sgqlc Documentation

Release 16.3

ProFUSION Sistemas e Solucoes LTDA

Jun 21, 2023

CONTENTS

This package offers an easy-to-use GraphQL client. The source code is extensively documented, so to get started, have a look at the following modules:

- Use *sgqlc.endpoint* to access GraphQL endpoints, notably *sgqlc.endpoint.http* provides `HTTPEndpoint` that makes use of `urllib.request.urlopen()`.
- To declare GraphQL schema types as Python classes, use *sgqlc.types*.
- These type classes can then be used by *sgqlc.operation* to generate and interpret GraphQL queries.
- *sgqlc.codegen* offers code generation to help using *sgqlc.types* from schema introspection results (`schema.json`) and *sgqlc.operation* using GraphQL Domain Specific Language (DSL) executable documents.
- *sgqlc.types.datetime* provides bindings for `datetime` and ISO 8601, while *sgqlc.types.relay* exposes `Node`, `PageInfo` and `Connection` types, useful for pagination.

license
ISC

CHAPTER

ONE

TABLE OF CONTENTS

1.1 *sgqlc* module

This package offers an easy-to-use GraphQL client. The source code is extensively documented, so to get started, have a look at the following modules:

- Use *sgqlc.endpoint* to access GraphQL endpoints, notably *sgqlc.endpoint.http* provides `HTTPEndpoint` that makes use of `urllib.request.urlopen()`.
- To declare GraphQL schema types as Python classes, use *sgqlc.types*.
- These type classes can then be used by *sgqlc.operation* to generate and interpret GraphQL queries.
- *sgqlc.codegen* offers code generation to help using *sgqlc.types* from schema introspection results (`schema.json`) and *sgqlc.operation* using GraphQL Domain Specific Language (DSL) executable documents.
- *sgqlc.types.datetime* provides bindings for `datetime` and ISO 8601, while *sgqlc.types.relay* exposes `Node`, `PageInfo` and `Connection` types, useful for pagination.

  license
  ISC

1.2 sgqlc-codegen Tool

1.2.1 Generate SGQLC Code

Downloading schema.json

The schema can be downloaded using *sgqlc.introspection*.

Generating Schema Types

While one can manually write the schema using *sgqlc.types*, it can be a daunting task. This can be automated if the `schema.json` is available:

```
sgqlc-codegen schema --docstrings schema.json my_schema.py
```

One may omit `--docstrings` to save space or speed up file loading (however Python's -OO/$PYTHONOPTIMIZE will drop them).

Generating Operations

If you're savvy enough to write GraphQL executable documents (aka "queries") using their Domain Specific Language (DSL), or they exist somehow, they can be used to generate *sgqlc.operation.Operation* and can be used to serialize and interpret results. The following command will use the `schema.json`, the `my_schema` (generated in the previous section) and the `my_operations.gql` with the GraphQL executable document to generate the `my_operations.py`. This file can then be imported and the functions called to create the *sgqlc.operation.Operation*.

```
sgqlc-codegen operation \
    --schema schema.json \
    my_schema \
    my_operations.py \
    my_operations.gql
```

See examples:

- GitHub defining a single parametrized (variables) query ListIssues and generates sample_operations.py.
- Shopify uses shopify_operations.gql defining all the operations, including fragments and variables, and outputs the SGQLC code. See the generated shopify_operations.py.

license
ISC

1.2.2 sgqlc-codegen Command Line Options

Generate sgqlc-based code

```
usage: sgqlc-codegen [-h] {schema,operation} ...
```

Positional Arguments

command Possible choices: schema, operation

Sub-commands

schema

Generate sgqlc types using GraphQL introspection data

```
sgqlc-codegen schema [-h] [--schema-name SCHEMA_NAME] [--docstrings]
                     [--exclude-default-types EXCLUDE_DEFAULT_TYPES [EXCLUDE_DEFAULT_
↪TYPES ...]]
                     [--add-scalar-imports ADD_SCALAR_IMPORTS [ADD_SCALAR_IMPORTS ...]]
                     [schema.json] [schema.py]
```

Positional Arguments

schema.json The input schema as JSON file. Usually the output from introspection query.

Default: <_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>

schema.py The output schema as Python file using sgqlc.types. Defaults to the input schema name with .py extension.

Named Arguments

--schema-name, -s The schema name to use. Defaults to output (or input) basename without extension and invalid python identifiers replaced with "_".

--docstrings, -d Include schema descriptions in the generated file as docstrings

Default: False

--exclude-default-types Exclude the use of sgqlc types in generated client

Default: []

--add-scalar-imports Specify "ScalarName=import.file" to automatically import "ScalarName" whenever this scalar is used in the schema

Default: []

operation

Generate sgqlc operations using GraphQL (DSL)

```
sgqlc-codegen operation [-h] [--schema SCHEMA] [--short-names]
                        schema-name [operations.py] [operation.gql ...]
```

Positional Arguments

schema-name The schema name to use in the imports. It must be in the form: *modname:symbol*. It may contain leading . to change the import statement to *from . import* using that as path. If *:symbol* is omitted, then *modname* is used.

operations.py The output operations as Python file using sgqlc.operation. Defaults to the stdout

Default: <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>

operation.gql The input GraphQL (DSL) with operations

Default: [<_io.TextIOWrapper name='<stdin>' mode='r' encoding='utf-8'>]

Named Arguments

--schema	The input schema as JSON file. Usually the output from introspection query. If given, the operations will be validated.
--short-names, -s	Use short selection names Default: False

1.3 *sgqlc.types* module

1.3.1 GraphQL Types in Python

This module fulfill two purposes:

- declare GraphQL schema in Python, just declare classes inheriting *Type*, *Interface* and fill them with *Field* (or base types: `str`, `int`, `float`, `bool`). You may as well declare *Enum* with `__choices__` or *Union* and `__types__`. Then `__str__()` will provide nice printout and `__repr__()` will return the GraphQL declarations (which can be tweaked with `__to_graphql__()`, giving indent details). `__bytes__()` is also provided, mapping to a compact `__to_graphql__()` version, without indent.
- Interpret GraphQL JSON data, by instantiating the declared classes with such information. While for scalar types it's just a pass-thru, for *Type* and *Interface* these will use the fields to provide native object with attribute or key access mapping to JSON, instead of `json_data['key']['other']` you may use `obj.key.other`. Newly declared types, such as `DateTime` will take care to generate native Python objects (ie: `datetime.datetime`). Setting such attributes will also update the backing store object, including converting back to valid JSON values.

These two improve usability of GraphQL **a lot**, pretty much like Django's Model helps to access data bases.

Field may be created explicitly, with information such as target type, arguments and GraphQL name. However, more commonly these are auto-generated by the container: GraphQL name, usually `aFieldName` will be created from Python name, usually `a_field_name`. Basic types such as `int`, `str`, `float` or `bool` will map to `Int`, `String`, `Float` and `Boolean`.

The end-user classes and functions provided by this module are:

- *Schema*: top level object that will contain all declarations. For single-schema applications, you don't have to care about this since types declared without an explicit `__schema__ = SchemaInstance` member will end in the `global_schema`.
- *Scalar*: "pass thru" everything received. Base for other scalar types:
  - *Int*: `int`
  - *Float*: `float`
  - *String*: `str`
  - *Boolean*: `bool`
  - *ID*: `str`
- *Enum*: also handled as a `str`, but GraphQL syntax needs them without the quotes, so special handling is done. Validation is done using `__choices__` member, which is either a string (which will be splitted using `str.split()`) or a list/tuple of strings with values.
- *Union*: defines the target type of a field may be one of the given `__types__`.
- Container types: *Type*, *Interface* and *Input*. These are similar in usage, but GraphQL needs them defined differently. They are composed of *Field*. A field may have arguments (*ArgDict*), which is a set of *Arg*.

Arguments may contain default values or *Variable*, which will be sent alongside the query (this allows to generate the query once and use variables, letting the server to use both together).

- *non_null()*, maps to GraphQL `Type!` and enforces the object is not `None`.
- *list_of()*, maps to GraphQL `[Type]` and enforces the object is a list of `Type`.

This module only provide built-in scalar types. However, three other modules will extend the behavior for common conventions:

- *sgqlc.types.datetime* will declare `DateTime`, `Date` and `Time`, mapping to Python's `datetime`. This also allows fields to be declared as `my_date = datetime.date`,
- *sgqlc.types.uuid* will declare `UUID`, mapping to Python's `uuid`. This also allows fields to be declared as `my_uuid = uuid.UUID`,
- *sgqlc.types.relay* will declare `Node` and `Connection`, matching Relay Global Object Identification and Cursor Connections, which are widely used.

Code Generator

If you already have `schema.json` or access to a server with introspection you may use the `sgqlc-codegen schema` to automatically generate the type definitions for you.

The generated code should be stable and can be committed to repositories, leading to minimum `diff` when updated. It may include docstrings, which improves development experience at the expense of larger files.

See examples:

- GitHub downloads the schema using introspection and generates a schema using GraphQL descriptions as Python docstrings, see the generated github_schema.py.
- Shopify downloads the schema (without descriptions) using introspection and generates a schema without Python docstrings, see the generated shopify_schema.py.

Examples

Common Usage

Common usage is to create *Type* subclasses with fields without arguments and do not use an explicit `__schema__`, resulting in the types being added to the `global_schema`. Built-in scalars can be declared using the Python classes, with *sgqlc.types* classes or with explicit *Field* instances, the *ContainerTypeMeta* takes care to make sure they are all instance of *Field* at the final class:

```
>>> class TypeUsingPython(Type):
...     a_int = int
...     a_float = float
...     a_string = str
...     a_boolean = bool
...     a_id = id
...     not_a_field = 1 # not a BaseType subclass or mapped python class
...
>>> TypeUsingPython  # or repr(TypeUsingPython), prints out GraphQL!
type TypeUsingPython {
  aInt: Int
  aFloat: Float
  aString: String
```

(continues on next page)

(continued from previous page)

```
  aBoolean: Boolean
  aId: ID
}
>>> TypeUsingPython.a_int  # or repr(Field), prints out GraphQL!
aInt: Int
>>> TypeUsingPython.a_int.name
'a_int'
>>> TypeUsingPython.a_int.graphql_name  # auto-generated from name
'aInt'
>>> TypeUsingPython.a_int.type  # always a :mod:`sgqlc.types` class
scalar Int
>>> TypeUsingPython.__schema__ is global_schema
True
>>> global_schema  # or repr(Schema), prints out GraphQL!
schema {
  scalar Int
  scalar Float
  scalar String
  scalar Boolean
  scalar ID
  type TypeUsingPython {
    aInt: Int
    aFloat: Float
    aString: String
    aBoolean: Boolean
    aId: ID
  }
}
```

You can then use some standard Python operators to check fields in a *Type*:

```
>>> 'a_float' in TypeUsingPython
True
>>> 'x' in TypeUsingPython
False
>>> for field in TypeUsingPython:  # iterates over :class:`Field`
...     print(repr(field))
...
aInt: Int
aFloat: Float
aString: String
aBoolean: Boolean
aId: ID
```

As mentioned, fields can be created with basic Python types (simpler), with *sgqlc.types* or with *Field* directly:

```
>>> class TypeUsingSGQLC(Type):
...     a_int = Int
...     a_float = Float
...     a_string = String
...     a_boolean = Boolean
...     a_id = ID
```

(continues on next page)

(continued from previous page)

```
...
>>> TypeUsingSGQLC  # or repr(TypeUsingSGQLC), prints out GraphQL!
type TypeUsingSGQLC {
  aInt: Int
  aFloat: Float
  aString: String
  aBoolean: Boolean
  aId: ID
}
```

Using `Field` instances

Allows for greater control, such as explicitly define the `graphql_name` instead of generating one from the Python `name`. It is also used to declare field arguments:

```
>>> class TypeUsingFields(Type):
...     a_int = Field(int)         # can use Python classes
...     a_float = Field(float)
...     a_string = Field(String)   # or sgqlc.types classes
...     a_boolean = Field(Boolean)
...     a_id = Field(ID, graphql_name='anotherName') # allows customizations
...     pow = Field(int, args={'base': int, 'exp': int}) # with arguments
...     # more than 3 arguments renders each into new line
...     many = Field(int, args={'a': int, 'b': int, 'c': int, 'd': int})
...
>>> TypeUsingFields  # or repr(TypeUsingFields), prints out GraphQL!
type TypeUsingFields {
  aInt: Int
  aFloat: Float
  aString: String
  aBoolean: Boolean
  anotherName: ID
  pow(base: Int, exp: Int): Int
  many(
    a: Int
    b: Int
    c: Int
    d: Int
  ): Int
}
```

Adding types to specific `Schema`

Create a schema instance and assign as `__schema__` class member. Note that previously defined types in `base_schema` are inherited, and by default `global_schema` is used as base schema:

```
>>> my_schema = Schema(global_schema)
>>> class MySchemaType(Type):
...     __schema__ = my_schema
...     i = int
...
>>> class MyOtherType(Type):
...     i = int
...
>>> 'TypeUsingPython' in my_schema
True
>>> 'MySchemaType' in global_schema
False
>>> 'MySchemaType' in my_schema
True
>>> 'MyOtherType' in global_schema
True
>>> 'MyOtherType' in my_schema  # added after my_schema was created!
False
>>> my_schema.MySchemaType  # access types as schema attributes
type MySchemaType {
  i: Int
}
>>> my_schema['MySchemaType']  # access types as schema items
type MySchemaType {
  i: Int
}
>>> for t in my_schema:
...     print(repr(t))
...
scalar Int
scalar Float
...
type MySchemaType {
  i: Int
}
```

Inheritance and Interfaces

Inheriting another type inherits all fields:

```
>>> class MySubclass(TypeUsingPython):
...     sub_field = int
...
>>> MySubclass
type MySubclass {
  aInt: Int
  aFloat: Float
```

(continues on next page)

(continued from previous page)

```
  aString: String
  aBoolean: Boolean
  aId: ID
  subField: Int
}
```

Interfaces are similar, however they emit `implements IfaceName`:

```
>>> class MyIface(Interface):
...     sub_field = int
...
>>> MyIface
interface MyIface {
  subField: Int
}
>>> class MySubclassWithIface(TypeUsingPython, MyIface):
...     pass
...
>>> MySubclassWithIface
type MySubclassWithIface implements MyIface {
  aInt: Int
  aFloat: Float
  aString: String
  aBoolean: Boolean
  aId: ID
  subField: Int
}
```

Although usually types are declared first, they can be declared after interfaces as well. Note order of fields respect inheritance order:

```
>>> class MySubclassWithIface2(MyIface, TypeUsingPython):
...     pass
...
>>> MySubclassWithIface2
type MySubclassWithIface2 implements MyIface {
  subField: Int
  aInt: Int
  aFloat: Float
  aString: String
  aBoolean: Boolean
  aId: ID
}
```

Cross References (Loops)

If types link to themselves, declare as strings so they are lazy-evaluated:

```
>>> class LinkToItself(Type):
...     other = Field('LinkToItself')
...     non_null_other = non_null('LinkToItself')
...     list_other = list_of('LinkToItself')
...     list_other_non_null = list_of(non_null('LinkToItself'))
...     non_null_list_other_non_null = non_null(list_of(non_null(
...         'LinkToItself')))
...
>>> LinkToItself
type LinkToItself {
  other: LinkToItself
  nonNullOther: LinkToItself!
  listOther: [LinkToItself]
  listOtherNonNull: [LinkToItself!]
  nonNullListOtherNonNull: [LinkToItself!]!
}
>>> LinkToItself.other.type
type LinkToItself {
  other: LinkToItself
  nonNullOther: LinkToItself!
  listOther: [LinkToItself]
  listOtherNonNull: [LinkToItself!]
  nonNullListOtherNonNull: [LinkToItself!]!
}
```

Also works for two unrelated types:

```
>>> class CrossLinkA(Type):
...     other = Field('CrossLinkB')
...     non_null_other = non_null('CrossLinkB')
...     list_other = list_of('CrossLinkB')
...     list_other_non_null = list_of(non_null('CrossLinkB'))
...     non_null_list_other_non_null = non_null(list_of(non_null(
...         'CrossLinkB')))
...
>>> class CrossLinkB(Type):
...     other = Field('CrossLinkA')
...     non_null_other = non_null('CrossLinkA')
...     list_other = list_of('CrossLinkA')
...     list_other_non_null = list_of(non_null('CrossLinkA'))
...     non_null_list_other_non_null = non_null(list_of(non_null(
...         'CrossLinkA')))
...
>>> CrossLinkA
type CrossLinkA {
  other: CrossLinkB
  nonNullOther: CrossLinkB!
  listOther: [CrossLinkB]
  listOtherNonNull: [CrossLinkB!]
```

(continues on next page)

(continued from previous page)

```
  nonNullListOtherNonNull: [CrossLinkB!]!
}
>>> CrossLinkB
type CrossLinkB {
  other: CrossLinkA
  nonNullOther: CrossLinkA!
  listOther: [CrossLinkA]
  listOtherNonNull: [CrossLinkA!]
  nonNullListOtherNonNull: [CrossLinkA!]!
}
>>> CrossLinkA.other.type
type CrossLinkB {
  other: CrossLinkA
  nonNullOther: CrossLinkA!
  listOther: [CrossLinkA]
  listOtherNonNull: [CrossLinkA!]
  nonNullListOtherNonNull: [CrossLinkA!]!
}
>>> CrossLinkB.other.type
type CrossLinkA {
  other: CrossLinkB
  nonNullOther: CrossLinkB!
  listOther: [CrossLinkB]
  listOtherNonNull: [CrossLinkB!]
  nonNullListOtherNonNull: [CrossLinkB!]!
}
```

Special Attribute Names

Attributes starting with _ are ignored, however if for some reason you must use such attribute name, then declare **ALL** attributes in that class (no need to repeat inherited attributes from interfaces) using the `__field_names__`, which should have a tuple of strings:

```
>>> class TypeUsingSpecialAttributes(Type):
...     __field_names__ = ('_int', '_two_words')
...     _int = int
...     _two_words = str
...     not_handled = float # not declared!
...
>>> TypeUsingSpecialAttributes  # or repr(TypeUsingSpecialAttributes)
type TypeUsingSpecialAttributes {
  _int: Int
  _twoWords: String
}
>>> TypeUsingSpecialAttributes._int  # or repr(Field), prints out GraphQL!
_int: Int
>>> TypeUsingSpecialAttributes._int.name
'_int'
>>> TypeUsingSpecialAttributes._int.graphql_name  # auto-generated from name
'_int'
```

Note that while the leading underscores (_) are preserved, the rest of internal underscores are converted to camel case:

```
>>> TypeUsingSpecialAttributes._two_words
_twoWords: String
>>> TypeUsingSpecialAttributes._two_words.name
'_two_words'
>>> TypeUsingSpecialAttributes._two_words.graphql_name
'_twoWords'
```

> **Note:** Take care with the double underscores __ as Python mangles the name with the class name in order to "protect" and it will result in `AttributeError`

Note that undeclared fields won't be handled, but they still exist as regular python attributes, in this case it references the `float` class:

```
>>> TypeUsingSpecialAttributes.not_handled
<class 'float'>
```

Non GraphQL Attributes

The `__field_names__` may also be used to allow non-GraphQL attributes that would otherwise be handled as such, this explicitly limits the scope where SGQLC will handle.

