QGIS プラグイン 実践的開発セミナー

山手　規裕
（Pacific Spatial Solutions株式会社）

自己紹介

山手　規裕（Pacific Spatial Solutions株式会社）

測量士、ソフトウェア開発技術者、森林情報士（GIS、森林航測、リモートセンシング一級）

QGISはおそらくv0.8から使用しています。

プラグイン、およびQGISアプリケーションの開発を行なっています。

本セミナーの進め方

あらかじめダウンロードしていただいたプラグインのソースコード（https://github.com/Arctictern265/routing_example/tree/files）をテキストエディタなどで開いていただき、コードを参照しながら説明していきます。

ご不明な点などございましたら随時ご質問ください。

QGISプラグイン開発の準備

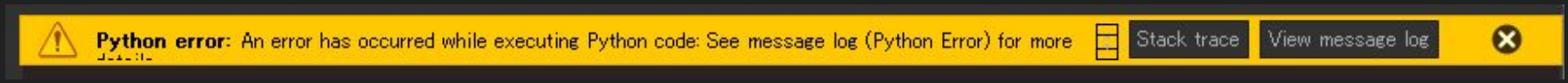
プラグイン開発は最低限QGISとテキストエディタがあればできますが、以下のプラグインがあるとより便利です。

- PluginBuilder
  - 対話形式でプラグインの雛形を作成してくれるプラグイン
  - プラグインのツールボタンを押すとダイアログ（またはドッキングパネル）が表示されるという雛形を作成してくれる
- PluginReloader
  - プラグインの変更を反映させてくれるプラグイン

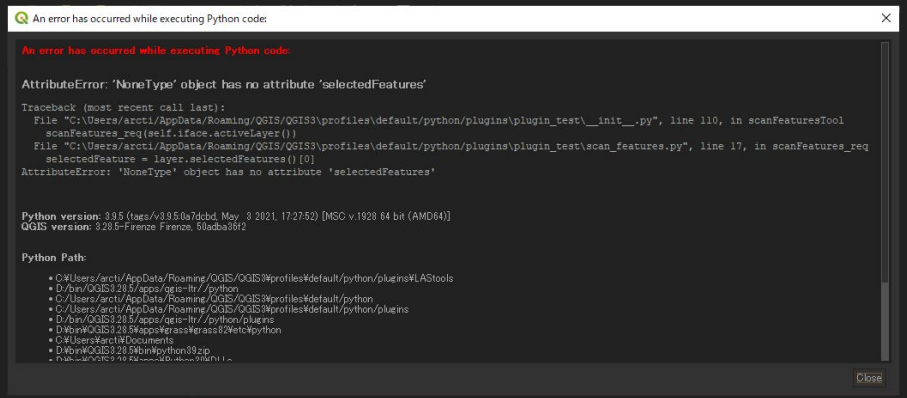
その他、Eclipse+PyDevなどのIDEを使ってデバッグ環境を構築することもできますが、本セミナーでは割愛します。

エラーメッセージは消さないで！

プラグインの開発をしているとエラーメッセージはしょっちゅう目にしますが、Pythonのエラーメッセージはかなり親切に原因箇所と内容を知らせてくれます。



[Stack trace]ボタンを押すとエラーのダイアログが表示されます。



本セミナーで使用するプラグインの内容

プラグインを起動するとレイヤ選択ダイアログが表示されます。選択後[OK]ボタンを押すとカーソルが十字になります。1点目をキャンバス上でクリックするとマーカーが表示されます。2点目をクリックすると最短ルートを検索し、検索結果をキャンバス上に表示します。

本セミナーで使用するプラグインの内容

本プラグインの機能は大きく分けて以下の4つの部分から構成されています。

- レイヤ選択ダイアログの表示
- キャンバスから始点、終点をクリックするマップツール
- 検索の実行部分
- 検索結果をキャンバス上に表示
- （右クリックでキャンセル、1点目のマーカー表示）

