PyQGIS the comfortable way

Tricks to efficiently work with Python and QGIS

Matthias Kuhn

Olten, 13.6.2019

OPENGIS.ch

ANDROID · [Q]GIS · WEB

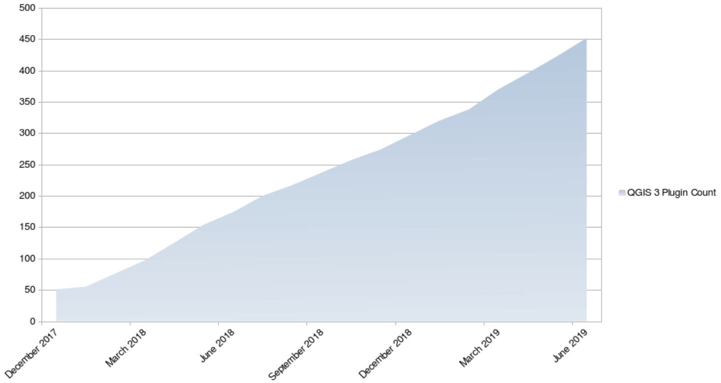
Matthias Kuhn

- QGIS Core Developer
- Co-Founder and CTO of OPENGIS.ch Ltd
- Skier and Mountaineer

Version 0.9 'Ganymede' (2007)

- **Python bindings - This is the major focus of this release it is now possible to create plugins using python. It is also possible to create GIS enabled applications written in python that use the QGIS libraries.**
- Removed automake build system - QGIS now needs CMake for compilation.
- Many new GRASS tools added (with thanks to http://faunalia.it/)
- Map Composer updates
- Crash fix for 2.5D shapefiles
- The QGIS libraries have been refactored and better organised.
- Improvements to the GeoReferencer

Plugin ecosystem



(because every presentation needs a trend graph)

Optimizing PyQGIS

- Various collections of "common pyqgis helper functions" have been written to "make things easier".

See: `http://osgeo-org.1560.x6.nabble.com/QGIS-Developer-Common-PyQGIS-functions-for-QGIS-3-td539564 html`

Common PyQGIS functions for QGIS 3

“Wouldn’t it be possible to provide such a collection of common pyqgis functions not only from private persons/projects but from the QGIS-project itself so users could add common functions? I think the chances would be higher that such a "official" collection would be used in the long run and constantly extended."

— Thomas Baumann, QGIS Developer Mailing List

The goal

API first	Make flexible and easy to use APIs. Benefits Python and C++.
Pythonic	Implement “Pythonic" constructs. Leverage modern Python language features.

Decorators

What is a Decorator

“A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated."

— https://wiki.python.org/moin/PythonDecorators

A simpler explanation

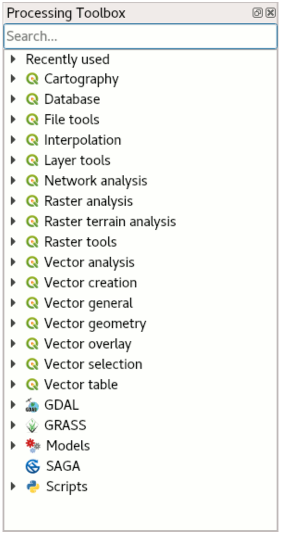
Decorators help to write code that is easier to write and read. It helps to avoid repeating "boilerplate code".
It's *syntactic sugar*.

Expression functions

```python
@qgsfunction(args='auto', group='Custom')
def sum(value1, value2, feature, parent):
    """
    Calculates the sum of the two parameters value1
        and value2.
    <h2>Example usage:</h2>
    <ul>
      <li>my_sum(5, 8) -> 13</li>
      <li>my_sum("field1", "field2") -> 42</li>
    </ul>
    """
    return value1 + value2
```

Processing

- Modular data processing pipelines
- Less effort to create the GUI
- But: A lot of boilerplate code
  - Processing provider
  - Methods for input and output definition
  - Methods for help
  - Method for the algorithm itself