Utilities

One can obtain fields as container attributes or items:

```
>>> TypeUsingPython.a_int
aInt: Int
>>> TypeUsingPython['a_int']
aInt: Int
```

However they raise exceptions if doesn't exist:

```
>>> TypeUsingPython.does_not_exist
Traceback (most recent call last):
  ...
AttributeError: TypeUsingPython has no field does_not_exist
>>> TypeUsingPython['does_not_exist']
Traceback (most recent call last):
  ...
KeyError: 'TypeUsingPython has no field does_not_exist'
```

Fields show in `dir()` alongside with non-fields (sorted):

```
>>> for name in dir(TypeUsingPython):
...     if not name.startswith('_'):
...         print(name)
a_boolean
a_float
```

(continues on next page)

(continued from previous page)

```
a_id
a_int
a_string
not_a_field
```

Unless *non_null()* is used, containers can be created for `None`:

```
>>> TypeUsingPython(None)
TypeUsingPython()
```

```
>>> TypeUsingPython.__to_json_value__(None) # returns None
```

For instances, field values can be obtained or set attributes or items, when setting a known field, it also updates the backing store:

```
>>> json_data = {'aInt': 1}
>>> obj = TypeUsingPython(json_data)
>>> obj.a_int
1
>>> obj['a_int']
1
>>> obj.a_int = 2
>>> json_data['aInt']
2
>>> obj['a_int'] = 3
>>> json_data['aInt']
3
>>> obj['a_float'] = 2.1 # known field!
>>> json_data['aFloat']
2.1
>>> obj.a_float = 3.3 # known field!
>>> json_data['aFloat']
3.3
```

Unknown fields raise exceptions when obtained, but are allowed to be set, however doesn't update the backing store:

```
>>> obj.does_not_exist
Traceback (most recent call last):
  ...
AttributeError: 'TypeUsingPython' object has no attribute 'does_not_exist'
>>> obj['does_not_exist']
Traceback (most recent call last):
  ...
KeyError: 'TypeUsingPython(a_int=3, a_float=3.3) has no field does_not_exist'
>>> obj['does_not_exist'] = 'abc' # unknown field, no updates to json_data
>>> json_data['does_not_exist']
Traceback (most recent call last):
  ...
KeyError: 'does_not_exist'
```

While `repr()` prints out summary in Python-friendly syntax, `bytes()` can be used to get compressed JSON with sorted keys:

```
>>> print(repr(obj))
TypeUsingPython(a_int=3, a_float=3.3)
>>> print(bytes(obj).decode('utf-8'))
{"aFloat":3.3,"aInt":3}
```

license
ISC

class sgqlc.types.**Arg**(*typ*, *graphql_name=None*, *default=None*)

Bases: *BaseItem*

GraphQL *Field* argument.

```
>>> class MyTypeWithArgument(Type):
...     a = Field(str, args={'arg_name': int}) # implicit
...     b = Field(str, args={'arg': Arg(int)})  # explicit + Python
...     c = Field(str, args={'arg': Arg(Int)})  # explicit + sgqlc.types
...     d = Field(str, args={'arg': Arg(int, default=1)})
...
>>> MyTypeWithArgument
type MyTypeWithArgument {
  a(argName: Int): String
  b(arg: Int): String
  c(arg: Int): String
  d(arg: Int = 1): String
}
```

__init__(*typ*, *graphql_name=None*, *default=None*)

Parameters

- **typ** (*Scalar*, *Type* or str) – the *Scalar* or *Type* derived class. If this would cause a cross reference and the other type is not declared yet, then use the string name to query in the schema.
- **graphql_name** (*str*) – the name to use in JSON object, usually `aName`. If `None` or empty, will be created from python, converting `a_name` to `aName` using *BaseItem._to_graphql_name()*
- **default** – The default value for field. May be a value or *Variable*.

class sgqlc.types.**ArgDict**(**lst*, ***mapping*)

Bases: `OrderedDict`

The *Field* Argument Dict.

Common usage is inside *Field*:

```
>>> class MyType(Type):
...     a = Field(Int, args={'argument1': String})     # implicit
...     b = Field(Int, args=ArgDict(argument1=String)) # explicit
...
>>> print(repr(MyType))
type MyType {
  a(argument1: String): Int
  b(argument1: String): Int
}
```

(continues on next page)

(continued from previous page)

```
>>> print(repr(MyType.a))
a(argument1: String): Int
>>> print(repr(MyType.a.args))
(argument1: String)
>>> print(repr(MyType.b))
b(argument1: String): Int
>>> print(repr(MyType.b.args))
(argument1: String)
>>> print(repr(MyType.b.args['argument1']))
argument1: String
>>> print(bytes(MyType.b.args['argument1']).decode('utf-8'))
argument1: String
```

This takes care to ensure values are *Arg*. In the example above, we're not passing *Arg*, rather just a type (*String*) and it's working internally to create *Arg*. For ease of use, can be created in various forms. Note they must be added to a container field to be useful, which would call `ArgDict._set_container()` for you, here called manually for testing purposes:

```
>>> ad = ArgDict(name=str)
>>> ad._set_container(global_schema, None)  # done automatically by Field
>>> print(ad)
(name: String)
```

```
>>> ad = ArgDict(name=String)
>>> ad._set_container(global_schema, None)  # done automatically by Field
>>> print(ad)
(name: String)
```

```
>>> ad = ArgDict({'name': str})
>>> ad._set_container(global_schema, None)  # done automatically by Field
>>> print(ad)
(name: String)
```

```
>>> ad = ArgDict(('name', str), ('other', int))
>>> ad._set_container(global_schema, None)  # done automatically by Field
>>> print(ad)
(name: String, other: Int)
```

```
>>> ad = ArgDict((('name', str), ('other', int)))
>>> ad._set_container(global_schema, None)  # done automatically by Field
>>> print(ad)
(name: String, other: Int)
```

Note that for better understanding, more than 3 arguments are printed in multiple lines:

```
>>> ad = ArgDict(a=int, b=float, c=non_null(str), d=list_of(int))
>>> ad._set_container(global_schema, None)  # done automatically by Field
>>> print(ad)
(
  a: Int
  b: Float
```

(continues on next page)

(continued from previous page)

```
  c: String!
  d: [Int]
)
>>> print(bytes(ad).decode('utf-8'))
(
a: Int
b: Float
c: String!
d: [Int]
)
```

This is also the case for input values:

```
>>> print('fieldName' + ad.__to_graphql_input__({
...     'a': 1, 'b': 2.2, 'c': 'hi', 'd': [1, 2],
... }))
fieldName(
    a: 1
    b: 2.2
    c: "hi"
    d: [1, 2]
  )
```

Variables can be handled using *Variable* instances:

```
>>> print('fieldName' + ad.__to_graphql_input__({
...     'a': Variable('a'),
...     'b': Variable('b'),
...     'c': Variable('c'),
...     'd': Variable('d'),
... }))
fieldName(
    a: $a
    b: $b
    c: $c
    d: $d
  )
```

__init__(**lst*, ***mapping*)

__repr__()

Return repr(self).

__str__()

Return str(self).

class sgqlc.types.**BaseItem**(*typ*, *graphql_name=None*)

Bases: object

Base item for *Arg* and *Field*.

Each parameter has a GraphQL type, such as a derived class from *Scalar* or *Type*, this is used for nesting, conversion to native Python types, generating queries, etc.

__init__(*typ*, *graphql_name=None*)

Parameters

- **typ** (*Scalar*, *Type* or str) – the *Scalar* or *Type* derived class. If this would cause a cross reference and the other type is not declared yet, then use the string name to query in the schema.
- **graphql_name** (*str*) – the name to use in JSON object, usually `aName`. If `None` or empty, will be created from python, converting `a_name` to `aName` using `Arg._to_graphql_name()`

__repr__()

Return repr(self).

__str__()

Return str(self).

classmethod _to_graphql_name(*name*)

Converts a Python name, `a_name` to GraphQL: `aName`.

Note that leading underscores (_) are preserved.

```
>>> BaseItem._to_graphql_name('a_name')
'aName'
>>> BaseItem._to_graphql_name('__underscore_prefixed')
'__underscorePrefixed'
>>> BaseItem._to_graphql_name('__typename__')
'__typename'
```

classmethod _to_python_name(*graphql_name*)

Converts a GraphQL name, `aName` to Python: `a_name`.

Note that an underscore is appended if the name is a Python keyword.

```
>>> BaseItem._to_python_name('aName')
'a_name'
>>> BaseItem._to_python_name('for')
'for_'
>>> BaseItem._to_python_name('__typename')
'__typename__'
```

class sgqlc.types.**BaseMeta**(*name*, *bases*, *namespace*)

Bases: `type`

Automatically adds class to its schema

__ensure__(*t*)

Checks if `t` is subclass of `BaseType` or if a mapping is known.

```
>>> BaseType.__ensure__(Int)
scalar Int
>>> BaseType.__ensure__(int)
scalar Int
>>> BaseType.__ensure__(bytes)
Traceback (most recent call last):
  ...
TypeError: Not BaseType or mapped: <class 'bytes'>
```

__init__(*name*, *bases*, *namespace*)

__repr__()

Return repr(self).

__str__()

Return str(self).

class sgqlc.types.**BaseType**

Bases: object

Base shared by all GraphQL classes.

__weakref__

list of weak references to the object (if defined)

class sgqlc.types.**Boolean**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Maps GraphQL `Boolean` to Python `bool`.

converter

alias of bool

class sgqlc.types.**ContainerType**(*json_data*, *selection_list=None*)

Bases: `BaseTypeWithTypename`

Container of *Field*.

For ease of use, fields can be declared by sub classes in the following ways:

- `name = str` to create a simple string field. Other basic types are allowed as well: `int`, `float`, `str`, `bool`, `uuid.UUID`, `datetime.time`, `datetime.date` and `datetime.datetime`. These are only used as identifiers to translate using `map_python_to_graphql` dict. Note that `id`, although is not a type, maps to `ID`.
- `name = TypeName` for subclasses of `BaseType`, such as pre-defined scalars (*Int*, etc) or your own defined types, from *Type*.
- `name = Field(TypeName, graphql_name='differentName', args={...})` to explicitly define more field information, such as GraphQL JSON name, query parameters, etc.

The metaclass *ContainerTypeMeta* will normalize all of those members to be instances of *Field*, as well as provide useful container protocol such as `__contains__`, `__getitem__`, `__iter__` and so on.

Fields from all bases (interfaces, etc) are merged.

Members started with underscore (_) are not processed.

__contains__(*name*)

Checks if for a known field name in the **instance**.

Unlike `name in SubclassOfType`, which checks amongst all declared fields, this matches only fields that exist in the object, based on the `json_data` used to create the object, and the one that provides the backing store:

```
>>> json_data = { 'aInt': 1, 'aFloat': 2.1 }
>>> obj = global_schema.TypeUsingPython(json_data)
>>> 'a_int' in obj
True
>>> 'a_float' in obj
```

(continues on next page)

(continued from previous page)

```
True
>>> 'a_string' in obj  # in class, but not instance
False
>>> 'a_string' in obj.__class__
True
```

After it's set for the given instance, then becomes true:

```
>>> obj.a_string = 'hello world'  # known field
>>> 'a_string' in obj  # now in instance
True
```

__getitem__(*name*)

Get the field given its name.

Considering `TypeUsingPython`, previously declared in the module documentation:

```
>>> global_schema.TypeUsingPython['a_int']
aInt: Int
```

```
>>> global_schema.TypeUsingPython['unknown_field']
Traceback (most recent call last):
  ...
KeyError: 'TypeUsingPython has no field unknown_field'
```

__init__(*json_data*, *selection_list=None*)

__iter__()

Iterate over known fields of the **instance**.

Unlike `iter(SubclassOfType)`, which iterates over all declared fields, this iterator matches only fields that exist in the object, based on the `json_data` used to create the object, and the one that provides the backing store:

```
>>> json_data = { 'aInt': 1, 'aFloat': 2.1 }
>>> obj = global_schema.TypeUsingPython(json_data)
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
a_int 1
a_float 2.1
>>> for field in obj.__class__:
...     print(repr(field))
aInt: Int
aFloat: Float
aString: String
aBoolean: Boolean
aId: ID
```

After it's set for the given instance, then it's included in the iterator:

```
>>> obj.a_string = 'hello world'  # known field
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
```

(continues on next page)

(continued from previous page)

```
a_int 1
a_float 2.1
a_string 'hello world'
```

However that's valid for known *Field* for the given *ContainerType* subclasses:

```
>>> obj.new_attr = 'some value'  # unknown field, not in 'iter'
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
a_int 1
a_float 2.1
a_string 'hello world'
```

__len__()

Checks how many fields are set in the **instance**.

```
>>> json_data = { 'aInt': 1, 'aFloat': 2.1 }
>>> obj = global_schema.TypeUsingPython(json_data)
>>> len(obj)
2
```

```
>>> obj.a_string = 'hello world'  # known field
>>> len(obj)
3
```

__repr__()

Return repr(self).

__setattr__(*name*, *value*)

Sets the attribute value, if a *Field* updates backing store.

Considering TypeUsingPython, previously declared in the module documentation:

```
>>> json_data = {'aInt': 1,  'aFloat': 2.1}
>>> obj = global_schema.TypeUsingPython(json_data)
>>> obj.a_int, obj.a_float
(1, 2.1)
>>> obj.a_int = 123
>>> obj.a_int, obj.a_float
(123, 2.1)
>>> json_data['aInt']
123
```

However that's valid for known *Field* for the given *ContainerType* subclasses:

```
>>> obj.new_attr = 'some value'  # no field, no backing store updates
>>> obj.new_attr
'some value'
>>> json_data['new_attr']
Traceback (most recent call last):
  ...
KeyError: 'new_attr'
>>> json_data['newAttr']
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
  ...
KeyError: 'newAttr'
```

__setitem__(*name*, *value*)

Set the item, maps to `setattr(self, name, value)`

__str__()

Return str(self).

class sgqlc.types.**ContainerTypeMeta**(*name*, *bases*, *namespace*)

Bases: `BaseMetaWithTypename`

Creates container types, ensures fields are instance of Field.

__dir__()

Specialized __dir__ implementation for types.

__init__(*name*, *bases*, *namespace*)

class sgqlc.types.**Enum**(*json_data*, *selection_list=None*)

Bases: *BaseType*

This is an abstract class that enumerations should inherit and define `__choices__` class member with a list of strings matching the choices allowed by this enumeration. A single string may also be used, in such case it will be split using `str.split()`.

Note that `__choices__` is not set in the final class, the metaclass will use that to build members and provide the `__iter__`, `__contains__` and `__len__` instead.

The instance constructor will never return instance of *Enum*, rather the string, if that matches.

Examples:

```
>>> class Colors(Enum):
...     __choices__ = ('RED', 'GREEN', 'BLUE')
...
>>> Colors('RED')
'RED'
>>> Colors(None) # returns None
>>> Colors('MAGENTA')
Traceback (most recent call last):
  ...
ValueError: Colors does not accept value MAGENTA
```

Using a string will automatically split and convert to tuple:

```
>>> class Fruits(Enum):
...     __choices__ = 'APPLE ORANGE BANANA'
...
>>> Fruits.__choices__
('APPLE', 'ORANGE', 'BANANA')
>>> len(Fruits)
3
```

Enumerations have a special syntax in GraphQL, no quotes:

```
>>> print(Fruits.__to_graphql_input__(Fruits.APPLE))
APPLE
>>> print(Fruits.__to_graphql_input__(None))
null
```

And for JSON it's a string as well (so JSON encoder adds quotes):

```
>>> print(json.dumps(Fruits.__to_json_value__(Fruits.APPLE)))
"APPLE"
```

Variables are passed thru:

```
>>> Fruits(Variable('var'))
$var
```

static __new__(*json_data*, *selection_list=None*)

class sgqlc.types.**Field**(*typ*, *graphql_name=None*, *args=None*)

Bases: *BaseItem*

Field in a *Type* container.

Each field has a GraphQL type, such as a derived class from *Scalar* or *Type*, this is used for nesting, conversion to native Python types, generating queries, etc.