PluginBuilderの生成コード（__init__.py）

```python
# noinspection PyPep8Naming
def classFactory(iface):  # pylint: disable=invalid-name
    """Load RoutingExample class from file RoutingExample.

    :param iface: A QGIS interface instance.
    :type iface: QgsInterface
    """
    #
    from .routing_example import RoutingExample
    return RoutingExample(iface)
```

`classFactory()` 関数はQGIS起動時にQGISから呼び出される関数。
プラグインはメインとなるクラスのオブジェクトを一つ返す。

引数の`iface`は`QgisInterface`オブジェクトで、このオブジェクトを通して QGIS本体の操作や情報のやり取りを行うことができる。

PluginBuilderの生成コード（routing_example.py）

__init__()（抜粋）

```
def __init__(self, iface):

        self.iface = iface


        self.plugin_dir = os.path.dirname(__file__)


        self.actions = []


        self.menu = self.tr(u'&RoutingExamplePlugin')
```

QgisInterfaceオブジェクトをメンバ変数に保存

プラグインのディレクトリをメンバ変数に保存。追加のファイルとかを使いたいときに便利

アクションとプラグインのメニューの名前を初期化。アクションはこの後追加する。

PluginBuilderの生成コード（routing_example.py）

`add_action()` ： routing_example.py 136～208行目

アクションを登録する関数。
アクションとはメニューやツールボタンを押されたときに実行するもの。アクションとコールバック関数を紐づけて使用する。
PluginBuilderが生成した直後の状態ではこの後の `initGui()` で1つアクションが追加される。

`initGui()` ： routing_example.py 210～221行目

QGISから呼び出される関数。初期状態ではアクションを一つ登録している。
アクションに対するコールバック関数として `self.run` を指定している。
ここで追加されたアクションは、QGISのメニューとツールバーに表示される。

`unload()` ： routing_example.py 224～230行目

QGISから呼び出される関数。プラグインが無効化されるときに呼び出される。
アクションの登録を削除している。

PluginBuilderの生成コード（routing_example.py）

`run()` ： routing_example.py 233行目～

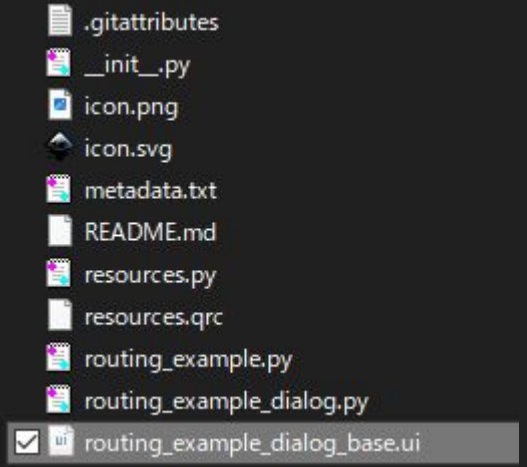
`initGui()`関数で登録されたアクションに対するコールバック関数。初期状態では以下のコードが書かれています（コメント行は除いています）。

```
    def run(self):
        if self.first_start == True:
            self.first_start = False
            self.dlg = RoutingExampleDialog()

        self.dlg.show()
        result = self.dlg.exec_()
        if result:
            pass
```

PluginBuiderが生成したダイアログを表示し、
[OK]ボタンが押されたら（`result`変数）なにか処理を実行。
初期状態では`pass`となっている。

