Новое API для Аксиомы.ГИС на языке Python

(api2, axioma2 api, новое апи)

> *В чем отличие от старого апи на Питоне?*

- Простота и удобство

Новое апи менее полное, но за счет этого более простое и удобное. Так для большинства задач новое апи будет простым и достаточным, а для более специфичных задач остается старое апи.

- Лицензия

Старое апи использует PyQt5 (как и в QGIS), которое имеет ограничения по лицензии. Так ваш модуль должен быть выпущен под лицензией GPL или совместимой с ней. Это накладывает ограничения на бизнес модель модуля - скорее это продажа поддержки, в то время как код модуля открытый и бесплатный. (Поддержку на вашу разработку может начать продавать также любой желающий.) Либо вам придется преобретать коммерческую лицензию PyQt5.

Новое апи использует PySide2 он же Qt for Python. Он идеально подходит для комерческого использования. Любая ваша разработка с его использованием будет совместима с LGPL и её можно лицензировать на свое усмотрение.

- Новое апи поставляется с Руководством Разработчика

> *Чего (пока) нет в новом апи, по сравнению со старым?*

- Регистрация растров
- Рабочие наборы
- Готовые диалоги
- Прямой доступ к координатам геометрий

> *Что будет со старым апи?*

Мы продолжаем его поддерживать. Аксиома 3 будет иметь оба апи на выбор. Но нужно понимать что старое апи прекращает развиваться - только исправление ошибок; в то время как в новом апи будут активно появляться новые возможности.

В декабрьском релизе axioma2 api имеет "предварительную версию" - это подготовка к выпуску Аксиомы.ГИС 3, в котором апи уже будет иметь "рекомендуемый" статус. В этот предварительный период мы продолжим его разработку и учтем отзывы пользователей. Окончательная версия может несколько отличаться от предварительной.

Как читать эту документацию

Действительная документация составлена таким образом, чтобы описать общие принципы и подходы решения тех или иных базовых задач, необходимых при обработке геопространственных данных. Более полное описание всевозможных классов, свойств, методов, исключений, функций и их параметров содержится в Справочнике Функций.

> **NOTE:** axioma2 api имеет Справочник Функций

Документ логически структурирован. Рекомендуется читать его в прямом порядке от начала до конца, чтобы получить общее представление о возможностях, предоставляемых API.

Инициализация ядра

Аксиома.Питон может быть использован 3 способами:

- в приложении Аксиома.ГИС из панели `Консоль Python` ;
- в модуле, который будет загружен в Аксиоме.ГИС;
- как SDK, независимо от десктопного приложения Аксиома.ГИС.

В первых двух случаях ядро будет инициализировано за вас. В последнем случае для использования axioma2 api необходимо самостоятельно инициализировать ядро. Если забыть это сделать, то при попытке использования будет брошено исключение, которое напомнит произвести инициализацию.

In [1]:
```
from axioma2.cs import CoordSystem


try:
    crs = CoordSystem.from_epsg(4326)
except RuntimeError as error:
    print(f"Поймано исключение: {error}")
```

```
Поймано исключение: axioma2.core.Core is not initialized
```

Инициализация ядра выглядит следующим образом:

In [2]:
```
from axioma2 import init_axioma
init_axioma() # инициализация


from axioma2.cs import CoordSystem
crs = CoordSystem.from_epsg(4326)
crs.name
```

Чтобы упростить импорт разных типов, используйте агрегирующий модуль `axioma2all` , ключающий все классы и функции. Например:

In [3]:
```
from axioma2all import *  # импортируем все типы Аксиомы в текущее пространство имен

cyl = CoordSystem.from_epsg(4088)
cyl.name
```

Out[3]:
```
'World Equidistant Cylindrical (Sphere)'
```

Работа с Координатными системами (КС)

Классы и функции для работы с КС находятся в модуле `axioma2.cs` . Основные типы:

- CoordSystem
- CoordTransformer

In [4]:
```
merc = CoordSystem.from_epsg(3395)
merc.name
```

Out[4]:
```
'Меркатора WGS84'
```

Доступные функции создания КС из представления

Из модуля `axioma2.cs`

- from_epsg(int) -> CoordSystem
- from_wkt(str) -> CoordSystem
- from_proj(str) -> CoordSystem
- from_prj(str) -> CoordSystem
- from_units(LinearUnit) -> CoordSystem
- from_string(str) ->CoordSystem

Аналогичные свойства для обратной конвертации КС в представление

- epsg -> int
- wkt -> str
- proj -> str
- prj -> str
- wgs84_params -> List[float]

Функция `from_string` позволяет создавать КС из "универсального представления" - строки с префиксом типа, двоеточием и значением. Возможные префиксы: «proj», «wkt», «epsg», «prj».

In [5]:
```
crs = CoordSystem.from_string('prj:Earth Projection 12, 62, "m", 0')
crs.name
```

Out[5]:
```
'Робинсона NAD27'
```

Трасформация координат из одной КС в другую

In [6]:
```
from PySide2.QtCore import QPointF


transformer = CoordTransformer('epsg:4326', 'epsg:26953')
coordinate = QPointF(55.76, 37.6)
result = transformer.transform(coordinate)
f"Point({result.x()}, {result.y()})"
```

Out[6]:
```
'Point(8513601.095442554, 9873107.576049749)'
```

Работа с таблицами

Открытие таблиц

Работа с источником данных начинается с открытия `Объекта данных` спомощью функций `open` , `openfile` . Для таблиц возвращаемый Объект данных будет `Таблицей` .

In [7]:
```
from axioma2.core import io


table = io.openfile('../path/to/datadir/worldcap.tab')
table.name
```

Out[7]:
```
'worldcap'
```

Некоторые форматы могут содержать несколько таблиц в одном файле, например GeoPackage. В таком случае нужно указать в параметре `dataobject=` , какую таблицу из файла вы хотите открыть.

In [8]:
```
table = io.openfile('../path/to/datadir/example.gpkg', dataobject='world')
table.name
```

Out[8]:
```
'world'
```

Источники данных и дополнительные параметры

Источником данных может быть не только файл. Например, это может быть База Данных. Также для открытия или создания некоторых объектов данных может понадобиться указать дополнительные параметры, применимые только для этого типа источника.

