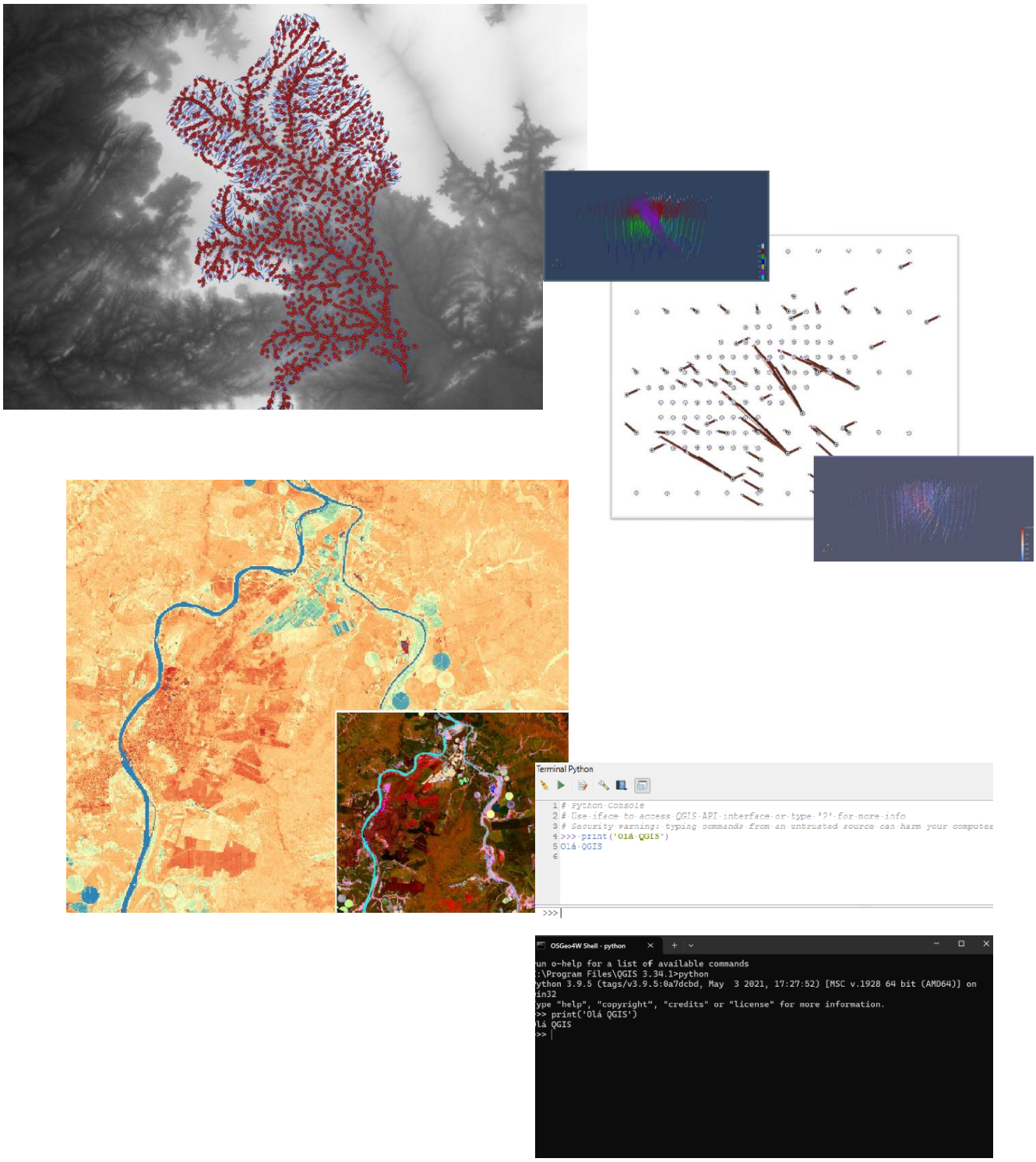
Criando PLUGINS com pyQGIS



André Luiz Lima Costa, M.Sc., P.Geo., FAIG

3ª edição - Julho 2024

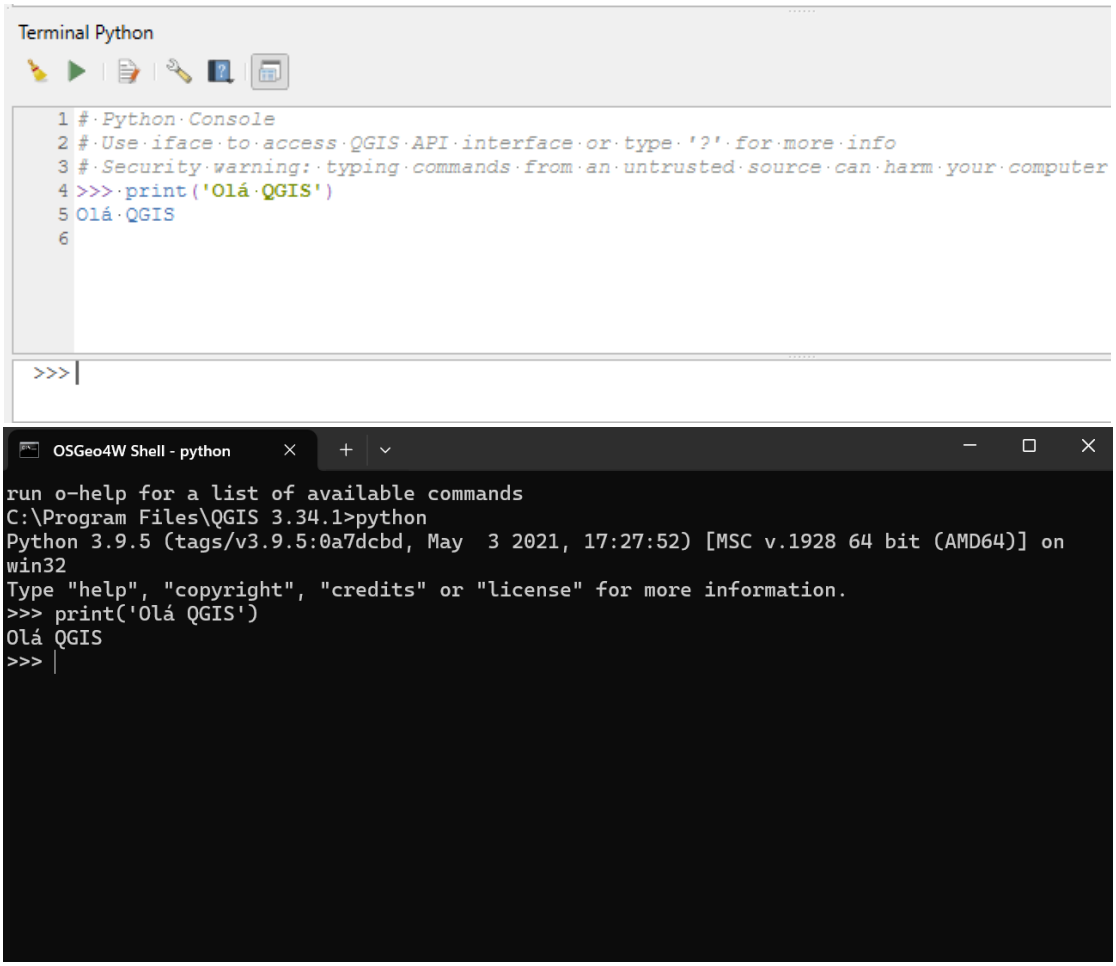
Dados

Esta publicação e os dados usados estão disponíveis em:

https://gdatasystems.com/pyqgis/pyQGISC.pdf
https://gdatasystems.com.br/pyqgis/index.php

VISITE:

https://gdatasystems.com

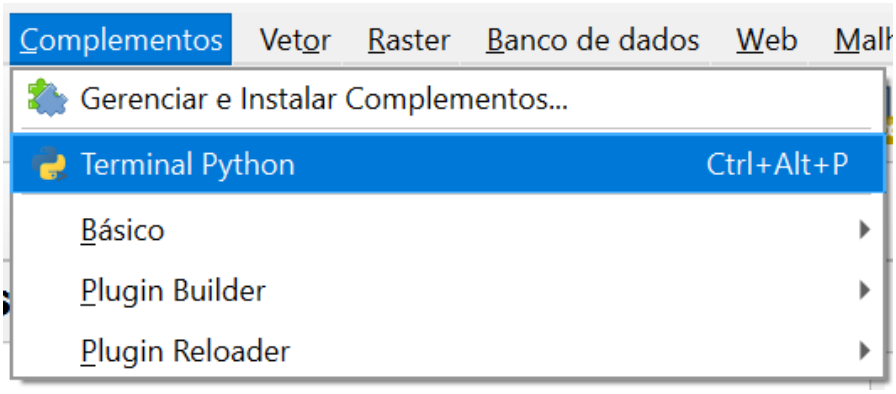


Aprendendo Python no QGIS

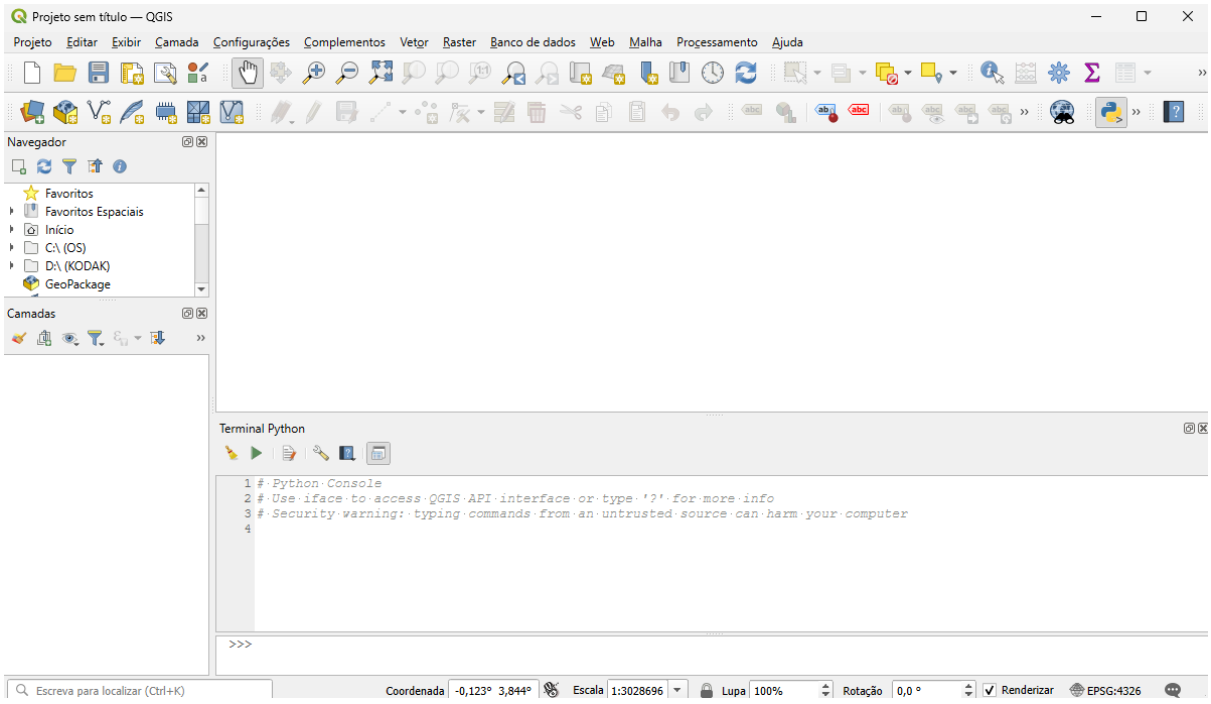
Usando o Terminal Python ou o Terminal OSGeo4W

1 – O Terminal Python

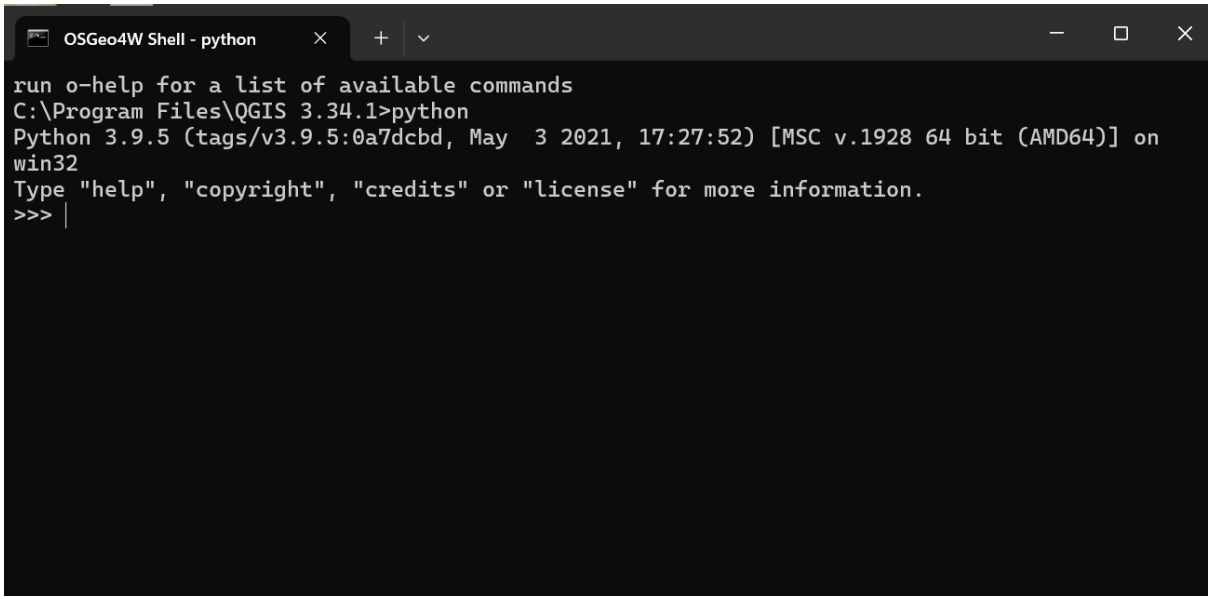
Para iniciar o terminal python no QGIS vá em Complementos→Termina python ou pressione **Ctrl-Alt-P**.



Veremos na parte inferior da tela o Terminal python aberto.



Esse é um tutorial de introdução à linguagem python que usa o terminal python do QGIS como nosso ambiente de aprendizado. Caso prefira, pode também utilizar o terminal diretamente via programa OSGeo4W Shell e digitando python para inicializar o terminal.



2 - Fundamentos da linguagem Python

Comentários de script iniciam com # em python.

```
>>> # um comentário
>>> print('QGIS') # outro comentário
QGIS
>>> a ='# isso não é um comentário pois está entre aspas'
```

Tipos Numéricos

Python possui primitivamente os tipos numéricos de números inteiros (int), ponto flutuante (float) e complexo (complex).

```
>>> a = 1
>>> b = 3.2
>>> c = 4 + 3j
```

A função type(variável) retorna o tipo da variável.

```
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'complex'>
```

O resultado de uma divisão sempre será do tipo float.

```
>>> d = a/a
>>> d
1.0
>>> type(d)
<class 'float'>
```

O console pode funcionar como uma calculadora também.

```
>>> 2+2
4
>>> 2**8 #dois elevado a oito
256
>>> 82%3 #resto inteiro da divisão
1
>>> 82//3 #quociente da divisão
27
>>> 3-2
1
>>> (50-5*6)/4 # Parênteses tem precedente na operação
5.0
```

Tipo Alfanumérico

Em python o tipo str é usado para palavras, textos e caracteres alfanuméricos.

```
>>> nome = 'Manuel'
>>> type(nome)
<class 'str'>
>>> letra = 'a'
>>> type(letra)
<class 'str'>
```

Um objeto da classe str nada mais é do que uma matriz de valores sequenciados onde o

primeiro valor (caractere) corresponde ao índice 0 da matriz e o último valor ao índice -1, o penúltimo -2 e assim sucessivamente.

```
>>> nome[0]
'M'
>>> nome[1]
'a'
>>> nome[-1]
'l'
>>> nome[-2]
'e'
>>> nome[-6]
'M'
>>> nome[2:5] #note que o valor do índice 5 não está incluído
'nue'
```

Um objeto str uma vez definido é imutável e se tentarmos mudar o valor de um objeto str uma mensagem de erro aparecerá.

```
>>> nome[2]='t'
Traceback (most recent call last):
File "/usr/lib/python3.7/code.py", line 90, in runcode
exec(code, self.locals)
File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

A função len() retorna o comprimento do objeto str.

```
>>> palavra='anticonstitucionalissimamente'
>>> len(palavra)
29
```

Podemos usar aspas simples ou duplas para delimitar um objeto str. Isso é prático para para podermos definir strings com estas:

```
>>> s1 = "Bom dia senhor O'Brien"
>>> print(s1)
Bom dia senhor O'Brien
>>> s2= 'Quoth the raven "Nevermore". '
>>> print(s2)
Quoth the raven "Nevermore".
```

Tipo lista

Python possui diversos tipos compostos de dados, a lista (list) é uma delas.
Uma lista é um conjunto de objetos de determinado tipo ou de tipos distintos armazenados em uma lista.
Listas são declaradas dentro de colchetes onde cada objeto (item) dela é separado por vírgula.

```
>>> lista = [3200,2670,3100,3000]
>>> caipi = ['gelo', 'limão','açúcar',51,15.99]
```

Assim como str, cada item de uma lista pode ser acessado usando índices.

```
>>> lista[0]
3200
>>> lista[1]
2670
>>> lista[3]
```

```
3000
>>> lista[1:3]
[2670, 3100]
>>> type(caipi [0])
<class 'str'>
>>> type(caipi [3])
<class 'int'>
>>> type(caipi [4])
<class 'float'>
>>> caipi [-1]
15.99
```
Ao contrário do objeto str, os valores de itens de uma lista podem ser modificados
```
>>> caipi[3]='pinga'
>>> caipi
['gelo', 'limão', 'açúcar', 'pinga', 15.99]
```
Uma lista pode conter outras listas como itens. E acessamos cada item dessa lista interna usando um índice adicional.
```
>>> caipi[1]=['abacaxi','maracujá','limão']
>>> caipi
['gelo', ['abacaxi', 'maracujá', 'limão'], 'açúcar', 'pinga', 15.99]
>>> caipi[1][1]
'maracujá'
>>> caipi[1][-1]
'limão'
```
Podemos adicionar itens a uma lista já existente usando adição ou a função append().
```
>>> caipi = caipi + ['guardanapo','canudo']
>>> caipi.append('copo')
>>> caipi
['gelo', ['abacaxi', 'maracujá', 'limão'], 'açúcar', 'pinga', 15.99,'guardanapo','canudo', 'copo']
```
Alguns métodos de interação com listas são mostrados abaixo:
Criamos a seguinte lista vazia inicialmente.
```
>>> lis=[]
>>> lis
[]
```
Adicionando itens na lista com o método extend().
```
>>> lis.extend([1,2,3])
>>> lis
[1, 2, 3]
```
Adicionando um item de valor 10 na posição predeterminada (0) com o método insert().
```
>>> lis.insert(0,10)
>>> lis
[10, 1, 2, 3]
```
Removendo da lista a primeira ocorrência do valor passado pelo método remove().
```
>>> lis.remove(2)
>>> lis
[10, 1, 3]
```
Podemos copiar uma lista usando o método copy().
```
>>> lis2=lis.copy()
>>> lis2
[10, 1, 3]
```

Revertemos a ordem dos itens de uma lista com o método reverse().

```
>>> lis2.reverse()
>>> lis2
[3, 1, 10]
```

Ordenamos uma lista usando o método sort().

```
>>> lis2.sort()
>>> lis2
[1, 3, 10]
```

O método pop() remove e retorna o item indicado pelo índice dado, se nenhum índice é fornecido o último item é removido da lista e retornado.

```
>>> lis3 = [1,1,2,3,4,4,4,3,2,3,1]
>>> lis3.pop(1)
1
>>> lis3
[1, 2, 3, 4, 4, 4, 3, 2, 3, 1]
```

O método index(x[,início[,fim]]) retorna o índice (na base 0) da primeira ocorrência do item x. O segundo argumento mostra a partir de qual e até qual índice da lista procurar. Caso não encontre um item com o valor uma mensagem de erro é retornada.

```
>>> lis3.index(1,1,10)
9
>>> lis3.index(1)
0
>>> lis3.index(1,1)
9
>>> lis3.index(1,1,10)
9
>>> lis3.index(8,1,10)
Traceback (most recent call last):
File "C:/PROGRA~1/QGIS3~1.4/apps/Python37/lib/code.py", line 90, in
runcode
exec(code, self.locals)
File "<input>", line 1, in <module>
ValueError: 8 is not in list
```

O método count(x) retorna o número de vezes que o item x aparece na lista.

```
>>> lis3.count(4)
3
```

O método clear() remove todos os itens da lista.

```
>>> lis3.clear()
>>> lis3
[]
```

Podemos usar a instrução del para deletar itens de uma lista usando índices em vez de valores.

```
>>> lis3 = [1,1,2,3,4,4,4,3,2,3,1]
>>> del lis3[2]
>>> lis3
[1, 1, 3, 4, 4, 4, 3, 2, 3, 1]
>>> del lis3[5:8]
>>> lis3
[1, 1, 3, 4, 4, 3, 1]
```

Tuples

Assim como str e listas, tuples são dados em sequência usados em python. As diferenças principais entre uma tuple e uma lista é que tuples são declaradas usando vírgulas para separar os itens e esses itens são imutáveis.

```
>>> t = 'banana', 3,45, False, "oi!"
>>> t
('banana', 3, 45, False, 'oi!')
>>> t[0]
'banana'
>>> doist =t,(1,2,3,4,5)
>>> doist
(('banana', 3, 45, False, 'oi!'), (1, 2, 3, 4, 5))
>>> doist[0]
('banana', 3, 45, False, 'oi!')
>>> doist[0][0]
'banana'
>>> doist[1][0]
1
>>> t[0]='maçã'
Traceback (most recent call last):
File "C:/PROGRA~1/QGIS3~1.4/apps/Python37/lib/code.py", line 90, in
runcode
exec(code, self.locals)
File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Sets

Sets ou conjuntos são outro tipo de dados em sequência que python utiliza. Sets são definidos dentro de chaves {} e resultam em uma simples aparição de cada um de seus itens e estes não são indexados.

```
>>> conjunto = {'rio','terra','fogo','fogo','água','terra'}
>>> conjunto
{'terra', 'fogo', 'rio', 'água'}
>>> 'rio' in conjunto
True
>>> 'mar' in conjunto
False
>>> num ={1,2,3,2,3,2,1,23,'a'}
>>> num
{1, 2, 3, 'a', 23}
>>> cidade=set('Pindamonhongaba')
>>> cidade
{'p', 'a', 'h', 'm', 'g', 'n', 'i', 'o', 'b', 'd'}\
```

Dicionários

Dicionários são um tipo de dados bastante usado em python. Um dicionário possui sempre uma chave e um valor, esta chave é usada no lugar de um índice numérico que

é usado numa lista. Essa chave deve ser única e um dicionário vazio pode ser criado usando {}.

```
>>> dicio={'nome':'andre','sobrenome':'costa'}
>>> dicio['idade']=56
>>> dicio
{'nome': 'andre', 'sobrenome': 'costa', 'idade': 56}
>>> list(dicio)
['nome', 'sobrenome', 'idade']
>>> sorted(dicio)
['idade', 'nome', 'sobrenome']
>>> dicio[1]
Traceback (most recent call last):
File "C:/PROGRA~1/QGIS3~1.4/apps/Python37/lib/code.py", line 90, in
runcode
exec(code, self.locals)
File "<input>", line 1, in <module>
KeyError: 1
>>> del dicio['idade']
>>> dicio
{'nome': 'andre', 'sobrenome': 'costa'}
>>> dicio2=dict([('código', 34), ('senha', 65483), ('acessos', 8)])
>>> dicio2
{'código': 34, 'senha': 65483, 'acessos': 8}
```

Funções Internas

As seguintes funções internas são usadas para conversão de tipos e informações de tipos.

```
>>> f=-5.789
>>> round(f,2)
-5.79
>>> abs(f)
5.789
>>> int(f)
-5
>>> str(f)
'-5.789'
>>> lis=[2,3,45,12,78,1,-17,3]
>>> min(lis)
-17
>>> max(lis)
78
>>> sum(lis)
127
>>> sorted(lis)
[-17, 1, 2, 3, 3, 12, 45, 78]
>>> i=255
>>> hex(i)
'0xff'
>>> bin(i)
'0b11111111'
```

```
>>> float(i)
255.0
>>> chr(i)
'ÿ'
>>> ord('ÿ')
255
>>> oct(i)
'0o377'
>>> ascii(i)
'255'
>>> pow(2,10)
1024
>>> bool(1<2 and 3>2)
True
>>> bool(1<2 and 3>4)
False
```

As seguintes palavras são reservadas da linguagem python e não podem ser usadas para definir variáveis.

and	**except**	**lambda**	**with**
as	**finally**	**nonlocal**	**while**
assert	**False**	**None**	**yield**
break	**for**	**not**	
class	**from**	**or**	
continue	**global**	**pass**	
def	**if**	**raise**	
del	**import**	**return**	
elif	**in**	**True**	
else	**is**	**try**	

E essas são as funções internas:

abs()	**delattr()**	**hash()**	**memoryview()**	**set()**
all()	**dict()**	**help()**	**min()**	**setattr()**
any()	**dir()**	**hex()**	**next()**	**slice()**
ascii()	**divmod()**	**id()**	**object()**	**sorted()**
bin()	**enumerate()**	**input()**	**oct()**	**staticmethod()**
bool()	**eval()**	**int()**	**open()**	**str()**
breakpoint()	**exec()**	**isinstance()**	**ord()**	**sum()**
bytearray()	**filter()**	**issubclass()**	**pow()**	**super()**
bytes()	**float()**	**iter()**	**print()**	**tuple()**
callable()	**format()**	**len()**	**property()**	**type()**
chr()	**frozenset()**	**list()**	**range()**	**vars()**
classmethod()	**getattr()**	**locals()**	**repr()**	**zip()**
compile()	**globals()**	**() reversed()**	**__import__()**	
complex()	**hasattr()**	**max()**	**round()**	

2 - Controles de fluxo

Como toda linguagem de programação, python utiliza controles de fluxo de programa. Mas antes disso vamos falar um pouco de recuo de script. Obrigatoriamente em python temos de usar recuo de blocos de código uma vez que não os separamos por chaves {}, assim todo bloco de código, seja ele uma classe, função ou um controle de fluxo, deve ser indentado. Num bloco com recuo no console, >>> se transforma em ... indicando que estamos dentro de um bloco no script. Quando todas as instruções do bloco estão finalizadas, teclamos ‘enter’ na linha final com ...

if elif else

O if talvez seja a instrução mais conhecida em programação. Ela checa se (if) uma condição ou se outras condições (elif) são atendidas. Se nenhuma condição for atendida, podemos também instruir que algo seja feito (else).

```
>>> x = int(input("Diga um número inteiro: "))
```

Diga um número inteiro: 2

```
>>> if x<0:
... print('O número é negativo')
... elif x>0:
... print('O número é positivo')
... else:
... print('O número é zero')
...
```

O número é positivo

for

A repetição (loop) for é definida como “executar/repetir as instruções nos termos predefinidos”. No exemplo abaixo a repetição é feita para cada palavra da variável palavras e a palavra e o comprimento dela é impresso como resultado.

```
>>> palavras = ['Olá!','Vamos','aprender','python?']
>>> for palavra in palavras:
... print(palavra,len(palavra))
...
```

Olá! 4
Vamos 5
aprender 8
python? 7

while

A repetição while é definida com “enquanto o que foi predefinido não ocorrer, vai executando/repetindo”.

```
>>> a, b = 0, 1
>>> while a < 1000:
... print(a, end=',')
... a, b = b, a+b
...
```

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,

range()

A forma que python interage com uma série numérica é usando a função range().

```
>>> for i in range(10):
... print(i,end=',')
...
```

0,1,2,3,4,5,6,7,8,9,

Vamos supor que não sabemos a quantidade de itens numa lista mas queremos interagir (no caso listar) cada um destes itens. Usamos range para nos auxiliar.

```
>>> palavras = ['Olá!','Vamos','aprender','python?']
>>> for i in range(len(palavras)):
... print(palavras[i],i)
...
Olá! 0
Vamos 1
aprender 2
python? 3
```

Podemos definir o início e final de range() e o passo também.

```
>>> list(range(50,57))
[50, 51, 52, 53, 54, 55, 56]
>>> list(range(10,5,-1))
[10, 9, 8, 7, 6]
>>> list(range(-100,-50,10))
[-100, -90, -80, -70, -60]
```

Break, continue e else em loops

A instrução break quebra a execução da repetição for ou while mais interna.
Repetições também podem ter uma instrução else; ela é executada quando a repetição termina mas não quando ela é interrompida por uma instrução break. Veja o exemplo abaixo:

```
>>> for n in range(2,10):
... for x in range(2, n):
... if n % x == 0:
... print(n, 'é igual a ', x, '*', n//x)
... break
... else:
... print(n, 'é um número primo')
...
2 é um número primo
3 é um número primo
4 é igual a 2 * 2
5 é um número primo
6 é igual a 2 * 3
7 é um número primo
8 é igual a 2 * 4
9 é igual a 3 * 3
```

A instrução continue, avança para a próxima interação da repetição:

```
>>> for num in range(2, 6):
... if num % 2 == 0:
... print("Número par", num)
... continue
... print("Número ímpar", num)
...
Número par 2
Número ímpar 3
Número par 4
Número ímpar 5
```

3 - Funções

Uma função é um conjunto de instruções organizadas e reusáveis para executar uma tarefa. Em python definimos uma função usando def seguido do nome da função e dos argumentos ou parâmetros passados dentro de parênteses. Veja o exemplo abaixo:

```
>>> def fibo(n):
... """Calcula a série de Fibonacci até o limite informado"""
... a,b=0,1
... while a<n:
... print(a,end=' ')
... a,b=b,a+b
... print()
...
>>> fibo(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>> fibo(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo(10000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765
```

Uma outra forma de escrever essa função, utilizando retornando no formato de lista, é mostrada a seguir:

```
>>> def fibo2(n):
... """Calcula e retorna a série de Fibonacci até o limite
informado"""
... resultado=[]
... a,b=0,1
... while a<n:
... resultado.append(a)
... a,b=b,a+b
... return resultado
...
>>> fibo2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Podemos assinalar uma função a uma variável.

```
>>> f=fibo2
>>> f(10)
[0, 1, 1, 2, 3, 5, 8]
```

Se passarmos nossa função como argumento para a função help teremos a string de documentação (entre as aspas duplas repetidas três vezes) como informação retornada. Essa é uma maneira de documentar o que uma função faz em python.

```
>>> help(fibo2)
Help on function fibo2 in module __main__:
fibo2(n)
Calcula e retorna a série de Fibonacci até o limite informado
```

É possível definir funções com número de argumentos variáveis.

Valor padrão predefinido é a forma mais comum onde podemos predefinir o valor de um ou mais parâmetros de uma função.

```
>>> def pergunta(texto, tentativas=4, msg='Tente de novo!'):
... while True:
... ok = input(texto)
... if ok in ('Sim', 'sim', 's', 'S'):
```

```
... return True
... if ok in ('n', 'não', 'N', 'Não', 'nao', 'Nao'):
... return False
... tentativas = tentativas - 1
... if tentativas < 0:
... raise ValueError('Resposta Inválida.')
... print(msg)
...
>>> pergunta('Está com fome?')
Está com fome?j
Tente de novo!
Esta com fome?Sim
True
>>> pergunta('Está com fome?',1,'Não entendi a resposta!')
Está com fome?iop
Não entendi a resposta!
Está com fome?poi
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 10, in pergunta
ValueError: Resposta Inválida.
```

Vamos ver agora um exemplo prático de como trabalhar com funções criando um nosso primeiro script. Crie o arquivo **temp_converter.py** com o seguinte código.

```
#!/usr/bin/env python3
'''Converte temperaturas de Celsius para Fahrenheit, Kelvin para
Celsius e Kelvin
para Fahrenheit.
Uso:
Carregue usando import
import temp_converter as tc
Autor:
Seu Nome - 03.12.2019'''
def celsius_para_fahr(temp_celsius):
   return 9/5 * temp_celsius + 32
def kelvins_para_celsius(temp_kelvins):
   return temp_kelvins - 273.15
def kelvins_para_fahr(temp_kelvins):
   temp_celsius = kelvins_para_celsius(temp_kelvins)
   temp_fahr = celsius_para_fahr(temp_celsius)
   return temp_fahr
```

Agora vamos importar nosso script e usar funções dele.

```
>>> import temp_converter as tc
>>> print("O ponto de congelamento da água em Fahrenheint é:",
tc.celsius_para_fahrenheint(0))
O ponto de congelamento da água em Fahrenheit é: 32.0
>>> print('O Zero absoluto em Fahrenheit é:',
tc.kelvins_para_fahr(0))
O Zero absoluto em Fahrenheit é: -459.66999999999996
```

4 - Módulos

Em python um module (módulo) é simplesmente um arquivo com extensão .py com classes, funções e demais instruções. Nosso script acima é um exemplo de um módulo. Um package (pacote) é uma forma de organizar vários módulos em uma entidade maior. Algumas linguagens chamam módulos e pacotes de library. Um módulo é carregado usando o comando **`import`** e podemos renomear um módulo usando **`as`**

```
>>> import math as m
>>> m.sqrt(81)
9
```

Também podemos importar uma simples função de um módulo usando from

```
>>> from math import sqrt
>>> sqrt(9)
3
```

Ou podemos importar um submódulo de um módulo.

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
<Figure size 432x288 with 0 Axes>
```

Usaremos o tempo todo vários módulos, essa é e força da linguagem python com inúmeros módulos existentes para as mais diversas funções. Vamos iniciar vendo os módulos pandas e matplotlib.

5 - Pandas

A biblioteca pandas foi desenvolvida como uma alternativa a linguagem R para lidar com estrutura de dados mais complexas. Hoje é uma biblioteca razoável largamente utilizadas em diversas áreas da ciência de dados. Panda tira vantagem em utilizar outra biblioteca chamada numpy escrita em C e portanto, bastante rápida e eficiente ao lidar com dados em grandes volumes. Os seguintes formatos podem ser importados e exportados usando pandas:

Formato	Tipo de dado	Le	Escreve
texto	CSV	read_csv	to_csv
texto	JSON	read_json	to_json
texto	HTML	read_html	to_html
texto	Local clipboard	read_clipboard	to_clipboard
binário	MS Excel	read_excel	to_excel
binário	HDF5 Format	read_hdf	to_hdf
binário	Feather Format	read_feather	to_feather
binário	Msgpack	read_msgpack	to_msgpack
binário	Stata	read_stata	to_stata
binário	SAS	read_sas	
binário	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Baixe o arquivo **assay.csv** e inicie o python.

Primeiramente vamos trabalhar com pandas lendo o conteúdo desse arquivo.

```
>>> import pandas as pd
>>> data = pd.read_csv('assay.csv')
>>> data.head()
 HOLEID FROM  TO  AU
0 M001  0.0  2.5 0.00
1 M001  2.5  5.0 0.71
2 M001  5.0  7.5 0.96
3 M001  7.5 10.0 0.48
4 M001 10.0 12.5 1.42]
>>> type(data)
<class 'pandas.core.frame.DataFrame'>
>>> len(data)
17664
>>> data.shape
(17664, 4)
>>> data.columns.values
array(['HOLEID', 'FROM', 'TO', 'AU'], dtype=object)
>>> data.dtypes
HOLEID   object
FROM    float64
TO      float64
AU      float64
dtype: object
```

Podemos selecionar colunas de dados da seguinte forma:

```
>>> selecao=data[['FROM','AU']]
```

```
>>> selecao.head()
  FROM   AU
0  0.0 0.00
1  2.5 0.71
2  5.0 0.96
3  7.5 0.48
4 10.0 1.42
```
Aplicando estatística descritiva nos dados
```
>>> selecao.mean()
pressao_min 940.133461
FROM   159.362913
AU     0.673208
dtype: float64
>>> selecao.max()
FROM   497.5
AU     30.5
dtype: float64
>>> selecao.min()
FROM   0.0
AU     0.0
dtype: float64
>>> selecao.median()
FROM   144.40
AU     0.34
dtype: float64
>>> selecao.std()
FROM   113.517974
AU     1.138792
dtype: float64
>>> selecao.describe()
                FROM             AU
count  17664.000000  17664.000000
mean     159.362913      0.673208
std      113.517974      1.138792
min        0.000000      0.000000
25%       65.000000      0.100000
50%      144.400000      0.340000
75%      227.500000      0.802500
max      497.500000     30.500000
```
Vamos agora usar pandas para fazer o caminho inverso, de lista de dados para arquivo
Um arquivo chamado estacoes.csv.
```
>>> estacao=['E-01','E-02','E-03','E-04','E-05','E-06']
>>> latitude=[-3.34,-3.23,-3.12,-3.32,-3.33,-3.19]
>>> longitude=[-60.12,-60.43,-60.11,-60.54,-59.87,-60.00]
>>> dadoEst = pd.DataFrame(data = {"Estação" : estacao, "latitude"
:latitude,"longitude" : longitude})
>>> dadoEst
Estação latitude longitude
0 E-01 -3.34 -60.12
```

```
1 E-02 -3.23 -60.43
2 E-03 -3.12 -60.11
3 E-04 -3.32 -60.54
4 E-05 -3.33 -59.87
5 E-06 -3.19 -60.00
>>> dadoEst.to_csv('estacoes.csv')
```

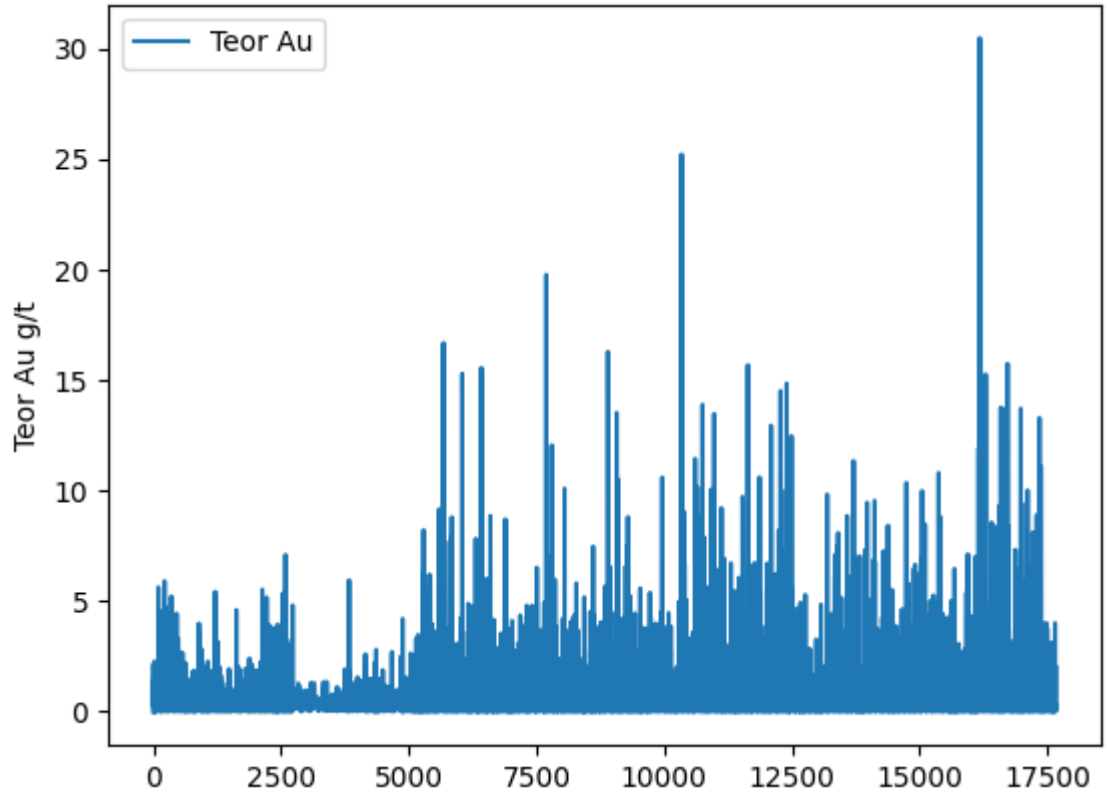
Por último, criando um data frame vazio.

```
>>> df = pd.DataFrame()
>>> print(df)
Empty DataFrame
Columns: []
Index: []
```

6 - Matplotlib

Vamos mostrar simplificadamente como podemos criar gráficos com python. Existem diversas bibliotecas para a criação de gráficos mas aqui vamos usar o matplotlib com o auxílio de pandas para criar alguns gráficos básicos.

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> data = pd.read_csv('assay.csv')
>>> plt.plot(data[['AU']], label='Teor Au')
[<matplotlib.lines.Line2D object at 0x7f55bbc70d90>]
>>> plt.ylabel('Teor Au g/t')
Text(0, 0.5, 'Precipitação mm')
>>> plt.legend()
<matplotlib.legend.Legend object at 0x7f55bcee2b50>
>>> plt.show()
```



Vamos usar o numpy para criar uma série sequencial de números entre 0 e 2 e plotar a sequência linear, quadrática e cúbica da série para ilustrar como criar um gráfico com mais de uma curva.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> x = np.linspace(0, 2, 100)
>>> plt.plot(x, x, label='linear')
>>> plt.plot(x, x**2, label='quadrática')
>>> plt.plot(x, x**3, label='cúbica')
>>> plt.ylabel('Eixo Y')
>>> plt.xlabel('Eixo X')
>>> plt.title("Gráfico Simples")
>>> plt.legend()
```

```
>>> plt.show()
```

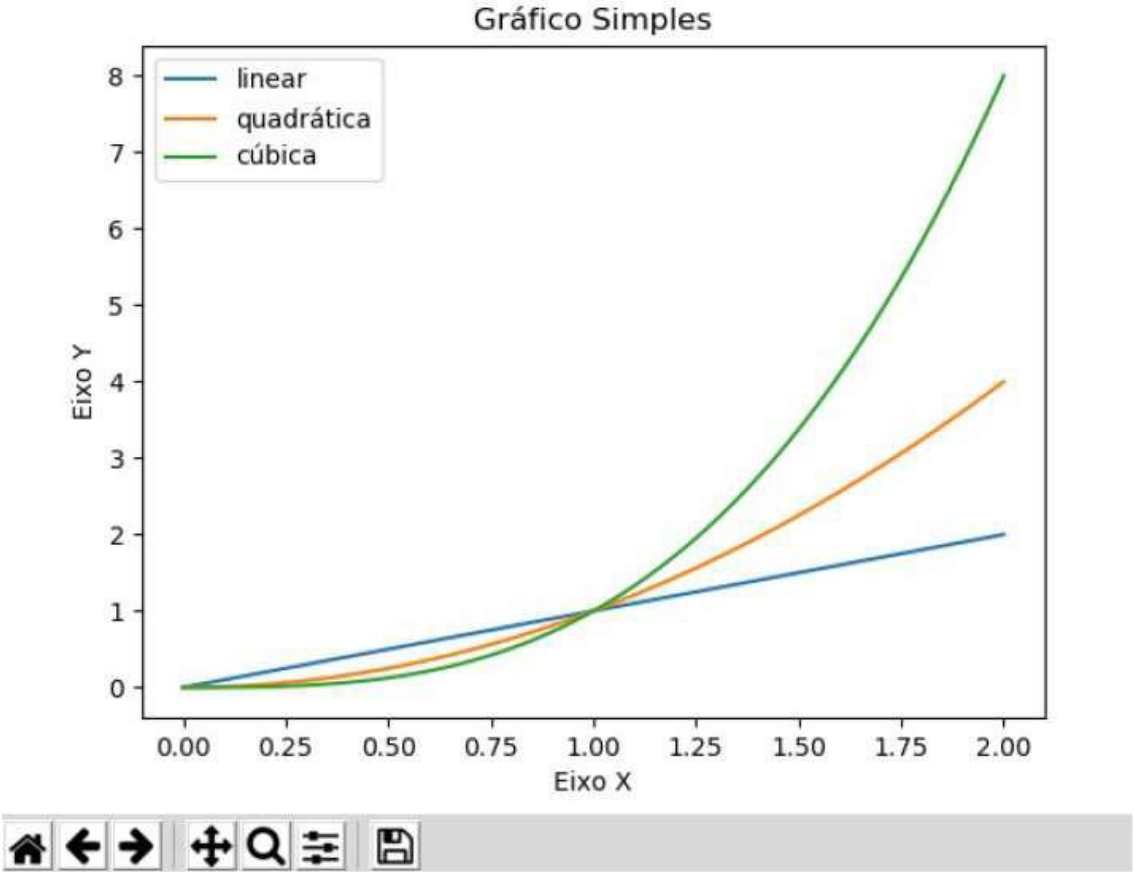
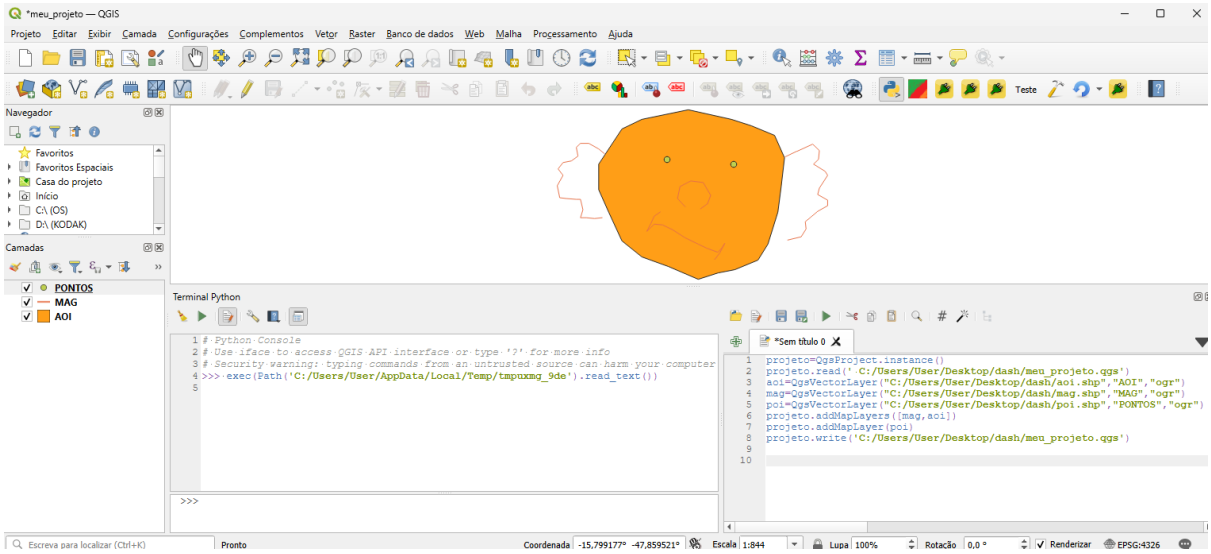


Gráfico Simples
linear
quadrática
cúbica
Eixo Y
Eixo X
0
1
2
3
4
5
6
7
8
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

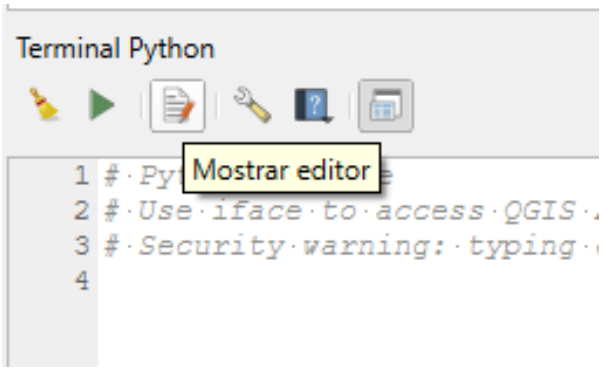


Usando pyQGIS

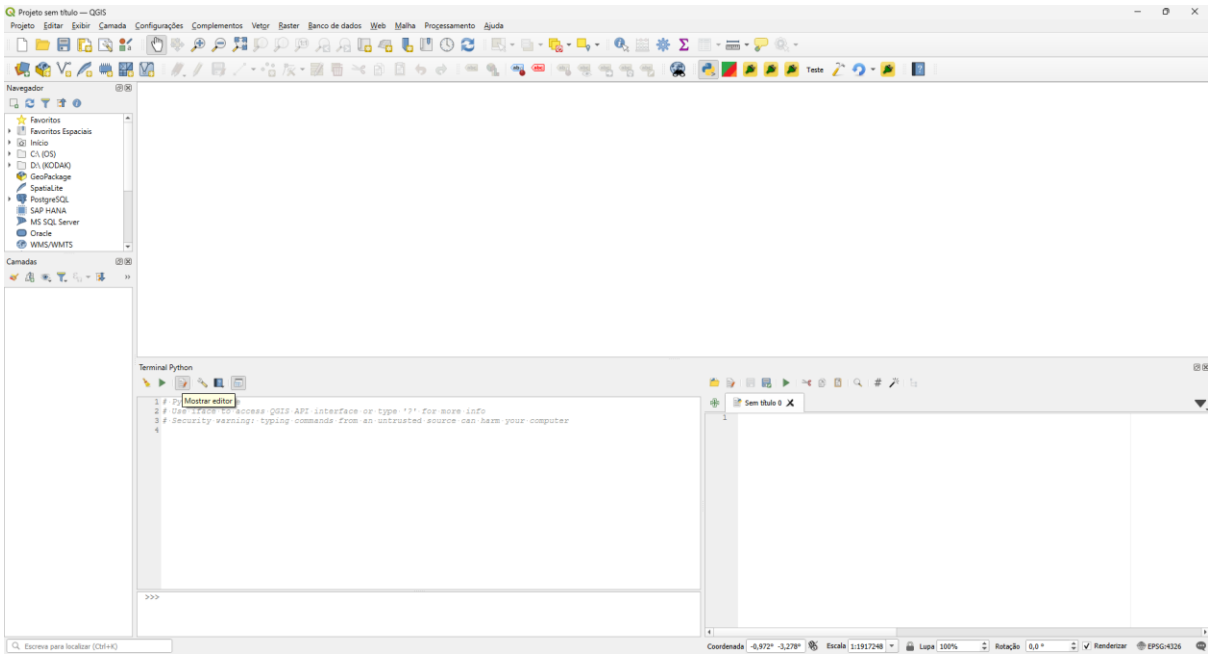
Introdução às Primeiras Classes

Preparando o terreno

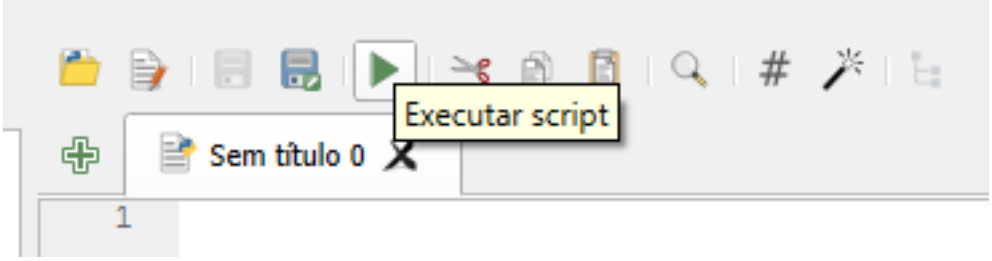
Vamos abrir o QGIS e deixar tudo preparado para podermos operar o ambiente usando o terminal python (CTRL+ALT+P) e clique no botão Mostrar editor:



Teremos a tela desta forma.



Estamos prontos para trabalhar. No editor na parte inferior direita escreveremos o código e para executar executaremos com:



Classes do pyQGIS

Vamos introduzir as classes do PyQGIS na medida que avançamos nos pontos cobertos.

Uma classe em python é a definição de um tipo de objeto e de métodos (funções) associados a este objeto. Por exemplo um projeto, uma camada raster, uma camada vetorial, etc.

A biblioteca PyQGIS é bastante extensa com diversas classes. Vamos aqui cobrir algumas classes mais básicas e essenciais para a partir deste ponto termos uma boa base para desenvolver scripts mais complexos.

Classe Projeto (*QgsProject*) - Criar e ler um Projeto

Um projeto armazena um conjunto de informações sobre camadas, estilos, layouts, anotações etc. Como se trata de uma classe singleton, criamos um objeto usando o método QgsProject.instance(). Vamos mostrar como criar um projeto vazio chamado meu_projeto.qgs usando o método write().

```
projeto= QgsProject.instance()
# ajuste caminho para o arquivo a ser criado no seu sistema
projeto.write('C:/Users/User/Desktop/dash/meu_projeto.qgs')
print(projeto.fileName())
C:/Users/User/Desktop/dash /meu_projeto.qgs
```

Agora vamos sair do QGIS e entrar novamente para carregarmos o projeto que criamos usando python.

```
projeto=QgsProject.instance()
projeto.read(' C:/Users/User/Desktop/dash/meu_projeto.qgs')
print(projeto.fileName())
C:/Users/User/Desktop/dash /meu_projeto.qgs
```

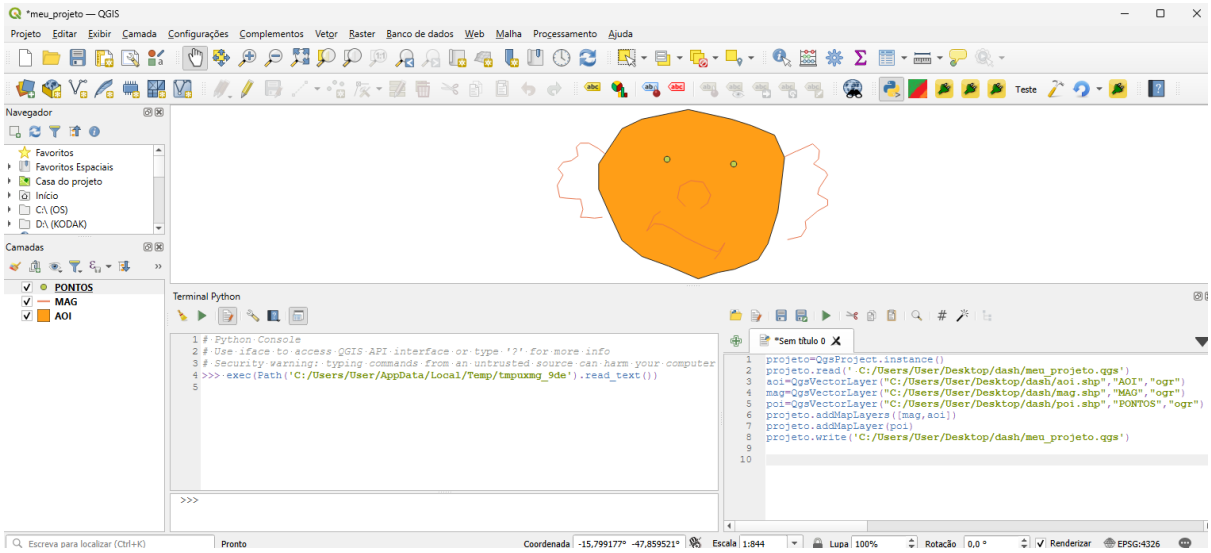
Na medida que formos vendo as outras classes, vamos ver outros métodos associados à classe Projeto.