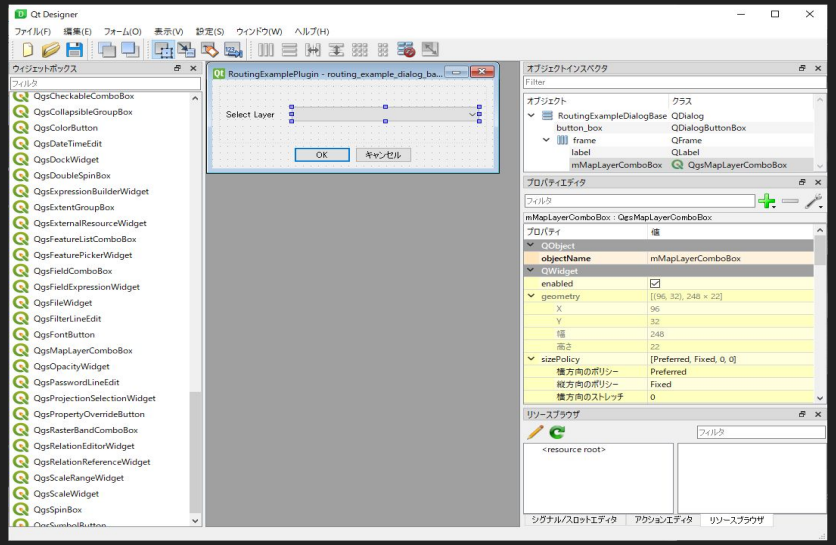
レイヤ選択ダイアログ

QGISに付属している「Qt Designer with QGIS custom widgets」から、PluginBuilderが作成したUIファイル（.ui）を開きます。



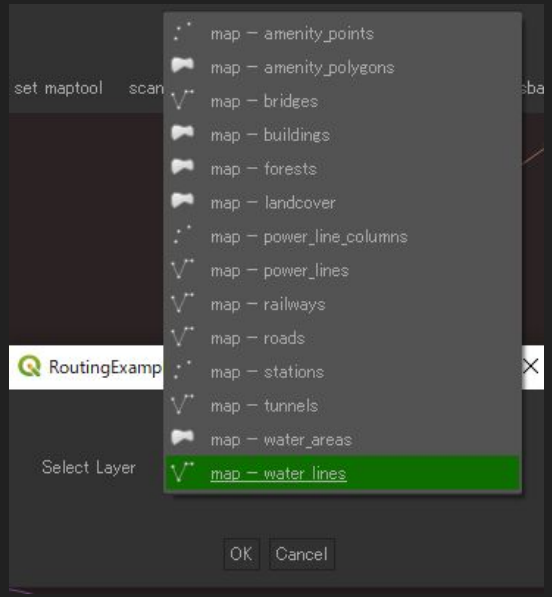
レイヤ選択ダイアログの作成

「Qt Designer with QGIS custom widgets」には、通常のQt Designerが提供するウィジェットの他に、QGIS独自のウィジェットがあります。この中から「QgsMaplayerComoboBox」を選択してダイアログに配置します。



QgsMapLayerComboBox

現在読み込まれているレイヤを自動的に読み取って、コンボボックスで選択できるようになるという優れもの！



選択されたレイヤを取得するには

`QgsMapLayerComboBox::currentLayer()`

を使用します。

レイヤ選択ダイアログのコード説明

```
import os

from qgis.PyQt import uic
from qgis.PyQt import QtWidgets

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


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

        self.setupUi(self)
```

.UIファイルを読み取って
FORM_CLASSを作成

FORM_CLASSを継承することで、ダイアログ内のウィジェットを参照することができるようになる。

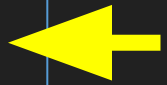
レイヤ選択ダイアログを閉じたあとの処理

routing_example.pyの250行目～

```
self.layer = self.dlg.mMapLayerComboBox.currentLayer()

self.tool = QgsMapToolEmitPoint(self.iface.mapCanvas())

self.tool.canvasClicked.connect(self.registerPoints)

self.iface.mapCanvas().setMapTool(self.tool)
```

選択されたレイヤを取り出す
戻り値は`QgsVectorLayer`型

マップツールオブジェクトを作成



マップツールでクリック時のスロットを設定

マップキャンバスにマップツールを設定

マップツール

マップキャンバス上で何らかの操作を行うためのもの。QgsMapToolクラスの派生クラスである必要がある。

QGIS APIでは以下の派生クラスがある。該当する機能のものがない場合はQgsMapToolクラスの派生クラスを自作する必要がある。

- QgsMapToolEdit
  - QgsMapTOolAdvancedDigitizing
    - QgsMapToolCapture
      - QgsMapToolCaptureLayerGeometry
    - QgsMapToolModifyAnnotation
- QgsMapToolEmitPoint
- QgsMapToolExtent
- QgsMapToolIdentify
  - QgsMapToolIdentifyFeature
- QgsMapToolPan
- QgsMapToolZoom