Для открытия таких Объектов данных используется функция `open`, которая принимает словарь со всеми необходимыми параметрами.

Пример открытия таблицы из базы данных Postgre

```
from axioma2.core import io
definition = {
    "src": "<Адрес сервера БД>",
    "port": 5432,
    "db": "<Имя базы данных>",
    "user": "<Имя прользователя>",
    "password": "<Пароль>",
    "dataobject": '"DataAxi"."World"',
    "provider": "PgDataProvider"
}
table = io.open(definition)
```

Функцию `open` можно использовать и для рассмотренного нами ранее простого открытия из файла. Можно сказать, что она более универсальна. Функция `openfile` реализована через функцию `open`:

In [9]:
```
definition = { 'src': '../path/to/datadir/example.gpkg', 'dataobject': 'world'
}
table = io.open(definition)
table.name
```

Out[9]: 'world'

Открытие источников с множеством таблиц

При открытии Таблицы функцией `open` есть параметр `dataobject`. Он содержит имя таблицы, которую необходимо открыть. Для источников данных с единственной Таблицей этот параметр можно опустить. Для источников с множеством таблиц, например, База данных или файл GeoPackage, этот параметр обязателен.

Для получения всех доступных таблиц используется функция `read_contents`:

In [10]:
```
from axioma2.core import io
io.read_contents('../path/to/datadir/example.gpkg')
```

Out[10]:
```
['world', 'worldcap']
```

Для источников с единственной таблицей список будет содержать одно имя:

In [11]:
```
io.read_contents({'src': '../path/to/datadir/worldcap.tab'})
```

Out[11]:
```
['worldcap']
```

Провайдеры

Дополнительным параметром при открытии таблиц является Провайдер. Обычно Аксиома может сама определить, каким провайдером открывать файл.

Получим список загруженных провайдеров. Это пары значений (Идентификатор: Описание)

In [12]:
```
io.loaded_providers()
```

Out[12]:
```
{'CsvDataProvider': 'Файловый провайдер: Текст с разделителями',
 'DwgDgnFileProvider': 'Провайдер DWG и DGN (Версия 5.0.19.123)',
 'GdalDataProvider': 'Растровый провайдер GDAL',
 'MifMidDataProvider': 'Провайдер данных MIF-MID',
 'OgrDataProvider': 'Векторный провайдер OGR',
 'PgDataProvider': 'PostgreSQL',
 'RestDataProvider': 'ArcGIS REST',
 'SqliteDataProvider': 'Векторный провайдер sqlite',
 'TabDataProvider': 'MapInfo',
 'TerplanDataProvider': 'Провайдер территориального планирования',
 'TileDataProvider': 'Растровый провайдер тайлов',
 'TmsDataProvider': 'Тайловые сервисы',
 'WfsDataProvider': 'Web Feature Service',
 'WmsDataProvider': 'Web Map Service',
 'WmtsDataProvider': 'Web Map Tile Service',
 'XlsDataProvider': 'Провайдер чтения файлов Excel'}
```

Но иногда провайдер необходимо задать явно. Например, если один тип файлов может быть открыт несколькими провайдерами, или при подключении к СУБД.

In [13]:
```
table = io.openfile('../path/to/datadir/example.gpkg', dataobject='worldcap',
provider='SqliteDataProvider')
table.name
```

Out[13]:
```
'worldcap'
```

У таблицы можно узнать провайдер, которым она была открыта:

In [14]:
```
table.provider
```

Out[14]:
```
'SqliteDataProvider'
```

Схема таблицы

Записи таблицы имеют фиксированную структуру, повторяющую столбцы таблицы. Схема в Аксиома.Питон представлена словарем с элементами 'geometry' и 'properties'. Геометрия это специальный атрибут указывающий на то, что таблица является пространственной. Остальные атрибуты перечислены в 'properties' в том же порядке, что и столбцы таблицы.

In [15]:
```
table = io.openfile('../path/to/datadir/worldcap.tab')
table.schema()
```

Out[15]:
```
{'coordsystem': 'prj:Earth Projection 12, 62, "m", 0',
 'properties': [['Столица', 'string:25'],
  ['Capital', 'string:25'],
  ['Страна', 'string:30'],
  ['Country', 'string:30'],
  ['Cap_Pop', 'decimal:8.0']]}
```

Схема имеет простую стандартрую структуру и с ней просто работать. Например, можно легко вывести все имена атрибутов:

In [16]:
```
schema = table.schema()
[attr[0] for attr in schema['properties']]
```

Out[16]:
```
['Столица', 'Capital', 'Страна', 'Country', 'Cap_Pop']
```

Типы атрибутов

Тип атрибута представляется строкой. В нем может быть указана максимальная длина - для строк и десятичного типа через двоеточие, например, `string:254`. И точность - для десятичного типа через точку, например, `decimal:7.3`.

Доступные типы:

- string - строка
- int - целое число
- double - вещественное число с плавающей запятой
- decimal - вещественное число с фиксированной запятой
- bool - логическое значение
- date - дата
- time - время
- datetime - дата и время

Специальный атрибут Система Координат 'coordsystem' содержит значение КС в "универсальном представлении" и указывает на то, что таблица пространственная - может содержать геометрию и стиль.

