich, nur müssen wir die weiteren Dimensionen beachten:

Nehmen wir vorheriges Array, so gilt folgendes:

```
a[:]
```

hat den Wert

```
array([[ 1.,  2.,  3.],  
       [ 4.,  5.,  6.]])
```

also das gesamte Array

```
a[0,:]
```

hat den Wert `array([1., 2., 3.])`

```
a[1,:]
```

hat den Wert `array([4., 5., 6.])`

```
a[:,2]
```

hat den Wert `array([3., 6.])`

Wenn wir die Dimension abfragen wollen, können wir das mit dem Attribut „shape“. Dies liefert ein Tuple mit der Grösse des Arrays, also in unserem Beispiel wäre

```
a.shape
```

(2, 3) also ein 2x3 grosses Array. (2 Zeilen, 3 Spalten)

Wir können mit „in“ relativ einfach überprüfen, ob ein bestimmter Wert im Array vorkommt:

```
5 in a
```

liefert `True` und

```
20 in a
```

liefert `False`

Es gibt noch weitere Möglichkeiten ein Array zu erststellen:

`np.zeros (m, n)` erstellt ein $m \times n$ grosses Array welches mit 0 gefüllt ist.

`np.ones (m, n)` erstellt ein $m \times n$ grosses Array welches mit 1 gefüllt ist.

`np.arange (...)` erstellt ein Array mit fortlaufenden Werten. Es gibt dabei verschiedene Parameter:

`np.arange (n)`: Erstellt ein Array mit Werten im Bereich $[0, n[$

`np.arange (a, b)`: Erstellt ein Array mit Werten im Bereich $[a, b[$

`np.arange (a, b, s)`: Erstellt ein Array mit Werten im Bereich $[a, b[$ mit Schrittweite s.

Mit `np.linspace (...)` können Arrays mit einem Bereich und einer bestimmten Anzahl Elementen erstellt werden.

`np.linspace (a, b, n)` erstellt ein Array im Bereich $[a, b]$ mit n Elementen. zum Beispiel:

`np.linspace (0, 1, 11)` erstellt zum Beispiel

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ])
```

Der Parameter `dtype=Typ` kann bei diesen Funktionen angehängt werden, falls der Datentyp angegeben werden soll.

Falls ein Array mit zufälligen Werten erstellt werden soll, geht das mit `np.random.random (n)`, welches ein Array mit n zufälligen Werten im Bereich $[0, 1[$ erstellt.

7.3 Mathematische Funktionen und Konstanten in Numpy

Numpy hat zahlreiche Mathematische Funktionen, welche mit Arrays funktionieren.

7.3.1 Trigonometrische Funktionen

Die wichtigsten trigonometrische Funktionen von numpy sind in der folgenden Tabelle aufgeführt. Dabei steht x jeweils für ein Array. Der Rückgabewert ist ein Array mit den jeweiligen Funktionswerten (elementweise ausgeführt).

np.sin(x)	Sinus Funktion
np.cos(x)	Cosinus Funktion
np.tan(x)	Tangens Funktion
np.arcsin(x)	Arkussinus Funktion
np.arccos(x)	Arkuskosinus Funktion
np.arctan(x)	Arkustangens Funktion
np.arctan2(x1, x2)	Arkustangens Funktion mit korrektem Quadranten (Elementweise x1/x2)
np.rad2deg(x)	Umwandlung Radian nach Grad
np.deg2rad(x)	Umwandlung Grad nach Radian

7.3.2 Arithmetische Funktionen

Die wichtigsten arithmetischen Funktionen von numpy sind:

<code>np.add(x1, x2)</code>	Elementweises Addieren
<code>np.subtract(x1, x2)</code>	Elementweises Subtrahieren
<code>np.multiply(x1, x2)</code>	Elementweises Multiplizieren
<code>np.divide(x1, x2)</code>	Elementweises Dividieren

7.3.3 Weitere Mathematische Funktionen.

Es gibt zahlreiche weitere Mathematische Funktionen. Eine Liste sämtlicher Funktionen sind unter: <http://docs.scipy.org/doc/numpy/reference/routines.math.html> zu finden.

7.3.4 Mathematische Konstanten

In numpy sind einige Konstanten eingebaut, dazu gehören `np.pi` für π und `np.e` für e. Weitere Konstanten sind im Modul scipy zu finden:
<https://numpy.org/doc/stable/reference/constants.html>

7.4 NumPy und Matplotlib

7.4.1 Plotten einer Sinuskurve

NumPy und Matplotlib funktionieren sehr gut zusammen. Wollen wir beispielsweise eine Sinuskurve im Bereich $[0, 2\pi]$ zeichnen so können wir mit Numpy das relativ einfach realisieren.

Zunächst importieren wir die erforderlichen Module:

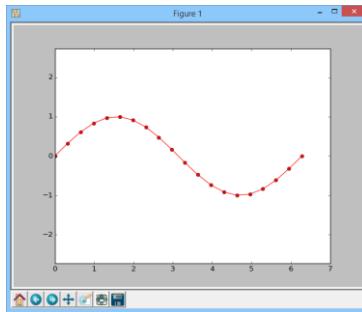
```
import numpy as np
import matplotlib.pyplot as plt
```

Dann erstellen wir ein Array x mit dem Wertebereich und ein Array y mit den Funktionswerten:

```
x = np.linspace(0, 2*np.pi, 20)
y = np.sin(x)
```

Danach können plotten:

```
plt.plot(x,y, "ro-")
plt.axis("equal")
plt.show()
```



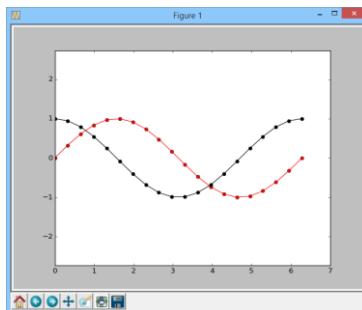
7.4.2 Plotten mehrerer Funktionen

Selbstverständlich können auch mehrere Funktionen geplottet werden. Wenn wir beispielsweise eine Sinuskurve und eine Cosinuskurve plotten wollen, so rufen wir einfach mehrmals die Funktion `plot` auf:

```
x = np.linspace(0, 2*np.pi, 20)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x,y1, "ro-")
plt.plot(x,y2, "ko-")

plt.axis("equal")
plt.show()
```

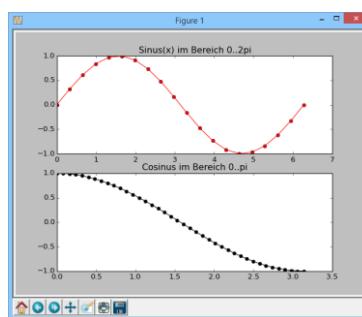


7.5 Subplots

Es können auch sogenannte „Subplots“ realisiert werden. Dies sind einfach mehrere Plots welche innerhalb eines Gitters angelegt werden. Zum Initialisieren wird die Funktion subplot verwendet:

```
plt.subplot(AnzahlZeilen, AnzahlSpalten, Nummer)
```

```
x1 = np.linspace(0, 2*np.pi, 20)
x2 = np.linspace(0, np.pi, 40)
y1 = np.sin(x1)
y2 = np.cos(x2)
plt.subplot(2,1,1)
plt.plot(x1,y1, "ro-")
plt.title("Sinus(x) im Bereich 0..2pi")
plt.subplot(2,1,2)
plt.plot(x2,y2, "ko-")
plt.title("Cosinus im Bereich 0..pi")
plt.show()
```



7.5.1 Plotten von Figuren

Mit Matplotlib lassen sich auch geometrische Formen relativ einfach zeichnen. Dazu benötigen wir das Axes Objekt, welches das Koordinatensystem definiert:

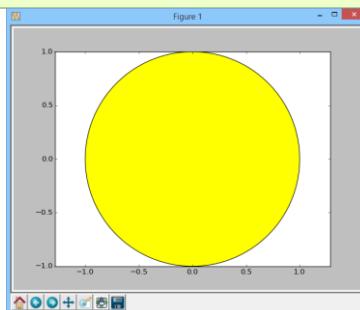
```
axes = plt.axes()
```

Auf diesem axes Objekt können wir unsere Figuren hinzufügen.
Sehen wir uns dazu folgendes komplettes Beispiel an:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches

axes = plt.axes()

circle = patches.Circle((0, 0), radius=1.0, facecolor=(1,1,0))
axes.add_patch(circle)
plt.axis("equal")
plt.plot()
```



in dem Beispiel haben wir mit `plt.Circle()` einen Kreis erstellt. Dazu haben wir das Zentrum $(0, 0)$ angegeben und ein Radius definiert. Zudem haben wir mit `facecolor` (oder `fc`) noch die Füllfarbe definiert. Die Farbe der Kante könnten wir mit `edgecolor` (oder `ec`) setzen. Die Liniendicke der Kante kann mit `linewidth` (oder `lw`) definiert werden.

Die wichtigsten Figuren(„Patches“) in sind:

Bogen	<pre>Arc(xy,width,height,angle=0.0,theta1=0.0,theta2=360.0)</pre> <p>xy: Zentrum als tuple z.B. (0,0) width: Länge der horizontalen Achse height: Länge der vertikalen Achse angle: Rotation in Grad (im Gegenuhrzeigersinn) theta1: Start-Winkel in Grad theta2: End-Winkel in Grad</p>	
Pfeil	<pre>Arrow(x, y, dx, dy, width=1.0)</pre> <p>x,y: Startposition dx, dy: Länge und Richtung width: Breite</p>	
Kreis	<pre>Circle(xy, radius=5)</pre> <p>xy: Zentrum als tuple radius: Radius des Kreises</p>	
Ellipse	<pre>Ellipse(xy, width, height, angle=0.0)</pre> <p>xy: Zentrum als tuple width: horizontale Achse height: vertikale Achse angle: Winkel (in Grad, Gegenuhrzeigersinn)</p>	
Polygon	<pre>Polygon(xy, closed=True)</pre> <p>xy: Liste von x,y-Koordinaten, z.B. [[x0,y0],[x1,y1],[x2,y2],...,[xn,yn]] closed: True, wenn Endpunkt mit Startpunkt verbunden werden soll</p>	
Rechteck	<pre>Rectangle(xy, width, height, angle=0.0)</pre> <p>xy: Koordinate unten links als tuple width: Breite height: Höhe angle: Winkel (in Grad, Gegenuhrzeigersinn)</p>	

8 Numerisches Python II – Matplotlib und PyQt5

8.1 Plotten mit Python Script

In Kapitel 7 haben wir bereits Matplotlib kennengelernt. Wir haben dazu Jupyter Lab verwendet um unsere Grafik darzustellen. Wir können das nun auch in einem Python-File versuchen:

```
import matplotlib.pyplot as plt
import numpy as np

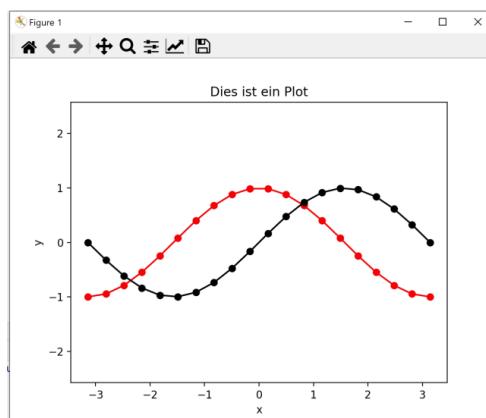
x = np.linspace(0, 2*np.pi, 20)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x,y1, "ro-")
plt.plot(x,y2, "ko-")

plt.title("Titel")
plt.xlabel("x")
plt.ylabel("y")

plt.axis("equal")
plt.show()
```

Wie wir sehen wird ein Fenster geöffnet und die Kurven werden geplottet. Dazu wird intern PyQt5 verwendet! Aber wir haben gar kein Fenster erstellt.