__bytes__()

Prints GraphQL without indentation.

```
>>> print(repr(global_schema.TypeUsingFields.many))
many(
    a: Int
    b: Int
    c: Int
    d: Int
  ): Int
>>> print(bytes(global_schema.TypeUsingFields.many).decode('utf-8'))
many(
a: Int
b: Int
c: Int
d: Int
): Int
```

__init__(*typ*, *graphql_name=None*, *args=None*)

Parameters

- **typ** (*Scalar*, *Type* or str) – the *Scalar* or *Type* derived class. If this would cause a cross reference and the other type is not declared yet, then use the string name to query in the schema.
- **graphql_name** (*str*) – the name to use in JSON object, usually `aName`. If `None` or empty, will be created from python, converting `a_name` to `aName` using *BaseItem._to_graphql_name()*

- **args** (*ArgDict*) – The field parameters as a *ArgDict* or compatible type (dict, or iterable of key-value pairs). The value may be a mapped Python type (ie: `str`), explicit type (ie: `String`), type name (ie: `"String"`, to allow cross references) or *Arg* instances.

class sgqlc.types.**Float**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Maps GraphQL `Float` to Python `float`.

converter

alias of `float`

class sgqlc.types.**ID**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Maps GraphQL `ID` to Python `str`.

converter

alias of `str`

class sgqlc.types.**Input**(**args*, ***kwargs*)

Bases: *ContainerType*

GraphQL `input Name`.

Input types are similar to *Type*, but they are used as argument values. They have more restrictions, such as no *Interface*, *Union* or *Type* are allowed as field types. Only scalars or *Input*.

Note: SGQLC currently doesn't enforce the field type restrictions imposed by the server.

```
>>> class MyInput(Input):
...     a_int = int
...     a_float = float
...
>>> MyInput
input MyInput {
  aInt: Int
  aFloat: Float
}
>>> print(MyInput.__to_graphql_input__({'a_int': 1, 'a_float': 2.2}))
{aInt: 1, aFloat: 2.2}
```

```
>>> a_var = Variable('input')
>>> print(MyInput.__to_graphql_input__(a_var))
$input
```

__init__(*_json_obj=None*, *_selection_list=None*, ***kwargs*)

Create the type given a json object or keyword arguments.

```
>>> class AnotherInput(Input):
...     a_str = str
...
>>> class TheInput(Input):
...     a_int = int
...     a_float = float
```

(continues on next page)

(continued from previous page)

```
...     a_nested = AnotherInput
...     a_nested_list = list_of(AnotherInput)
...
```

It can be constructed using fields and values as a regular Python class:

```
>>> TheInput(a_int=1, a_float=1.2, a_nested=AnotherInput(a_str='hi'))
TheInput(a_int=1, a_float=1.2, a_nested=AnotherInput(a_str='hi'))
```

Or can be given as a dict (ie: JSON data) as parameter:

```
>>> TheInput({'aInt': 1, 'aFloat': 1.2, 'aNested': {'aStr': 'hi'}})
TheInput(a_int=1, a_float=1.2, a_nested=AnotherInput(a_str='hi'))
```

It can be printed to GraphQL DSL using `__to_graphql_input__()`:

```
>>> value = TheInput(a_int=1, a_float=1.2,
...                  a_nested=AnotherInput(a_str='hi'),
...                  a_nested_list=[AnotherInput(a_str='there')])
>>> print(TheInput.__to_graphql_input__(value))
{aInt: 1, aFloat: 1.2, aNested: {aStr: "hi"}, aNestedList: [{aStr: "there"}]}
>>> value = TheInput({'aInt': 1, 'aFloat': 1.2, 'aNested': {'aStr': 'hi'},
...           'aNestedList': [{'aStr': 'there'}]})
>>> print(TheInput.__to_graphql_input__(value))
{aInt: 1, aFloat: 1.2, aNested: {aStr: "hi"}, aNestedList: [{aStr: "there"}]}
```

The nested types (lists, non-null) can also take an already realized value, see `AnotherInput` below:

```
>>> value = TheInput({'aInt': 1, 'aFloat': 1.2, 'aNested': {'aStr': 'hi'},
...           'aNestedList': [AnotherInput(a_str='there')]})
>>> print(TheInput.__to_graphql_input__(value))
{aInt: 1, aFloat: 1.2, aNested: {aStr: "hi"}, aNestedList: [{aStr: "there"}]}
```

Input types can be constructed from variables. Note that the input variable can't be an element of a list, the list itself must be a variable:

```
>>> a_var = Variable('input')
>>> value = TheInput(a_var)
>>> print(TheInput.__to_graphql_input__(value))
$input
>>> value = TheInput(a_int=1, a_float=1.2,
...                  a_nested=a_var,
...                  a_nested_list=[AnotherInput(a_str='there')])
>>> print(TheInput.__to_graphql_input__(value))
{aInt: 1, aFloat: 1.2, aNested: $input, aNestedList: [{aStr: "there"}]}
>>> value = TheInput(a_int=1, a_float=1.2,
...                  a_nested=AnotherInput(a_str='hi'),
...                  a_nested_list=a_var)
>>> print(TheInput.__to_graphql_input__(value))
{aInt: 1, aFloat: 1.2, aNested: {aStr: "hi"}, aNestedList: $input}
```

`None` will print `null`:

```
>>> print(TheInput.__to_graphql_input__(None))
null
```

```
>>> value = TheInput(a_int=None, a_float=None, a_nested=None,
...                  a_nested_list=None)
>>> print(TheInput.__to_graphql_input__(value))
{aInt: null, aFloat: null, aNested: null, aNestedList: null}
```

Unless fields are non-nullable, then `ValueError` is raised:

```
>>> class TheNonNullInput(Input):
...     a_int = non_null(int)
...     a_float = non_null(float)
...     a_nested = non_null(AnotherInput)
...     a_nested_list = non_null(list_of(non_null(AnotherInput)))
```

```
>>> TheNonNullInput(a_int=None, a_float=1.2,
...                 a_nested=AnotherInput(a_str='hi'),
...                 a_nested_list=[AnotherInput(a_str='there')]
...                 )
Traceback (most recent call last):
  ...
ValueError: Int! received null value
>>> TheNonNullInput(a_int=1, a_float=None,
...                 a_nested=AnotherInput(a_str='hi'),
...                 a_nested_list=[AnotherInput(a_str='there')]
...                 )
Traceback (most recent call last):
  ...
ValueError: Float! received null value
>>> TheNonNullInput(a_int=1, a_float=1.2,
...                 a_nested=None,
...                 a_nested_list=[AnotherInput(a_str='there')]
...                 )
Traceback (most recent call last):
  ...
ValueError: AnotherInput! received null value
>>> TheNonNullInput(a_int=1, a_float=1.2,
...                 a_nested=AnotherInput(a_str='hi'),
...                 a_nested_list=[None]
...                 )
Traceback (most recent call last):
  ...
ValueError: AnotherInput! received null value
>>> TheNonNullInput(a_int=1, a_float=1.2,
...                 a_nested=AnotherInput(a_str='hi'),
...                 a_nested_list=None
...                 )
Traceback (most recent call last):
  ...
ValueError: [AnotherInput!]! received null value
```

> **Note:** `selection_list` parameter makes no sense and is ignored, it's only provided to cope with the `ContainerType` interface.

static __new__(*cls*, **args*, ***kwargs*)

class sgqlc.types.**Int**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Maps GraphQL `Int` to Python `int`.

```
>>> Int # or repr()
scalar Int
>>> str(Int)
'Int'
>>> bytes(Int)
b'scalar Int'
```

converter

alias of `int`

class sgqlc.types.**Interface**(**args*, ***kwargs*)

Bases: *ContainerType*

GraphQL `interface Name`.

If the subclass also adds *Interface* to the class declarations, then it will emit `interface Name implements Iface1, Iface2`, also making their fields automatically available in the final class.

Whenever interfaces are instantiated, if there is a `__typename` in `json_data` and the type is known, it will automatically create the more specific type. Otherwise it instantiates the interface itself:

```
>>> class SomeIface(Interface):
...     i = int
...
>>> class TypeWithIface(Type, SomeIface):
...     pass
...
>>> data = {'__typename': 'TypeWithIface', 'i': 123}
>>> SomeIface(data)
TypeWithIface(i=123)
>>> data = {'__typename': 'UnknownType', 'i': 123}
>>> SomeIface(data)
SomeIface(i=123)
```

static __new__(*cls*, **args*, ***kwargs*)

class sgqlc.types.**Scalar**(*json_data*, *selection_list=None*)

Bases: *BaseType*

Basic scalar types, passed thru (no conversion).

This may be used directly if no special checks or conversions are needed. Otherwise use subclasses, like *Int*, *Float*, *String*, *Boolean*, *ID*...

Scalar classes will never produce instance of themselves, rather return the converted value (int, bool...)

```
>>> class MyTypeWithScalar(Type):
...     v = Scalar
...
>>> MyTypeWithScalar({'v': 1}).v
1
>>> MyTypeWithScalar({'v': 'abc'}).v
'abc'
```

Variables are passed thru:

```
>>> MyTypeWithScalar({'v': Variable('var')}).v
$var
```

static __new__(*cls*, *json_data*, *selection_list=None*)

class sgqlc.types.**Schema**(*base_schema=None*)

Bases: object

The schema will contain declared types.

There is a default schema called `global_schema`, a singleton that is automatically assigned to every type that does not provide its own schema.

Once types are constructed, they are automatically added to the schema as properties of the same name, for example *Int* is exposed as `schema.Int`, `schema['Int']` or `schema.scalar['Int']`.

New schema will inherit the types defined at `base_schema`, which defaults to `global_schema`, at the time of their creation. However types added to `base_schema` after the schema creation are not automatically picked by existing schema. The copy happens at construction time.

New types may be added to schema using `schema += type` and removed with `schema -= type`. However those will not affect their member `type.__schema__`, which remains the same (where they where originally created).

The schema is an iterator that will report all registered types.

__bytes__()

GraphQL schema without indentation.

```
>>> print(bytes(global_schema).decode('utf-8'))
schema {
scalar Int
scalar Float
scalar String
...
}
```

__contains__(*key*)

Checks if the type name is known in this schema.

Considering `TypeUsingPython`, previously declared in the module documentation:

```
>>> 'TypeUsingPython' in global_schema
True
>>> 'UnknownTypeName' in global_schema
False
```

__getattr__(*key*)

Get the type using schema attribute.

Considering `TypeUsingPython`, previously declared in the module documentation:

```
>>> global_schema.TypeUsingPython
type TypeUsingPython {
  aInt: Int
  aFloat: Float
  aString: String
  aBoolean: Boolean
  aId: ID
}
```

```
>>> global_schema.UnknownTypeName
Traceback (most recent call last):
  ...
AttributeError: UnknownTypeName
```

One can use `Schema.kind.Type` syntax as well, it exposes an `ODict` object:

```
>>> global_schema.scalar.Int
scalar Int
>>> global_schema.scalar['Int']
scalar Int
>>> global_schema.scalar.UnknownTypeName
Traceback (most recent call last):
  ...
AttributeError: ... has no field UnknownTypeName
>>> global_schema.type.TypeUsingPython
type TypeUsingPython {
...
>>> for t in global_schema.type.values():
...     print(repr(t))
...
type TypeUsingPython {
...
type TypeUsingSGQLC {
...
type TypeUsingFields {
...
type MyOtherType {
...
type MyType {
...
}
```

__getitem__(*key*)

Get the type given its name.

Considering `TypeUsingPython`, previously declared in the module documentation:

```
>>> global_schema['TypeUsingPython']
type TypeUsingPython {
```

(continues on next page)

(continued from previous page)

```
  aInt: Int
  aFloat: Float
  aString: String
  aBoolean: Boolean
  aId: ID
}
```

```
>>> global_schema['UnknownTypeName']
Traceback (most recent call last):
  ...
KeyError: 'UnknownTypeName'
```

__iadd__(*typ*)

Manually add a type to the schema.

Types are automatically once their class is created. Only use this if you're copying a type from one schema to another.

Note that the type name `str(typ)` must not exist in the schema, otherwise `ValueError` is raised.

To remove a type, use `schema -= typ`.

As explained in the *sgqlc.types* documentation, the newly created schema will inherit types from the base schema only at creation time:

```
>>> my_schema = Schema(global_schema)
>>> class MySchemaType(Type):
...     __schema__ = my_schema
...     i = int
...
>>> 'MySchemaType' in global_schema
False
>>> 'MySchemaType' in my_schema
True
```

But `__iadd__` and `__isub__` can be used to add or remove types:

```
>>> global_schema += MySchemaType
>>> 'MySchemaType' in global_schema
True
>>> global_schema -= MySchemaType
>>> 'MySchemaType' in global_schema
False
```

Note that different type with the same name can't be added:

```
>>> my_schema2 = Schema(global_schema)
>>> class MySchemaType(Type):   # redefining, different schema: ok
...     __schema__ = my_schema2
...     f = float
...
>>> my_schema += MySchemaType
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  ...
ValueError: Schema already has MySchemaType=MySchemaType
```

__init__(*base_schema=None*)

__isub__(*typ*)

Remove a type from the schema.

This may be of use to override some type, such as *sgqlc.types.datetime.Date* or *sgqlc.types.datetime.DateTime*.

__iter__()

Schema provides an iterator over *BaseType* subclasses:

```
>>> for t in global_schema:
...     print(repr(t))
...
scalar Int
scalar Float
...
type MyType {
...
}
```

__repr__()

Return repr(self).

__str__()

Short schema, using only type names.

Instead of declaring the whole schema in GraphQL notation as done by `repr()`, just list the type names:

```
>>> print(str(global_schema))
{Int, Float, String, Boolean, ID, ...}
```

class sgqlc.types.**String**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Maps GraphQL `String` to Python `str`.

converter

alias of `str`

class sgqlc.types.**Type**(*json_data*, *selection_list=None*)

Bases: *ContainerType*

GraphQL `type Name`.

If the subclass also adds *Interface* to the class declarations, then it will emit `type Name implements Iface1, Iface2`, also making their fields automatically available in the final class.

class sgqlc.types.**Union**(*json_data*, *selection_list=None*)

Bases: `BaseTypeWithTypename`

This is an abstract class that union of multiple types should inherit and define `__types__`, a list of pre-defined *Type*.

```
>>> class IntOrFloatOrString(Union):
...     __types__ = (Int, float, 'String')
...
>>> IntOrFloatOrString # or repr(), prints out GraphQL!
union IntOrFloatOrString = Int | Float | String
>>> Int in IntOrFloatOrString
True
>>> 'Int' in IntOrFloatOrString  # may use type names as well
True
>>> int in IntOrFloatOrString  # may use native Python types as well
True
>>> ID in IntOrFloatOrString
False
>>> len(IntOrFloatOrString)
3
>>> for t in IntOrFloatOrString:
...     print(repr(t))
scalar Int
scalar Float
scalar String
```

Failing to define types will raise exception:

```
>>> class FailureUnion(Union):
...     pass
Traceback (most recent call last):
  ...
ValueError: FailureUnion: missing __types__
```

Whenever instantiating the type, pass a JSON object with `__typename` (done automatically using fragments via `__as__`):

```
>>> class TypeA(Type):
...     i = int
...
>>> class TypeB(Type):
...     s = str
...
>>> class TypeU(Union):
...     __types__ = (TypeA, TypeB)
...
>>> data = {'__typename': 'TypeA', 'i': 1}
>>> TypeU(data)
TypeA(i=1)
>>> data = {'__typename': 'TypeB', 's': 'hi'}
>>> TypeU(data)
TypeB(s='hi')
```

It nicely handles unknown types:

```
>>> data = {'v': 123}
>>> TypeU(data) # no __typename
UnknownType()
```

(continues on next page)

(continued from previous page)

```
>>> data = {'__typename': 'TypeUnknown', 'v': 123}
>>> TypeU(data) # auto-generates empty types
TypeUnknown()
>>> data = None
>>> TypeU(data)
```

Variables are passed thru:

```
>>> TypeU(Variable('var'))
$var
```

static __new__(*json_data*, *selection_list=None*)

class sgqlc.types.**Variable**(*name*, *graphql_name=None*)

Bases: object

GraphQL variable: `$varName`

Usually given as *Arg* default value:

```
>>> class MyTypeWithVariable(Type):
...     f = Field(str, args={'first': Arg(int, default=Variable('var'))})
...
>>> MyTypeWithVariable
type MyTypeWithVariable {
  f(first: Int = $var): String
}
>>> print(repr(MyTypeWithVariable.f.args['first'].default))
$var
>>> print(str(MyTypeWithVariable.f.args['first'].default))
$var
>>> print(bytes(MyTypeWithVariable.f.args['first'].default).decode('utf8'))
$var
```

__init__(*name*, *graphql_name=None*)

__repr__()

Return repr(self).