今回の例ではQgsMapToolEmitPoint が使える

QgsMapToolEmitPoint

キャンバス上をクリックしたときにその時押されたボタンとクリックした位置をシグナルで発行するマップツール。

```
self.tool = QgsMapToolEmitPoint(self.iface.mapCanvas())


self.tool.canvasClicked.connect(self.registerPoints)


self.iface.mapCanvas().setMapTool(self.tool)
```

マップツールオブジェクトを作成するときはマップキャンバスオブジェクトを引数に渡す

マップツールでクリック時のスロットを設定

マップキャンバスにマップツールを設定

※シグナル／スロットについて

シグナル／スロットは、一般的にはイベント（シグナル）とイベントリスナ（スロット）のことで、Qt独自の用語。

PyQtでの基本的な用法は以下のとおり。

```
self.tool.canvasClicked.connect(self.registerPoints)
```

シグナル発行元オブジェクト

シグナル

シグナルを受け取るオブジェクトとスロット

`QgsMapToolEmitPoint::canvasClicked` シグナルは引数としてクリックした点（`QgsPointXY` 型）と押されたボタンの種類（`Qt::MouseButton` 型）を送るので、スロット側ではこれらを引数とした関数を実装する必要がある。

マップツールのスロットの実装

`registerPoints()`関数（routing_example.py　255～259行目まで）

あらかじめ、__init__()関数で以下のメンバ変数を初期化しておく

```
        self.points = []
        self.rubberBand = QgsRubberBand(self.iface.mapCanvas(), QgsWkbTypes.LineGeometry)
```

```
def registerPoints(self,
                   point: QgsPointXY,
                   button: Qt.MouseButton):
    if button == Qt.LeftButton:
        self.points.append(point)
        if len(self.points) == 1:
            self.placeStartPos(point)
```

スロットとなる関数の定義

押されたボタンの判定

左ボタンならクリックした点の座標を追加

1点目なら`placeStartPos()`を呼び出す

マップツールのスロットの実装

`registerPoints()`関数（routing_example.py　260～266行目まで）

```
elif len(self.points) == 2:
    res, errorMsg, route =
            createShortestPath(self.layer, self.points)

    if res == False:
        QMessageBox.information(
                        self.iface.mainWindow(),
                        "info", errorMsg)
        self.points.clear()
        return
```

２点目だったらルート検索を実行

ルート検索に失敗したらエラーメッセージを表示して、１点目のマーカーを削除して関数を抜ける

マップツールのスロットの実装

`registerPoints()`関数（routing_example.py　268～273行目まで）

```
self.rubberBand.reset(QgsWkbTypes.LineGeometry)
self.rubberBand.setStrokeColor(QColor.fromRgb(0, 255, 0, 128))
self.rubberBand.setWidth(4.0)
self.rubberBand.setToGeometry(QgsGeometry.fromPolylineXY(route))
self.points.clear()
QgsProject.instance().annotationManager().clear()
```

ルート検索結果をキャンバス上に表示（後ほど説明します）
クリックされた2点と始点のアノテーションを消去

マップツールのスロットの実装

`registerPoints()`関数（routing_example.py　274～277行目まで）

```
else:
    self.points.clear()
    self.rubberBand.reset(QgsWkbTypes.LineGeometry)
    QgsProject.instance().annotationManager().clear()
```

左ボタン以外が押されたときはクリックされたポイントをクリアしてルート検索結果を削除（リセット）、１点目のポイントマーカーもクリア
（これも後ほど説明します）

ルート検索コードの詳細

`createShortestPath()`関数（routing_example.py　60行目）

```
def createShortestPath(layer: QgsVectorLayer, points: List) -> Any:
```