Processing Algorithm

```python
class GeoCoding(Algorithm):
    INPUT: 'INPUT'
    OUTPUT: 'OUTPUT'
    COLUMN_PREFIX: 'COLUMN_PREFIX'

    def name(self):
        return 'geocoding'

    def initAlgorithm(self, config=None):
        self.addParameter(
            QgsProcessingParameterFeatureSource(
                self.INPUT,
                self.tr('Address Layer')
            )
        )
    def displayName(self):
    def group(self):
    def shortHelpString(self):
    ...
```

processing.alg decorator

```python
@alg(name="geocode", label=alg.tr("GeoCode"))
@alg.input(type=alg.SOURCE, name="INPUT", label="Adress layer")
@alg.input(type=alg.SINK, name="OUTPUT", label="Output layer")
def geocode(instance, parameters, context, feedback, inputs):
    """
    Geocode locations. Addresses in, points out.
    May produce multiple points for an address if ambiguous.
    """
    source = instance.parameterAsSource(parameters, "INPUT", context)
    (sink, dest\_id) = instance.parameterAsSink(parameters, "OUTPUT", context, source.fields(), QgsWkbTypes.Point, QgsCoordinateReferenceSystem(4326))

    GeoCoder.resolve(source, sink)

    return {"OUTPUT": dest\_id}
```

processing.alg decorator

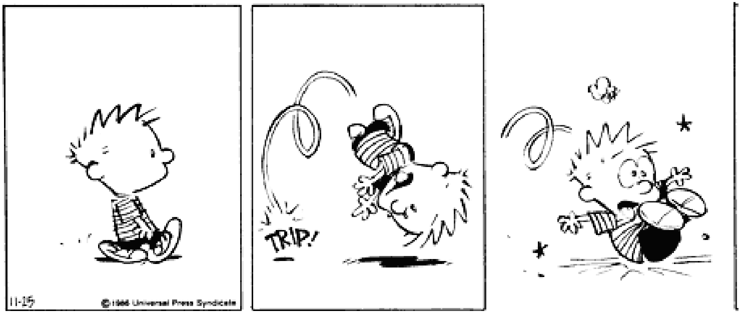
```
@alg(name="geocode", label=alg.tr("GeoCode"))
```

Validity checks

```python
@check.register(type=QgsAbstractValidityCheck.TypeLayoutCheck)
def layout_map_crs_choice_check(context, feedback):
    layout = context.layout
    results = []
    for i in layout.items():
        if isinstance(i, QgsLayoutItemMap) and i.crs().authid() == 'EPSG:3857':
            res = QgsValidityCheckResult()
            res.type = QgsValidityCheckResult.Warning
            res.title='Map projection is misleading'
            res.detailedDescription='The projection for the map item {} is set to <i>Web Mercator (EPSG:3857)</i> which misrepresents areas and shapes. Consider using an appropriate local projection instead.'.format(i.displayName())
            results.append(res)

        return results
```

When things go wrong



Converting a geometry to a point

```
QgsGeometry.fromWkt('POINT(3 4)').asPoint()
```

Converting a geometry to a point

```
QgsGeometry.fromWkt('POINT(3 4)').asPoint()
```

```
<QgsPointXY: POINT(3 4)>
```

Converting a geometry to a point

```
QgsGeometry.fromWkt('LINESTRING((3 4), (7 8))').
    asPoint()
```

Converting a geometry to a point

```python
QgsGeometry.fromWkt('LINESTRING((3 4), (7 8))').
    asPoint()
```

```
<QgsPointXY: POINT(0 0)>
```

Converting a geometry to a point

```
QgsGeometry.fromWkt('LINESTRING((3 4), (7 8))').
    asPoint()
```

```
<QgsPointXY: POINT(0 0)>
```

... only until QGIS 3.4

Converting a geometry to a point

```
QgsGeometry.fromWkt('LINESTRING((3 4), (7 8))').
    asPoint()
```

Converting a geometry to a point

```
QgsGeometry.fromWkt('LINESTRING((3 4), (7 8))').
    asPoint()
```

```
Traceback (most recent call last):
  File "plugins/somewhere/plugin.py",
                line 90, in broken_method
TypeError: LineString geometry cannot be converted
           to a point. Only Point types are permitted.
```

When things go wrong

```
mp = QgsMultiPoint()
mp.addGeometry(QgsPoint(1,1))
mp.addGeometry(QgsPoint(2,2))
```