__str__()

Return str(self).

static _to_graphql_name(*name*)

Converts a Python name, `a_name` to GraphQL: `aName`.

sgqlc.types.**list_of**(*t*)

Generates list of types ([t])

The example below highlights the usage including its usage with lists:

- `non_null_list_of_int` means it must be a list, not `None`, however list elements may be `None`, ie: `[None, 1, None, 2]`;
- `list_of_non_null_int` means it may be `None` or be a list, however list elements must not be `None`, ie: `None` or `[1, 2]`;

- `non_null_list_of_non_null_int` means it must be a list, not `None` **and** the list elements must not be `None`, ie: `[1, 2]`.

```
>>> class TypeWithListFields(Type):
...     list_of_int = list_of(int)
...     list_of_float = list_of(Float)
...     list_of_string = Field(list_of(String))
...     non_null_list_of_int = non_null(list_of(int))
...     list_of_non_null_int = list_of(non_null(int))
...     non_null_list_of_non_null_int = non_null(list_of(non_null(int)))
...
>>> TypeWithListFields
type TypeWithListFields {
  listOfInt: [Int]
  listOfFloat: [Float]
  listOfString: [String]
  nonNullListOfInt: [Int]!
  listOfNonNullInt: [Int!]
  nonNullListOfNonNullInt: [Int!]!
}
```

It takes care to enforce proper type, including non-null checking on its elements when creating instances. Giving proper JSON data:

```
>>> json_data = {
...     'listOfInt': [1, 2],
...     'listOfFloat': [1.1, 2.1],
...     'listOfString': ['hello', 'world'],
...     'nonNullListOfInt': [None, 1, None, 2],
...     'listOfNonNullInt': [1, 2, 3],
...     'nonNullListOfNonNullInt': [1, 2, 3, 4],
... }
>>> obj = TypeWithListFields(json_data)
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
...
list_of_int [1, 2]
list_of_float [1.1, 2.1]
list_of_string ['hello', 'world']
non_null_list_of_int [None, 1, None, 2]
list_of_non_null_int [1, 2, 3]
non_null_list_of_non_null_int [1, 2, 3, 4]
```

Note that lists that are **not** enclosed in `non_null()` can be `None`:

```
>>> json_data = {
...     'listOfInt': None,
...     'listOfFloat': None,
...     'listOfString': None,
...     'nonNullListOfInt': [None, 1, None, 2],
...     'listOfNonNullInt': None,
...     'nonNullListOfNonNullInt': [1, 2, 3],
... }
>>> obj = TypeWithListFields(json_data)
```

(continues on next page)

(continued from previous page)

```
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
...
list_of_int None
list_of_float None
list_of_string None
non_null_list_of_int [None, 1, None, 2]
list_of_non_null_int None
non_null_list_of_non_null_int [1, 2, 3]
```

Types will be converted, so although not usual (since GraphQL gives you the proper JSON type), this can be done:

```
>>> json_data = {
...     'listOfInt': ['1', '2'],
...     'listOfFloat': [1, '2.1'],
...     'listOfString': ['hello', 2],
...     'nonNullListOfInt': [None, '1', None, 2.1],
...     'listOfNonNullInt': ['1', 2.1, 3],
...     'nonNullListOfNonNullInt': ['1', 2.1, 3, 4],
... }
>>> obj = TypeWithListFields(json_data)
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
...
list_of_int [1, 2]
list_of_float [1.0, 2.1]
list_of_string ['hello', '2']
non_null_list_of_int [None, 1, None, 2]
list_of_non_null_int [1, 2, 3]
non_null_list_of_non_null_int [1, 2, 3, 4]
```

Giving incorrect (nonconvertible) JSON data will raise exceptions:

```
>>> json_data = { 'listOfInt': 1 }
>>> obj = TypeWithListFields(json_data)
Traceback (most recent call last):
  ...
ValueError: TypeWithListFields selection 'list_of_int': ...
```

```
>>> json_data = { 'listOfInt': ['x'] }
>>> obj = TypeWithListFields(json_data)
Traceback (most recent call last):
  ...
ValueError: TypeWithListFields selection 'list_of_int': ...
```

```
>>> json_data = { 'listOfNonNullInt': [1, None] }
>>> obj = TypeWithListFields(json_data)
Traceback (most recent call last):
  ...
ValueError: TypeWithListFields selection 'list_of_non_null_int': ...
```

Lists are usable as input types as well:

```
>>> class TypeWithListInput(Type):
...     a = Field(str, args={'values': Arg(list_of(int), default=[1, 2])})
...     b = Field(str, args={'values': Arg(list_of(int))})
...
>>> TypeWithListInput
type TypeWithListInput {
  a(values: [Int] = [1, 2]): String
  b(values: [Int]): String
}
```

```
>>> print(json.dumps(list_of(int).__to_json_value__([1, 2])))
[1, 2]
>>> print(json.dumps(list_of(int).__to_json_value__(None)))
null
```

Lists can be of complex types, for instance *Input*:

```
>>> class SomeInput(Input):
...     a = int
>>> SomeInputList = list_of(SomeInput)
>>> SomeInputList([{'a': 123}])
[SomeInput(a=123)]
```

Variables may be given as constructor parameters:

```
>>> SomeInputList(Variable('lst'))
$lst
```

Or already realized lists:

```
>>> SomeInputList(SomeInputList([{'a': 123}]))
[SomeInput(a=123)]
```

sgqlc.types.**non_null**(*t*)

Generates non-null type (t!)

```
>>> class TypeWithNonNullFields(Type):
...     a_int = non_null(int)
...     a_float = non_null(Float)
...     a_string = Field(non_null(String))
...
>>> TypeWithNonNullFields
type TypeWithNonNullFields {
  aInt: Int!
  aFloat: Float!
  aString: String!
}
```

Giving proper JSON data:

```
>>> json_data = {'aInt': 1, 'aFloat': 2.1, 'aString': 'hello'}
>>> obj = TypeWithNonNullFields(json_data)
>>> obj
TypeWithNonNullFields(a_int=1, a_float=2.1, a_string='hello')
```

Giving incorrect JSON data:

```
>>> json_data = {'aInt': None, 'aFloat': 2.1, 'aString': 'hello'}
>>> obj = TypeWithNonNullFields(json_data)
Traceback (most recent call last):
  ...
ValueError: TypeWithNonNullFields selection 'a_int': ...
>>> json_data = {'aInt': 1, 'aFloat': None, 'aString': 'hello'}
>>> obj = TypeWithNonNullFields(json_data)
Traceback (most recent call last):
  ...
ValueError: TypeWithNonNullFields selection 'a_float': ...
>>> json_data = {'aInt': 1, 'aFloat': 2.1, 'aString': None}
>>> obj = TypeWithNonNullFields(json_data)
Traceback (most recent call last):
  ...
ValueError: TypeWithNonNullFields selection 'a_string': ...
```

Note: Note that **missing** keys in JSON data are not considered `None`, and they won't show in `iter(obj)`, `__str__()` or `__repr__()`

```
>>> json_data = {'aInt': 1, 'aFloat': 2.1}
>>> obj = TypeWithNonNullFields(json_data)
>>> obj # repr()
TypeWithNonNullFields(a_int=1, a_float=2.1)
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
...
a_int 1
a_float 2.1
```

1.3.2 Sub Modules

- *sgqlc.types.datetime module*
- *sgqlc.types.relay module*

1.4 *sgqlc.types.datetime* module

1.4.1 GraphQL Types for `datetime`

Maps:

- `datetime.time` to GraphQL *Time*
- `datetime.date` to GraphQL *Date*
- `datetime.datetime` to GraphQL *DateTime*

You may either explicitly use this module classes or `datetime`, as they will be automatically recognized by the framework.

Conversions assume ISO 8601 encoding.

Examples

```
>>> from sgqlc.types import Type
>>> class MyDateTimeType(Type):
...     time1 = datetime.time
...     time2 = Time
...     date1 = datetime.date
...     date2 = Date
...     datetime1 = datetime.datetime
...     datetime2 = DateTime
...
>>> MyDateTimeType # or repr() to print out GraphQL
type MyDateTimeType {
  time1: Time
  time2: Time
  date1: Date
  date2: Date
  datetime1: DateTime
  datetime2: DateTime
}
>>> json_data = {
...     'time1': '12:34:56',
...     'time2': '12:34:56-03:00', # set timezone GMT-3h
...     'date1': '2018-01-02',
...     'date2': '20180102', # compact form is accepted
...     'datetime1': '2018-01-02T12:34:56Z', # Z = GMT/UTC
...     'datetime2': '20180102T123456-0300', # compact form is accepted
... }
>>> obj = MyDateTimeType(json_data)
>>> for field_name in obj:
...     print(field_name, repr(obj[field_name]))
time1 datetime.time(12, 34, 56)
time2 datetime.time(12, 34, 56, tzinfo=...(...-1, ...75600)))
date1 datetime.date(2018, 1, 2)
date2 datetime.date(2018, 1, 2)
datetime1 datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=....utc)
datetime2 datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=...75600)))
```

Pre-converted types are allowed:

```
>>> json_data = { 'time1': datetime.time(12, 34, 56) }
>>> obj = MyDateTimeType(json_data)
>>> obj.time1
datetime.time(12, 34, 56)
```

However, invalid encoded strings are not:

```
>>> json_data = { 'time1': '12-3' }
>>> obj = MyDateTimeType(json_data)
Traceback (most recent call last):
   ...
```

(continues on next page)

(continued from previous page)

```
ValueError: MyDateTimeType selection 'time1': ...
>>> json_data = { 'date1': '12-3' }
>>> obj = MyDateTimeType(json_data)
Traceback (most recent call last):
  ...
ValueError: MyDateTimeType selection 'date1': ...
>>> json_data = { 'datetime1': '2018-01-02X12-34-56Z' }
>>> obj = MyDateTimeType(json_data)
Traceback (most recent call last):
  ...
ValueError: MyDateTimeType selection 'datetime1': ...
```

license
ISC

class sgqlc.types.datetime.**Date**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Date encoded as string using ISO8601 (YYYY-MM-SS)

While not a standard GraphQL type, it's often used, so expose to make life simpler.

```
>>> Date('2018-01-02')
datetime.date(2018, 1, 2)
>>> Date('20180102') # compact form
datetime.date(2018, 1, 2)
```

Pre-converted values are allowed:

```
>>> Date(datetime.date(2018, 1, 2))
datetime.date(2018, 1, 2)
```

It can also serialize to JSON:

```
>>> Date.__to_json_value__(datetime.date(2018, 1, 2))
'2018-01-02'
>>> Date.__to_json_value__('2018-01-02')
'2018-01-02'
>>> Date.__to_json_value__(None)
```

class sgqlc.types.datetime.**DateTime**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Date and Time encoded as string using ISO8601 (YYYY-mm-ddTHH:MM:SS[.mmm][+/-HH:MM])

While not a standard GraphQL type, it's often used, so expose to make life simpler.

```
>>> DateTime('2018-01-02T12:34:56') # naive, no timezone
datetime.datetime(2018, 1, 2, 12, 34, 56)
>>> DateTime('2018-01-02T12:34:56Z') # Z = GMT/UTC
datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=datetime.timezone.utc)
>>> DateTime('2018-01-02T12:34:56-05:30')
datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=..., ...70200)))
>>> DateTime('2018-01-02T12:34:56+05:30')
datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=...(...19800)))
```

(continues on next page)

(continued from previous page)

```
>>> DateTime('20180102T123456') # compact form
datetime.datetime(2018, 1, 2, 12, 34, 56)
>>> DateTime('20180102T123456Z') # compact form, GMT/UTC
datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=datetime.timezone.utc)
>>> DateTime('20180102T123456-0530')
datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=..., ...70200)))
>>> DateTime('20180102T123456+0530')
datetime.datetime(2018, 1, 2, 12, 34, 56, tzinfo=...(...19800)))
```

Pre-converted values are allowed:

```
>>> DateTime(datetime.datetime(2018, 1, 2, 12, 34, 56))
datetime.datetime(2018, 1, 2, 12, 34, 56)
```

It can also serialize to JSON:

```
>>> dt = datetime.datetime(2018, 1, 2, 12, 34, 56)
>>> DateTime.__to_json_value__(dt)
'2018-01-02T12:34:56'
>>> DateTime.__to_json_value__('2018-01-02T12:34:56')
'2018-01-02T12:34:56'
>>> tzinfo = datetime.timezone.utc
>>> DateTime.__to_json_value__(dt.replace(tzinfo=tzinfo))
'2018-01-02T12:34:56+00:00'
>>> DateTime.__to_json_value__(None)
```

class sgqlc.types.datetime.**Time**(*json_data*, *selection_list=None*)

Bases: *Scalar*

Time encoded as string using ISO8601 (HH:MM:SS[.mmm][+/-HH:MM])

While not a standard GraphQL type, it's often used, so expose to make life simpler.

```
>>> Time('12:34:56') # naive, no timezone
datetime.time(12, 34, 56)
>>> Time('12:34:56Z') # Z = GMT/UTC
datetime.time(12, 34, 56, tzinfo=datetime.timezone.utc)
>>> Time('12:34:56-05:30')
datetime.time(12, 34, 56, tzinfo=...(...-1, ...70200)))
>>> Time('12:34:56+05:30')
datetime.time(12, 34, 56, tzinfo=...(...19800)))
>>> Time('123456') # compact form
datetime.time(12, 34, 56)
>>> Time('123456Z') # compact form, GMT/UTC
datetime.time(12, 34, 56, tzinfo=datetime.timezone.utc)
>>> Time('123456-0530')
datetime.time(12, 34, 56, tzinfo=...(...-1, ...70200)))
>>> Time('123456+0530')
datetime.time(12, 34, 56, tzinfo=...(...19800)))
```

Pre-converted values are allowed:

```
>>> Time(datetime.time(12, 34, 56))
datetime.time(12, 34, 56)
```

It can also serialize to JSON:

```
>>> Time.__to_json_value__(datetime.time(12, 34, 56))
'12:34:56'
>>> tzinfo = datetime.timezone.utc
>>> Time.__to_json_value__(datetime.time(12, 34, 56, 0, tzinfo))
'12:34:56+00:00'
>>> Time.__to_json_value__('12:34:56')
'12:34:56'
>>> Time.__to_json_value__(None)
```

1.5 *sgqlc.types.relay* module

1.5.1 GraphQL Types for Relay

Exposes `Node` and `Connection`, matching Global Object Identification and Cursor Connections, which are widely used.

Examples

```
>>> from sgqlc.types import Type, Field, list_of
>>> class NodeBasedInterface(Node):
...     a_int = int
...
>>> NodeBasedInterface # or repr()
interface NodeBasedInterface implements Node {
  id: ID!
  aInt: Int
}
>>> class NodeBasedType(Type, Node):
...     a_int = int
...
>>> NodeBasedType # or repr()
type NodeBasedType implements Node {
  id: ID!
  aInt: Int
}
```

Connection subclasses will get `page_info` and `__iadd__` to merge 2 connections:

```
>>> class MyEdge(Type):
...     node = NodeBasedType
...     cursor = str
...
>>> class MyConn(Connection):
...    nodes = list_of(NodeBasedType)
...    edges = list_of(MyEdge)
...
>>> MyConn # or repr()
type MyConn {
```

(continues on next page)

(continued from previous page)

```
  pageInfo: PageInfo!
  nodes: [NodeBasedType]
  edges: [MyEdge]
}
>>> class MyTypeWithConn(Type):
...     conn = Field(MyConn, args=connection_args())
...
>>> MyTypeWithConn # or repr()
type MyTypeWithConn {
  conn(
    after: String
    before: String
    first: Int
    last: Int
  ): MyConn
}
```

Given `json_data1` being the contents of the GraphQL query:

```
query {
   getMyTypeWithConn(id: "...") {
     conn(first: 2) { # first page
       pageInfo { startCursor, endCursor, hasNextPage, hasPreviousPage }
       nodes { id, aInt }
       edges { cursor, node { id, aInt } }
     }
   }
}
```

```
>>> json_data1 = { # page 1 (2 elements of 4)
...     'pageInfo': {
...         'startCursor': 'cursor-1',
...         'endCursor': 'cursor-2',
...         'hasNextPage': True,
...         'hasPreviousPage': False,
...     },
...     'nodes': [
...         {'id': '1111', 'aInt': 1},
...         {'id': '2222', 'aInt': 2},
...     ],
...     'edges': [
...         {'cursor': 'cursor-1', 'node': {'id': '1111', 'aInt': 1}},
...         {'cursor': 'cursor-2', 'node': {'id': '2222', 'aInt': 2}},
...     ],
... }
>>> conn1 = MyConn(json_data1)
>>> print(conn1.page_info)
PageInfo(end_cursor=cursor-2, start_cursor=cursor-1, has_next_page=True...
>>> for n in conn1.nodes:
...     print(repr(n))
NodeBasedType(id='1111', a_int=1)
NodeBasedType(id='2222', a_int=2)
```

(continues on next page)

(continued from previous page)

```
>>> for e in conn1.edges:
...     print(repr(e))
MyEdge(node=NodeBasedType(id='1111', a_int=1), cursor='cursor-1')
MyEdge(node=NodeBasedType(id='2222', a_int=2), cursor='cursor-2')
```

We'd execute the query to fetch the second page as `json_data2`:

```
query {
   getMyTypeWithConn(id: "...") {
     conn(first: 2, after: "cursor-2") { # second page
       pageInfo { startCursor, endCursor, hasNextPage, hasPreviousPage }
       nodes { id, aInt }
       edges { cursor, node { id, aInt } }
     }
   }
}
```

```
>>> json_data2 = { # page 2 (2 elements of 4)
...     'pageInfo': {
...         'startCursor': 'cursor-3',
...         'endCursor': 'cursor-4',
...         'hasNextPage': False,
...         'hasPreviousPage': True,
...     },
...     'nodes': [
...         {'id': '3333', 'aInt': 3},
...         {'id': '4444', 'aInt': 4},
...     ],
...     'edges': [
...         {'cursor': 'cursor-3', 'node': {'id': '3333', 'aInt': 3}},
...         {'cursor': 'cursor-4', 'node': {'id': '4444', 'aInt': 4}},
...     ],
... }
>>> conn2 = MyConn(json_data2)
>>> print(conn2.page_info)
PageInfo(end_cursor=cursor-4, start_cursor=cursor-3, has_next_page=False...
>>> for n in conn2.nodes:
...     print(repr(n))
NodeBasedType(id='3333', a_int=3)
NodeBasedType(id='4444', a_int=4)
>>> for e in conn2.edges:
...     print(repr(e))
MyEdge(node=NodeBasedType(id='3333', a_int=3), cursor='cursor-3')
MyEdge(node=NodeBasedType(id='4444', a_int=4), cursor='cursor-4')
```

One can merge `conn2` into `conn1`, also updating its backing store `json_data1`:

```
>>> conn1 += conn2
>>> print(conn1.page_info)
PageInfo(end_cursor=cursor-4, start_cursor=cursor-1, has_next_page=False...
>>> for n in conn1.nodes:
...     print(repr(n))
```

(continues on next page)