ルート検索に必要な情報は、対象とするレイヤと始点終点の2点なので、これらの情報を引数とする関数を宣言して中身を実装する形としました。

ルート検索コードの詳細

createShortestPath()関数（routing_example.py　61～62行目）

```
director = QgsVectorLayerDirector(
        layer, -1, '', '', '', QgsVectorLayerDirector.DirectionBoth)
```

QgsVectorLayerDirector クラスは進行方向の定義を行うためのクラス。コンストラクタの引数は以下のとおり。

- 対象とするレイヤ
- 方向を示すフィールドのインデックス
- 順方向を示す文字列
- 逆方向を示す文字列
- 両方向を示す文字列
- デフォルトの進行方向

フィールドのインデックスは、属性テーブルの列を左側から0、1、2、、、の順で指定する。

方向を示す文字列とは、例えば順方向の文字列で ’f’を指定した場合、インデックスで指定した属性の値が ’f’のときは順方向にしか勧めないことを示す。

ルート検索コードの詳細

`createShortestPath()`関数（routing_example.py　63行目）

```
director.addStrategy(QgsNetworkDistanceStrategy())
```

`QgsVectorLayerDirector`クラスで検索する際のコストとして要素の長さを使用するように指定。

他には`QgsNetworkSpeedStrategy`がある。こちらは移動時間を示すフィールドとデフォルトの移動時間などをコンストラクタの引数として指定する。

ルート検索コードの詳細

`createShortestPath()`関数（routing_example.py　64行目）

```
builder = QgsGraphBuilder(layer.sourceCrs())
```

`QgsGraphBuilder`クラスは、この後使用するQgsGraphオブジェクトを構築するためのクラス。引数にはレイヤの空間参照（ QgsCoordinateReferenceSystem型）が必要。

その他オプションで座標変換を有効にするかどうか（デフォルトでは　True）、トポロジ構築時の 許容値（デフォルトでは0.0）、楕円体を示す文字列（デフォルトでは ”WGS84”）を指定することができる。

ほぼデフォルトのままで問題ないと思います。

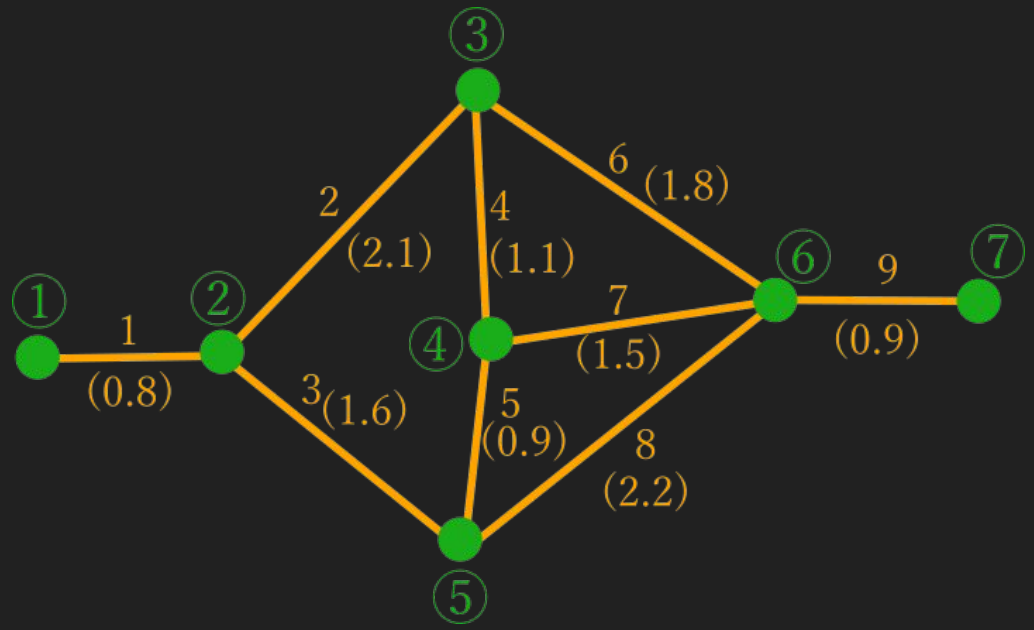
ルート検索コードの詳細

createShortestPath()関数（routing_example.py　65行目）

```
tiedPoints = director.makeGraph(builder, points)
```