8.2 Matplotlib und PyQt5

Über das backend qt5agg kann mit PyQt5 gerendert werden («qt5 with antigrain», AGG ist eine platformunabhängige Grafikbibliothek für subpixelgenaues Rendering)

```
from PyQt5.QtWidgets import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.backends.backend_qt5agg import NavigationToolbar2QT as NavigationToolbar
import matplotlib.pyplot as plt
import numpy as np

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

        layout = QVBoxLayout()

        figure = plt.figure(figsize=(16,9))
        self.canvas = FigureCanvas(figure)
        #self.addToolBar(NavigationToolbar(canvas, self))

        button1 = QPushButton("y=sin(x)")
        button2 = QPushButton("y=cos(x)")

        button1.clicked.connect(self.plot1)
        button2.clicked.connect(self.plot2)

        layout.addWidget(self.canvas)
        layout.addWidget(button1)
        layout.addWidget(button2)

        center = QWidget()
        center.setLayout(layout)

        self.setCentralWidget(center)
        self.show()

    def plot1(self):
        plt.clf() # clear figure: löscht die aktuelle Figure
        x = np.linspace(-np.pi, np.pi, 20)
        y = np.sin(x)
        plt.plot(x, y, "ro-")
        plt.axis("equal")

        self.canvas.draw()
```

```

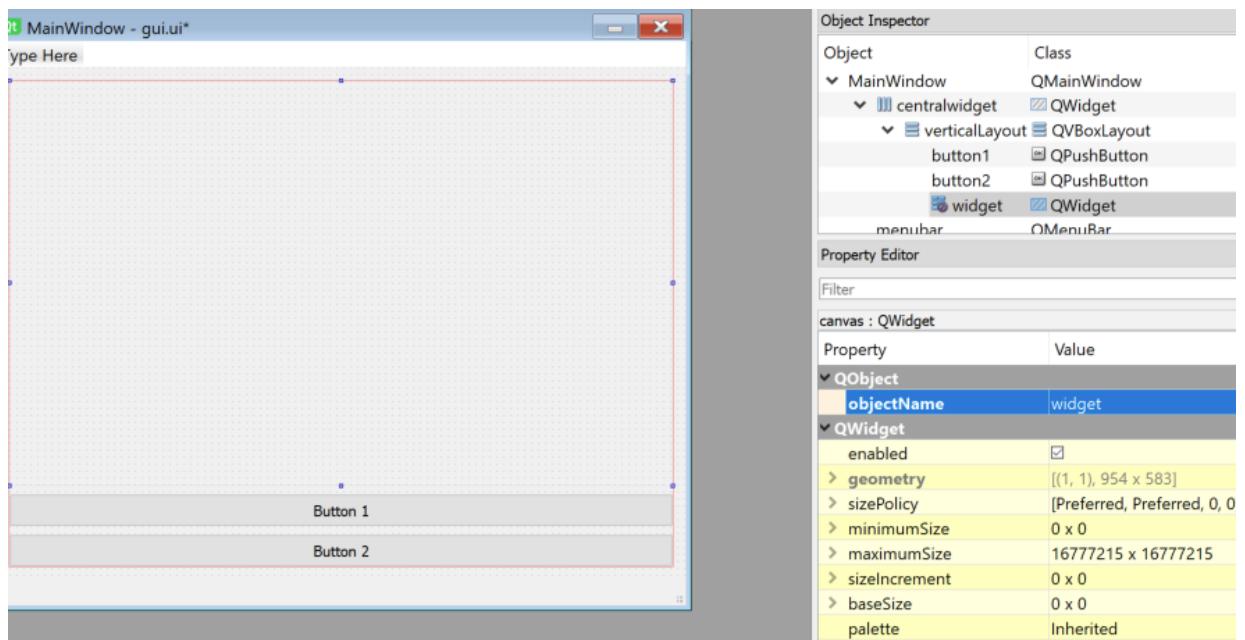
def plot2(self):
    plt.clf() # clear figure
    x = np.linspace(-np.pi, np.pi, 20)
    y = np.cos(x)
    plt.plot(x, y, "ko-")
    plt.axis("equal")

    self.canvas.draw()

app = QApplication([])
window = Window()
app.exec()

```

Selbstverständlich kann auch der Qt-Designer verwendet werden, dazu wird einfach ein neutrales QWidget eingesetzt und später im Code ersetzt. Man könnte man es einfach auch weglassen.



```

from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from PyQt5.uic import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as
FigureCanvas
import matplotlib.pyplot as plt
import numpy as np

```

```

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        loadUi("gui.ui", self)
        self.show()

        figure = plt.figure(figsize=(16, 9))
        self.canvas = FigureCanvas(figure)
        self.verticalLayout.removeWidget(self.widget)
        self.verticalLayout.insertWidget(0, self.canvas)

        self.button1.clicked.connect(self.button1click)
        self.button2.clicked.connect(self.button2click)

    def button1click(self):
        plt.clf()
        x = np.linspace(0, np.pi, 20)
        y = np.sin(x)
        plt.plot(x, y, "ro-")
        plt.axis("equal")

        self.canvas.draw()

    def button2click(self):
        plt.clf()
        x = np.linspace(0, np.pi, 20)
        y = np.cos(x)
        plt.plot(x, y, "ko-")
        plt.axis("equal")

        self.canvas.draw()

app = QApplication([])
window = MyWindow()
app.exec()

```

Wir können auch noch einen horizontalen Slider hinzufügen und den Wertebereich 10 bis 100 setzen (minimum, maximum). Der Objektname sei «slider»

```

self.slider.valueChanged.connect(self.slidermoved)

def slidermoved(self, value):
    plt.clf()
    x = np.linspace(0, np.pi, value)
    y = np.cos(x)
    plt.plot(x, y, "bo-")
    plt.axis("equal")
    self.canvas.draw()

```

8.3 Daten mit QLineEdit, QSlider oder QRadioButton einlesen

```
from PyQt5.QtWidgets import *
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
import matplotlib.pyplot as plt
import numpy as np

class Window(QMainWindow):
    def __init__(self):
        super().__init__()
        layout = QVBoxLayout()

        figure = plt.figure(figsize=(16,9))
        self.canvas = FigureCanvas(figure)

        self.x = QLineEdit("np.linspace(0, np.pi, 20)")
        self.y = QLineEdit("np.sin(x)")
        self.button = QPushButton("Plot")

        self.button.clicked.connect(self.plot)

        layout.addWidget(self.canvas)
        layout.addWidget(self.x)
        layout.addWidget(self.y)
        layout.addWidget(self.button)

        center = QWidget()
        center.setLayout(layout)

        self.setCentralWidget(center)
        self.show()

    def plot(self):
        plt.clf()
        try:
            x = eval(self.x.text())
            y = eval(self.y.text())
        except:
            QMessageBox.error(self, "Error", "Please specify x and y")

        plt.plot(x, y, "ro-")
        plt.axis("equal")

        self.canvas.draw()

app = QApplication([])
window = Window()
app.exec()
```

9 Projektionen und Vektordaten

9.1 Das Modul pyproj

Proj.4 ist eine Bibliothek zur Konversion von verschiedenen (kartografischen) Projektions-Systemen resp. Referenzsystemen. Die Bibliothek wurde ursprünglich von Gerald Everding (USGS) entwickelt, ist heute aber ein OSGeo Projekt, welches von Frank Warmerdam unterhalten wird. Proj.4 wird in der Programmiersprache C entwickelt, daraus wird ein Python-Modul abgeleitet.

(siehe auch: <https://www.swisstopo.admin.ch/de/karten-daten-online/calculation-services.html>)

Installation (Anaconda-Prompt)

```
conda install pyproj
```

Proj.4: <https://trac.osgeo.org/proj/>

pyproj (Python Modul): <https://code.google.com/p/pyproj/>

Wichtige Referenzsysteme:

Geografisches WGS84 (3D): EPSG: 4326

Web-Mapping Mercator (z.B. Google Maps): EPSG: 3857
(EPSG: 900913 nicht verwenden!)

CH1903/LV03: EPSG: 21781

CH1903r/LV95: EPSG: 2056

EPSG = European Petroleum Survey Group

Weitere Referenzsysteme respektive deren Definitionen können unter folgendem Link betrachtet werden: <http://epsg.io>.

9.1.1 Koordinatentransformation

Um eine Koordinatentransformation durchzuführen wird ein Transformer-Objekt erstellt. Dabei wird die Beschreibung des Referenzsystems am einfachsten als EPSG Code angegeben.

```
from pyproj import Transformer

transformer = Transformer.from_crs('EPSG:2056', 'EPSG:4326')

resultat = transformer.transform(2600000, 1200000)

print(resultat)
```

Die Beschreibung des Koordinatensystems kann auch ein WKT-String (OGC) oder über proj4 Parameter definiert werden. Wir verwenden im Moment der Einfachheit halber zunächst nur einen EPSG Code.

9.1.2 Koordinatentransformation eines CSV Datensatzes

Das amtliche Ortschaftenverzeichnis mit Postleitzahl und Perimeter kann unter folgendendem Link als csv heruntergeladen werden:

Direkter Link LV95: http://data.geo.admin.ch/ch.swisstopo-vd.ortschaftenverzeichnis_plz/PLZO_CSV_LV95.zip

```
import csv
from pyproj import Transformer

transformer = Transformer.from_crs('EPSG:2056', 'EPSG:4326')

file = open("PLZO_CSV_LV95.csv", encoding="cp1252")

reader = csv.DictReader(file, delimiter=";")
for row in reader:
    x = float(row["E"])
    y = float(row["N"])
    lat, lng = transformer.transform(x, y)

    Ortschaftsname = row["Ortschaftsname"]
    PLZ = row["PLZ"]
    Zusatzziffer = row["Zusatzziffer"]
    Gemeindenname = row["Gemeindenname"]
    Kanton = row["Kantonskürzel"]

    print(Ortschaftsname, PLZ, Zusatzziffer, Gemeindenname, Kanton, lng, lat)

file.close()
```

9.1.3 Shapely

Shapely ist eine Bibliothek für die Analyse und Bearbeitung geometrischer Objekte im kartesischen Raum. Shapely vereinfacht auch den Daten-Zugriff aus ogr. Die Dokumentation ist verfügbar unter: <http://toblerity.org/shapely/>

Die Installation über die Anaconda Prompt erfolgt mit:

```
conda install shapely
```

Mit WKT lassen sich geometrische Objekte relativ einfach darzustellen:

Typ	Beispiel
Point	POINT (30 10)
LineString	LINESTRING (30 10, 10 30, 40 40)
Polygon	POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))
	POLYGON ((35 10, 45 45, 15 40, 10 20, 35 10), (20 30, 35 35, 30 20, 20 30))
MultiPoint	MULTIPPOINT ((10 40), (40 30), (20 20), (30 10)) MULTIPPOINT (10 40, 40 30, 20 20, 30 10)
MultiLineString	MULTILINESTRING ((10 10, 20 20, 10 40), (40 40, 30 30, 40 20, 30 10))
MultiPolygon	MULTIPOLYGON (((30 20, 45 40, 10 40, 30 20)), ((15 5, 40 10, 10 20, 5 10, 15 5)))
	MULTIPOLYGON (((40 40, 20 45, 45 30, 40 40)), ((20 35, 10 30, 10 10, 30 5, 45 20, 20 35), (30 20, 20 15, 20 25, 30 20)))

Quelle: http://en.wikipedia.org/wiki/Well-known_text

in ogr kann eine Geometrie auch direkt aus einem WKT erstellt werden, dies funktioniert folgendermassen:

```
import ogr
```

Sehen wir uns zunächst folgendes Beispiel an:

Mit Shapely generieren wir ein Punkt.

```
from shapely.geometry import Point  
  
FHNW = Point([47.534792, 7.641529])  
FHNW.wkt
```

```
from shapely.geometry import Point  
  
BSL = Point([47.598829, 7.529104])  
BSL.wkt
```

Alternativ können wir anstelle von Point auch LineString, Polygon, MultiPolygon usw. verwenden.

Laden wir nun ein Multipolygon (Schweiz) als wkt-String:

Dazu verwenden wir das submodul «wkt»:

```
from shapely import wkt
```

Und öffnen das Multipolygon der Schweiz:

```
file = open("schweiz.wkt")  
ch = file.read()  
file.close()  
  
schweiz = wkt.loads(ch)
```

Mit Matplotlib können wir die aussenhülle des Polygons darstellen. Bei Multipolygons müssen wir die einzelnen Polygone mit einer for-Schleife iterieren.

```
import matplotlib.pyplot as plt  
  
for geometry in schweiz.geoms:  
    x,y = geometry.exterior.xy  
    plt.plot(x,y)  
  
plt.show()
```

9.1.4 Binäre Operationen

Eine weitere wichtige geometrische Operation ist die Beziehung zwischen zwei Objekten. Shapely stellt unter anderem folgende binäre Operationen zur Verfügung:

contains	Returns True if the interior of the object intersects the interior of the other but does not contain it, and the dimension of the intersection is less than the dimension of the one or the other.
intersects	Returns True if the boundary and interior of the object intersect in any way with those of the other.
within	Returns True if the object's boundary and interior intersect only with the interior of the other (not its boundary or exterior).
touches	Returns True if the objects have at least one point in common and their interiors do not intersect with any part of the other.
crosses	Returns True if the interior of the object intersects the interior of the other but does not contain it, and the dimension of the intersection is less than the dimension of the one or the other.
equals	Returns True if the set-theoretic boundary, interior, and exterior of the object coincide with those of the other.

(mehr unter: <http://toblerity.org/shapely/manual.html#binary-predicates>)

Beispiel: Wir testen, ob die Punkte innerhalb der Schweiz sind:

```
BSL.within(schweiz)
```

```
FHNW.within(schweiz)
```

9.1.5 Mehrere Polygone und Operationen

Mit Shapely können wir auch Mengenoperationen mit geometrischen Objekten ausführen.

Wir erstellen zwei Polygone:

```
wkt1 = "POLYGON (( -5 -5, 5 -5, 5 5, -5 5, -5 -5))"
wkt2 = "POLYGON ((1 -1, 4 -1, 4 1, 1 1, 1 4, -1 4, -1 1, -4 1, -4 -1, -1 -1, -1 -4, 1 -4, 1 -1))"
quadrat = shapely.wkt.loads(wkt1)
kreuz = shapely.wkt.loads(wkt2)
```

Hinweis: Interiors und exteriors sind immer im entgegengesetzten Uhrzeigersinn!

```
import matplotlib.pyplot as plt

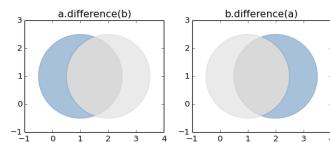
x,y = quadrat.exterior.xy
plt.plot(x,y, 'b-')

x,y = kreuz.exterior.xy
plt.plot(x,y, 'r-')

plt.show()
```

Differenz

```
diff = shape1.difference(shape2)
```

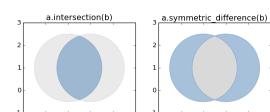


Schnittmenge (Intersection)

```
inter = shape1.intersection(shape2)
```

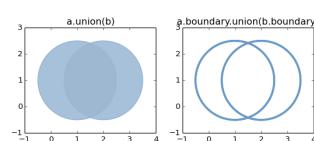
Symmetrische Differenz

```
sdiff = shape1.symmetric_difference(shape2)
```



Vereinigung (Union)

```
uni = shape1.union(shape2)
```



9.1.6 Delaunay Triangulierung

```
from shapely.ops import triangulate
import shapely.wkt
import numpy as np
wkt = "POLYGON ((1 -1, 4 -1, 4 1, 1 1, 1 4, -1 4, -1 1, -4 1, -4 -1, -1 -1, -1 -4, 1 -4, 1 -1))"
data = shapely.wkt.loads(wkt)
triangles = triangulate(data)
```

```
for tri in triangles:
    x, y = tri.exterior.coords.xy
    plt.plot(x,y)
plt.show()
```

10 Einführung QGIS

Dieses Kapitel basiert unter anderem auf dem „QGIS User Guide“ und dem „QGIS Training Manual“.