(continued from previous page)

```
NodeBasedType(id='1111', a_int=1)
NodeBasedType(id='2222', a_int=2)
NodeBasedType(id='3333', a_int=3)
NodeBasedType(id='4444', a_int=4)
>>> for e in conn1.edges:
...     print(repr(e))
MyEdge(node=NodeBasedType(id='1111', a_int=1), cursor='cursor-1')
MyEdge(node=NodeBasedType(id='2222', a_int=2), cursor='cursor-2')
MyEdge(node=NodeBasedType(id='3333', a_int=3), cursor='cursor-3')
MyEdge(node=NodeBasedType(id='4444', a_int=4), cursor='cursor-4')
>>> import json
>>> print(json.dumps(json_data1, sort_keys=True, indent=2))
{
  "edges": [
    {
      "cursor": "cursor-1",
      "node": {
        "aInt": 1,
        "id": "1111"
      }
    },
    {
      "cursor": "cursor-2",
      "node": {
        "aInt": 2,
        "id": "2222"
      }
    },
    {
      "cursor": "cursor-3",
      "node": {
        "aInt": 3,
        "id": "3333"
      }
    },
    {
      "cursor": "cursor-4",
      "node": {
        "aInt": 4,
        "id": "4444"
      }
    }
  ],
  "nodes": [
    {
      "aInt": 1,
      "id": "1111"
    },
    {
      "aInt": 2,
      "id": "2222"
    },
```

(continues on next page)

(continued from previous page)

```
    {
      "aInt": 3,
      "id": "3333"
    },
    {
      "aInt": 4,
      "id": "4444"
    }
  ],
  "pageInfo": {
    "endCursor": "cursor-4",
    "hasNextPage": false,
    "hasPreviousPage": false,
    "startCursor": "cursor-1"
  }
}
```

When merging, the receiver connection can be empty:

```
>>> json_data0 = {}
>>> conn0 = MyConn(json_data0)
>>> conn0 += conn1
>>> print(conn0.page_info)
PageInfo(end_cursor=cursor-4, start_cursor=cursor-1, has_next_page=False...
>>> for n in conn0.nodes:
...     print(repr(n))
NodeBasedType(id='1111', a_int=1)
NodeBasedType(id='2222', a_int=2)
NodeBasedType(id='3333', a_int=3)
NodeBasedType(id='4444', a_int=4)
>>> for e in conn0.edges:
...     print(repr(e))
MyEdge(node=NodeBasedType(id='1111', a_int=1), cursor='cursor-1')
MyEdge(node=NodeBasedType(id='2222', a_int=2), cursor='cursor-2')
MyEdge(node=NodeBasedType(id='3333', a_int=3), cursor='cursor-3')
MyEdge(node=NodeBasedType(id='4444', a_int=4), cursor='cursor-4')
>>> print(json.dumps(json_data0, sort_keys=True, indent=2))
{
  "edges": [
    {
      "cursor": "cursor-1",
      "node": {
        "aInt": 1,
        "id": "1111"
      }
    },
    {
      "cursor": "cursor-2",
      "node": {
        "aInt": 2,
        "id": "2222"
      }
```

(continues on next page)

(continued from previous page)


```
    },
    {
      "cursor": "cursor-3",
      "node": {
        "aInt": 3,
        "id": "3333"
      }
    },
    {
      "cursor": "cursor-4",
      "node": {
        "aInt": 4,
        "id": "4444"
      }
    }
  ],
  "nodes": [
    {
      "aInt": 1,
      "id": "1111"
    },
    {
      "aInt": 2,
      "id": "2222"
    },
    {
      "aInt": 3,
      "id": "3333"
    },
    {
      "aInt": 4,
      "id": "4444"
    }
  ],
  "pageInfo": {
    "endCursor": "cursor-4",
    "hasNextPage": false,
    "hasPreviousPage": false,
    "startCursor": "cursor-1"
  }
}
```


license
ISC

class sgqlc.types.relay.**Connection**(*json_data*, *selection_list=None*)

Bases: *Type*

Cursor Connections based on Relay specification.

https://facebook.github.io/relay/graphql/connections.htm

Note: This class exposes += (in-place addition) operator to append information from another connection into

this. The usage is as follow, if `obj.connection.page_info.has_next_page`, then you should query the next page using `after=obj.connection.page_info.end_cursor`. The resulting object should be `obj.connection += obj2.connection`, this will add the contents of `obj2.connection` to `obj.connection`, resetting `obj.connection.page_info.has_next_page`, `obj.connection.page_info.end_cursor` and the JSON backing store, if any.

---

class sgqlc.types.relay.**Node**(**args*, ***kwargs*)

Bases: *Interface*

Global Object Identification based on Relay specification.

https://facebook.github.io/relay/graphql/objectidentification.htm

class sgqlc.types.relay.**PageInfo**(*json_data*, *selection_list=None*)

Bases: *Type*

Connection page information.

https://facebook.github.io/relay/graphql/connections.htm

sgqlc.types.relay.**connection_args**(**lst*, ***mapping*)

Returns the default parameters for connection.

Extra parameters may be given as argument, both as iterable, positional tuples or mapping.

By default, provides:

- `after: String`
- `before: String`
- `first: Int`
- `last: Int`

1.6 *sgqlc.types.uuid* module

1.6.1 GraphQL Types for `uuid`

Maps:

- `uuid.UUID` to GraphQL *UUID*

You may either explicitly use this module class: UUID or `uuid`, as they will be automatically recognized by the framework.

Examples

With both module and class :

```
>>> from sgqlc.types import Type
>>> from sgqlc.types.uuid import UUID
>>> import uuid
>>> class UUIDTestType(Type):
...     uuid1=UUID
...     uuid2=uuid.UUID
```

(continues on next page)

(continued from previous page)

```
...
>>> UUIDTestType
type UUIDTestType {
  uuid1: UUID
  uuid2: UUID
}
>>> nested_json_data = {}
>>> nested_json_data['uuid1'] = '94fda4fb-d574-470b-82e2-0f4ec2a2db20'
>>> nested_json_data['uuid2'] = '94fda4fb-d574-470b-82e2-0f4ec2a2db21'
>>> obj = UUIDTestType(nested_json_data)
>>> for field in obj:
...     print(field, repr(obj[field]))
...
uuid1 UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db20')
uuid2 UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db21')
```

With valid uuid value:

```
>>> class MyUuidType(Type):
...     uuid = UUID
...
>>> MyUuidType # or repr() to print out GraphQL
type MyUuidType {
  uuid: UUID
}
>>> json_data = {'uuid': '94fda4fb-d574-470b-82e2-0f4ec2a2db20'}
>>> obj = MyUuidType(json_data)
>>> obj.uuid
UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db20')
```

Pre-converted type is allowed:

```
>>> json_data = {'uuid': UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db20')}
>>> obj = MyUuidType(json_data)
>>> obj.uuid
UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db20')
```

```
>>> UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db20')
UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db20')
```

With input serialize to JSON: >>> UUID.__to_json_value__('94fda4fb-d574-470b-82e2-0f4ec2a2db20') '94fda4fb-d574-470b-82e2-0f4ec2a2db20' >>> UUID.__to_json_value__(UUID('94fda4fb-d574-470b-82e2-0f4ec2a2db20')) '94fda4fb-d574-470b-82e2-0f4ec2a2db20' >>> UUID.__to_json_value__(None) # doctest: +ELLIPSIS

Invalid encoded strings are not:

```
>>> json_data = {'uuid': 'test 123'}
>>> obj = MyUuidType(json_data)
Traceback (most recent call last):
  ...
ValueError: MyUuidType selection 'uuid': ...
```

```
>>> json_data = {'uuid': 123}
>>> obj = MyUuidType(json_data)
Traceback (most recent call last):
   ...
ValueError: MyUuidType selection 'uuid': ...
```

With null input value:

```
>>> json_data = {'uuid': None}
>>> obj = MyUuidType(json_data)
>>> obj.uuid
```

All types of UUID supported:

```
>>> UUID('5e0d844c-b5bf-11ed-afa1-0242ac120002') # with UUID1
UUID('5e0d844c-b5bf-11ed-afa1-0242ac120002')
>>> UUID('000003e8-b5bf-21ed-9300-325096b39f47') # with UUID2
UUID('000003e8-b5bf-21ed-9300-325096b39f47')
>>> UUID('d5946b2b-447f-3fdf-8366-6c747863484a') # with UUID3
UUID('d5946b2b-447f-3fdf-8366-6c747863484a')
>>> UUID('54f4530b-3052-47b3-9231-b00f8d423448') # with UUID4
UUID('54f4530b-3052-47b3-9231-b00f8d423448')
>>> UUID('5a54a66f-bd21-559f-92fe-7ede767ed4b3') # with UUID5
UUID('5a54a66f-bd21-559f-92fe-7ede767ed4b3')
```

class sgqlc.types.uuid.**UUID**(*json_data*, *selection_list=None*)

Bases: *Scalar*

1.7 *sgqlc.operation* module

1.7.1 Generate Operations (Query and Mutations) using Python

Note: This module could be called "query", however it should also generate mutations and a class `Query` could lead to mistakes, since the users should define their own root `Query` class with the top-level queries in their GraphQL schema.

Users create instance of *Operation* using the `Schema.Query` or `Schema.Mutation` types. From there they proceed accessing members, which will produce *Selector* instances, that once called will produce *Selection* instances, which are automatically added to a *SelectionList* in the parent (operation or selection). The following annotated GraphQL code helps to understand the Python mapping:

```
query { # Operation
   parent(arg: "value") { # Selector, called with arguments
      child { # Selector, called without arguments
         field # Selector called without arguments, Selection without alias
         alias: field(other: 123)
      }
      sibling { x { y } }
   }
}
```

```
op = Operation(Query)
parent = op.parent(arg='value')

child = parent.child
child.field()
child.field(other=123, __alias__='alias')

parent.sibling.x.y()
```

Operation implements `__str__()` and `__repr__()` to generate the GraphQL query for you. It also provide `__bytes__()` to produce compact output, without indentation. It can be passed to *sgqlc.endpoint.base.BaseEndpoint.__call__()* as is.

Another convenience is the `__add__()` to apply the operation to a resulting JSON data, interpreting the results and producing convenient objects:

```
endpoint = HTTPEndpoint(url)
data = endpoint(op)

obj = op + data
print(obj.parent.child.field)
print(obj.parent.sibling.x.y)
```

Performance

When the endpoint is called passing *Operation* it will serialize the operation to a string. This may be costly depending on the operation size and will be done on **every endpoint usage**, there is no caching. Internally it does:

```
query = bytes(op).decode('utf-8')
```

Then it's advised that those looking for extra performance to do this externally and pass the resulting string. A faster version of the code in the previous section is:

```
endpoint = HTTPEndpoint(url)
query = bytes(op).decode('utf-8')
data = endpoint(query)  # faster if used multiple times

# The rest of the code is the same and uses 'op':
obj = op + data
print(obj.parent.child.field)
print(obj.parent.sibling.x.y)
```

This also hints at our second optimization: avoid creating operations using arguments with values that changes. Replace those with the `Variable`, this will allow the query to be converted to string only once and will also help the server – some of them employ caching.

```
# slower version
for a in args:
    op = Operation(Query)           # creates a new operation again
    my_query = op.my_query(arg=a)  # only thing that changed!
    my_query.field()                # do all the field processing again
    data = endpoint(op)             # serializes again
    obj = op + data
```

(continues on next page)

(continued from previous page)

```
    process_my_query(obj)

# faster version
from sgqlc.types import Arg, String, Variable
op = Operation(Query, variables={
   'a': Arg(String),  # this must match the my_query arg type!
})
my_query = op.my_query(arg=Variable('a'))
my_query.field()
query = bytes(op).decode('utf-8')
for a in args:
    data = endpoint(query, variables={'a': a})  # variables are plain JSON!
    obj = op + data
    process_my_query(obj)
```

Unfortunately SGQLC does not implements Automatic Persisted Queries yet, but that technique can be implemented on top of SGQLC. Contributions are welcome ;-)

Code Generator

If you are savvy enough to write GraphQL executable documents using their Domain Specific Language (DSL) and already have `schema.json` or access to a server with introspection you may use the `sgqlc-codegen operation` to automatically generate the SGQLC Operations for you.

The generated code should be stable and can be committed to repositories, leading to minimum `diff` when updated.

See examples:

- GitHub defining a single parametrized (variables) query ListIssues and generates sample_operations.py.
- Shopify uses shopify_operations.gql defining all the operations, including fragments and variables, and outputs the SGQLC code. See the generated shopify_operations.py.

Examples

Let's start defining the types, including the schema root `Query`:

```
>>> from sgqlc.types import *
>>> from datetime import datetime
>>> class Actor(Interface):
...     login = non_null(str)
...
>>> class User(Type, Actor):
...     name = str
...
>>> class Organization(Type, Actor):
...     location = str
...
>>> class ActorConnection(Type):
...     actors = Field(list_of(non_null(Actor)), args={'login': non_null(str)})
...
>>> class Assignee(Type):
...     email = non_null(str)
```

(continues on next page)

(continued from previous page)

```
...
>>> class UserOrAssignee(Union):
...     __types__ = (User, Assignee)
...
>>> class Issue(Type):
...     number = non_null(int)
...     title = non_null(str)
...     body = str
...     reporter = non_null(User)
...     assigned = UserOrAssignee
...     commenters = ActorConnection
...
>>> class ReporterFilterInput(Input):
...     name_contains = str
...
>>> class IssuesFilter(Input):
...     reporter = list_of(ReporterFilterInput)
...     start_date = non_null(datetime)
...     end_date = datetime
...
>>> class Repository(Type):
...     id = ID
...     name = non_null(str)
...     owner = non_null(Actor)
...     issues = Field(list_of(non_null(Issue)), args={
...         'title_contains': str,
...         'reporter_login': str,
...         'filter': IssuesFilter,
...     })
...
>>> class Query(Type):
...     repository = Field(Repository, args={'id': non_null(ID)})
...
>>> class Mutation(Type):
...     add_issue = Field(Issue, args={
...         'repository_id': non_null(ID),
...         'title': non_null(str),
...         'body': str,
...     })
...
>>> global_schema
schema {
  ...
  interface Actor {
    login: String!
  }
  type User implements Actor {
    login: String!
    name: String
  }
  type Organization implements Actor {
    login: String!
```

(continues on next page)

(continued from previous page)

```
    location: String
  }
  type ActorConnection {
    actors(login: String!): [Actor!]
  }
  type Assignee {
    email: String!
  }
  union UserOrAssignee = User | Assignee
  type Issue {
    number: Int!
    title: String!
    body: String
    reporter: User!
    assigned: UserOrAssignee
    commenters: ActorConnection
  }
  input ReporterFilterInput {
    nameContains: String
  }
  input IssuesFilter {
    reporter: [ReporterFilterInput]
    startDate: DateTime!
    endDate: DateTime
  }
  type Repository {
    id: ID
    name: String!
    owner: Actor!
    issues(titleContains: String, reporterLogin: String, filter: IssuesFilter): [Issue!]
  }
  type Query {
    repository(id: ID!): Repository
  }
  type Mutation {
    addIssue(repositoryId: ID!, title: String!, body: String): Issue
  }
}
```

Selecting to Generate Queries

Then let's select numbers and titles of issues of repository with identifier `repo1`:

```
>>> op = Operation(Query)
>>> repository = op.repository(id='repo1')
>>> repository.issues.number()
number
>>> repository.issues.title()
title
>>> op # or repr(), prints out GraphQL!
query {
```

(continues on next page)

(continued from previous page)

```
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
```

You can see we stored `op.repository(id='repo1')` result in a variable, later reusing it. Executing this statement will emit a new *Selection* and only one field selection is allowed in the selection list (unless an alias is used). Trying the code below will **error**:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.number() # ok!
number
>>> op.repository(id='repo1').issues.title() # fails
Traceback (most recent call last):
  ...
ValueError: repository already have a selection repository(id: "repo1") {
  issues {
    number
  }
}. Maybe use __alias__ as param?
```

That is, if you wanted to query for two repositories, you should use `__alias__` argument in the call. But here would **not** produce the query we want, as seen below:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.number()
number
>>> op.repository(id='repo1', __alias__='alias').issues.title()
title
>>> op  # not what we want in this example, 2 independent queries
query {
  repository(id: "repo1") {
    issues {
      number
    }
  }
  alias: repository(id: "repo1") {
    issues {
      title
    }
  }
}
```

In our case, to get the correct query, do as in the first example and save the result of `op.repository(id='repo1')`.