Classe Vetor (*QsgVectorLayer*) – Adicionando Camada Vetorial

Um objeto do tipo camada vetorial é usado para carregarmos e interagirmos com camadas do tipo vetor. Vamos carregar o nosso projeto e adicionar nele três camadas vetor criadas anteriormente usando os métodos de projeto addMapLayer e addMapLayers. Criamos um objeto de classe camada vetorial usando o método QgsVectorLayer() passando o caminho para o arquivo vetorial, o identificador que nossa camada terá, e a biblioteca a ser usada (ogr nesse caso). Por último salvamos o nosso projeto com o método write().

```
projeto=QgsProject.instance()
projeto.read(' C:/Users/User/Desktop/dash/meu_projeto.qgs')
aoi=QgsVectorLayer("C:/Users/User/Desktop/dash/aoi.shp","AOI","ogr")
mag=QgsVectorLayer("C:/Users/User/Desktop/dash/mag.shp","MAG","ogr")
poi=QgsVectorLayer("C:/Users/User/Desktop/dash/poi.shp","PONTOS","ogr")
projeto.addMapLayers([mag,aoi])
projeto.addMapLayer(poi)
projeto.write('C:/Users/User/Desktop/dash/meu_projeto.qgs')
```

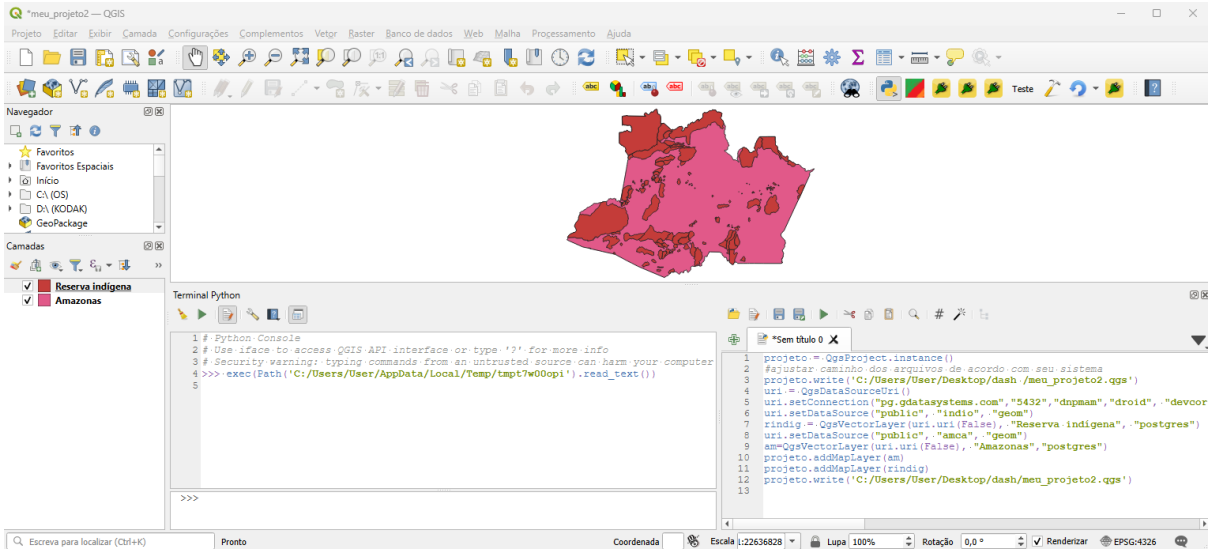
Algo similar ao apresentado abaixo deverá aparecer:



Agora vamos fazer uso de uma outra classe para podermos carregar uma camada vetorial localizada em um banco de dados Postgis remoto. A classe é a QgsDataSourceUri e usaremos os métodos setConnection() e setDataSource() para extrairmos uma tabela espacial vetorial. Criaremos um projeto novo, adicionaremos uma camada local e uma camada remota Postgis e por último vamos gravar o projeto.

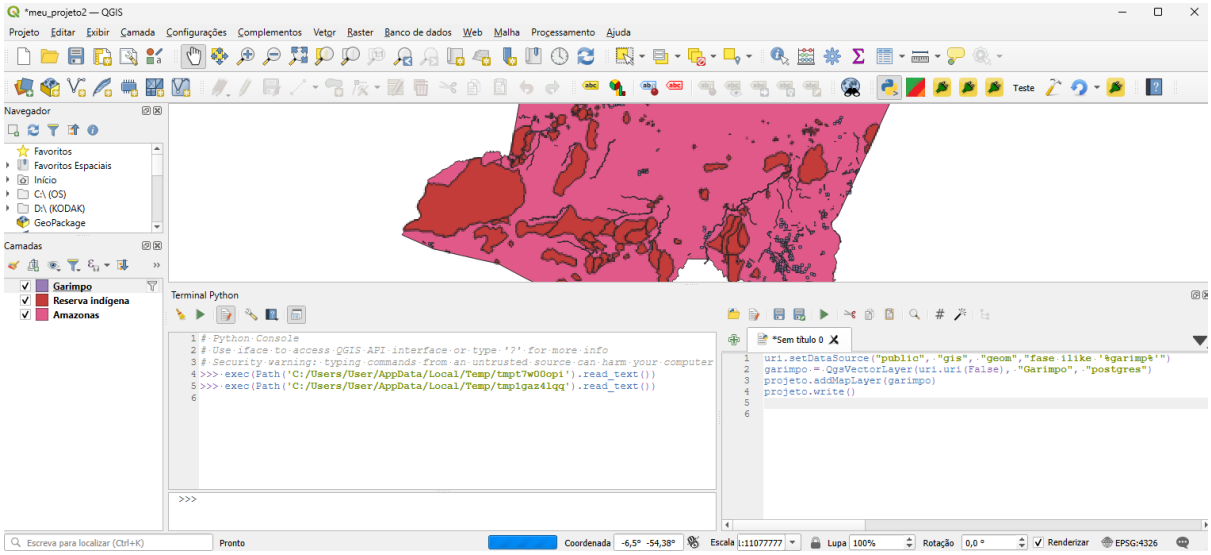
```
projeto = QgsProject.instance()
#ajustar caminho dos arquivos de acordo com seu sistema
projeto.write('C:/Users/User/Desktop/dash /meu_projeto2.qgs')
uri = QgsDataSourceUri()
uri.setConnection("pg.gdatasystems.com","5432","dnpmam","droid", "devcor")
uri.setDataSource("public", "indio", "geom")
rindig = QgsVectorLayer(uri.uri(False), "Reserva indígena", "postgres")
uri.setDataSource("public", "amca", "geom")
am=QgsVectorLayer(uri.uri(False), "Amazonas","postgres")
projeto.addMapLayer(am)
projeto.addMapLayer(rindig)
projeto.write('C:/Users/User/Desktop/dash/meu_projeto2.qgs')
```

O método setConnection() tem como parâmetros o endereço do servidor (IP ou DNS), a porta (geralmente 5432), o banco de dados, o usuário e a senha. O método setDataSource() tem como parâmetros o esquema da tabela, o nome da tabela e a coluna com o elemento geométrico espacial. Um projeto conforme o ilustrado abaixo deverá aparecer.



Alternativamente podemos adicionar uma cláusula SQL WHERE como o quarto argumento. Vamos ver um exemplo onde extraímos uma camada vetorial somente os requerimentos de garimpo e permissão de lavra garimpeira dos requerimentos do estado do Amazonas e adicionamos ela no nosso projeto já criado acima.

```
uri.setDataSource("public", "gis", "geom","fase ilike '%garimp%'")
garimpo = QgsVectorLayer(uri.uri(False), "Garimpo", "postgres")
projeto.addMapLayer(garimpo)
projeto.write()
```



Antes de movermos para o próximo tópico vamos dar uma olhada em alguns métodos da Classe Projeto (QsgProject) relacionados a classe Camadas Vetoriais (QsgVectorLayer).

count retorna o número de camadas válidas do projeto.

```
projeto.count()
```

3

mapLayers retorna um mapa das camadas existentes do projeto.

```
projeto.mapLayers()
```

{'Amazonas_741f8310_bd3a_4b38_bc38_0a1147ea2769': <QgsVectorLayer: 'Amazonas' (postgres)>, 'Garimpo_d47cdf32_a1fb_4c37_8145_7127996261f2': <QgsVectorLayer: 'Garimpo' (postgres)>, 'Reserva_ind_gena_362f9fab_b918_42e4_94d9_e92fb105bead': <QgsVectorLayer: 'Reserva indígena' (postgres)>}

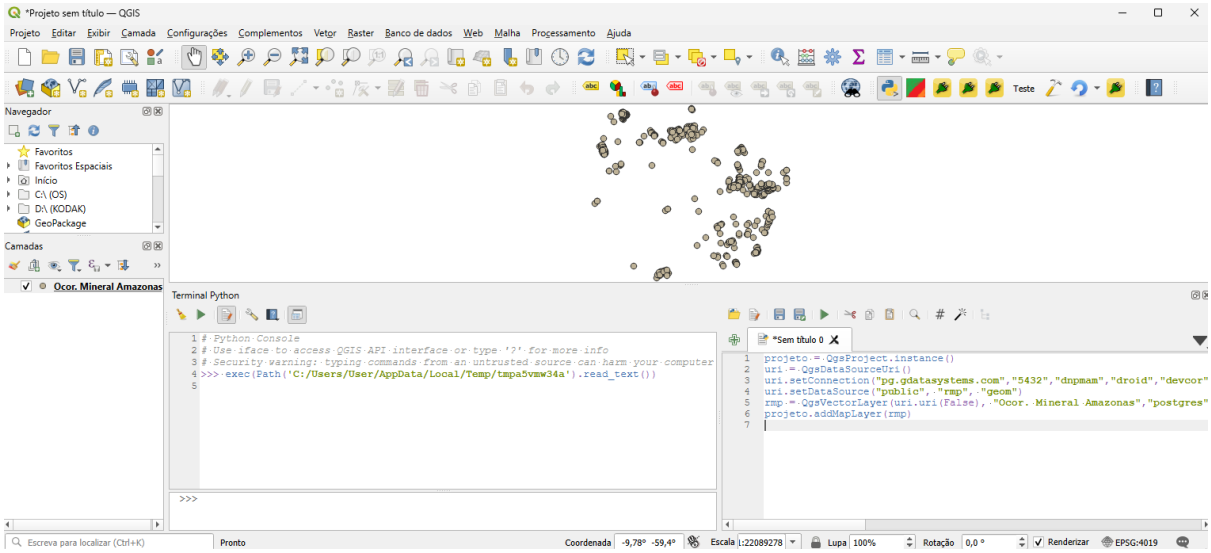
Interagindo com informações de objetos da classe Vector

Podemos obter diversas informações sobre objetos vetoriais tais como, projeções, extensão, número de elementos, valores e nomes dos campos de atributos (colunas) e até criar um metadata da camada (com a informação existente).

Vamos primeiro carregar um dado do tipo ponto de um banco Postgis remoto.

```
projeto = QgsProject.instance()
uri = QgsDataSourceUri()
uri.setConnection("pg.gdatasystems.com","5432","dnpmam","droid","devcor")
uri.setDataSource("public", "rmp", "geom")
rmp = QgsVectorLayer(uri.uri(False), "Ocor. Mineral Amazonas","postgres")
projeto.addMapLayer(rmp)
```

A camada será carregada conforme a ilustração mostrada abaixo.



Vamos agora acessar as informações mais usadas de maneira geral. Outras informações existem no objeto camada, veja a documentação para mais detalhes.

O sistema de referência de coordenadas CRS (Coordinate Reference System)

O método crs() retorna o sistema de referência de coordenada original do objeto camada que o invoca.

```
crs=rmp.crs()
print(crs.description())
```
Unknown datum based upon the GRS 1980 ellipsoid

A extensão da Camada

Com o método extent() de um objeto camada podemos obter os valores máximos e mínimos das coordenadas em X (Easting ou Longitude) e Y (Northing ou Latitude). O método retorna um objeto do tipo retângulo com diversos parâmetros além de X e Y máximos e mínimos tais como area, width, height, center, invert, etc. Veja a documentação para mais informações.

```
extensão=rmp.extent()
min_x=extensão.xMinimum()
max_x=extensão.xMaximum()
min_y=extensão.yMinimum()
max_y=extensão.yMaximum()
print(min_x,min_y,max_x,max_y)
-70.1 -9.53860000030878 -56.7497 2.21390000007294
```

Quantidade de itens

O método featureCount() retorna quantos itens o objeto camada possui.

```
num_elementos=rmp.featureCount()
print("número de elementos: ", num_elementos)
número de elementos: 624
```

Obtendo informações dos campos de atributos

Com o método fields() obtemos a informação sobre todos os campos de atributos tais como nome, tipo etc.

```
for field in rmp.fields():
    print (field.name(),field.typeName())
codigo_obj text
TOPONIMIA text
latitude float8
longitude float8
SUBST_PRIN text
subst_sec text
abrev text
STATUS_ECO text
grau_de_im text
metodo_geo text
erro_metod text
data_cad text
classe_uti text
tipologia text
classe_gen text
modelo_dep text
assoc_geoq text
rocha_enca text
rocha_hosp text
textura_mi text
tipos_alte text
extrmin_x_ text
```

assoc_mine text
origem text
municipio text
uf text

Metadata de camada vetorial

O método htmlMetadata() gera um metadata da camada no formato html que pode ser copiado para um novo arquivo e visualizado em um navegador da web;

```
metadata=rmp.htmlMetadata()
print (metadata)
```

```
< html>
<body>
<h1>Informação do provedor</h1>
<hr>
<table class="list-view">
<tr><td class="highlight">Nome</td><td>Ocor. Mineral
Amazonas</td></tr>
<tr><td class="highlight">fonte</td><td>dbname='dnpmam'
host=pg.amazeone.com.br port=5432 user='droid' key='tid'
checkPrimaryKeyUnicity='1' table="public"."rmp" (geom)</td></tr>
<tr><td class="highlight">Armazenamento</td><td>PostgreSQL database
with PostGIS extension</td></tr>
<tr><td class="highlight">Comentário</td><td></td></tr>
<tr><td class="highlight">Codificação</td><td></td></tr>
<tr><td class="highlight">Geometria</td><td>Point (Point)</td></tr>
<tr><td class="highlight">SRC</td><td>EPSG:4019 - Unknown datum based
upon the GRS 1980 ellipsoid - Geográfico</td></tr>
<tr><td class="highlight">Extensão</td><td>-70.0999999999999943,-
9.5386000003087794 : -56.7496999999999971,2.2139000000729401</td></tr>
<tr><td class="highlight">Unidade</td><td>graus</td></tr>
<tr><td class="highlight">Contagem de feições</td><td>624</td></tr>
</table>
<br><br><h1>Identificação</h1>
<hr>
<table class="list-view">
<tr><td class="highlight">Identifier</td><td></td></tr>
<tr><td class="highlight">Parent Identifier</td><td></td></tr>
<tr><td class="highlight">Title</td><td></td></tr>
<tr><td class="highlight">Type</td><td>dataset</td></tr>
<tr><td class="highlight">Language</td><td></td></tr>
<tr><td class="highlight">Abstract</td><td></td></tr>
<tr><td class="highlight">Categories</td><td></td></tr>
<tr><td class="highlight">Keywords</td><td>
</td></tr>
</table>
<br><br>
<h1>Extensão</h1>
<hr>
<table class="list-view">
<tr><td class="highlight">CRS</td><td>EPSG:4019 - Unknown datum based
upon the GRS 1980 ellipsoid - Geographic</td></tr>
<tr><td class="highlight">Spatial Extent</td><td></td></tr>
<tr><td class="highlight">Temporal Extent</td><td></td></tr>
</table>
<br><br>
<h1>Acesso</h1>
<hr>
<table class="list-view">
<tr><td class="highlight">Fees</td><td></td></tr>
<tr><td class="highlight">Licenses</td><td></td></tr>
<tr><td class="highlight">Rights</td><td></td></tr>
```

<tr><td class="highlight">Constraints</td><td></td></tr>
</table>

<h1>Campos</h1>
<hr>
<table class="list-view">
<tr><td class="highlight">Contagem</td><td>26</td></tr>
</table>

<table width="100%" class="tabular-view">
<tr><th>Campo</th><th>Tipo</th><th>Comprimento</th><th>Precisão</th><t
h>Comentário</th></tr>
<tr ><td>codigo_obj</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>TOPONIMIA</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>latitude</td><td>float8</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>longitude</td><td>float8</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>SUBST_PRIN</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>subst_sec</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>abrev</td><td>text</td><td>-1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>STATUS_ECO</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>grau_de_im</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>metodo_geo</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>erro_metod</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>data_cad</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>classe_uti</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>tipologia</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>classe_gen</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>modelo_dep</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>assoc_geoq</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>rocha_enca</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>rocha_hosp</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>textura_mi</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>tipos_alte</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>extrmin_x_</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>assoc_mine</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>origem</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr ><td>municipio</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
<tr class="odd-row"><td>uf</td><td>text</td><td>-
1</td><td>0</td><td></td></tr>
</table>

<h1>Contatos</h1>
<hr>
No contact yet.


```
<h1>Links</h1>
<hr>
<p>No links yet.</p>
<br><br>
<h1>Histórico</h1>
<hr>
<p>No history yet.</p>
<br><br>
</body>
</html>
```

Essa informação em HTML acima apareceria em um navegador assim:

Informação do provedor

Nome	Ocor. Mineral Amazonas
fonte	dbname='dnpmam' host=pg.amazeone.com.br port=5432 user='droid' key='tid' checkPrimaryKeyUnicity='1' table="public"."rmp" (geom)
Armazenamento	PostgreSQL database with PostGIS extension
Comentário	
Codificação	
Geometria	Point (Point)
SRC	EPSG:4019 - Unknown datum based upon the GRS 1980 ellipsoid - Geográfico
Extensão	-70.0999999999999943,- 9.5386000003087794 : -56.7496999999999971,2.2139000000729401
Unidade	graus
Contagem de feições	624

Identificação

Identifier	
Parent Identifier	
Title	
Type	dataset
Language	
Abstract	
Categories	
Keywords	

Extensão

CRS	EPSG:4019 - Unknown datum based upon the GRS 1980 ellipsoid - Geographic
Spatial Extent	
Temporal Extent	

Acesso

Fees	
Licenses	
Rights	
Constraints	

Campos

Contagem 26

Comentário

Campo	Tipo	Comprimento	Precisão
codigo_obj	text	- 1	0
TOPONIMIA	text	- 1	0
latitude	float8	- 1	0
longitude	float8	- 1	0
SUBST_PRIN	text	- 1	0
subst_sec	text	- 1	0
abrev	text	-1	0
STATUS_ECO	text	- 1	0
grau_de_im	text	- 1	0
metodo_geo	text	- 1	0
erro_metod	text	- 1	0
data_cad	text	- 1	0
classe_uti	text	- 1	0
tipologia	text	- 1	0
classe_gen	text	- 1	0
modelo_dep	text	- 1	0
assoc_geoq	text	- 1	0
rocha_enca	text	- 1	0
rocha_hosp	text	- 1	0
textura_mi	text	- 1	0
tipos_alte	text	- 1	0
extrmin_x_	text	- 1	0
assoc_mine	text	- 1	0
origem	text	- 1	0
municipio	text	- 1	0
uf	text	- 1	0

Contatos

No contact yet.

Links

No links yet.

Histórico

No history yet.

Obtendo os elementos de cada item da camada vetorial

Com o método getFeatures() carregamos todos os dados da camada, onde cada item é armazenado como uma lista. O código abaixo imprime cada um dos itens em formato de lista.

```
elementos=rmp.getFeatures()
for e in elementos:
    attr=e.attributes()
    print (attr)
```

['25368', 'APUI', -7.7972, -58.8558, 'Calcário', NULL, 'cc', 'Não explotado', 'Depósito', 'Levantamento em Carta 1:250.000', '250 a 1.000 m', '2001/11/26', 'Insumos para agricultura', NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, '7', 'APUI', 'AM']

...

...

['46734', 'RIO MANICORÉ', -6.11, -61.5669, 'Argila', NULL, 'arg', '(Não determinado)', 'Depósito', 'GPS Manual pré 25/05/2000', '50 a 200 m', '2006/11/24', 'Material de uso na construção civil', NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, '2', 'MANICORE', 'AM']

3.4 Criando objeto vetorial

Vamos agora ver os passos para criarmos objetos vetoriais usando python no Qgis. Vamos criar objetos do tipo ponto, linha e polígono para ilustrar o processo.

Ponto

Primeiro definimos o objeto ponto com CRS 4326 (WGS84) com o nome Cidades na memória. Nesse objeto usamos um dataProvider para criar os campos de atributos do objeto vetorial ponto com três atributos (nome, população e IDH) e adicionamos eles no objeto ponto (vponto).

```
vponto = QgsVectorLayer("Point?crs=EPSG:4326", "Cidades","memory")
dPr = vponto.dataProvider()
dPr.addAttributes([QgsField("nome", QVariant.String),QgsField("populacao",
                        QVariant.Int), QgsField("idh", QVariant.Double)])
vponto.updateFields()
```

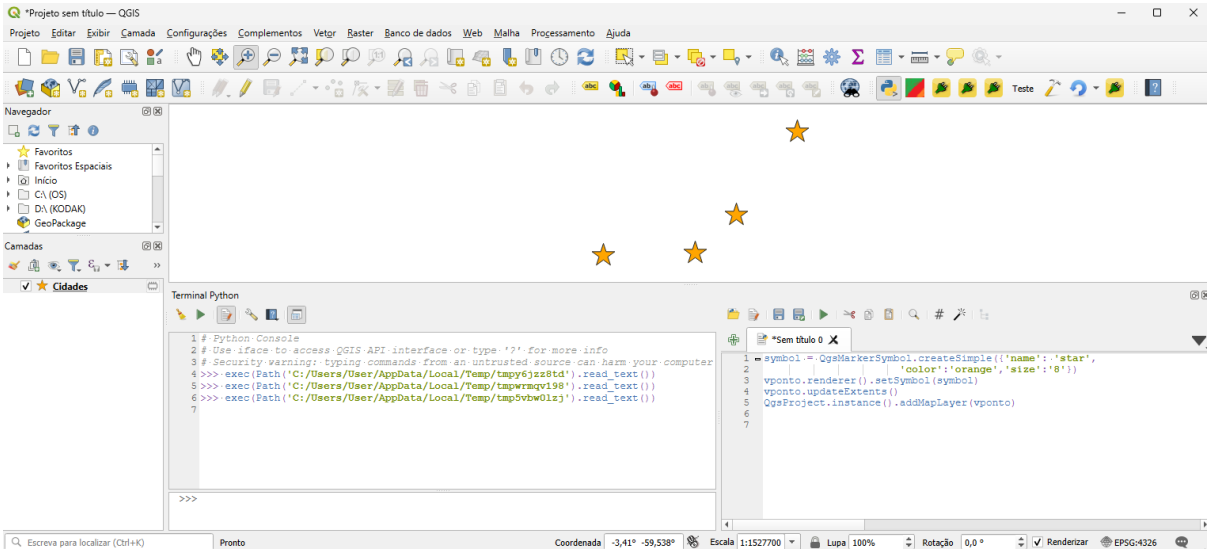
Uma vez criado o objeto ponto e seus campos de atributo vamos adicionar dados nele usando um objeto feature (elemento). Definimos a geometria que será um ponto nesse caso com coordenadas x e y e adicionaremos os atributos deste ponto.

```
elem = QgsFeature()
elem.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(-59.9936,-3.0925)))
elem.setAttributes(["Manaus", 2182763, 0.737])
dPr.addFeature(elem)
elem.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(-60.6253,-3.2872)))
elem.setAttributes(["Manacapuru ", 97377, 0.614])
dPr.addFeature(elem)
elem.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(-60.1883,-3.2756)))
elem.setAttributes(["Iranduba ", 48296, 0.613])
dPr.addFeature(elem)
elem.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(-59.7014,-2.6968)))
elem.setAttributes(["Rio Preto Da Eva ", 33347, 0.611])
dPr.addFeature(elem)
```

Para finalizar vamos configurar a aparência do símbolo mostrado no mapa como estrelas de cor laranjas e de tamanho 8. Atualizamos a extensão do mapa e adicionamos nosso objeto no mapa.

```
symbol = QgsMarkerSymbol.createSimple({'name': 'star',
                        'color':'orange','size':'8'})
vponto.renderer().setSymbol(symbol)
vponto.updateExtents()
QgsProject.instance().addMapLayer(vponto)
```

A aparência do mapa e de sua tabela de atributos será conforme a imagem mostrada abaixo.



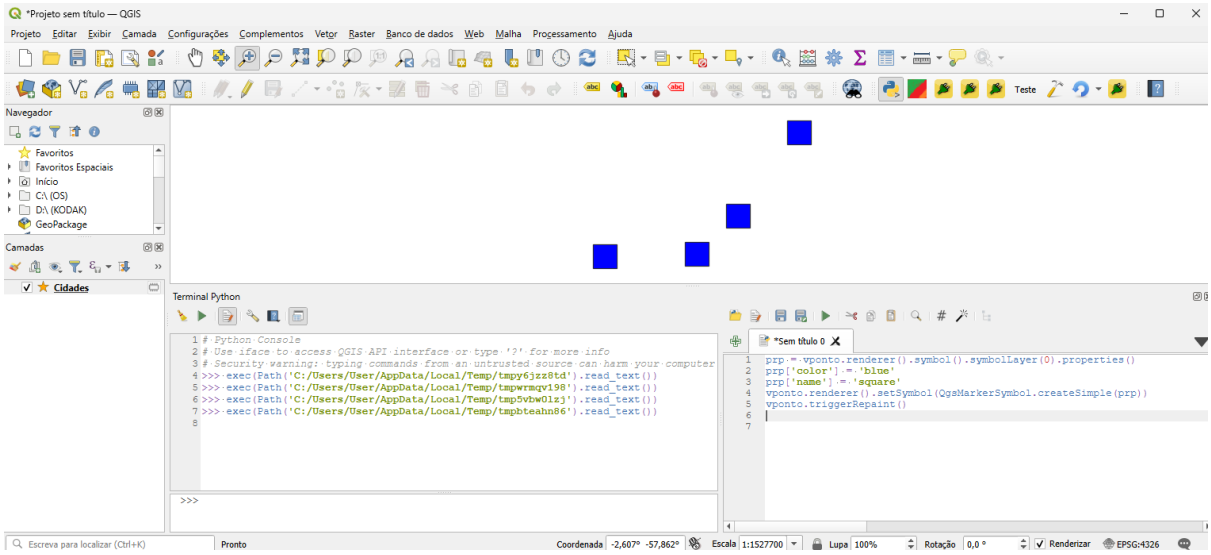
Os parâmetros abaixo podem ser usados com QgsMarkerSymbol.createSimple():

'angle': '0',
'color': '255,165,0,255',
'horizontal_anchor_point': '1',
'joinstyle': 'bevel',
'name': 'star',
'offset': '0,0',
'offset_map_unit_scale': '3x:0,0,0,0,0,0',
'offset_unit': 'MM',
'outline_color': '35,35,35,255',
'outline_style': 'solid',
'outline_width': '0',
'outline_width_map_unit_scale': '3x:0,0,0,0,0,0',
'outline_width_unit': 'MM',
'scale_method': 'diameter',
'size': '8',
'size_map_unit_scale': '3x:0,0,0,0,0,0',
'size_unit': 'MM',
'vertical_anchor_point': '1'

Veja a documentação para cada uma das opções que podem ser usadas com cada parâmetro listado acima. Vamos Modificar a aparência do símbolo que criamos acima.

```
prp = vponto.renderer().symbol().symbolLayer(0).properties()
prp['color'] = 'blue'
prp['name'] = 'square'
vponto.renderer().setSymbol(QgsMarkerSymbol.createSimple(prp))
vponto.triggerRepaint()
```

Veja o resultado abaixo.



Finalizamos mostrando como escrever o nosso objeto ponto em um arquivo.

```
QgsVectorFileWriter.writeAsVectorFormat(vponto,'C:/Users/User/Desktop/dash/cidaIDH.shp','utf-8', driverName='ESRI Shapefile')
```

Linha

De forma semelhante ao que fizemos com pontos, criar linhas usando python/Qgis é só uma questão de usarmos uma série (lista) de pontos para cada elemento criado.

Definimos o objeto linha (linestring) com CRS 4326 (WGS84) com o nome Vias na memória. Criamos o dataProvider para adicionar os campos de atributos do objeto vetorial linestring com dois atributos (nome e número) e adicionamos eles no objeto linestring (vlinha).

```
vlinha = QgsVectorLayer("Linestring?crs=EPSG:4326", "Vias","memory")
dPr = vlinha.dataProvider()
dPr.addAttributes([QgsField("nome",QVariant.String),QgsField("número",QVariant.Int)])
vlinha.updateFields()
```

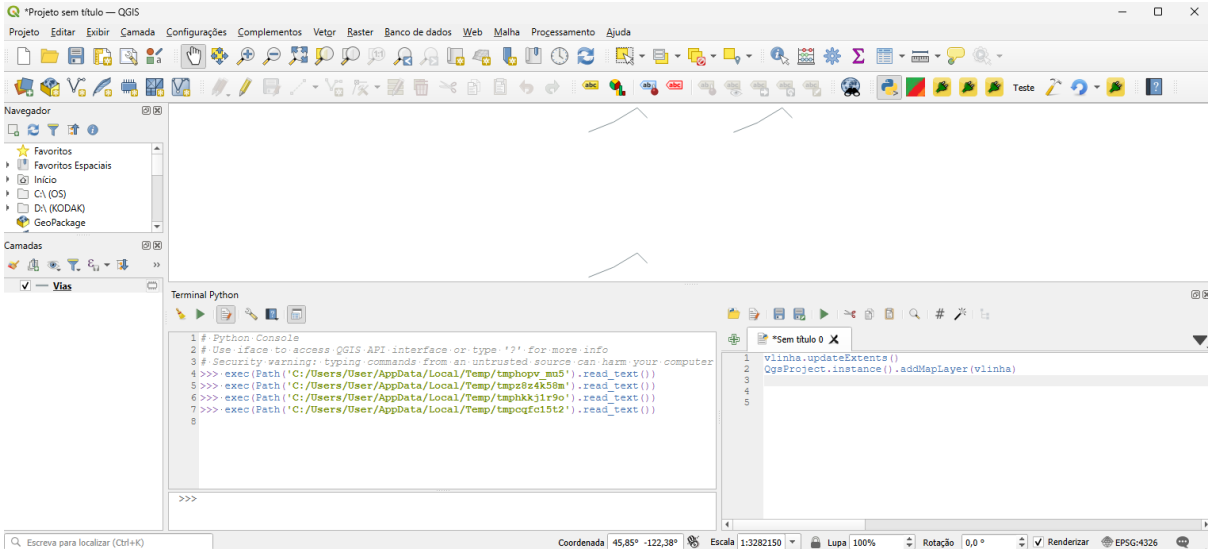
Adicionamos as linhas usando um objeto feature (elemento). Mas antes criamos a lista de pontos que farão parte de cada um dos elementos do tipo linha e inserimos os atributos de cada elemento. Vamos inserir três linhas.

```
elem = QgsFeature()
pontos =[QgsPoint(-124,48.4), QgsPoint(-123.5,48.6 ), QgsPoint(-123,48.9),QgsPoint(-122.8,48.7)]
elem.setGeometry(QgsGeometry.fromPolyline(pontos))
elem.setAttributes(["Rota ", 1])
dPr.addFeature(elem)
pontos =[QgsPoint(-121,48.4), QgsPoint(-120.5,48.6 ), QgsPoint(-120,48.9),QgsPoint(-119.8,48.7)]
elem.setGeometry(QgsGeometry.fromPolyline(pontos))
elem.setAttributes(["Rota ", 2])
dPr.addFeature(elem)
```

```
pontos =[QgsPoint(-124,45.4), QgsPoint(-123.5,45.6 ), QgsPoint(-
123,45.9),QgsPoint(-122.8,45.7)]
elem.setGeometry(QgsGeometry.fromPolyline(pontos))
elem.setAttributes(["Rota ", 3])
dPr.addFeature(elem)
```

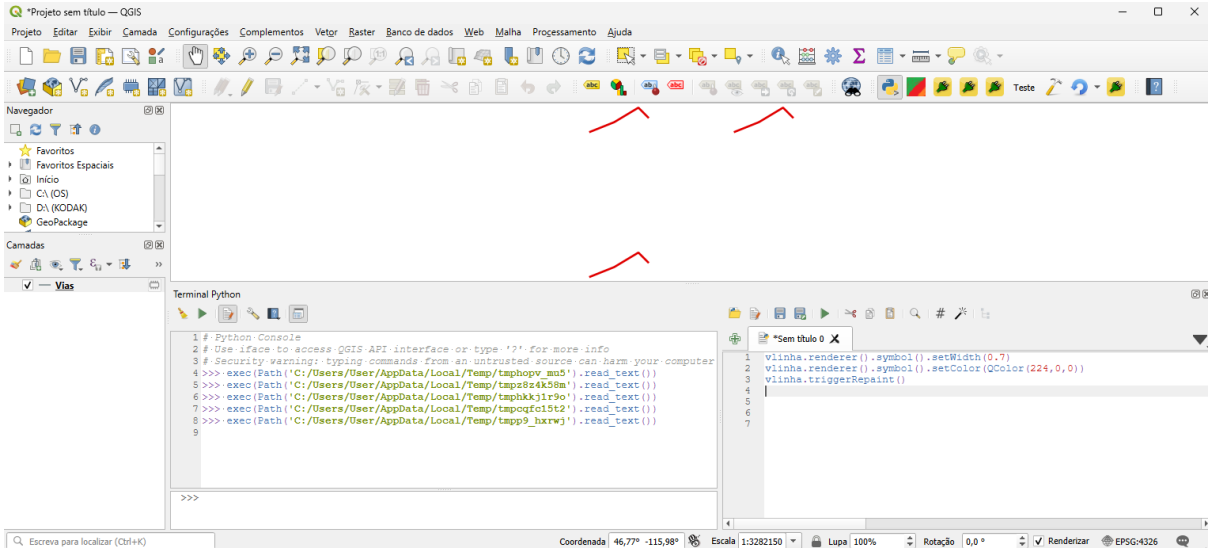
Atualizamos a extensão do objeto e o adicionamos ao mapa (canvas).

```
vlinha.updateExtents()
QgsProject.instance().addMapLayer(vlinha)
```



Podemos mudar a aparência de nosso objeto vetorial linestring usando:

```
vlinha.renderer().symbol().setWidth(0.7)
vlinha.renderer().symbol().setColor(QColor(224,0,0))
vlinha.triggerRepaint()
```



Para finalizar, vamos gravar o recém-criado vetor num arquivo do tipo shapefile.

```
QgsVectorFileWriter.writeAsVectorFormat(vlinha,'C:/Users/User/Deskto
p/dash/vias.shp', 'utf-8', driverName='ESRI Shapefile')
```

Polígono

Criamos polígonos usando python/Qgis de forma idêntica à forma que criamos linhas só que nesse caso o último ponto se liga ao primeiro ponto informado. Definimos o objeto polígono com CRS 4326 (WGS84) com o nome Fazendas na memória.

Criamos o dataProvider para adicionar os campos de atributos do objeto vetorial polígono com dois atributos (nome e número) e adicionamos eles no objeto polígono (vpgon).

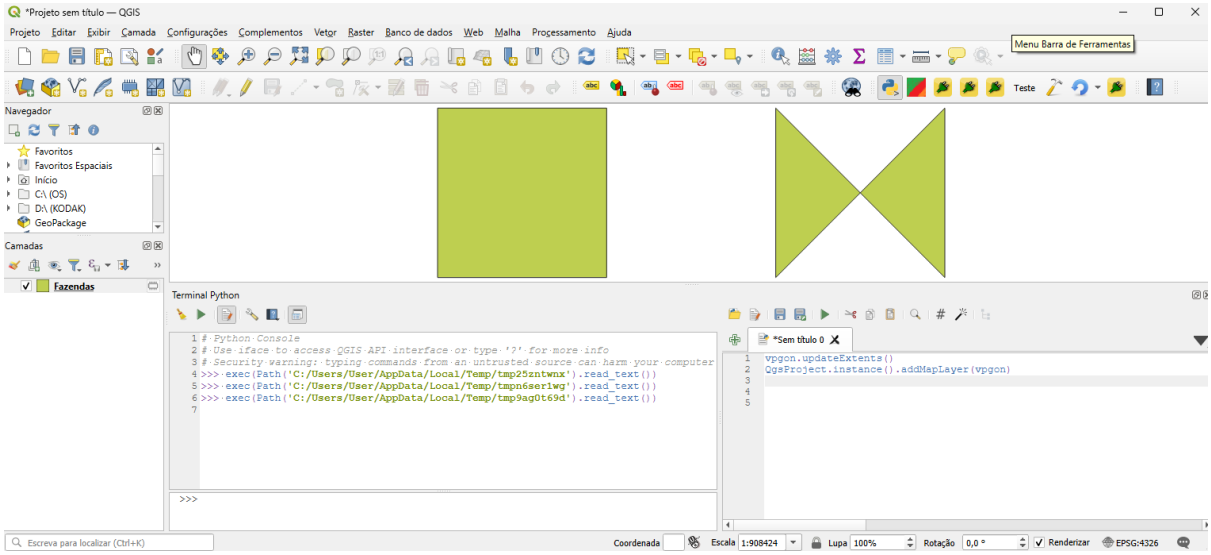
```
vpgon = QgsVectorLayer("Polygon?crs=EPSG:4326", "Fazendas","memory")
dPr = vpgon.dataProvider()
dPr.addAttributes([QgsField("nome",QVariant.String),QgsField("número
",QVariant.Int)])
vpgon.updateFields()
```

Adicionamos os polígonos usando um objeto feature (elemento). Mas antes criamos a lista de pontos que farão parte de cada um dos elementos do tipo polígono (em colchete duplo) e inserimos os atributos de cada elemento. Vamos inserir dois polígonos.

```
elem = QgsFeature()
pontos =[[QgsPointXY(-124,48), QgsPointXY(-123,48 ), QgsPointXY(-
123,49),QgsPointXY(-124,49)]]
elem.setGeometry(QgsGeometry.fromPolygonXY(pontos))
elem.setAttributes(["Fazenda Abre Campo ", 1])
dPr.addFeature(elem)
pontos =[[QgsPointXY(-122,48), QgsPointXY(-121,49 ), QgsPointXY(-
121,48),QgsPointXY(-122,49)]]
elem.setGeometry(QgsGeometry.fromPolygonXY(pontos))
elem.setAttributes(["Fazenda Vista Alegre ", 2])
dPr.addFeature(elem)
```

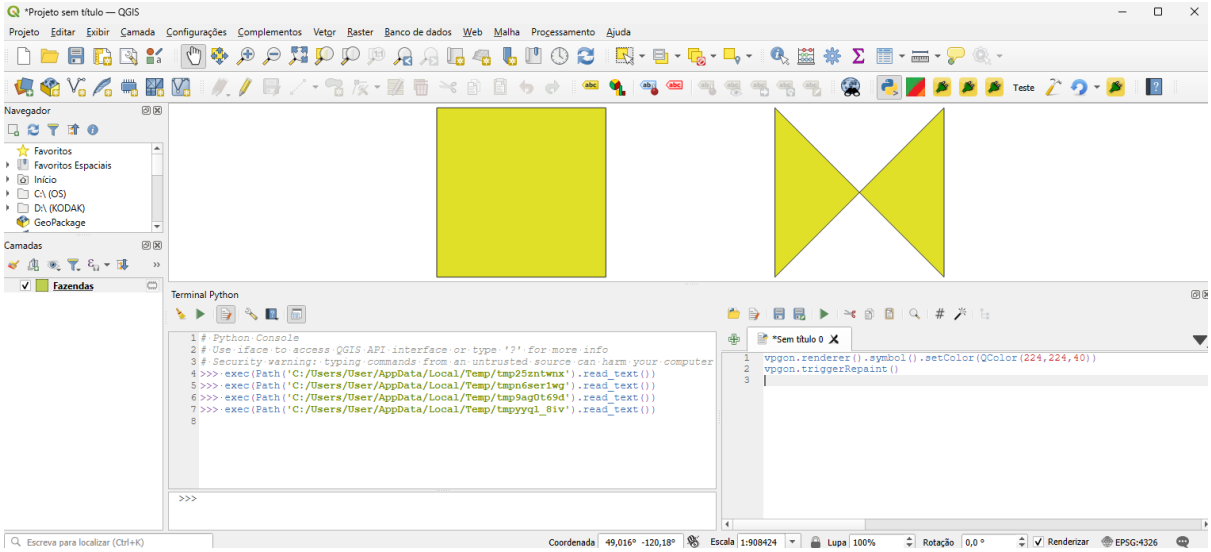
Atualizamos a extensão do objeto e o adicionamos ao mapa (canvas).

```
vpgon.updateExtents()
QgsProject.instance().addMapLayer(vpgon)
```



Podemos mudar a aparência de nosso objeto vetorial polígono usando:

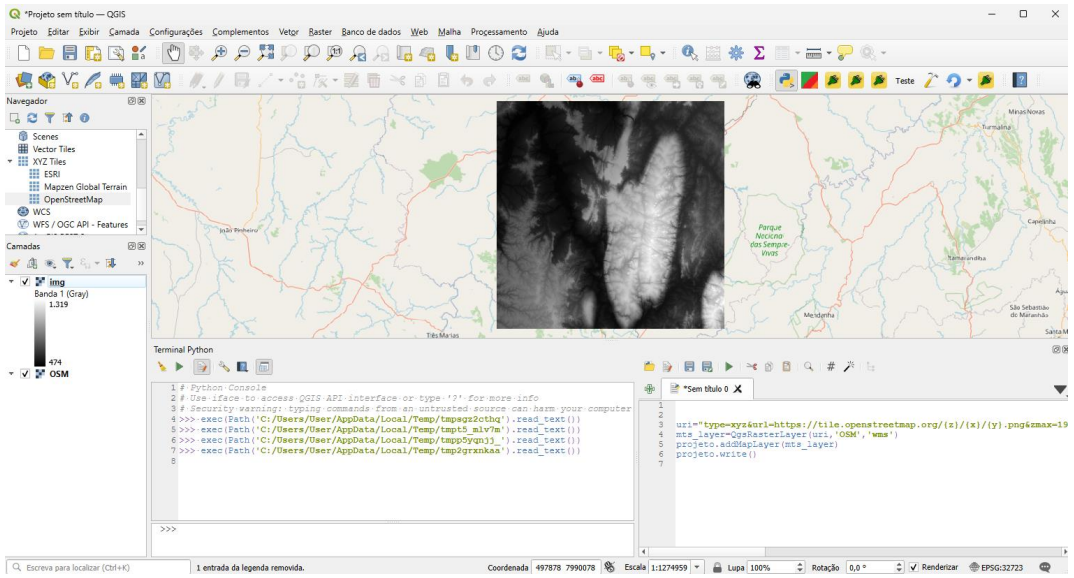
```
vpgon.renderer().symbol().setColor(QColor(224,224,40))
vpgon.triggerRepaint()
```

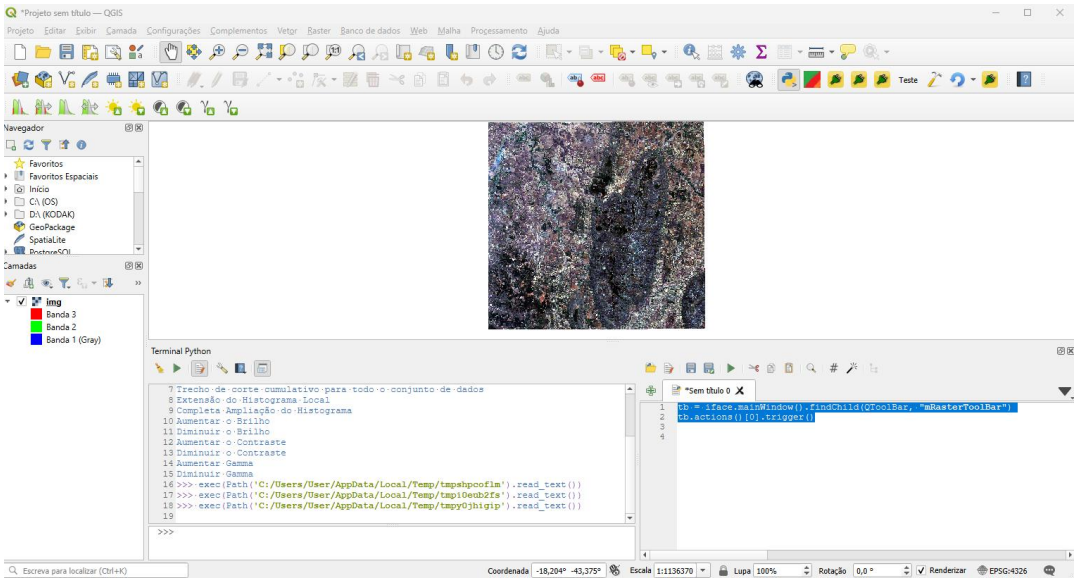


Gravamos o vetor recém-criado em um arquivo do tipo shapefile usando.

```
QgsVectorFileWriter.writeAsVectorFormat(vpgon,'C:/Users/User/Desktop
/dash/fazendas.shp','utf-8', driverName='ESRI Shapefile')
```

No próximo módulo veremos a classe ***QsgRasterLayer***.





Usando pyQGIS

Classe Raster - *QgsRasterLayer*

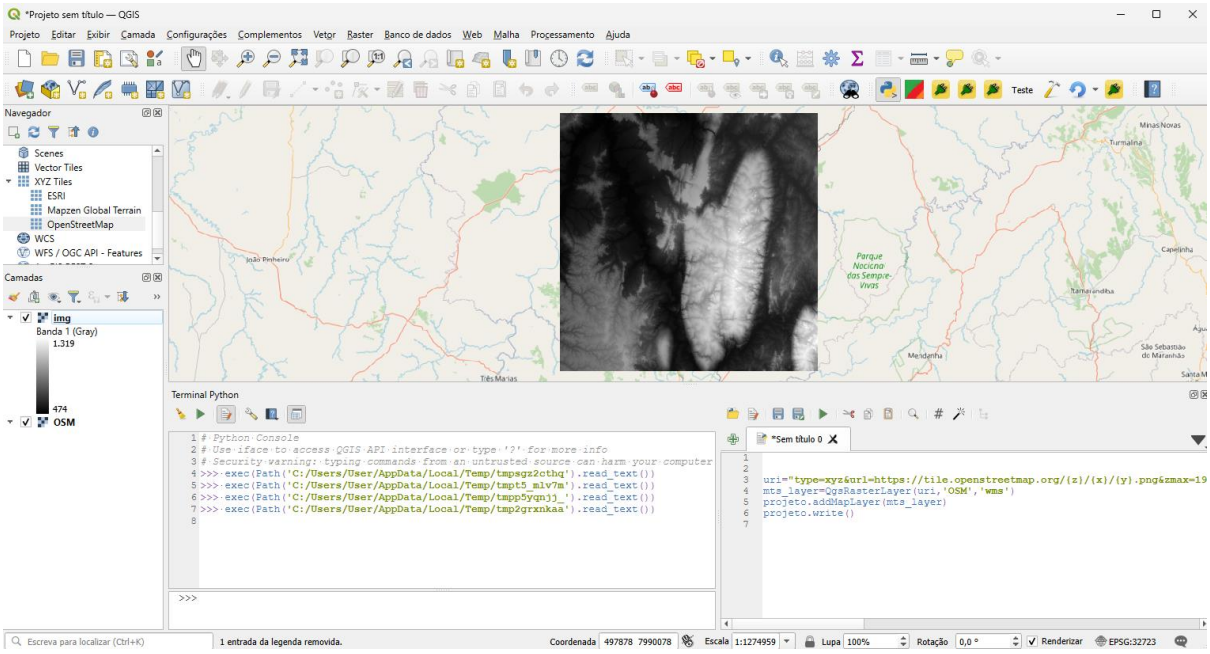
Classe Raster - *QgsRasterLayer*

Similar à forma que adicionamos camadas vetoriais, podemos adicionar imagens raster no nosso projeto usando objeto da a classe raster. Vamos criar um projeto e adicionar uma imagem raster nele.

```
projeto=QgsProject.instance()
camadaR =QgsRasterLayer("C:/Users/user/desktop/dash/dem2.tif","img")
projeto. addMapLayer(camadaR)
projeto.write('C:/Users/user/desktop/dash/meu_projeto3.qgs')
```

Podemos também adicionar dados do tipo raster usando provedores do tipo TMS (TileMapService) ou WMS (WebMapaService).

```
uri="type=xyz&url=https://tile.openstreetmap.org/{z}/{x}/{y}.png&zmax=19&zmin=0"
mts_layer=QgsRasterLayer(uri,'OSM','wms')
projeto.addMapLayer(mts_layer)
projeto.write()
```

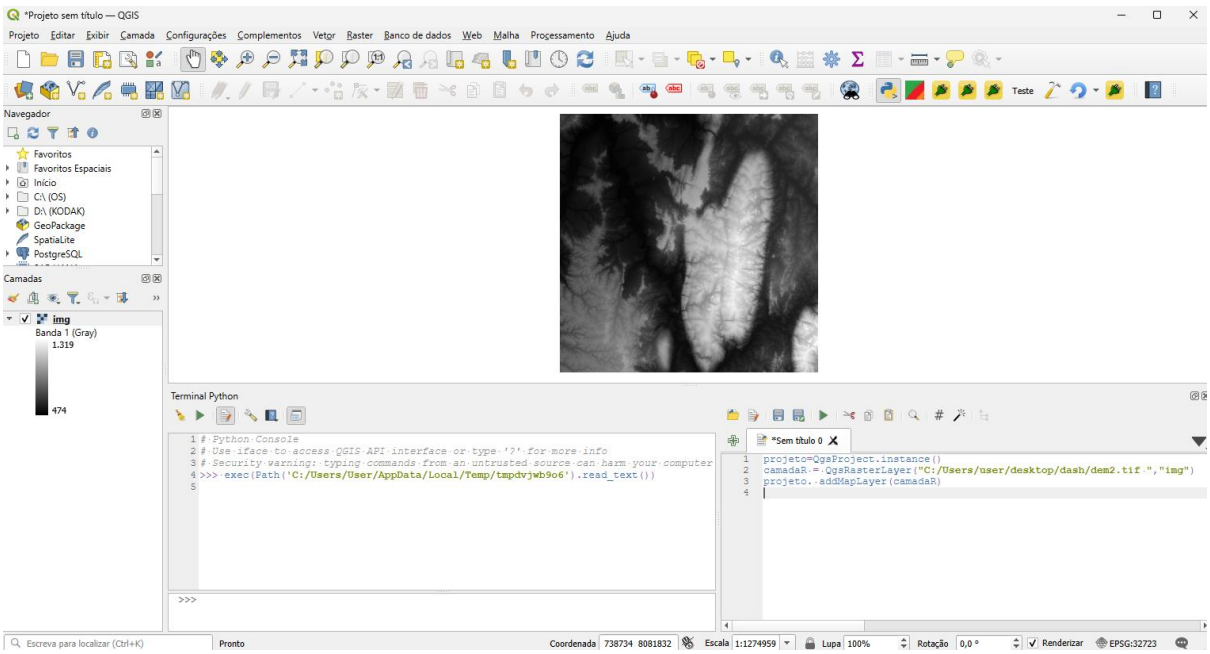


Interagindo com informações de objetos da classe Raster

Podemos extrair informações relevantes de um objeto raster também tais como dimensões, resoluções, número de bandas, valor de um pixel, etc. Vamos carregar uma imagem inicialmente.

```
projeto=QgsProject.instance()
camadaR = QgsRasterLayer("C:/Users/user/desktop/dash/dem2.tif ","img")
projeto. addMapLayer(camadaR)
```

Isso abrirá a imagem como mostrado abaixo:



Informações de dimensão do objeto Raster

Podemos acessar informações de parâmetros dimensionais de uma imagem raster usando métodos específicos para o tal.

```
camadaR.width(), camadaR.height() #largura e altura
```

(3649, 3650)

```
camadaR.extent() #extensão da imagem na unidade da coordenada
```

<QgsRectangle: 500000 7990240, 609780 8100040>

```
camadaR.crs().description() #sistema de referência
```

'WGS 84 / UTM zone 23S'

```
camadaR.rasterUnitsPerPixelX() #resolução em X
```

30.08495478213209

```
camadaR.rasterUnitsPerPixelY() #resolução em Y
```

30.08219178082192

Essas importantes informações sobre o raster poderão ser usadas para análises espacias futuras. Existem outras formas de usar os métodos para obtermos a mesma informação.

Podemos calcular a resolução em X usando o código abaixo em vez de usar o método rasterUnitsPerPixelX():

```
(camadaR.extent().xMaximum()-camadaR.extent().xMinimum())/camadaR.width()
```

30.08495478213209

Podemos também visualizar informações usando o método htmlMetadata().

```
camadaR.htmlMetadata()
```

Texto HTML gerado acimo visualizado no navegador.

Informação do provedor

Extensão	500000.0000000000000000,7990240.0000000000000000 : 609780.0000000000000000,8100040.0000000000000000
Largura	3649
Altura	3650
tipo de dado	Int16 - Inteiro de 16 bits com sinal
Descrição do driver GDAL	GTiff
Metadados do driver GDAL	GeoTIFF
Descrição do registro	C:/Users/user/desktop/dash/dem2.tif
Compressão	
Banda 1	• STATISTICS_APPROXIMATE=YES • STATISTICS_MAXIMUM=1319 • STATISTICS_MEAN=739.82844614963 • STATISTICS_MINIMUM=474 • STATISTICS_STDDEV=201.15310812442 • STATISTICS_VALID_PERCENT=100 • Escala: 1 • Deslocamento: 0
Mais informação	• AREA_OR_POINT=Area • TIFFTAG_XRESOLUTION=1 • TIFFTAG_YRESOLUTION=1
Dimensões	X: 3649 Y: 3650 Bandas: 1
Origem	500000.0000000000000000,8100040.0000000000000000
Tamanho do Pixel	30.0849547821320904,-30.0821917808219102

Sistema de referência de coordenadas (SRC)

Nome	EPSG:32723 - WGS 84 / UTM zone 23S
Unidades	metros
Type	Projetado
Método	Universal Transverse Mercator (UTM)
Celestial Body	Earth
Precisão	Com base na *World Geodetic System 1984 ensemble* (EPSG:6326), que tem uma precisão limitada de **no máximo 2 metros**.
Referência	Dinâmico (depende de um dado que não está fixado na placa)

Identificação

Identifier	
Parent Identifier	
Title	
Type	dataset
Language	
Abstract	
Categories	
Keywords	

Extensão

CRS	EPSG:32723 - WGS 84 / UTM zone 23S - Projected
Spatial Extent	
Temporal Extent	

Acesso

Fees	
Licenses	
Rights	
Constraints	

Bandas

Contagem de bandas 1

Número	Banda	Sem Dados	Mín	Máx
1	Banda 1	n/a	474.0000000000	1319.0000000000

Contatos

No contact yet.

Referências

No links yet.

Histórico

No history yet.