10.1 Was ist QGIS ?

Ein Geoinformationssystem resp. Geographisches Informationsystem (GIS) ist ein Informationssystem (IS), mit dem raumbezogene Daten digital **erfasst, bearbeitet, organisiert, analysiert und präsentiert** werden.

Heute gibt es eine Vielzahl an GIS von verschiedenen Herstellern, wie zum Beispiel ArcGIS von ESRI oder GeoMedia von Intergraph. Es gibt auch einige Open Source GIS, wie zum Beispiel QGIS. Open Source bedeutet, dass der Source Code der gesamten Applikation frei verfügbar ist. Das wichtigste dabei ist nicht, dass dieses GIS gratis ist, sondern dass dieses GIS frei weiterentwickelt werden kann – unabhängig von einem einzigen kommerziellen Anbieter. So ist QGIS heute auch in über 30 Sprachen verfügbar, dank freiwilligen Übersetzern. Softwareentwickler können Erweiterungen schreiben – dabei kann C++ oder Python verwendet werden.

Warum sollte QGIS verwendet werden ? Einige der Gründe sind:

- Gratis erhältlich – keine Kosten
- Die Software ist frei – Wenn neue Funktionalität gewünscht wird, kann diese entweder selbst implementiert werden oder man sponsert die Entwicklung dieser Funktionalität.
- QGIS ist wird immer weiterentwickelt
- Eine Vielzahl von Dokumentationen ist verfügbar. Es gibt auch eine grosse Community – man kann auch die Entwickler oder andere QGIS Nutzende direkt fragen.
- QGIS gibt es für Windows, MacOS und Linux.

QGIS unterstützt gängige Vektorformate und Rasterdaten wie z.B. Shapefiles oder GeoTIFF. Auch räumliche Datenbanken (wie z.B. PostGIS) werden unterstützt. Auch OGC Services wie z.B. WMS oder WFS Dienste können eingebunden werden.

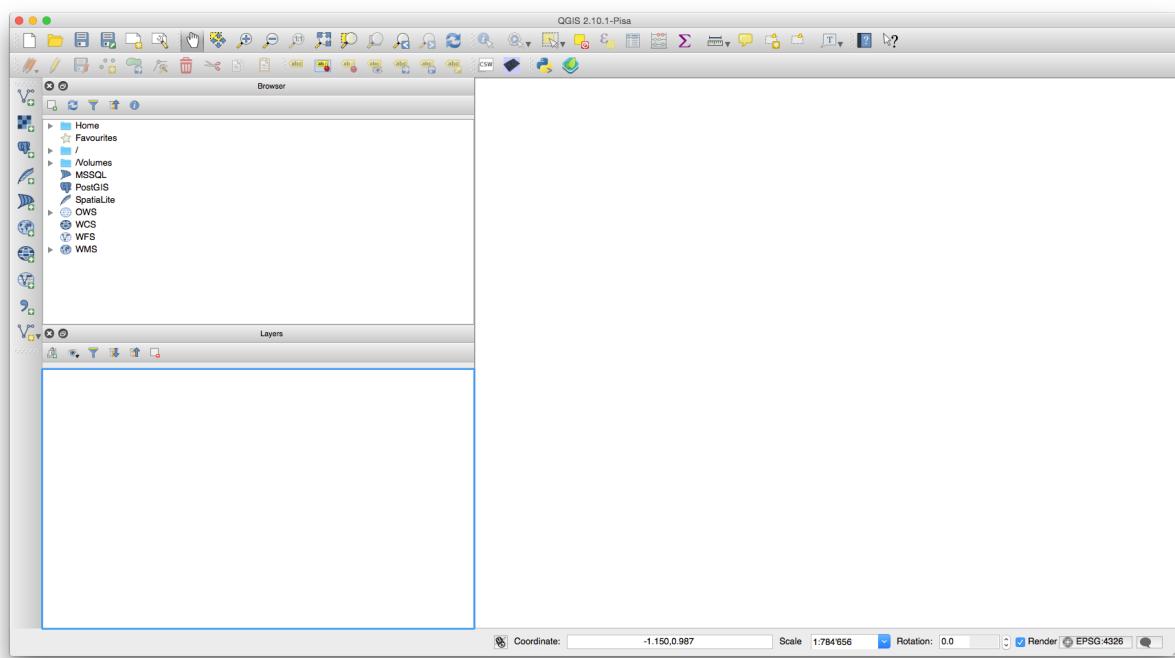
Falls irgendeine wichtige Funktion fehlt, so kann diese also Plugin oder im Core implementiert werden!

Die erste Version von QGIS wurde im Juli 2002 veröffentlicht. Die Version 1.0.0 wurde im Januar 2009 freigegeben und die Version 2.0.0 im September 2013.

QGIS ist unter <http://www.qgis.org/> erhältlich.

Für die ersten Versuche verwenden wir einige Demo-Daten, welche auf Moodle erhältlich sind ([demo.zip](#)).

Wird QGIS gestartet sehen wir folgendes Fenster.



10.2 Arbeiten mit Vektordaten

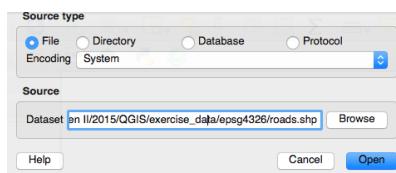
QGIS verwendet die OGR-Bibliothek (<http://www.gdal.org>) um Vektordatenformate zu lesen und schreiben. Unter anderem werden ESRI Shapefiles, MapInfo, MicroStation Formate, AutoCAD DXF, PostGIS, SpatiaLite, Oracle Spatial, MSSQL Spatial Datenbanken.

Auch PostgreSQL wird von QGIS unterstützt. Das OpenStreetMap Vektorformat wird auch direkt unterstützt.

In QGIS können Vektordaten auch aus zip-Archiven (und gzip-Archiven) gelesen werden, dies ist v.a. bei Shapefiles sehr nützlich, da diese aus verschiedenen Files bestehen.

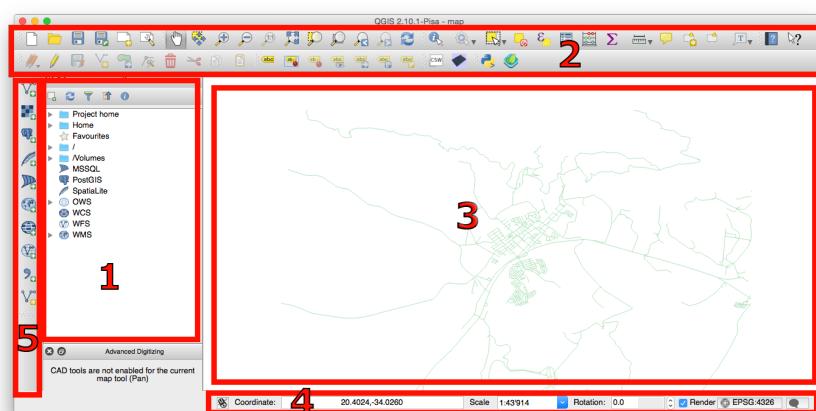
Wir sehen eine neue leere Karte. Auf der linken Seite befindet sich das Symbol Mit diesem können wir einen Vektor-Layer hinzufügen. Wir wählen das folgende File:

`exercise_data/epsg4326/roads.shp`



Mit dem Symbol kann nun das Projekt gespeichert werden. Wir speichern das Projekt unter `map.qgs`

Die Benutzeroberfläche ist in QGIS folgendermassen aufgebaut:

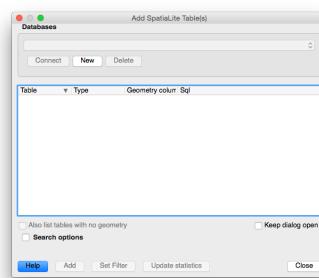


- 1: Layer-Liste / Browser
- 2: Toolbars
- 3: Map Canvas
- 4: Status Bar
- 5: Side Toolbar

Zunächst sehen wir uns alle Attribute des Layers an. Dies geschieht durch Klick auf den Layer („roads“) und dann auf das Symbol . Nun wird eine Tabelle mit allen Attributen angezeigt:

	osm_id	v	name	highway	waterway	aeraleyay	barrier	man_made	other_tags
109	67535871	NULL		unclassified	NULL	NULL	NULL	NULL	'surface=>unpaved'
108	60380056	Andrew Whyte Street		residential	NULL	NULL	NULL	NULL	NULL
107	79115510	NULL		unclassified	NULL	NULL	NULL	NULL	NULL
106	79115508	NULL		unclassified	NULL	NULL	NULL	NULL	NULL
105	75030053	NULL		primary	NULL	NULL	NULL	NULL	'width=>760"
104	75030042	NULL		primary	NULL	NULL	NULL	NULL	'bridge=>yes','type=>"1","...
103	75030039	NULL		trunk	NULL	NULL	NULL	NULL	'bridge=>yes','type=>"1","m...
102	75030030	NULL		trunk	NULL	NULL	NULL	NULL	'maxspeed=>120','oneway=>...
101	66245666	NULL		unclassified	NULL	NULL	NULL	NULL	NULL
100	66240041	NULL		path	NULL	NULL	NULL	NULL	NULL
99	66240002	NULL		path	NULL	NULL	NULL	NULL	NULL
98	66050291	Kanon Street		residential	NULL	NULL	NULL	NULL	NULL
97	66050297	Tinney Street		residential	NULL	NULL	NULL	NULL	NULL
96	66050295	Van Oudshoorn Road		residential	NULL	NULL	NULL	NULL	NULL
95	65819191	Heemraad Street		residential	NULL	NULL	NULL	NULL	'oneway=>-1"
94	59408087	Jansen Street		residential	NULL	NULL	NULL	NULL	NULL
93	59408085	Baker		residential	NULL	NULL	NULL	NULL	'bicycle=>yes','cycleway=>...
92	59408085	Van Imhoff		residential	NULL	NULL	NULL	NULL	'bicycle=>yes','cycleway=>...

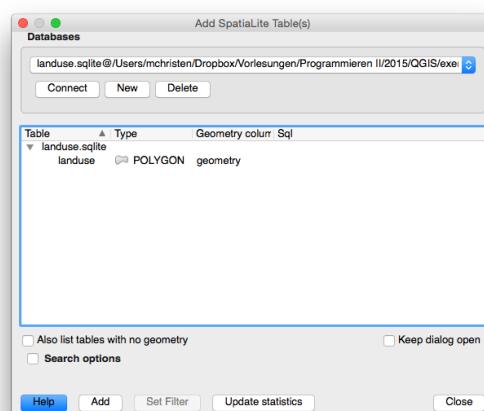
Nun laden wir Daten aus einer Datenbank. Wir fügen Vektordaten aus einer „SpatiaLite“ Datenbank hinzu. Dazu klicken wir auf das Icon .



Im Dialog wird dann mit „New“ eine neue Datenbankverbindung aufgebaut. In unserem Fall ist dies folgendes File:

```
exercise_data/epsg4326/landuse.sqlite
```

Danach wird die Datenbank mittels dem „Connect“-Button verbunden:



Danach wird „landuse“ (Landnutzung) gewählt und mittels „Add“ hinzugefügt. Die Karte sollte danach etwa so aussehen:



Wir sehen, dass der „landuse“ Layer den „roads“ Layer überdeckt. Wir können dies im „Layers“ Tool einfach mittels Drag & Drop anpassen, so dass „landuse“ der erste Layer ist.

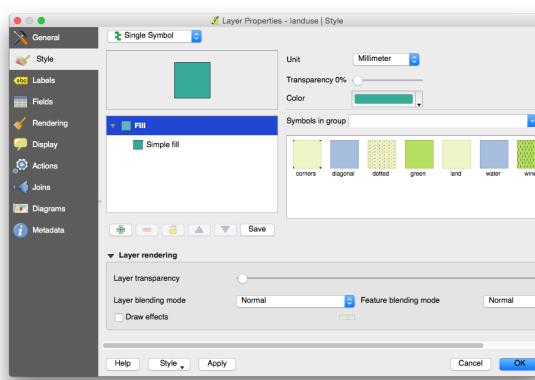
Nun können wir auch noch weitere Vektor-Layer hinzufügen:

```
places.shp, buildings.shp, rivers.shp, water.shp
```

Die Reihenfolge der Layer sollte dann: landuse, water, rivers, roads, buildings, places (von unten nach oben) sein. Unsere Karte sieht dann etwa so aus:



Diese Karte ist natürlich alles andere als schön, aber wir können dies über die Layereigenschaften anpassen. Zunächst machen wir einen Rechtsklick auf den „landuse“ Layer und wählen „Eigenschaften“. Dann erscheint dieses Fenster:



Danach können wir „Einfache Füllung“ wählen und die Farbe beispielsweise auf grau setzen.

Nun öffnen wir die Eigenschaften des „landuse“ Layers wieder und setzen den „Randstil“ auf „Kein Stift“.