Aliases may be used to rename fields everywhere, not just in the topmost query, and for other reasons other than allow two calls with the same name. One may use it to translate API fields to something else, example:

```
>>> op = Operation(Query)
>>> repository = op.repository(id='repo1')
```

(continues on next page)

(continued from previous page)

```
>>> repository.issues.number(__alias__='code')
code: number
>>> op # or repr(), prints out GraphQL!
query {
  repository(id: "repo1") {
    issues {
      code: number
    }
  }
}
```

Last but not least, in the first example you can also note that we're not calling `issues`, just accessing its members. This is a shortcut for an empty call, and the handle is saved for you (ease of use):

```
>>> op = Operation(Query)
>>> repository = op.repository(id='repo1')
>>> repository.issues().number()
number
>>> repository.issues().title()
title
>>> op # or repr(), prints out GraphQL!
query {
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
```

This could be rewritten saving the issues selector:

```
>>> op = Operation(Query)
>>> issues = op.repository(id='repo1').issues()
>>> issues.number()
number
>>> issues.title()
title
>>> op
query {
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
```

Or even simpler with `__fields__(*names, **names_and_args)`:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.__fields__('number', 'title')
>>> op
```

(continues on next page)

(continued from previous page)

```
query {
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.__fields__(
...     number=True,
...     title=True,
... )
>>> op
query {
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
```

Which also allows to include all but some fields:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.__fields__(
...     __exclude__=('body', 'reporter', 'commenters'),
... )
>>> op
query {
  repository(id: "repo1") {
    issues {
      number
      title
      assigned {
        __typename
        ... on User {
          login
          name
        }
        ... on Assignee {
          email
        }
      }
    }
  }
}
```

Or using named arguments:

```
>>> op = Operation(Query)
```

(continues on next page)

(continued from previous page)

```
>>> op.repository(id='repo1').issues.__fields__(
...     body=False,
...     reporter=False,
...     commenters=False,
... )
>>> op
query {
  repository(id: "repo1") {
    issues {
      number
      title
      assigned {
        __typename
        ... on User {
          login
          name
        }
        ... on Assignee {
          email
        }
      }
    }
  }
}
```

If no arguments are given to `__fields__()`, then it defaults to include every member, and this is done recursively:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.__fields__()
>>> op
query {
  repository(id: "repo1") {
    issues {
      number
      title
      body
      reporter {
        login
        name
      }
      assigned {
        __typename
        ... on User {
          login
          name
        }
        ... on Assignee {
          email
        }
      }
      commenters {
        actors {
```

(continues on next page)

(continued from previous page)

```
          login
        }
      }
    }
  }
}
```

Named arguments may be used to provide fields with argument values:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').__fields__(
...     issues={'title_contains': 'bug'}, # adds field and children
... )
>>> op
query {
  repository(id: "repo1") {
    issues(titleContains: "bug") {
      number
      title
      body
      reporter {
        login
        name
      }
      assigned {
        __typename
      }
    }
  }
}
```

Arguments can be given as tuple of key-value pairs as well:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').__fields__(
...     issues=(('title_contains', 'bug'),), # adds field and children
... )
>>> op
query {
  repository(id: "repo1") {
    issues(titleContains: "bug") {
      number
      title
      body
      reporter {
        login
        name
      }
      assigned {
        __typename
      }
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

By default `__typename` is only included when selecting `Union`, if that should be included in every `Type`, then you must specify `__typename__` as a selected field. It's handled recursively:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.__fields__('__typename__')
>>> op
query {
  repository(id: "repo1") {
    issues {
      __typename
      number
      title
      body
      reporter {
        __typename
        login
        name
      }
      assigned {
        __typename
        ... on User {
          login
          name
        }
        ... on Assignee {
          email
        }
      }
      commenters {
        __typename
        actors {
          __typename
          login
        }
      }
    }
  }
}
```

Or included using `__typename__=True`:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.__fields__(__typename__=True)
>>> op
query {
  repository(id: "repo1") {
    issues {
      __typename
      number
```

(continues on next page)

(continued from previous page)

```
      title
      body
      reporter {
        __typename
        login
        name
      }
      assigned {
        __typename
        ... on User {
          login
          name
        }
        ... on Assignee {
          email
        }
      }
      commenters {
        __typename
        actors {
          __typename
          login
        }
      }
    }
  }
}
```

If a field of a container type (interface or type) is used without explicit fields as documented above, all of its fields will be added automatically. It will avoid dependency loops and limit the allowed nest depth to 2 by default, but that can be overridden with an explicit `auto_select_depth` to `__to_graphql__()` (which is used by `str()`, `repr()` and the likes):

```
>>> op = Operation(Query)
>>> op.repository(id='repo1') # printed with depth=2 (default)
repository(id: "repo1") {
  id
  name
  owner {
    login
  }
  issues {
    number
    title
    body
  }
}
>>> op # the whole query printed with depth=2 (default)
query {
  repository(id: "repo1") {
    id
    name
```

(continues on next page)

(continued from previous page)

```
    owner {
      login
    }
    issues {
      number
      title
      body
    }
  }
}
```

```
>>> print(op.__to_graphql__(auto_select_depth=1)) # omits owner/issues
query {
  repository(id: "repo1") {
    id
    name
  }
}
>>> print(op.__to_graphql__(auto_select_depth=3)) # shows reporter
query {
  repository(id: "repo1") {
    id
    name
    owner {
      login
    }
    issues {
      number
      title
      body
      reporter {
        login
        name
      }
      assigned {
        __typename
      }
    }
  }
}
>>> print(op.__to_graphql__(auto_select_depth=4)) # shows assigned sub-types
query {
  repository(id: "repo1") {
    id
    name
    owner {
      login
    }
    issues {
      number
      title
      body
```

(continues on next page)

(continued from previous page)

```
      reporter {
        login
        name
      }
      assigned {
        __typename
        ... on User {
          login
          name
        }
        ... on Assignee {
          email
        }
      }
      commenters {
        actors {
          login
        }
      }
    }
  }
}
```

If `__typename` is to be automatically selected, then use `typename=True`:

```
>>> print(op.__to_graphql__(auto_select_depth=4, typename=True))
query {
  repository(id: "repo1") {
    __typename
    id
    name
    owner {
      __typename
      login
    }
    issues {
      __typename
      number
      title
      body
      reporter {
        __typename
        login
        name
      }
      assigned {
        __typename
        ... on User {
          login
          name
        }
        ... on Assignee {
```

(continues on next page)

(continued from previous page)

```
          email
        }
      }
      commenters {
        __typename
        actors {
          __typename
          login
        }
      }
    }
  }
}
```

Note: The built-in object type `__typename` would cause issues with Python's name mangling as it would be translated to the private class member name. In order to avoid this issue, whenever selecting GraphQL's `__typename` use the `__typename__` Python name. Example:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').__typename__()
__typename
>>> op
query {
  repository(id: "repo1") {
    __typename
  }
}
```

Interpret Query Results

Given the operation explained above:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.__fields__('number', 'title')
>>> op
query {
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
```

After calling the GraphQL endpoint, you should get a JSON object that matches the one below:

```
>>> json_data = {'data': {
...     'repository': {'issues': [
...         {'number': 1, 'title': 'found a bug'},
```

(continues on next page)

(continued from previous page)

```
...         {'number': 2, 'title': 'a feature request'},
...     ]},
... }}
```

To interpret this, simply add the data to the operation:

```
>>> obj = op + json_data
>>> repository = obj.repository
>>> for issue in repository.issues:
...     print(issue)
Issue(number=1, title=found a bug)
Issue(number=2, title=a feature request)
```

Which are instances of classes declared in the beginning of example section:

```
>>> repository.__class__ is Repository
True
>>> repository.issues[0].__class__ is Issue
True
```

While it's mostly the same as creating instances yourself:

```
>>> repository = Repository(json_data['data']['repository'])
>>> for issue in repository.issues:
...     print(issue)
Issue(number=1, title=found a bug)
Issue(number=2, title=a feature request)
```

The difference is that it will handle **aliases** for you:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1', __alias__='r_name1').issues.__fields__(
...     number='code', title='headline',
... )
>>> op.repository(id='repo2', __alias__='r_name2').issues.__fields__(
...     'number', 'title',
... )
>>> op
query {
  r_name1: repository(id: "repo1") {
    issues {
      code: number
      headline: title
    }
  }
  r_name2: repository(id: "repo2") {
    issues {
      number
      title
    }
  }
}
```

```
>>> json_data = {'data': {
...     'r_name1': {'issues': [
...         {'code': 1, 'headline': 'found a bug'},
...         {'code': 2, 'headline': 'a feature request'},
...     ]},
...     'r_name2': {'issues': [
...         {'number': 10, 'title': 'something awesome'},
...         {'number': 20, 'title': 'other thing broken'},
...     ]},
... }}
>>> obj = op + json_data
>>> for issue in obj.r_name1.issues:
...     print(issue)
Issue(code=1, headline=found a bug)
Issue(code=2, headline=a feature request)
>>> for issue in obj.r_name2.issues:
...     print(issue)
Issue(number=10, title=something awesome)
Issue(number=20, title=other thing broken)
```

Updating also reflects on the correct backing store:

```
>>> obj.r_name2.name = 'repo2 name'
>>> json_data['data']['r_name2']['name']
'repo2 name'
```

It also works with auto selection:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1')
repository(id: "repo1") {
  id
  name
  owner {
    login
  }
  issues {
    number
    title
    body
  }
}
>>> json_data = {'data': {
...     'repository': {'id': 'repo1', 'name': 'Repo #1'},
... }}
>>> obj = op + json_data
>>> obj.repository.name
'Repo #1'
```

And also if `__typename__` is selected:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1', __typename__=True)
```

(continues on next page)

(continued from previous page)

```
repository(id: "repo1") {
  __typename
  id
  name
  owner {
    __typename
    login
  }
  issues {
    __typename
    number
    title
    body
  }
}
>>> json_data = {'data': {
...     'repository': {
...         '__typename': 'Repository', 'id': 'repo1', 'name': 'Repo #1',
...         'owner': {'__typename': 'Actor', 'login': 'name'},
...         'issues': [{
...             '__typename': 'Issue', 'number': 1, 'title': 'title',
...         }],
...     },
... }}
>>> obj = op + json_data
>>> obj.repository
Repository(__typename__='Repository', id='repo1', name='Repo #1', owner=Actor(__typename_
↪_='Actor', login='name'), issues=[Issue(__typename__='Issue', number=1, title='title
↪')])
```

Error Reporting

If the returned data contains only `errors` and no `data`, the interpretation will raise an error `GraphQLErrors`:

```
>>> json_data = {'errors': [{'message': 'some message'}]}
>>> try:
...     obj = op + json_data
... except GraphQLErrors as ex:
...     print('Got error:', repr(ex))
...     print(ex.errors)
Got error: GraphQLErrors('some message'...
[{'message': 'some message'}]
```

If there are mixed data and errors, the object is returned with `__errors__` attribute set to the errors:

```
>>> json_data = {
...     'errors': [{'message': 'some message'}],
...     'data': {
...         'repository': {'id': 'repo1', 'name': 'Repo #1'},
...     },
... }
```

(continues on next page)

(continued from previous page)

```
>>> obj = op + json_data
>>> obj.repository.name
'Repo #1'
>>> obj.__errors__
[{'message': 'some message'}]
```

Mutations

Mutations are handled as well, just use that as *Operation* root type:

```
>>> op = Operation(Mutation)
>>> op.add_issue(repository_id='repo1', title='an issue').__fields__()
>>> op
mutation {
  addIssue(repositoryId: "repo1", title: "an issue") {
    number
    title
    body
    reporter {
      login
      name
    }
    assigned {
      __typename
      ... on User {
        login
        name
      }
      ... on Assignee {
        email
      }
    }
    commenters {
      actors {
        login
      }
    }
  }
}
```

Inline Fragments & Interfaces

When a field specifies an interface such as the `Repository.owner` in our example, only the interface fields can be queried. However, the actual type may implement much more, and to solve that in GraphQL we usually do an inline fragment `... on ActualType { field1, field2 }`.

To achieve that we use the `__as__(ActualType)` on the selection list, example:

```
>>> op = Operation(Query)
>>> repo = op.repository(id='repo1')
```

(continues on next page)

(continued from previous page)

```
>>> repo.owner.login() # interface fields can be declared as usual
login
>>> repo.owner().__as__(Organization).location() # location field for Orgs
location
>>> repo.owner.__as__(User).name() # name field for Users
name
>>> repo.issues().assigned.__as__(Assignee).email()
email
>>> repo.issues().assigned.__as__(User).login()
login
>>> repo.issues().commenters().actors().login()
login
>>> repo.issues().commenters().actors().__as__(Organization).location()
location
>>> repo.issues().commenters().actors().__as__(User).name()
name
>>> op
query {
  repository(id: "repo1") {
    owner {
      login
      __typename
      ... on Organization {
        location
      }
      ... on User {
        name
      }
    }
    issues {
      assigned {
        __typename
        ... on Assignee {
          email
        }
        ... on User {
          login
        }
      }
      commenters {
        actors {
          login
          __typename
          ... on Organization {
            location
          }
          ... on User {
            name
          }
        }
      }
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Note that `__typename` is automatically selected so it can create the proper type when interprets the results:

```
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'User',
...     'login': 'user',
...     'name': 'User Name',
...     },
...     'issues': [
...       {
...           'assigned': {'__typename': 'Assignee', 'email': 'e@mail.com'},
...           'commenters': {
...               'actors': [
...                   {'login': 'user', '__typename': 'User', 'name': 'User Name'},
...                   {'login': 'a-company', '__typename': 'Organization', 'location':
↪'that place'}
...               ]
...           }
...       },
...       {
...           'assigned': {'__typename': 'User', 'login': 'xpto'},
...           'commenters': {
...               'actors': [
...                   {'login': 'user', '__typename': 'User', 'name': 'User Name'},
...                   {'login': 'xpto', '__typename': 'User'}
...               ]
...           }
...       },
...     ],
... }}}
>>> obj = op + json_data
>>> obj.repository.owner
User(login='user', __typename__='User', name='User Name')
>>> for i in obj.repository.issues:
...     print(i)
Issue(assigned=Assignee(__typename__=Assignee, email=e@mail.com),␣
↪commenters=ActorConnection(actors=[User(login='user', __typename__='User', name='User␣
↪Name'), Organization(login='a-company', __typename__='Organization', location='that␣
↪place')]))
Issue(assigned=User(__typename__=User, login=xpto),␣
↪commenters=ActorConnection(actors=[User(login='user', __typename__='User', name='User␣
↪Name'), User(login='xpto', __typename__='User')]))
```

```
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'Organization',
...     'login': 'a-company',
...     'name': 'A Company',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
```

(continues on next page)

(continued from previous page)

```
Organization(login='a-company', __typename__='Organization')
```

If the returned type doesn't have an explicit type fields, the Interface field is returned:

```
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'SomethingElse',
...     'login': 'something-else',
...     'field': 'value',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
Actor(login='something-else', __typename__='SomethingElse')
```

In the unusual situation where `__typename` is not returned, it's going to behave as the interface type as well:

```
>>> json_data = {'data': {'repository': {'owner': {
...     'login': 'user',
...     'name': 'User Name',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
Actor(login='user')
```

Auto-selection works on inline fragments (casts) as well:

```
>>> op = Operation(Query)
>>> repo = op.repository(id='repo1')
>>> repo.owner.__as__(User).__fields__()
>>> op
query {
  repository(id: "repo1") {
    owner {
      __typename
      ... on User {
        login
        name
      }
    }
  }
}
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'User', 'login': 'user', 'name': 'User Name',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
User(__typename__='User', login='user', name='User Name')
```

Named Fragments

Named fragments are a way to reuse selection blocks and allow optimizations to be employed. They also allow shorter documents if the fragment is used more than once.

They are similar to Inline Fragments described above as they allow selecting on interfaces and unions.

```
>>> org_loc_frag = Fragment(Organization, 'OrganizationLocationFragment')
>>> org_loc_frag.location()
location
>>> org_login_frag = Fragment(Organization, 'OrganizationLoginFragment')
>>> org_login_frag.login()
login
>>> user_frag = Fragment(User, 'UserFragment')
>>> user_frag.name()
name
>>> assignee_frag = Fragment(Assignee, 'AssigneeFragment')
>>> assignee_frag.email()
email
>>> op = Operation(Query)
>>> repo = op.repository(id='repo1')
>>> repo.owner.login() # interface fields can be declared as usual
login
>>> repo.owner().__fragment__(org_loc_frag)
>>> repo.owner().__fragment__(org_login_frag)  # can do many on the same type
>>> repo.owner.__fragment__(user_frag)
>>> repo.issues().assigned.__fragment__(assignee_frag)
>>> repo.issues().assigned.__fragment__(user_frag)
>>> repo.issues().commenters().actors().login()
login
>>> repo.issues().commenters().actors().__fragment__(org_loc_frag)
>>> repo.issues().commenters().actors().__fragment__(user_frag)
>>> op
query {
  repository(id: "repo1") {
    owner {
      login
      __typename
      ...OrganizationLocationFragment
      ...OrganizationLoginFragment
      ...UserFragment
    }
    issues {
      assigned {
        __typename
        ...AssigneeFragment
        ...UserFragment
      }
      commenters {
        actors {
          login
          __typename
          ...OrganizationLocationFragment
```

(continues on next page)

(continued from previous page)

```
          ...UserFragment
        }
      }
    }
  }
}
fragment OrganizationLocationFragment on Organization {
  location
}
fragment OrganizationLoginFragment on Organization {
  login
}
fragment UserFragment on User {
  name
}
fragment AssigneeFragment on Assignee {
  email
}
```

Note that `__typename` is automatically selected so it can create the proper type when interprets the results:

```
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'User',
...     'login': 'user',
...     'name': 'User Name',
...     },
...     'issues': [
...       {
...           'assigned': {'__typename': 'Assignee', 'email': 'e@mail.com'},
...           'commenters': {
...               'actors': [
...                   {'login': 'user', '__typename': 'User', 'name': 'User Name'},
...                   {'login': 'a-company', '__typename': 'Organization', 'location':
↪'that place'}
...               ]
...           }
...       },
...       {
...           'assigned': {'__typename': 'User', 'name': 'User'},
...           'commenters': {
...               'actors': [
...                   {'login': 'user', '__typename': 'User', 'name': 'User Name'},
...                   {'login': 'xpto', '__typename': 'User'}
...               ]
...           }
...       },
...     ],
... }}}
>>> obj = op + json_data
>>> obj.repository.owner
User(login='user', __typename__='User', name='User Name')
>>> for i in obj.repository.issues:
```

(continues on next page)

(continued from previous page)

```
...     print(i)
Issue(assigned=Assignee(__typename__=Assignee, email=e@mail.com),␣
↪commenters=ActorConnection(actors=[User(login='user', __typename__='User', name='User␣
↪Name'), Organization(login='a-company', __typename__='Organization', location='that␣
↪place')]))
Issue(assigned=User(__typename__=User, name=User),␣
↪commenters=ActorConnection(actors=[User(login='user', __typename__='User', name='User␣
↪Name'), User(login='xpto', __typename__='User')]))
```

```
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'Organization',
...     'login': 'a-company',
...     'location': 'somewhere',
...     'name': 'A Company',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
Organization(login='a-company', __typename__='Organization', location='somewhere')
```

If the returned type doesn't have an explicit type fields, the Interface field is returned:

```
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'SomethingElse',
...     'login': 'something-else',
...     'field': 'value',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
Actor(login='something-else', __typename__='SomethingElse')
```