QgsVectorLayerDirectorクラスを使用してグラフを作成する。
引数はQgsGraphBuilderオブジェクトとクリックされた２点の座標（QgsPointXY型のリスト）。
戻り値はレイヤにスナップされた始点、終点の座標（QgsPointXY型のリスト）。
作成されたグラフはこの後builder変数から取得する。

ルート検索のグラフについて



上の図のようなネットワークに数字のようにノード IDを付与し、右表のようなテーブルを作る。

id	start	end	cost
1	1	2	0.8
2	2	3	2.1
3	2	5	1.6
4	3	4	1.1
5	4	5	0.9
6	3	6	1.8
7	4	6	1.5
8	5	6	2.2
9	6	7	0.9

ルート検索のグラフについて

クリックされた点は通常ルートレイヤのノードとは一致していないので、ルート上の最も近い点にスナップさせて新たなノードを作成し、グラフに組み込むことで始点終点からの検索が可能になる。

ルート検索コードの詳細

createShortestPath()関数（routing_example.py　66～67行目）

```
tStart, tStop = tiedPoints
```

始点と終点を65行目のリストから取り出す（QgsPointXY型）

```
graph = builder.graph()
```

QgsGraphBuilderオブジェクトから作成されたグラフ
（QgsGraph型）を取得する

ルート検索コードの詳細

`createShortestPath()`関数（routing_example.py　68～69行目）

```
    idxStart = graph.findVertex(tStart)
    idxEnd = graph.findVertex(tStop)
```