Nun ändern wir die Farben der anderen Layers, und zwar so dass:

- Der waters-Layer dunkelblau wird
- Der rivers-Layer hellblau wird

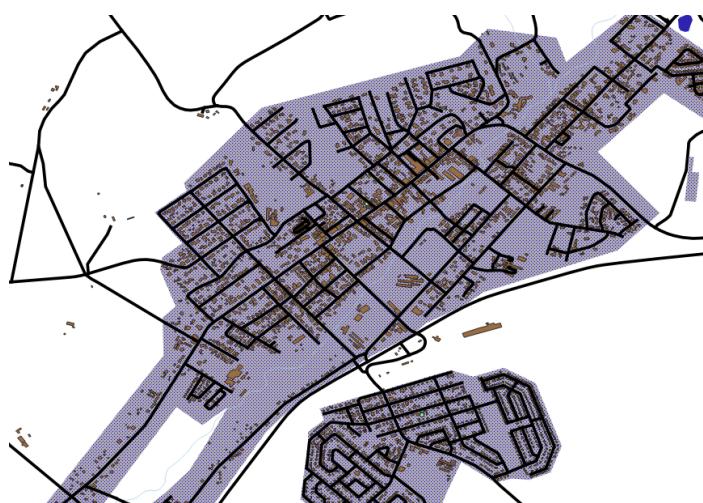
Es ist auch möglich gewisse Layer anhand des Massstabs darzustellen. Beispielsweise macht es wenig Sinn alle Gebäude schon von weit entfernt darzustellen – auch aus Performancegründen.

Dazu öffnen wir die Eigenschaften des „buildings“ Layer. Im Tab „Allgemein“ kann „Massstabsabhängige Sichtbarkeit“ aktiviert werden. Dort setzen wir Minimum auf 1:10000 und das Maximum auf 1:1.

Nun gehen wir zurück auf die Eigenschaften des landuse-Layers und fügen mit  einen Symbollayer hinzu und setzen dabei folgendes:

- Den Randstil auf „Kein Stift“.
- Den Füllstil auf „Dense 6“.

Nun ändern wir den style des roads-Layer. Wir fügen einen neuen Symbollayer hinzu, setzen die Basis Stiftbreite auf 0.3 und den Stiftstil auf „Gestrichelte Linie“. Den neuen Symbollayer setzen wir auf Strichdicke 1.3. Das Resultat sieht dann etwa so aus:



Dies ist allerdings nicht das gewünschte Resultat, da wir die Reihenfolge der Symbollayer umdrehen müssen!

Dies geschieht mittels Drag & Drop.
Die Strassentypen können auch klassifiziert werden.

10.3 Arbeiten mit Rasterdaten

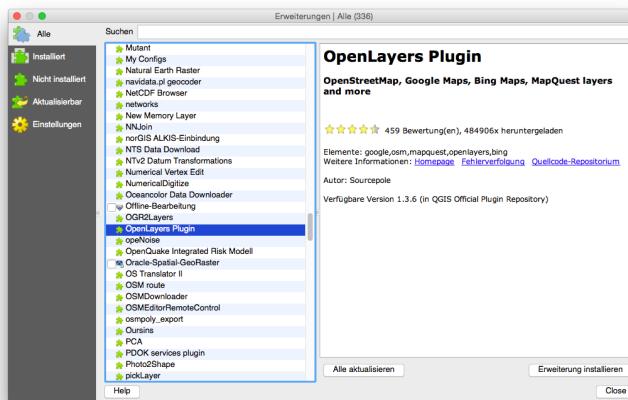
QGIS verwendet die GDAL Bibliothek (<http://www.gdal.org>) um Rasterdatenformate zu lesen und schreiben. Es werden weit mehr als 100 Rasterformate unterstützt, wie z.B. Arc/Info Binary Grid, Arc/Info ASCII Grid, GeoTIFF, Erdas Imagine.

Rasterdaten werden ganz einfach mit dem Symbol  hinzugefügt. Wir werden Rasterdaten nur am Rande betrachten.

10.4 Plugins

In QGIS können wir eine Vielzahl an Plugins installieren. Wir installieren nun das „OpenLayers Plugin“. Dies geschieht über das Menu „Erweiterungen/Erweiterungen verwalten und installieren“. Wir werden hier nur zwei Erweiterungen ansehen.

10.4.1 OpenLayers Plugin



Im „Web“ Menu ist nun der Eintrag „OpenLayers Plugin“ zu sehen. Dort wählen wir „OpenStreetMap/OpenStreetMap“ aus. Danach sehen wir einen neuen Layer „OpenStreetMap“ und die Karte aus OpenStreetMap. Auch das Referenzsystem (Bezugssystem) hat geändert. Wir sehen nun „EPSG:3857“ in der Statusbar.

10.5 Programmieren mit QGIS

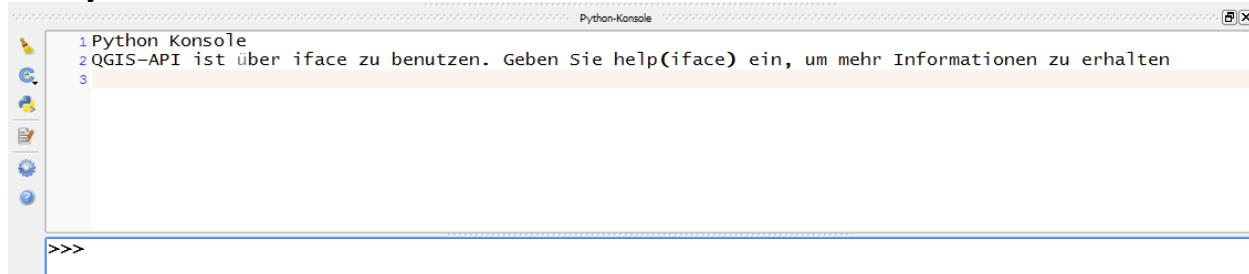
In diesem Kapitel lernen wir innerhalb von QGIS zu programmieren. Als Testdaten verwenden wir das GeoPackage von natural-earth:
<https://www.naturalearthdata.com/downloads/>

10.5.1 Verwendung der Python Konsole

Soll etwas innerhalb von QGIS entwickelt werden, so kann dazu die Python Konsole verwendet werden. Meistens sind es kleine Skripts die über diesen Weg ausgeführt werden.

Die Python Konsole kann über das Menu „Erweiterungen/Python Konsole“ geöffnet werden oder auch ganz einfach mit Klick auf das Python-Konsole Icon.

Die Python Konsole erscheint:



Wenn die Konsole geöffnet wird, so wird automatisch folgender Python Code ausgeführt:

```
from qgis.core import *
import qgis.utils
```

Das heisst Teile des qgis Moduls werden automatisch importiert. Dies ist nützlich, da es mühsam wäre dies jedesmal immer wieder neu einzutippen.

10.5.2 Projekte Laden und Speichern

Wir starten QGIS und fügen aus dem Natural Earth GeoPackage den Layer ne_50m_airports zu.

In der Python Konsole schreiben wir nun:

```
project = QgsProject.instance()
project.write("C:/data/QGIS_Projekt/projekt.qgs")
```

Nun könnten wir änderungen Vornehmen und das Projekt über Python speichern:

```
project.write()
```

oder um das Projekt unter einem anderen Namen zu speichern rufen wir auf:

```
project.write("C:/data/QGIS_Projekt/neues_projekt.qgs")
```

Die Methoden `read()` und `write()` geben übrigens jeweils True oder False zurück, falls überprüft werden soll, ob das Laden resp. Speichern funktioniert hat.

Um ein Projekt zu laden rufen wir einfach folgendes auf:

```
project.read("C:/data/QGIS_Projekt/projekt.qgs")
```

10.5.3Arbeiten mit Layern

Um alle Layer-Objekte zu erhalten kann das über das `mapCanvas()` Objekt geschehen. Die MapCanvas Klasse ist dafür verantwortlich alle GIS-Objekte darzustellen. Wir grifen auf den MapCanvas und die Layer folgendermassen zu:

```
canvas = iface.mapCanvas()
layers = canvas.layers()
```

Um die Namen aller Layers auszugeben können wir folgendes definieren:

```
for l in layers:
    print(l.name())
```

Um zu überprüfen, ob es sich wirklich um Vektor-Layer handelt, können wir folgendes schreiben:

```
for l in layers:
    if l.type() == QgsMapLayer.VectorLayer:
        print(l.name())
```

Im Falle eines Rasters, wäre der Layer-Typ dann einfach
`QgsMapLayer.RasterLayer`

Die Dokumentation, inklusive dem Klassendiagramm zum Map-Layer finden wir auf:
<http://qgis.org/api/classQgsMapLayer.html>

Diese Dokumentation ist allerdings für C++, kann aber leicht auch für Python verstanden werden.

Sehen wir uns mal die Features des ersten Layers an („ne_10m_airports“):

```
airports = layers[0]
features = airports.getFeatures():
for f in features:
    print(f.id())
```

Der Einfachheit halber betrachten wir mal das erste Feature in der Liste:

```
f = list(airports.getFeatures())[0]

id = f.id()                      # die ID des Features erhalten
geometry = f.geometry()          # auf die Geometrie des Features zugreifen
```

Nun können wir überprüfen, um was für eine Geometrie es sich handelt. Bei den Flughäfen sind es jedoch alles Punkte (
<https://qgis.org/pyqgis/3.0/core/Wkb/QgsWkbTypes.html>)

```
if geometry.type() == QgsWkbTypes.PointGeometry:
    print("Typ ist Punkt")
elif geometry.type() == QgsWkbTypes.LineString:
    print("Typ ist Linie")
elif geometry.type() == QgsWkbTypes.Polygon:
    print("Typ ist Polygon")
```

Wir können auch von jedem Feature auf die Attribute zugreifen, dies geschieht folgendermassen:

```
attrib = f.attributes()
print(attrib)
```

Attrib ist eine Liste mit sämtlichen Attributen des Features.

Um die Definition der Attribute zu erhalten können wir im Layer auf die „pendingFields“ zugreifen:

```
for field in airports.fields():
    print(field.name() + ", " + field.typeName())
```

Also das Feld 3 wäre in unserem Beispiel der Name des Flughafens.

Die Geometrie erhalten wir folgendermassen:

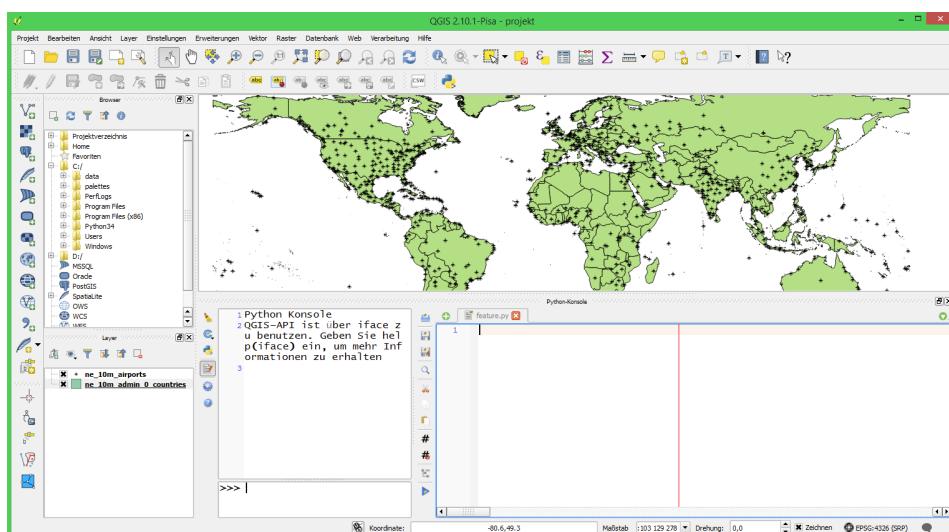
```
wkt = geometry.asWkt()
```

```
point = geometry.asPoint()
print(point.x(), point.y())
```

Nun müssen wir zurück zur QGIS Oberfläche: In QGIS haben wir einen aktiven Layer – das ist der Layer der gerade gewählt ist. Dies geschieht mit der `activeLayer` Methode. Falls kein Layer gewählt ist, so wird `None` zurückgegeben.

```
layer = iface.activeLayer()
```

Wir fügen den `admin0` layer aus dem GeoPackage hinzu. Dann selektieren wir im GUI den Layer „`ne_10m_admin_0_countries`“. Wir wählen in der Python Konsole nun den Button „Editor Anzeigen“ und speichern das File als „`feature.py`“ ab:



Nun schreiben wir ein kleines Programm, bei dem die Schweiz rot eingefärbt werden soll. Zunächst müssen wir die „`SelectionColor`“ des Map Canvas auf rot setzen. Dies geschieht folgendermassen
(<https://qgis.org/pyqgis/3.0/core/Vector/QgsVectorLayer.html>):

```
from PyQt5.QtGui import QColor

canvas = iface.mapCanvas()
canvas.setSelectionColor(QColor("red"))
```

Dann können wir mit der Methode `setSelectedFeatures` der Layer-Klasse die Features über dessen id wählen:

```
layer = iface.activeLayer()
layer.select(93)
layer.select(92)
layer.select(51)

layer.deselect(51)
```

10.5.4 Einen Vektor Layer erzeugen

Die Klasse „QgsVectorLayer“ erzeugt einen Vector Layer. Standardmäßig können in QGIS mindestens folgende Layer erstellt werden:

- Shapefile-Layer – ein ESRI Shapefile
- SpatiaLite-Layer – ein Datenbanklayer mit SpatiaLite
- Temporärlayer – ein Layer welcher im Speicher (temporär) gehalten wird

Zunächst importieren wir alle Klassen von PyQt5.QtCore, da wir auf Funktionalität von Qt zurückgreifen müssen:

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

Der einfachste Vektorlayer ist ein Temporärlayer. Dieser wird z.B. folgendermassen erstellt:

```
layer = QgsVectorLayer("Point?crs=epsg:4326",
                      "temporary_points",
                      "memory")
```

Jeder Layer verfügt um einen sogenannten „Data-Provider“, welcher über gewisse Eigenschaften verfügt, welche formatspezifisch sind. So hat z.B: ein Shapefile andere Fähigkeiten als ein PostGIS Layer. Der Data-Provider kann über die Layer-Klasse erhalten werden:

```
provider = layer.dataProvider()
```

Die Fähigkeiten des Data-Providers können über die „Capabilities“ abgefragt werden:

```
caps = provider.capabilities()

if caps & QgsVectorDataProvider.AddFeatures:
    print(u"Juhuuu!! Es können neue Features hinzugefügt werden!")
```

Es können u.a. folgende Fähigkeiten des Data-Providers abgefragt werden:

QGsVectorDataProvider.