Raster com uma banda de valores

Vamos ver agora métodos para raster de uma banda. Os métodos abaixo informam o número de bandas e o tipo da imagem raster. 0 para cinza ou não definido de banda única, 1 para paletado de banda única e 2 para multibanda.

```
camadaR.bandCount()
```

1

```
camadaR.rasterType()
```

<RasterLayerType.GrayOrUndefined: 0>

A função dataProvider() funciona como uma interface entre o objeto raster os seus dados individuais, seu método sample() toma dois valores, um objeto ponto (coordenadas XZ) e o número da banda. Se a coordenada for dentro da imagem e a banda existir o resultado será um tuple com o valor do pixel e se o dado é verdadeiro ou não.

```
valor=camadaR.dataProvider().sample(QgsPointXY(687567, 7460876),1)
valor
(nan, False)

valor2=camadaR.dataProvider().sample(QgsPointXY(547802,8043049), 1)
valor2
(980.0, True)
```

A rampa de cor assinalada ao objeto raster pode ser checada usando o método type() do método renderer(). O tipo singlebandgray é o padrão inicial.

```
>>> camadaR.renderer().type()

'singlebandgray'
```

Podemos alterar via python a rampa de cores, o processo é mostrado abaixo. O processo envolve na criação de um objeto do tipo ColorRampShader e definimos a rampa de cor de preenchimento como sendo do tipo interpolado.

```
fcn = QgsColorRampShader()
fcn.setColorRampType(QgsColorRampShader.Interpolated)
```

Criamos agora uma lista com as cores representando os dois valores extremos do raster (0 e 2046 que serão interpolados entre azul e amarelo. Em seguida adicionamos esta lista como item do ColorRampShader criado acima.

```
lista = [ QgsColorRampShader.ColorRampItem(0, QColor(0,0,255)),
QgsColorRampShader.ColorRampItem(2046, QColor(255,255,0))]
fcn.setColorRampItemList(lista)
```

O próximo passo é criarmos o RasterShader (preenchedor de cor) e associarmos o RampShader a ele.

```
shader = QgsRasterShader()
shader.setRasterShaderFunction(fcn)
```

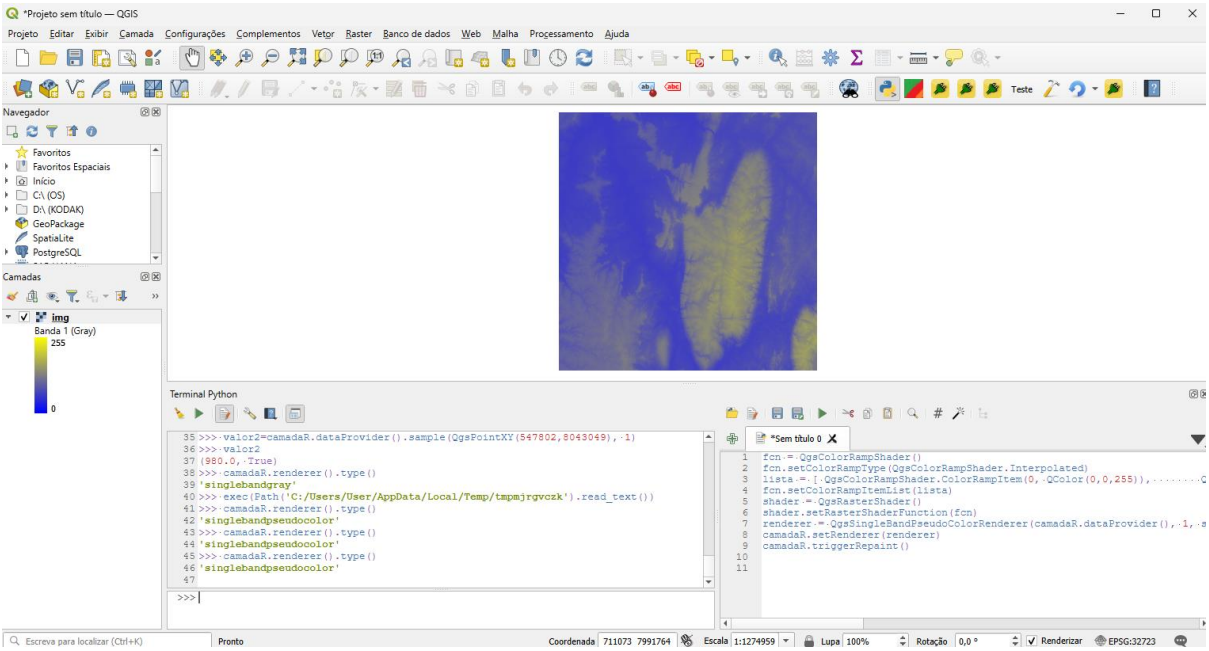
Finalmente criamos o objeto renderizador de cor com: dados do objeto raster, banda 1 e shader acima. Em seguida aplicamos este ao objeto raster e chamamos a repintura do objeto.

```
renderer = QgsSingleBandPseudoColorRenderer(camadaR.dataProvider(), 1, shader)
camadaR.setRenderer(renderer)
camadaR.triggerRepaint()
```

Se chamarmos o tipo novamente podemos ver a mudança.

```
camadaR.renderer().type()

'singlebandpseudocolor'
```

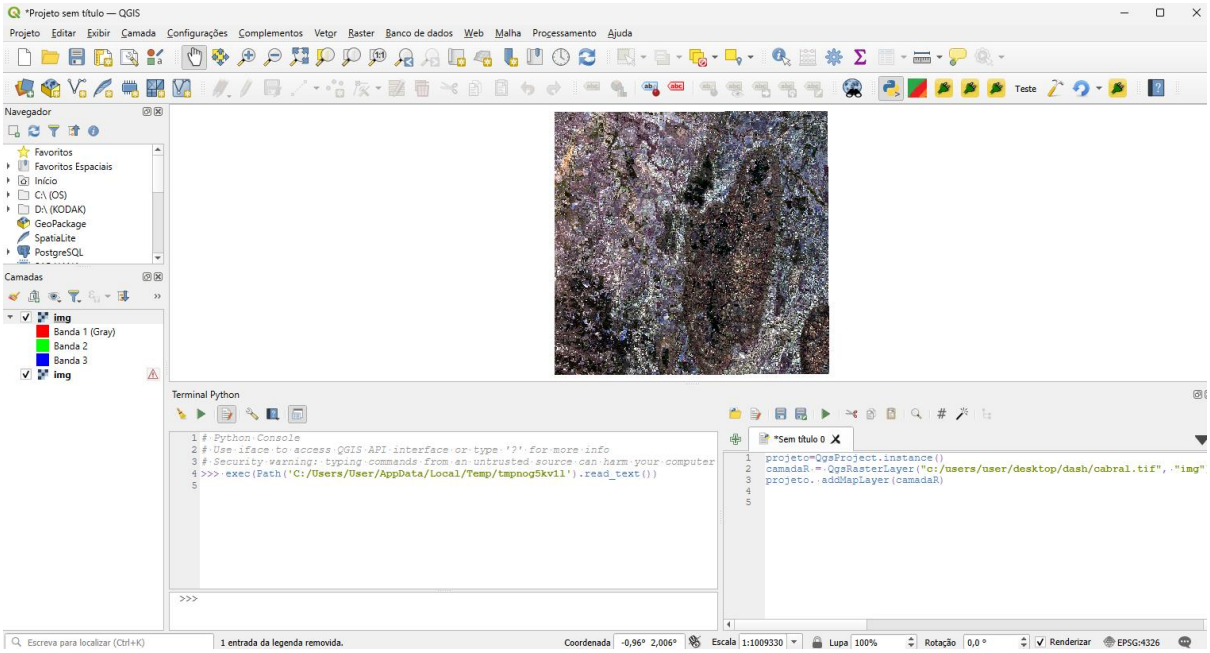
O resultado é mostrado na imagem abaixo.



Raster com mais de uma banda de valores

Vamos ver agora métodos para raster de mais de uma banda. Vamos trabalhar um pouco agora com imagem raster de 3 bandas. Carregamos o raster de forma similar e vamos extrair algumas de suas informações.

```
projeto=QgsProject.instance()
camadaR = QgsRasterLayer("c:/users/user/desktop/dash/cabral.tif",
"img")
projeto. addMapLayer(camadaR)
```



Podemos ver algumas das informações usando:

```
camadaR.bandCount()# número de bandas
```

3

```
camadaR.rasterType() # 2 para multi banda
```

<RasterLayerType.MultiBand: 2>

```
# valor do pixel na banda 1
valor=camadaR.dataProvider().sample(QgsPointXY(547802,8043049),1)
>>> valor
```

(893.0, True)

```
# valor do pixel na banda 2
valor=camadaR.dataProvider().sample(QgsPointXY(547802,8043049),2)
valor
```

(781.0, True)

```
# valor do pixel na banda 3
valor=camadaR.dataProvider().sample(QgsPointXY(547802,8043049),3)
valor
```

(719.0, True)

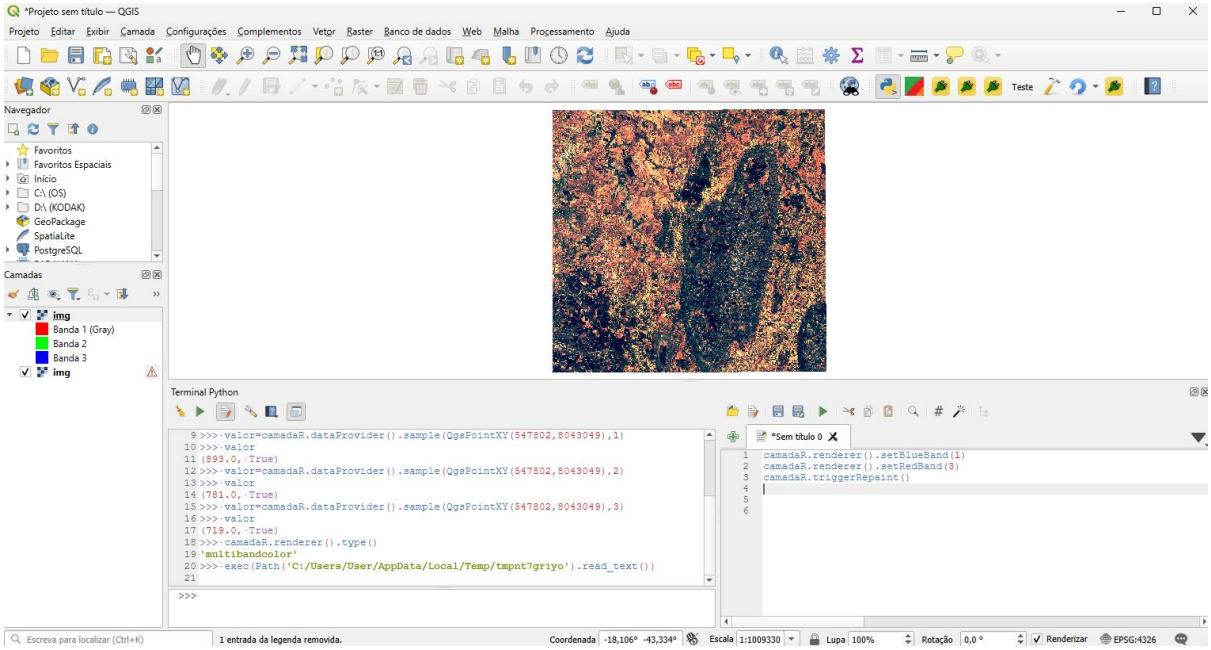
```
camadaR.renderer().type()
```

'multibandcolor'

Vamos ver abaixo como modificar a imagem para que a banda 1 fique no canal azul (B) e a banda 3 fique no canal Vermelho (R).

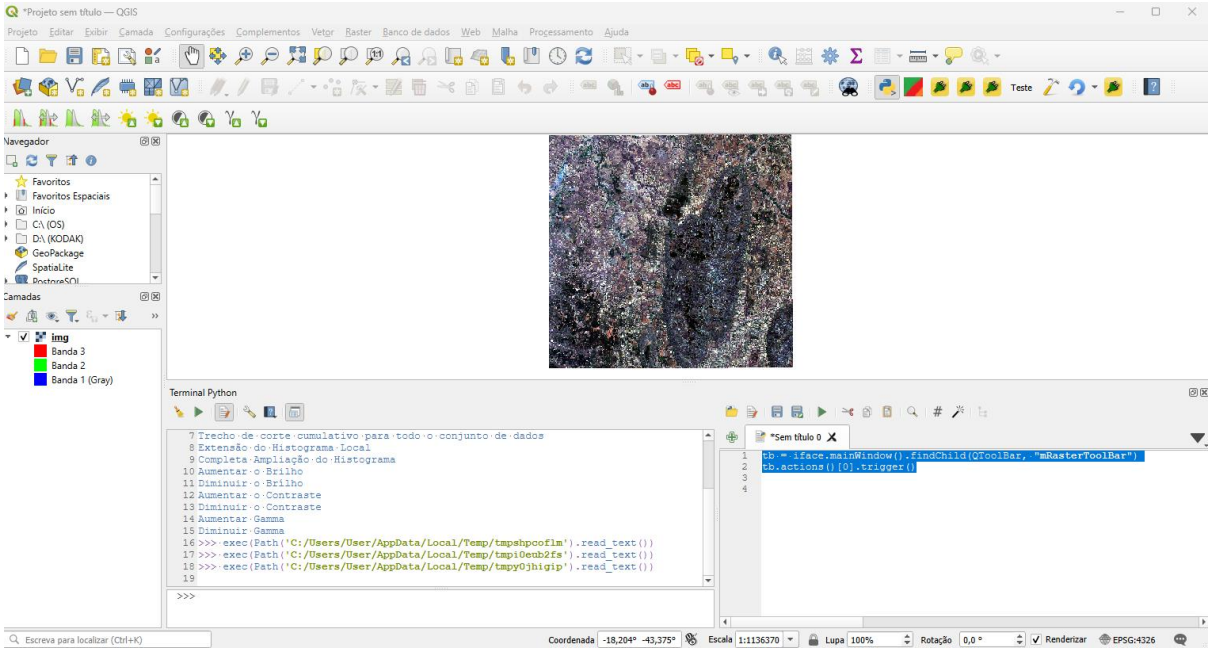
```
camadaR.renderer().setBlueBand(1)
camadaR.renderer().setRedBand(3)
camadaR.triggerRepaint()
```

Note que o histograma da imagem não foi apropriadamente ajustado porque ainda usa os valores de máximo e mínimo das bandas anteriores.



Para ajustar execute

```
tb = iface.mainWindow().findChild(QToolBar, "mRasterToolBar")
tb.actions()[0].trigger()
```



Criando objeto raster

Imagens raster também podem ser criadas via script de forma bem eficiente usando uma lista de dados pontuais com um determinado valor. Vamos aqui criar um raster mostrando a temperatura média de uma área com base em informações pontuais de vários locais. O arquivo CSV tfinal.csv tem os dados com coordenadas, e respectivos valores. Vamos carregar a informação em um objeto do tipo QsgInterpolator camada de dados (layerData).

```
uri="file:///c:/users/user/desktop/dash/tfinal.csv?
type=csv&xField=LONGITUDE&yField=LATITUDE&crs=epsg:4326"
camada = QgsVectorLayer(uri, 'Converte', "delimitedtext")
c_data = QgsInterpolator.LayerData()
c_data.source = camada
c_data.zCoordInterpolation = False
c_data.interpolationAttribute = 6
c_data.sourceType = QgsInterpolator.SourcePoints
```

Executaremos a interpolação usando o inverso da distância ponderada (IDW) ao quadrado (coeficiente 2).

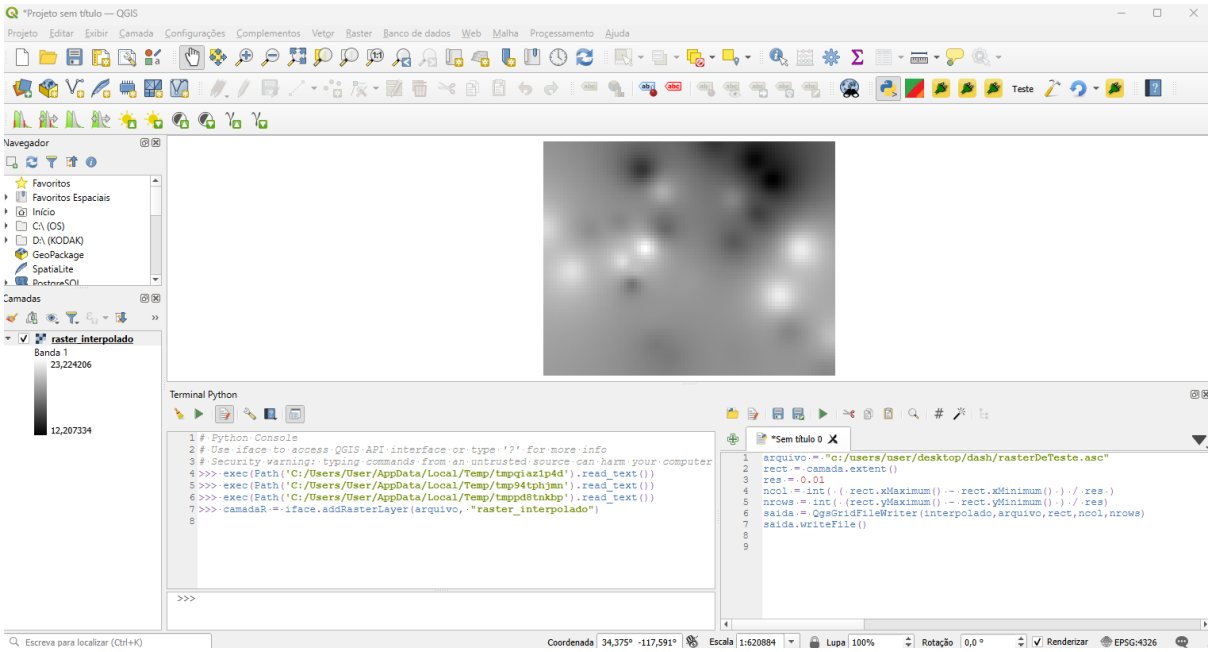
```
interpolado = QgsIDWInterpolator([c_data])
interpolado.setDistanceCoefficient(2)
```

Agora definimos qual arquivo será criado e os parâmetros do grid a ser usado.

```
arquivo = "c:/users/user/desktop/dash/rasterDeTeste.asc"
rect = camada.extent()
res = 0.01
ncol = int( ( rect.xMaximum() - rect.xMinimum() ) / res )
nrows = int( (rect.yMaximum() - rect.yMinimum() ) / res)
saida = QgsGridFileWriter(interpolado,arquivo,rect,ncol,nrows)
saida.writeFile()
```

Carregamos o arquivo do grid usando.

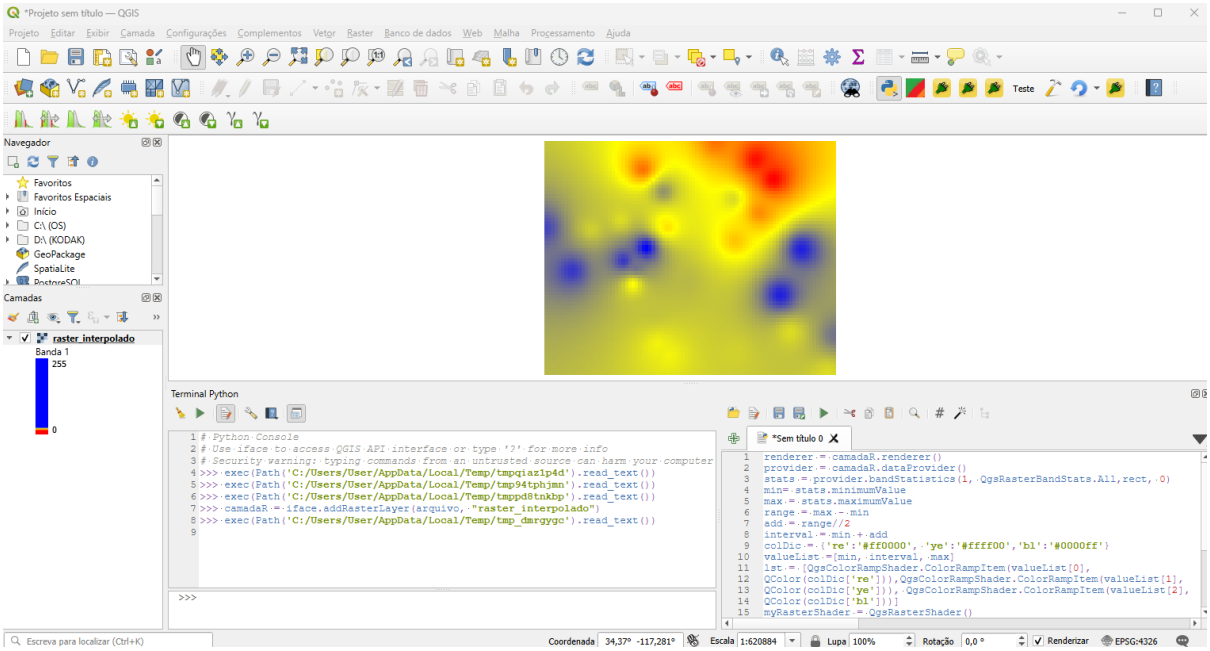
```
camadaR = iface.addRasterLayer(arquivo, "raster_interpolado")
```



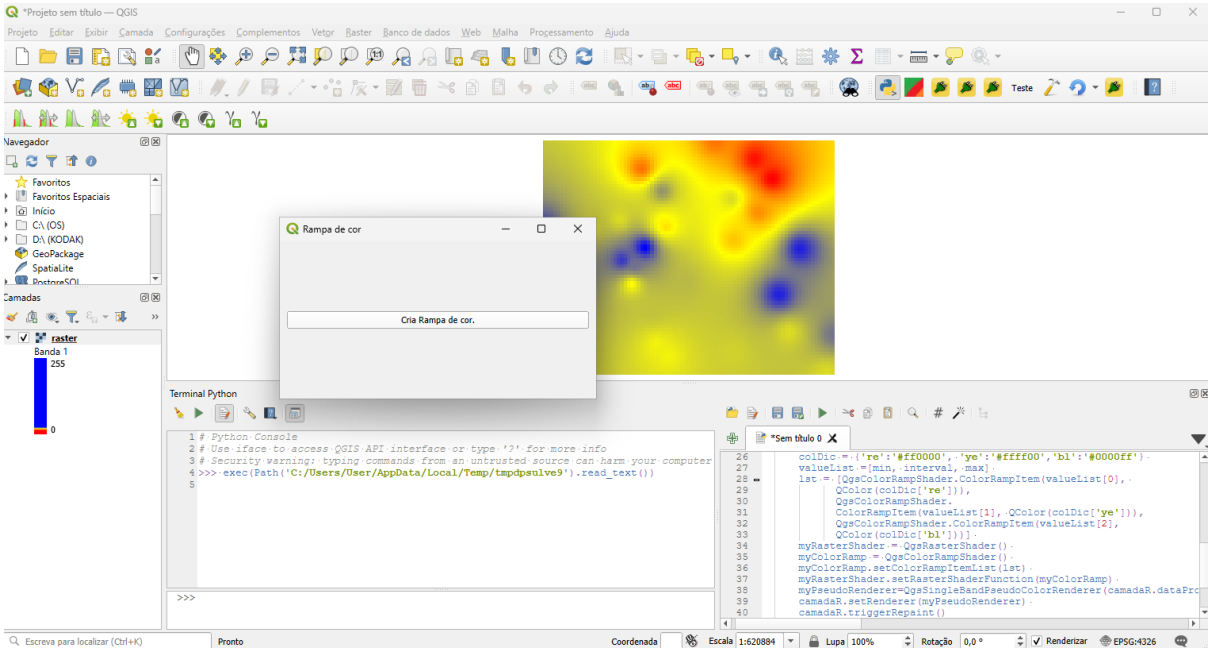
Podemos alterar a aparência do raster que criamos usando o código seguinte.

```
renderer = camadaR.renderer()
provider = camadaR.dataProvider()
stats = provider.bandStatistics(1, QgsRasterBandStats.All,rect, 0)
min= stats.minimumValue
max = stats.maximumValue
range = max - min
add = range//2
interval = min + add
colDic = {'re':'#ff0000', 'ye':'#ffff00','bl':'#0000ff'}
valueList =[min, interval, max]
lst = [QgsColorRampShader.ColorRampItem(valueList[0],
QColor(colDic['re'])),QgsColorRampShader.ColorRampItem(valueList[1],
QColor(colDic['ye'])),
QgsColorRampShader.ColorRampItem(valueList[2],
QColor(colDic['bl']))]
myRasterShader = QgsRasterShader()
myColorRamp = QgsColorRampShader()
myColorRamp.setColorRampItemList(lst)
myRasterShader.setRasterShaderFunction(myColorRamp)
myPseudoRenderer
=QgsSingleBandPseudoColorRenderer(camadaR.dataProvider(),
camadaR.type(), myRasterShader)
camadaR.setRenderer(myPseudoRenderer)
camadaR.triggerRepaint()
```

O resultado da rampa de cores criada aplicado no raster será.



No próximo módulo veremos outras classes do pyQGIS e suas utilidades.



Usando pyQGIS

Classes, Informações Adicionais e Cheat Sheet

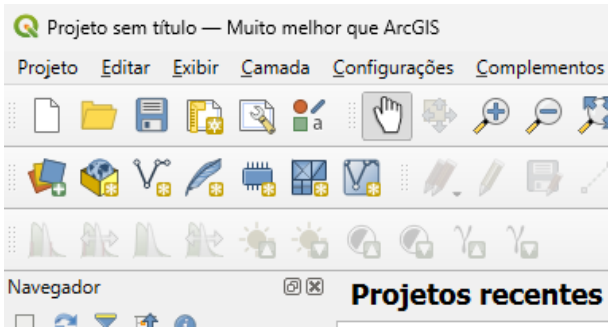
iface - QgisInterface

A classe QgisInterface e iniciada automaticamente quando o QGIS é iniciado e ela é representada pela variável iface (de interface). O iface é o seu programa QGIS e representa a tela principal, os menus, a barra de ferramentas e as demais partes do programa.

No módulo anterior usamos o iface em duas ocasiões, uma para acessar uma ferramenta da barra de ferramenta de raster, e outra para carregar um raster gerado por interpolação. Vamos agora ver outras funções da classe iface.

Mudando o título do nome do programa

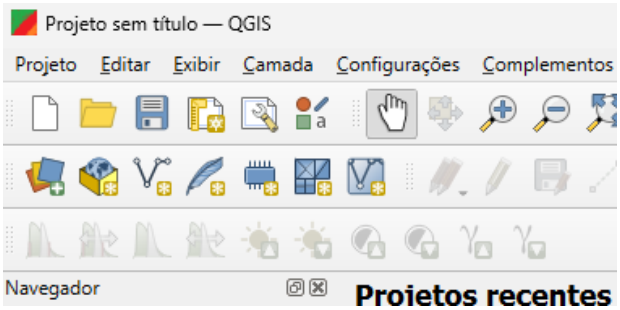
```
titulo = iface.mainWindow().windowTitle()
novo_titulo = titulo.replace('QGIS', 'Muito melhor que ArcGIS')
iface.mainWindow().setWindowTitle(novo_titulo)
```



Mudando o ícone do programa

Mudamos o ícone usando o código abaixo. O **os.path.expanduser('~')** te leva para o pasta raiz de usuário no seu sistema.

```
import os
icone = 'icon.png'# use um arquivo png 64x64 pixels
dire = os.path.join(os.path.expanduser('~'), 'desktop/dash/')
path = os.path.join(dire, icone)
icone2 = QIcon(path)
iface.mainWindow().setWindowIcon(icone2)
```

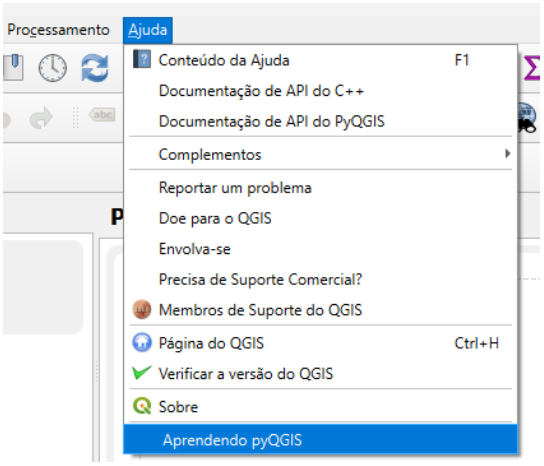


Adicionando um novo item de menu ajuda

Com o seguinte código adicionamos um nove item no menu Ajuda que direciona para a nossa página do Aprendendo pyQGIS.

```
import webbrowser
def abreSite():
```

```
    webbrowser.open('https://gdatasystems.com/pyqgis')

acao = QAction('Aprendendo pyQGIS ')
acao.triggered.connect(abreSite)
iface.helpMenu().addSeparator()
iface.helpMenu().addAction(acao)
```

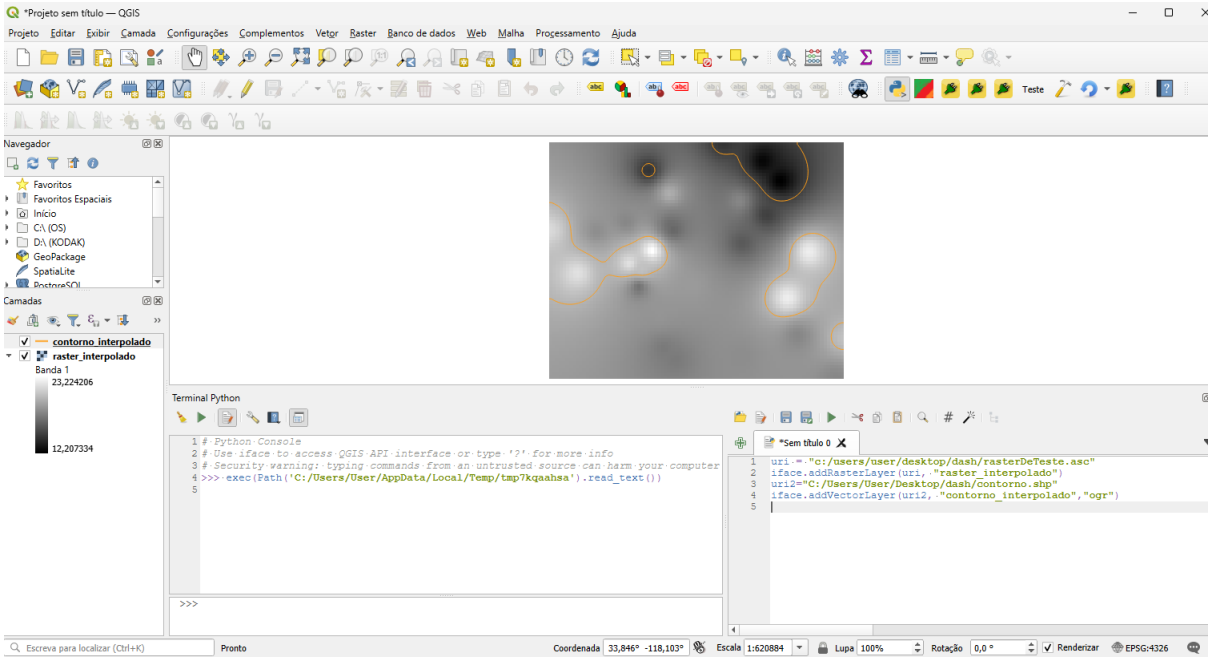


Clicando no item a página vai abrir no navegador.

Adicionando uma nova camada

Como já fizemos no módulo anterior podemos adicionar novas camadas usando o iface.

```
uri = "c:/users/user/desktop/dash/rasterDeTeste.asc"
iface.addRasterLayer(uri, "raster_interpolado")
uri2="C:/Users/User/Desktop/dash/contorno.shp"
iface.addVectorLayer(uri2, "contorno_interpolado","ogr")
```



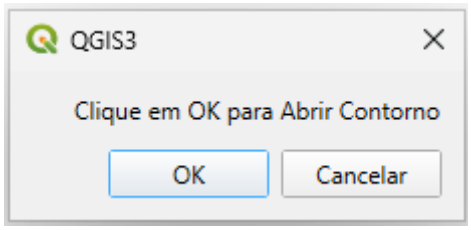
PyQt

Qt é um conjunto de ferramentas ou plataforma open-source de widget (controles do tipo botões, entrada de texto, rótulos, etc) para criar interfaces gráficas de usuário (GUI) e aplicativos multiplataforma. O QGIS é construído sobre esta plataforma. O PyQt é a interface Python do Qt. O PyQt fornece classes e funções de interação com os widgets QT.

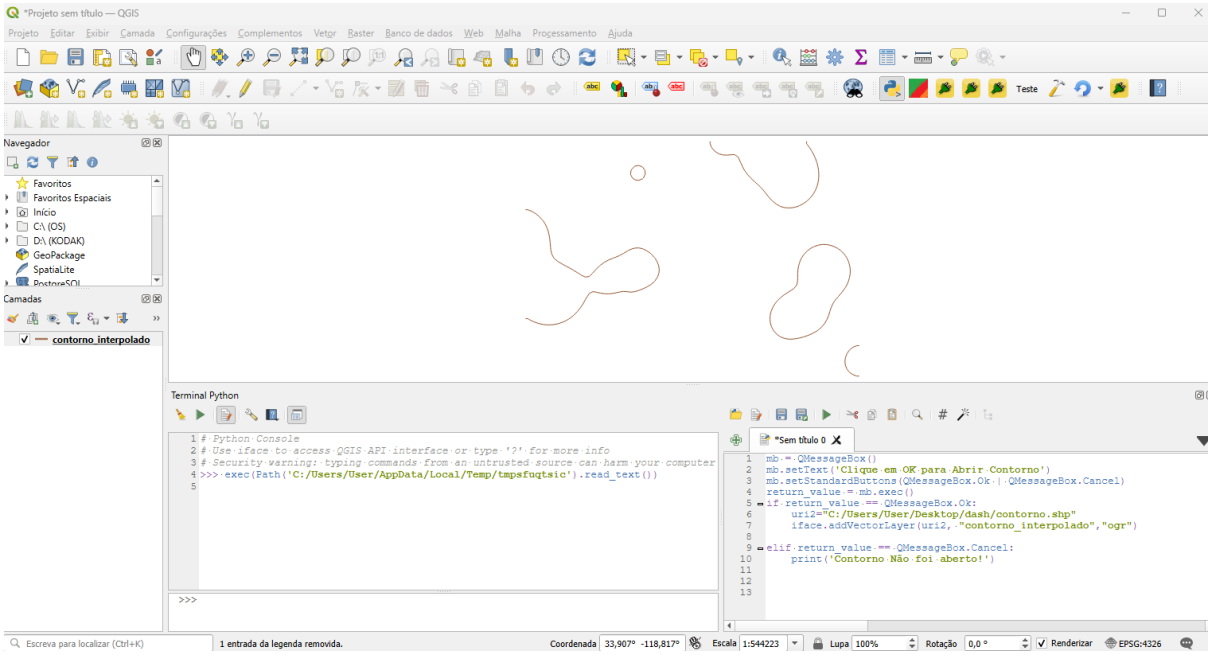
QMessageBox

Um exemplo de como usar a classe Qt diretamente no QGIS.

```
mb = QMessageBox()
mb.setText('Clique em OK para Abrir Contorno')
mb.setStandardButtons(QMessageBox.Ok | QMessageBox.Cancel)
return_value = mb.exec()
if return_value == QMessageBox.Ok:
    uri2="C:/Users/User/Desktop/dash/contorno.shp"
    iface.addVectorLayer(uri2, "contorno_interpolado","ogr")

elif return_value == QMessageBox.Cancel:
    print('Contorno Não foi aberto!')
```



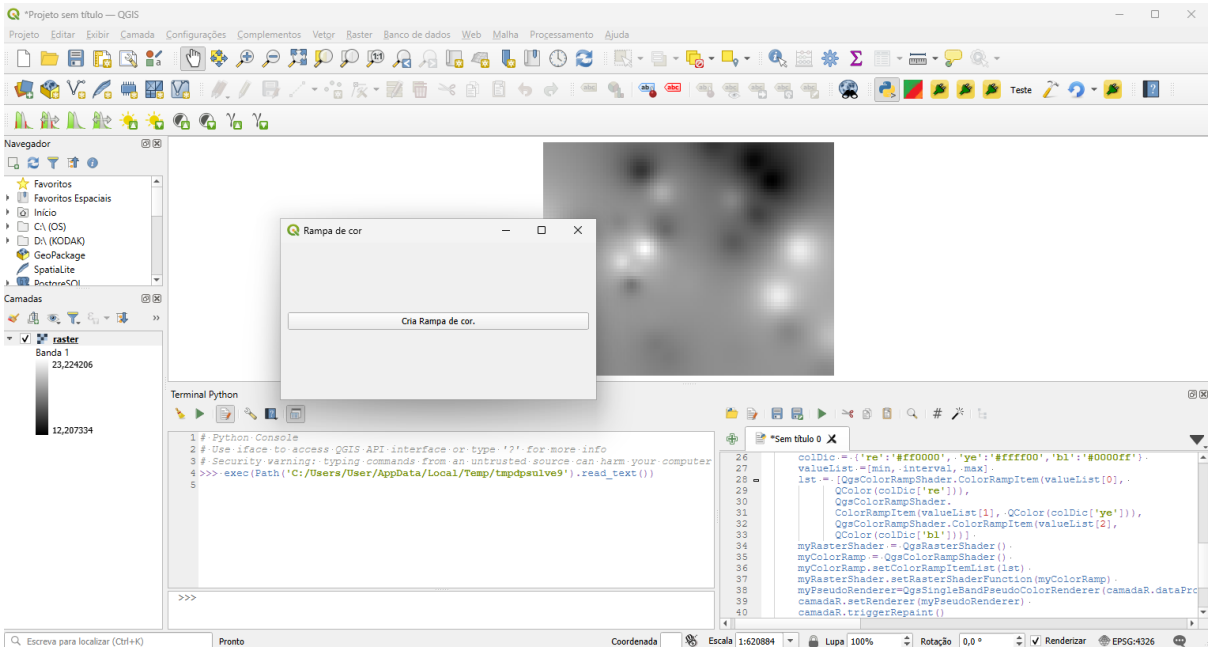
Ao clicar OK teremos:



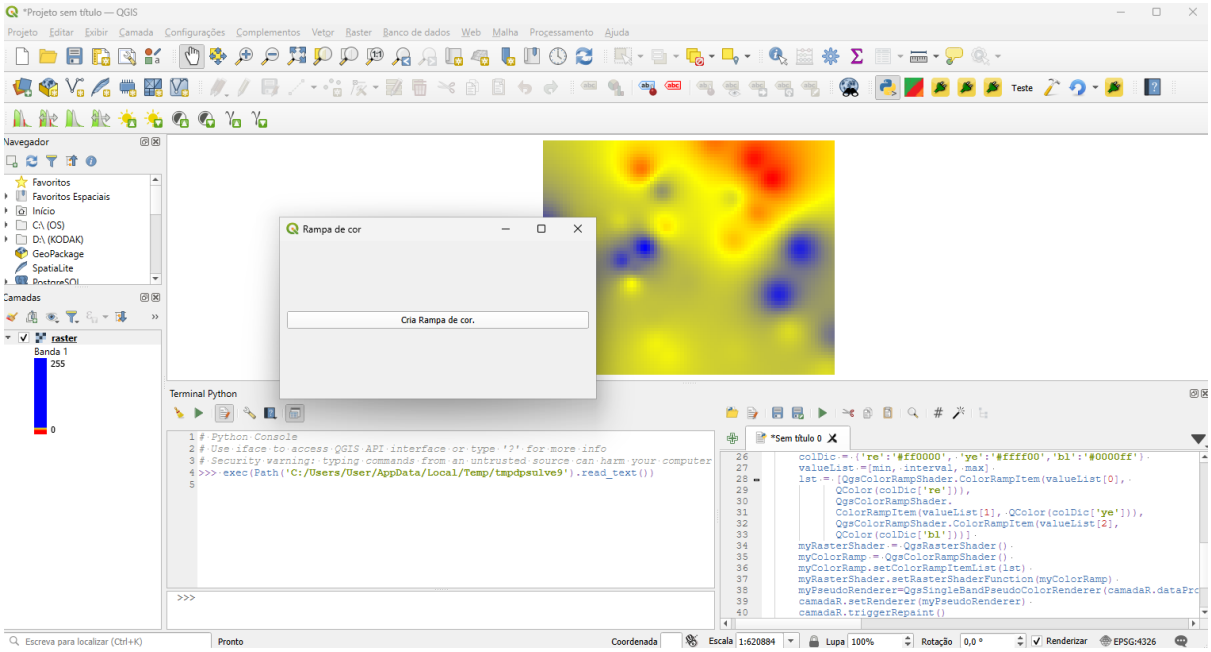
Agora somente um exemplo mais elaborado para ilustrar como a interação entre QGIS e QT pode ser feita, mas não se preocupe, temos os Plugins para facilitar tudo. Isso é somente uma demonstração. Para os pligins usaremos o QtDesigner para criar as Interfaces Gráficas visualmente.

```
class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("Rampa de cor")
        self.setGeometry(100, 100, 400, 200)
        central_widget = QWidget()
        self.setCentralWidget(central_widget)
        dialog_button = QPushButton("Cria Rampa de cor.")
        dialog_button.clicked.connect(self.rampa)
        layout = QVBoxLayout()
        layout.addWidget(dialog_button)
        central_widget.setLayout(layout)

    def rampa(self):
        camadaR = iface.activeLayer()
        renderer = camadaR.renderer()
        provider = camadaR.dataProvider()
        rect = camadaR.extent()
        stats = provider.bandStatistics(1,
                QgsRasterBandStats.All,rect, 0)
        min= stats.minimumValue
        max = stats.maximumValue
        range = max - min
        add = range//2
        interval = min + add
        colDic = {'re':'#ff0000', 'ye':'#ffff00','bl':'#0000ff'}
        valueList =[min, interval, max]
        lst = [QgsColorRampShader.ColorRampItem(valueList[0],
              QColor(colDic['re'])),
              QgsColorRampShader.
              ColorRampItem(valueList[1], QColor(colDic['ye'])),
              QgsColorRampShader.ColorRampItem(valueList[2],
              QColor(colDic['bl']))]
        myRasterShader = QgsRasterShader()
        myColorRamp = QgsColorRampShader()
        myColorRamp.setColorRampItemList(lst)
        myRasterShader.setRasterShaderFunction(myColorRamp)
        myPseudoRenderer=
            QgsSingleBandPseudoColorRenderer(camadaR.dataProvider(),
            camadaR.type(), myRasterShader)
        camadaR.setRenderer(myPseudoRenderer)
        camadaR.triggerRepaint()
arquivo = "c:/users/user/desktop/dash/rasterDeTeste.asc"
iface.addRasterLayer(arquivo, "raster")
window = MyWindow()
window.show()
```

Ao executar teremos:



E clicando no botão teremos:



Executando um script pyQGS fora do Qgis

Podemos executar processamentos do QGIS sem iniciar a interface gráfica, usando somente scripts. Para fazer isso temos que criar um script com a seguinte estrutura mínima.

```
from qgis.core import *
# indique onde o programa qgis está localizado no seu computador
QgsApplication.setPrefixPath("/usr", True)
# Crie uma referência à QgsApplication. Usando False como segundo
argumento
# para não ativar a interface gráfica.
qgs = QgsApplication([], False)
# inicie o qgis
qgs.initQgis()
############################################
# Escreva o código de processamento aqui###
############################################
# finalize o script usando:
qgs.exitQgis()
```

Vamos mostrar como isso funciona criando um script que criará um grid raster a partir de um arquivo texto CSV com alguns pontos. O mesmo procedimento do último exemplo do mõdulo anterior. Nomeie o arquivo de **criaRaster.py**. Substitua users/user/desktop/dash apropriadamente para o seu sistema.

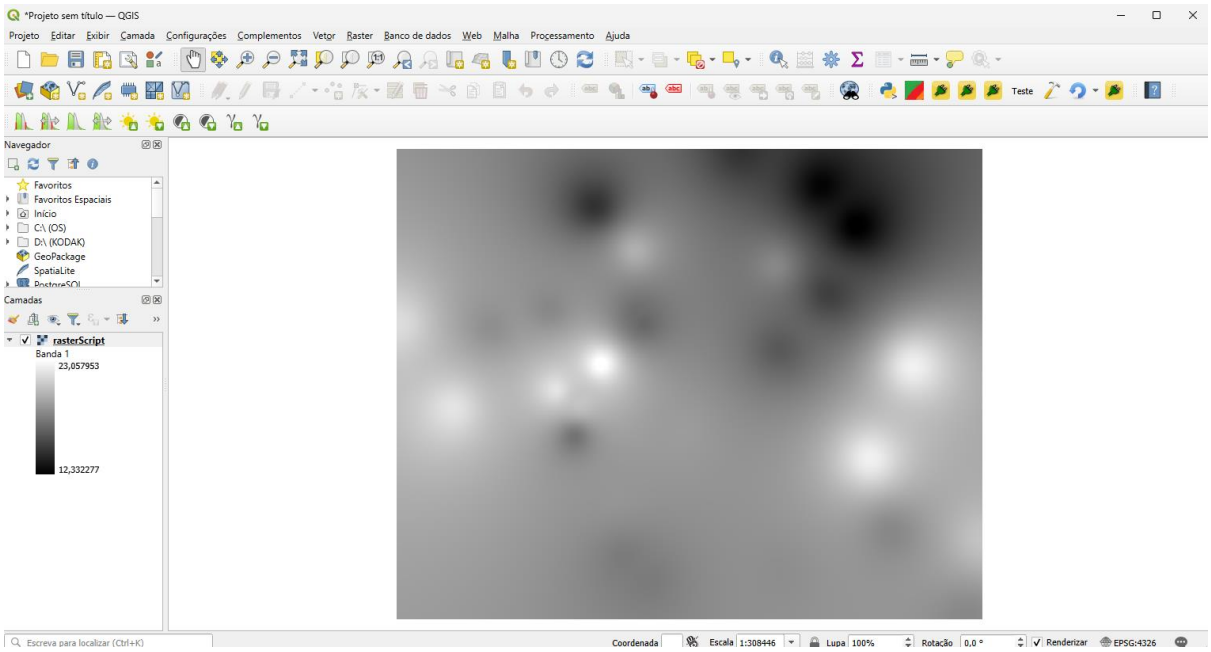
```
from qgis.core import *
from qgis.analysis import *
import os
QgsApplication.setPrefixPath("C:/ProgramFiles/QGIS3.34.1/bin/",
True)
qgs = QgsApplication([], False)
qgs.initQgis()
uri="file:///users/user/desktop/dash/tfinal.csv?type=csv&xField=LONG
ITUDE&yField=LATITUDE&crs=epsg:4326"
camada = QgsVectorLayer(uri, 'Converte', "delimitedtext")
if not camada.isValid():
    print("A camada não carregou apropriadamente!")
else:
    c_data = QgsInterpolator.LayerData()
    c_data.source = camada
    c_data.zCoordInterpolation = False
    c_data.interpolationAttribute = 6
    c_data.sourceType = QgsInterpolator.SourcePoints
    interpolado = QgsIDWInterpolator([c_data])
    interpolado.setDistanceCoefficient(2)
    arquivo = os.path.expanduser('~')+"/desktop/dash/rasterScript.asc"
    rect = camada.extent()
    res = 0.001
    ncol = int( ( rect.xMaximum() - rect.xMinimum() ) / res )
    nrows = int( (rect.yMaximum() - rect.yMinimum() ) / res)
    saida = QgsGridFileWriter(interpolado,arquivo,rect,ncol,nrows)
    saida.writeFile()
```

```
qgs.exitQgis()
```

Abrir o shell OSGeo4W, navegar até a pasta do script **criaRaster.py** e executar usando:

```
python-qgis criaRaster.py
```

Ao abrirmos o raster rasterScript.asc criado pelo script acima no QGIS veremos o seguinte.



Colinha do pyQGS

A seguir temos uma cola (cheat sheet) das classes principais para uma rápida referência. Em azul temos as bibliotecas a serem carregadas no caso de usar em Plugins ou em scripts fora do QGIS.

QgsInterface() iface

Aparência

```
from qgis.PyQt.QtWidgets import QApplication

app = QApplication.instance()
app.setStyleSheet(".QWidget {color:blue;background-color:yellow;}")
```

Você pode carregar o stylesheet de um arquivo com

```
from qgis.PyQt.QtWidgets import QApplication
app = QApplication.instance()
with  open("testdata/file.qss") as qss_file_content:
    app.setStyleSheet(qss_file_content.read()
```

Mudando o ícone e título do programa

```
from qgis.PyQt.QtGui import QIcon
icon = QIcon("/caminho/para/ele/icon.png")
iface.mainWindow().setWindowIcon(icon)
iface.mainWindow().setWindowTitle("ArcGIS só que não....")
```

Barra de Ferramentas

Remove e adiciona barra de ferramenta

```
toolbar = iface.helpToolBar()
parent = toolbar.parentWidget()
parent.removeToolBar(toolbar)
parent.addToolBar(toolbar)
```

Remove ações da barra de ferramenta

```
actions = iface.attributesToolBar().actions()
iface.attributesToolBar().clear()
iface.attributesToolBar().addAction(actions[4])
iface.attributesToolBar().addAction(actions[3])
```

Menus

Remove e adiciona Menu

```
menu = iface.helpMenu()
menubar = menu.parentWidget()
menubar.removeAction(menu.menuAction())
menubar.addAction(menu.menuAction())
```

Canvas (tela do mapa)

Acessando o canvas

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

Mudando a cor do canvas

```
from qgis.PyQt.QtCore import Qt
iface.mapCanvas().setCanvasColor(Qt.black)
iface.mapCanvas().refresh()
```

Mudando intervalo de atualização do mapa

```
from qgis.core import QgsSettings
# para 150 milisegundos
QgsSettings().setValue("/qgis/map_update_interval", 150)
```

Camadas

Adicionando uma camada vetor do tipo shapefile

```
camada = iface.addVectorLayer("caminho/aoi.shp", "Camada", "ogr")
if not camada or not camada.isValid():
    print("Falha ao carregar camada!")
```

Adicionando uma camada raster do tipo tif

```
from qgis.core import QgsRasterLayer
arquivoR = "caminho/srtm.tif"
camada = QgsRasterLayer(arquivo, "Raster")
if not camada.isValid():
    print("Falha ao carregar camada!")
```

Obtendo a camada ativa

```
camada = iface.activeLayer()
```

Listando todas as camadas

```
from qgis.core import QgsProject
QgsProject.instance().mapLayers().values()
```

Obtendo o nome das camadas

```
from qgis.core import QgsProject
nomes = [layer.name() for layer in QgsProject.instance().mapLayers().values()]
print("Camadas = {}".format(nomes))
```

Obtendo camada pelo seu nome e fazendo ela ativa

```
from qgis.core import QgsProject
camada = QgsProject.instance().mapLayersByName("NomeCamada")[0]
iface.setActiveLayer(layer)
```

Adicionando um novo elemento

```
from qgis.core import QgsGeometry, QgsPointXY, QgsFeature
pr = camada.dataProvider()
feat = QgsFeature()
feat.setGeometry(QgsGeometry.fromPointXY(QgsPointXY(10,10)))
pr.addFeatures([feat])
```

Obtendo os elementos

```
for f in camada.getFeatures():
    print (f)
```

Obtendo elementos selecionados

```
for f in camada.selectedFeatures():
    print (f)
```

Obtendo o ID dos elementos selecionados

```
selected_ids = camada.selectedFeatureIds()
print(selected_ids)
```

Criando uma camada na memória com os elementos selecionados

```
from qgis.core import QgsFeatureRequest
camadam= layer.materialize(QgsFeatureRequest().setFilterFids(layer.selectedFeatureIds()))
QgsProject.instance().addMapLayer(camadam)
```

Obtendo a geometria da camada

```
# Camada tipo Point
for f in camada.getFeatures():
    geom = f.geometry()
    print ('%f, %f' % (geom.asPoint().y(), geom.asPoint().x()))
```

Transladando uma geometria

```
from qgis.core import QgsFeature, QgsGeometry
f = QgsFeature()
geom = QgsGeometry.fromWkt("POINT(7 45)")
geom.translate(1, 1)
f.setGeometry(geom)
print(f.geometry())
```

Assinalando o Sistema de referência de coordenadas (CRS) de uma camada

```
from qgis.core import QgsProject, QgsCoordinateReferenceSystem
for camada in QgsProject.instance().mapLayers().values():
    camada.setCrs(QgsCoordinateReferenceSystem('EPSG:4326'))
```

Checando o CRS de uma camada

```
from qgis.core import QgsProject
for camada in QgsProject.instance().mapLayers().values():
    crs = camada.crs().authid()
    camada.setName('{} ({})'.format(camada.name(), crs))
```

Uma função para ocultar uma coluna de atributo

```
from qgis.core import QgsEditorWidgetSetup
def fieldVisibility (camada,fname):
    setup = QgsEditorWidgetSetup('Hidden', {})
    for i, column in enumerate(layer.fields()):
        if column.name()==fname:
            camada.setEditorWidgetSetup(idx, setup)
            break
        else:
            continue
```

Criando camada na memória a partir de WKT

```
from qgis.core import QgsVectorLayer,QgsFeature,QgsGeometry,QgsProject

camada = QgsVectorLayer('Polygon?crs=epsg:4326', 'MS', 'memory')
pr = camada.dataProvider()
```

```
f = QgsFeature()
geom = QgsGeometry.fromWkt("POLYGON ((-88.82 34.99,-88.09 34.89,-
88.39 30.34,-89.57 30.18,-89.73 31,-91.63 30.99,-90.87 32.37,-91.23
33.44,-90.93 34.23,-90.30 34.99,-88.82 34.99))")
poly.setGeometry(geom)
pr.addFeatures([f])
camada.updateExtents()
QgsProject.instance().addMapLayers([camada])
```

Lendo as camadas de um GeoPackage de vetores

```
from qgis.core import QgsDataProvider
arquivo = "caminho/seupacote.gpkg"
camada = QgsVectorLayer(arquivo, "teste", "ogr")
subCamadass = camada.dataProvider().subLayers()

for subCamada in subCamadass:
    nome = subCamada.split(QgsDataProvider.SUBLAYER_SEPARATOR)[1]
    uri = "%s|layername=%s" % (arquivo, nome,)
    # Cria a Camada
    subCamadav = QgsVectorLayer(uri, nome, 'ogr')
    # Adiciona a camada ao mapa
    QgsProject.instance().addMapLayer(subVamadav)
```

Adicionando uma camada Tile (Camada XYZ)

```
from qgis.core import QgsRasterLayer, QgsProject
def loadXYZ(url, name):
    rasterLyr = QgsRasterLayer("type=xyz&url=" + url, name, "wms")
    QgsProject.instance().addMapLayer(rasterLyr)

url= 'https://tile.openstreetmap.org/{z}/{x}/{y}.png&zmax=19&zmin=0'
loadXYZ(url, 'OpenStreetMap')
```

Adicionando uma camada de um banco de dados Postgis

```
from qgis.core import QgsVectorLayer, QgsProject, QgsDataSourceUri
uri = QgsDataSourceUri()
uri.setConnection("servidorDNS","5432","dbase","user", "pswrd")
uri.setDataSource("schema", "tablename", "geom")
camada = QgsVectorLayer(uri.uri(False), "Remota", "postgres")
QgsProject.instance().addMapLayer(rasterLyr)
```

Removendo todas as camadas do Projeto

```
QgsProject.instance().removeAllMapLayers()
```

Removendo tudo do projeto

```
QgsProject.instance().clear()
```

Criando Plugins QGIS com pyQGIS

Arquitetura do plugin e conceitos básicos

1 - O mínimo necessário

Para iniciarmos mostraremos como criar um plugin mínimo que consiste de dois arquivos: **metadata.txt** e **__init__.py** . Estes arquivos devem ser colocados em uma nova pasta (mínimo) dentro da pasta de plugins. Normalmente, para cada tipo de sistema, a pasta fica em:

Linux
~/.local/share/QGIS/QGIS3/profiles/default/python/plugins/minimo
Windows
C:\Users\USER\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins\minimo
macOS
~/Library/Application Support/QGIS/QGIS3/profiles/default/python/plugins/minimo

Onde ~ é o diretório home do usuário no Linux ou no mac e USER é a pasta do usuário no windows.

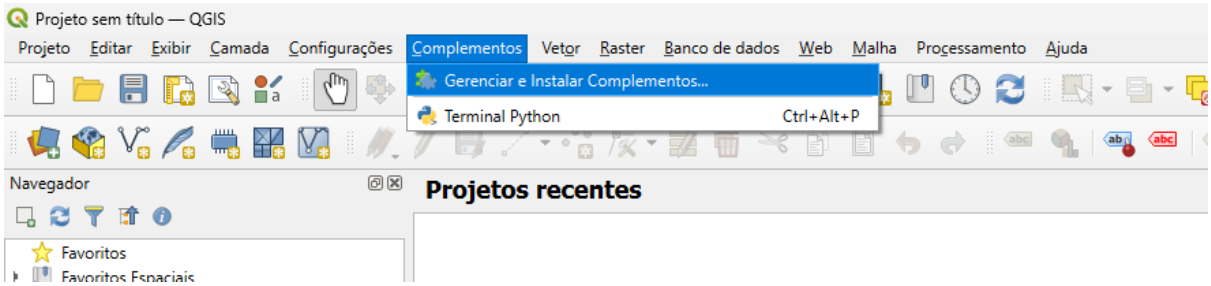
Arquivo **metadata.txt** com os oito campos obrigatórios

```
[general]
name=Minimo
description=Plugin Mínimo
about=Sobre este plugin
version=1.0
qgisMinimumVersion=3.0
author=Você
email=seu@email.com
repository=URL para o repositório do código na web
```

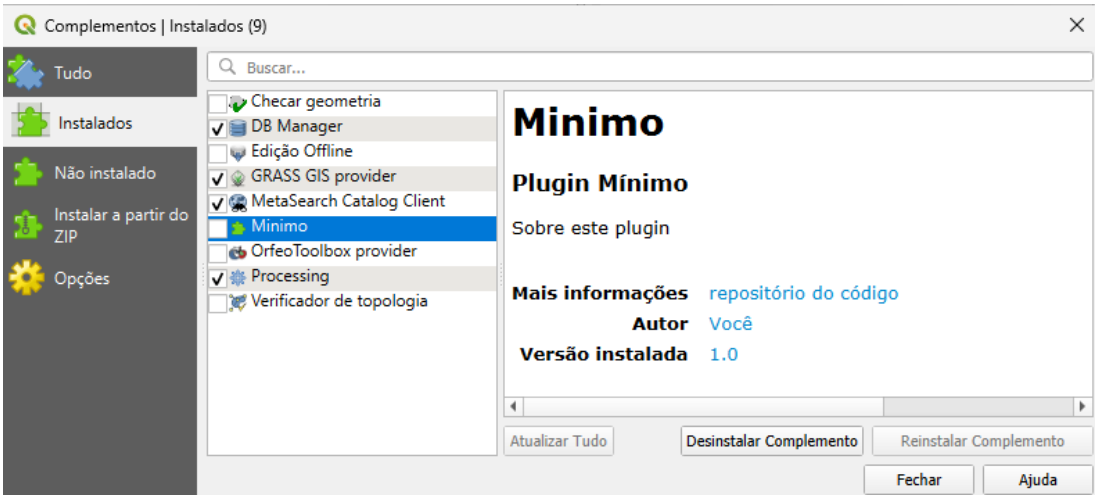
Arquivo **__init__.py**

```
#-----------------------------------------------------------------------
from PyQt5.QtWidgets import QAction, QMessageBox

def classFactory(iface):
    return Minimo(iface)

class Minimo:
    def __init__(self, iface):
        self.iface = iface

    def initGui(self):
        self.action = QAction('Teste', self.iface.mainWindow())
        self.action.triggered.connect(self.run)
        self.iface.addToolBarIcon(self.action)

    def unload(self):
        self.iface.removeToolBarIcon(self.action)
        del self.action

    def run(self):
        QMessageBox.information(None,'Plugin Mínimo','Kreegah
bundolo!')
```

Depois de ter criado os arquivos na nova pasta dentro da pasta plugins do QGIS, inicie o QGIS. Vá no menu **Complementos->Gerenciar e instalar Complementos**.