グラフからスナップされた始点、終点のノードインデックスを取得する。
このインデックスはこの後で最適ルートを取り出すのに使用される。

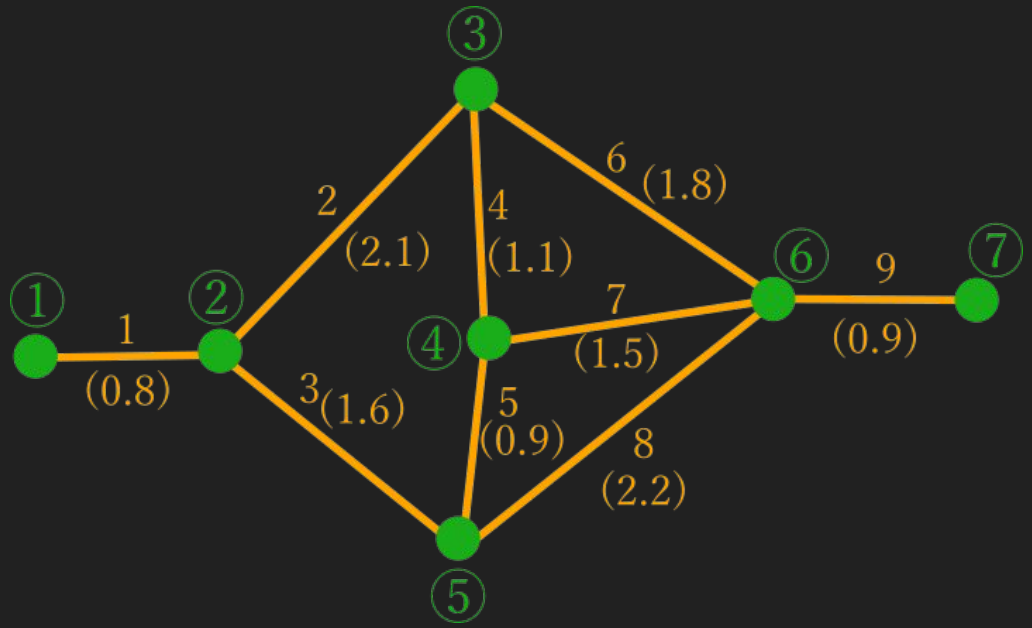
ルート検索コードの詳細

`createShortestPath()`関数（routing_example.py　70～71行目）

```
(tree, costs) = QgsGraphAnalyzer.dijkstra(
        graph, idxStart, 0)
```

Dijkstra法による最短ルートを実行する。`QgsGraphAnalyzer::dijkstra()`関数の引数は、順にQgsGraphオブジェクト、始点インデックス、基準コスト（通常は0で大丈夫のようです）を指定する。

Dijkstra法による最短経路検索



tree
-1
1
2
5
3
8
9

costs
0.0
0.8
2.9
3.3
2.4
4.6
5.5

Dijkstra法ではすべてのノードに対して始点から最短となる場合の直前のエッジIDと累積コストが算出される

ルート検索コードの詳細

`createShortestPath()`関数（routing_example.py　72～73行目）

```
    if tree[idxEnd] == -1:
        return (False, "no route found", [])
```

Dijkstra法による検索の結果で、終点に該当するインデックスが-1ということは始点からのルートが存在しないことを示している。この場合はルートが見つからないというメッセージを返して関数を抜ける。

ルート検索コードの詳細

`createShortestPath()`関数（routing_example.py　74～77行目）

```
route = [graph.vertex(idxEnd).point()]
while idxEnd != idxStart:
    idxEnd = graph.edge(tree[idxEnd]).fromVertex()
    route.insert(0, graph.vertex(idxEnd).point())
```

treeには始点から最短となる場合の直前のエッジインデックスしか情報がないので、終点からエッジを辿って始点までのノードをリストに追加していく。

```
idxEnd = graph.edge(tree[idxEnd]).fromVertex()
```

`graph.edge()`関数はインデックスで指定されたエッジ（`QgsGraphEdge`型）を返す。`fromVertex()`はエッジの始点インデックスを返す。

ルート検索結果の表示

`__init__()`関数（routing_example.py　118行目）

```
self.rubberBand = QgsRubberBand(self.iface.mapCanvas(), QgsWkbTypes.LineGeometry)
```

ここではルート検索結果の図形表示に`QgsRubberBand`を使用している。
`QgsRubberBand`クラスはコンストラクタでマップキャンバスを引数で指定する。これにより`QgsRubberBand`オブジェクトに図形を追加されたときにマップキャンバスに表示されるようになる。
コンストラクタの第2引数で要素の種別（ここではラインジオメトリ）をしていする。

ルート検索結果の表示

`registerPoints()`関数（routing_example.py　268～272行目）

```
self.rubberBand.reset(QgsWkbTypes.LineGeometry)
self.rubberBand.setStrokeColor(QColor.fromRgb(0, 255, 0, 128))
self.rubberBand.setWidth(4.0)
self.rubberBand.setToGeometry(QgsGeometry.fromPolylineXY(route))
```

`QgsRubberBand::reset()`関数でラバーバンドを消去する。このとき引数でジオメトリ種別を指定する必要がある。
線の色、線幅を指定し、ルート検索で取り出された頂点列を`QgsGeometry`型に変換してラバーバンドにセットしている。

１点目のアノテーションを表示

placeStartPos()関数（routing_example.py　281～292行目）

```
def placeStartPos(self, pos: QgsPointXY):
    markerProp = {"color": "0,255,0", "size": "3.0", "outline_width": "0.2", "outline_color": "0,0,0"}
    markerSymbol = QgsMarkerSymbol.createSimple(markerProp)
    doc = QTextDocument()
    doc.setHtml("<style>h1 {font: 24pt bold; color: black;}</style><h1>start point</h1>")
    annotation = QgsTextAnnotation()
    annotation.setDocument(doc)
    annotation.setMapPosition(pos)
    annotation.setMarkerSymbol(markerSymbol)
    annotation.setFrameOffsetFromReferencePoint(QPointF(20, -30))
    annotation.setMapPositionCrs(self.iface.mapCanvas().mapSettings().destinationCrs())
    QgsProject.instance().annotationManager().addAnnotation(annotation)
```