AddFeatures
DeleteFeatures

```
ChangeAttributeValues
AddAttributes
DeleteAttributes
SaveAsShapefile
CreateSpatialIndex
SelectAtId
ChangeGeometries
SelectGeometryAtId
SelectEncoding
```

Nun muss der Vektor-Layer „editierbar“ gemacht werden, dies geschieht mit:

```
layer.startEditing()
```

Um die Felder der Attribute zu definieren wird über den Data-Provider die Methode „addAttribute“ verwendet. Als Parameter wird eine Liste von QgsField verwendet. Ein QgsField besteht aus einem Namen und einem Datentyp (QVariant.String, QVariant.Int, QVariant.Double).

```
provider.addAttribute([QgsField("name", QVariant.String),
                      QgsField("gewichtung", QVariant.Int)])
```

Nun können die Features erstellt und hinzugefügt werden. Ein Feature wird mit der Klasse QgsFeatures erstellt. Dem Feature kann dann die Geometrie und die Attribute übergeben werden und über den Data-Provider hinzugefügt werden:

```
feature = QgsFeature()
feature.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(7.6386, 47.5336)))
feature.setAttributes(["Muttenz", 5])
provider.addFeatures([feature])
```

Um das editieren zu beenden wird beim Layer die Methode „commitChanges“ aufgerufen:

```
layer.commitChanges()
```

Nun muss noch der Umfang des Layers berechnet werden, respektive angepasst werden. Dies geht mit der Methode „updateExtents()“ des Layers.

```
layer.updateExtents()
```

Um den Layer in QGIS in der Legende hinzuzufügen, wird noch folgendes aufgerufen:

```
project = QgsProject.instance()
project.addMapLayer(layer)
```

Der Temporärlayer kann nun gespeichert werden. Ein Shapefile kann folgendermassen erzeugt werden:

```
QgsVectorFileWriter.writeAsVectorFormat(layer,
                                         "myshape.shp",
```

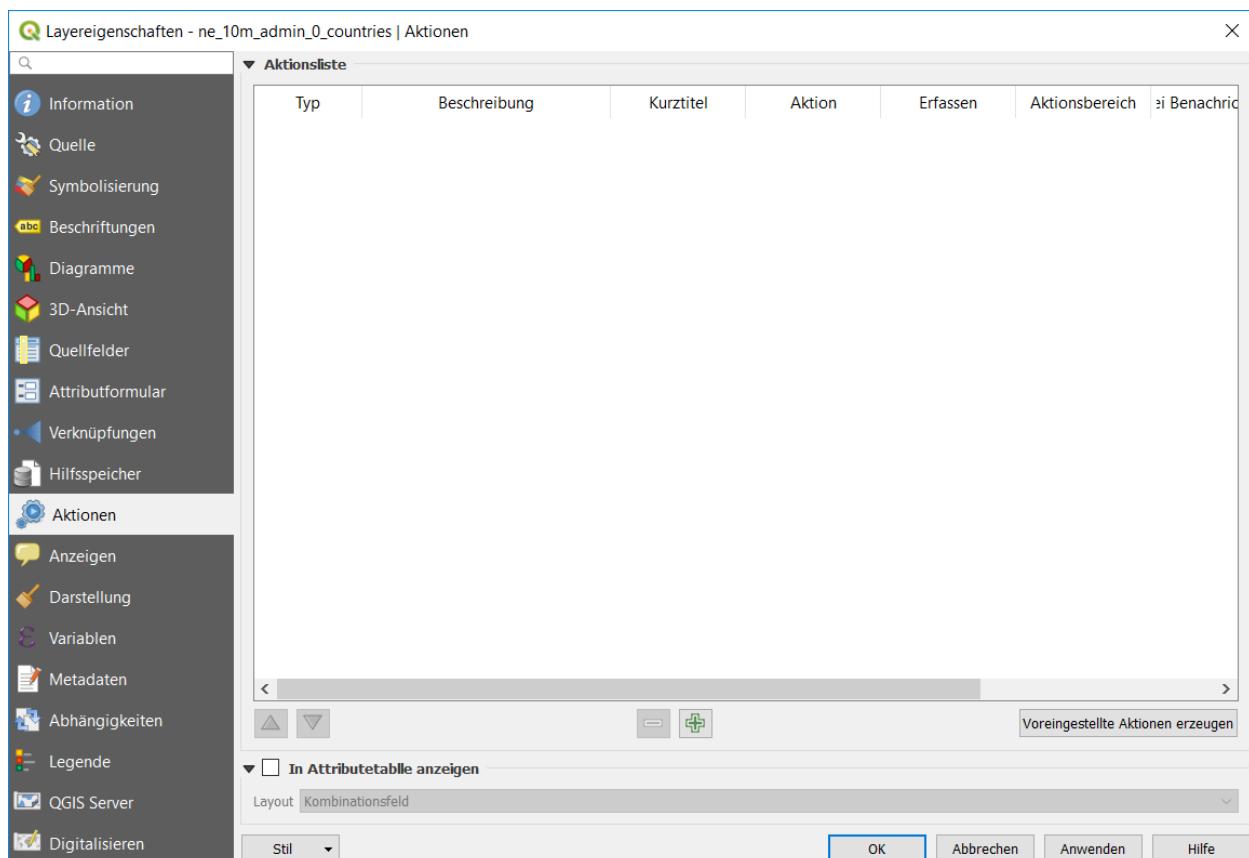
```
provider.encoding(),  
provider.crs()
```

10.6 Python Actions in QGIS

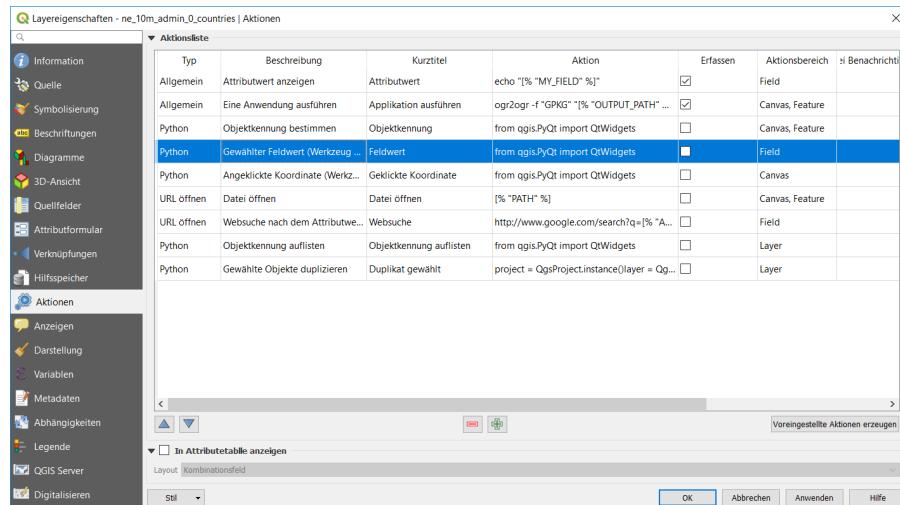
In diesem Kapitel lernen wir wie wir QGIS neue Funktionalität hinzufügen mit Python & Actions.

10.6.1 Erstellen einer Python Action

Wir erstellen ein neues QGIS Projekt und fügen den Layer «ne_110m_admin_0_countries» aus dem GeoPackage von Natural Earth hinzu. Mit einem Rechtsklick auf den Layer wählen wir im Popup-Menu «Eigenschaften...». Dort wählen wir den «Aktoinen» Tab:



Dort klicken wir auf den Button «Voreingestellt Aktionen erzeugen»

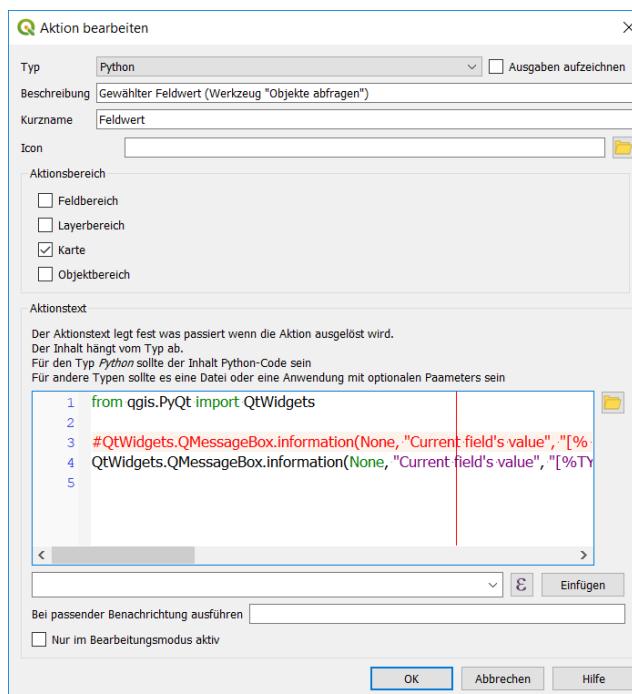


Wir wählen «Gewählter Feldwert» mit einem Doppelklick. Dort ändern wir den code zu folgendem:

```
from PyQt5 import QtWidgets

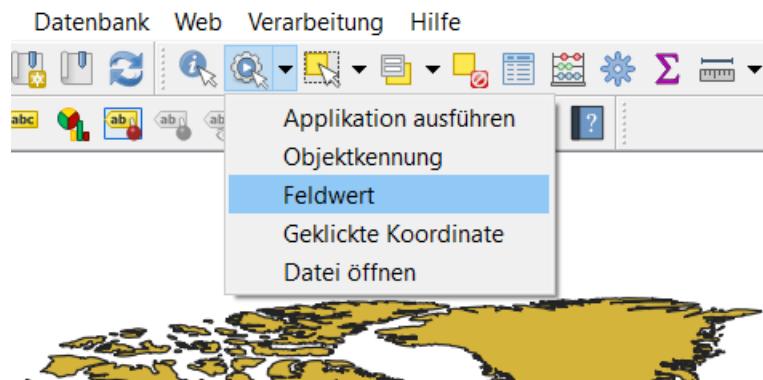
QtWidgets.QMessageBox.information(None, "Current field's value", "[%TYPE%]")
```

Wir ändern den Aktionsbereich auf «Karte» und deselektieren «Feldbereich».



Wir bestätigen 2x mit Ok.

Nun wählen wir die Aktion «Feldwert»



Wir klicken auf die Länder und das Python-Script wird jeweils ausgeführt!

10.6.2 Eine neue Python Action erstellen

Wir estellen nun eine komplett neue Action, welche jeweils die Geometrie in die Python Konsole ausgibt. Dieses Beispiel zeigt, wie wir Layer Actions mit den QGIS Python Funktionalität kombinieren lassen.

```
from qgis.core import *
from qgis.utils import *

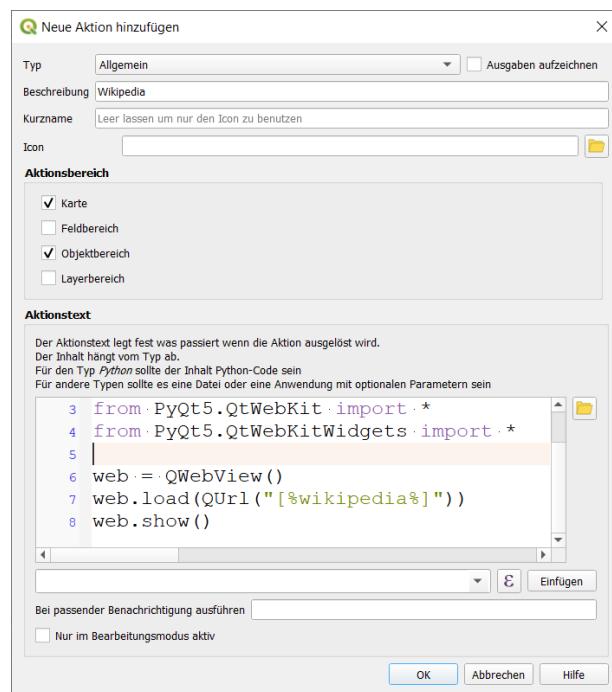
clicked = "[% $id %]"

if clicked != "":
    clickedid = int(clicked)
    print(clicked, type(clicked))
    layer = iface.activeLayer()
    features = layer.getFeatures()
    for feature in features:
        if feature.id() == clickedid:
            print("found!")
            print(feature.geometry().asWkt())
```

10.6.3 Beispiel: URL öffnen

Im Airport Datensatz ist eine Spalte «wikipedia». Diese URL soll im Webbrower geöffnet werden.