Создание схемы таблицы. Вспомогательные функции

Создание схемы и атрибутов в виде текста позволяет делать синтаксические ошибки и может оказаться сложным. Для удобства используйте вспомогательные функции для задания атрибутов из модуля `axioma2.core`:

- attr.string(name: str, length: int = 80)
- attr.decimal(name: str, length: int = 10, precision: int = 5)
- attr.int(name: str)
- attr.double(name: str)
- attr.bool(name: str)
- attr.date(name: str)
- attr.time(name: str)
- attr.datetime(name: str)

и для создания схемы таблицы:

- attr.schema(*attributes, coordsystem=None)

Например, создадим схему, аналогичную схеме таблицы выше:

In [17]:
```
schema = attr.schema(
    attr.string('Столица', 25),
    attr.string('Capital', 25),
    attr.string('Страна', 30),
    attr.string('Country', 30),
    attr.decimal('Cap_Pop', 8, 5),
    coordsystem='prj:Earth Projection 12, 62, "m", 0'
)

schema
```

Out[17]:
```
{'coordsystem': 'prj:Earth Projection 12, 62, "m", 0',
 'properties': [['Столица', 'string:25'],
  ['Capital', 'string:25'],
  ['Страна', 'string:30'],
  ['Country', 'string:30'],
  ['Cap_Pop', 'decimal:8.5']]}
```

Обратные функции позволяют получить длину или точность типа:

- get_length(attribute) -> int
- get_precision(attribute) -> int

Выведем название, длину и точность всех "простых" атрибутов из схемы выше:

In [18]:
```
for attribute in schema['properties']:
    print('Название:%8s; Длина:%3d; Точность:%2d;'
          %(attribute.name, attribute.length, attribute.precision))
```

```
Название: Столица; Длина: 25; Точность: 0;
Название: Capital; Длина: 25; Точность: 0;
Название:  Страна; Длина: 30; Точность: 0;
Название: Country; Длина: 30; Точность: 0;
Название: Cap_Pop; Длина:  8; Точность: 5;
```

Записи

Запись, которая получается при чтении из таблицы, во многом повторяет словарь Python `dict` . В ней содержатся пары (Имя столбца: Значение). Причем значение приводится к типу столбца. Например, если это числовое поле int, а его значение 42, то запись будет содержать число 42, а не строку "42".

Прочитанная запись никак не ссылается на таблицу и является копией. Присвоенная к переменной запись не будет инвалидирована после продвижения итератора или даже после закрытия таблицы. Но поэтому и изменения, внесенные в эту запись-копию, никак не повлияют на значения в таблице.

In [19]:
```
table = io.openfile('../path/to/datadir/example.gpkg', dataobject='worldcap')
read_copy = next(table.items())
print(f'До изменения: capital = {read_copy["capital"]}')
read_copy['capital'] = 'New Value'
print(f'После изменения: capital = {read_copy["capital"]}')
read_again = next(table.items())
print(f'Снова читаем из таблицы: capital = {read_again["capital"]}')
```

```
До изменения: capital = Abidjan
После изменения: capital = New Value
Снова читаем из таблицы: capital = Abidjan
```

Чтение записей

Объект Таблица дает возможность работать с Записями. Метод items() возвращает итератор по записям.

In [20]:
```
features = table.items()

read_count = 0
for feature in features:
    read_count += 1
print(read_count)
```

```
215
```

Возвращается именно Итератор. При чтении он движется вперед и в конце становится пустым.

In [21]:
```
sum(1 for feature in features)
```

Out[21]:
```
0
```

Чтобы начать чтение сначала, нужно создать новый итератор.

In [22]:
```
features = table.items()
sum(1 for feature in features)
```

Out[22]:
```
215
```

Атрибуты

Доступ атрибутам осуществляется по имени. Так же как для словаря, если атрибут с заданым именем несуществует, то бросается исключение `KeyError` .

In [23]:
```
feature = next(table.items())
try:
    feature['unknown_key']
except KeyError as e:
    print(f'Поймано исключение: {e}')
```

```
Поймано исключение: "Key 'unknown_key' not found"
```

Проверка существования атрибута спомощью ключевого слова `in` так же, как в словаре:

```
In [24]: 'unknown_key' in feature, 'capital' in feature
Out[24]: (False, True)
```

Геометрический атрибут

Доступ к специальному атрибуту Геометрия через константу `GEOMETRY_ATTR` :

```
In [25]: feature = next(table.items())
         geom = feature[GEOMETRY_ATTR]
         geom.wkt
Out[25]: 'POINT (-379483.313800274 569367.593387095)'
```

Проверка существования:

```
In [26]: GEOMETRY_ATTR in feature
Out[26]: True
```

Идентификаторы записей

Записи имеют значение `id` . Это значение зависит от типа данных. При этом не гаратнируется порядок, начальное значение или отсутствие разрывов между соседними записями. Также идентификатор необязательно является числом.

Создание таблиц

Создавать таблицы несколько сложнее, чем открывать готовые. Для них нужно определить схему, КС, Провайдер и пр. Все эти параметры были рассмотрены выше.

Создадим простую непространственную таблицу с двумя текстовыми столбцами `country` и `capital` :

```
In [27]: definition = {
             'src': '../path/to/datadir/newtable.tab',
             'dataobject': 'new table',
             'schema': attr.schema(
                 attr.string('country', 60),
                 attr.string('capital', 60),
             )
         }
         newtable = io.create(definition)
         newtable.name
Out[27]: 'new table'
```

Проверим схему, автоматически выбранный провайдер и факт отсутствия записей - новая таблица пустая:

In [28]:
```
newtable.provider
```

Out[28]:
```
'TabDataProvider'
```

In [29]:
```
newtable.schema()
```

Out[29]:
```
{'properties': [['country', 'string:60'], ['capital', 'string:60']]}
```

In [30]:
```
newtable.count()
```

Out[30]:
```
0
```

Добавим в таблицу несколько записей из вселенной Властелин Колец:

In [31]:
```
features_to_insert = [
    Feature({'country': 'Мордор', 'capital': 'Барад-Дур'}),
    Feature(country='Гондор', capital='Минас Тирит'),  # создание с использова
нием **kwargs
    Feature({'country': 'Рохан'}),  # не обязательно подавать все значения, он
и будут пустыми
]
newtable.insert(features_to_insert)
# Количество записей в таблице после вставки
newtable.count()
```

Out[31]:
```
3
```

Редактирование таблиц

Один из возможных способов редактирования таблиц - открыть исходную таблицу, создать целевую таблицу с той же или другой структурой. И при чтении записей из исходной таблицы редактировать их и записывать в целевую. В результате получится отредактированная копия.

Например, для таблицы `world` необходимо оставить только колонки "Страна" и "Население"; и оставить только те страны, население которых больше `100 миллионов` .

Вот один из вариантов, как это можно реализовать.