Selecione **Instalados** e o nosso plugin **Minimo** aparecerá na lista.

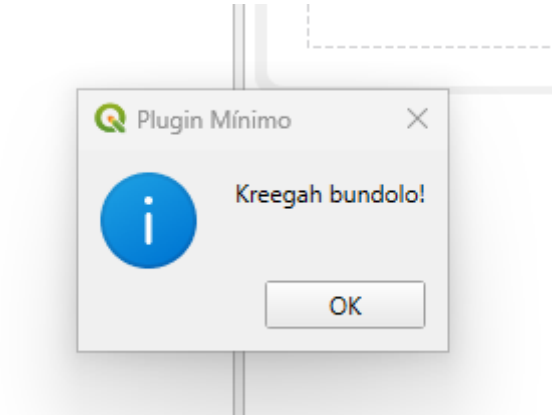


Selecione ele e feche.

O seguinte Icone “Teste” aparecerá na barra de ferramentas:

Clique nele.

Pronto, seu primeiro plugin com pyQGIS foi executado mostrando a seguinte mensagem:



O arquivo metadata.txt contém as informações sobre o plugin tais como quem criou, e-mail de quem criou, repositório web dos arquivos e informações que serão apresentadas quando adicionamos o plugin.

O arquivo __init__.py é requerido pelo sistema import do Python. Ele contém a função classFactory() que é chamada quando carregamos o plugin no QGIS.

Aqui colocamos o código fonte da classe do plugin dentro do arquivo também, mas normalmente a classe do plugin (class Minimo) iria em um arquivo separado. Veremos mais adiante como fazemos isso.

Dentro da classe Mínimo temos 4 funções:

__init__

Onde ganhamos acesso à interface do QGIS.

initGui

Essa função é chamada quando o plugin é carregado. Aqui se faz toda a inicialização das variáveis do plugin, inclusive as da interface gráfica do usuário (GUI) que detalharemos adiante.

unload

Essa função é chamada quando desativamos o plugin

run

Nessa função temos a execução do plugin propriamente dita.

Neste exemplo ainda não temos nenhuma interface gráfica de usuário (GUI). A seguir mostraremos como construir um plugin básico mais completo que esse.

2 - O básico

O plugin básico apresenta mais arquivos que são basicamente os diálogos (GUI) do plugin, os recursos para o diálogo e plugin.

O diálogo consiste em um arquivo XML com a extensão **.ui** que será transformado em um arquivo **.py** para ser acionado quando rodamos o plugin.

O arquivo de recurso é o **resources.qrc** que também é em XML e será transformado em arquivo **resources.py** para ser acionado ao executarmos o plugin. Um exemplo de recurso é um ícone para o plugin que aqui será o arquivo icon.png (download em https://gdatasystems.com/pyqgis/index.php).

Vamos separar a classe principal do plugin do arquivo __init__.py conforme mencionamos anteriormente, assim teremos basicamente que escrever o código do plugin nesse arquivo somente.

Na pasta de plugin crie uma nova pasta chamada **básico** e carregue os seguintes arquivos:

metadata.txt

```
[general]
name=Basico
description=Plugin Basico
about=Sobre este plugin
version=1.0
qgisMinimumVersion=3.0
author=Você
email=seu@email.com
repository=URL para o repositório do código na web
```

__init__.py

```
def classFactory(iface):
    from .basico import Basico
    return Basico(iface)
```

basico_dialog.py

```
import os
from PyQt5 import uic
from PyQt5 import QtWidgets

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


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

resouces.qrc

```
<RCC>
    <qresource prefix="/plugins/basico" >
        <file>icon.png</file>
    </qresource>
</RCC>
```

basico.ui **(**esse arquivo pode ser feito automaticamente com o QtDesigner que é instalado junto com o QGIS. Use este por praticidade, mostraremos como usar o QtDesigner mais adiante**.)**

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>Dialog</class>
 <widget class="QDialog" name="Dialog">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>201</width>
    <height>115</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Básico</string>
  </property>
  <widget class="QDialogButtonBox" name="buttonBox">
   <property name="geometry">
    <rect>
     <x>10</x>
     <y>50</y>
     <width>171</width>
     <height>32</height>
    </rect>
   </property>
   <property name="orientation">
    <enum>Qt::Horizontal</enum>
   </property>
   <property name="standardButtons">
    <set>QDialogButtonBox::Cancel|QDialogButtonBox::Ok</set>
   </property>
  </widget>
  <widget class="QgsFileWidget" name="mQgsFileWidget">
   <property name="geometry">
    <rect>
     <x>20</x>
     <y>10</y>
     <width>151</width>
     <height>27</height>
    </rect>
   </property>
  </widget>
 </widget>
```

```
 <customwidgets>
  <customwidget>
   <class>QgsFileWidget</class>
   <extends>QWidget</extends>
   <header>qgsfilewidget.h</header>
  </customwidget>
 </customwidgets>
 <resources/>
 <connections>
  <connection>
   <sender>buttonBox</sender>
   <signal>accepted()</signal>
   <receiver>Dialog</receiver>
   <slot>accept()</slot>
   <hints>
    <hint type="sourcelabel">
     <x>248</x>
     <y>254</y>
    </hint>
    <hint type="destinationlabel">
     <x>157</x>
     <y>274</y>
    </hint>
   </hints>
  </connection>
  <connection>
   <sender>buttonBox</sender>
   <signal>rejected()</signal>
   <receiver>Dialog</receiver>
   <slot>reject()</slot>
   <hints>
    <hint type="sourcelabel">
     <x>316</x>
     <y>260</y>
    </hint>
    <hint type="destinationlabel">
     <x>286</x>
     <y>274</y>
    </hint>
   </hints>
  </connection>
 </connections>
</ui>
```

basico.py

```
from PyQt5.QtCore import QSettings, QTranslator, QCoreApplication
from PyQt5.QtGui import QIcon
from PyQt5.QtWidgets import QAction, QMessageBox
from qgis.gui import QgsFileWidget
import os.path
from .resources import *
from .basico_dialog import BasicoDialog

class Basico:
    def __init__(self, iface):
```

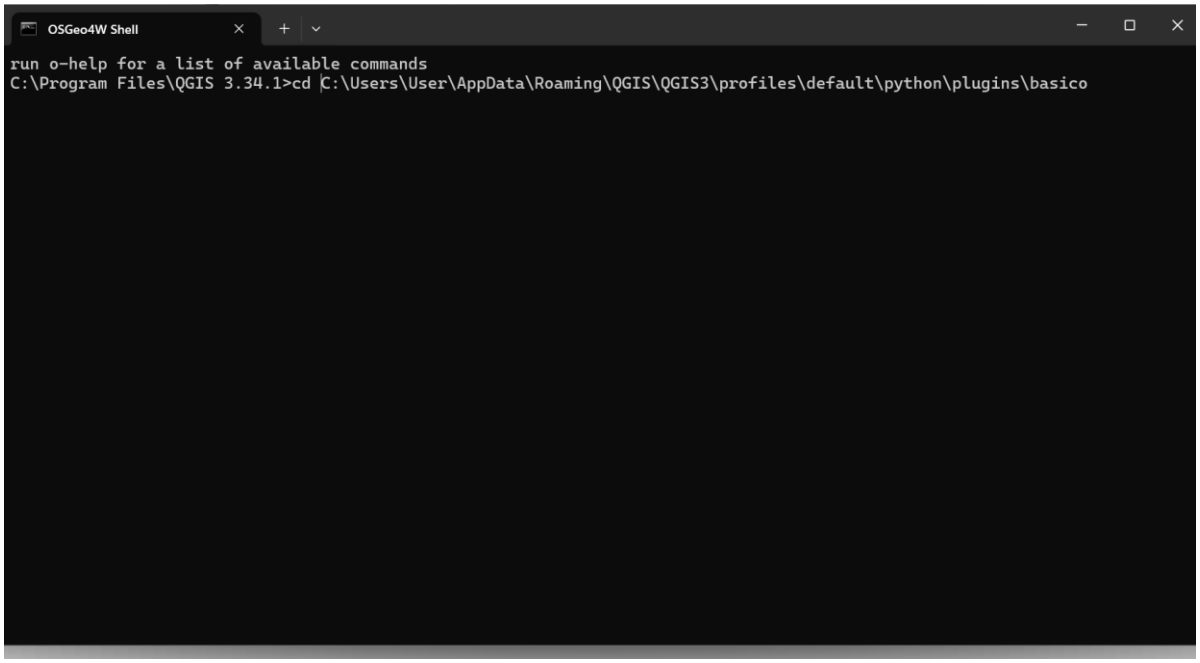
```
        self.iface = iface
        self.actions = []
        self.menu = '&Básico'
        self.first_start = None

    def add_action(
        self,
        icon_path,
        text,
        callback,
        enabled_flag=True,
        add_to_menu=True,
        add_to_toolbar=True,
        status_tip=None,
        whats_this=None,
        parent=None):
        icon = QIcon(icon_path)
        action = QAction(icon, text, parent)
        action.triggered.connect(callback)
        action.setEnabled(enabled_flag)

        if status_tip is not None:
            action.setStatusTip(status_tip)

        if whats_this is not None:
            action.setWhatsThis(whats_this)

        if add_to_toolbar:
            self.iface.addToolBarIcon(action)

        if add_to_menu:
            self.iface.addPluginToMenu(
                self.menu,
                action)

        self.actions.append(action)

        return action

    def initGui(self):
        icon_path = ':/plugins/basico/icon.png'
        self.add_action(
            icon_path,
            text='Básico',
            callback=self.run,
            parent=self.iface.mainWindow())

        self.first_start = True

    def unload(self):
        for action in self.actions:
            self.iface.removePluginMenu(
                '&Básico',
                action)
            self.iface.removeToolBarIcon(action)
```

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

        self.dlg.show()
        result = self.dlg.exec_()
        if result:
            arqui=self.dlg.mQgsFileWidget.filePath()
```

Copie na pasta também o arquivo **icon.png** que foi baixado.

Um último passo é gerar o arquivo **resouces.py** a partir do arquivo **resouces.qrc** acima. No Windows isso é feito usando um programa que foi instalado junto com o QGIS chamado **pyrcc5** que deve ser executado usando o OSGeo4w shell (linha de comando).

Navegue até a pasta onde fica o plugin usando cd C:\Users\.....

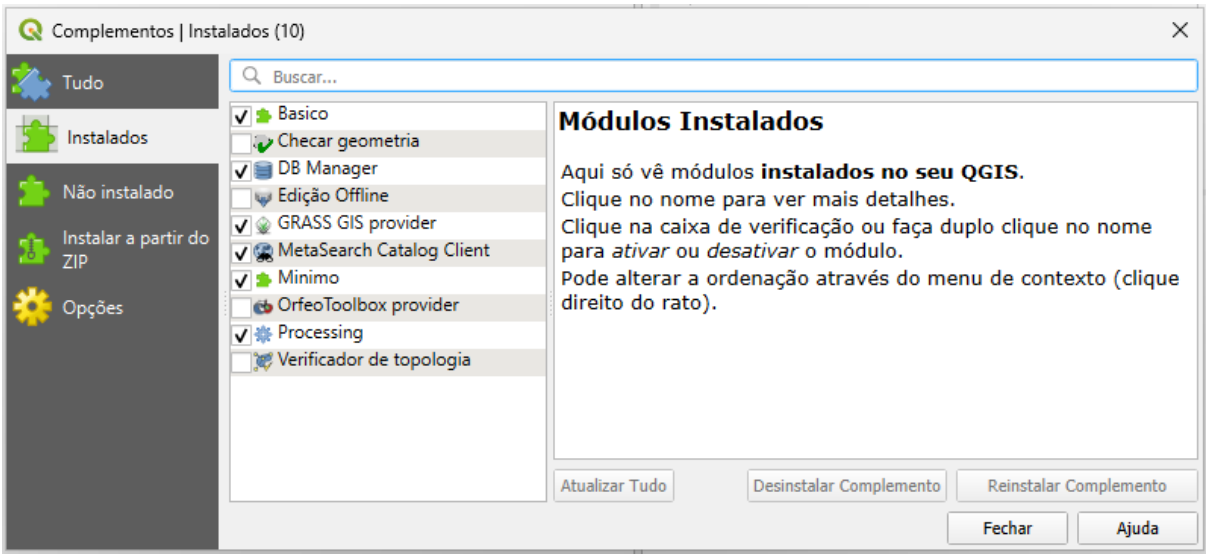


E digite:

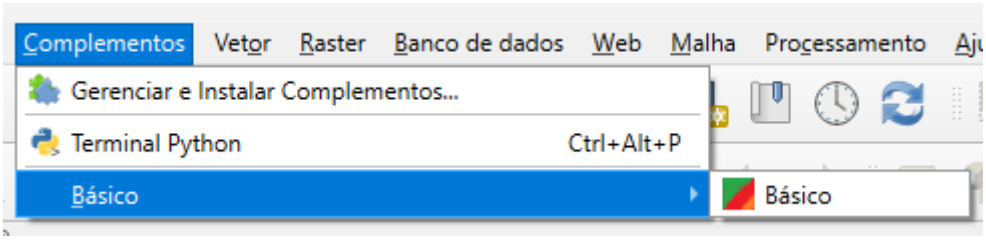
```
pyrcc5 -o resources.py resources.qrc
```

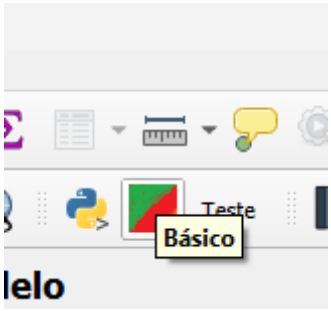
Assim o arquivo **resources.py** é criado na pasta certa. Pode digitar **exit** na linha de comando para fechar a shell.

Inicie o QGIS e instale o plugin **Basico** da mesma forma que fizemos com o plugin **Minimo** acima.

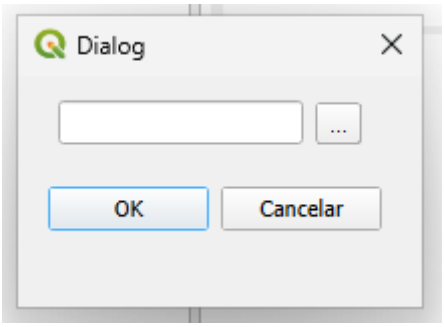


Um item no menu complementos e um ícone na barra de ferramenta serão criados para acessar o plugin.





Este plugin não faz nada no momento, foi só para exemplificar como construir um plugin diretamente.



No próximo módulo vamos criar o primeiro plugin funcional e usaremos o **QtDesigner** e o **PluginBuilder** para automatizar a criação de plugins.

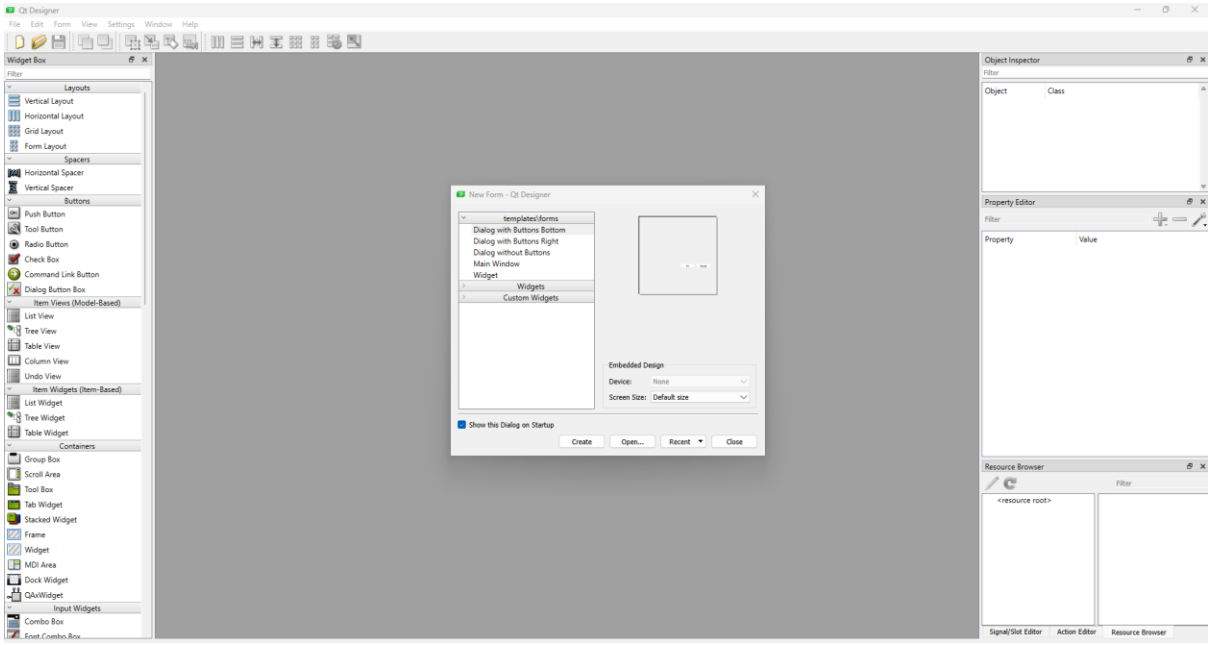
Criando Plugins QGIS com pyQGIS

QtDesigner, PluginBuilder e um plugin de verdade

1 - O QtDesiner e o PluginBuilder

Para facilitar a nossa vida existem duas ferramentas que auxiliam, e bastante, na criação de plugins. A primeira é o QtDesigner para construirmos as interfaces gráficas do usuário (GUI) e o PluginBuilder que cria praticamente todos os arquivos do plugin para nós.

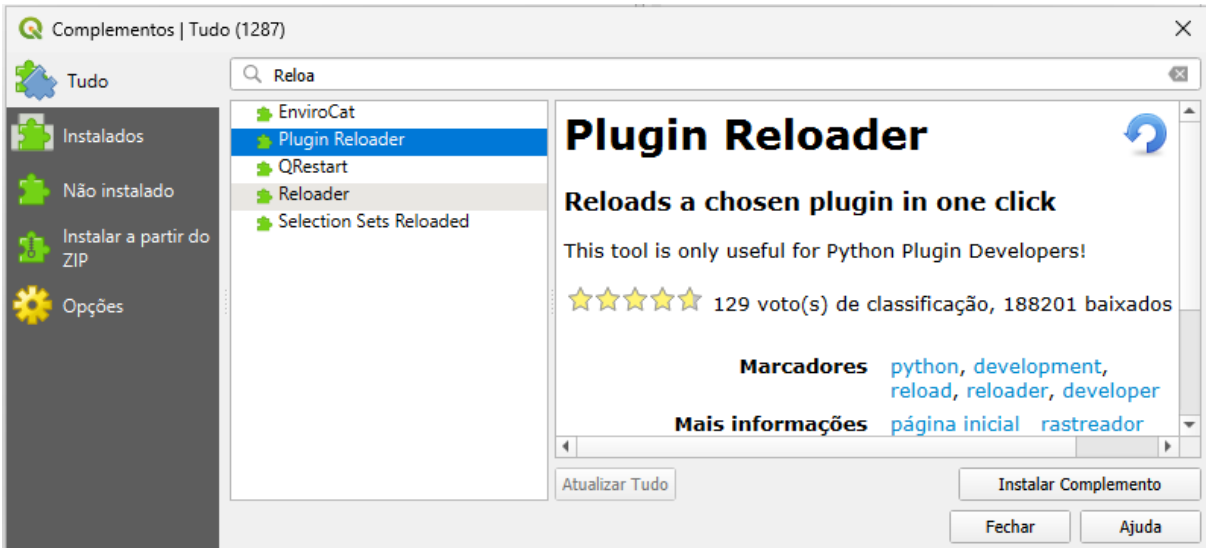
O QtDesigner já vem com o QGIS e a tela inicial dele é:



Usaremos ele para criar o nosso plugin de verdade deste módulo.

O PluginBuilder precisa ser instalado e fazemos isso abrindo o QGIS e indo menu **Complementos->Gerenciar e instalar Complementos**. Selecione o botão Tudo e digite builder na caixa de busca. O Plugin Builder 3 deve aparecer. Instale o complemento.

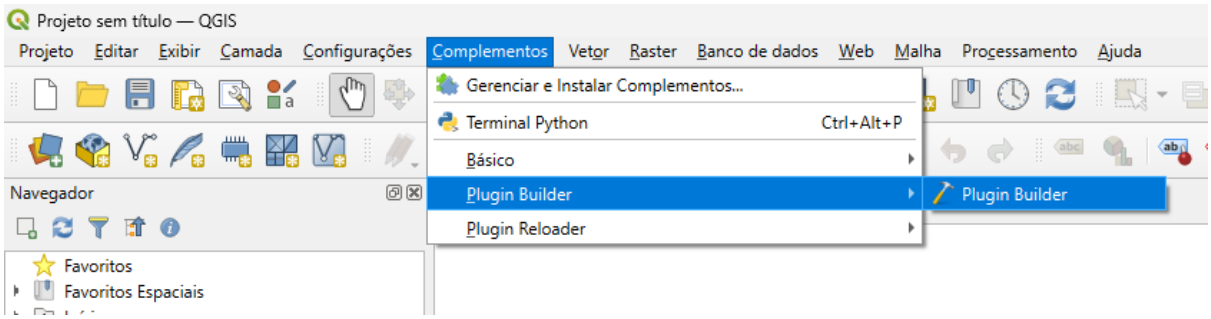
Outra ferramenta interessante de se ter é o Plugin Reloader. Instale ele também.



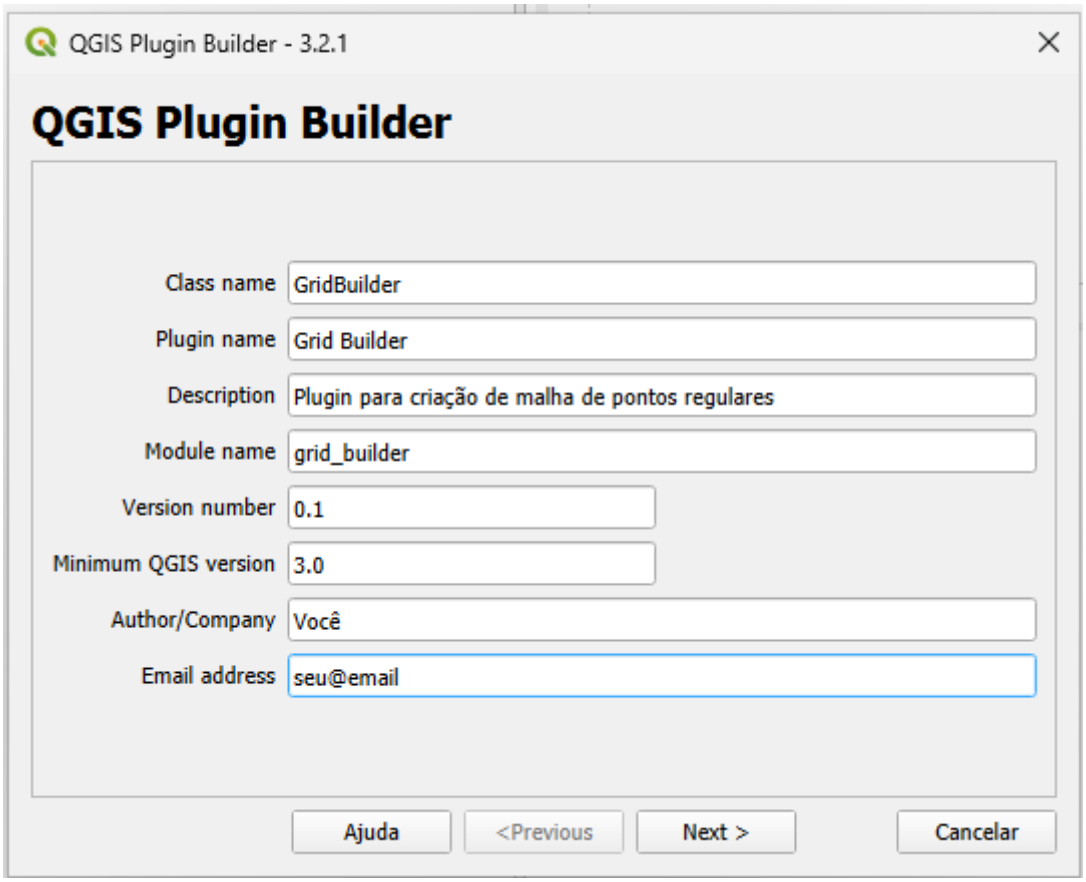
Vamos agora usar essas ferramentas na construção do nosso primeiro plugin funcional.

2 - Construindo o esqueleto do Grid_Builder no Plugin Builder 3

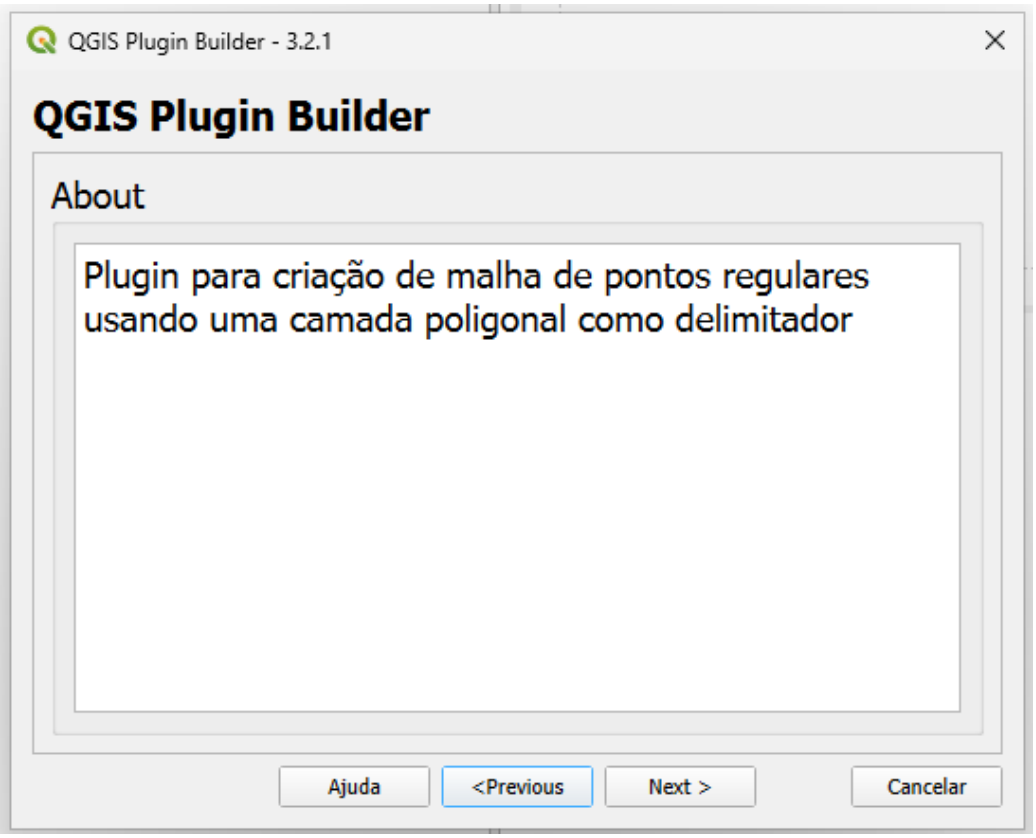
Vamos iniciar pelo Plugin Builder:



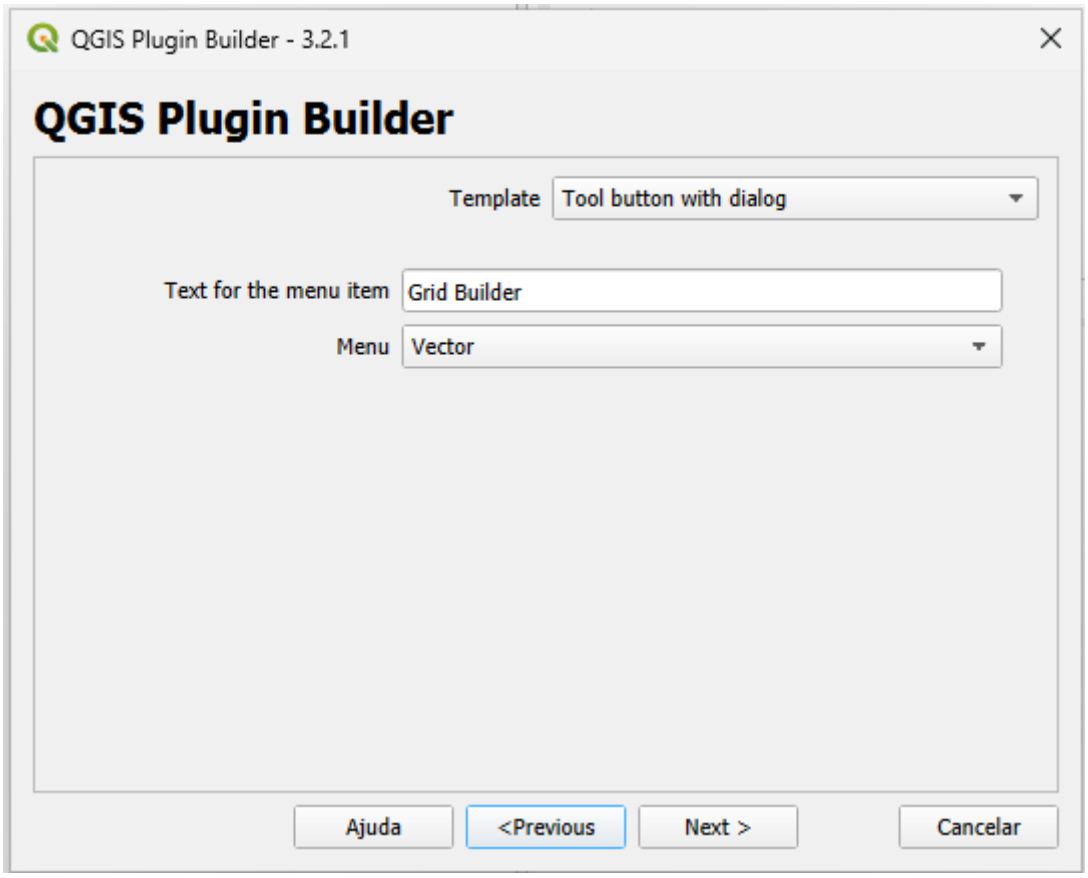
Inicie o plugin e preencha os campos dos formulários conforme as imagens a seguir:



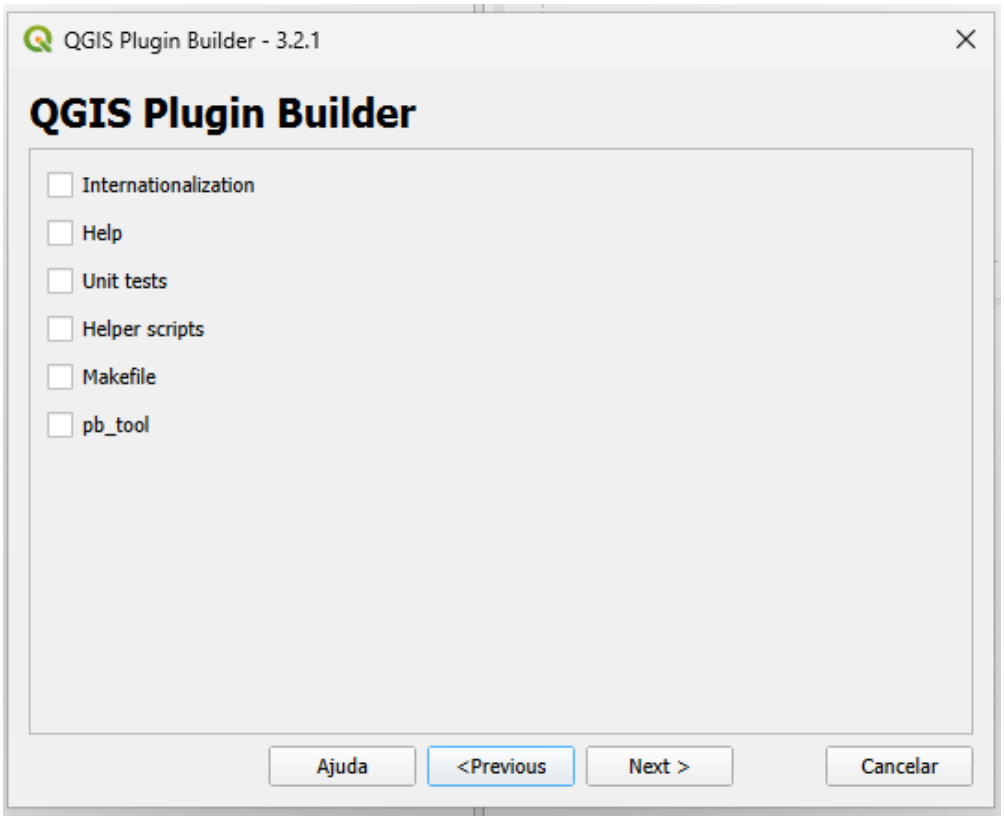
Esse primeiro formulário será usado na criação do arquivio metadata.txt e na definição do nome das classes do plugin.



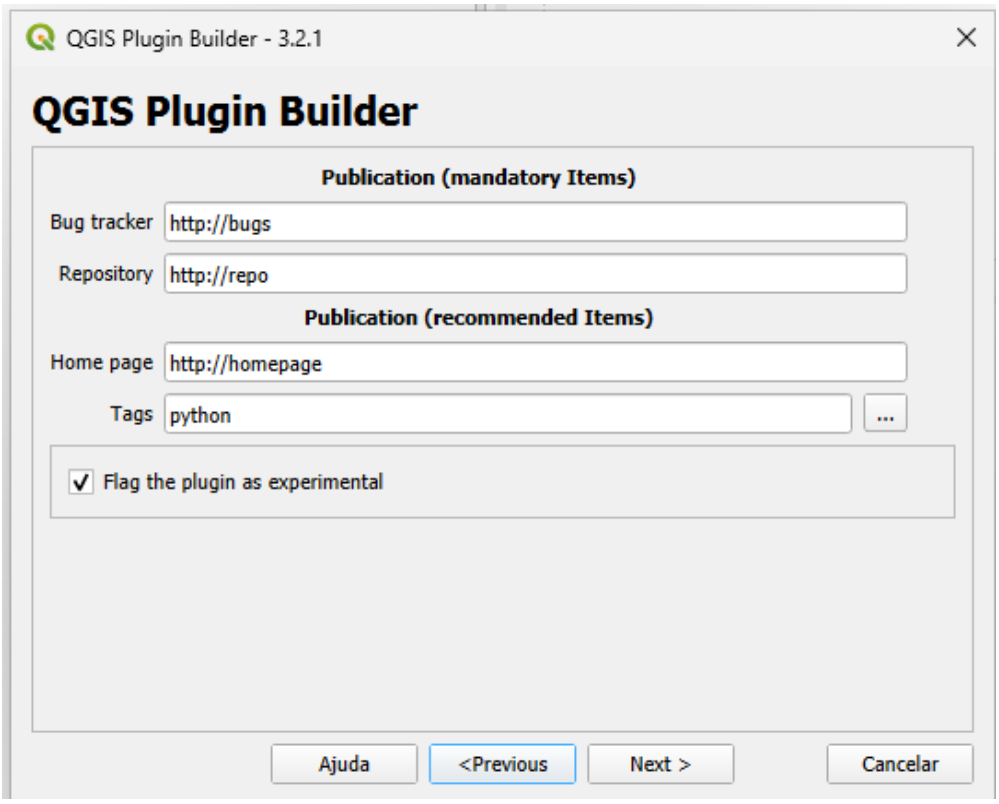
Descrição mais detalhada sobre o plugin que também será colocado no arquivo metadata.txt.



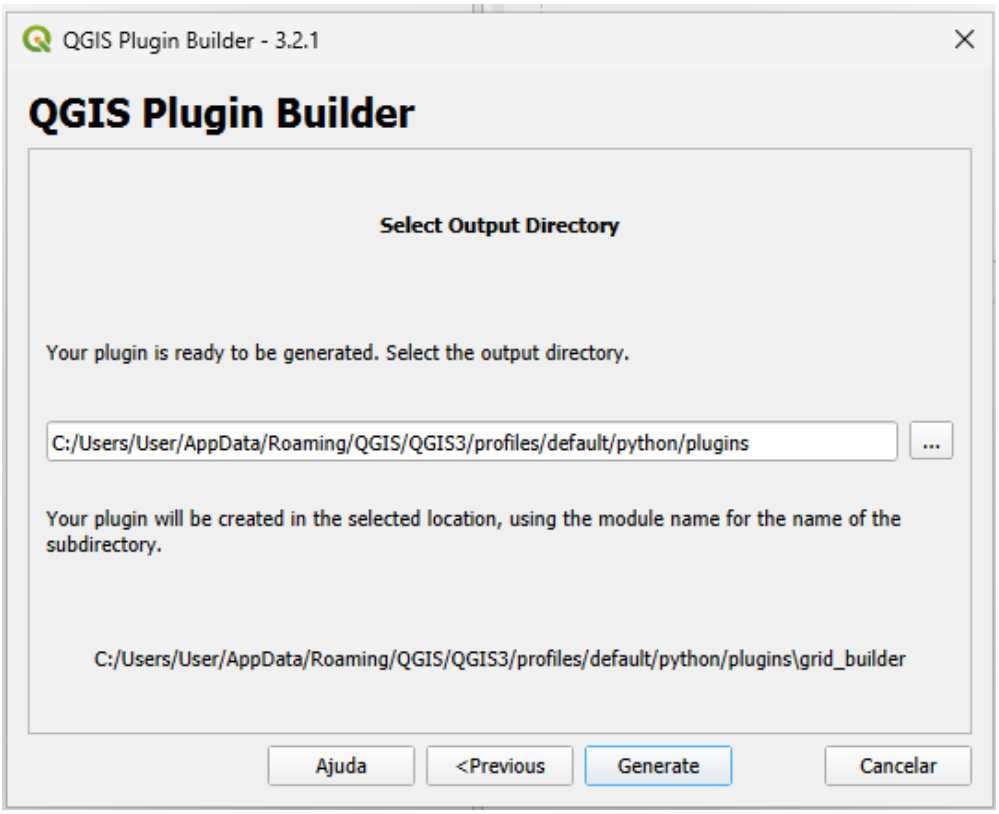
Template (tipo) do plugin, texto que vai aparecer no menu e em qual menu será listado o plugin.



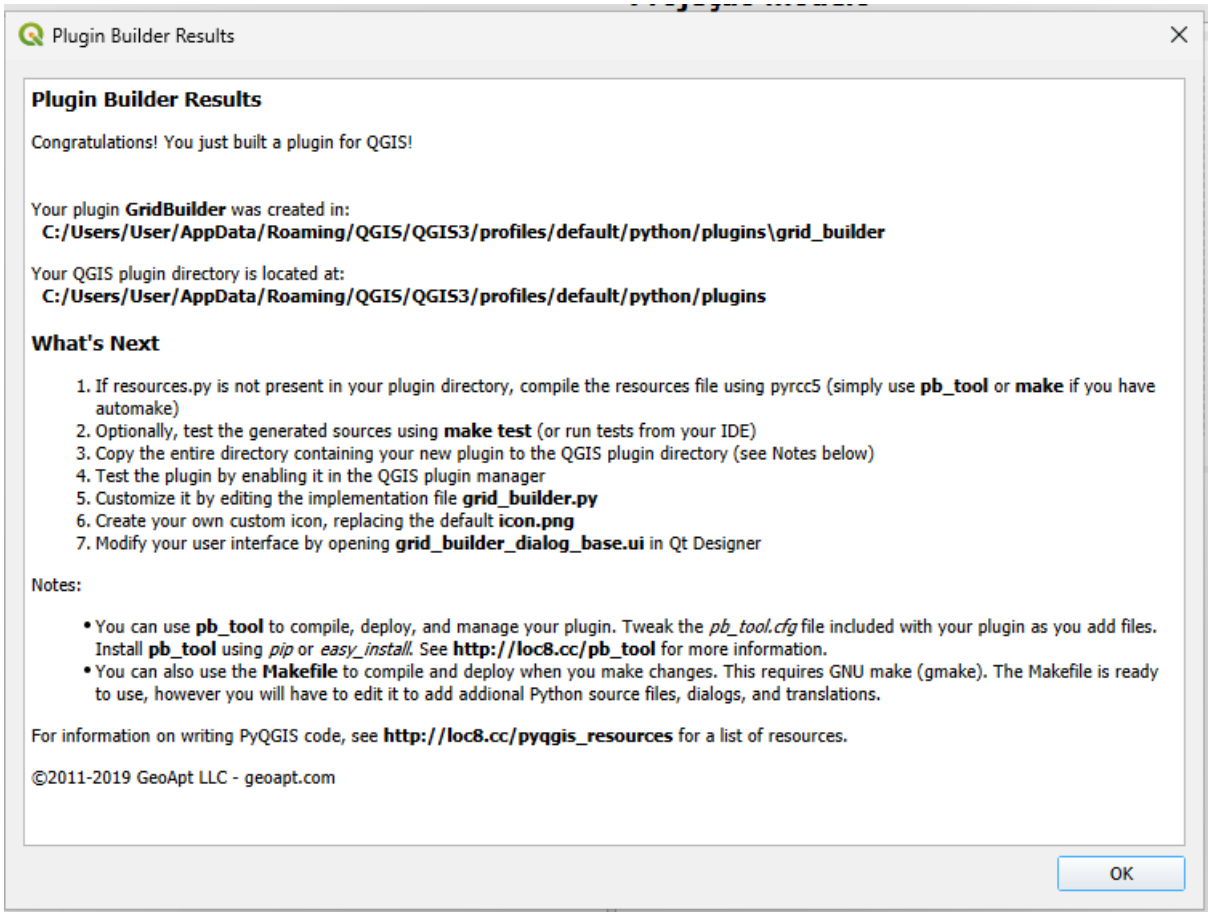
Desmarque todos para esse plugin,



Cheque a caixa de plugin experimental pois não iremos distribuir esse plugin no momento.

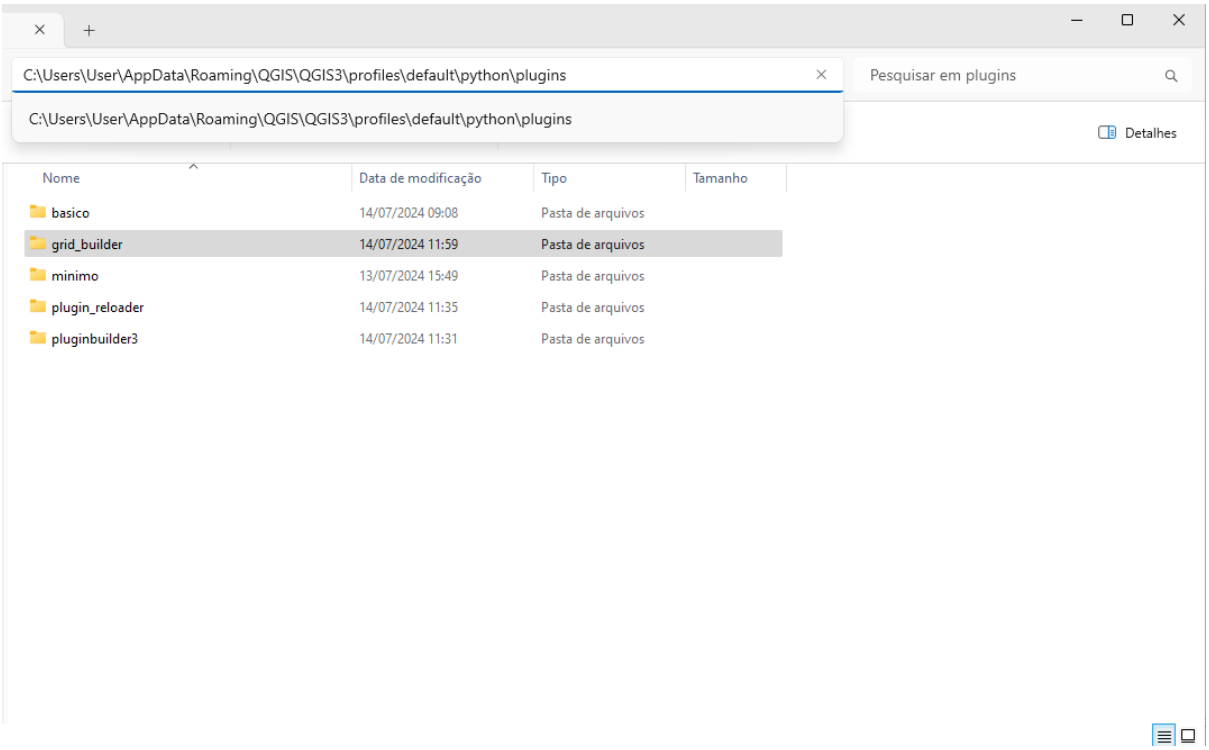


A pasta de plugin do sistema (nesse caso em sistema Windows). Clique **Generate** após selecionar o diretório de plugins.

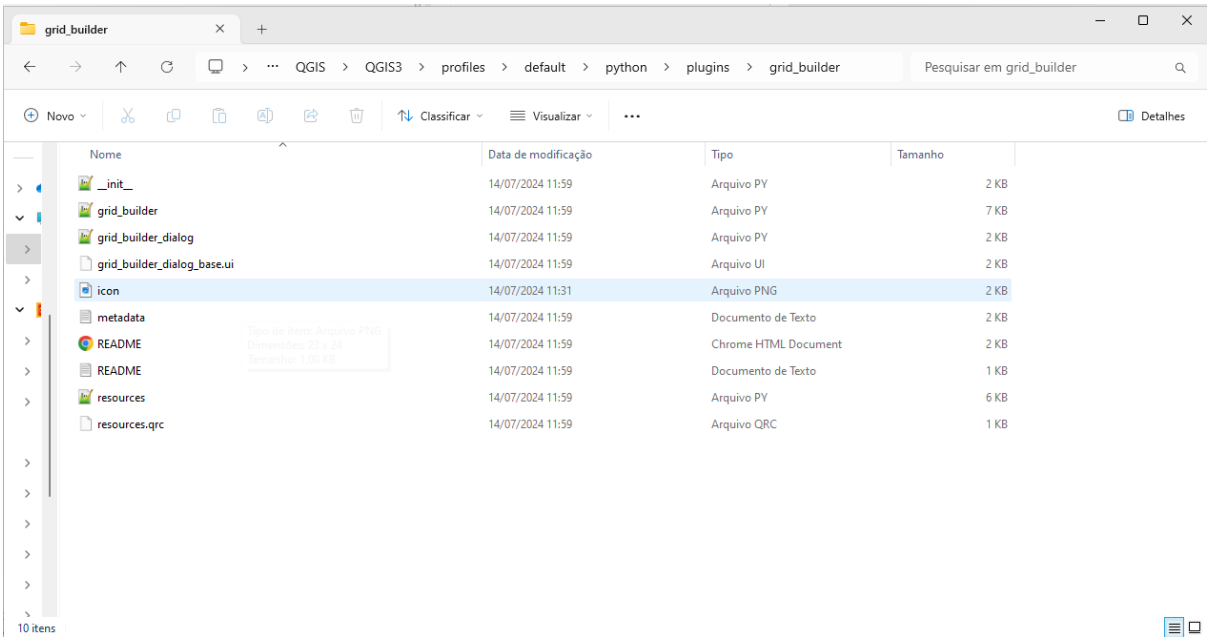


Pronto, os arquivos base de seu plugin foram criados na pasta:

C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins\grid_builder



Os arquivos gerados:



Os oito arquivos necessários mais dois arquivos README com instruções do PluginBuilder foram criados automaticamente.

Com base no README gerado abaixo vamos aos próximos passos.

Plugin Builder Results

Congratulations! You just built a plugin for QGIS!

Your plugin **GridBuilder** was created in:
C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins\grid_builder

Your QGIS plugin directory is located at:
C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins

What's Next

1. If resources.py is not present in your plugin directory, compile the resources file using pyrcc5 (simply use **pb_tool** or **make** if you have automake)
2. Optionally, test the generated sources using **make test** (or run tests from your IDE)
3. Copy the entire directory containing your new plugin to the QGIS plugin directory (see Notes below)
4. Test the plugin by enabling it in the QGIS plugin manager
5. Customize it by editing the implementation file **grid_builder.py**
6. Create your own custom icon, replacing the default **icon.png**
7. Modify your user interface by opening **grid_builder_dialog_base.ui** in Qt Designer

Notes:

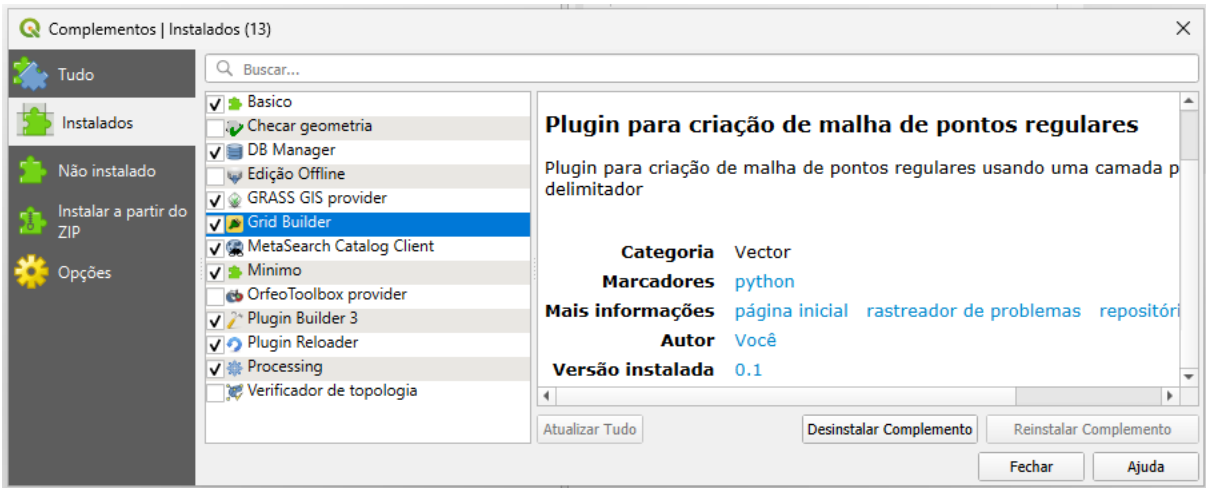
- You can use **pb_tool** to compile, deploy, and manage your plugin. Tweak the *pb_tool.cfg* file included with your plugin as you add files. Install **pb_tool** using *pip* or *easy_install*. See **http://loc8.cc/pb_tool** for more information.
- You can also use the **Makefile** to compile and deploy when you make changes. This requires GNU make (gmake). The Makefile is ready to use, however you will have to edit it to add addional Python source files, dialogs, and translations.

For information on writing PyQGIS code, see **http://loc8.cc/pyqgis_resources** for a list of resources.

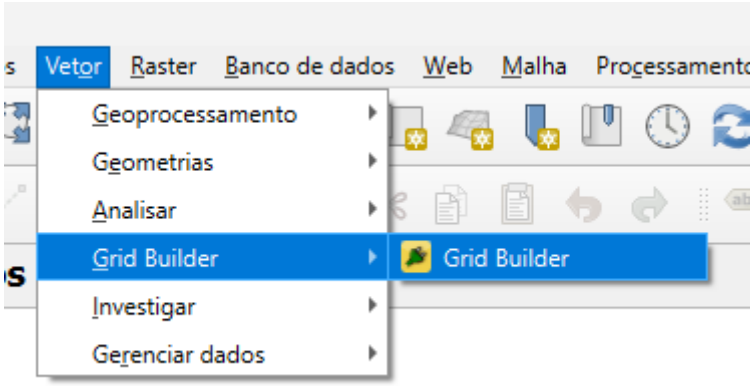
O arquivo resouces.py foi gerado automaticamente, desta forma não precisaremos de executar o pyrcc5, somente se quisermos mudar o icone padrão do PluginBuilder.

Os arquivos já estão no diretório correto de plugins.

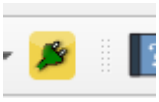
Vamos testar ele iniciando o QGIS e abrindo o **Complementos->Gerenciar e instalar Complementos**. Em Instalados vemos que ele não foi instalado ainda. Marque ele e instale para testarmos.



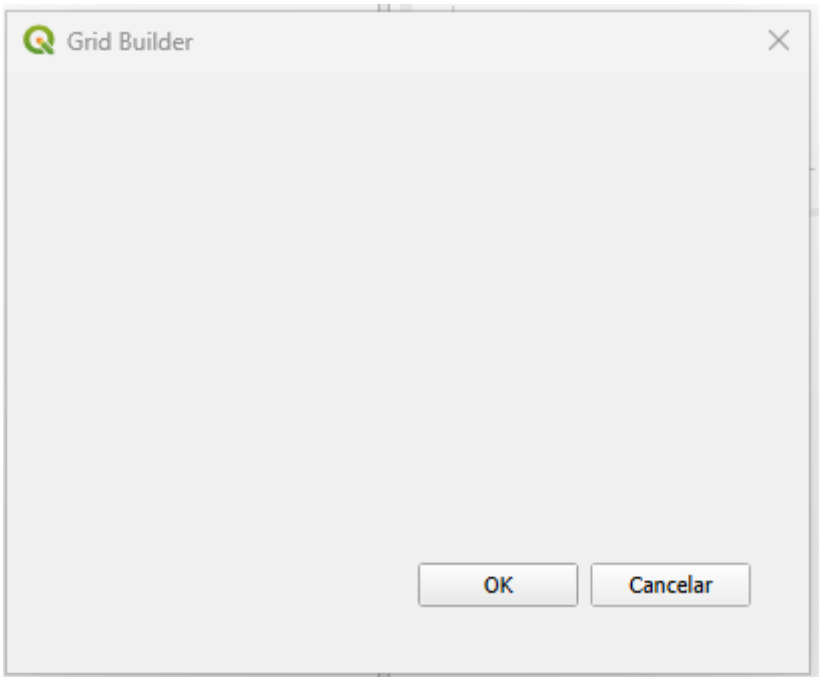
Inicie ele pelo Menu Vetor.



Ou pelo ícone na barra de ferramentas.



Funcionando, mas sem funcionalidade ainda.

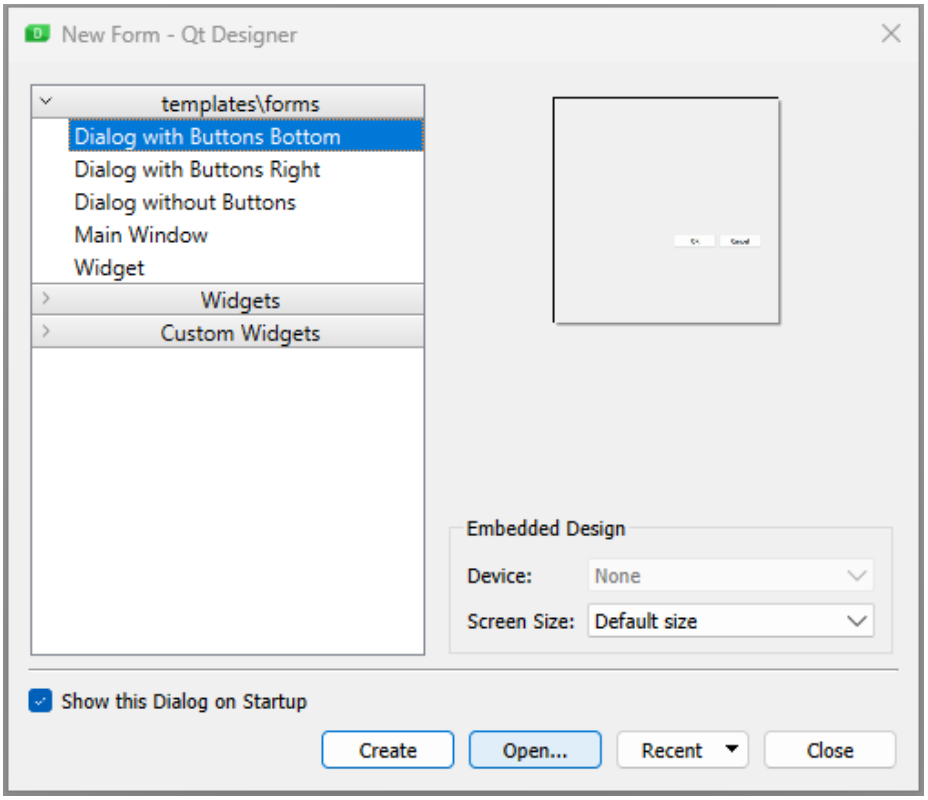


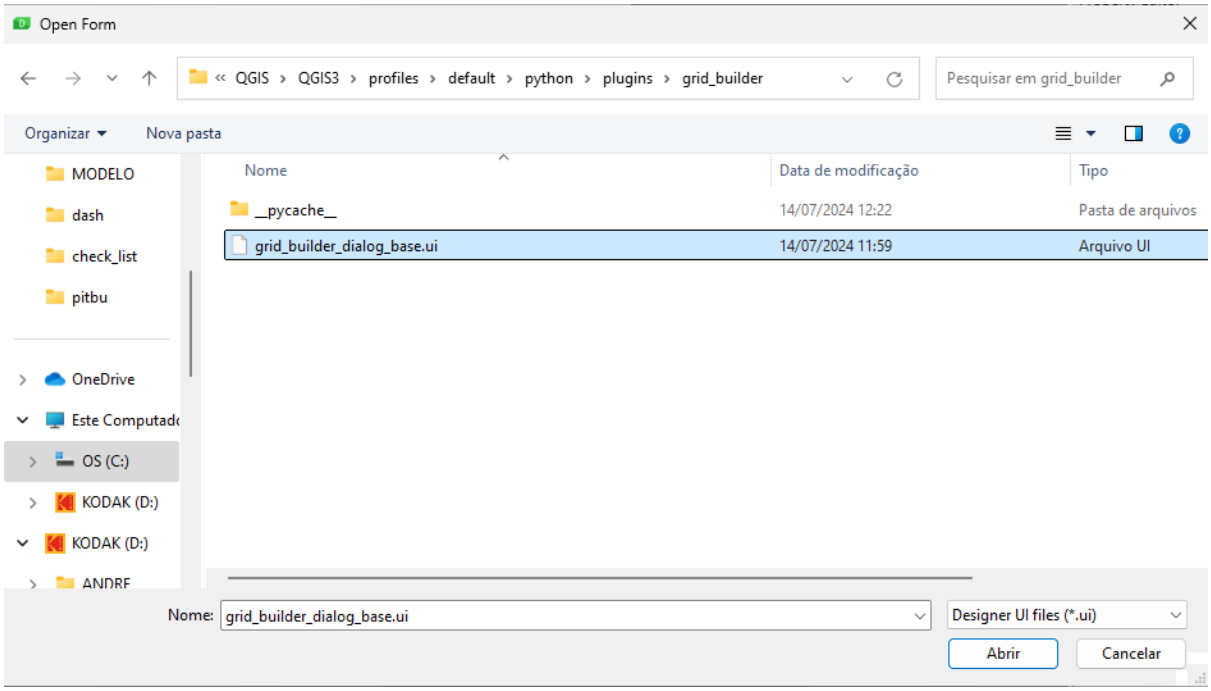
Vamos construir a interface gráfica do usuário (GUI) e adicionar a funcionalidade agora na próxima seção.

3 - O plugin Grid_Builder

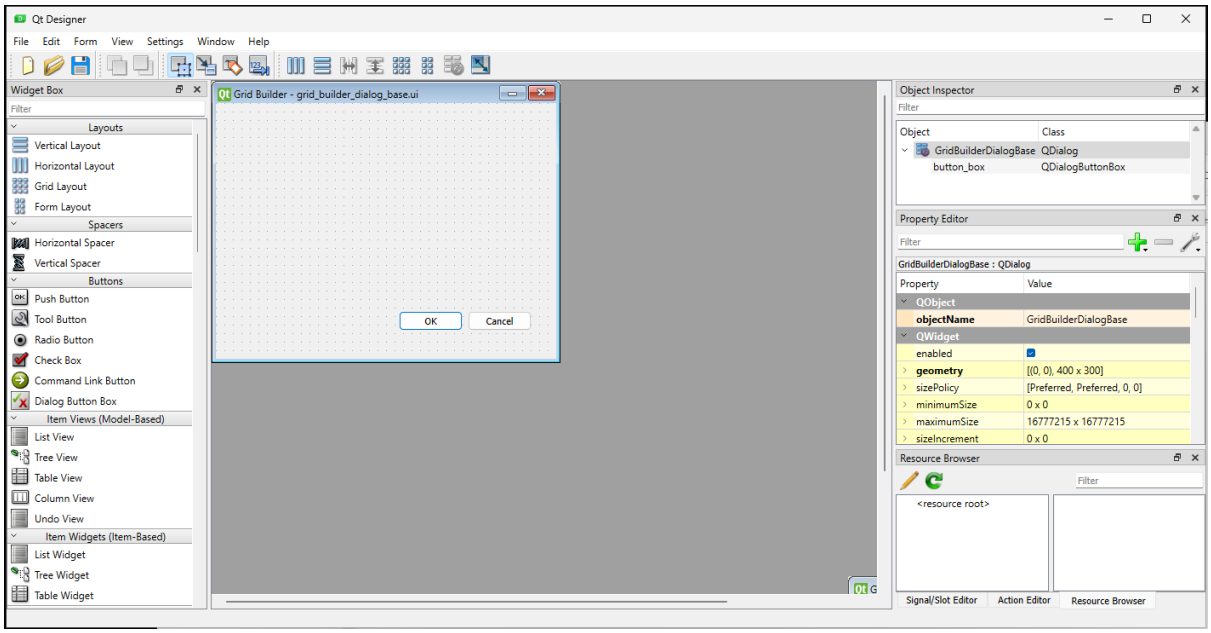
O plugin Grid Builder, como o próprio nome já fala constrói malha de pontos regulares usando uma camada espacial do tipo polígono para delimitar onde a malha será criada. Essa ferramenta, embora simples, é muito útil para criar malhas de amostragem.

Vamos iniciar pelo QtDesigner para criarmos a nossa interface gráfica de usuário (GUI). Abra o arquivo **grid_builder_dialog_base.ui** localizado em C:\Users\User\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins\grid_builder no primeiro diálogo do programa em **Open.**

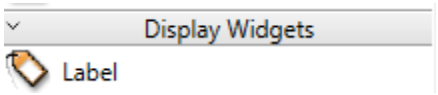


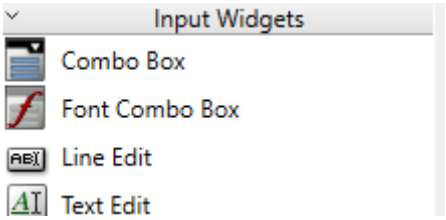


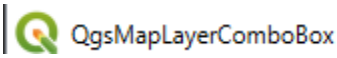
A seguinte tela aparecerá:



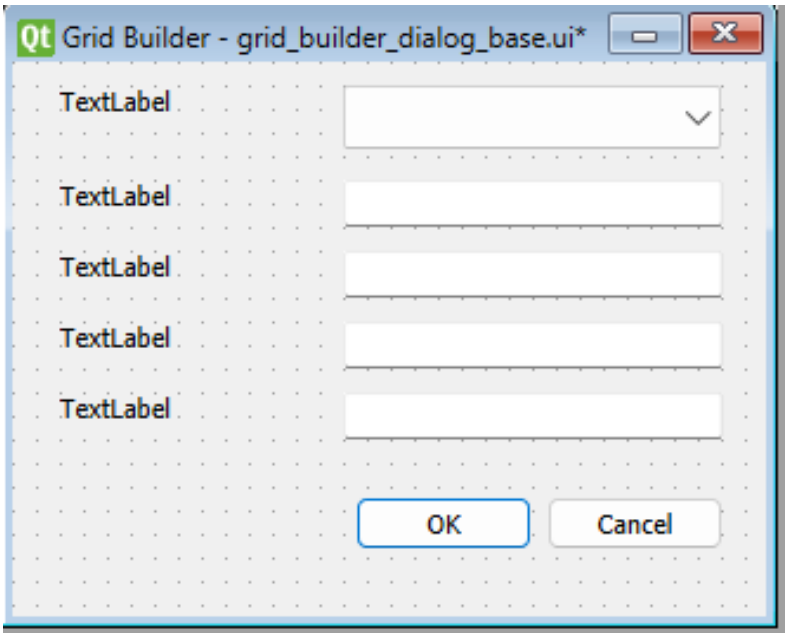
Vamos adicionar 5 widgets do tipo Label, 4 widgets do tipo Line Edit e um widget do tipo MapLayerComboBox. Basta clicar no Widget e arrastar até a janela do diálogo.



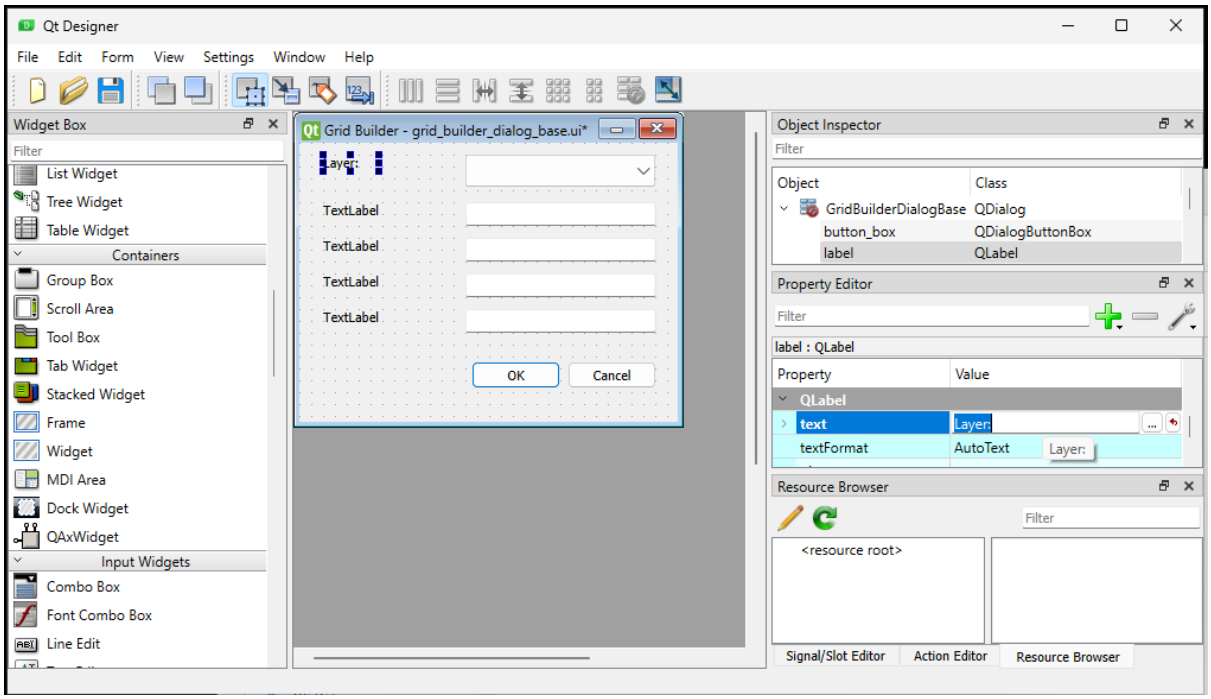




Teremos o diálogo mais ou menos com o seguinte formato:



Clique no Primeiro TextLabel e no property Editor altere o campo text de TextLabel para **Layer:** conforme abaixo:



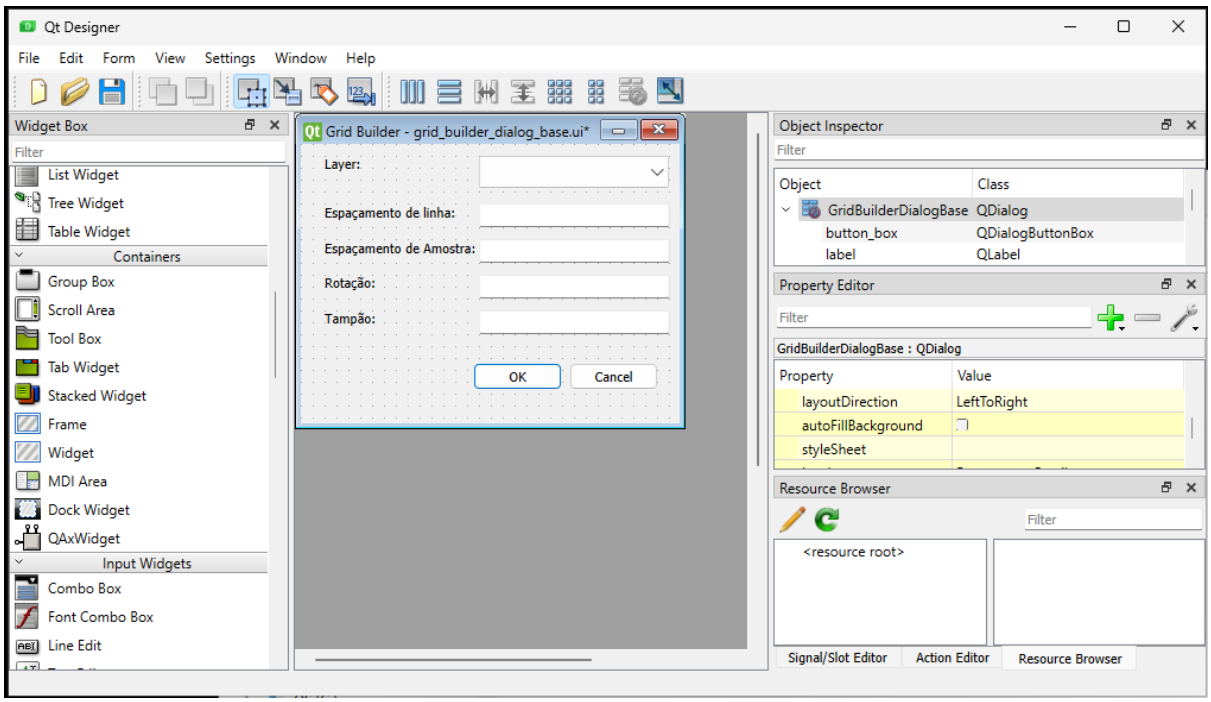
Repita o procedimento para os demais TextLabel renomeando eles para

Espaçamento de linha:

Espaçamento de Amostra:

Rotação:

Tampão:



O widget mMapLayerComboBox ficará inalterado, faremos a configuração pelo código python.

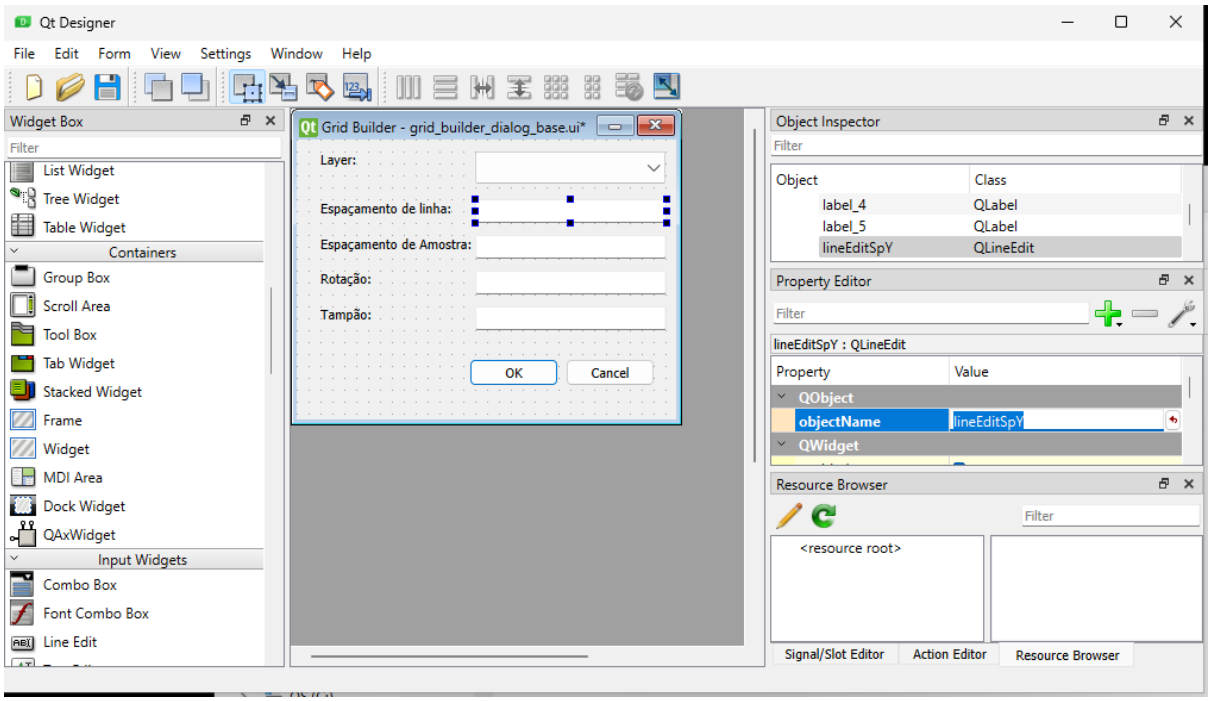
Agora altere a propriedade objectName dos campos LineEdit para:

lineEditSpY

lineEditSpX

lineEditRota

lineEditBuf



Pronto, salve o diálogo e feche o QtDesigner.

Vamos agora editar o arquivo **grid_builder.py** para realizar a tarefa. Vamos ter de adicionar algumas bibliotecas de suporte via **import** e adicionar o código na função **run** que vai fazer a validação inicial dos campos e a criação da camada de pontos da malha.

As bibliotecas serão (adicionar as faltantes):

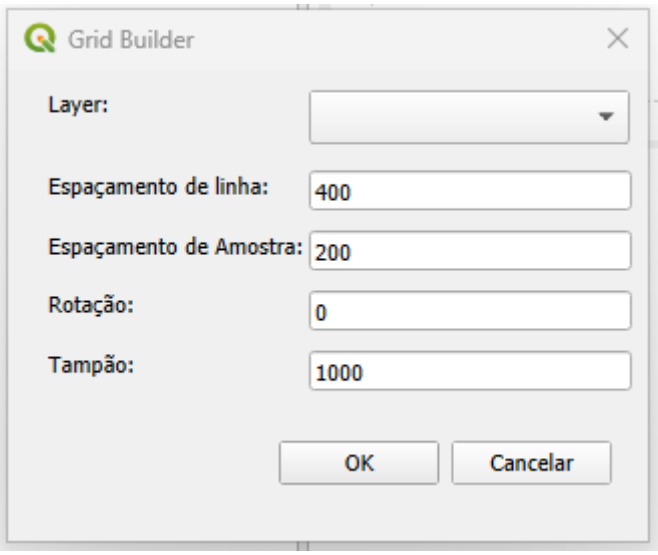
```
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication
from qgis.PyQt.QtGui import QIcon, QIntValidator
from qgis.PyQt.QtWidgets import QAction,QMessageBox