Dazu erstellen wie eine Python Action:



```

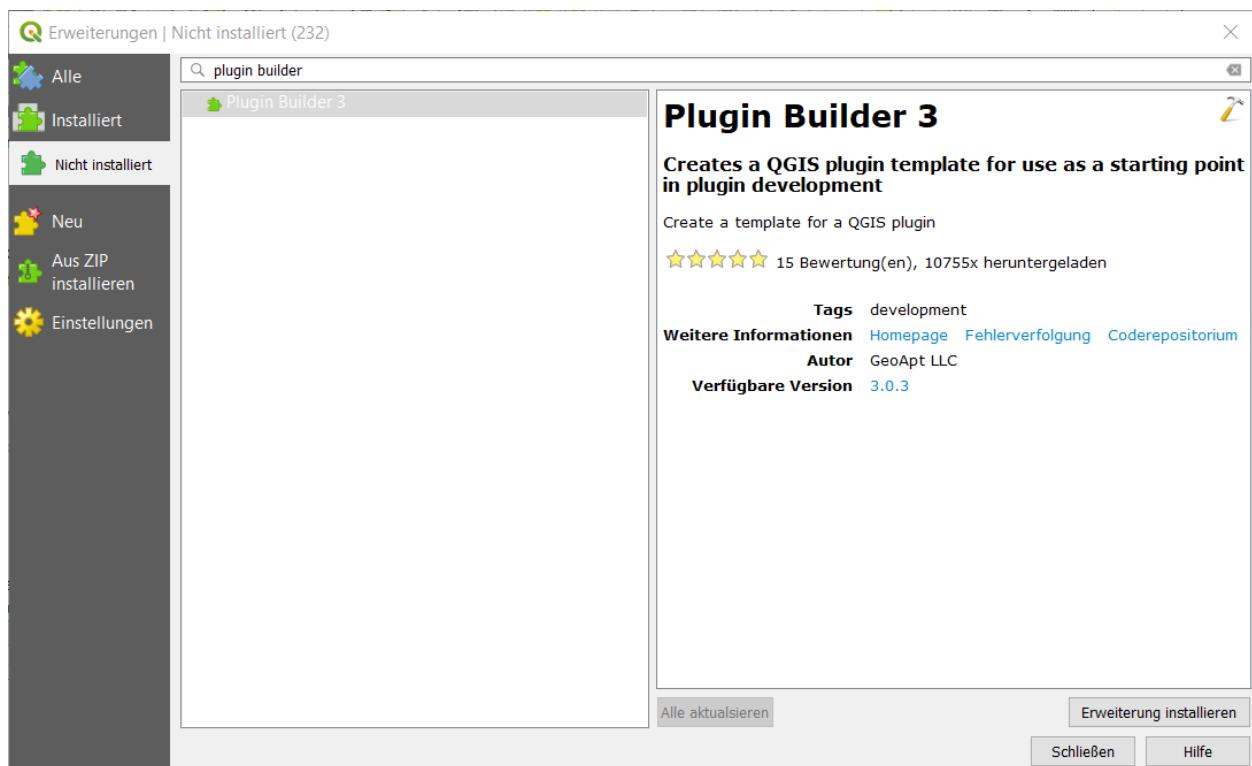
from PyQt5 import QtCore
from PyQt5.QtCore import *
from PyQt5.QtWebKit import *
from PyQt5.QtWebKitWidgets import *
web = QWebView()
web.load(QUrl("[%wikipedia%"]))
web.show()

```

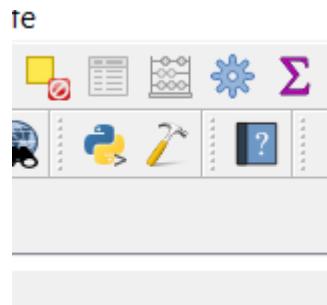
10.7 Erstellen von QGIS Plugins

Wir haben nun gesehen wie Python-Skripte in QGIS verwendet werden können. Nun werden wir nun lernen wie **Plugins** erstellt werden.

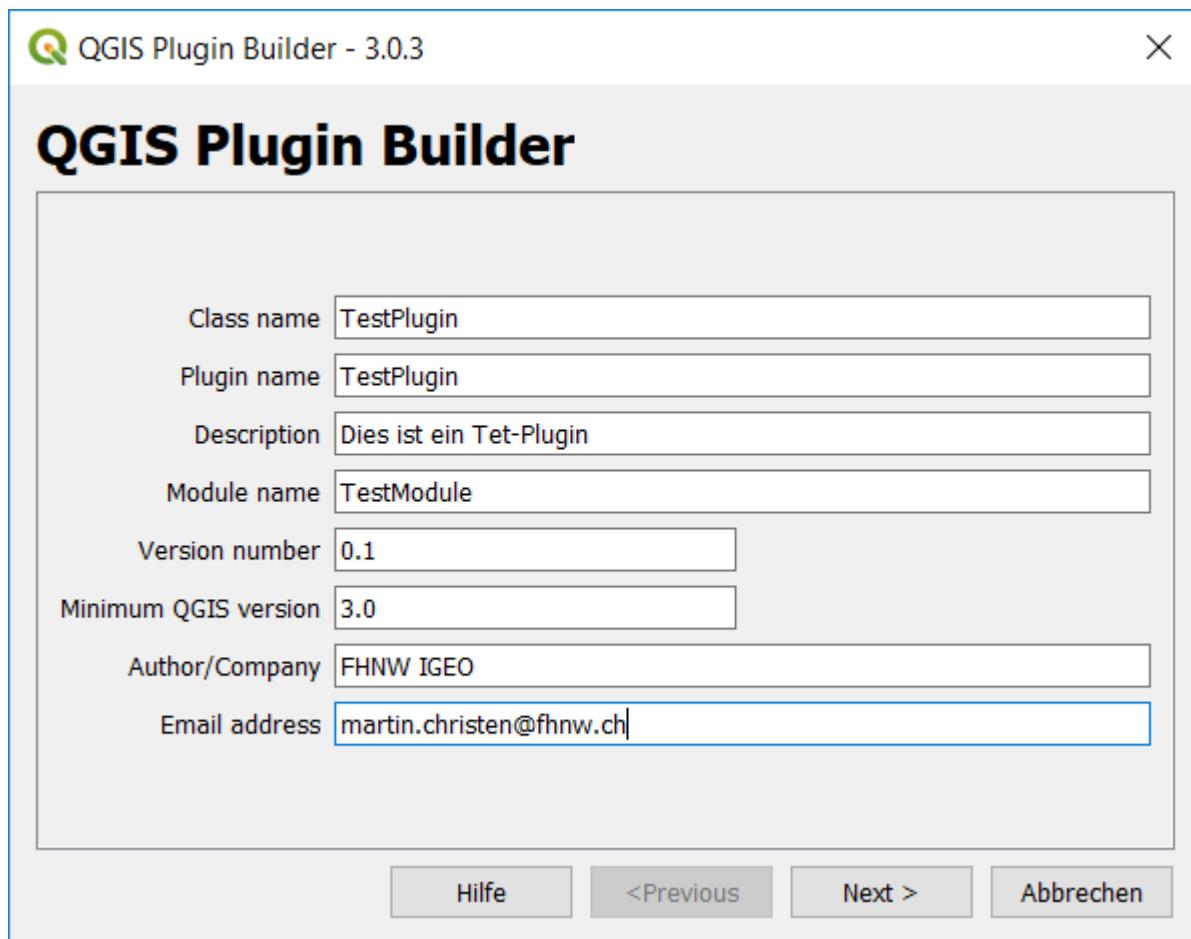
Es gibt in QGIS ein Plugin, welches die Erstellung von neuen Plugins erleichtert. Dieses Plugin heisse «Plugin Builder 3» und kann ganz einfach installiert werden:



Nach erfolgter Installation kann der Plugin Builder (Hammer-Symbol) gestartet werden:

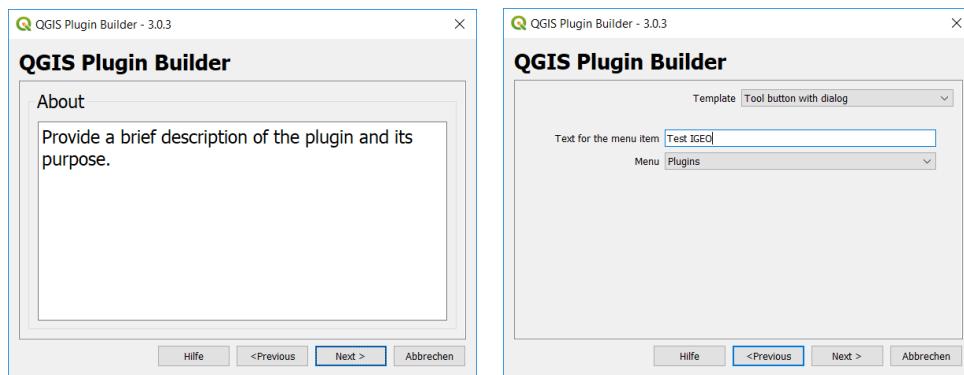


Nach dem Start des Plugin Builders wird ein Fenster geöffnet, in dem mehrere Informationen abgefragt werden:

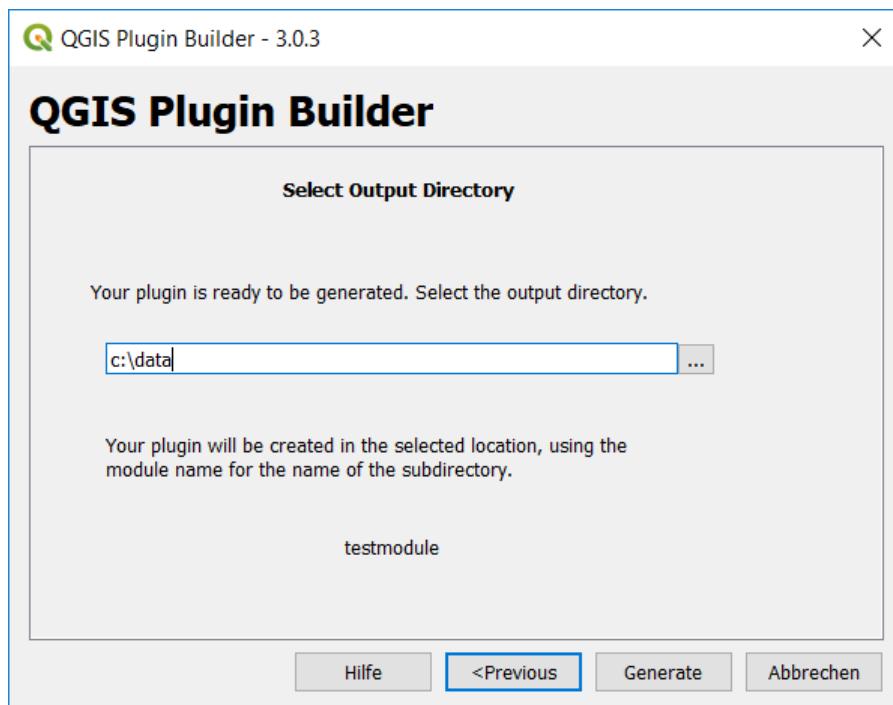


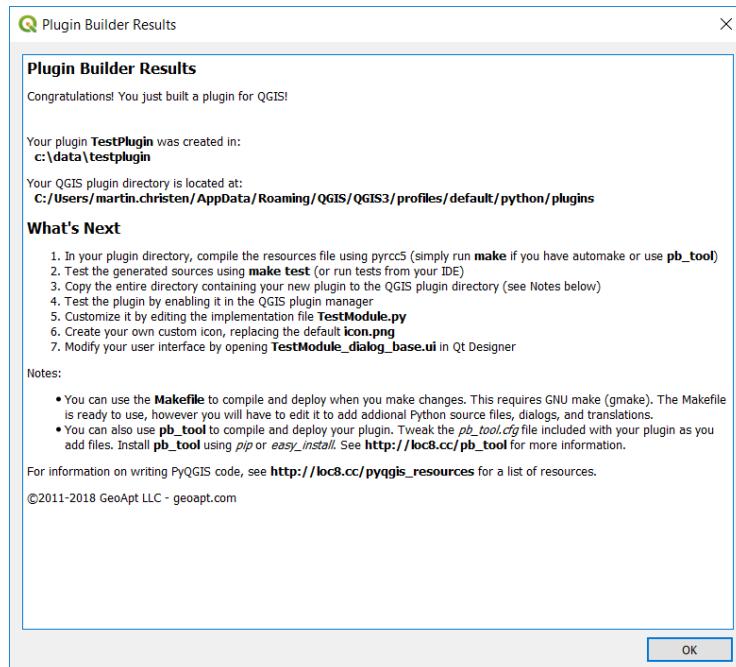
Wir füllen nun den Dialog auf. Der Klassename und das Plugin soll «TestPlugin» heißen.