In [32]:
```
def is_over_100million(feature) -> bool:
    return feature['Население'] > 100_000_000


orig_table = io.openfile('../path/to/datadir/world.tab')
definition = {
    'src': '../path/to/datadir/edit/edited_world.tab',
    'schema': attr.schema(
        attr.string('Страна'),
        attr.int('Население'),
    )
}
copy_table = io.create(definition)
orig_features = orig_table.items()

filtered_features = filter(is_over_100million, orig_features)
copy_table.insert(filtered_features)

# Посмотрим результат
for f in copy_table.items():
    print(f['Страна'])
```

```
Бангладеш
Бразилия
Китай
Индия
Индонезия
Япония
Российская Федерация
Соединенные Штаты Америки
```

Запросы

SQL-запросы являются очень мощным инструментом. Часто его использование окажется наиболее эффективным/производительным. Вот пример запроса с тем же результатом, что и пример редактирования выше:

In [33]:
```
from axioma2.core import io


orig_table = io.openfile('../path/to/datadir/world.tab')
query_text = 'SELECT Страна, Население FROM world WHERE Население > 100000000'
query_table = io.query(query_text, orig_table)

# Посмотрим результат запроса
for f in query_table.items():
    print(f['Страна'])
```

```
Бангладеш
Бразилия
Китай
Индия
Индонезия
Япония
Российская Федерация
Соединенные Штаты Америки
```

Агрегирующие запросы

Также поддерживаются агрегирующие запросы. Например, для запроса выше можно получить количество стран, население которых больше 100 миллионов.

In [34]:
```
query_table.count()
```

Out[34]: 8

Но если нас интересует только количество, то можно явно составить такой запрос:

In [35]:
```
query_text = 'SELECT count(*) as result FROM world WHERE Население > 10000000
0'
query_table = io.query(query_text, orig_table)
# прочитаем результат
feature = next(query_table.items())
feature['result']
```

Out[35]: 8

Результат запроса будет содержаться в результирующей таблице в первой записи. Для удобства чтения результатов из первой строки есть вспомогательная функция `query_aggregate`. Посмотрим на её использование (эквивалентно примеру выше):

In [36]:
```
query_text = 'SELECT count(*) FROM world WHERE Население > 100000000'
count, = io.query_aggregate(query_text, orig_table)
print(count)
```

```
8
```

Функция возвращает кортеж (tuple). Так если запрос содержит несколько агрегирующих функций или других значений, то результаты легко присвоить к разным переменным.

In [37]:
```
query_text = 'SELECT MIN(Население), MAX(Население) FROM world'
min_pop, max_pop = io.query_aggregate(query_text, orig_table)
print(f'Минимальное значение = {min_pop}')
print(f'Максимальное значение = {max_pop}')
```

```
Минимальное значение = 0
Максимальное значение = 1130510638
```

Геометрия

Модуль работы с геометрией `axioma2.core.geometry`

Типы

Типы доступны в перечислении `aximoa2.core.geometry.Type` . Основные доступные типы геометрий:

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- MultiGeometry

Расширенные типы (не поддерживаются многими провайдерами):

- Arc
- Ellipse
- Rectange
- RoundedRectange
- Text

Свойства

- `area: float`
- `length: float`
- `bounds: QRectF`
- `type: Type`
- `name: str`

Продемонстрируем их использование:

In [38]:
```
wkt = 'POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))'
polygon = Geometry.from_wkt(wkt)
print(polygon.type)
print(polygon.wkt)
print(f'Название: {polygon.name}')
print(f'Площадь: {polygon.area}')
print(f'Периметр: {polygon.length}')
```

```
axioma2.cpp_core_geometry.Type.Polygon
POLYGON ((0 0, 0 10, 10 10, 10 0, 0 0))
Название: Полигон
Площадь: 100.0
Периметр: 40.0
```

Функции сериализации в `WKT` :

- Geometry.wkt -> str
- Geometry.from_wkt(str) -> Geometry

Преобразования

Функции для перепроецирования и афинного преобразования:

- reproject(crs)
- affine_transform(QTransform)

In [39]:
```
from PySide2.QtGui import QTransform


point = Geometry.from_wkt('POINT (10 10)')
# сдвиг на 10 по оси X и на -10 по оси Y
translate = QTransform.fromTranslate(10, -10)
new_point = point.affine_transform(translate)
new_point.wkt
```

Out[39]:
```
'POINT (20 0)'
```

Пространственные отношения

Предикаты

Именованные отношения, возвращающие `True` / `False` :

- equals(Geometry)
- almost_equals(Geometry, tolerance: float)
- contains(Geometry)
- crosses(Geometry)
- disjoint(Geometry)
- intersects(Geometry)
- overlaps(Geometry)
- touches(Geometry)
- within(Geometry)
- covers(Geometry)

In [40]:
```
line = Geometry.from_wkt('LINESTRING (0 0, 1 1)')
point = Geometry.from_wkt('POINT (0.5 0.5)')
line.contains(point)
```

Out[40]:
```
True
```

Отношения DE-9IM

Функция `relate()` проверяет все DE-9IM (https://en.wikipedia.org/wiki/DE-9IM) отношения между объектами. Предикаты выше являются их частными случаями.

```
In [41]: line.relate(point)
```

```
Out[41]: '0F1FF0FF2'
```

Аналитические методы

Аналитические методы, в отличии от предикатов и основных методов, возвращают новые объекты.

- boundary()
- centroid()
- difference(other)
- intersection(other)
- symmetric_difference(other)
- union(other)

Конструктивные

- buffer(distance, resolution=16, cap_style=1, join_style=1,mitre_limit=5.0)
- convex_hull()
- envelope()

```
In [42]: print(line.wkt)
         line.centroid().wkt
```

```
LINESTRING (0 0, 1 1)
```

```
Out[42]: 'POINT (0.5 0.5)'
```

```
In [43]: line.envelope().wkt
```

```
Out[43]: 'POLYGON ((0 0, 1 0, 1 1, 0 1, 0 0))'
```

Стиль

Стиль представлен соответствующим типом `Style`. Доступно создание из строки формата MapBasic. Для упрощения создания в модуле `axioma.style` есть объект `mi` для формирования строки.