In the unusual situation where `__typename` is not returned, it's going to behave as the interface type as well:

```
>>> json_data = {'data': {'repository': {'owner': {
...     'login': 'user',
...     'name': 'User Name',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
Actor(login='user')
```

Auto-selection works on fragments as well:

```
>>> auto_sel_user = Fragment(User, 'AutoSelectedUser')
>>> auto_sel_user.__fields__()
>>> op = Operation(Query)
>>> op.repository(id='repo1').owner.__fragment__(auto_sel_user)
>>> op
query {
  repository(id: "repo1") {
    owner {
      __typename
      ...AutoSelectedUser
```

(continues on next page)

(continued from previous page)

```
    }
  }
}
fragment AutoSelectedUser on User {
  login
  name
}
>>> json_data = {'data': {'repository': {'owner': {
...     '__typename': 'User', 'login': 'user', 'name': 'User Name',
... }}}}
>>> obj = op + json_data
>>> obj.repository.owner
User(__typename__='User', login='user', name='User Name')
```

Utilities

Starting with the first selection example:

```
>>> op = Operation(Query)
>>> repository = op.repository(id='repo1')
>>> repository.issues.number()
number
>>> repository.issues.title()
title
```

One can get a indented print out using `repr()`:

```
>>> print(repr(op))
query {
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
>>> print(repr(repository))
repository(id: "repo1") {
  issues {
    number
    title
  }
}
>>> print(repr(repository.issues.number()))
number
```

Note that *Selector* is different:

```
>>> print(repr(repository.issues.number))
Selector(field=number)
```

Or can get a compact print out without indentation using `bytes()`:

```
>>> print(bytes(op).decode('utf-8'))
query {
repository(id: "repo1") {
issues {
number
title
}
}
}
>>> print(bytes(repository).decode('utf-8'))
repository(id: "repo1") {
issues {
number
title
}
}
>>> print(bytes(repository.issues.number()).decode('utf-8'))
number
```

Selection and *Selector* both implement `len()`:

```
>>> len(op)                       # number of selections (here: top level)
1
>>> len(repository.issues())      # number of selections
2
>>> len(repository.issues)        # number of selections (implicit empty call)
2
>>> len(repository.issues.title()) # leaf is always 1
1
```

Selection and *Selector* both implement `dir()` to also list fields:

```
>>> for name in dir(repository.issues()): # on selection also yields fields
...     if not name.startswith('_'):
...         print(name)
assigned
body
commenters
number
reporter
title
>>> for name in dir(repository.issues): # same for selector
...     if not name.startswith('_'):
...         print(name)
assigned
body
commenters
number
reporter
title
>>> for name in dir(repository.issues.number()): # no fields for scalar
...     if not name.startswith('_'):
...         print(name)
```

(continues on next page)

(continued from previous page)

```
>>> for name in dir(repository.issues.number): # no fields for scalar
...     if not name.startswith('_'):
...         print(name)
```

Classes also implement `iter()` to iterate over selections:

```
>>> for i, sel in enumerate(op):
...     print('#%d: %s' % (i, sel))
#0: repository(id: "repo1") {
  issues {
    number
    title
  }
}
>>> for i, sel in enumerate(repository):
...     print('#%d: %s' % (i, sel))
#0: issues {
  number
  title
}
>>> for i, sel in enumerate(repository.issues):
...     print('#%d: %s' % (i, sel))
#0: number
#1: title
>>> for i, sel in enumerate(repository.issues()):
...     print('#%d: %s' % (i, sel))
#0: number
#1: title
>>> for i, sel in enumerate(repository.issues.number()):
...     print('#%d: %s' % (i, sel))
#0: number
```

Given a *Selector* one can query a **selection** given its alias:

```
>>> op = Operation(Query)
>>> op.repository(id='repo1').issues.number()
number
>>> op.repository(id='repo2', __alias__='alias').issues.title()
title
>>> type(op['repository'])  # it's the selector, not a selection!
<class 'sgqlc.operation.__init__.Selector'>
>>> op['repository'].__selection__() # default selection
repository(id: "repo1") {
  issues {
    number
  }
}
>>> op['repository'].__selection__('alias') # aliased selection
alias: repository(id: "repo2") {
  issues {
    title
  }
}
```

Which is useful to query the selection alias and arguments:

```
>>> op['repository'].__selection__('alias').__alias__
'alias'
>>> op['repository'].__selection__('alias').__args__
{'id': 'repo2'}
>>> op['repository'].__selection__().__args__
{'id': 'repo1'}
```

To get the arguments of the default (non-aliased) one can use the shortcut:

```
>>> op['repository'].__args__
{'id': 'repo1'}
```

license
ISC

class sgqlc.operation.**Operation**(*typ=None*, *name=None*, ***args*)

Bases: object

GraphQL Operation: query or mutation.

The given type must be one of `schema.Query` or `schema.Mutation`, defaults to `global_schema.Query` or whatever is defined as `global_schema.query_type`.

The operation has an internal *sgqlc.operation.SelectionList* and will proxy attributes and item access to it, thus offering selectors and automatically handling selections:

```
op = Operation()
op.parent.field.child()
op.parent.field(param1=value1, __alias__'q2').child()
```

Once data is fetched and parsed as JSON object containing the field `data`, the operation can be used to interpret this data using the addition operator (no clearly named method to avoid clashing with selections):

```
op = Operation()
op.parent.field.child()

endpoint = HTTPEndpoint('http://my.server.com/graphql')
json_data = endpoint(op)

parent = op + json_data
print(parent.field.child)
```

Example usage:

```
>>> op = Operation(global_schema.Query)
>>> op.repository
Selector(field=repository)
>>> repository = op.repository(id='repo1')
>>> repository.issues.number()
number
>>> repository.issues.title()
title
>>> op # or repr(), prints out GraphQL!
query {
```

(continues on next page)

(continued from previous page)

```
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
```

The root type can be omitted, then `global_schema.Query` or whatever is defined as `` `global_schema.query_type `` is used:

```
>>> op = Operation()  # same as Operation(global_schema.Query)
>>> op.repository
Selector(field=repository)
```

Operations can be named:

```
>>> op = Operation(name='MyOp')
>>> repository = op.repository(id='repo1')
>>> repository.issues.number()
number
>>> repository.issues.title()
title
>>> op # or repr(), prints out GraphQL!
query MyOp {
  repository(id: "repo1") {
    issues {
      number
      title
    }
  }
}
```

Operations can also have argument (variables), in this case it must be named (otherwise a name is created based on root type name, such as `"Query"`):

```
>>> from sgqlc.types import Variable
>>> op = Operation(repo_id=str, reporter_login=str)
>>> repository = op.repository(id=Variable('repo_id'))
>>> issues = repository.issues(reporter_login=Variable('reporter_login'))
>>> issues.__fields__('number', 'title')
>>> op # or repr(), prints out GraphQL!
query Query($repoId: String, $reporterLogin: String) {
  repository(id: $repoId) {
    issues(reporterLogin: $reporterLogin) {
      number
      title
    }
  }
}
```

If variable name conflicts with the parameter, you can pass them as a single `variables` parameter containing a dict.

```
>>> from sgqlc.types import Variable
>>> op = Operation(name='MyOperation', variables={'name': str})
>>> op.repository(id=Variable('name')).name()
name
>>> op # or repr(), prints out GraphQL!
query MyOperation($name: String) {
  repository(id: $name) {
    name
  }
}
```

Complex argument types are also supported as JSON object (GraphQL names and raw types) or actual types:

```
>>> op = Operation()
>>> repository = op.repository(id='sgqlc')
>>> issues = repository.issues(filter={
...     'reporter': [{'nameContains': 'Gustavo'}],
...     'startDate': '2019-01-01T00:00:00+00:00',
... })
>>> issues.__fields__('number', 'title')
>>> op # or repr(), prints out GraphQL!
query {
  repository(id: "sgqlc") {
    issues(filter: {reporter: [{nameContains: "Gustavo"}], startDate: "2019-01-
↪01T00:00:00+00:00"}) {
      number
      title
    }
  }
}
```

```
>>> from datetime import datetime, timezone
>>> from sgqlc.types import global_schema
>>> op = Operation()
>>> repository = op.repository(id='sgqlc')
>>> issues = repository.issues(filter=global_schema.IssuesFilter(
...     reporter=[global_schema.ReporterFilterInput(name_contains='Gustavo')],
...     start_date=datetime(2019, 1, 1, tzinfo=timezone.utc),
... ))
>>> issues.__fields__('number', 'title')
>>> op # or repr(), prints out GraphQL!
query {
  repository(id: "sgqlc") {
    issues(filter: {reporter: [{nameContains: "Gustavo"}], startDate: "2019-01-
↪01T00:00:00+00:00"}) {
      number
      title
    }
  }
}
```

Selectors can be acquired as attributes or items, but they must exist in the target type:

```
>>> op = Operation()
>>> op.repository
Selector(field=repository)
>>> op['repository']
Selector(field=repository)
>>> op.does_not_exist
Traceback (most recent call last):
  ...
AttributeError: query {
} has no field does_not_exist
>>> op['does_not_exist']
Traceback (most recent call last):
  ...
KeyError: 'Query has no field does_not_exist'
```

__init__(*typ=None*, *name=None*, ***args*)

__repr__()

Return repr(self).

__str__()

Return str(self).

__weakref__

list of weak references to the object (if defined)

class sgqlc.operation.**Selection**(*alias*, *field*, *args*, *typename=None*)

Bases: object

Select a field with in a container type.

> **Warning:** Do not create instances directly, use *sgqlc.operation.Selector* instead.

A selection matches the GraphQL statement to select a field from a type, it may contain an alias and parameters:

```
query {
  parent {
    field
    field(param1: value1, param2: value2)
    alias: field(param1: value1, param2: value2)
  }
}
```

Attributes or items access will result in *sgqlc.operation.Selector* matching the **target type** field:

```
parent.field.child
```

For container types one can provide a batch of fields using *sgqlc.operation.Selection.__fields__()*:

```
# just field1 and field2
parent.field.child.__fields__('field1', 'field2')
parent.field.child.__fields__(field1=True, field2=True)
```

(continues on next page)

(continued from previous page)

```
# field1 with parameters
parent.field.child.__fields__(field1=dict(param1='value1'))

# all but field2
parent.field.child.__fields__(field2=False)
parent.field.child.__fields__(field2=None)
parent.field.child.__fields__(__exclude__=('field2',))
```

If `__fields__()` is not explicitly called, then all fields are included. Note that this may lead to huge queries since it will result in recursive inclusion of all fields.

Selectors will create selections when items or attributes are accessed, this is done by implicitly calling the selector with empty parameters.

However leafs (ie: scalars) must be **explicitly** called, otherwise they won't generate a selection

```
# OK
parent.field.child()
# NOT OK: doesn't create a selection for child.
parent.field.child
```

__dir__()

Default dir() implementation.

__fields__(**names*, ***names_and_args*)

Select fields of a container type.

See *sgqlc.operation.SelectionList.__fields__()*.

__get_all_fields_selection_list(*depth*, *trail*, *typename*)

Create a new SelectionList, select all fields and return it

__init__(*alias*, *field*, *args*, *typename=None*)

__repr__()

Return repr(self).

__select_all__(*depth*, *trail*, *typename*)

Select all fields in the current SelectionList

__str__()

Return str(self).

class sgqlc.operation.**SelectionList**(*typ*)

Bases: `object`

List of *sgqlc.operation.Selection* in a type.

Warning: Do not create instances directly, use *sgqlc.operation.Operation* instead.

Create a selection list using a type to query its fields. Once fields are accessed, they will create *sgqlc.operation.Selector* object for that field, this allows to match the type structure, with easy to use API:

```
parent.field.child()
parent.field(param1=value1).child()
```

Direct usage example (not recommended):

```
>>> sl = SelectionList(global_schema.Repository)
>>> sl += Selection('x', global_schema.Repository.id, {})
>>> sl # or repr()
{
  x: id
}
>>> print(bytes(sl).decode('utf-8')) # no indentation
{
x: id
}
>>> sl.id    # or any other field from Repository returns a Selector
Selector(field=id)
>>> sl['id'] # also as get item
Selector(field=id)
>>> sl.x    # not the alias
Traceback (most recent call last):
  ...
AttributeError: {
  x: id
} has no field x
>>> sl['x'] #
Traceback (most recent call last):
  ...
KeyError: 'Repository has no field x'
>>> sl.__type__ # returns the type the selection operates on
type Repository {
  id: ID
  name: String!
  owner: Actor!
  issues(titleContains: String, reporterLogin: String, filter: IssuesFilter):␣
↪[Issue!]
}
```

__as__(*typ*)

Create a child selection list on the given type.

The selection list will be result in an inline fragment in the query with an additional query for `__typename`, which is later used to create the proper type when the results are interpreted.

The newly created selection list is shared for all users of the same type in this selection list.

__fields__(**names*, ***names_and_args*)

Select fields of a container type.

This is a helper to automate selection of fields of container types, such as giving a list of names to include, with or without parameters (passed as a mapping `name=args`).

If no arguments are given, all fields are included.

If the keyword argument `__exclude__` is given a list of names, then all but those fields will be included. Alternatively one can exclude fields using `name=None` or `name=False` as keyword argument.

If a list of names is given as positional arguments, then only those names will be included. Alternatively one can include fields using `name=True`. To include fields with selection parameters, then use

name=dict(...) or name=list(...). To include fields without arguments and with aliases, use the shortcut name='alias'.

The special built-in field __typename is not selected by default. In order to select it, provide __typename__=True as a parameter.

```
# just field1 and field2
parent.field.child.__fields__('field1', 'field2')
parent.field.child.__fields__(field1=True, field2=True)

# field1 with parameters
parent.field.child.__fields__(field1=dict(param1='value1'))

# field1 renamed (aliased) to alias1
parent.field.child.__fields__(field1='alias1')

# all but field2
parent.field.child.__fields__(field2=False)
parent.field.child.__fields__(field2=None)
parent.field.child.__fields__(__exclude__=('field2',))

# all and also include __typename
parent.field.child.__fields__(__typename__=True)
parent.field.child.__fields__('__typename__')
```

__init__(*typ*)

__repr__()

Return repr(self).

__str__()

Return str(self).

class sgqlc.operation.**Selector**(*parent*, *field*)

Bases: object

Creates selection for a given field.

> **Warning:** Do not create instances directly, use *sgqlc.operation.SelectionList* instead.

Selectors are callable objects that will create *sgqlc.operation.Selection* entries in the parent *sgqlc.operation.SelectionList*.

Selectors will create selections when items or attributes are accessed, this is done by implicitly calling the selector with empty parameters.

However leafs (ie: scalars) must be **explicitly** called, otherwise they won't generate a selection

```
# OK
parent.field.child()
# NOT OK: doesn't create a selection for child.
parent.field.child
```

To select all fields from a container type, use *sgqlc.operation.Selection.__fields__()*, example:

```
# just field1 and field2
parent.field.child.__fields__('field1', 'field2')
parent.field.child.__fields__(field1=True, field2=True)

# field1 with parameters
parent.field.child.__fields__(field1=dict(param1='value1'))

# all but field2
parent.field.child.__fields__(field2=False)
parent.field.child.__fields__(field2=None)
parent.field.child.__fields__(__exclude__=('field2',))
```

Note: GraphQL limits a single selection per type, as the field name is used in the return object. If you want to select the same field multiple times, like as using different parameters, then provide the `__alias__` parameter to the selector:

```
# FAILS:
parent.field.child(param1='value1')
parent.field.child(param2='value2')

# OK
parent.field.child(param1='value1')
parent.field.child(param2='value2', __alias__='child2')
```

---

property __args__

Shortcut for self.__selection__().__args__

__as__(*typ*)

Create a selection list on the given type.

The selection list will be result in an inline fragment in the query with an additional query for `__typename`, which is later used to create the proper type when the results are interpreted.

__call__(***args*)

Create a selection with the given parameters.

To provide an alias, use `__alias__` keyword argument.

__dir__()

Default dir() implementation.

property __fields__

Calls the selector without arguments, creating a *Selection* instance and return *Selection.__fields__()* method, ready to be called.

To query the actual field this selector operates, use `self.__field__`

__fragment__(*fragment*)

Spread the given fragment in the selection list.

__init__(*parent*, *field*)

__repr__()

Return repr(self).

__selection__(*alias=None*)

Return the selection given its alias

__str__()

Return str(self).

1.8 *sgqlc.endpoint* module

1.8.1 Access GraphQL endpoints using Python

This package provide the following modules:

- *sgqlc.endpoint.base*: with abstract class *sgqlc.endpoint.base.BaseEndpoint* and helpful logging utilities to transform errors into JSON objects.
- *sgqlc.endpoint.http*: concrete *sgqlc.endpoint.http.HTTPEndpoint* using urllib.request.urlopen().
- *sgqlc.endpoint.requests*: concrete *sgqlc.endpoint.requests.RequestsEndpoint* using requests.
- sgqlc.endpoint.websocket: concrete sgqlc.endpoint.websocket.WebSocketEndpoint using websocket._core.create_connection().

Unless otherwise stated the endpoints follow a pattern:

- construct the endpoint giving constants such as address, timeout...
- call the endpoint given an operation and variables

The given variables must be a **plain JSON-serializeable object** (dict with string keys and values being one of dict, list, tuple, str, int, float, bool, None... – json.dumps() is used) and the keys must **match exactly** the variable names (no name conversion is done, no dollar-sign prefix $ should be used).