# Initialize Qt resources from file resources.py
from .resources import *
# Import the code for the dialog
from .grid_builder_dialog import GridBuilderDialog
import os.path
from qgis.core import QgsProject, QgsPointXY, QgsGeometry, QgsVectorLayer,QgsFeature
from qgis.core import QgsPoint, QgsField, QgsMapLayerType, QgsWkbTypes
```

A função **run** ficará assim:

```
def run(self):
        if self.first_start == True:
            self.first_start = False
            self.dlg = GridBuilderDialog()
            self.dlg.mMapLayerComboBox.setShowCrs(True)
```

```
            self.map_layers = QgsProject.instance().mapLayers().values()
            self.allow_list = [
                                lyr.id() for lyr in self.map_layers if lyr.type() ==
QgsMapLayerType.VectorLayer
                                and lyr.geometryType()== QgsWkbTypes.PolygonGeometry
                            ]
            self.except_list = [l for l in self.map_layers if l.id() not in self.allow_list]
            self.dlg.mMapLayerComboBox.setExceptedLayerList(self.except_list)
            onlyInt = QIntValidator()# mudar para QDoubleValidator se for usar lat long também
            self.dlg.lineEditSpY.setText('400')
            self.dlg.lineEditSpY.setValidator(onlyInt)
            self.dlg.lineEditSpX.setText('200')
            self.dlg.lineEditSpX.setValidator(onlyInt)
            self.dlg.lineEditRota.setText('0')
            self.dlg.lineEditRota.setValidator(onlyInt)
            self.dlg.lineEditBuf.setText('1000')
            self.dlg.lineEditBuf.setValidator(onlyInt)

        self.dlg.show()
        result = self.dlg.exec_()
        if result:
            if self.dlg.lineEditSpY.text()=='' or self.dlg.lineEditSpX.text()=='' or
self.dlg.lineEditRota.text()=='' or self.dlg.lineEditBuf.text()=='':
                QMessageBox.warning(self.iface.mainWindow(),
                         'Erro',
                         "Entre todos os campos por favor\nSaindo...")
                return
            layer = self.dlg.mMapLayerComboBox.currentLayer()
            feats = [ feat for feat in layer.getFeatures()]
            points = []
            spacing_y = int(self.dlg.lineEditSpY.text())
            spacing_x = int(self.dlg.lineEditSpX.text())
            rotacao= int(self.dlg.lineEditRota.text())
            extensao= int(self.dlg.lineEditBuf.text())
            #executar o código
            #------------------
            #
            for feat in feats:
                centroid = feat.geometry().centroid().asPoint()
                extent = feat.geometry().boundingBox()
                xmin=int(round(extent.xMinimum()-extensao, -2))
                ymin=int(round(extent.yMinimum()-extensao, -2))
                xmax=int(round(extent.xMaximum()+extensao, -2))
                ymax=int(round(extent.yMaximum()+extensao, -2))
                rows = int(((ymax) - (ymin))/spacing_y)
                cols = int(((xmax) - (xmin))/spacing_x)
                x = xmin
                y = ymax
                geom_feat = feat.geometry()
                for i in range(rows+1):
                    for j in range(cols+1):
                        pt = QgsPointXY(x,y)
                        tmp_pt = QgsGeometry.fromPointXY(pt)
                        tmp_pt.rotate(rotacao, centroid)
                        if tmp_pt.within(geom_feat):
                            points.append(tmp_pt.asPoint())
                        x += spacing_x
                    x = xmin
                    y -= spacing_y

            epsg = layer.crs().postgisSrid()
            #gerando pontos
            uri = "PointZ?crs=epsg:" + str(epsg) + "&field=id:integer""&index=yes"
            mem_layer = QgsVectorLayer(uri,'gridpoints','memory')
            prov = mem_layer.dataProvider()
            feats = [ QgsFeature() for i in range(len(points)) ]
            for i, feat in enumerate(feats):
                feat.setAttributes([i])
                feat.setGeometry(QgsPoint(points[i].x(),points[i].y(),0.0))

            prov.addFeatures(feats)
            QgsProject.instance().addMapLayer(mem_layer)
            QMessageBox.information(self.iface.mainWindow(),
                         'Pronto',
                         "Pontos de amostragem criados!")
            Return
```

Antes de comentarmos o código adicionado vamos testar o plugin. Abra o QGIS e o plugin será carregado já com as alterações feitas. Ao iniciarmos o plugin teremos:

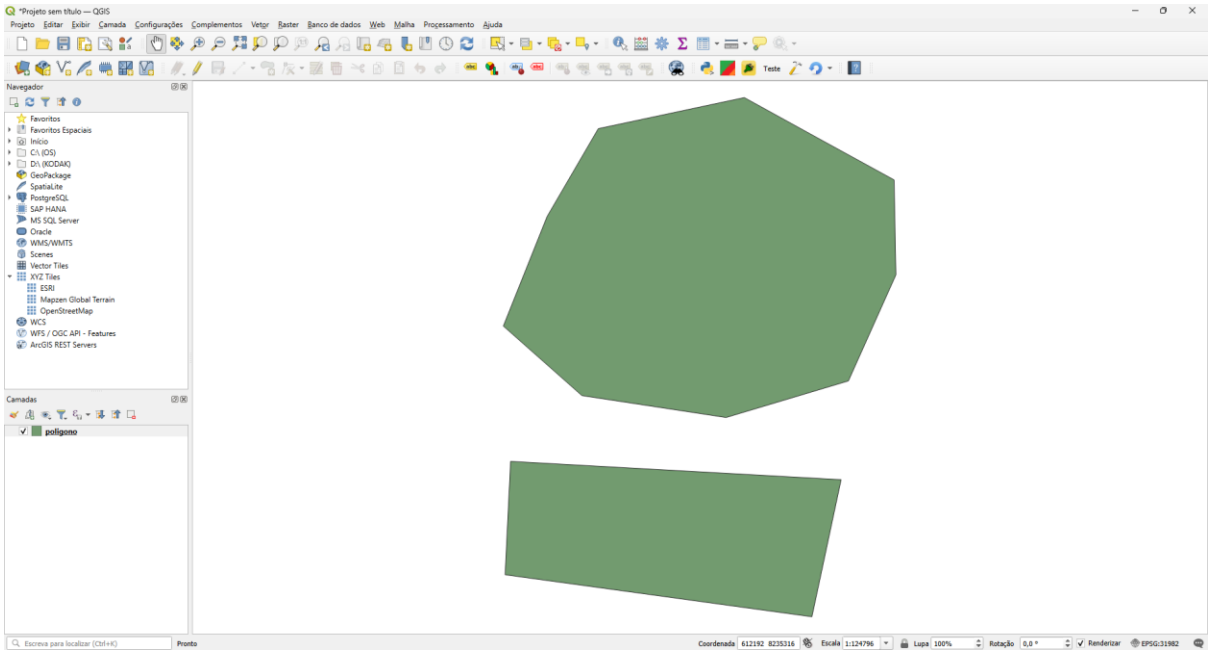


Primeiro crie ou abra uma camada do tipo polígono para executar o plugin Grid Builder.

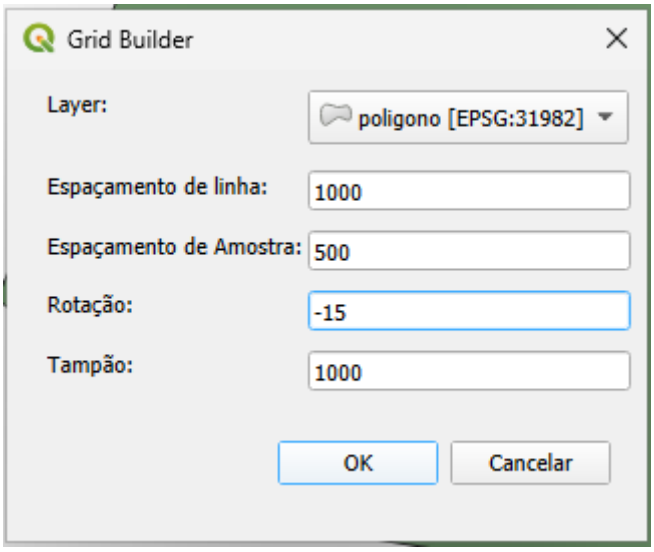
Clique em cancelar e crie um polígono (em coordenadas UTM, lat long funciona, mas os espaçamentos e o tampão devem ser entrados como decimal de grau e os campos de entrada validam somente inteiros, para aceitar decimais é preciso mudar o "validator" para QDoubleValidator no código).

Caso prefira carregue a camada polígono do site em :
https://gdatasystems.com/pyqgis/index.php

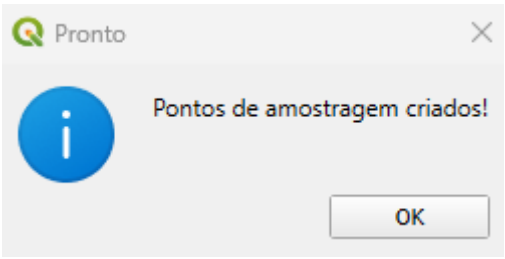
Camada polígono carregada.

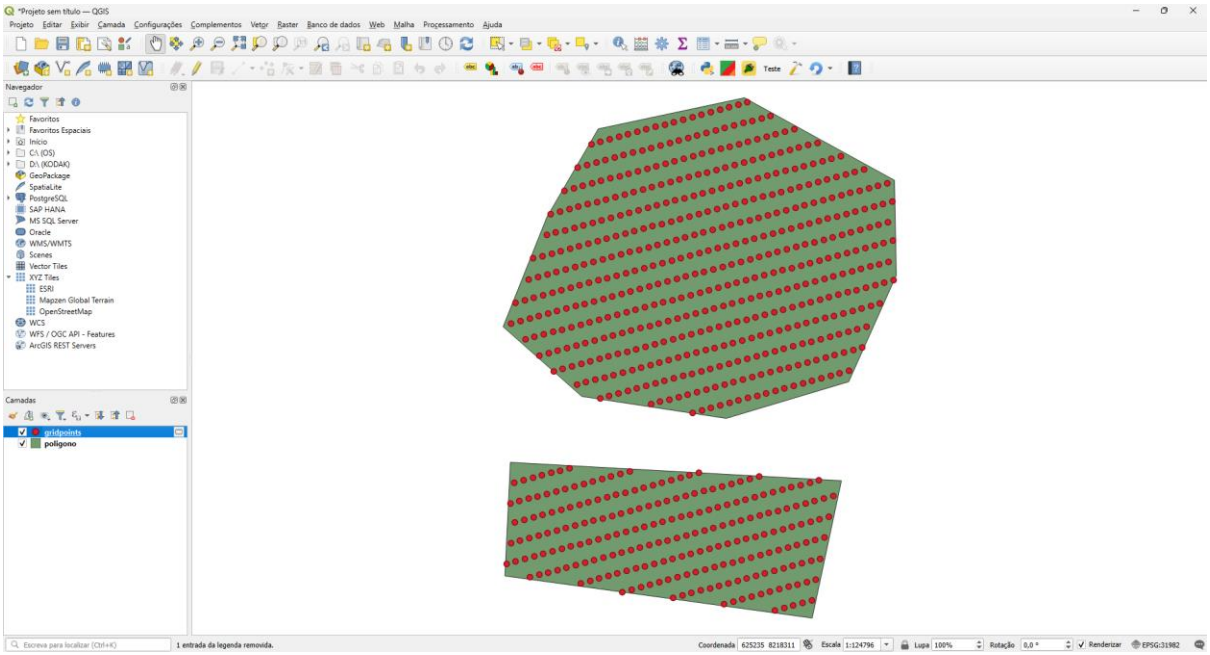


Execute o plugin com os seguintes parâmetros:



A seguinte mensagem aparece e a camada temporária gridpoint é criada.





Agora é só salvar a camada criada.

Este bloco do código inicializa os widgets do plugin carregando os polígonos abertos no combo box e adicionando os valores iniciais nas linhas editáveis.

```
def run(self):
        if self.first_start == True:
            self.first_start = False
            self.dlg = GridBuilderDialog()
            self.dlg.mMapLayerComboBox.setShowCrs(True)
            self.map_layers = QgsProject.instance().mapLayers().values()
            self.allow_list = [
                                lyr.id() for lyr in self.map_layers if lyr.type() ==
QgsMapLayerType.VectorLayer
                                and lyr.geometryType()== QgsWkbTypes.PolygonGeometry
                           ]
            self.except_list = [l for l in self.map_layers if l.id() not in self.allow_list]
            self.dlg.mMapLayerComboBox.setExceptedLayerList(self.except_list)
            onlyInt = QIntValidator()mudar para QDoubleValidator se for usar lat long também
            self.dlg.lineEditSpY.setText('400')
            self.dlg.lineEditSpY.setValidator(onlyInt)
            self.dlg.lineEditSpX.setText('200')
            self.dlg.lineEditSpX.setValidator(onlyInt)
            self.dlg.lineEditRota.setText('0')
            self.dlg.lineEditRota.setValidator(onlyInt)
            self.dlg.lineEditBuf.setText('1000')
            self.dlg.lineEditBuf.setValidator(onlyInt)
```

O bloco seguinte checa se os campos estão todos preenchidos e assinala eles às variáveis do programa.

```
        if result:
            if self.dlg.lineEditSpY.text()=='' or self.dlg.lineEditSpX.text()=='' or
self.dlg.lineEditRota.text()=='' or self.dlg.lineEditBuf.text()=='':
                QMessageBox.warning(self.iface.mainWindow(),
                         'Erro',
                         "Entre todos os campos por favor\nSaindo...")
                return
            layer = self.dlg.mMapLayerComboBox.currentLayer()
            feats = [ feat for feat in layer.getFeatures()]
            points = []
            spacing_y = int(self.dlg.lineEditSpY.text())
            spacing_x = int(self.dlg.lineEditSpX.text())
            rotacao= int(self.dlg.lineEditRota.text())
            extensao= int(self.dlg.lineEditBuf.text())
```

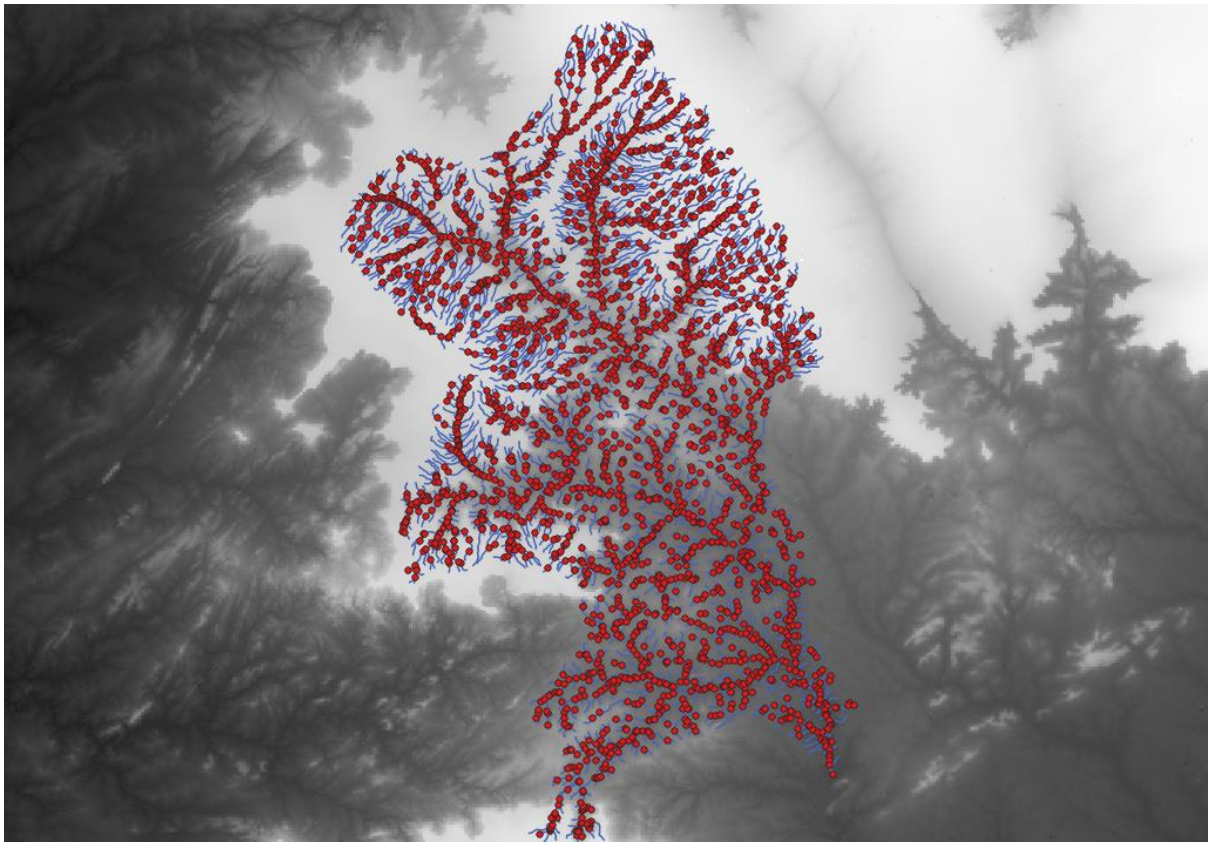
Aqui lemos o polígono e assinalamos sua extensão, calculamos o buffer e checamos qual sistema de coordenada devemos usar para gerar os pontos.

```
            #executar o código
            #------------------
            #
            for feat in feats:
                centroid = feat.geometry().centroid().asPoint()
                extent = feat.geometry().boundingBox()
                xmin=int(round(extent.xMinimum()-extensao, -2))
                ymin=int(round(extent.yMinimum()-extensao, -2))
                xmax=int(round(extent.xMaximum()+extensao, -2))
                ymax=int(round(extent.yMaximum()+extensao, -2))
                rows = int(((ymax) - (ymin))/spacing_y)
                cols = int(((xmax) - (xmin))/spacing_x)
                x = xmin
                y = ymax
```

Finalmente criamos os pontos e carregamos eles em um arquivo na memória.

```
geom_feat = feat.geometry()
    for i in range(rows+1):
        for j in range(cols+1):
            pt = QgsPointXY(x,y)
            tmp_pt = QgsGeometry.fromPointXY(pt)
            tmp_pt.rotate(rotacao, centroid)
            if tmp_pt.within(geom_feat):
                points.append(tmp_pt.asPoint())
            x += spacing_x
        x = xmin
        y -= spacing_y

epsg = layer.crs().postgisSrid()
uri = "PointZ?crs=epsg:" + str(epsg) + "&field=id:integer""&index=yes"
mem_layer = QgsVectorLayer(uri,'gridpoints','memory')
prov = mem_layer.dataProvider()
feats = [ QgsFeature() for i in range(len(points)) ]
for i, feat in enumerate(feats):
    feat.setAttributes([i])
    feat.setGeometry(QgsPoint(points[i].x(),points[i].y(),0.0))

prov.addFeatures(feats)
QgsProject.instance().addMapLayer(mem_layer)
QMessageBox.information(self.iface.mainWindow(),
             'Pronto',
             "Pontos de amostragem criados!")
Return
```

No próximo módulo avançaremos um pouco mais com um novo exemplo de plugin que usa uma biblioteca externa, não padrão do QGIS.



Criando Plugins QGIS com pyQGIS

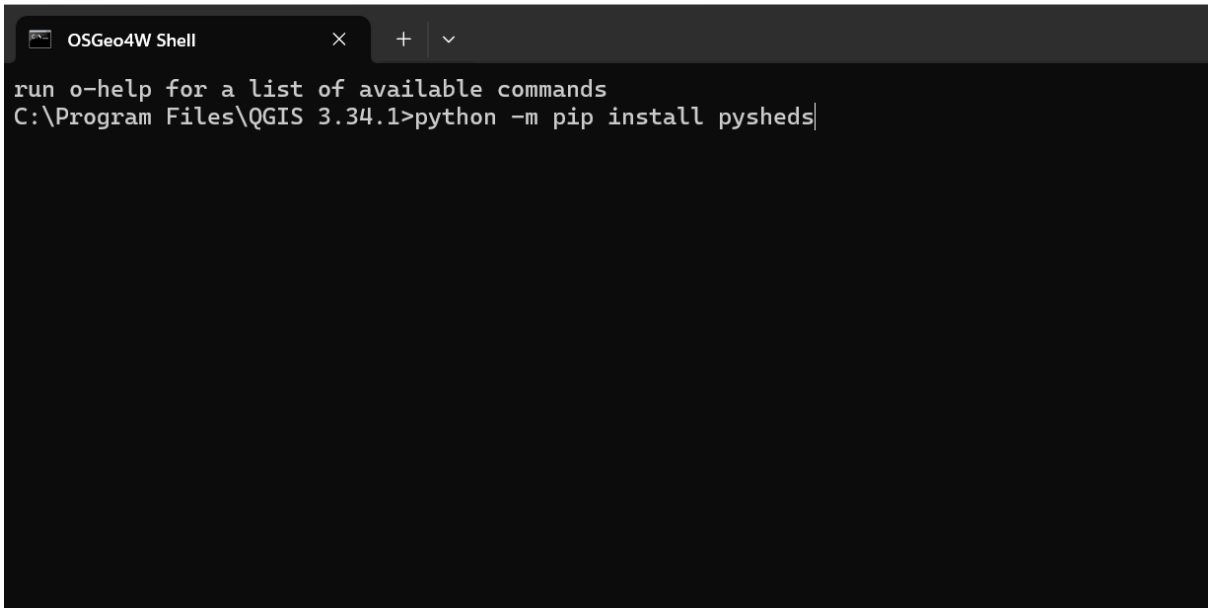
Usando bibliotecas externas no ambiente pyQGIS. Novo plugin Stream Builder

1 - O ambiente pyQGIS

O QGIS possui um ambiente python dedicado e para instalar novas bibliotecas usamos o shell OSGeo4W. Nesse exemplo de plugin vamos utilizar as bibliotecas **pysheds** e **fiona**.

Inicie o shell e digite:

```
python -m pip install pysheds
```



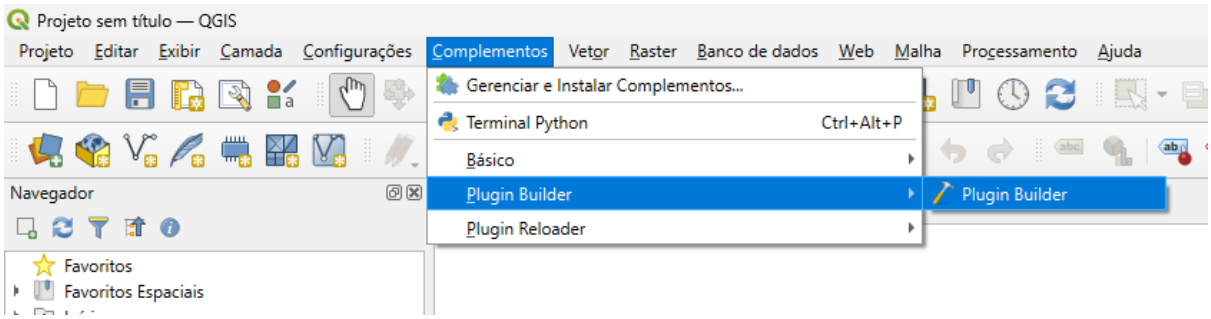
Após instalar o pysheds repita o procedimento instalando a biblioteca fiona:

```
python -m pip install fiona
```

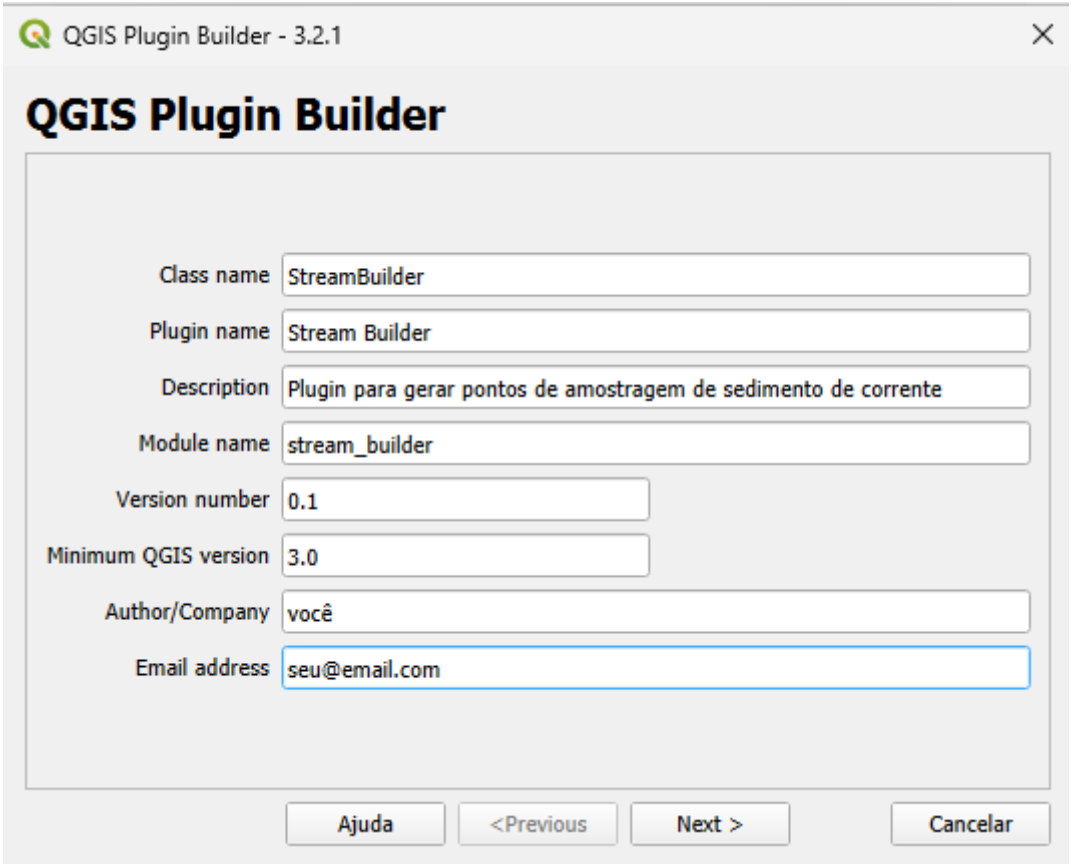
As bibliotecas necessárias para nosso plugin foram instaladas. Procederemos agora com a construção do plugin novo.

2 - Construindo o esqueleto do Stream_Builder no Plugin Builder 3

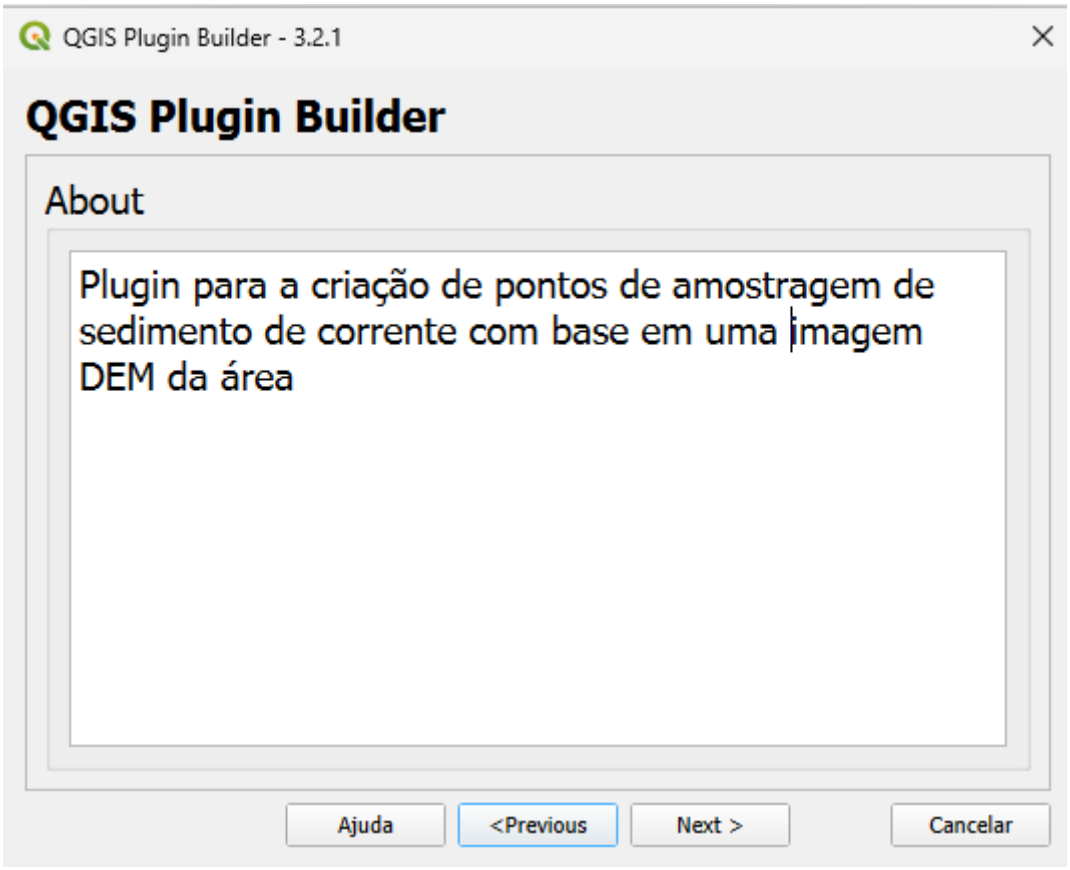
Inicie o Plugin Builder:



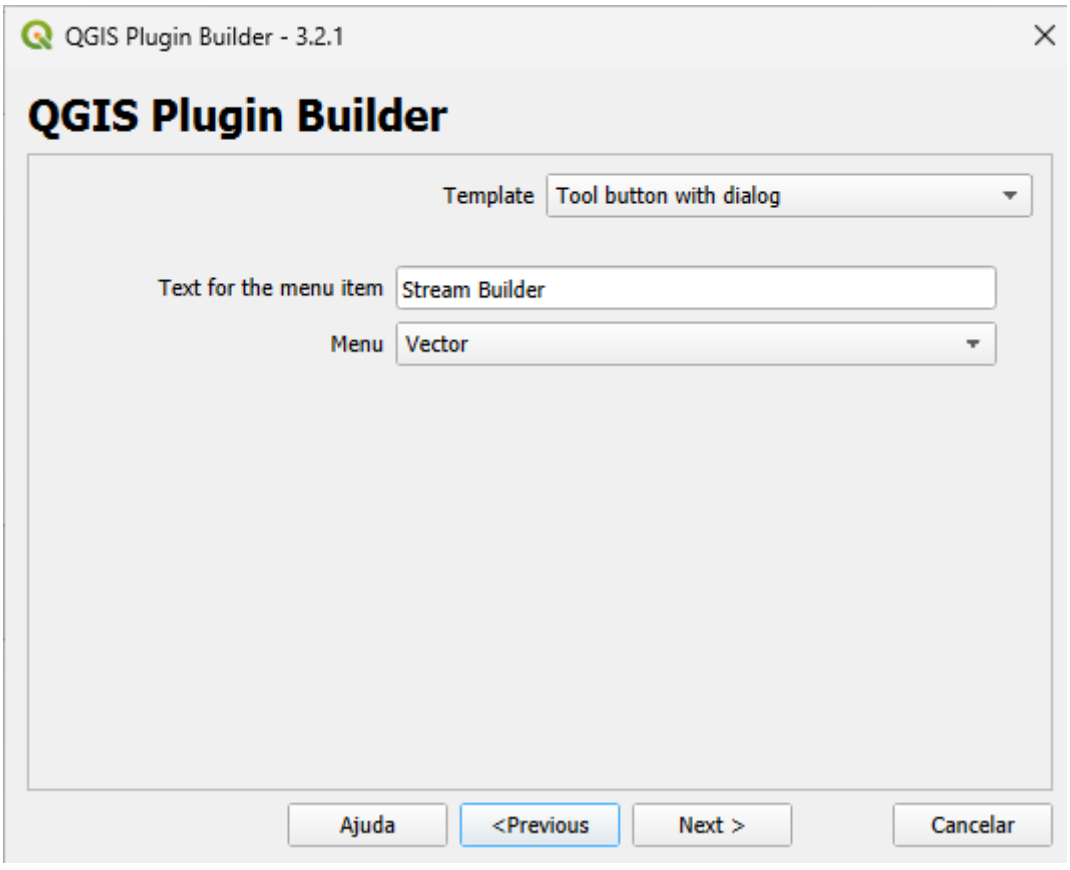
Preencha os campos dos formulários conforme as imagens a seguir:



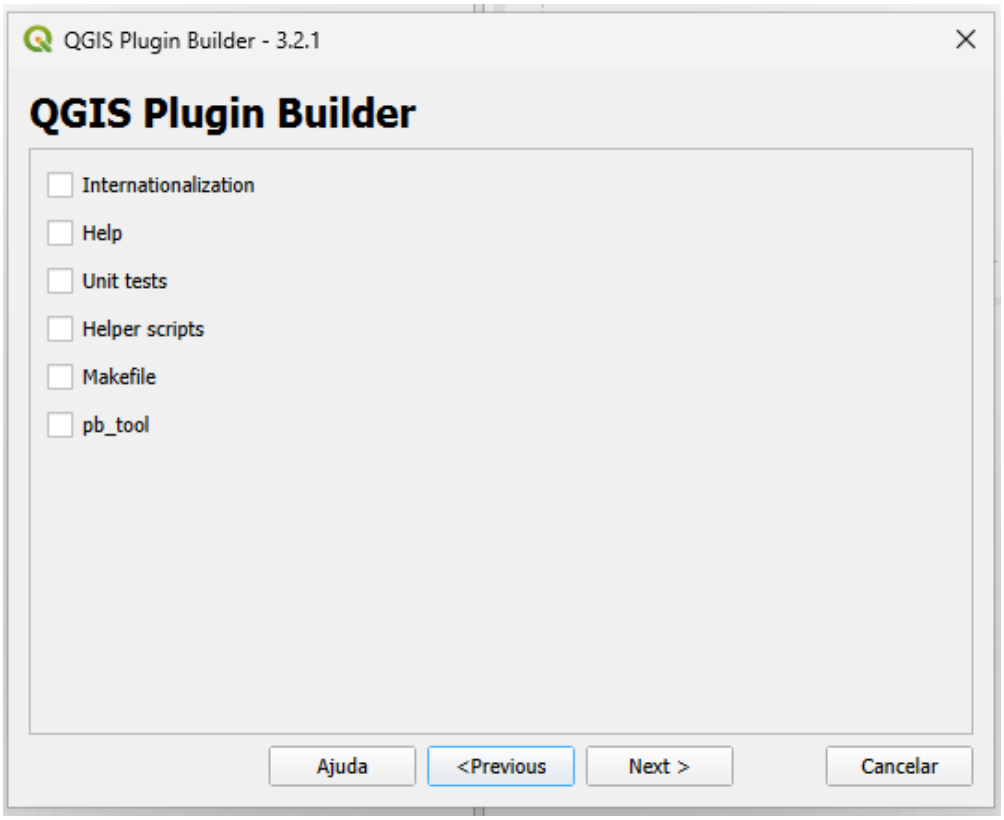
Esse primeiro formulário será usado na criação do arquivo metadata.txt e na definição do nome das classes do plugin.



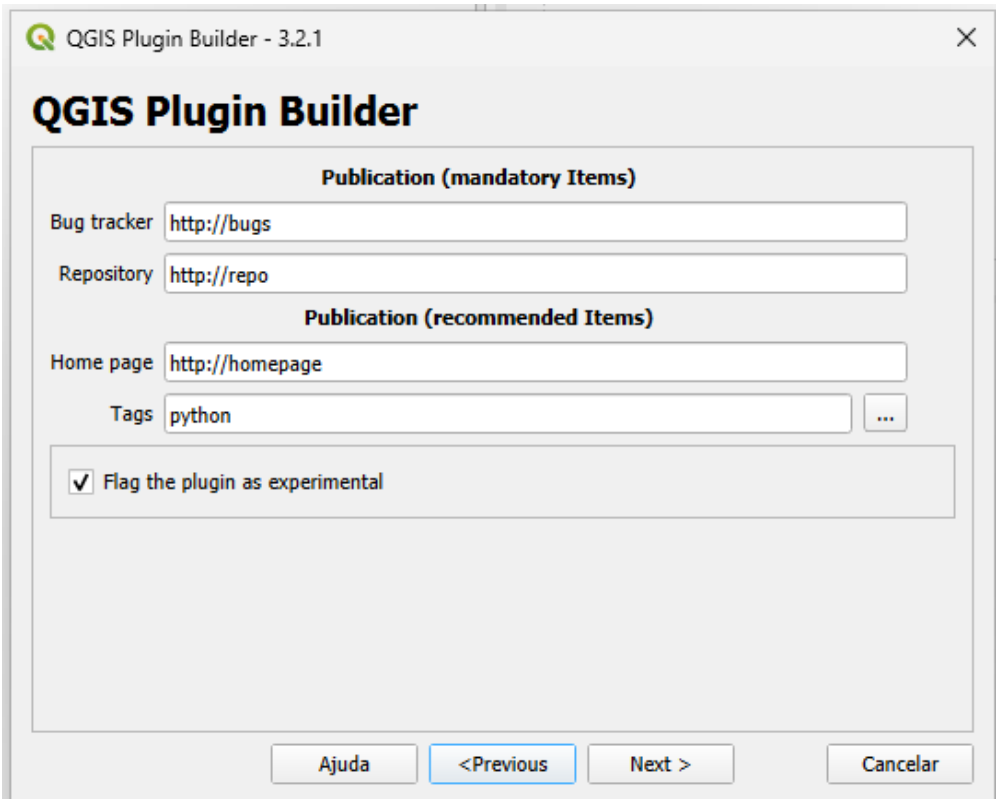
Descrição mais detalhada sobre o plugin que também será colocado no arquivo metadata.txt.



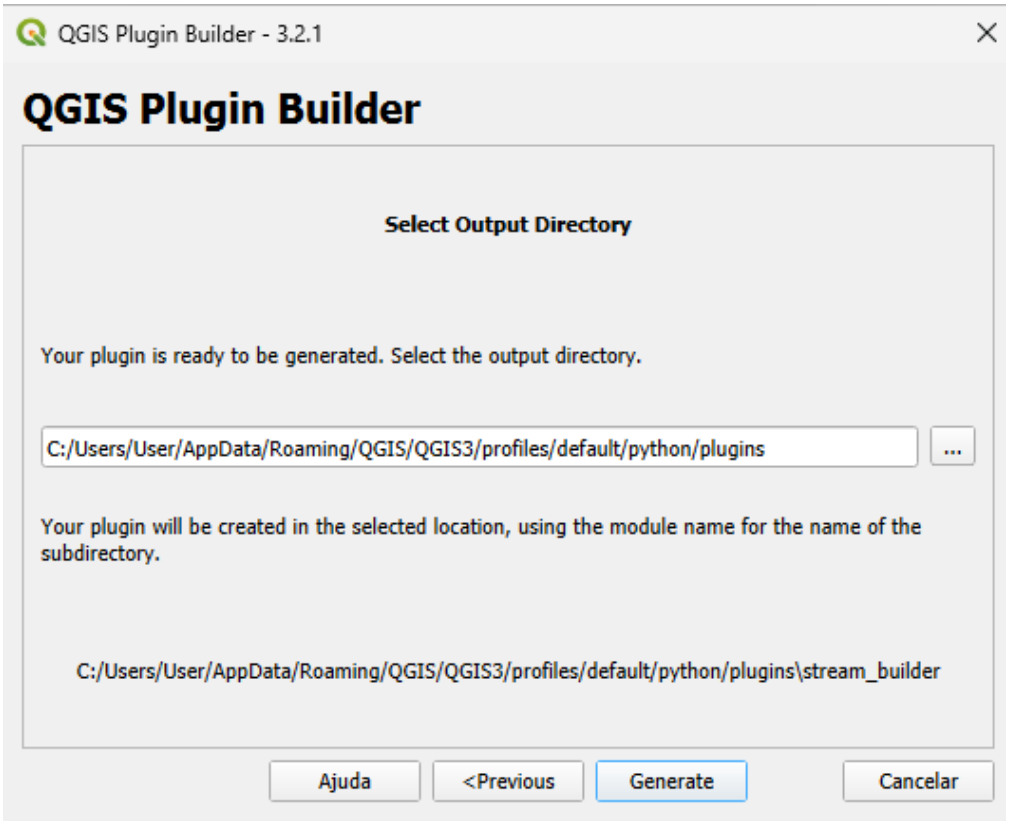
Template (tipo) do plugin, texto que vai aparecer no menu e em qual menu será listado o plugin.



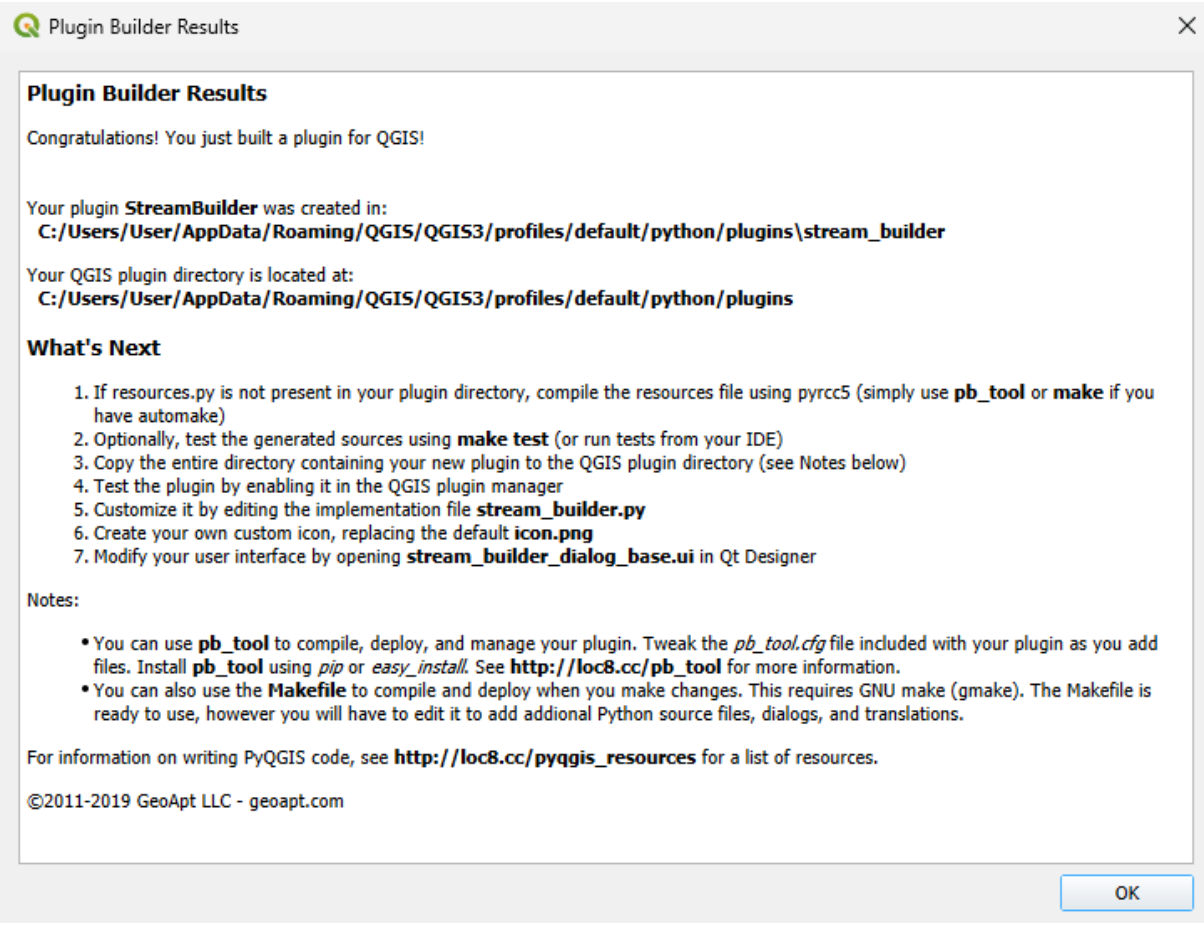
Desmarque todos para esse plugin,



Cheque a caixa de plugin experimental pois não iremos distribuir esse plugin no momento.

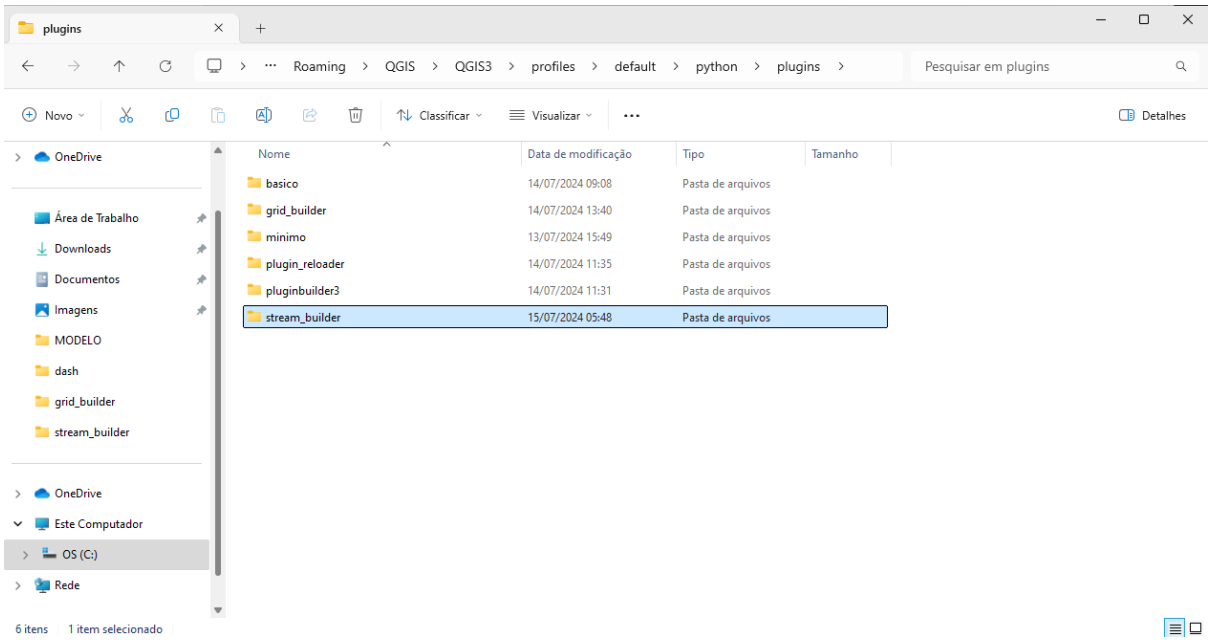


A pasta de plugin do sistema (nesse caso em sistema Windows). Clique **Generate** após selecionar o diretório de plugins.

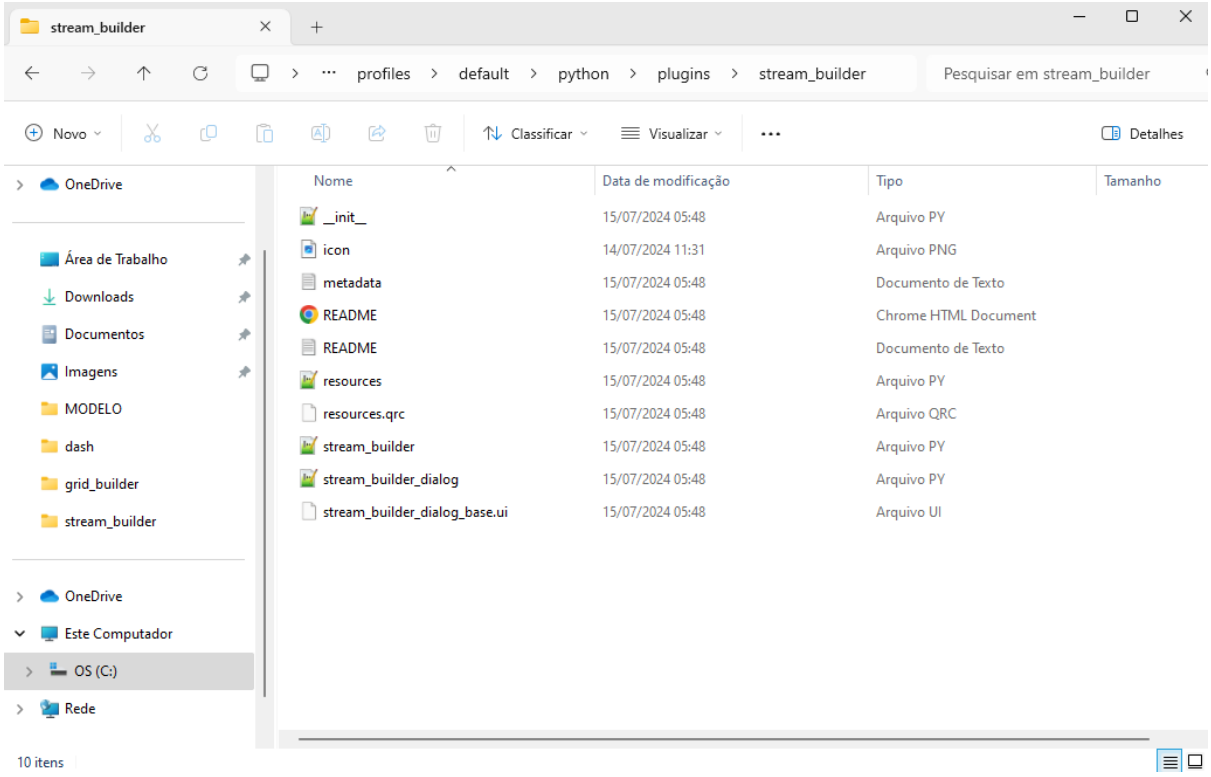


Pronto, os arquivos base de seu plugin foram criados na pasta:

C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins\stream_builder

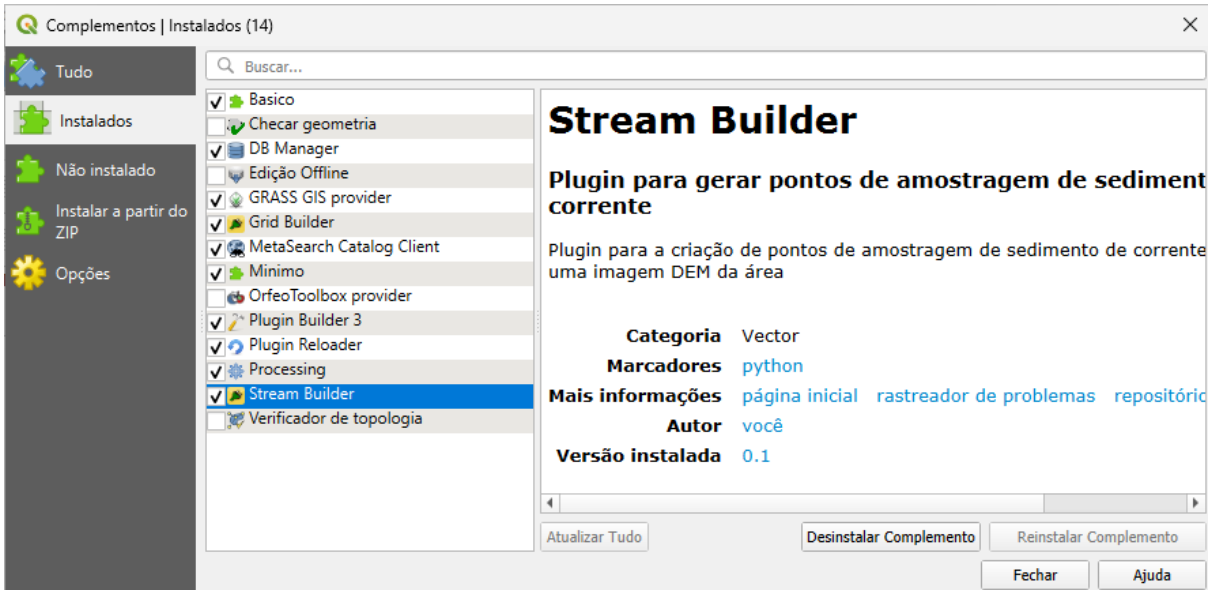


Os arquivos gerados:

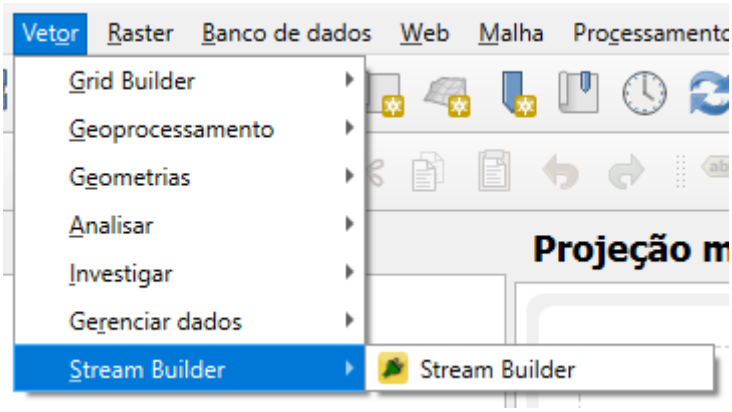


Os oito arquivos necessários mais dois arquivos README com instruções do PluginBuilder foram criados automaticamente.

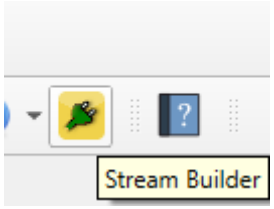
Vamos testar ele iniciando o QGIS e abrindo o **Complementos->Gerenciar e instalar Complementos**. Em Instalados vemos que ele não foi instalado ainda. Marque ele e instale para testarmos.



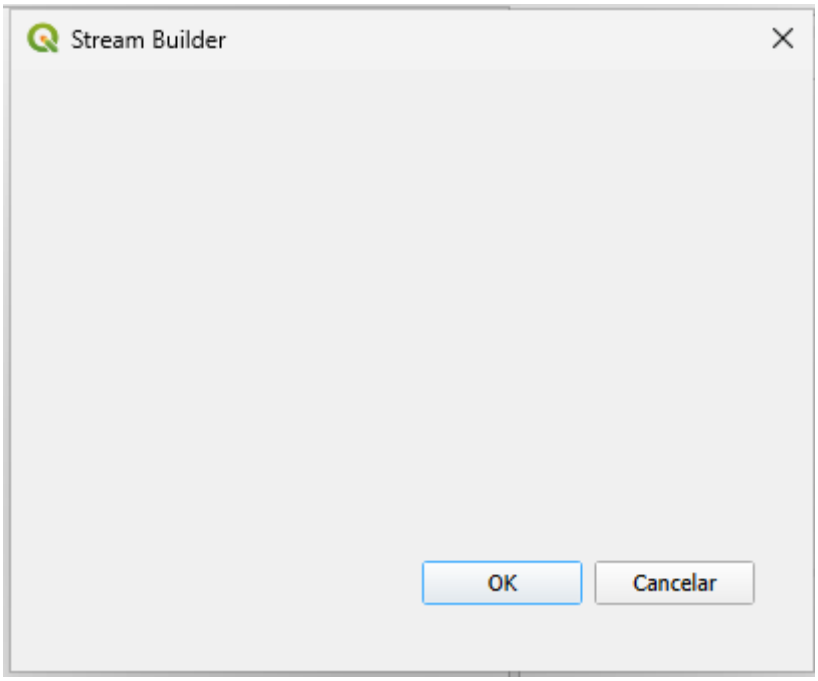
Inicie ele pelo Menu Vetor.



Ou pelo ícone na barra de ferramentas.



Funcionando, mas sem funcionalidade ainda.



Vamos construir a interface gráfica do usuário (GUI) e adicionar a funcionalidade agora na próxima seção.

3 - O plugin Stream_Builder

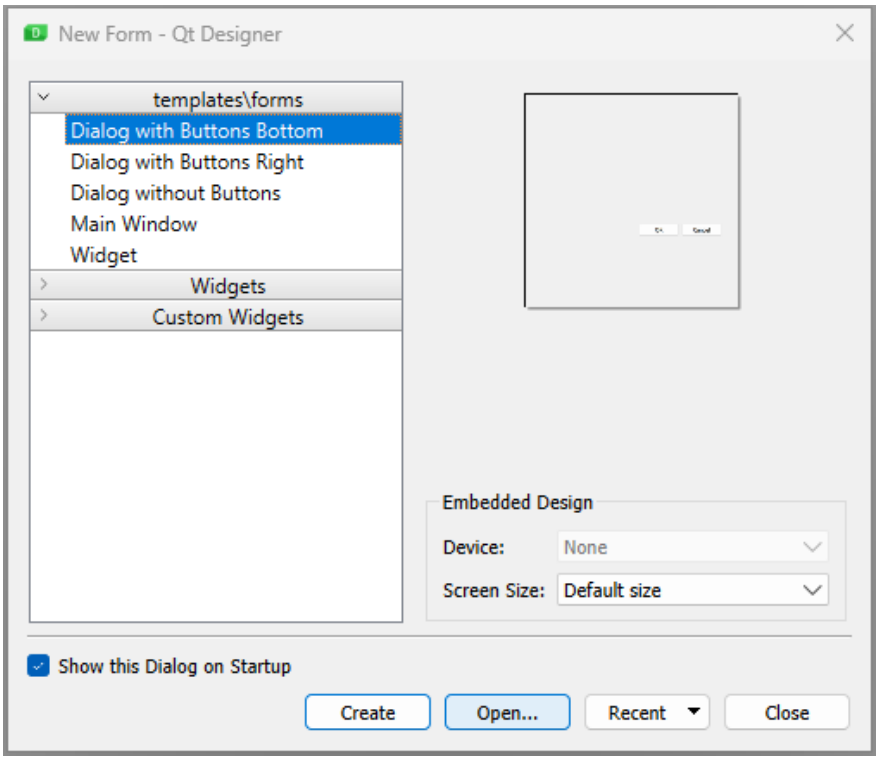
O plugin Stream Builder, executa a criação de pontos de amostragem de sedimento de corrente com base em uma imagem DEM em lat-long da área que deverá ser fornecida.

Abrir QtDesigner para criarmos a nossa interface gráfica de usuário (GUI).

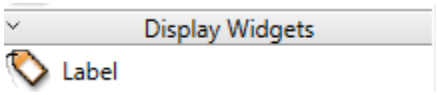
Abra o arquivo **stream_builder_dialog_base.ui** localizado em:

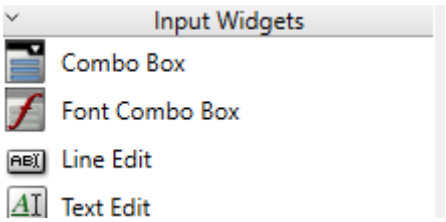
C:\Users\User\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins\stream_builder

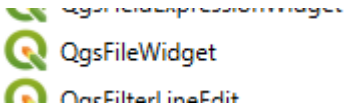
No primeiro diálogo do programa em **Open.**



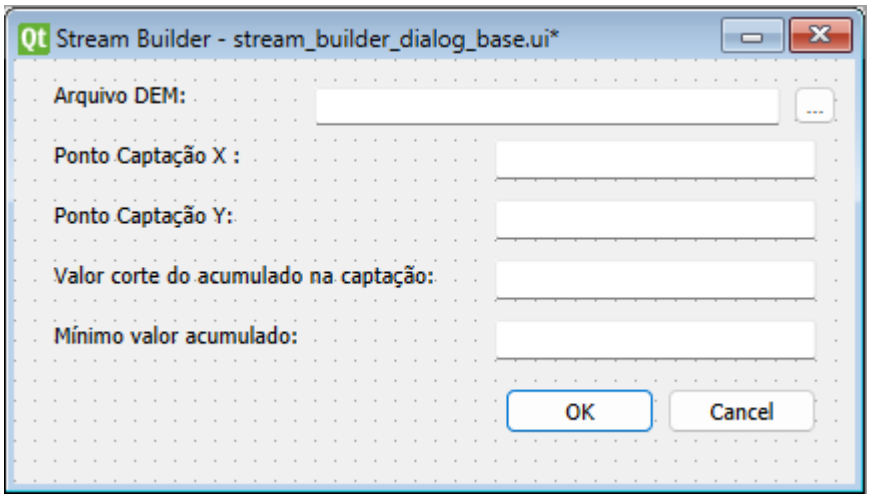
Vamos adicionar 5 widgets do tipo Label, 4 widgets do tipo Line Edit e um widget do tipo QgsFileWidget. Basta clicar no Widget e arrastar até a janela do diálogo.







A aparência final da interface deve ser:



Modifique o texto dos campos Label.

Deixo o campo QgsFileWidget inalterado.

Agora altere a propriedade objectName dos campos LineEdit na seguinte ordem:

lineEditX

lineEditY

lineEditSnap

lineEditAcc

Pronto, salve o diálogo e feche o QtDesigner.

Vamos agora editar o arquivo **stream_builder.py** para realizar a tarefa. Vamos ter de adicionar algumas bibliotecas de suporte via **import** e adicionar o código na função **run** que vai fazer a validação inicial dos campos e a criação da camada de pontos de amostragem de sedimento de corrente.

As bibliotecas serão (adicionar as faltantes):

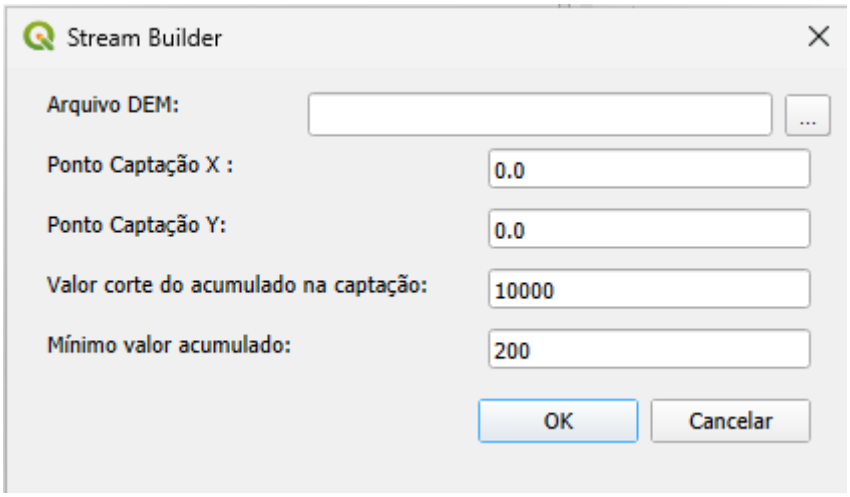
```
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication, QVariant
from qgis.PyQt.QtGui import QIcon, QIntValidator, QDoubleValidator
from qgis.PyQt.QtWidgets import QAction, QMessageBox
from qgis.core import QgsProject,  QgsRasterLayer, QgsMapLayerType, QgsWkbTypes,
QgsVectorLayer, QgsField
from .resources import *
from .stream_builder_dialog import StreamBuilderDialog
import os.path
import numpy as np
from pysheds.grid import Grid
from osgeo import gdal
from shapely import geometry, ops
import fiona
import itertools
import pandas as pd
import geopandas as gpd
import processing
```

A função **run** ficará assim:

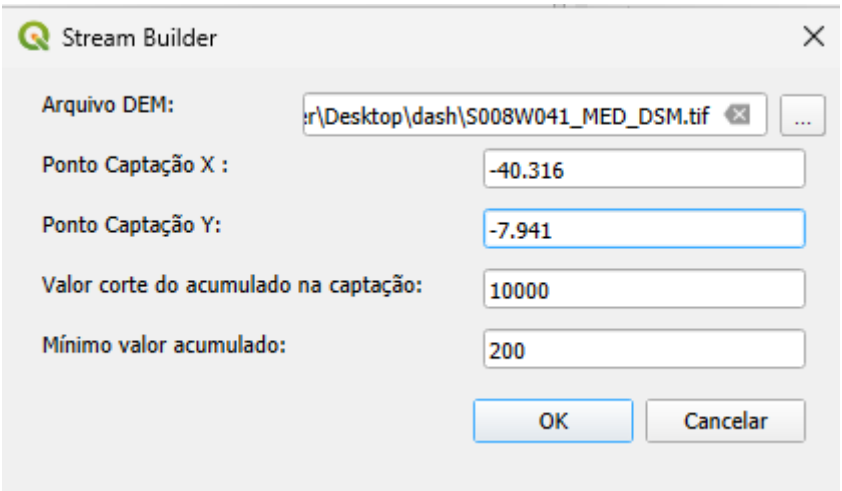
```
    def run(self):
        if self.first_start == True:
            self.first_start = False
            self.dlg = StreamBuilderDialog()
            onlyInt = QIntValidator()
            onlyDouble = QDoubleValidator()
            self.dlg.lineEditX.setText('0.0')
            self.dlg.lineEditX.setValidator(onlyDouble)
            self.dlg.lineEditY.setText('0.0')
            self.dlg.lineEditY.setValidator(onlyDouble)
            self.dlg.lineEditSnap.setText('10000')
            self.dlg.lineEditSnap.setValidator(onlyInt)
            self.dlg.lineEditAcc.setText('200')
            self.dlg.lineEditAcc.setValidator(onlyInt)

        self.dlg.show()
        result = self.dlg.exec_()
        # See if OK was pressed
        if result:
            if self.dlg.lineEditY.text()=='' or self.dlg.lineEditX.text()=='' or
self.dlg.lineEditSnap.text()=='' or self.dlg.lineEditAcc.text()=='':
                QMessageBox.warning(self.iface.mainWindow(),
                        'Erro',
                        "Entrar todos os campo por favor \nSaindo...")
                return
            layer = self.dlg.mQgsFileWidget.filePath()
            y = float(self.dlg.lineEditY.text())
            x = float(self.dlg.lineEditX.text())
            snapp= float(self.dlg.lineEditSnap.text())
            accc= float(self.dlg.lineEditAcc.text())
            grid = Grid.from_raster(layer)
            dem = grid.read_raster(layer)
            pit_filled_dem = grid.fill_pits(dem)
            flooded_dem = grid.fill_depressions(pit_filled_dem)
            inflated_dem = grid.resolve_flats(flooded_dem)
            dirmap = (64, 128, 1, 2, 4, 8, 16, 32)
            fdir = grid.flowdir(inflated_dem, dirmap=dirmap)
            acc = grid.accumulation(fdir, dirmap=dirmap)
            x_snap, y_snap = grid.snap_to_mask(acc > snapp, (x,y))
            catch = grid.catchment(x=x_snap, y=y_snap, fdir=fdir,
dirmap=dirmap,xytype='coordinate')
            grid.clip_to(catch)
            clipped_catch = grid.view(catch)
            branches = grid.extract_river_network(fdir, acc > accc, dirmap=dirmap)
            schema = {
                'geometry': 'LineString',
                'properties': {}
            }

            with fiona.open('rivers.shp', 'w',
                driver='ESRI Shapefile',
                crs=grid.crs.srs,
                schema=schema) as c:
                i = 0
                for branch in branches['features']:
                    rec = {}
                    rec['geometry'] = branch['geometry']
                    rec['properties'] = {}
                    rec['id'] = str(i)
                    c.write(rec)
                    i += 1
            layeriv = QgsVectorLayer('rivers.shp', 'Drenagem', "ogr")
            QgsProject.instance().addMapLayer(layeriv)
            df = gpd.read_file('rivers.shp')
            s = pd.Series(df.geometry.values, index=df.index).to_dict()
            intersections = []
            for i in itertools.combinations(s, 2):
                i1, i2 = i
                if s[i1].intersects(s[i2]): #If they intersect
                    intersections.append([s[i1].intersection(s[i2])])

            dfinter = gpd.GeoDataFrame(pd.DataFrame(data=intersections, columns=['geometry']),
geometry='geometry', crs="EPSG:4326")
            dfinter.to_file('river_intersections.shp')
```

```
ptsau0 = QgsVectorLayer('river_intersections.shp', 'Pontos', "ogr")
parameters = {'INPUT': ptsau0,'OUTPUT': "temp.shp"}
processing.run('native:deleteduplicategeometries', parameters)
ptsau = QgsVectorLayer('temp.shp', 'Pontos', "ogr")
ptsau.startEditing()
id_idx = ptsau.fields().lookupField('FID')
ptsau.renameAttribute(id_idx, 'id')
ptsau.commitChanges()
QgsProject.instance().addMapLayer(ptsau)
QMessageBox.information(self.iface.mainWindow(),
             'Pronto',
             "Pontos de amostragem para sedimento de corrente criados!")
return
```

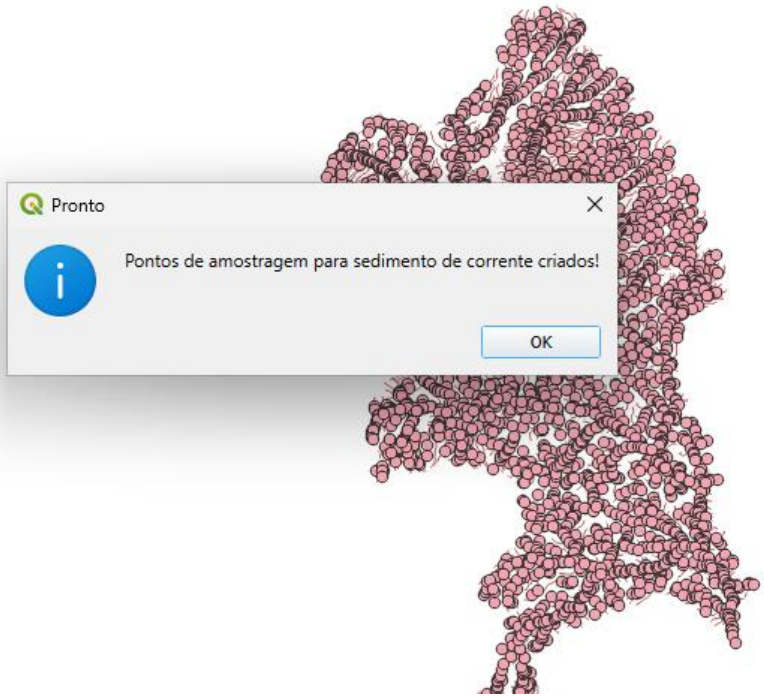
Antes de comentarmos o código adicionado vamos testar o plugin.
Baixe o arquivo DEM em https://gdatasystems.com/pyqgis/index.php **NOTA: O DEM deve estar em Latiyude-Longitude.**
Abra o QGIS e o plugin será carregado já com as alterações feitas. Ao iniciarmos o plugin teremos:



Execute o plugin com os seguintes parâmetros. A posição X e U da capitação deve ser sobre uma drenagem e a montante desse ponto será feita a análise:

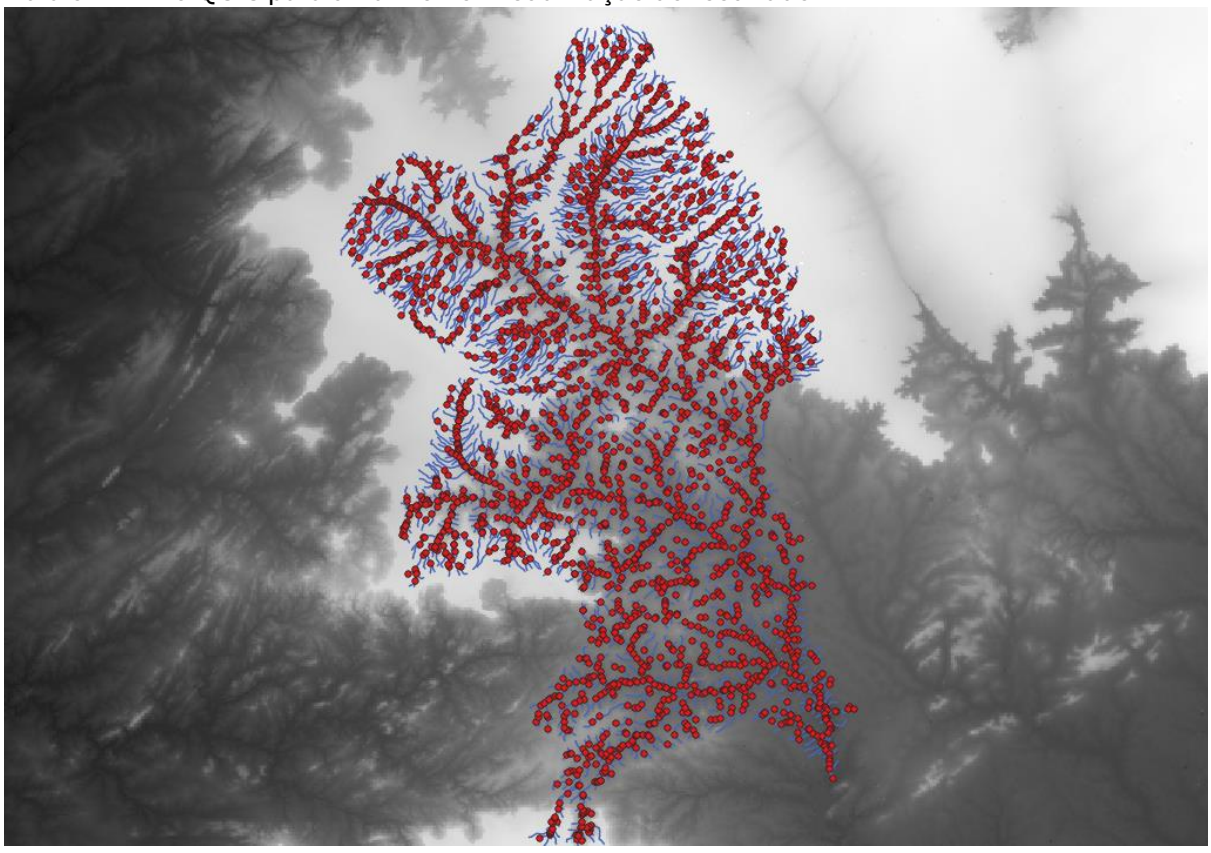


O processamento demora uns dois minutos e ao final teremos a mensagem:

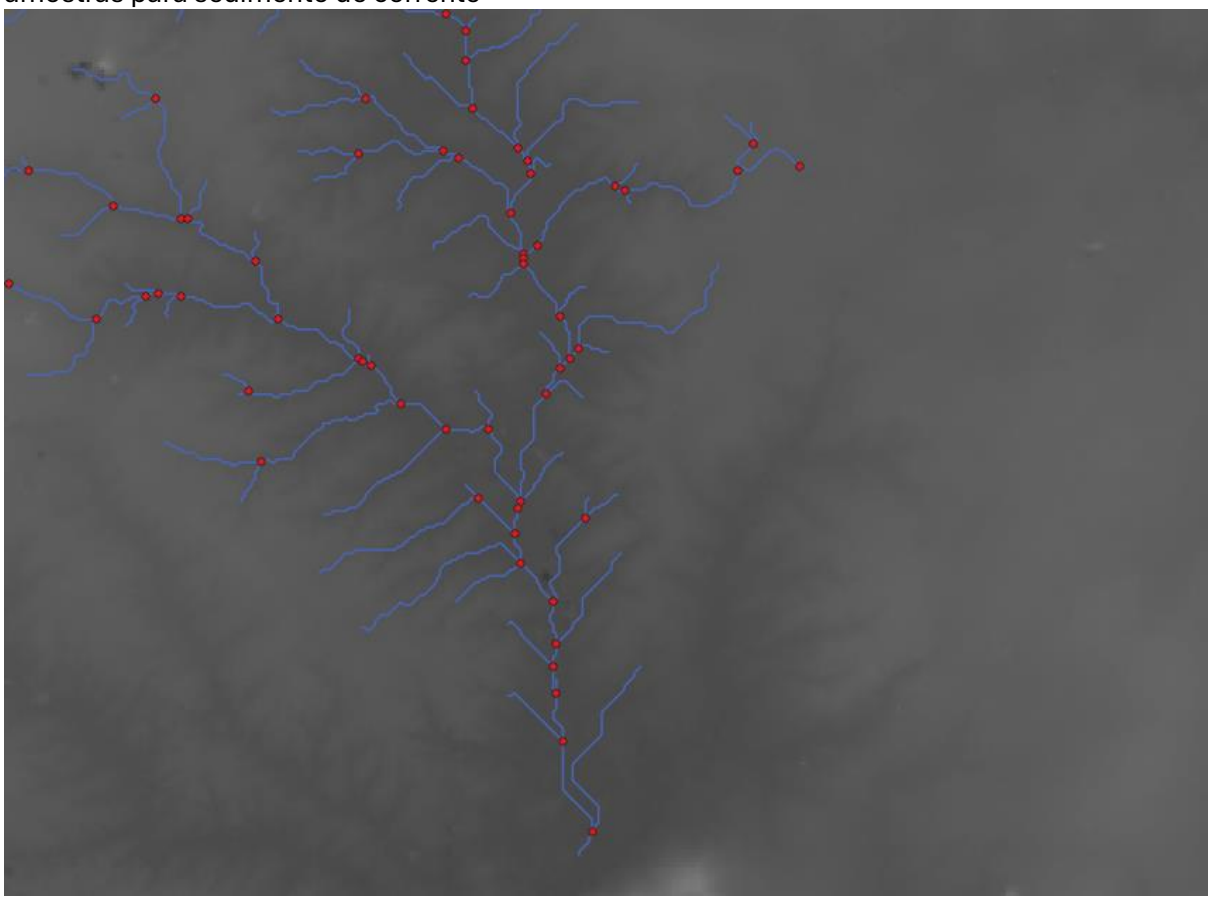


Agora, caso precise, é só salvar as camadas criadas na projeção desejada (drenagens e pontos de amostragem nas bifurcações).

Abra o DEM no QGIS para uma melhor visualização do resultado.



Zoom mostrando a drenagem extraída e os pontos de bifurcações para auxiliar na coleta de amostras para sedimento de corrente



Este bloco do código inicializa os widgets adicionando os valores iniciais nas linhas editáveis.

```
        if self.first_start == True:
            self.first_start = False
            self.dlg = StreamBuilderDialog()
            onlyInt = QIntValidator()
            onlyDouble = QDoubleValidator()
            self.dlg.lineEditX.setText('0.0')
            self.dlg.lineEditX.setValidator(onlyDouble)
            self.dlg.lineEditY.setText('0.0')
            self.dlg.lineEditY.setValidator(onlyDouble)
            self.dlg.lineEditSnap.setText('10000')
            self.dlg.lineEditSnap.setValidator(onlyInt)
            self.dlg.lineEditAcc.setText('200')
            self.dlg.lineEditAcc.setValidator(onlyInt)

        self.dlg.show()
        result = self.dlg.exec_()
```

O bloco seguinte checa se os campos estão todos preenchidos, assinala eles às variáveis do programa e lê o arquivo DEM.

```
            if self.dlg.lineEditY.text()=='' or self.dlg.lineEditX.text()=='' or
self.dlg.lineEditSnap.text()=='' or self.dlg.lineEditAcc.text()=='':
                QMessageBox.warning(self.iface.mainWindow(),
                         'Erro',
                         "Entrar todos os campo por favor \nSaindo...")
                return
            layer = self.dlg.mQgsFileWidget.filePath()
            y = float(self.dlg.lineEditY.text())
            x = float(self.dlg.lineEditX.text())
```

```
            snapp= float(self.dlg.lineEditSnap.text())
            accc= float(self.dlg.lineEditAcc.text())
            grid = Grid.from_raster(layer)
            dem = grid.read_raster(layer)
```

Aqui extraímos a drenagem do dem e criamos a camada **Drenagem**.

```
            pit_filled_dem = grid.fill_pits(dem)
            flooded_dem = grid.fill_depressions(pit_filled_dem)
            inflated_dem = grid.resolve_flats(flooded_dem)
            dirmap = (64, 128, 1, 2, 4, 8, 16, 32)
            fdir = grid.flowdir(inflated_dem, dirmap=dirmap)
            acc = grid.accumulation(fdir, dirmap=dirmap)
            x_snap, y_snap = grid.snap_to_mask(acc > snapp, (x,y))
            catch = grid.catchment(x=x_snap, y=y_snap, fdir=fdir,
dirmap=dirmap,xytype='coordinate')
            grid.clip_to(catch)
            clipped_catch = grid.view(catch)
            branches = grid.extract_river_network(fdir, acc > accc, dirmap=dirmap)
            schema = {
                'geometry': 'LineString',
                'properties': {}
            }
```

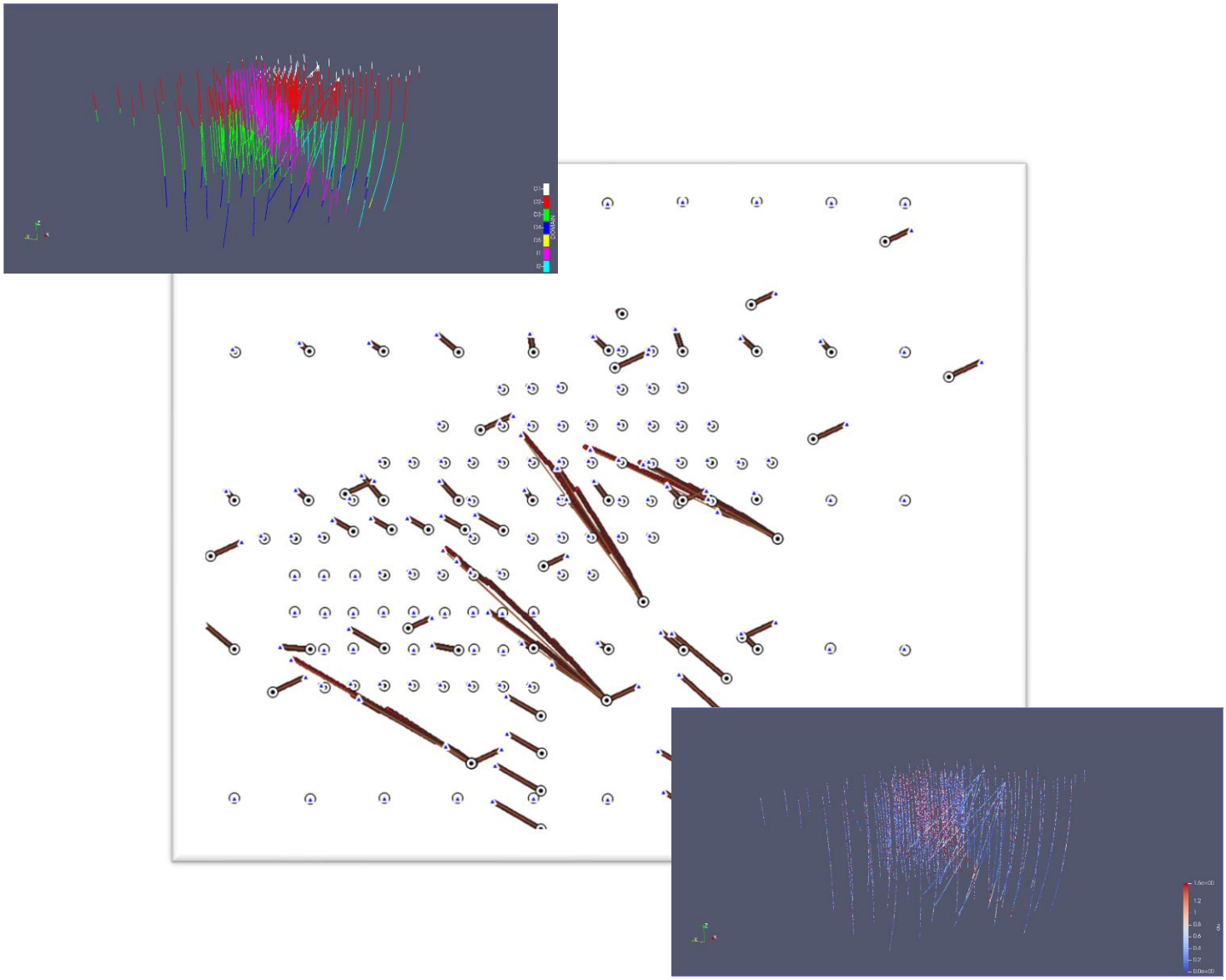
Finalmente criamos os pontos de amostragem nas bifurcações e criamos a camada de pontos **Pontos**.

```
    with fiona.open('rivers.shp', 'w',
                driver='ESRI Shapefile',
                crs=grid.crs.srs,
                schema=schema) as c:
                i = 0
                for branch in branches['features']:
                    rec = {}
                    rec['geometry'] = branch['geometry']
                    rec['properties'] = {}
                    rec['id'] = str(i)
                    c.write(rec)
                    i += 1
            layeriv = QgsVectorLayer('rivers.shp', 'Drenagem', "ogr")
            QgsProject.instance().addMapLayer(layeriv)
            df = gpd.read_file('rivers.shp')
            s = pd.Series(df.geometry.values, index=df.index).to_dict()
            intersections = []
            for i in itertools.combinations(s, 2):
                i1, i2 = i
                if s[i1].intersects(s[i2]): #If they intersect
                    intersections.append([s[i1].intersection(s[i2])])

            dfinter = gpd.GeoDataFrame(pd.DataFrame(data=intersections, columns=['geometry']),
geometry='geometry', crs="EPSG:4326")
            dfinter.to_file('river_intersections.shp')
            ptsau0 = QgsVectorLayer('river_intersections.shp', 'Pontos', "ogr")
            parameters = {'INPUT': ptsau0,'OUTPUT': "temp.shp"}
            processing.run('native:deleteduplicategeometries', parameters)
            ptsau = QgsVectorLayer('temp.shp', 'Pontos', "ogr")
            ptsau.startEditing()
            id_idx = ptsau.fields().lookupField('FID')
            ptsau.renameAttribute(id_idx, 'id')
            ptsau.commitChanges()
            QgsProject.instance().addMapLayer(ptsau)
            QMessageBox.information(self.iface.mainWindow(),
                        'Pronto',
                        "Pontos de amostragem para sedimento de corrente criados!")
                    return
```

No próximo vamos criar um plugin que efetua o desurvey de dados de furo de sondagem.

Referência do pySheds