Методы:

- Style.from_mapinfo() -> Style
- Style.to_mapinfo() -> str
- Style.to_ogr() -> str

Формирование строки:

- mi.pen(params) -> str
- mi.brush(params) -> str
- mi.sumbol_(params) -> str
- mi.areal(outline, fill) -> str

Создание:

- mi.create_pen(params) -> Style
- mi.create_brush(params) -> Style
- mi.create*sumbol*(params) -> Style
- mi.create_areal(outline, fill) -> Style

Пример работы со стилем в записи:

In [44]:
```
from axioma2.style import mi
from axioma2.core.dp import Feature, STYLE_ATTR

point = Geometry.from_wkt('POINT (55.4103 39.9025)')
# символ самолета, синий, размером 24
style = mi.create_symbol_compat(52, (0, 0, 255), 24)
feature = Feature(
    {'аэропорт': 'Домодедово'},
    geometry=point,
    style=style
)
STYLE_ATTR in feature
```

Out[44]:
```
True
```

In [45]:
```
feature[STYLE_ATTR].to_mapinfo()
```

Out[45]:
```
'Symbol (52, 255, 24)'
```

In [46]:
```
feature[STYLE_ATTR].to_ogr()
```

Out[46]:
```
'SYMBOL(c:#0000ff,s:24pt,id:"mapinfo-sym-35")'
```

Поменяем стиль на символ с красной звездой:

In [47]:
```
style = mi.create_symbol_compat(35, (255, 0, 0), 24)
feature[STYLE_ATTR] = style
feature[STYLE_ATTR].to_mapinfo()
```

Out[47]:
```
'Symbol (35, 16711680, 24)'
```

Отображение данных

Слой

Для отображения данных на базе таблицы или растра необходимо создать на основе этого источника данных слой

In [48]:
```
from axioma2.render import Layer

layer = Layer.create(table)
```

В зависимости от типа передаваемого объекта будет создан векторный или растровый слой.

Карта

Совокупность слоев образует карту. Порядок отрисовки слоев прямо зависит от их расположения в карте. То есть первый слой будет на самом верху, а последний - в самом низу.

Создадим карту с двумя слоями. Передадим в карту список таблиц - из них автоматически создадутся слои с параметрами по умолчанию.

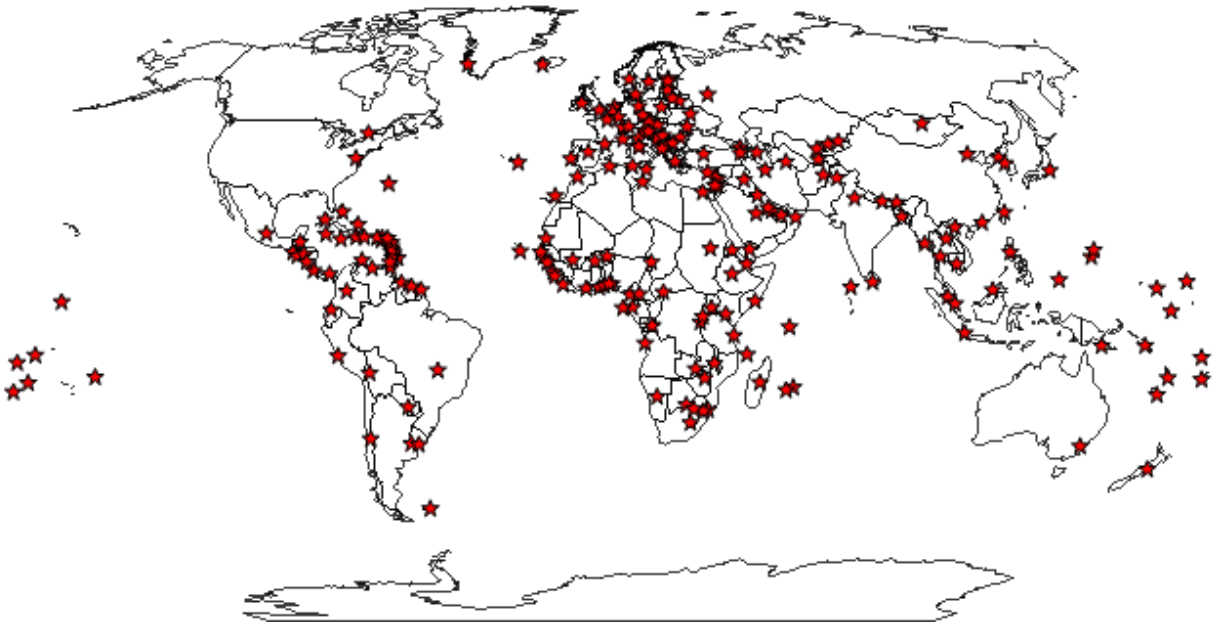
In [50]:
```
from axioma2.core import io
from axioma2.render import Map


world = io.openfile('../path/to/datadir/example.gpkg', dataobject='world')
capital = io.openfile('../path/to/datadir/worldcap.tab')
map = Map([capital, world])
```

Карту можно вывести в изображение - QImage, которое, например, можно сохранить в файл. В примере ниже мы отобразим полученную картинку в настоящем Руководстве разработчика - документе, который вы сейчас читаете.

In [51]:
```
image = map.to_image(680, 320)
# отобразим в документации
show_image(image)
```

Out[51]:



Полученную картинку можно сохранить как растр в файловой системе. Формат файла будет определяться его расширением:

In [52]:
```
image.save('../path/to/outdir/map.png')
```

Out[52]:
```
True
```

Мы указали только размеры изображения. Карта вывелась в Системе Координат (СК), наиболее подходящей для отображения слоев в ней. В нашем случае обе таблицы оказались в одной СК, которая и была выбрана для отображения.