Im Feld «About» wird danach einfach eine Beschreibung des Plugins angegeben, und danach wird «Tool button with dialog» ausgewählt und ein Menu Eintrag definiert. In unserem Fall «Test IGEO»



Danach wird die URL für den Bug-Tracker und das Respository (wo der Source Code sich befindet) angegeben. Da wir dies nicht unterstützen schreiben wir n/a (not available). Bei einem richtigen Plugin ist dies jedoch erforderlich. Dann geben wir an, wo das Plugin installiert werden soll:





In C:\Programme\QGIS ein File «OSGeo4W_New.bat» erstellen mit folgendem Inhalt:

```
@echo off
rem Root OSGEO4W home dir to the same directory this script exists in
call "%~dp0\bin\o4w_env.bat"
@echo off
call "%OSGEO4W_ROOT%\bin\o4w_env.bat"
call "%OSGEO4W_ROOT%\apps\grass\grass-7.4.2\etc\env.bat"
path %PATH%;%OSGEO4W_ROOT%\apps\qgis\bin
path %PATH%;%OSGEO4W_ROOT%\apps\grass\grass-7.4.2\lib
path %PATH%;%OSGEO4W_ROOT%\apps\Qt5\bin
SET PYTHONPATH=
SET PYTHONHOME=%OSGEO4W_ROOT%\apps\Python37
PATH %OSGEO4W_ROOT%\apps\Python37;%OSGEO4W_ROOT%\apps\Python37\Scripts;%PATH%

rem List available o4w programs
rem but only if osgeo4w called without parameters
@echo on
@if [%1]==[] (echo run o-help for a list of available commands & cmd.exe /k) else (cmd /c "%*")
```

Zunächst installieren wird das pb_tool:

```
pip install pb_tool
```

Dort geben wir «cd» (Current directory) das das Verzeichnis ein, wo das Plugin gespeichert wurde.

Also z.B:

```
cd c:\data\testplugin
```

Dann deployen wir das Plugin

```
pb_tool deploy
```

Nach Verändern des GUIs (designer.exe) geben wir jeweils ein:

```
pyrcc5 -o resources.py resources.qrc
```

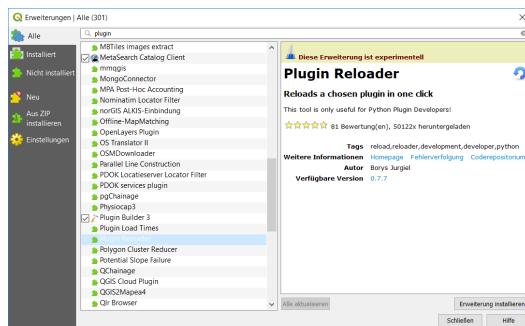
Nun starten wir QGIS erneut (es muss vorher geschlossen worden sein!).

Das Plugin kann gestartet werden!

10.8 QGIS Plugin Entwicklung mit PyCharm

10.8.1 Installation Plugin Reloader

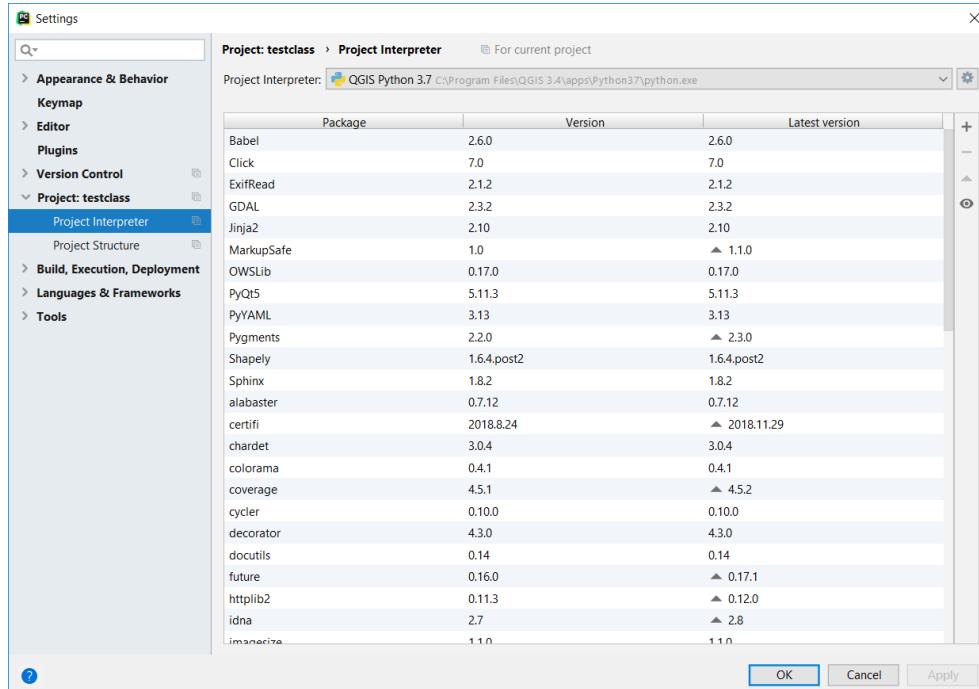
Zunächst installieren wir das QGIS Modul: «Plugin Reloader». Mit diesem müssen wir QGIS nicht jedesmal neu starten wenn das Plugin geändert wird.



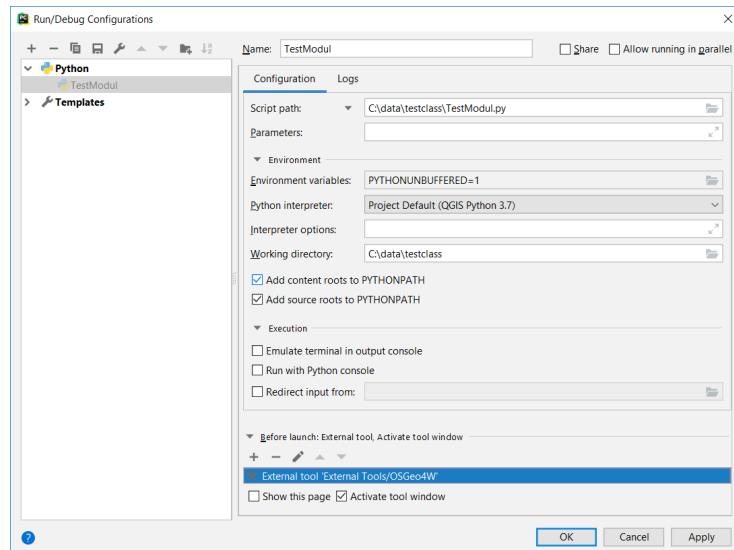
Um den Plugin Reloader zu installieren muss unter Einstellungen die Checkbox «Auch experimentelle Erweiterungen anzeigen» aktiviert sein.

10.8.2 PyCharm Konfigurieren

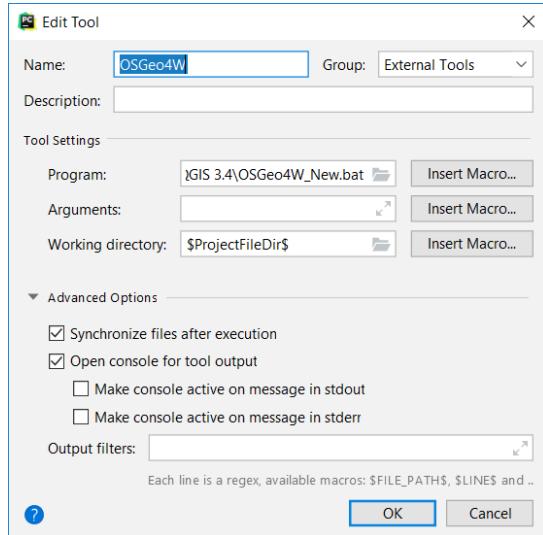
Zunächst Setzen wir den Python-Interpreter auf QGIS:



Dann die Run/Debug Konfiguratoin editieren. Externes Tool hinzufügen:
C:\Programme\QGIS 3.x\OSGeo4W_New.bat



Das externe Tool wird folgendermassen konfiguriert:

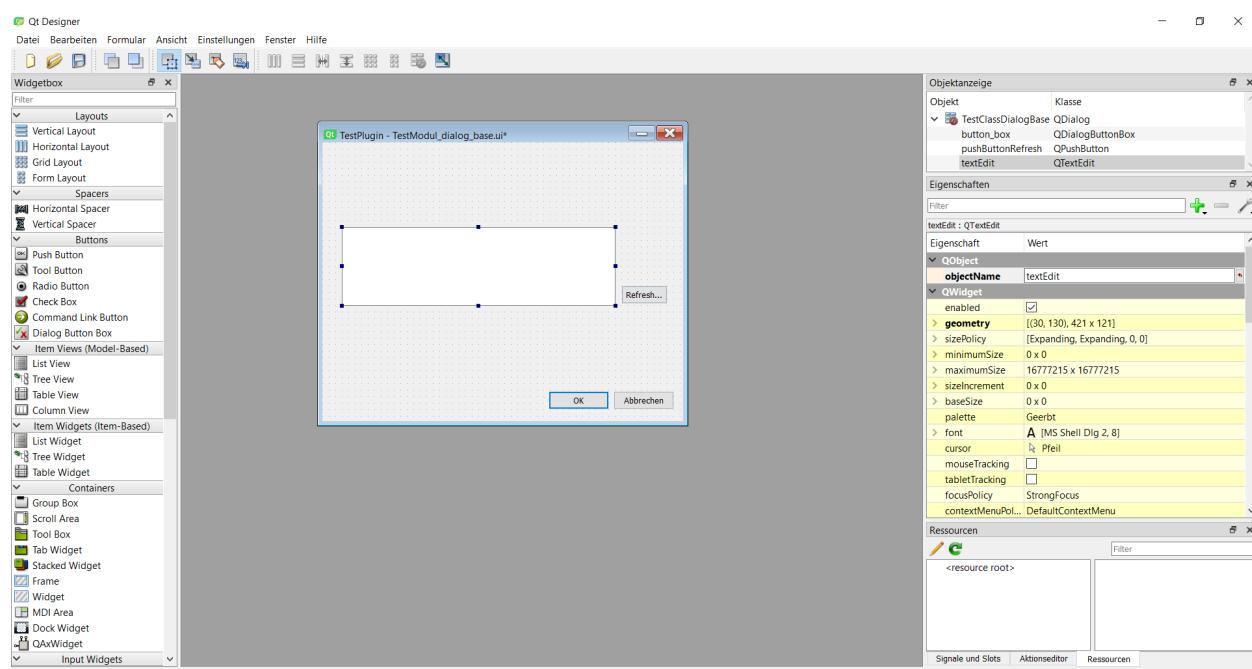


Hinweis: Mit PyCharm Professional kann man auch «Remote Debugging» betreiben.
Dies sehen wir hier nicht an.

10.8.3 Entwicklung eines Plugins

Wir editieren das ui-File. Fügen ein QTextEdit (read-only) hinzu mit Objektnamen "textEdit".

Und wir fügen ein Button "refresh" hinzu mit Objektnamen "pushButtonRefresh":



Wir können auch das icon.png editieren. Dies ist das Symbol unseres Plugins innerhalb QGIS.

Wir fügen im UI auch noch einen QgsMapLayerComboBox (ganz unten) hinzu
(Objektname: mapComboBox)

Nun erstellen wir folgenden Code im «Modulname_diglog.py» hinzu:

```
import os

from PyQt5 import uic
from PyQt5 import QtWidgets
from PyQt5.QtWidgets import QMessageBox

FORM_CLASS, _ = uic.loadUiType(os.path.join(
    os.path.dirname(__file__), 'TestModul_dialog_base.ui'))

class TestClassDialog(QtWidgets.QDialog, FORM_CLASS):
    def __init__(self, parent=None):
        """Constructor."""
        super(TestClassDialog, self).__init__(parent)
        self.setupUi(self)

        self.pushButtonRefresh.clicked.connect(self.refresh)

    def refresh(self):
        QMessageBox.information(self, "Info", "Refresh Button!")
```

Wir führen aus:

```
pb_tool deploy -y
```

Und wird starten das Plugin. Wenn keine Fehler im Code sind, funktioniert es.
Ansonsten: fix & reload!

Wir passen den Code an, so dass der aktuelle Layer aus der Dropbox angezeigt wird:

```
def refresh(self):
    layername = self.mapComboBox.currentText()
    QMessageBox.information(self, "Info", f"Refresh Button! {layername}")
```

Wir können auch weitere Dinge entwickeln:

```
def refresh(self):
    layername = self.mapComboBox.currentText()
    foundlayer = None
```

```

layers = QgsProject.instance().layerTreeRoot().children()
for i in layers:
    if i.name() == layername:
        foundlayer = i.layer()

if foundlayer:
    features = foundlayer.getFeatures()
    s = ""
    for feature in features:
        s = s + feature["Name"] + "\n"

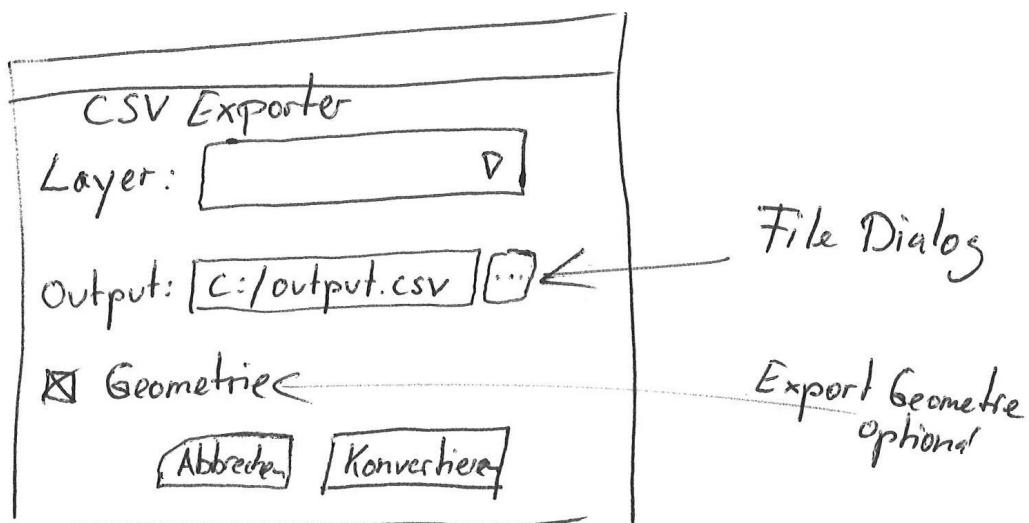
    self.TextEdit.setText(s)
else:
    QMessageBox.information(self, "Info", f"ERROR!!!! LAYER GONE!!!")
    self.TextEdit.setText("ERROR!!")