```
@misc{bartos_2020,
  title = {pysheds: simple and fast watershed delineation in python},
  author = {Bartos, Matt},
  url = {https://github.com/mdbartos/pysheds},
  year = {2020},
  doi = {10.5281/zenodo.3822494}
}
```



Criando Plugins QGIS com pyQGIS

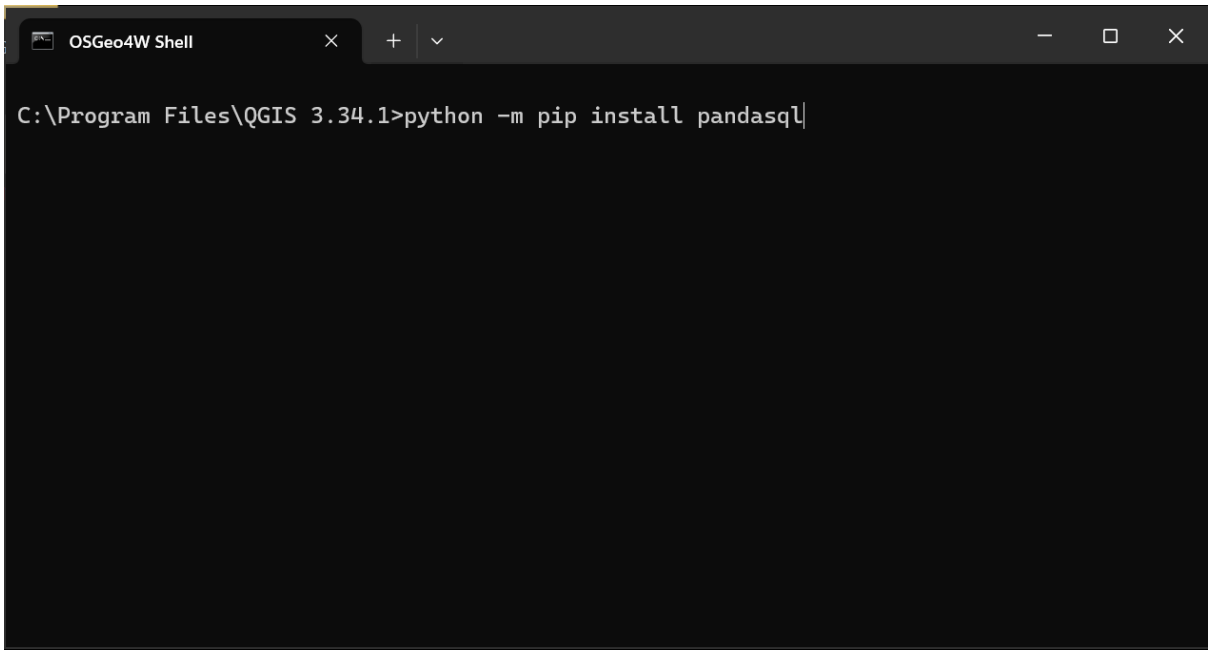
Plugin Drillhole1

1 - O ambiente pyQGIS

O QGIS possui um ambiente python dedicado e para instalar novas bibliotecas usamos o shell OSGeo4W. Nesse exemplo de plugin vamos utilizar as biblioteca **pandasql**.

Inicie o shell e digite:

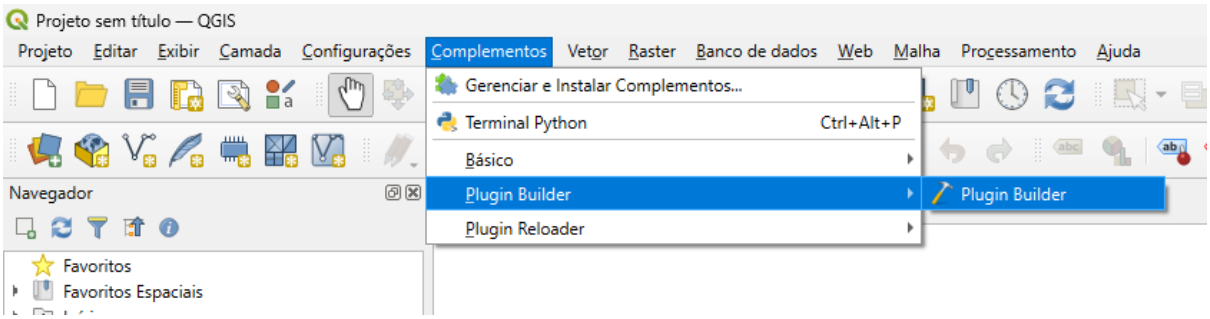
```
python -m pip install pandasql
```



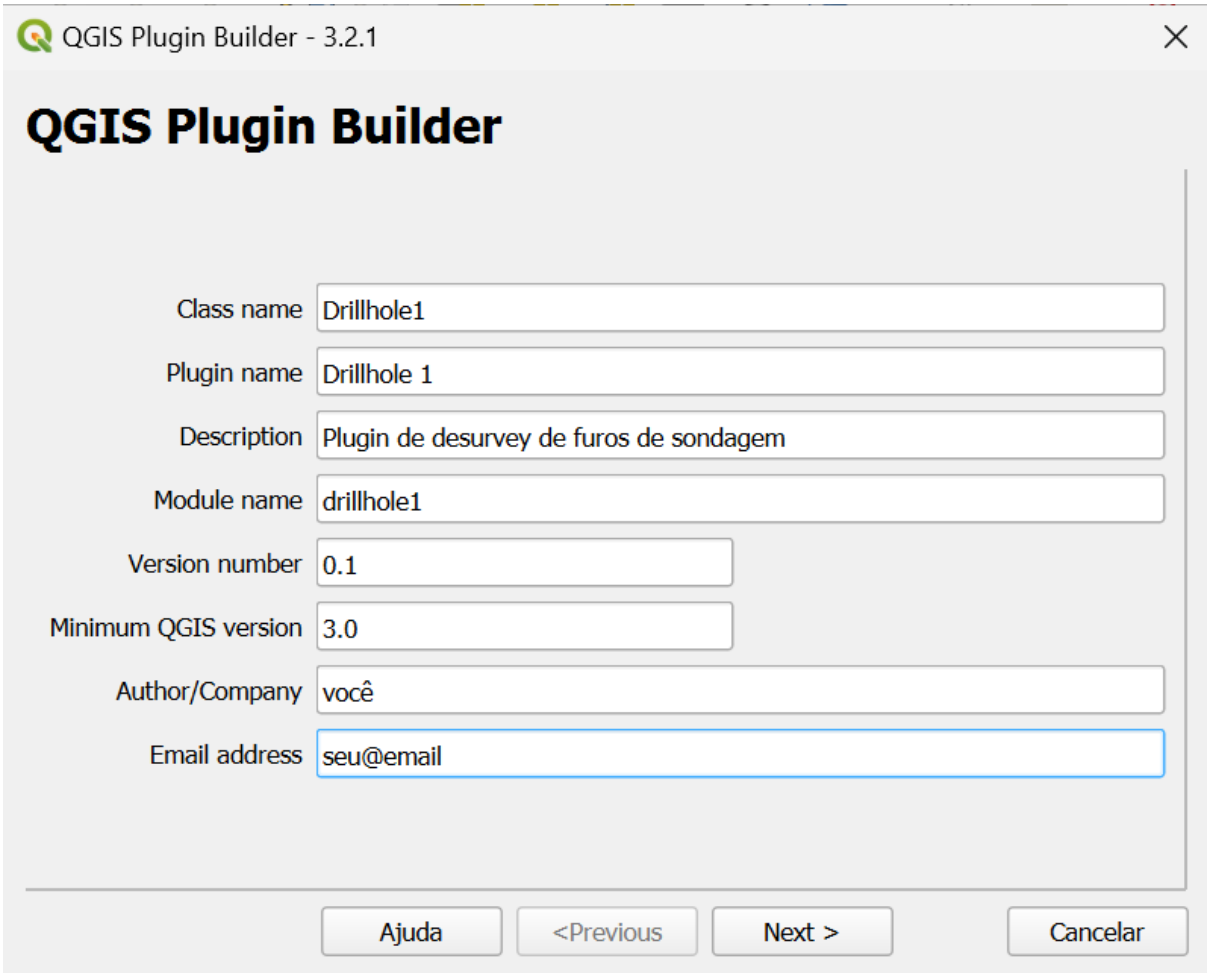
A biblioteca adicional necessária para nosso plugin foi instalada. Procederemos agora com a construção do plugin novo.

2 - Construindo o esqueleto do Drillhole1 no Plugin Builder 3

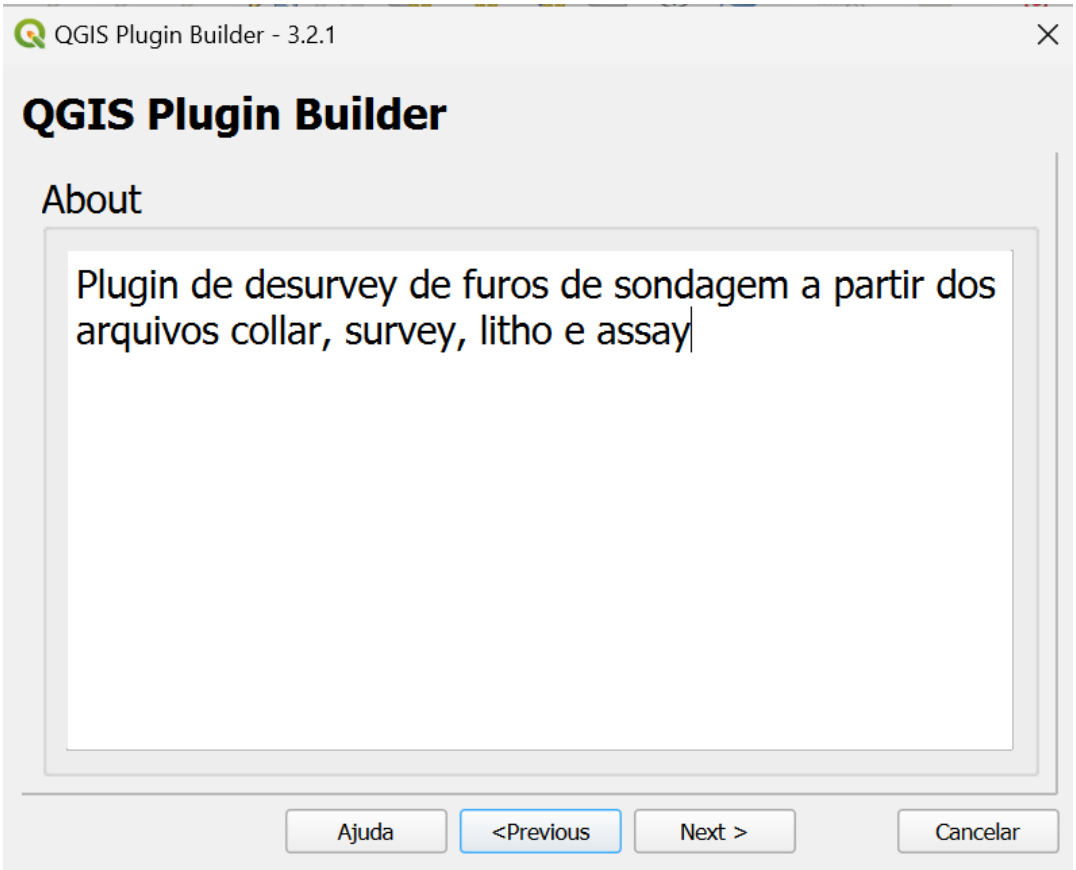
Inicie o Plugin Builder:



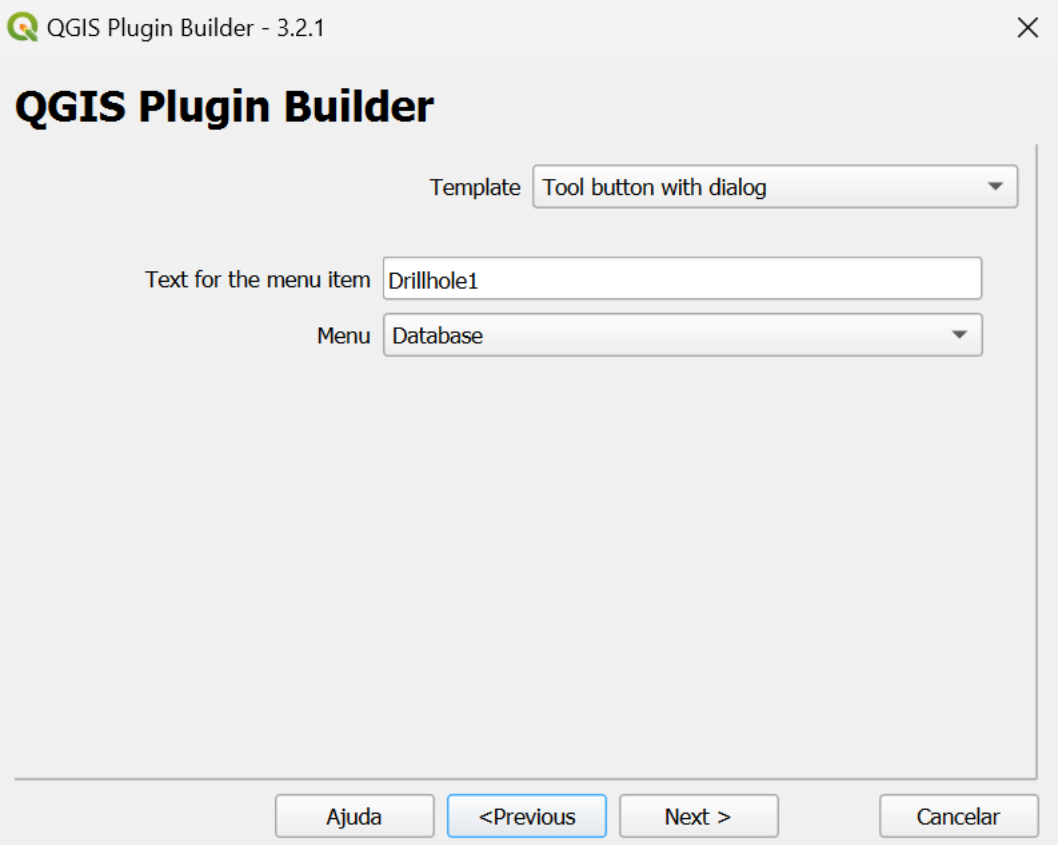
Preencha os campos dos formulários conforme as imagens a seguir:



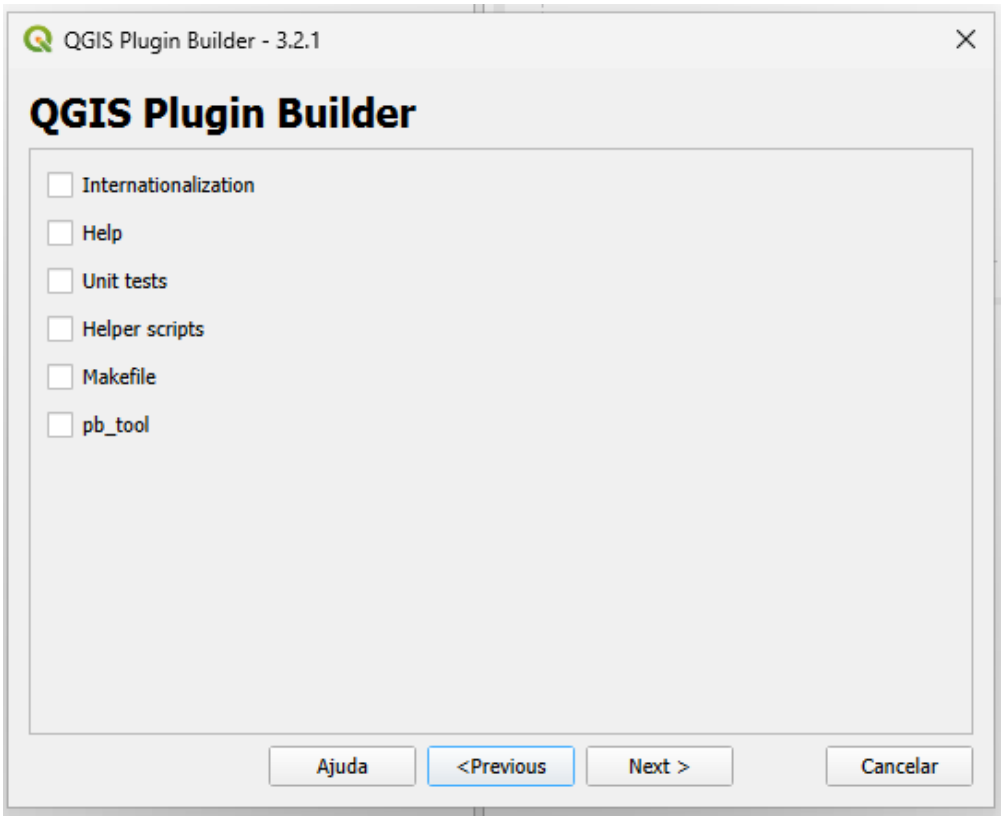
Esse primeiro formulário será usado na criação do arquivo metadata.txt e na definição do nome das classes do plugin.



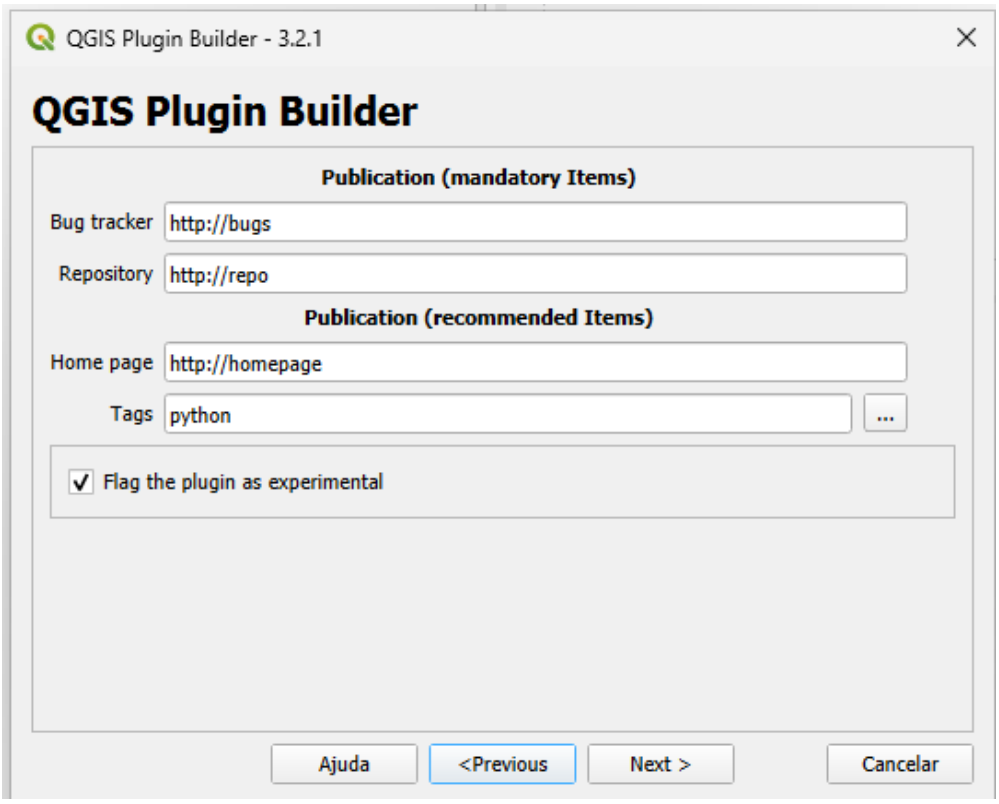
Descrição mais detalhada sobre o plugin que também será colocado no arquivo metadata.txt.



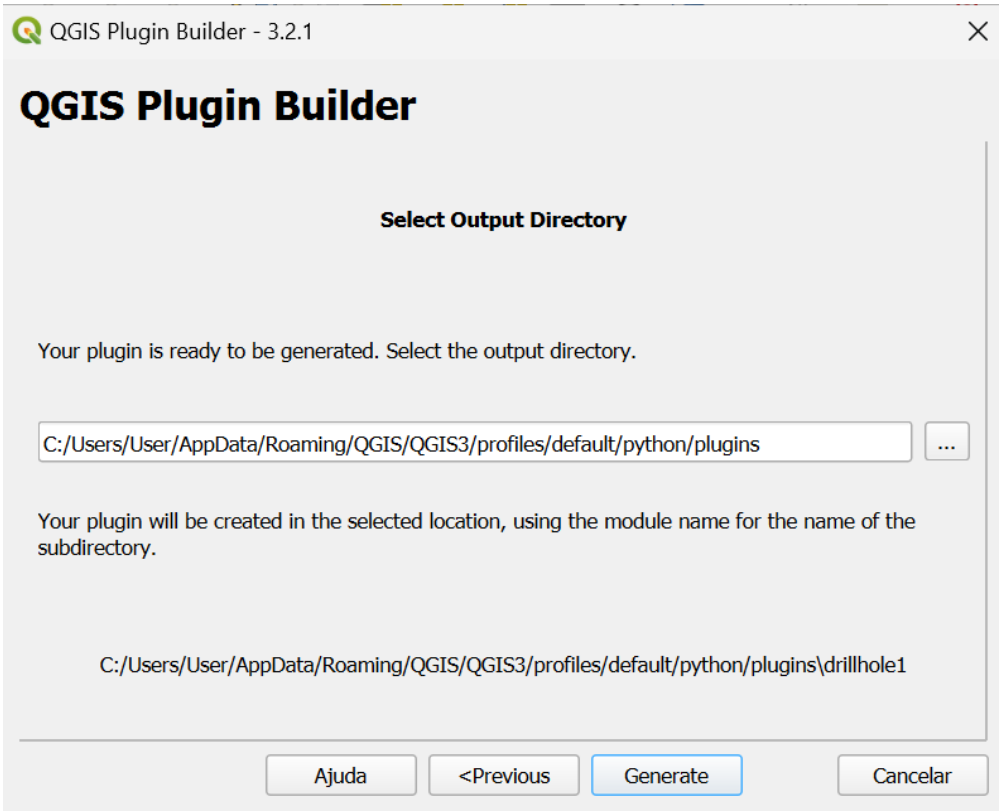
Template (tipo) do plugin, texto que vai aparecer no menu e em qual menu será listado o plugin.



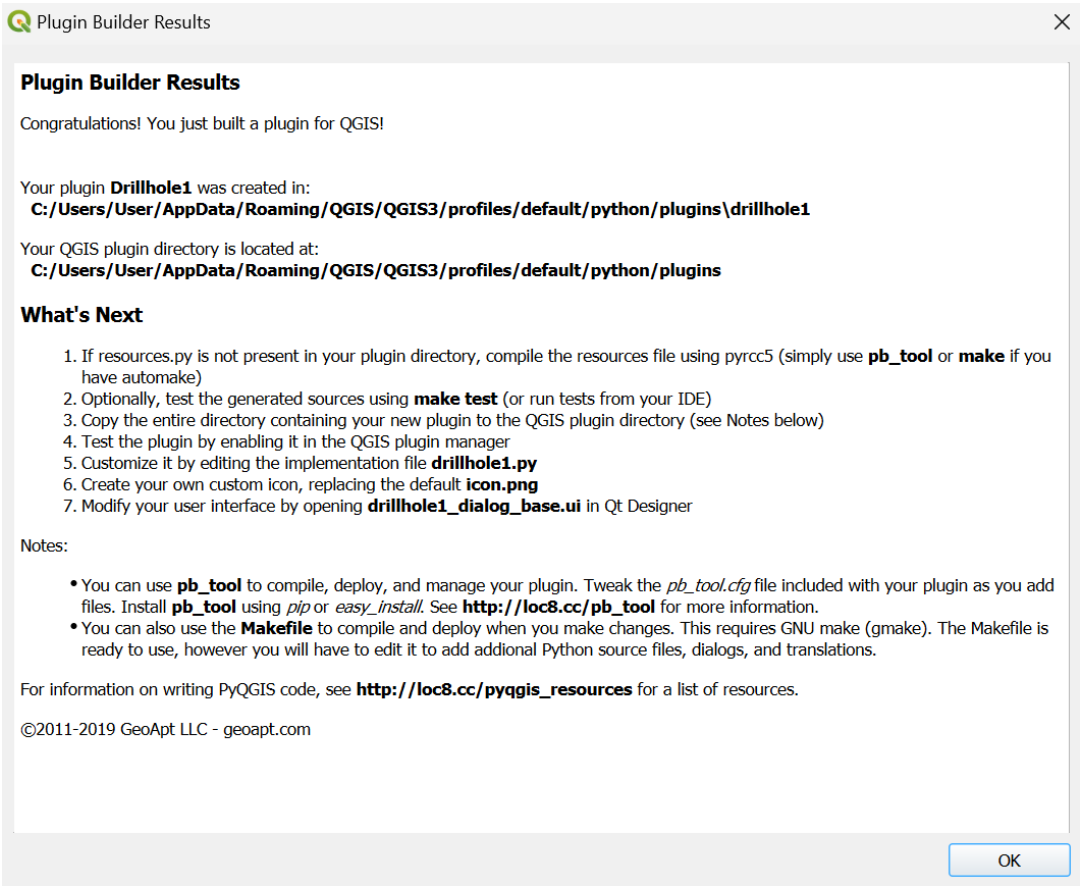
Desmarque todos para esse plugin,



Cheque a caixa de plugin experimental pois não iremos distribuir esse plugin no momento.



A pasta de plugin do sistema (nesse caso em sistema Windows). Clique **Generate** após selecionar o diretório de plugins.

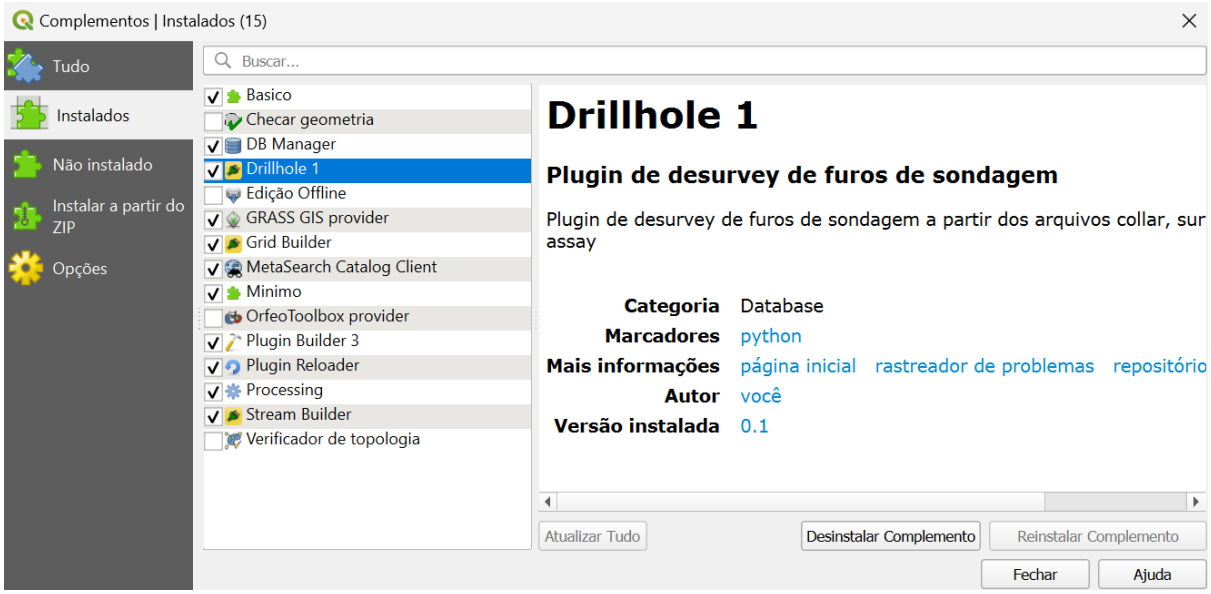


Pronto, os arquivos base de seu plugin foram criados na pasta:

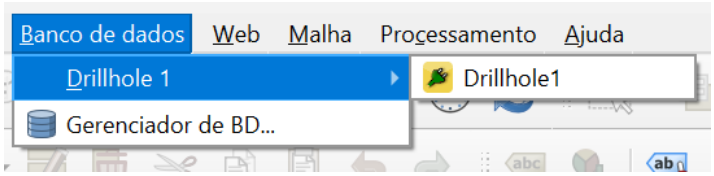
C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins\drillhole1

Os oito arquivos necessários mais dois arquivos README com instruções do PluginBuilder foram criados automaticamente.

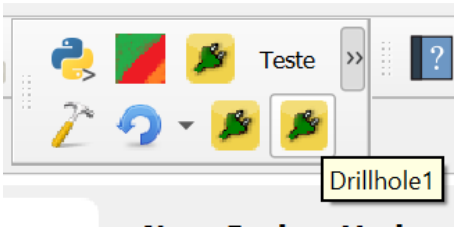
Vamos testar ele iniciando o QGIS e abrindo o **Complementos->Gerenciar e instalar Complementos**. Em Instalados vemos que ele não foi instalado ainda. Marque ele e instale para testarmos.



Inicie ele pelo Menu Vetor.



Ou pelo ícone na barra de ferramentas.



Funcionando, mas sem funcionalidade ainda.

Vamos construir a interface gráfica do usuário (GUI) e adicionar a funcionalidade agora na próxima seção.

3 - O plugin Drillhole1

O plugin Drillhole1 tem como objetivo abrir 4 arquivos de dados de furos de sondagem de exploração mineral típicos:

118olar.csv contendo informações de coordenadas da boca do furo em XYZ e profundidade atingida.
- Formato: HOLEID,X,Y,Z,ENDDEPTH

survey.csv contendo informações de direção e mergulho do furo e dos desvios entre boca do furo (collar) e ao longo do mesmo.
- Formato: HOLEID,AT,AZM,DIP

litho.csv contendo informações dos tipos geológicos, domínios e alteração e como estes variam dentro do furo.
- Formato: HOLEID,FROM,TO,DOMAIN,ROCKTYPE,WEATH

survey.csv contendo a análise de teor de um único elemento (au).
- Formato HOLEID,FROM,TO,AU

Os arquivos estão disponíveis em https://gdatasystems.com/pyqgis/index.php

Este plugin funciona somente com o formato descrito acima, em um próximo módulo faremos um plugin que nos permitirá formatar livremente os campos com base nos arquivos originais.

Agora execute o QtDesigner para criarmos a nossa interface gráfica de usuário (GUI).

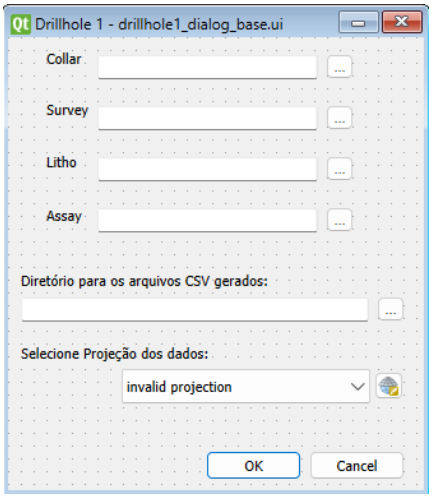
Abra o arquivo **drillhole1_dialog_base.ui** localizado em:

C:\Users\User\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins\drillhole1

No primeiro diálogo do programa em **Open.**

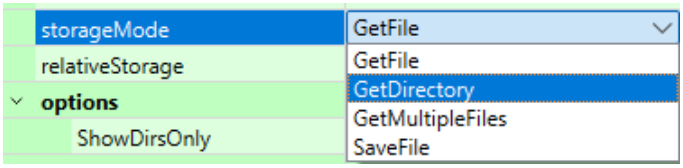
Vamos adicionar 5 widgets do tipo Label, 5 widgets do tipo QgsFileWidget e 1 QgsProjectionSelectionWidget. Basta clicar no Widget e arrastar até a janela do diálogo.

A aparência final da interface deve ser:



Modifique o texto dos campos Label.

No último QgsFileWidget altere o componente **StorageMode** para **GetDirectory**:



Agora altere a propriedade objectName dos campos QgsFileWidget na seguinte ordem.

mQgsFileWidgetCollar

mQgsFileWidgetSurvey

mQgsFileWidgetLitho

mQgsFileWidgetAssay

mQgsFileWidgetDir

Pronto, salve o diálogo e feche o QtDesigner.

Vamos agora editar o arquivo **drillhole1.py** para realizar a tarefa. Vamos ter de adicionar algumas bibliotecas de suporte via **import** e adicionar o código na função **run** que vai fazer a validação inicial dos campos e a criação da camada de pontos de amostragem de sedimento de corrente. Adicionaremos um arquivo **desurvey.py** com as funções separadamente.

As bibliotecas no arquivo **drillhole1.py** serão (adicionar as faltantes):

```
from qgis.PyQt.QtCore import Qsettings, Qtranslator, QcoreApplication,Qvariant
from qgis.PyQt.QtGui import Qicon, Qcolor
from qgis.PyQt.QtWidgets import Qaction, QmessageBox
from qgis.core import QgsProject, QgsVectorLayer,QgsFeature, QgsField, QgsGeometry
from qgis.core import QgsPoint,QgsVectorFileWriter,QgsMarkerSymbol,QgsStyle
# Initialize Qt resources from file resources.py
from .resources import *
# Import the code for the dialog
from .drillhole1_dialog import Drillhole1Dialog
import os.path
# Initialize Qt resources from file resources.py
from .desurvey import *
import pandas as pd
from pandasql import sqldf
```

Crie na mesma pasta do plugin o seguinte arquivo com as funções auxiliares **desurvey.py** :

```
import pandas as pd
import numpy as np
#-----------------------------------------------------------------------------
def tresD(co,es):
   linha = pd.DataFrame(columns = ['hole', 'prof', 'x','y','z','dip','az'])
   for I in range(0,len(co.index)):
      x=co.iat[I,2]
      y=co.iat[I,3]
      z=co.iat[I,4]
      di=0
      fur=es.loc[es['hole'] == co.iat[I,0]]
      fur=fur.sort_values('prof')
      ro=fur.shape[0]
      for j in range(0,ro):
         ang=fur.iat[j,6]
```

```
         d=fur.iat[j,1]-di
         di=d+di
         dip=fur.iat[j,5]
         if j>0:
            dip=fur.iat[j,5]-(fur.iat[j,5]-fur.iat[j-1,5])/2
         #--
         deltaz=d*np.sin(np.radians(dip))
         r=d*np.cos(np.radians(dip))
         x=round(x+r*np.sin(np.radians(ang)))
         y=round(y+r*np.cos(np.radians(ang)))
         z=round(z+deltaz)
         linha.loc[len(linha.index)]=[fur.iat[j,0] ,fur.iat[j,1],x
,y,z,fur.iat[j,5],fur.iat[j,6]]
         #--
      #---
   return linha.sort_values(['hole','prof'])


def calculo(azt,azb,dt,db,dp):
   dt = (90-dt)
   db = (90-db)
   dbt = db-dt
   abt = azb-azt
   d = np.arccos(np.cos(np.radians(dbt))-
np.sin(np.radians(dt))*np.sin(np.radians(db))*(1-np.cos(np.radians(abt))))
   r = 1
   if d==0:
    r = 1
   else:
    r = 2*np.tan(d/2)/d
   x = 0.5*dp*(np.sin(np.radians(dt))*np.sin(np.radians(azt))+
np.sin(np.radians(db))*np.sin(np.radians(azb)))*r
   y = 0.5*dp*(np.sin(np.radians(dt))*np.cos(np.radians(azt))+
np.sin(np.radians(db))*np.cos(np.radians(azb)))*r
   z = 0.5*dp*(np.cos(np.radians(dt))+np.cos(np.radians(db)))*r
   return [x,y,z]

def calcXYZ(hole,depth,Spts):
   t = Spts.loc[(Spts['hole']==hole) & (Spts['prof'] <=depth)]
   t = t.sort_values('prof',ascending=False)
   b = Spts.loc[(Spts['hole']==hole) & (Spts['prof'] > depth)]
   if b.shape[0] == 0:
    res = calculo(t.iat[0,6], t.iat[0,6], t.iat[0,5], t.iat[0,5], depth-t.iat[0,1])
    return [(t.iat[0,2]+res[0]), (t.iat[0,3]+res[1]), (t.iat[0,4]+res[2])]
   di = b.iat[0,1]- t.iat[0,1]
   rga = t.iat[0,5] - b.iat[0,5]
   stp = rga/di
   dpt = depth-t.iat[0,1]
   pang = stp*dpt
   if b.shape[0] >= 1:
      res = calculo(t.iat[0,6], b.iat[0,6],t.iat[0,5], t.iat[0,5]-pang, dpt)
      return[(t.iat[0,2]+res[0]), (t.iat[0,3]+res[1]), (t.iat[0,4]+res[2])]
   else:
      res = calculo(t.iat[0,6], t.iat[0,6], t.iat[0,5], t.iat[0,5], dpt)
      return [(t.iat[0,2]+res[0]), (t.iat[0,3]+res[1]), (t.iat[0,4]+res[2])]
```

A função **run** ficará assim:

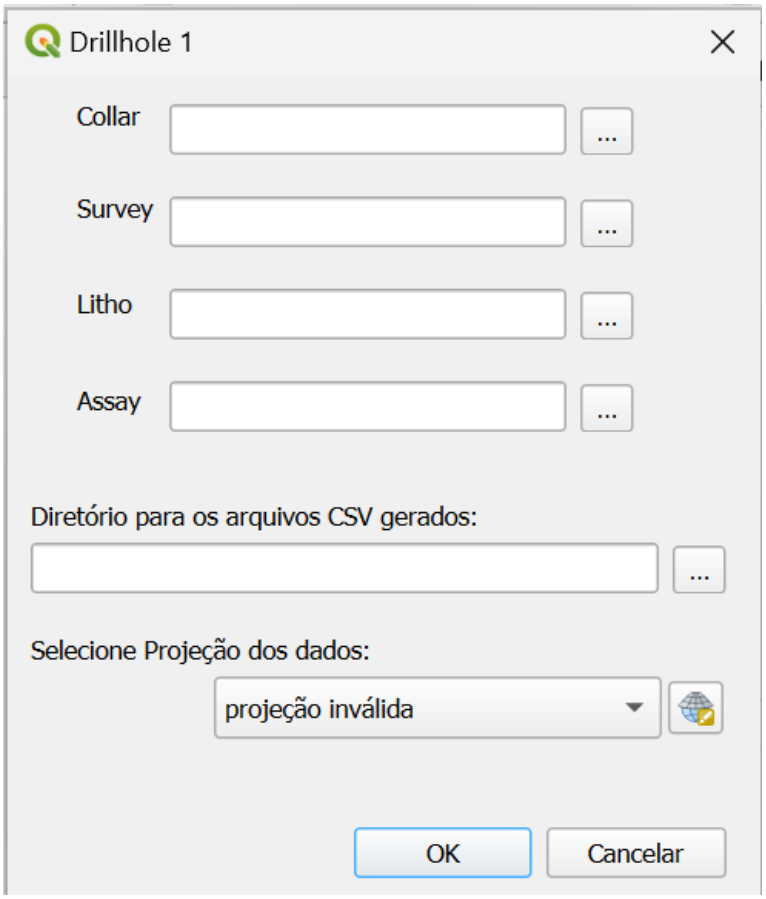
```
   def run(self):
               if self.first_start == True:
            self.first_start = False
            self.dlg = Drillhole1Dialog()

         # show the dialog
         self.dlg.show()
         # Run the dialog event loop
         result = self.dlg.exec_()
         # See if OK was pressed
```

```
        if result:
            col1=self.dlg.mQgsFileWidgetCollar.filePath()
            sur1=self.dlg.mQgsFileWidgetSurvey.filePath()
            lit1=self.dlg.mQgsFileWidgetLitho.filePath()
            asy1=self.dlg.mQgsFileWidgetAssay.filePath()
            dire=self.dlg.mQgsFileWidgetDir.filePath()
            if col1=="" or sur1=="" or lit1=="" or asy1=="" or dire=="":
                QmessageBox.warning(self.iface.mainWindow(),
                         'Erro',
                         "Entre todos os campos por favor\nSaindo...")
                return
            col = pd.read_csv (col1,sep=",")
            sur = pd.read_csv (sur1,sep=",")
            lit = pd.read_csv (lit1,sep=",")
            asy = pd.read_csv (asy1,sep=",")
            query = ''' SELECT col.holeid AS hole, col.x AS x, col.y AS y, col.z AS
z, asy.au AS au, asy."FROM" AS prof,lit."FROM" AS profl, asy."TO" AS toa, lit."TO"
AS tol, (asy."TO"-asy."FROM") AS lena, (lit."TO"-lit."FROM") AS lenl, lit.domain,
lit.rocktype, lit.weath FROM col INNER JOIN asy ON col.HOLEID = asy.HOLEID INNER
JOIN lit ON col.HOLEID = lit.HOLEID  WHERE  lit."FROM" BETWEEN asy."FROM" AND
(asy."TO"-0.02) ORDER BY hole,prof '''
            dat=sqldf(query, locals())
            query = ''' SELECT col.holeid AS hole,sur.at AS prof, col.x AS x, col.y
AS y, col.z AS z, sur.dip AS dip,sur.azm AS az FROM col INNER JOIN sur ON
col.holeid=sur.holeid WHERE sur.at=0 '''
            collar=sqldf(query, locals())
            collar['dip'] = collar['dip'].apply(lambda x: x*-1)
            query = ''' SELECT holeid AS hole, at AS prof, null AS x, null AS y,
null AS z, dip AS dip,azm FROM sur ORDER BY holeid,prof '''
            station=sqldf(query, locals())
            station['dip'] = station['dip'].apply(lambda x: x*-1)
            crs=self.dlg.mQgsProjectionSelectionWidget.crs()
            Spts=tresD(collar,station)
            for I in range(0,dat.shape[0]):
                dado = calcXYZ(dat.iat[i,0],dat.iat[i,5],Spts)
                dat.iat[i,1] = dado[0]
                dat.iat[i,2] = dado[1]
                dat.iat[I,3] = dado[2]

            Spts.to_csv(dire+"/topobase.csv",sep=',',index=False)
            temp = QgsVectorLayer("PointZ","desurveyed_intervals","memory")
            temp.setCrs(crs)
            temp_data = temp.dataProvider()
            # Criando os campos necessários
            temp_data.addAttributes([QgsField( "hole", Qvariant.String),
                                     QgsField( "x", Qvariant.Double),
                                     QgsField( "y", Qvariant.Double),
                                     QgsField( "z", Qvariant.Double),
                                     QgsField( "au", Qvariant.Double),
                                     QgsField( "prof", Qvariant.Double),
                                     QgsField( "profl", Qvariant.Double),
                                     QgsField( "toa", Qvariant.Double),
                                     QgsField( "tol", Qvariant.Double),
                                     QgsField( "lena", Qvariant.Double),
                                     QgsField( "lenl", Qvariant.Double),
                                     QgsField( "DOMAIN", Qvariant.String),
                                     QgsField( "ROCKTYPE", Qvariant.String),
                                     QgsField( "WEATH", Qvariant.String)])

            # Atualizando os campos
            temp.updateFields()
            temp.startEditing()
            # Adicionando cada ponto resultante do desurvey na camada ponto 3d
            for row in dat.itertuples():
                f = QgsFeature()
                f.setGeometry(QgsGeometry(QgsPoint(row.x,row.y,row.z)))
```

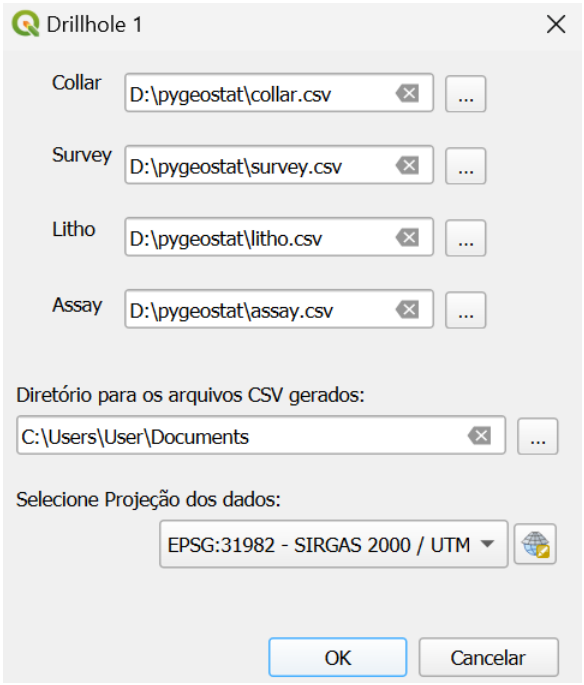
```
f.setAttributes([row.hole,row.x,row.y,row.z,row.au,row.prof,row.profl,row.toa,row.t
ol,row.lena,row.lenl,row.DOMAIN,row.ROCKTYPE,row.WEATH])
                temp_data.addFeature(f)

            # Aqui gravamos as mudanças das das camadas, adicionamos a camada e
exportamos como arquivo csv
            temp.updateExtents()
            temp.commitChanges()
            QgsProject.instance().addMapLayer(temp)

            v_layer = QgsVectorLayer('LineString?crs='+crs.authid(),
'drillholeline', 'memory')
            pr = v_layer.dataProvider()
            pr.addAttributes([QgsField( "hole", Qvariant.String)])
            v_layer.updateFields()

            topo_layer = QgsVectorLayer('PointZ?crs='+crs.authid(), 'collar',
'memory')
            prt = topo_layer.dataProvider()
            prt.addAttributes([QgsField( "hole", Qvariant.String)])
            topo_layer.updateFields()

            base_layer = QgsVectorLayer('PointZ?crs='+crs.authid(),
'base_of_survey', 'memory')
            prb = base_layer.dataProvider()
            prb.addAttributes([QgsField( "hole", Qvariant.String)])
            base_layer.updateFields()

            cabra="_"
            for index, row in Spts.iterrows():
                if cabra == "_":
                    start_point = QgsPoint(row['x'],row['y'],row['z'])
                    cabra=row.hole
                    f = QgsFeature()
                    f.setGeometry(QgsGeometry(start_point))
                    f.setAttributes([row.hole])
                    prt.addFeature(f)

                else:
                    if cabra!=row.hole and index>0:
                        end_point = QgsPoint(Spts.loc[(int(index) - 1),
'x'],Spts.loc[(int(index) - 1), 'y'],Spts.loc[(int(index) - 1), 'z'])
                        f = QgsFeature()
                        f.setGeometry(QgsGeometry(end_point))
                        f.setAttributes([row.hole])
                        prb.addFeature(f)
                        seg = QgsFeature()
                        seg.setGeometry(QgsGeometry.fromPolyline([start_point,
end_point]))
                        seg.setAttributes([row.hole])
                        pr.addFeatures([seg])
                        start_point = QgsPoint(row['x'],row['y'],row['z'])
                        cabra=row.hole
                        f = QgsFeature()
                        f.setGeometry(QgsGeometry(start_point))
                        f.setAttributes([row.hole])
                        prt.addFeature(f)

            QgsProject.instance().addMapLayers([v_layer])
            QgsProject.instance().addMapLayer(topo_layer)
            QgsProject.instance().addMapLayer(base_layer)

            renderer = temp.renderer()
            symbol1 = QgsMarkerSymbol.createSimple({'name':'dot red','color':
'red','size':'0.8'})
            symbol_layer1 = symbol1.symbolLayer(0)
            renderer.setSymbol(symbol1)
```

```
            temp.triggerRepaint()

            renderer = topo_layer.renderer()
            style = QgsStyle.defaultStyle().symbol('topo pop capital')
            renderer.setSymbol(style)
            renderer.symbol().setSize(2)
            topo_layer.triggerRepaint()

            renderer = base_layer.renderer()
            style = QgsStyle.defaultStyle().symbol('topo pop village')
            renderer.setSymbol(style)
            renderer.symbol().setSize(1.4)
            renderer.symbol().setColor(Qcolor("blue"))
            base_layer.triggerRepaint()