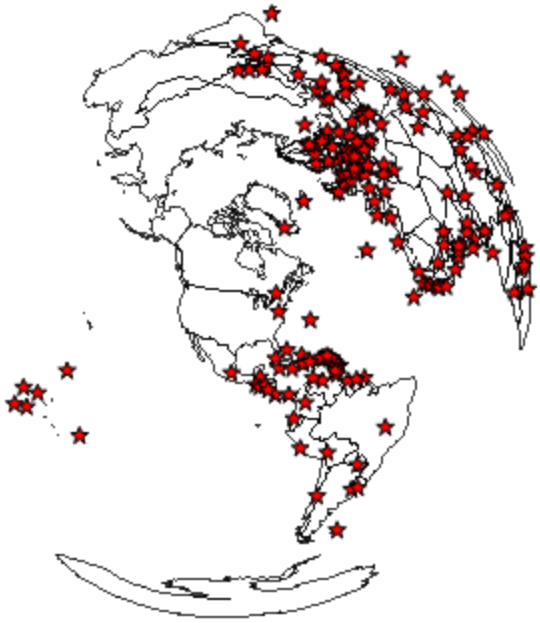
Теперь отрисуем эту же карту Азимутальной СК:

In [53]:
```
from axioma2.cs import CoordSystem

azimuth = CoordSystem.from_epsg(2163)
image = map.to_image(320, 320, coordsystem=azimuth)

show_image(image)
```

Out[53]:



Границы карты по умолчанию определились равными границам СК. Снова нарисуем нашу карту уже в Широте/Долготе в границах, примерно включающих Италию. Ограничивающий прямоугольник указываем в градусах:

In [54]:
```
longlat = CoordSystem.from_epsg(4326)
image = map.to_image(320, 320, coordsystem=longlat, bbox=(5, 35, 15, 15))

show_image(image)
```

Out[54]:

Теперь попробуем для слоя capital задать выражение для отображаемых меток, и для обоих слоев переопределить стиль оформления, сделав его однообразным. Заметим, что перечень доступных слоев карты доступен через свойство layers. Т.е. помимо передачи перечня слоев в конструктор карты, так-же возможно управление этим списком позже.

In [55]:
```
from axioma2.core.geometry import Style

lay_capital = map.layers[0]
lay_capital.label.text = 'Столица'
lay_capital.label.placementPolicy = VectorLayer.Label.DISALLOW_OVERLAP
lay_capital.overrideStyle = Style.from_mapinfo('Symbol (34,255,6)')
lay_world = map.layers['world']
lay_world.overrideStyle = Style.from_mapinfo('Pen (1, 2, 0) Brush (8, 255)')
image = map.to_image(320, 320, coordsystem=longlat, bbox=(5, 35, 15, 15))
show_image(image)
```

Out[55]:

В рамках примера по управлению слоями в конце удалим слой со столицами (самый верхний):

In [56]:
```
map.layers.remove(0)
```

Тематические слои

Так-же есть возможность для векторных слоев формирования и отрисовки тематических слоев. Т.е. применить оформление на базе атрибутивной информации. Рассмотрим на примере тематики по интервалам. Построим тематику по атрибутивному полю "Население" на 6 интервалов с равномерным распределением по количеству записей. Цвета распределим градиентом от желтого до красного.

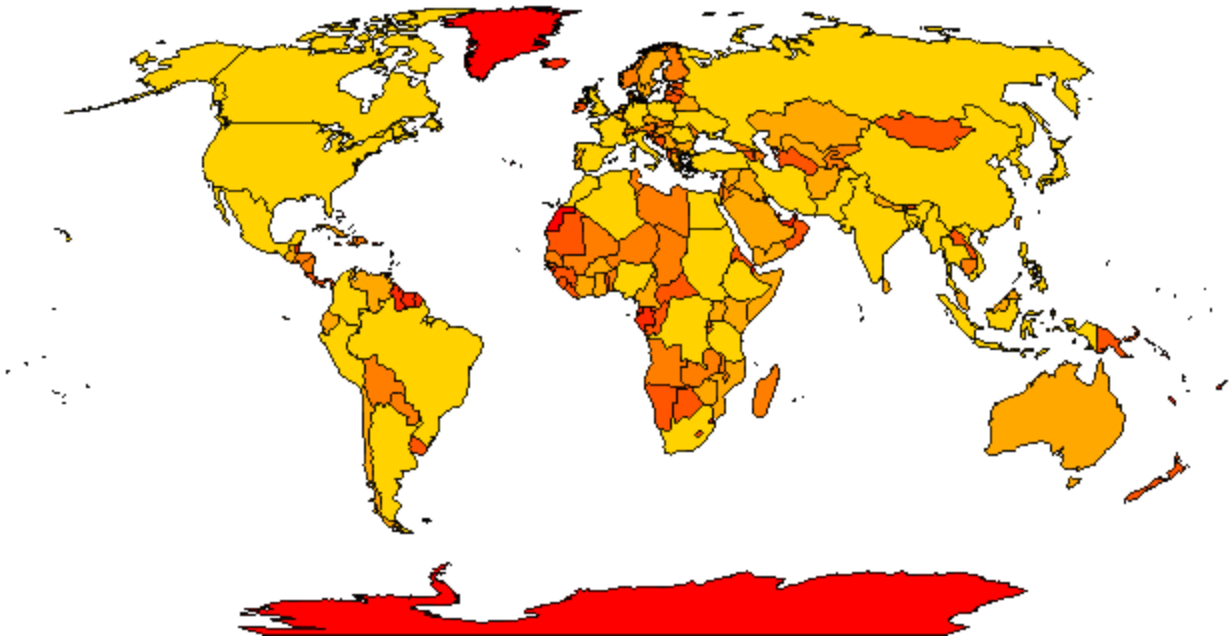
Тематические слои добавляются как дочерние к их базовому слою.

In [57]:
```
from axioma2.render import RangeThematicLayer
from PySide2.QtCore import Qt


world_layer = map.layers[0]
thematic = RangeThematicLayer('Население')
thematic.ranges = 6
thematic.colorMin = Qt.red
thematic.colorMax = Qt.yellow
thematic.splitType = RangeThematicLayer.EQUAL_COUNT
world_layer.thematic.add(thematic)

range_image = map.to_image(640, 320)
show_image(range_image)
```

Out[57]:



Легенда

Для вывода условных обозначений используется легенда. Попробуем отрисовать в растре ранее созданные слой и тематику по интервалам. Заметим, что легенду так-же можно отрисовать на одном растре вместе с картой.