```

10.8.4 Entwicklung eines komplexeren Plugins

QGIS Plugins in einem Texteditor zu entwickeln ohne den Code zu starten/testen ist oft sehr schwierig bis unmöglich, weil Fehler nicht leicht erkennbar sind.

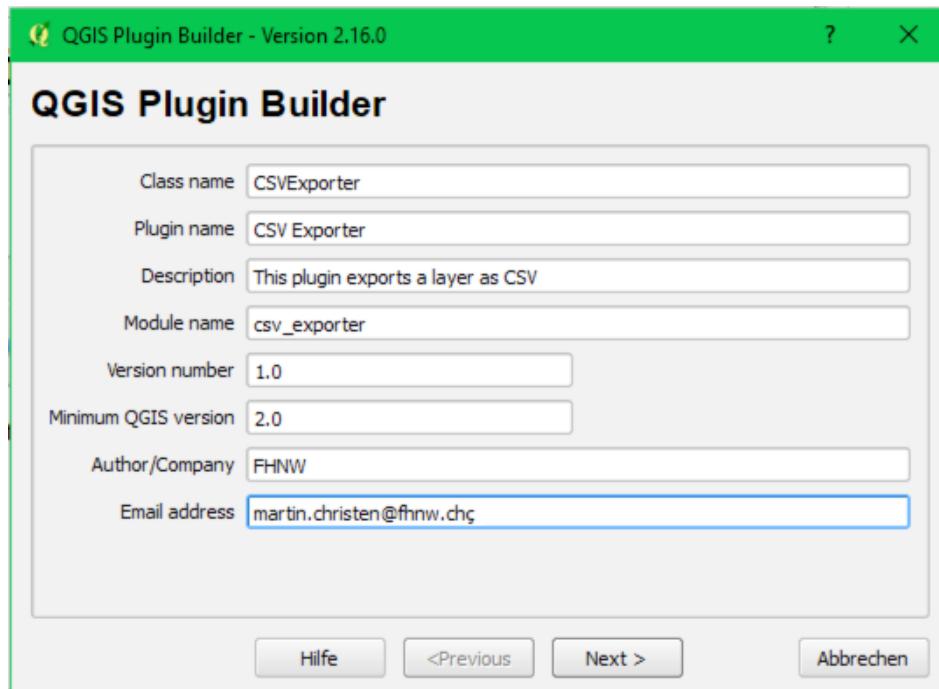
Wir implementieren nun ein einfaches CSV-Exporter-Plugin, welches folgendermassen spezifiziert wird:

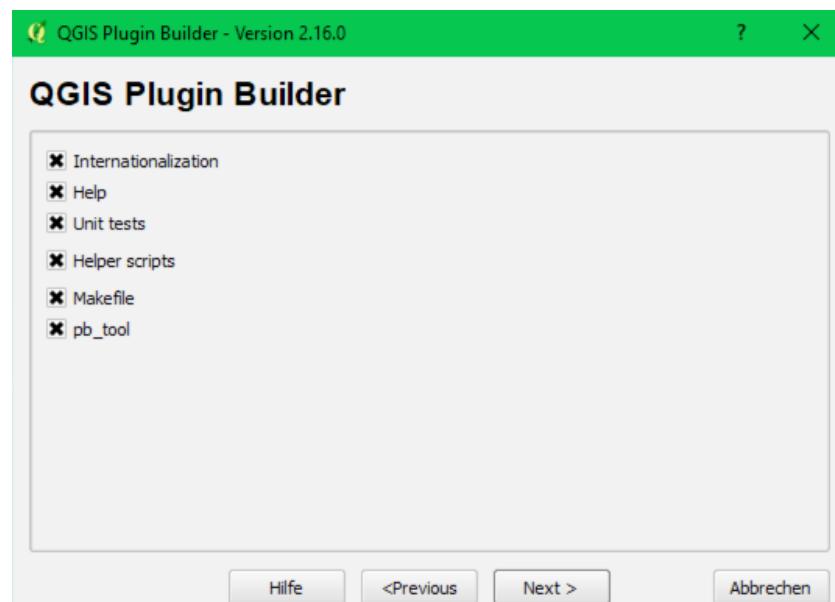
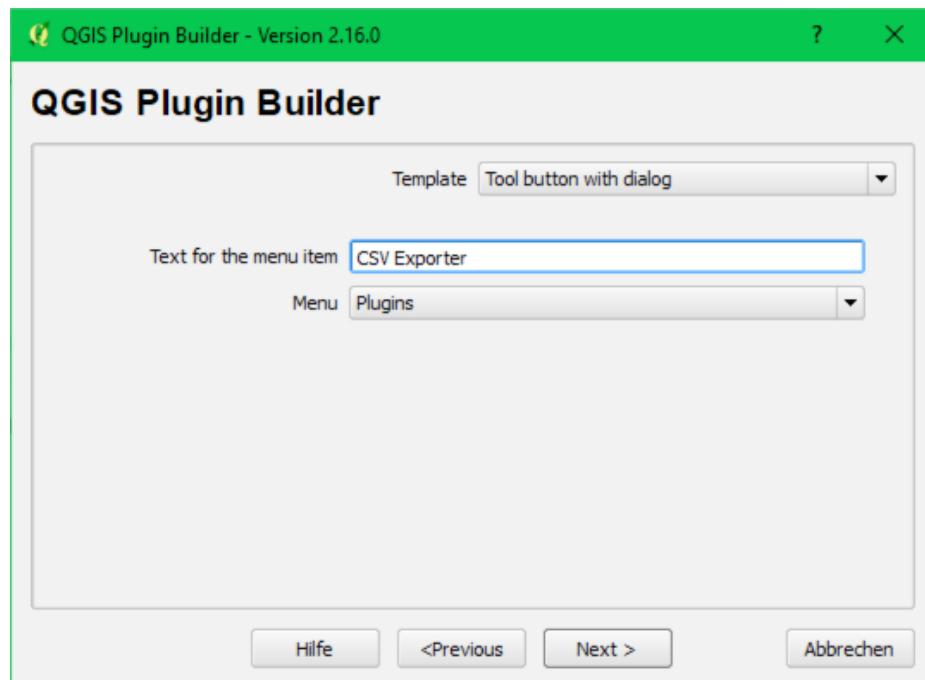


Zum Glück ist es möglich ein solches Plugin auch mit PyCharm zu entwickeln. Dazu kann folgendes .bat file verwendet werden, um PyCharm mit QGIS integration zu starten.

```
@echo off
SET OSGE04W_R00T="C:\Program Files\QGIS 2.18\
call "%OSGE04W_R00T%\bin\o4w_env.bat"
call "%OSGE04W_R00T%\apps\grass\grass-7.2.2\etc\env.bat"
@echo off
path %PATH%;%OSGE04W_R00T%\apps\qgis\bin
path %PATH%;%OSGE04W_R00T%\apps\grass\grass-7.2.2\lib
set PYTHONPATH=%PYTHONPATH%;%OSGE04W_R00T%\apps\qgis\python;
set PYTHONPATH=%PYTHONPATH%;%OSGE04W_R00T%\apps\Python27\Lib\site-packages
set QGIS_PREFIX_PATH=%OSGE04W_R00T%\apps\qgis
start "PyCharm QGIS" /B "C:\Program Files (x86)\JetBrains\PyCharm Community Edition 5.0.4\bin\pycharm.exe" %*
```

Wir erstellen zunächst in QGIS eim Plugin „CSV Exporter“:

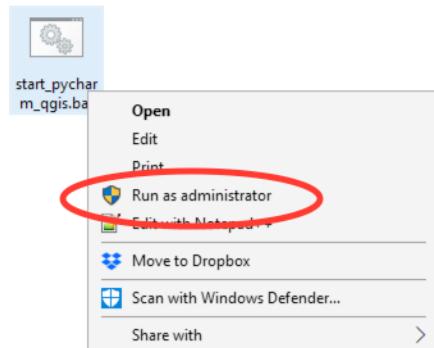




wichtig ist, das „pb_tool“ aktiviert ist“

Das Plugin wird nun gespeichert (z.B. unter C:/Develop/CSV_Exporter)

Danach wird PyCharm über das .bat File als Administrator gestartet. Dann öffnen wir das gerade erstellte Projekt (CSVExporter). Dies ist notwendig, da wir neue Python Module installieren wollen. Später können wir das .bat File auch ohne Administrator Rechte ausführen.



Ist das Projekt geladen, gehen wir in die „Python Console“:
Dort sollte jetzt “Python 2.7.5” stehen.

Danach wechseln wir in das **Terminal**.

Nun müssen wir pip installieren. Dies geschieht mit:

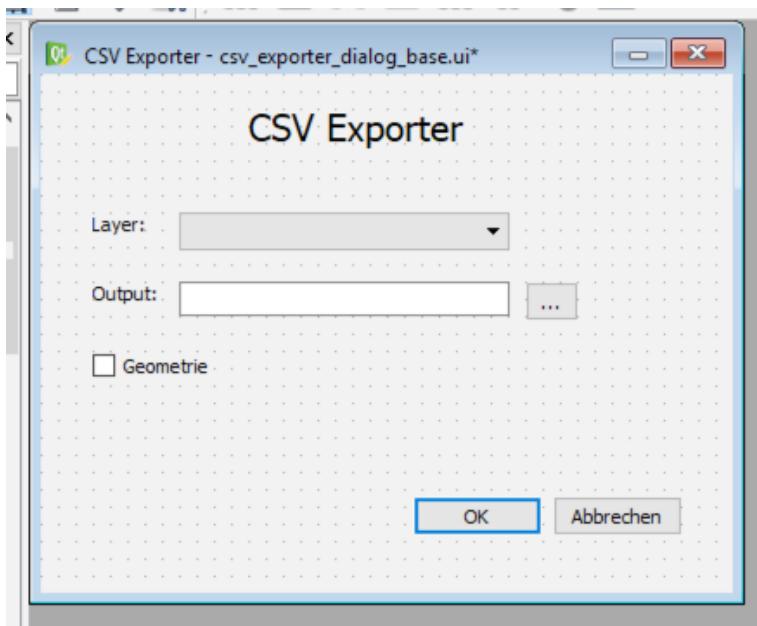
```
pip install requests==2.5.3
python -m pip install --upgrade pip setuptools wheel
```

Im Terminal wird nun das pb_tool (Publish Tool) installiert (dauert eine Weile)

```
python -m pip install pb_tool
Remote Debugging:
```

```
python -m pip pydevd
```

Nun wird mit dem Qt Designer das GUI erstellt:



Die Objektnamen für Checkbox, Combobox und LineEdit sind: geometrie, layer, filename

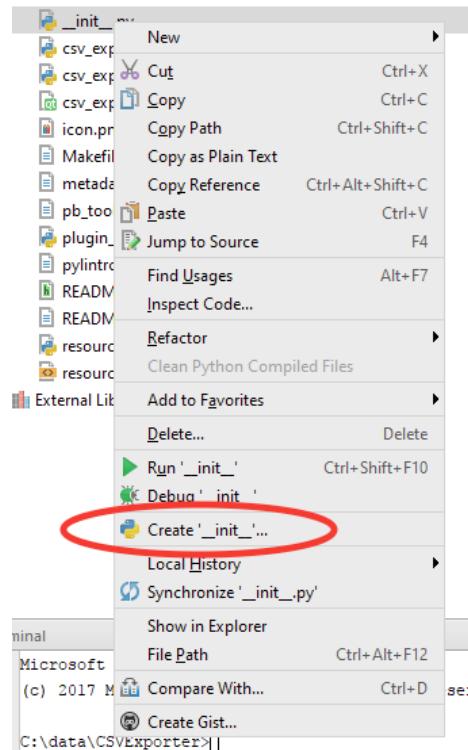
Nun starten wir PyCharm neu (bat-File ohne Administrator-Rechte)

Wir machen einen ersten deploy. Im Terminal geben wir ein:

```
pb_tool deploy
```

Dies installiert das Plugin.

Ein „Rechtsklick“ auf das File `__init__.py` öffnet das Popupmenu, bei dem wir „Create `__init__`“ wählen:



Und dort erstellen wir bei „Before launch: External Tool“ (mit +) zwei neue externe Tools:

Program: pb_tool

Parameters: deploy -y

Working Direktory: \$ProjectFileDir\$

Program: qgis

Parameters: --nologo -project c:/data/projekt.qgs

Working Direktory: \$ProjectFileDir\$

Nun starten wir das Programm.

Bei Start des Plugins sehen wir das Fenster...

Im Fuke csv_exporter.py fügen wir bei den imports folgendes hinzu (wir benötigen iface)

```
from qgis.utils import iface
```

In der Methode "run()" ergänzen wir:

```
layers = iface.legendInterface().layers()
layer_list = []
for layer in layers:
    layer_list.append(layer.name())

self.dlg.layer.clear()
self.dlg.layer.addItems(layer_list)
```

Im csv_exporter_dialog können wir dann die Logik für die Fileauswahl implementieren und danach das CSV File speichern (vgl. Kapitel 14.1.3)