            self.iface.layerTreeView().refreshLayerSymbology(temp.id())
            self.iface.layerTreeView().refreshLayerSymbology(topo_layer.id())
            self.iface.layerTreeView().refreshLayerSymbology(base_layer.id())
            #gravando o csv do desurvey e finalizando
            QgsVectorFileWriter.writeAsVectorFormat(temp,dire+"/desurveyed.csv",
"utf-8",driverName = "CSV" , layerOptions = ['GEOMETRY=AS_XYZ'])
            QmessageBox.information(self.iface.mainWindow(),'Pronto','Desurvey
executado!')
            return
```

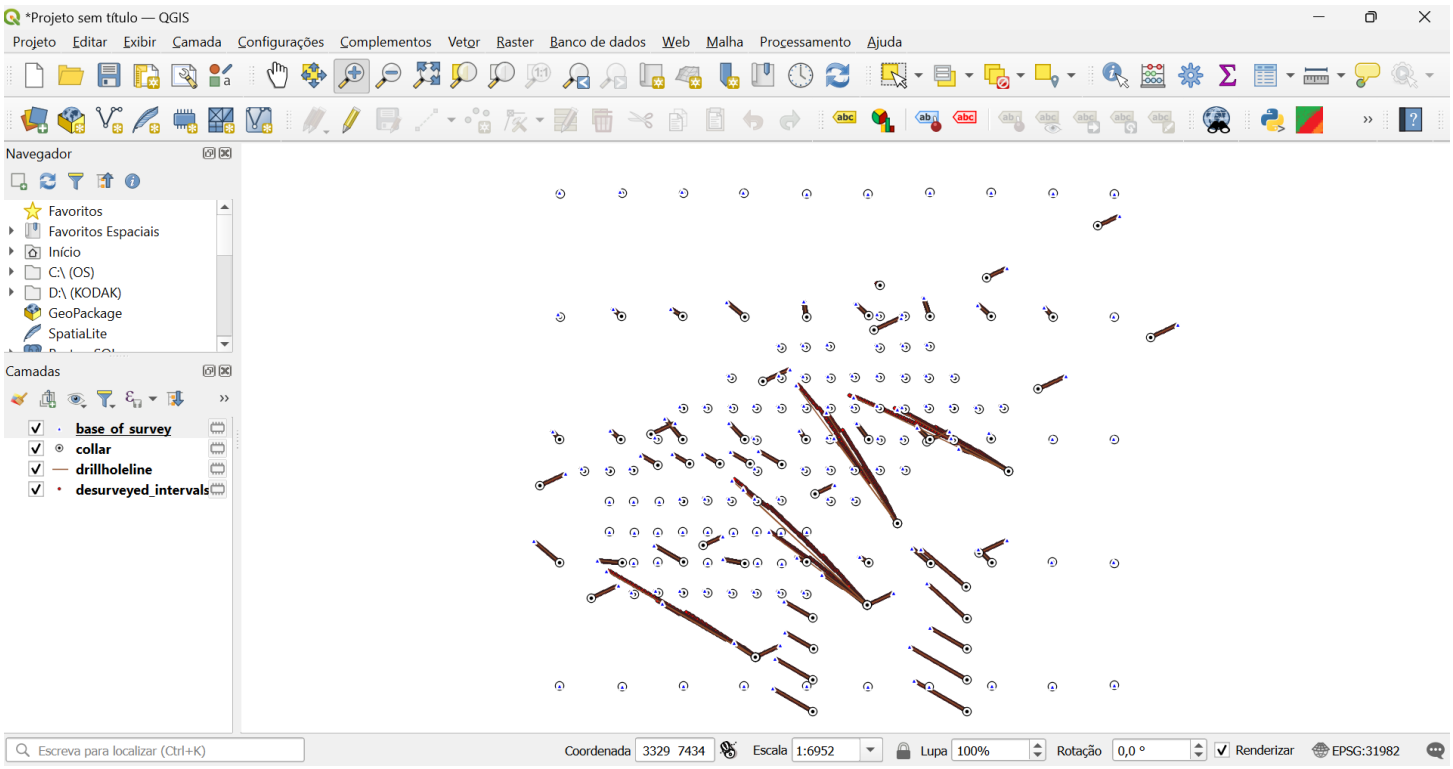
Baixe os arquivos CSV em https://gdatasystems.com/pyqgis/index.php
Abra o QGIS e o plugin será carregado já com as alterações feitas. Ao iniciarmos o plugin teremos:



Entre o caminho para os arquivos colar, survey, litho e assay. Defina o diretório onde os arquivos csv de resultado serão e escolha uma projeção UTM qualquer (os dados são em metros, mas não são referenciados a Datum nenhum).



Clique ok e aguarde uns 50 segundos para o processamento dos dados e geração dos arquivos e camadas temporárias resultantes. Ao concluir termos no QGIS:



A camada **desurveyed_intervals** representa o resultado do desurvey com todos os intervalos reprojetados em X, Y e Z e seus atributos, ver tabela de atributos abaixo:

desurveyed_intervals — Total de feições: 40493, Filtrado: 40493, Selecionado: 0

	hole	x	y	z	au	prof	profl	toa	tol	lena	lenl	DOMAIN	ROCKTYPE	WEATH
1	M001	3999	7499	727	0	0	0	2,5	2,5	2,5	2,5	D2	AvT	OX
2	M001	3998,99716594...	7499,00163624...	724,500002855...	0,71	2,5	2,5	5	5	2,5	2,5	D2	AvT	OX
3	M001	3998,98866377...	7499,00654496...	722,000022846...	0,96	5	5	7,5	7,5	2,5	2,5	D2	AvT	OX
4	M001	3998,97449357...	7499,01472613...	719,500077106...	0,48	7,5	7,5	10	10	2,5	2,5	D2	AvT	OX
5	M001	3998,95465543...	7499,02617969...	717,000182769...	1,42	10	10	12,5	12,5	2,5	2,5	D2	AvT	OX
6	M001	3998,92914947...	7499,04090557...	714,500356970...	2,16	12,5	12,5	15	15	2,5	2,5	D2	AvT	OX
7	M001	3998,89797588...	7499,05890365...	712,000616842...	1,34	15	15	17,5	17,5	2,5	2,5	D2	AvT	OX
8	M001	3998,86113487...	7499,08017381...	709,500979518...	0,39	17,5	17,5	20	20	2,5	2,5	D2	AvT	OX
9	M001	3998,81862669...	7499,10471592...	707,001462131...	0,43	20	20	22,5	22,5	2,5	2,5	D2	AvT	OX
10	M001	3998,77045163...	7499,13252980...	704,502081811...	0,87	22,5	22,5	25	25	2,5	2,5	D2	AvT	OX
11	M001	3998,71661003...	7499,16361527...	702,002855690...	0,23	25	25	27,5	27,5	2,5	2,5	D2	AvT	OX
12	M001	3998,65710225...	7499,19797210...	699,503800896...	0,34	27,5	27,5	30	30	2,5	2,5	D2	AvT	OX
13	M001	3998,59192870...	7499,23560007...	697,004934558...	0,94	30	30	32,5	32,5	2,5	2,5	D2	AvT	OX
14	M001	3998,52108983...	7499,27649891...	694,506273803...	1,42	32,5	32,5	35	35	2,5	2,5	D2	AvT	OX
15	M001	3998,44458612...	7499,32066834...	692,007835756...	0,88	35	35	37,5	37,5	2,5	2,5	D2	AvT	OX
16	M001	3998,36241810...	7499,36810807...	689,509637542...	0,26	37,5	37,5	40	40	2,5	2,5	D2	AvT	OX
17	M001	3998,27458633...	7499,41881777...	687,011696282...	0,5	40	40	42,5	42,5	2,5	2,5	D2	AvT	MX
18	M001	3998,18109141...	7499,47279709...	684,514029098...	0	42,5	42,5	45	45	2,5	2,5	D2	AvT	MX

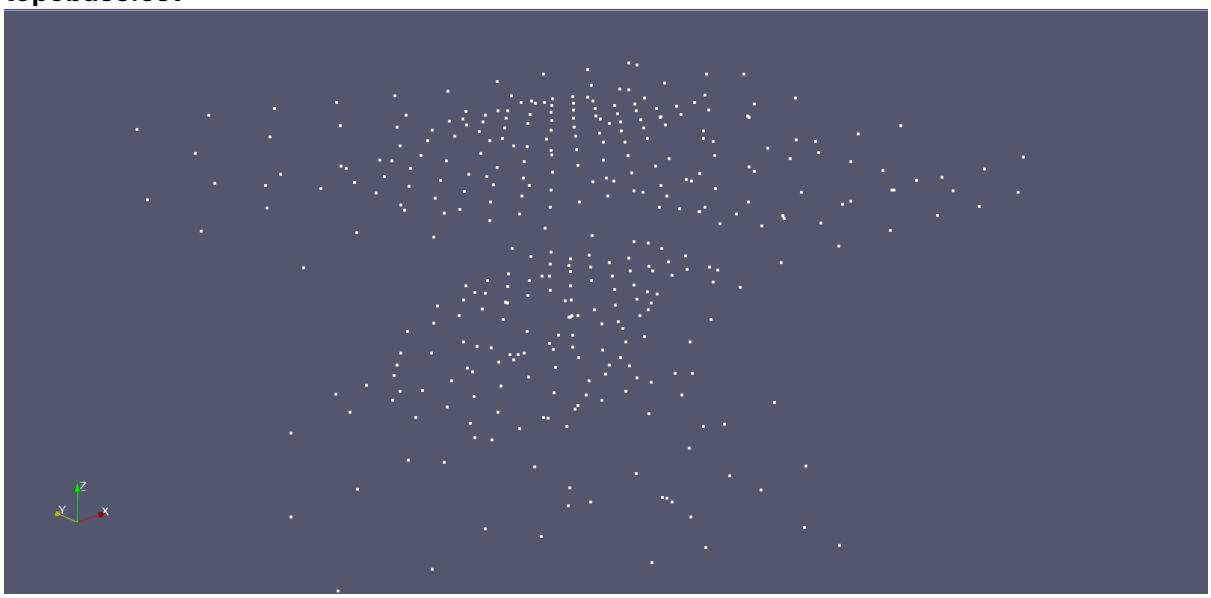
Mostrar todos os feições

A camada **collar** é posição da boca do furo e a camada **base_of_survey** a base levantada do furo.

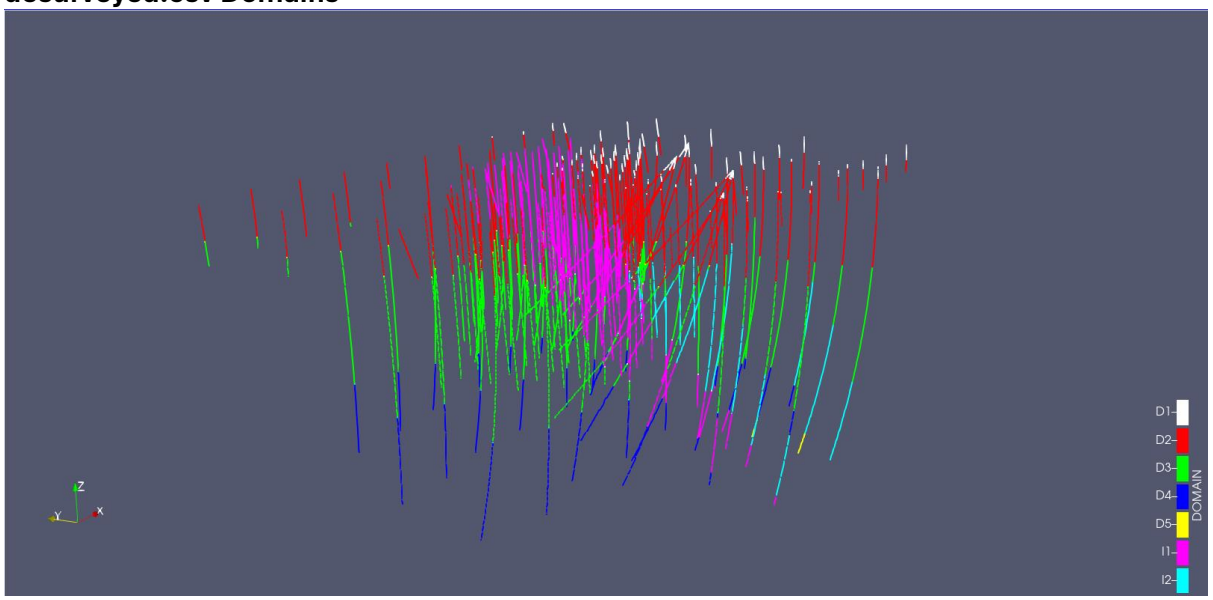
A camada **drillholeline** é a projeção em planta do furo.

Além dessas camadas foram gerados dois arquivos csv. O arquivo **desusrveyed.csv** com o resultado do desurvey e o arquivo **topobase.csv** com os pontos de medidas do survey no furo (estação). Abrindo estes arquivos com o Paraview usando o filtro TableToPoint teremos:

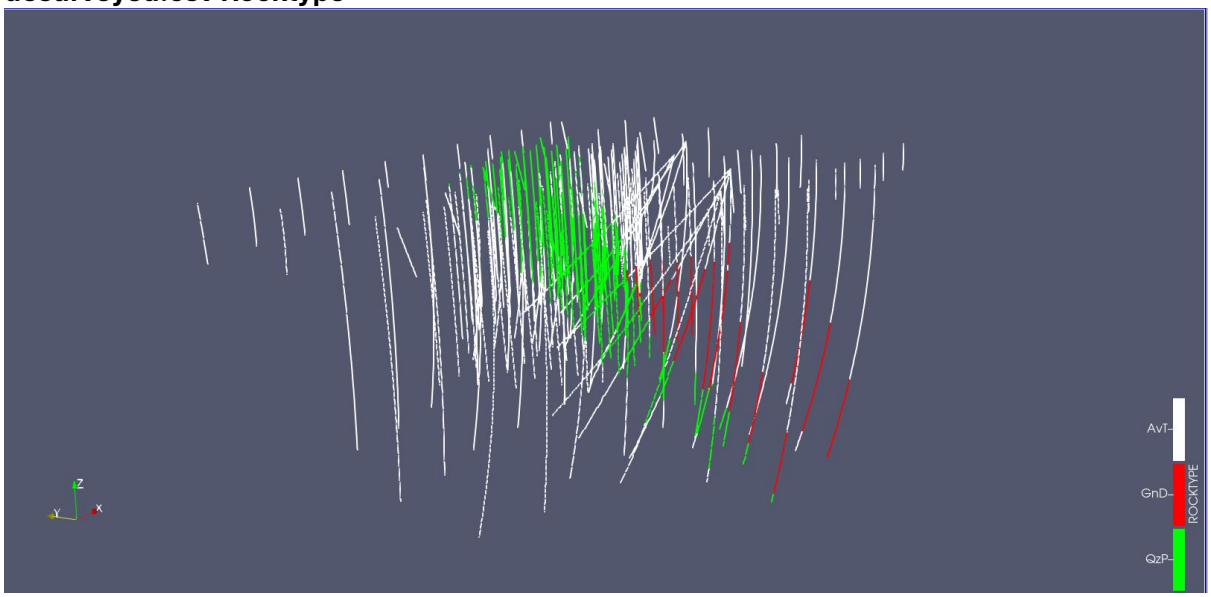
topobase.csv



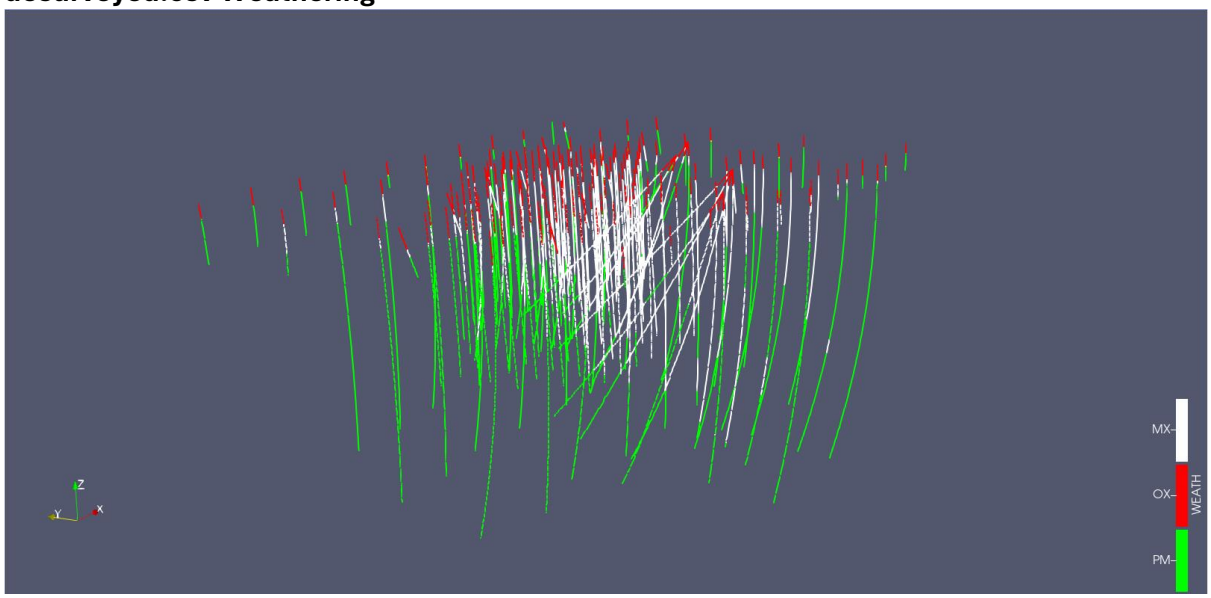
desurveyed.csv Domains



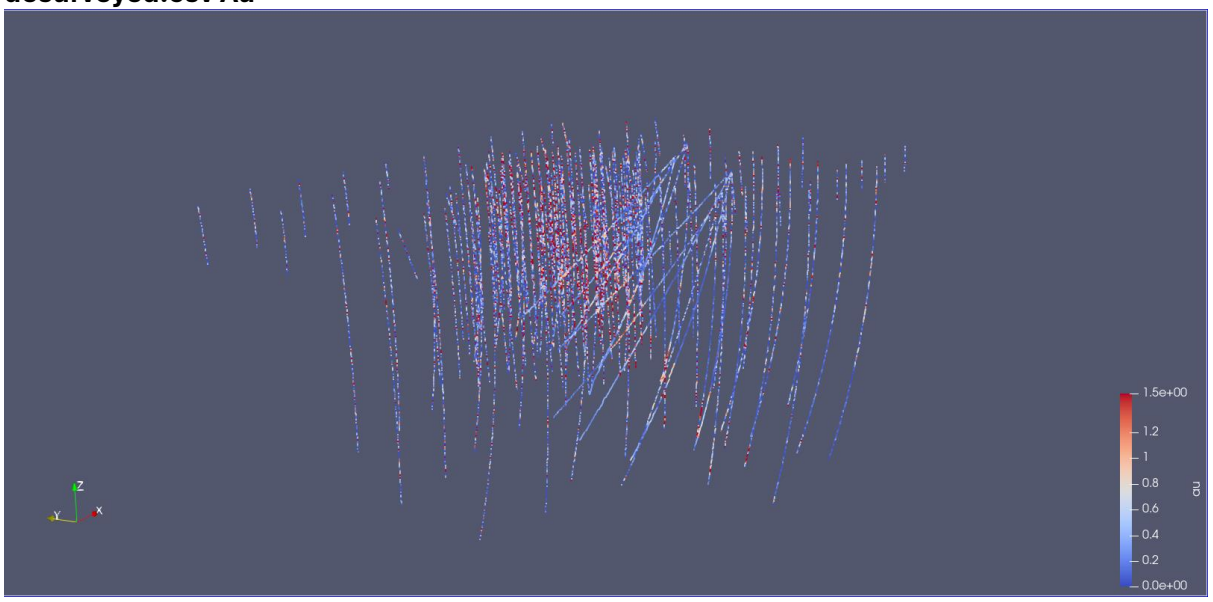
desurveyed.csv Rocktype



desurveyed.csv Weathering

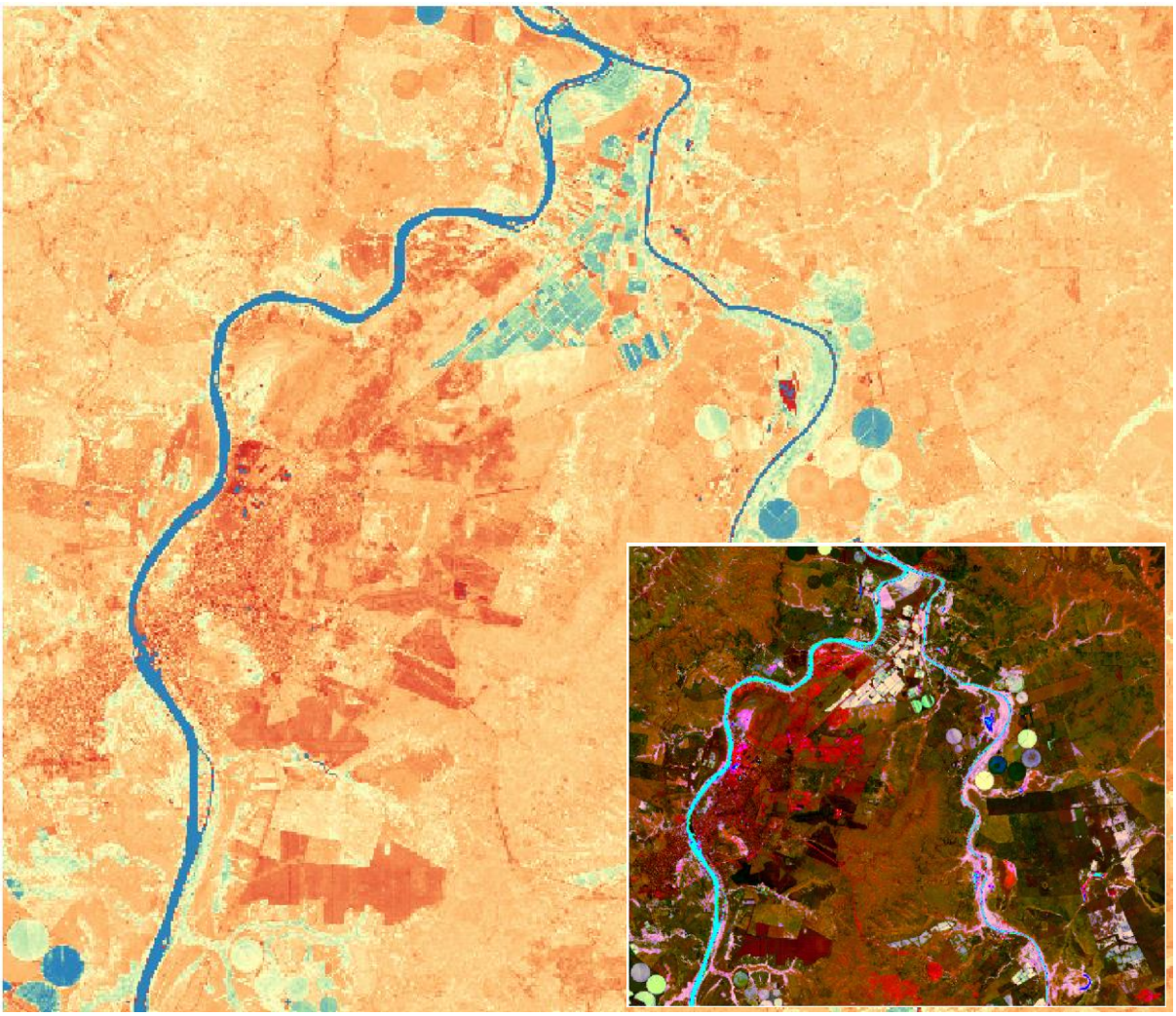


desurveyed.csv Au



Revisitaremos este plugin num módulo mais adiante implementando mais flexibilidade dos dados originais.

No próximo módulo vamos criar um plugin que executa processamento em imagens do Sentinel2.



Criando Plugins QGIS com pyQGIS

Plugin IndexOrama

1 - Introdução

Vamos criar um Plugin que abrirá as bandas 2, 3, 4, 5, 6, 7, 8A, 11 e 12 do Sentinel2 e criar um Stack destas bandas em um único arquivo multibanda (9 bandas). Deste arquivo vamos gerar uma series de índices normalizados, comuns em análises de vegetação, presença de água, umidade e solo exposto.

Os índices gerados serão:

NDVI - Normalized Difference Vegetation Index **(nir-red)/(nir+red)**

NDVIre1 - red-edge based NDVI **(nir-vnir)/(nir+vnir)**

SAVI - Soil Adjusted Vegetation Index **(nir-red)/(nir+red+0.5)*1.5**

NDWI – Normalized Difference Water Index **(green-nir)/(green +nir)**

MNDI – Modified Normalized Difference Water Index **(green-swir1)/(green +swir1)**

NDMI - Normalized Difference Moisture Index **(nir-swir1)/(nir +swir1)**

NDTI - Normalized Difference Tillage Index **(swir1-swir2)/(swir1 +swir2)**

NDBI - Normalized Difference Built-up Index **(swir1-nir)/(swir1 +nir)**

Onde:

red = banda4 vermelho visível

nir = banda8A infravermelho próximo

vnir – banda5 infravermelho muito próximo

green = banda3 verde visível

swir = banda11 infravermelho de onda curta

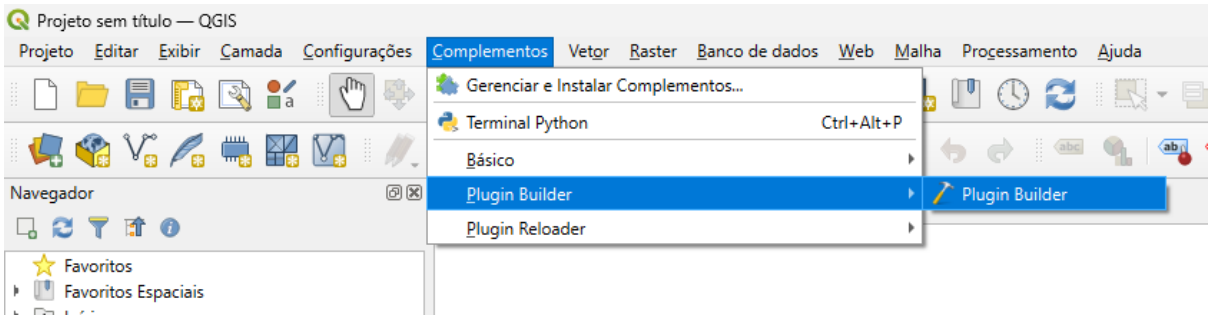
swir2 = banda12 infravermelho de onda curta

As bandas originais foram recortadas e as bandas com 10m de resolução (2 ,3, 4 e 8A) tiveram a resolução transformada para 20 metros para serem compatíveis com a resolução das bandas 5, 11 e 12.

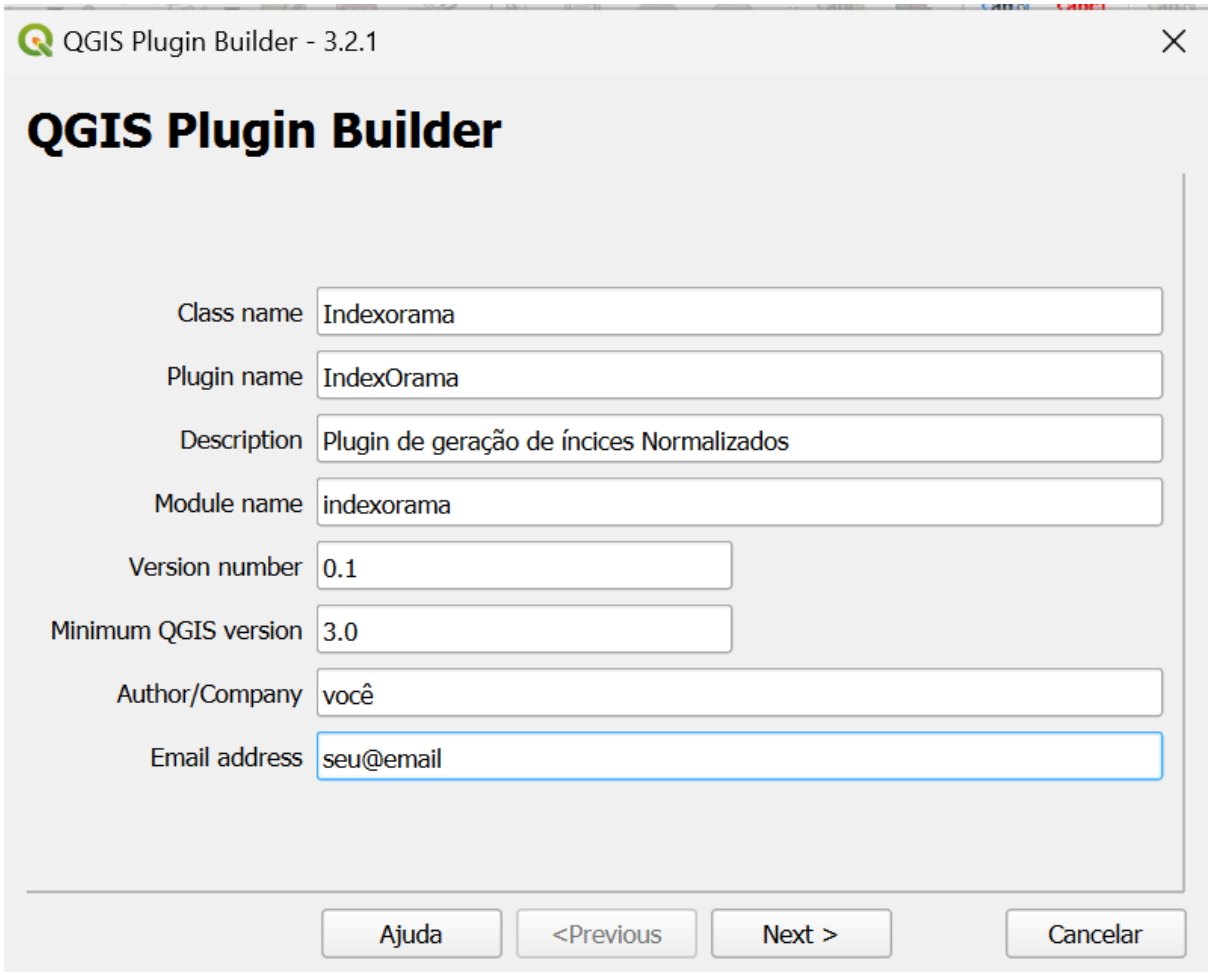
O plugin também gera uma composição RGB com os índices que é boa para a classificação do terreno.

2 - Construindo o esqueleto do IndexOrama no Plugin Builder 3

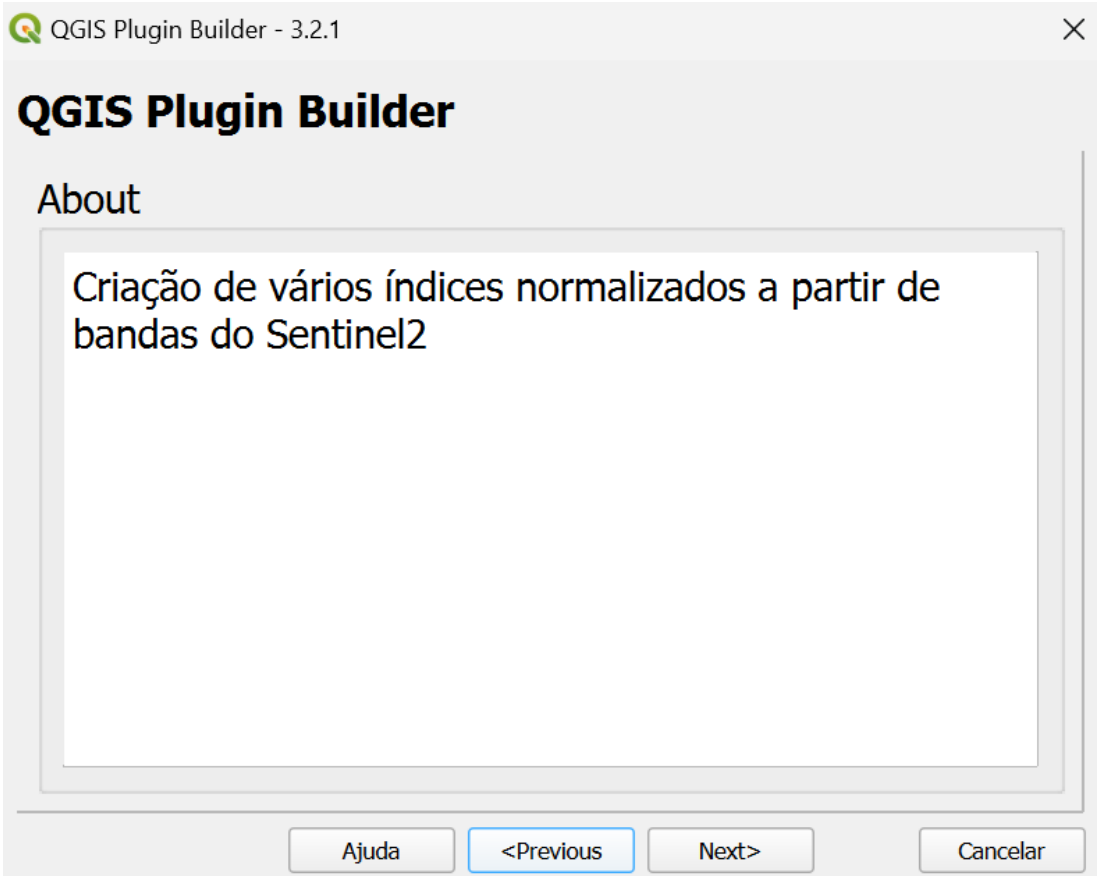
Inicie o Plugin Builder:



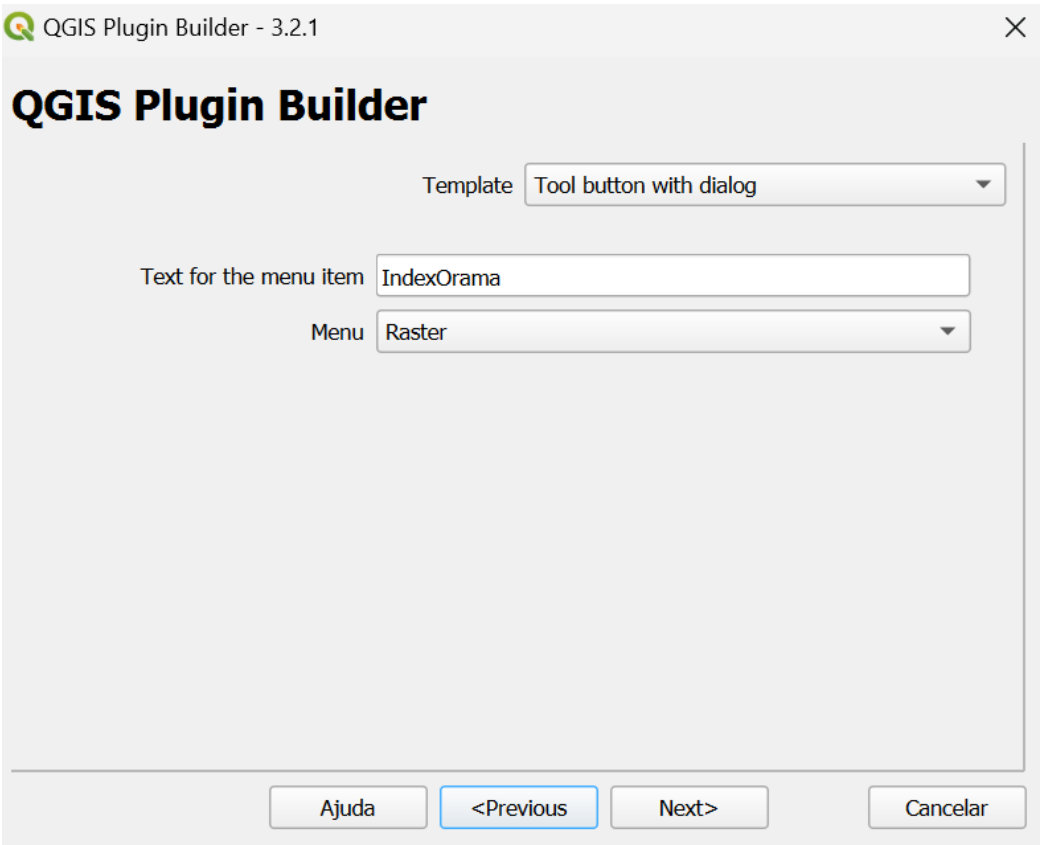
Preencha os campos dos formulários conforme as imagens a seguir:



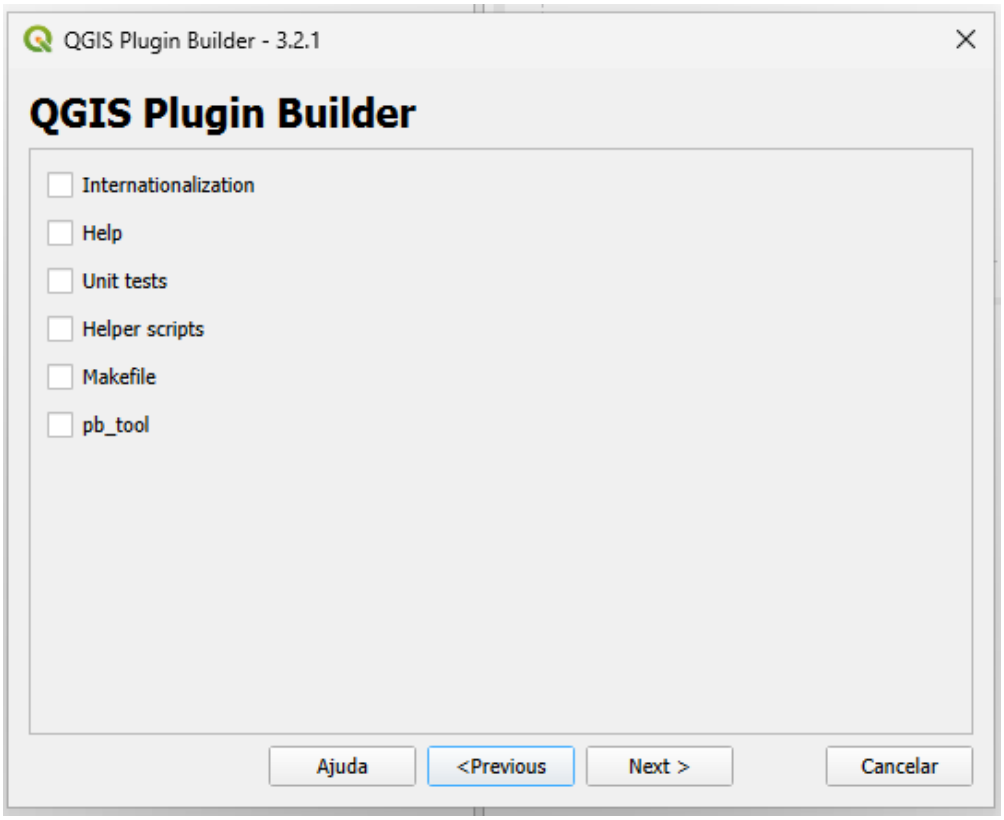
Esse primeiro formulário será usado na criação do arquivo metadata.txt e na definição do nome das classes do plugin.



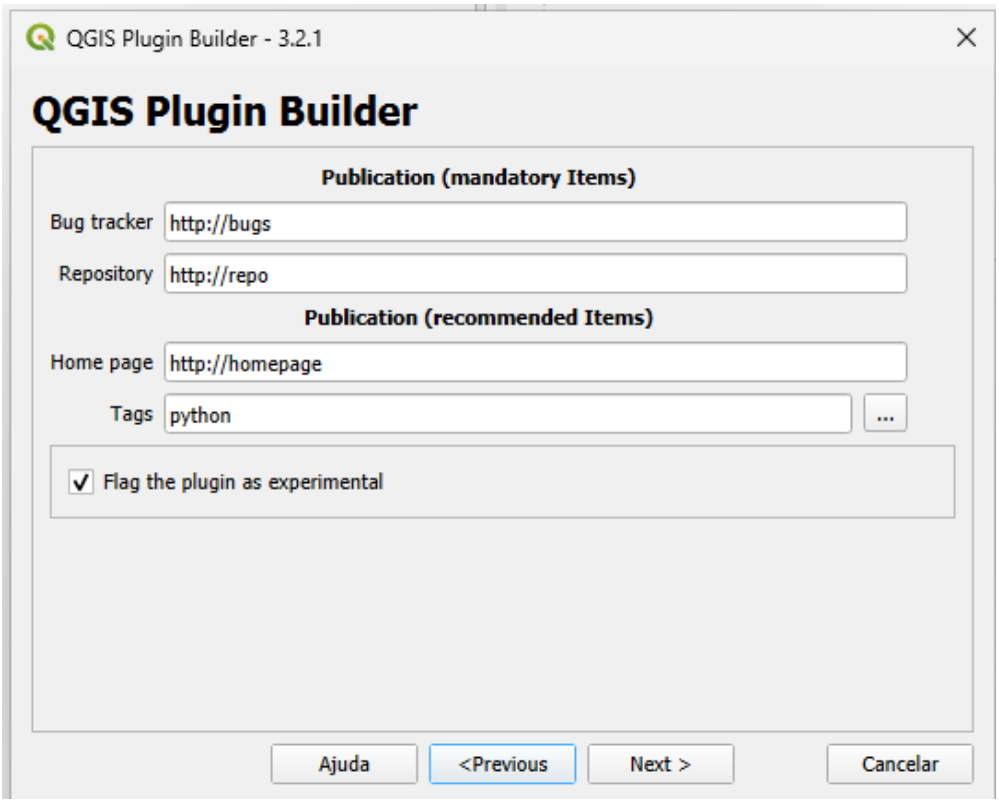
Descrição mais detalhada sobre o plugin que também será colocado no arquivo metadata.txt.



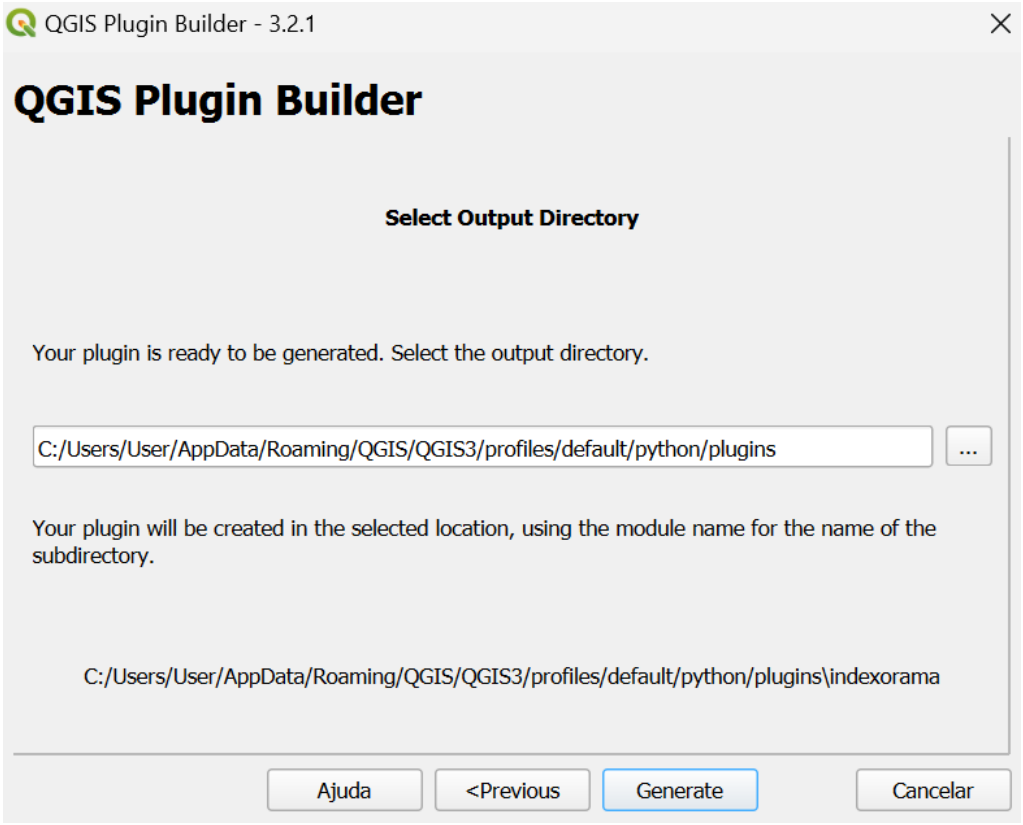
Template (tipo) do plugin, texto que vai aparecer no menu e em qual menu será listado o plugin.



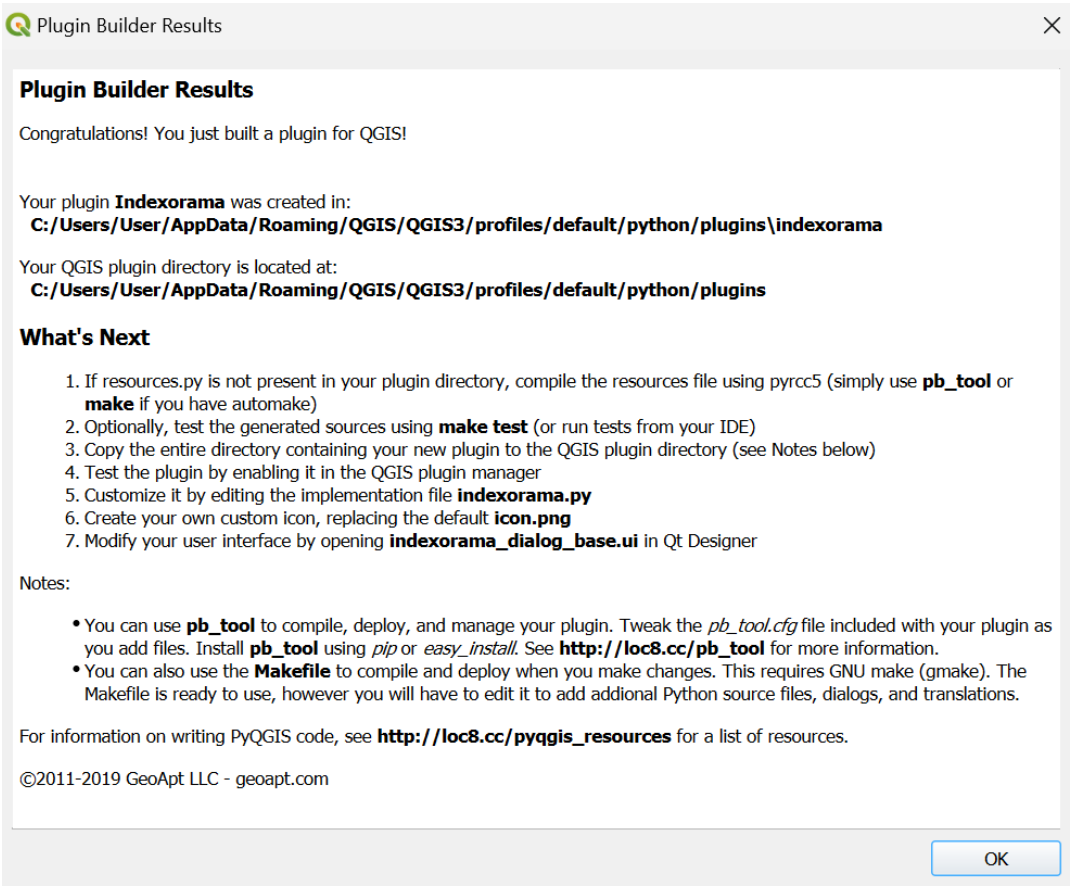
Desmarque todos para esse plugin,



Cheque a caixa de plugin experimental pois não iremos distribuir esse plugin no momento.



A pasta de plugin do sistema (nesse caso em sistema Windows). Clique **Generate** após selecionar o diretório de plugins.



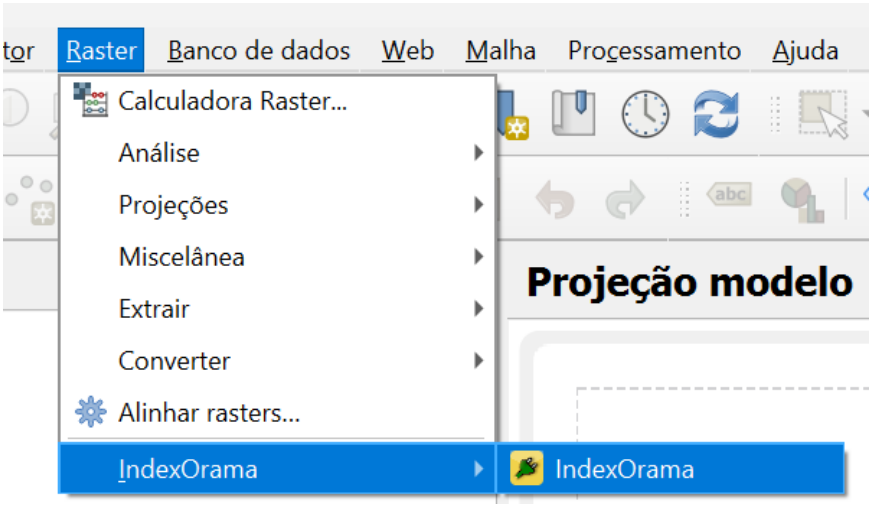
Pronto, os arquivos base de seu plugin foram criados na pasta:

C:/Users/User/AppData/Roaming/QGIS/QGIS3/profiles/default/python/plugins\indexorama

Os oito arquivos necessários mais dois arquivos README com instruções do PluginBuilder foram criados automaticamente.

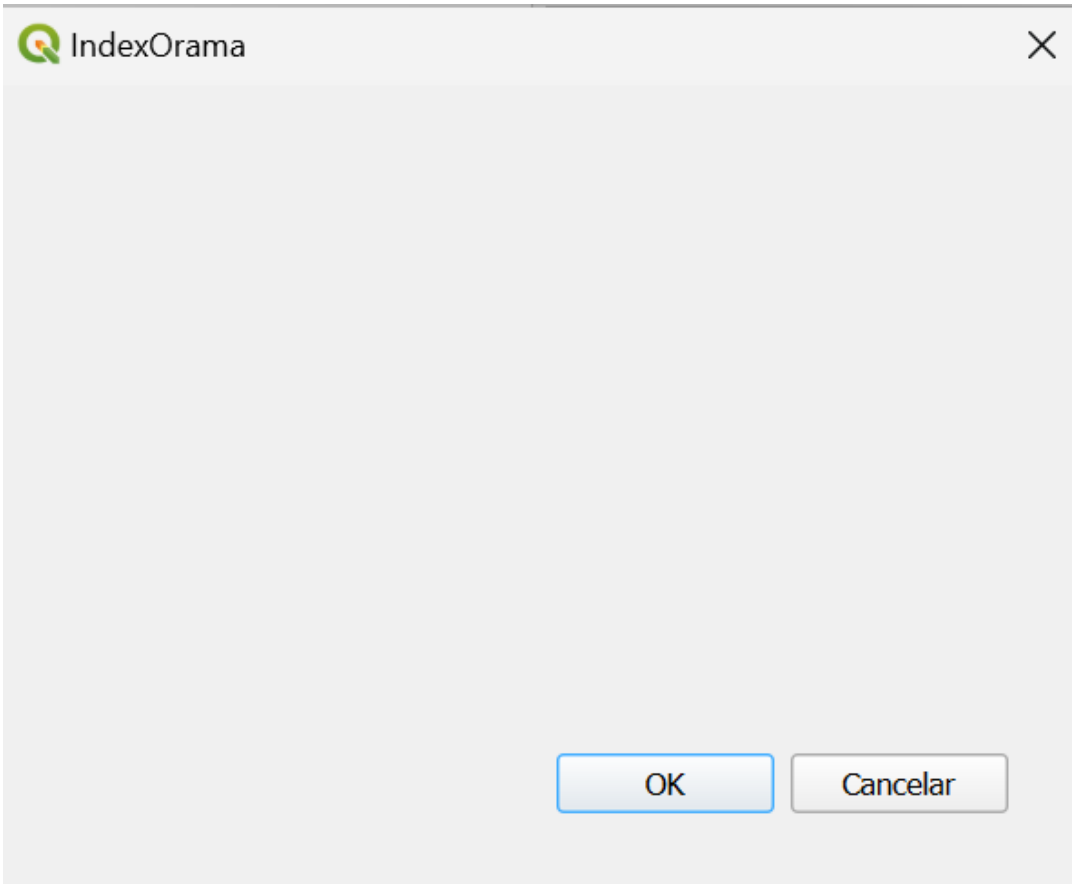
Vamos testar ele iniciando o QGIS e abrindo o **Complementos->Gerenciar e instalar Complementos**. Em Instalados vemos que ele não foi instalado ainda. Marque ele e instale para testarmos.

Inicie ele pelo Menu Vetor.



Ou pelo ícone na barra de ferramentas.

Funcionando, mas sem funcionalidade ainda.



Vamos construir a interface gráfica do usuário (GUI) e adicionar a funcionalidade agora na próxima seção.

3 - O plugin IndexOrama

A intrerface gráfica do plugin IndexOrama que tem como objetivo abrir 9 arquivos correspondentes a cada banda do sentinel2 e um widget para definir a pasta onde os arquivos de índices gerados serão gravados.

NOTA: As bandas usadas têm que estar todas na mesma resolução. Ou seja, todas em 20m resolução ou todas em 10m de resolução.

Os arquivos com as bandas estão disponíveis em https://gdatasystems.com/pyqgis/index.php

Agora execute o QtDesigner para criarmos a nossa interface gráfica de usuário (GUI).

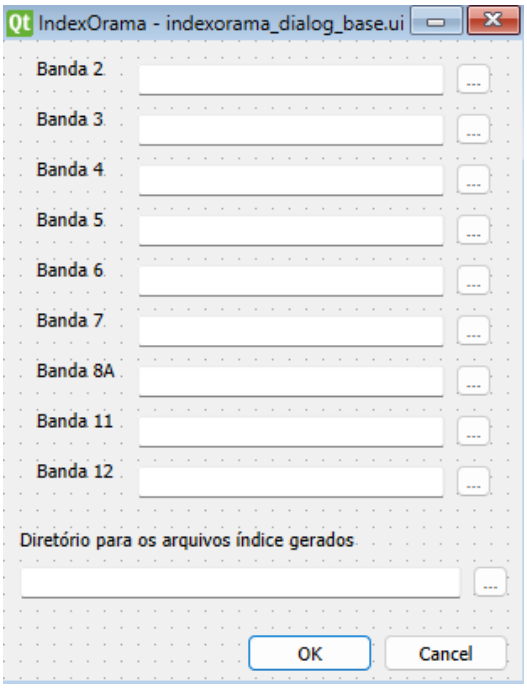
Abra o arquivo **drillhole1_dialog_base.ui** localizado em:

C:\Users\User\AppData\Roaming\QGIS\QGIS3\profiles\default\python\plugins\drillhole1

No primeiro diálogo do programa em **Open.**

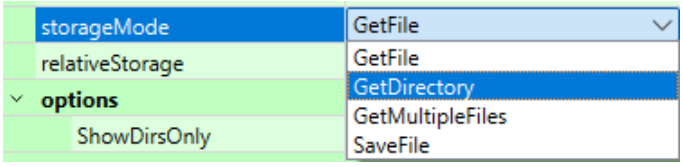
Vamos adicionar 5 widgets do tipo Label, 5 widgets do tipo QgsFileWidget e 1 QgsProjectionSelectionWidget. Basta clicar no Widget e arrastar até a janela do diálogo.

A aparência final da interface deve ser:



Modifique o texto dos campos Label.

No último QgsFileWidget altere o componente **StorageMode** para **GetDirectory**:



Agora altere a propriedade objectName dos campos QgsFileWidget na seguinte ordem.

mQgsFileWidget_1

mQgsFileWidget_2

mQgsFileWidget_3

mQgsFileWidget_4

mQgsFileWidget_5

mQgsFileWidget_6

mQgsFileWidget_7

mQgsFileWidget_8

mQgsFileWidget_9

mQgsFileWidgetDir

Pronto, salve o diálogo e feche o QtDesigner.

Vamos agora editar o arquivo **indexorama.py** para realizar a tarefa. Vamos ter de adicionar algumas bibliotecas de suporte via **import**.

As bibliotecas no arquivo **indexorama.py** serão (adicionar as faltantes):

```
from qgis.PyQt.QtCore import QSettings, QTranslator, QCoreApplication
from qgis.PyQt.QtGui import QIcon
from qgis.PyQt.QtWidgets import QAction, QMessageBox
from qgis.core import QgsProject,QgsRasterLayer
import processing
# Initialize Qt resources from file resources.py
from .resources import *
# Import the code for the dialog
from .indexorama_dialog import IndexoramaDialog
import os.path
import rasterio
import numpy as np
```

A função **run** ficará assim:

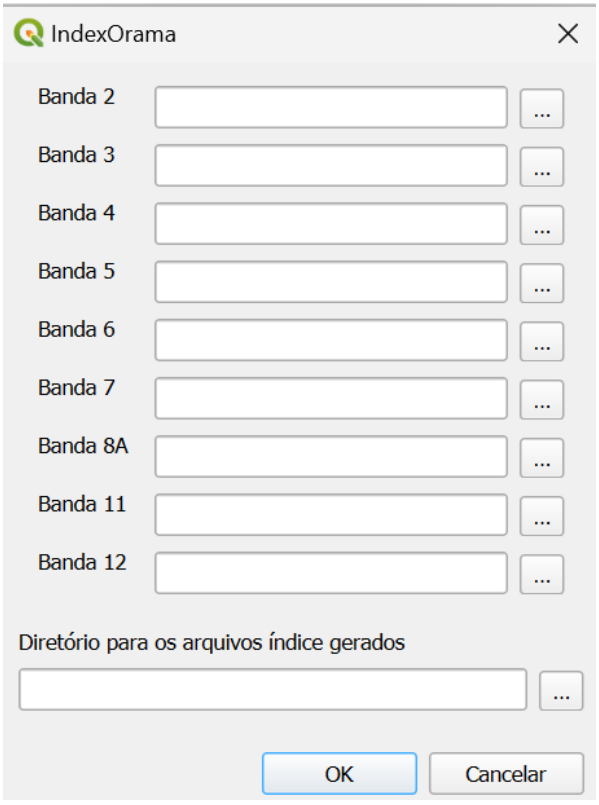
```
        def run(self):
         if self.first_start == True:
             self.first_start = False
             self.dlg = IndexoramaDialog()

         # show the dialog
         self.dlg.show()
         # Run the dialog event loop
         result = self.dlg.exec_()
         # See if OK was pressed
         if result:
             b2=self.dlg.mQgsFileWidget_1.filePath()
             b3=self.dlg.mQgsFileWidget_2.filePath()
             b4=self.dlg.mQgsFileWidget_3.filePath()
             b5=self.dlg.mQgsFileWidget_4.filePath()
             b6=self.dlg.mQgsFileWidget_5.filePath()
             b7=self.dlg.mQgsFileWidget_6.filePath()
             b8a=self.dlg.mQgsFileWidget_7.filePath()
```

```
            b11=self.dlg.mQgsFileWidget_8.filePath()
            b12=self.dlg.mQgsFileWidget_9.filePath()
            dire=self.dlg.mQgsFileWidgetDir.filePath()
            if b2=="" or b3=="" or b4=="" or b5=="" or b6=="" or b7=="" or b8a=="" 
or b11=="" or b12=="" or dire=="":
                QMessageBox.warning(self.iface.mainWindow(),
                        'Erro',
                        "Entre todos os campos por favor\nSaindo...")
                return

            band2=rasterio.open(b2)
            band3=rasterio.open(b3)
            band4=rasterio.open(b4)
            band5=rasterio.open(b5)
            band6=rasterio.open(b6)
            band7=rasterio.open(b7)
            band8a=rasterio.open(b8a)
            band11=rasterio.open(b11)
            band12=rasterio.open(b12)
            band2_rgb = band2.profile
            band2_rgb.update({"count": 9})
            with rasterio.open(dire+'\\fullstack.tiff', 'w', **band2_rgb) as dest:
                dest.write(band2.read(1),1)
                dest.write(band3.read(1),2)
                dest.write(band4.read(1),3)
                dest.write(band5.read(1),4)
                dest.write(band6.read(1),5)
                dest.write(band7.read(1),6)
                dest.write(band8a.read(1),7)
                dest.write(band11.read(1),8)
                dest.write(band12.read(1),9)

            stack=QgsRasterLayer(dire+"\\fullstack.tiff","stack")
            QgsProject.instance().addMapLayer(stack)
            output1 = dire+"\\ndvi.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':7,'INPUT_B':stack,'BAND_B':3,'FORMULA':'((A-
B)/(A+B))','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output1})
            output2 = dire+"\\ndvire1.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':7,'INPUT_B':stack,'BAND_B':4,'FORMULA':'((A-
B)/(A+B))','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output2})
            output3 = dire+"\\savi.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':7,'INPUT_B':stack,'BAND_B':3,'FORMULA':'((A-
B)/(A+B+0.5)*1.5)','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output
3})
            output4 = dire+"\\ndwi.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':7,'INPUT_B':stack,'BAND_B':2,'FORMULA':'((B-
A)/(B+A))','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output4})
            output5 = dire+"\\mndwi.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':8,'INPUT_B':stack,'BAND_B':2,'FORMULA':'((B-
A)/(B+A))','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output5})
            output6 = dire+"\\ndmi.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':7,'INPUT_B':stack,'BAND_B':8,'FORMULA':'((A-
B)/(A+B))','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output6})
            output7 = dire+"\\ndti.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':8,'INPUT_B':stack,'BAND_B':9,'FORMULA':'((A-
B)/(A+B))','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output7})
            output8 = dire+"\\ndbi.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':stack,'BAND_A':7,'INPUT_B':stack,'BAND_B':8,'FORMULA':'((B-
A)/(B+A))','NO_DATA':None,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output8})
            QgsProject.instance().addMapLayer(QgsRasterLayer(output1,"ndvi"))
```

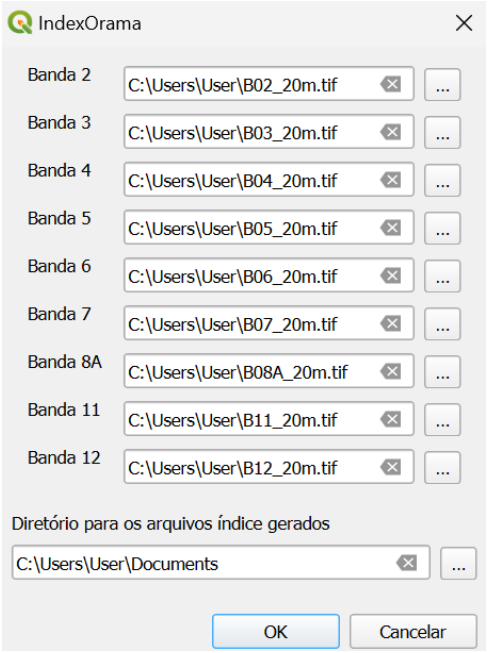
```
            QgsProject.instance().addMapLayer(QgsRasterLayer(output2,"ndvire1"))
            QgsProject.instance().addMapLayer(QgsRasterLayer(output3,"savi"))
            QgsProject.instance().addMapLayer(QgsRasterLayer(output4,"ndwi"))
            QgsProject.instance().addMapLayer(QgsRasterLayer(output5,"mndwi"))
            QgsProject.instance().addMapLayer(QgsRasterLayer(output6,"ndmi"))
            QgsProject.instance().addMapLayer(QgsRasterLayer(output7,"ndti"))
            QgsProject.instance().addMapLayer(QgsRasterLayer(output8,"ndbi"))
            NDBI=QgsRasterLayer(output8,"ndbi2")
            NDTI=QgsRasterLayer(output7,"ndti2")
            output9 = dire+"\\ndbi_ndti.tiff"
            processing.run("gdal:rastercalculator",
{'INPUT_A':NDBI,'BAND_A':1,'INPUT_B':NDTI,'BAND_B':1,'FORMULA':'A+B','NO_DATA':None
,'RTYPE':5,'OPTIONS':'','EXTRA':'','OUTPUT':output9})
            NDTI_NDBI=QgsRasterLayer(output9,"ndti_ndbi")
            bandb=rasterio.open(output5)
            bandg=rasterio.open(output3)
            bandr=rasterio.open(output9)
            band_rgb = bandr.profile
            band_rgb.update({"count": 3})
            with rasterio.open(dire+'\\classificado.tiff', 'w', **band_rgb) as
dest2:
                dest2.write(bandb.read(1),1)
                dest2.write(bandg.read(1),2)
                dest2.write(bandr.read(1),3)

            classi=QgsRasterLayer(dire+"\\classificado.tiff","classi")
            QgsProject.instance().addMapLayer(classi)
            QMessageBox.information(self.iface.mainWindow(),'Pronto','IndexOrama
executado!')
            return
```

Baixe os arquivos com as bandas em https://gdatasystems.com/pyqgis/index.php

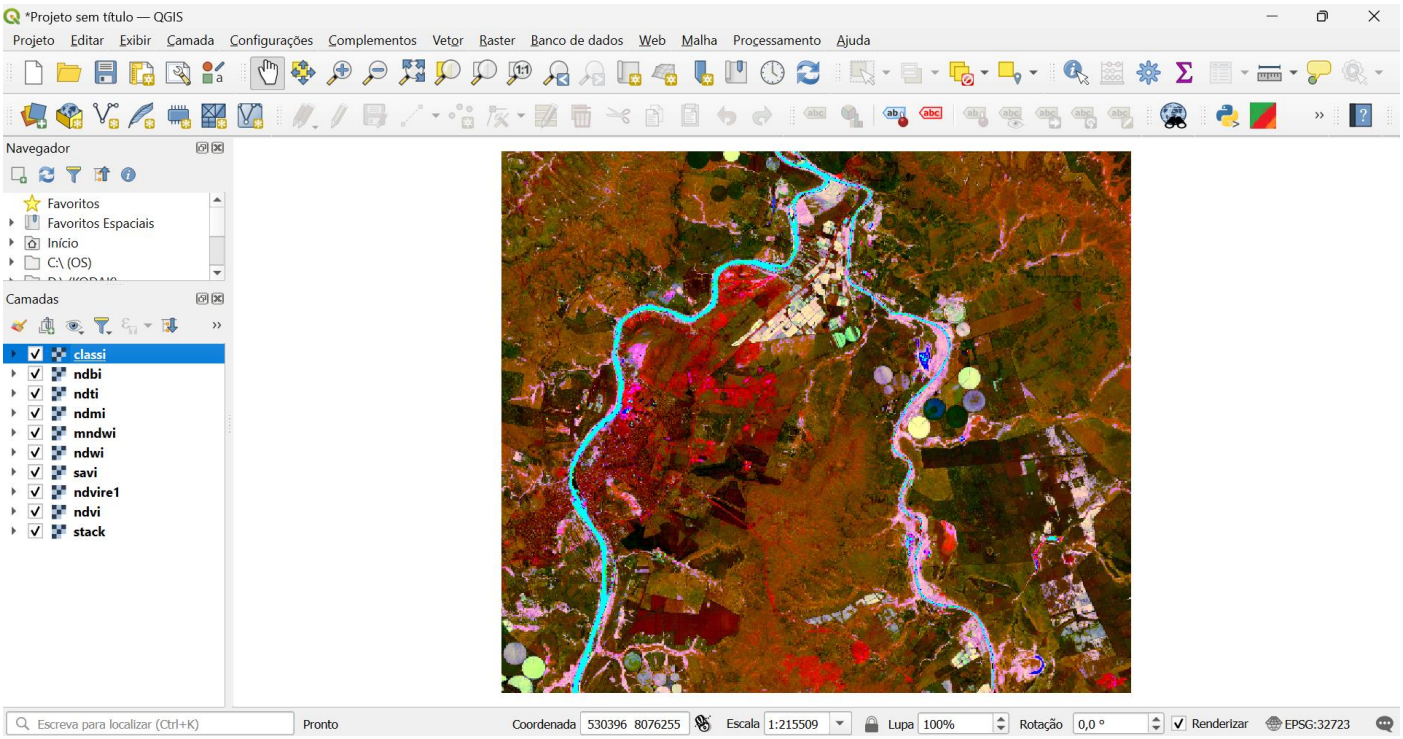
Abra o QGIS e o plugin será carregado já com as alterações feitas. Ao iniciarmos o plugin teremos:



Entre o caminho para os arquivos das bandas de acordo com título. Defina o diretório onde os arquivos gerados serão gravados.



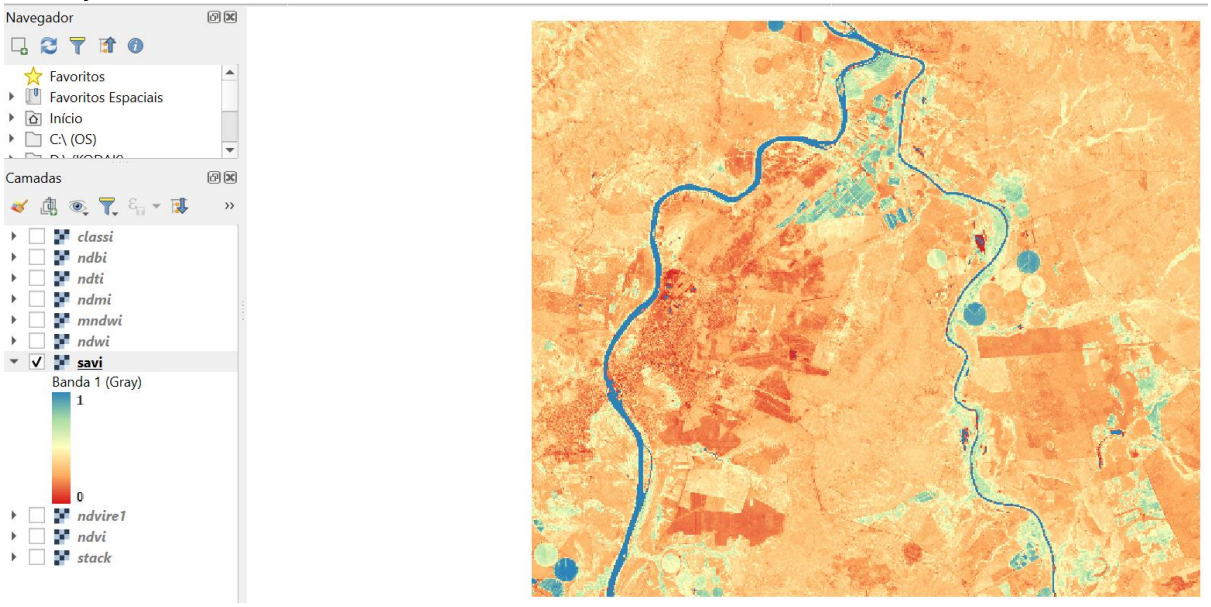
Clique ok e aguarde o processamento dos dados e geração dos arquivos de camadas resultantes. Ao concluir teremos no QGIS 10 camadas raster criadas. Uma composição de 9 bandas chamada stack, 8 camadas de uma banda dos índices e uma composição RGB que usa os índices (MNDWI no azul, SAVI no verde e NDTI+NDBI no vermelho) para uma “Classificação” do terreno.



Os índices apresentam-se em escala de cinza e a faixa deve ser ajustada apropriadamente de 0 a 1 para melhor resultado. Podemos também usar falsa cor para melhorar a visualização.

SAVI gerado originalmente:

SAVI ajustado de 0 a 1 e usando falsa cor:



Finalizamos aqui os exemplos de plugins. Fique ligado no site para novos módulos futuros de plugins avançados.