In [58]:
```
from axioma2.render import Legend, Context
from PySide2.QtGui import QImage, QPainter

legend_world = Legend(lay_world)
legend_world.position = (10, 10)

legend_thematic = Legend(thematic)
legend_thematic.position = (200, 10)

image = QImage(500, 200, QImage.Format_ARGB32_Premultiplied)
image.fill(Qt.white)
painter_legend = QPainter(image)
context_legend = Context(painter_legend)

legend_world.draw(context_legend)
legend_thematic.draw(context_legend)
painter_legend.end()

show_image(image)
```

Out[58]:

Напишем простую функцию объединения изображений:

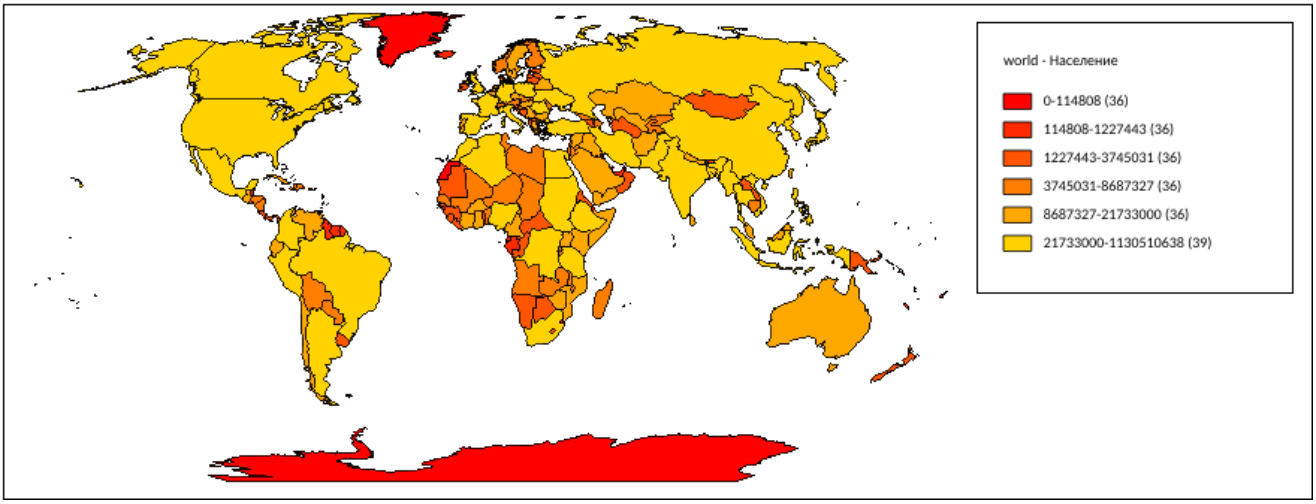
Создадим легенду, на этот раз сразу переведя в QImage, и скомпануем с тематическим слоем, полученным ранее:

In [60]:
```
from axioma2.render import Legend


legend_thematic = Legend(thematic)
legend_thematic.position = (10, 5)
legend_image = legend_thematic.to_image(200, 170)

show_image(
    hgroup(
        range_image,
        hgroup(legend_image, border=12),
        border=12
    )
)
```

Out[60]:



Отчет

Для вывода информации на печать предусмотрено создание отчетов. Отчет формируется на базе стандарного подхода работы с принтером в Qt. Создадим макет отчета, в который поместим геометрический объект и карту. Вывод сделаем в файл формата PDF. Для этого предварительно создадим объект принтера и установим необходимые свойства

In [61]:
```
from PySide2.QtPrintSupport import QPrinter

printer = QPrinter()
printer.setOutputFormat(QPrinter.PdfFormat)
printer.setOutputFileName('../path/to/outdir/report.pdf')
```

Далее, создадим сам отчет и в конструктор передадим созданный ранее принтер.

In [62]:
```
from axioma2.render import Report, GeometryReportItem, MapReportItem

report = Report(printer)
```

Создадим геометрический элемент и добавим его в отчет. Координаты в единицах измерения листа принтера.

In [63]:
```
geometryReportItem = GeometryReportItem()
geometryReportItem.geometry = Geometry.from_wkt('POLYGON ((10 10, 10 100, 100
 100, 50 50, 100 10, 10 10))')
geometryReportItem.style = Style.from_mapinfo(mi.brush(45, 255, 65535))
report.items.add(geometryReportItem)
```

Аналогично добавим карту.

In [64]:
```
from PySide2.QtCore import QRectF

mapReportItem = MapReportItem(QRectF(10, 120, 180, 100), map)
report.items.add(mapReportItem)
```

Контекст для печати по подобию рассмотренному контексту для карты.

In [65]:
```
from PySide2.QtGui import QPainter
from axioma2.render import Context


painterReport = QPainter(printer)
context = Context(painterReport)
```

Производим печать.

In [66]:
```
report.draw(context)
painterReport.end()
```

Out[66]: True

В результате в файловой системе мы получим файл report.pdf, который содержит геометрический элемент и карту.

Интерфейс

Создание кнопок

Расположение кнопки в интерфейсе Аксиома.ГИС определяется Вкладной и Группой. Например, вкладка "Основные" группа "Команды". В модуле `axioma2.gui.menubar` есть необходимые функции для создания кнопок.

In [67]:
```
from axioma2.gui import menubar


button = menubar.create_button('Простое действие', on_click=lambda: print('tri
ggered'))
position = menubar.get_position('Основные', 'Команды')
position.add(button)
```

Детально разберем, что делает этот пример.

Создается кнопка с текстом "Простое действие", и ,используя параметр `on_click` , привязывается нажатие на кнопку к анонимной лямбде, которая пичатает в консоль текст "triggered". Это обработчик нажатия кнопки. Обработчиком может служить любой callable-объект(функтор) без параметров, т.е. функции, лямбды, объекты с методом `__call__` . Также можно задать иконку кнопки параметром `icon=` . Иконкой может быть строка-ссылка на ресурс или объект типа `QIcon` .

Далее ищется расположение в интерфейсе. Если вкладка или группа с такими именами отсутствуют, то они будут созданы при добавлении кнопки.