When things go wrong

```
mp = QgsMultiPoint()
mp.addGeometry(QgsPoint(1,1))
mp.addGeometry(QgsPoint(2,2))
```

```
mp.geometryN(3)
```

When things go wrong

```
mp = QgsMultiPoint()
mp.addGeometry(QgsPoint(1,1))
mp.addGeometry(QgsPoint(2,2))
```

```
mp.geometryN(3)
```

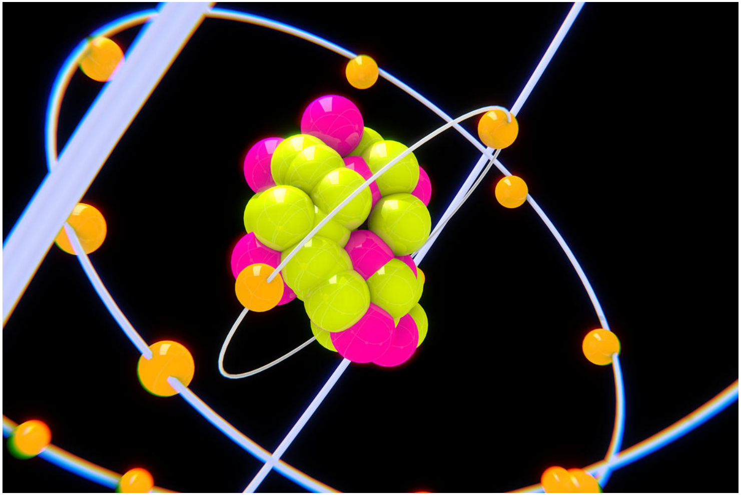
```
IndexError: 3
```

When things go wrong

```python
try:
    mp.geometryN(3)
except IndexError as e:
    show_error(self.tr('The geometry is too short.
        Input data linestrings need to have at least 4
         vertices.'))
```

When things go wrong

Atomic Operations



Atomic operations using with

```python
# Fix population from absolute to relative
layer.startEditing()
for feat in layer.getFeatures():
    feat['population'] = feat['population'] / feat['area']
    layer.updateFeature(feat)
layer.commitChanges()
layer.stopEditing()
```

Atomic operations using with

```python
# Fix population from absolute to relative
layer.startEditing()
for feat in layer.getFeatures():
    feat['population'] = feat['population'] / feat['
        area']
    layer.updateFeature(feat)
layer.commitChanges()
layer.stopEditing()
```

```
ZeroDivisionError: division by zero
```

Atomic Operations

Atomic operations using "with"

- Only part of the features are modified
- The layer may or may not be in edit state any more

Atomic operations using "with"

Let's introduce "with"

Atomic operations using "with"

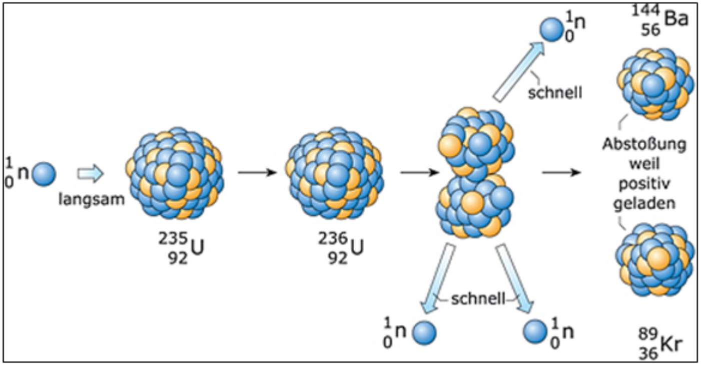
```
# Fix population from absolute to relative
with edit(layer):
    for feat in layer.getFeatures():
        feat['population'] = feat['population'] / feat
            ['area']
        layer.updateFeature(feat)
# Changes are committed automatically if no error
   occurred
```

Atomic operations using "with"