アノテーションとは一時的なポイントなどを表示するもので、ここでは
QgsTextAnnotationを使用している。

アノテーションの詳細

`placeStartPos()`関数（routing_example.py　282～283行目）

```
markerProp = {
  "color": "0,255,0",
  "size": "3.0",
  "outline_width": "0.2",
  "outline_color": "0,0,0"}
markerSymbol = QgsMarkerSymbol.createSimple(markerProp)
```

アノテーションのポイントを示すマーカーシンボルを作成。
シンボルの作成方法はイロイロあるが、ここでは`createSimple()`で作成している。この場合、マーカーの設定は辞書で設定して引数に渡す形式で作成する。

アノテーションの詳細

`placeStartPos()`関数（routing_example.py　284～285行目）

```
doc = QTextDocument()
doc.setHtml(
  "<style>h1 {font: 24pt bold; color: black;}</style>
   <h1>start  point</h1>")
```

テキストアノテーションのテキストは平文のテキストでも設定可能であるが、より目立つようにするためHTMLを使用した。HTMLを使用する場合は`QTextDocument`を使用する。

アノテーションの詳細

`placeStartPos()`関数（routing_example.py　286～290行目）

```
annotation = QgsTextAnnotation()
annotation.setDocument(doc)
annotation.setMapPosition(pos)
annotation.setMarkerSymbol(markerSymbol)
annotation.setFrameOffsetFromReferencePoint(QPointF(20, -30))
```

`QgsTextAnnotation`オブジェクトを作成。先に作成したテキストドキュメントとマーカー、およびクリックされた点の座標をセットする。
さらに、マーカーからテキストを表示させるフレームへのオフセットを指定する。

アノテーションの詳細

`placeStartPos()`関数（routing_example.py　291行目）

```
annotation.setMapPositionCrs(
        self.iface.mapCanvas().mapSettings().destinationCrs())
```

アノテーションの位置の空間参照をマップキャンバスの空間参照に設定する。本プラグインではあまり動作に影響はないが、設定していないとアノテーションを配置した後でマップキャンバスの空間参照を変更したときにアノテーションがついてこなくなる。

上記のコードの２行目がマップキャンバスの空間参照を取得している。このコードは頻繁に出てきます。

アノテーションの詳細

`placeStartPos()`関数（routing_example.py　292行目）

```
QgsProject.instance().annotationManager().addAnnotation(annotation)
```

アノテーションを追加する場合は上記のコードを実行する。
`QgsProject`オブジェクトは新規プロジェクトを作ったり、最初にベクタレイヤを読み込んだりしたときに１つ作られる。

`QgsProject`はレイヤの管理などのプロジェクト関連の操作を行うことができるもので、なにかとよく使います。

プラグインの改良

ステップアップとして、以下のような改造をしてみてください。

- ラバーバンドの色、線幅の変更
- 検索結果をラバーバンドではなく新規ベクタレイヤに保存する
- １点目のアノテーションの種類、マーカーの変更
- ルート検索で有方向検索やコストを使用してみる
- グラフ構築時のフィードバック関数を使用して進行状況を表示する

参考サイト

- PyQGIS 開発者用 Cookbook
  - https://docs.qgis.org/3.10/ja/docs/pyqgis_developer_cookbook/index.html
- QGIS Python API documentation
  - https://qgis.org/pyqgis/3.28/index.html
- QGIS API Documentation Class List
  - https://api.qgis.org/api/3.28/annotated.html
- Qt for Python
  - https://doc.qt.io/qtforpython-6/