В последней строке кнопка добавляется в заданное расположение.

Модули (плагины)

Чтобы создать модуль, руководствуйтесь следующим:

1. Придумать идею - функционал, решающий какую-то проблему.
2. Создать структуру плагина - файлы и папки.
3. Написать код.
4. Протестировать.
5. Опубликовать.

> **NOTE:** Изучайте исходный код готовых модулей

Структура модуля

Модуль для Аксиомы.ГИС это специально оформленный питоновский модуль с дополнительными файлами.

Рассмотрим структуру минимального модуля с обязательными параметрами и более расширенного:

Минимальный:

```
ru_mycompany_minimal_module  # папка с модулем
├── __init__.py  # точка входа
└── manifest.ini  # информация о модуле
```

Расширенный:

```
ru_mycompany_extended_module
├── documentation
│   └── index.html
├── business_logic.py
├── i18n
│   ├── translation_en.qm
│   └── translation_en.ts
├── __init__.py
├── manifest.ini
└── ui
    ├── form.ui
    ├── image.png
    └── logo.png
```

Идентификатор модуля

`ru_mycompany_minimal_module` - папка с модулем, она же - уникальный идентификатор модуля. Так же, как и при создании обычных питоновских модулей, избегайте конфликтов имен. Делайте имя модуля уникальным. Для этого сделуйте простому соглашению именования:

- используйте имя вашего веб-сайта или электронной почты, разделив на слова в обратном порядке;
- используйте только маленькие латинские буквы и символ нижнего подчеркивания `"_"`, т.е. `[a-z0-9_]`.

Так для модуля mymodule рекомендуемым идентификатором будет:

- для веб-сайта axioma-gis.ru - `ru_axioma_gis_mymodule`

- для почты andrey@yandex.ru - `ru_yandex_at_andrey_mymodule`

Точка входа

`__init__.py` - точка входа в модуль, так же как и для любого другого модуля Питона - является обязательным для системы импорта. Содержит основной код модуля или импортирует другие локальные файлы.

Необязательно

Может содержать класс `Plugin`. Тогда Аксиома.ГИС при загрузке модуля создаст его экземпляр, а при выгрузке - уничтожит его.

Пример модуля `__init__.py`

```
from PySide2.QtWidgets import QMessageBox


class Plugin:
    def __init__(self, iface):
        self.iface = iface
        menubar = iface.menubar
        self.__action = menubar.create_button('Пример действия',
                icon=':/icons/32px/run.png', on_click=self.show_message)
        position = menubar.get_position('Основные', 'Команды')
        position.add(self.__action)

    def unload(self):
        self.iface.menubar.remove(self.__action)

    def show_message(self):
        QMessageBox.information(None, 'Сообщение',
                'Пример выполнения действия по нажатию кнопки')
```

- `__init__` - передается доступ к объекту интерфейса Аксиомы как параметр `iface`.
- `unload` - вызывается, когда модуль выгружается.

В примере выполняется добавление кнопки и подключение действия по нажатию на нее (показ сообщения). При выгрузке кнопка удаляется из интерфейса.

Информация о модуле

`manifest.ini` - содержит основную информацию, версию, название и прочее. Является ini-файлом с простыми парами ключ=значение.

> **NOTE:** файл `manifest.ini` должен иметь кодировку UTF-8

Минимальный пример содержимого:

```
[general]
name=Пример модуля
description=Короткий текст с описанием модуля.
```

Параметры:

- `name` - короткая строка с именем модуля
- `description` - короткий текст с описанием

Необязательно

Для более подробного описания формата ini, поддерживаемого Аксиомой.ГИС, смотрите документацию configparser (https://docs.python.org/3/library/configparser.html).

Расшиненный пример содержимого:

```
; может содержать коментарии
; следующая секция обязательна
[general]
name=Пример модуля
description=Короткий текст с описанием модуля.
    Может быть многострочным.
; конец обязательных параметров

; необязательные параметры
version=1.0
author=Андрей
email=andrey@axioma-gis.ru
homepage=https://andrey.axioma-gis.ru
repository=https://github.com/andey/mymodule
license=New BSD

; секция локализации
[i18n]
name_en=Plugin example
description_en=Plugin example description.
```

Документация

Документация может быть написана в HTML файлах. Аксиома.ГИС откроет документацию в системном веб-браузере. Аксиома ищет документацию в папке с модулем documentation - файлы `index[locale].html` . Пользователь откроет документацию с суффиксом локали, совпадающим с языком системы. При отсутствии совпадения будет открываться файл без суффикса - `index.html` .

Переводы

Можно предусмотреть загрузку модуля на разных языках.

Название и описание самого модуля может быть переведено на другие языки в манифесте в секции `i18n` :

```
; секция локализации
[i18n]
name_en=Plugin example
description_en=Plugin example description.
name_fr=Exemple de plugin
description_fr=Description de l'exemple de plugin.
```

У пользователя отобразится название и описание в случае, если язык системы будет совпадать с суффиксом локали. Иначе отобразится название и описание из основной секции `general` .

Наиболее простой способ создания и сопровождения переводов строк из исходного кода - использование Qt Linguist.

> **NOTE:** Подробнее о переводе в документации Qt Linguist (https://doc.qt.io/qt-5/qtlinguist-index.html)

Основные этапы:

1. Отмечаются строки, предназначенные для перевода. Рекомендуется указывать идентификатор вашего модуля в качестве контекста перевода для избежания конфликтов имен с другими модулями.

1. Для экспорта строк используется утилита `lupdate`. Она проходит по файлам с исходным кодом и забирает все встечаемые строки, отмеченные для перевода. Результатом является файл с расширением `.ts` - простой структурированный xml файл со строками.

1. `.ts` - файл открывается в Qt Linguist и переводится на один или более языков.

1. После завершения перевода отдельных строк файл `.ts` "компилируется" в бинарный файл с расширением `.qm`, который будет загружен Аксиомой.ГИС. Для компиляции используется утилита `lrelease`.

   ```
   lrelease your_plugin.ts
   ```

1. `.qm` - файлы размещаются в подпапке `i18n` внутри модуля. Они будут загружены вместе с модулем.