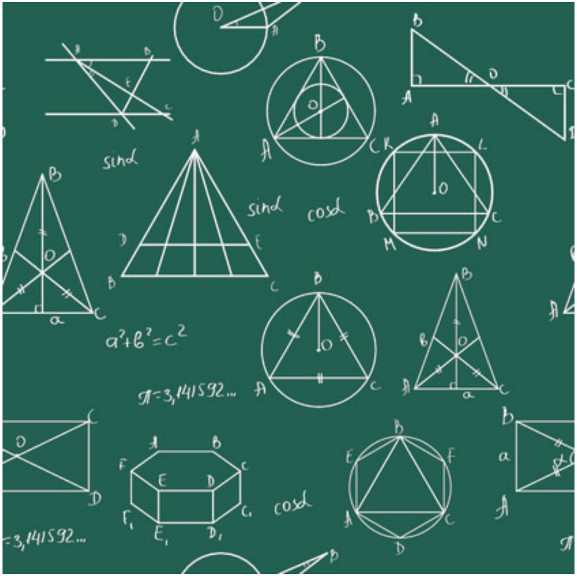
```python
# Fix population from absolute to relative
with edit(layer):
    for feat in layer.getFeatures():
        feat['population'] = feat['population'] / feat
            ['area']
        layer.updateFeature(feat)
# Changes are committed automatically if no error
    occurred
```

```
# Or if an error occurs, no changes are applied at all
ZeroDivisionError: division by zero
```

Atomic Operations



Working with geometries



Iterating vertices

```python
line = QgsGeometry.fromWkt('LINESTRING(1 1, 2 2)')
for vertex in line.vertices():
    print(vertex)
```

Iterating vertices

```python
line = QgsGeometry.fromWkt('LINESTRING(1 1, 2 2)')
for vertex in line.vertices():
    print(vertex)
```

```
<QgsPoint: Point (1 1)>
<QgsPoint: Point (2 2)>
```

Iterating parts

```python
multipoint = QgsGeometry.fromWkt('MULTIPOINT((1 1), (2 2), (3 3))')
for point in multipoint.parts():
    print(point)
```

Iterating parts

```python
multipoint = QgsGeometry.fromWkt('MULTIPOINT((1 1), (2 2), (3 3))')
for point in multipoint.parts():
    print(point)
```

```
<QgsPoint: Point (1 1)>
<QgsPoint: Point (2 2)>
<QgsPoint: Point (3 3)>
```

Representing objects

```
QgsPoint(2635450,1244252)
```

Representing objects

```
QgsPoint(2635450,1244252)
```

```
<qgis._core.QgsPoint object at 0x7fcd2b428ee8>
```

Representing objects, Since QGIS 3.2

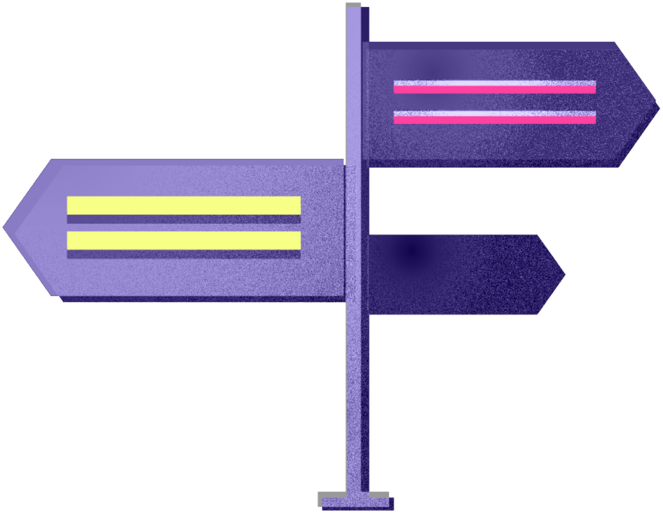
```
QgsPoint(2635450,1244252)
```

Representing objects, Since QGIS 3.2

```
QgsPoint(2635450,1244252)
```

```
<QgsPoint: Point (2635450 1244252)>
```

Outlook



Exception handling

- Exceptions are good
  - Exceptions help to fix problems
  - Exceptions help in case of data corruption
- More exceptions
- E.g. instead of return values

Easier initialization

- A lot of boilerplate code is required to get started with a standalone application
- Goal: reduce that

More pythonic constructs

- More decorators
- More iterators

Nice API

- But that is not Python specific

Start Coding

- Let's get to work

Thank you

Questions? Now or later...