Example using *sgqlc.endpoint.http.HTTPEndpoint*:

```
#!/usr/bin/env python3

import sys
import json
from sgqlc.endpoint.http import HTTPEndpoint

try:
    token, repo = sys.argv[1:]
except ValueError:
    raise SystemExit('Usage: <token> <team/repo>')

query = '''
query GitHubRepoIssues($repoOwner: String!, $repoName: String!) {
  repository(owner: $repoOwner, name: $repoName) {
    issues(first: 100) {
      nodes {
        number
        title
      }
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
'''

owner, name = repo.split('/', 1)
variables = {
    'repoOwner': owner,
    'repoName': name,
}

url = 'https://api.github.com/graphql'
headers = {
    'Authorization': 'bearer ' + token,
}

endpoint = HTTPEndpoint(url, headers)
data = endpoint(query, variables)

json.dump(data, sys.stdout, sort_keys=True, indent=2, default=str)
```

See more examples.

license
ISC

1.8.2 Sub Modules

- *sgqlc.endpoint.base module*
- *sgqlc.endpoint.http module*
- *sgqlc.endpoint.requests module*
- *sgqlc.endpoint.websocket module*

1.9 *sgqlc.endpoint.base* module

1.9.1 Base Endpoint

Base interface for endpoints.

See concrete implementations:

- *sgqlc.endpoint.http.HTTPEndpoint* using `urllib.request.urlopen()`.
- *sgqlc.endpoint.requests.RequestsEndpoint* using `requests`.
- `sgqlc.endpoint.websocket.WebSocketEndpoint` using `websocket._core.create_connection()`.

Example using *sgqlc.endpoint.http.HTTPEndpoint*:

```
#!/usr/bin/env python3

import sys
```

(continues on next page)

(continued from previous page)

```
import json
from sgqlc.endpoint.http import HTTPEndpoint

try:
    token, repo = sys.argv[1:]
except ValueError:
    raise SystemExit('Usage: <token> <team/repo>')

query = '''
query GitHubRepoIssues($repoOwner: String!, $repoName: String!) {
  repository(owner: $repoOwner, name: $repoName) {
    issues(first: 100) {
      nodes {
        number
        title
      }
    }
  }
}
'''

owner, name = repo.split('/', 1)
variables = {
    'repoOwner': owner,
    'repoName': name,
}

url = 'https://api.github.com/graphql'
headers = {
    'Authorization': 'bearer ' + token,
}

endpoint = HTTPEndpoint(url, headers)
data = endpoint(query, variables)

json.dump(data, sys.stdout, sort_keys=True, indent=2, default=str)
```

See more examples.

license
ISC

class sgqlc.endpoint.base.**BaseEndpoint**

Bases: object

GraphQL endpoint access.

The user of this class should create GraphQL queries and interpret the resulting object, created from JSON data, with top level properties:

Data
object matching the GraphQL requests, or `null` if only errors were returned.

Errors
list of errors, which are objects with the key "message" and optionally others, such as "lo-

cation" (for errors matching GraphQL input). Instead of raising exceptions, such as `json.JSONDecodeError` those are stored in the "exception" key. Subclasses should extend errors providing meaningful messages and extra payload.

Note: Both `data` and `errors` may be returned, for instance if a null-able field fails, it will be returned as null (Python `None`) in data the associated error in the array.

The class has its own `logging.Logger` which is used to debug, info, warning and errors. Note that subclasses may override this logger. Error logging and conversion to uniform data structure similar to GraphQL, with `{"errors": [...]}` is done by *BaseEndpoint._log_json_error()* and *BaseEndpoint._log_graphql_error()* methods. This last one will show the snippets of GraphQL that failed execution.

__call__(*query*, *variables=None*, *operation_name=None*)

Calls the GraphQL endpoint.

Parameters

- **query** (`str` or `bytes`.) – the GraphQL query or mutation to execute. Note that this is converted using `bytes()`, thus one may pass an object implementing `__bytes__()` method to return the query, eventually in more compact form (no indentation, etc).
- **variables** (*dict*) – variables (dict) to use with `query`. This is only useful if the query or mutation contains `$variableName`. Must be a **plain JSON-serializeable object** (dict with string keys and values being one of dict, list, tuple, str, int, float, bool, None… – `json.dumps()` is used) and the keys must **match exactly** the variable names (no name conversion is done, no dollar-sign prefix `$` should be used).
- **operation_name** (*str*) – if more than one operation is listed in `query`, then it should specify the one to be executed.

Returns

dict with optional fields `data` containing the GraphQL returned data as nested dict and `errors` with an array of errors. Note that both `data` and `errors` may be returned!

Return type

dict

Note: Subclasses **must** implement this method, should respect this base signature and may extend with extra parameters such as timeout, extra headers and so on.

__weakref__

list of weak references to the object (if defined)

_fixup_graphql_error(*data*)

Given a possible GraphQL error payload, make sure it's in shape.

This will ensure the given `data` is in the shape:

```
{"errors": [{"message": "some string"}]}
```

If `errors` is not an array, it will be made into a single element array, with the object in that format, with its string representation being the message.

If an element of the `errors` array is not in the format, then it's converted to the format, with its string representation being the message.

The input object is not changed, a copy is made if needed.

Returns
the given `data` formatted to the correct shape, a copy is made and returned if any fix up was needed.

Return type
dict

`_log_graphql_error`(*query*, *data*)

Log a `{"errors": [...]}` GraphQL return and return itself.

Parameters

- **query** (*str*) – the GraphQL query that triggered the result.
- **data** (*dict*) – the decoded JSON object.

Returns
the input `data`

Return type
dict

`_log_json_error`(*body*, *exc*)

Log a `json.JSONDecodeError`, converting to GraphQL's `{"data": null, "errors": [{"message": str(exc)...}]}`

Parameters

- **body** (*str*) – the string with JSON document.
- **exc** (`json.JSONDecodeError`) – the `json.JSONDecodeError`

Returns
GraphQL-compliant dict with keys `data` and `errors`.

Return type
dict

`static snippet`(*code*, *locations*, *sep=' | '*, *colmark=('-', '^')*, *context=5*)

Given a code and list of locations, convert to snippet lines.

return will include line number, a separator (`sep`), then line contents.

At most `context` lines are shown before each location line.

After each location line, the column is marked using `colmark`. The first character is repeated up to column, the second character is used only once.

Returns
list of lines of sources or column markups.

Return type
list

1.10 *sgqlc.endpoint.http* module

1.10.1 Synchronous HTTP Endpoint

This endpoint implements GraphQL client using `urllib.request.urlopen()` or compatible function.

This module provides command line utility:

```
$ python3 -m sgqlc.endpoint.http http://server.com/ '{ queryHere { ... } }'
```

Example using *sgqlc.endpoint.http.HTTPEndpoint*:

```
#!/usr/bin/env python3

import sys
import json
from sgqlc.endpoint.http import HTTPEndpoint

try:
    token, repo = sys.argv[1:]
except ValueError:
    raise SystemExit('Usage: <token> <team/repo>')

query = '''
query GitHubRepoIssues($repoOwner: String!, $repoName: String!) {
  repository(owner: $repoOwner, name: $repoName) {
    issues(first: 100) {
      nodes {
        number
        title
      }
    }
  }
}
'''

owner, name = repo.split('/', 1)
variables = {
    'repoOwner': owner,
    'repoName': name,
}

url = 'https://api.github.com/graphql'
headers = {
    'Authorization': 'bearer ' + token,
}

endpoint = HTTPEndpoint(url, headers)
data = endpoint(query, variables)

json.dump(data, sys.stdout, sort_keys=True, indent=2, default=str)
```

The `query` may be given as `bytes` or `str` as in the example, but it may be a *sgqlc.operation.Operation*, which will serialize as string while also providing convenience to interepret the results.

See more examples.

license
ISC

class sgqlc.endpoint.http.**HTTPEndpoint**(*url*, *base_headers=None*, *timeout=None*, *urlopen=None*, *method='POST'*)

Bases: *BaseEndpoint*

GraphQL access over HTTP.

This helper is very thin, just setups the correct HTTP request to GraphQL endpoint, handling logging of HTTP and GraphQL errors. The object is callable with parameters: `query`, `variables`, `operation_name`, `extra_headers` and `timeout`.

The user of this class should create GraphQL queries and interpret the resulting object, created from JSON data, with top level properties:

Data
object matching the GraphQL requests, or `null` if only errors were returned.

Errors
list of errors, which are objects with the key "message" and optionally others, such as "location" (for errors matching GraphQL input). Instead of raising exceptions, such as `urllib.error.HTTPError` or `json.JSONDecodeError` those are stored in the "exception" key.

Note: Both `data` and `errors` may be returned, for instance if a null-able field fails, it will be returned as null (Python `None`) in data the associated error in the array.

The class has its own `logging.Logger` which is used to debug, info, warning and errors. Error logging and conversion to uniform data structure similar to GraphQL, with `{"errors": [...]}` is done by *HTTPEndpoint._log_http_error()* own method, `BaseEndpoint._log_json_error()` and `BaseEndpoint._log_graphql_error()`. This last one will show the snippets of GraphQL that failed execution.

__call__(*query*, *variables=None*, *operation_name=None*, *extra_headers=None*, *timeout=None*)

Calls the GraphQL endpoint.

Parameters

- **query** (`str` or `bytes`.) – the GraphQL query or mutation to execute. Note that this is converted using `bytes()`, thus one may pass an object implementing `__bytes__()` method to return the query, eventually in more compact form (no indentation, etc).
- **variables** (*dict*) – variables (dict) to use with `query`. This is only useful if the query or mutation contains `$variableName`.
- **operation_name** (*str*) – if more than one operation is listed in `query`, then it should specify the one to be executed.
- **extra_headers** (*dict*) – dict with extra HTTP headers to use.
- **timeout** (*float*) – overrides the default timeout.

Returns
dict with optional fields `data` containing the GraphQL returned data as nested dict and `errors` with an array of errors. Note that both `data` and `errors` may be returned!

Return type
dict

__init__(*url*, *base_headers=None*, *timeout=None*, *urlopen=None*, *method='POST'*)

Parameters

- **url** (*str*) – the default GraphQL endpoint url.
- **base_headers** (*dict*) – the base HTTP headers to include in every request.
- **timeout** (*float*) – timeout in seconds to use with `urllib.request.urlopen()`. Optional (`None` uses default timeout).
- **urlopen** – function that implements the same interface as `urllib.request.urlopen()`, which is used by default.

__str__()

Return str(self).

_log_http_error(*query*, *req*, *exc*)

Log `urllib.error.HTTPError`, converting to GraphQL's `{"data": null, "errors": [{"message": str(exc)...}]}`

Parameters

- **query** (*str*) – the GraphQL query that triggered the result.
- **req** (`urllib.request.Request`) – `urllib.request.Request` instance that was opened.
- **exc** (`urllib.error.HTTPError`) – `urllib.error.HTTPError` instance

Returns

GraphQL-compliant dict with keys `data` and `errors`.

Return type

dict

1.11 *sgqlc.endpoint.requests* module

1.11.1 Synchronous HTTP Endpoint using python-requests

This endpoint implements GraphQL client using the `requests` library.

This module provides command line utility:

```
$ python3 -m sgqlc.endpoint.requests http://server.com/ '{ query { ... } }'
```

It's pretty much like *sgqlc.endpoint.http.HTTPEndpoint*, but using the `requests`. This comes with the convenience to use a `requests.Session` and `requests.auth` compatible authentication option (Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth), which is useful when third party libraries offer such helpers (ex: requests-aws).

Example using *sgqlc.endpoint.requests.RequestsEndpoint*:

```
#!/usr/bin/env python3

import sys
import json
from sgqlc.endpoint.requests import RequestsEndpoint
```

(continues on next page)

(continued from previous page)

```
try:
    token, repo = sys.argv[1:]
except ValueError:
    raise SystemExit('Usage: <token> <team/repo>')

query = '''
query GitHubRepoIssues($repoOwner: String!, $repoName: String!) {
  repository(owner: $repoOwner, name: $repoName) {
    issues(first: 100) {
      nodes {
        number
        title
      }
    }
  }
}
'''

owner, name = repo.split('/', 1)
variables = {
    'repoOwner': owner,
    'repoName': name,
}

url = 'https://api.github.com/graphql'
headers = {
    'Authorization': 'bearer ' + token,
}

endpoint = RequestsEndpoint(url, headers)
data = endpoint(query, variables)

json.dump(data, sys.stdout, sort_keys=True, indent=2, default=str)
```

The `query` may be given as `bytes` or `str` as in the example, but it may be a *sgqlc.operation.Operation*, which will serialize as string while also providing convenience to interepret the results.

See more examples.

license
ISC

class sgqlc.endpoint.requests.**RequestsEndpoint**(*url*, *base_headers=None*, *timeout=None*, *method='POST'*, *auth=None*, *session=None*)

Bases: *BaseEndpoint*

GraphQL access over HTTP.

This helper is very thin, just setups the correct HTTP request to GraphQL endpoint, handling logging of HTTP and GraphQL errors. The object is callable with parameters: `query`, `variables`, `operation_name`, `extra_headers` and `timeout`.

The user of this class should create GraphQL queries and interpret the resulting object, created from JSON data, with top level properties:

Data
object matching the GraphQL requests, or `null` if only errors were returned.

Errors
list of errors, which are objects with the key "message" and optionally others, such as "location" (for errors matching GraphQL input). Instead of raising exceptions, such as `requests.exceptions.HTTPError` or `json.JSONDecodeError` those are stored in the "exception" key.

Note: Both `data` and `errors` may be returned, for instance if a null-able field fails, it will be returned as null (Python `None`) in data the associated error in the array.

The class has its own `logging.Logger` which is used to debug, info, warning and errors. Error logging and conversion to uniform data structure similar to GraphQL, with `{"errors": [...]}` is done by *RequestsEndpoint._log_http_error()* own method, `BaseEndpoint._log_json_error()` and `BaseEndpoint._log_graphql_error()`. This last one will show the snippets of GraphQL that failed execution.

`__call__`(*query*, *variables=None*, *operation_name=None*, *extra_headers=None*, *timeout=None*)

Calls the GraphQL endpoint.

Parameters

- **query** (`str` or `bytes`.) – the GraphQL query or mutation to execute. Note that this is converted using `bytes()`, thus one may pass an object implementing `__bytes__()` method to return the query, eventually in more compact form (no indentation, etc).
- **variables** (*dict*) – variables (dict) to use with `query`. This is only useful if the query or mutation contains `$variableName`. Must be a **plain JSON-serializeable object** (dict with string keys and values being one of dict, list, tuple, str, int, float, bool, None… – `json.dumps()` is used) and the keys must **match exactly** the variable names (no name conversion is done, no dollar-sign prefix `$` should be used).
- **operation_name** (*str*) – if more than one operation is listed in `query`, then it should specify the one to be executed.
- **extra_headers** (*dict*) – dict with extra HTTP headers to use.
- **timeout** (*float*) – overrides the default timeout.

Returns
dict with optional fields `data` containing the GraphQL returned data as nested dict and `errors` with an array of errors. Note that both `data` and `errors` may be returned!

Return type
dict

`__init__`(*url*, *base_headers=None*, *timeout=None*, *method='POST'*, *auth=None*, *session=None*)

Parameters

- **url** (*str*) – the default GraphQL endpoint url.
- **base_headers** (*dict*) – the base HTTP headers to include in every request.
- **timeout** (*float*) – timeout in seconds to use with `urllib.request.urlopen()`. Optional (`None` uses default timeout).
- **method** – HTTP Method to use for the request, *POST* is used by default
- **auth** – `requests.auth` compatible authentication option. Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth. Optional.

- **session** – requests.Session object. Optional.

__str__()

Return str(self).

_log_http_error(*query*, *req*, *exc*)

Log requests.exceptions.HTTPError, converting to GraphQL's {"data": null, "errors": [{"message": str(exc)...}]}

Parameters

- **query** (str) – the GraphQL query that triggered the result.
- **req** (requests.Request) – requests.Request instance that was opened.
- **exc** (requests.exceptions.HTTPError) – requests.exceptions.HTTPError instance

Returns

GraphQL-compliant dict with keys data and errors.

Return type

dict

1.12 *sgqlc.endpoint.websocket* module

1.13 *sgqlc.introspection* module

1.13.1 Introspection

Provides the standard GraphQL Introspection Query, same as https://github.com/graphql/graphql-js/blob/master/src/utilities/introspectionQuery.js however allows to choose whether to include descriptions and deprecated fields.

Downloading schema.json

Usually services provide a schema.json file with the introspection results or offer a development server where the introspection query can be executed and saved as JSON:

```
python3 \
    -m sgqlc.introspection \
    --exclude-deprecated \
    -H "Authorization: bearer ${TOKEN}" \
    https://server.com/graphql \
    schema.json
```

If the descriptions are not needed, then they can be excluded (saves bandwith and space):

```
python3 \
    -m sgqlc.introspection \
    --exclude-deprecated \
    --exclude-description \
    -H "Authorization: bearer ${TOKEN}" \
    https://server.com/graphql \
    schema.json
```

license
ISC

sgqlc.introspection.**variables**(*include_description=True*, *include_deprecated=True*)

Return variables to be used with `IntrospectionQuery` GraphQL operation.

Parameters

- **include_description** (*bool*) – If field and type descriptions (documentation) should be included in the query.
- **include_deprecated** (*bool*) – If deprecated fields and enumeration values should be included. If so, then two extra fields will be added: `isDeprecated: Boolean` and `deprecationReason: String`.

Returns
dict with GraphQL variables.

Return type
dict

1.13.2 sgqlc.introspection Command Line Options

Introspect a GraphQL endpoint using HTTP

```
usage: python3 -m sgqlc.introspection [-h] [--exclude-deprecated]
                                      [--exclude-description]
                                      [--header HEADER] [--timeout TIMEOUT]
                                      [--verbose]
                                      url [outfile]
```

Positional Arguments

url GraphQL endpoint address.

outfile Where to write json. Default=stdout

Default: <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>

Named Arguments

--exclude-deprecated If given, will exclude deprecated fields and enumeration values.

Default: False

--exclude-description If given, will exclude description (documentation).

Default: False

--header, -H NAME=VALUE HTTP header to send. Example: "Authorization: bearer ${token}"

Default: []

--timeout, -t Request timeout, in seconds.

--verbose, -v Increase verbosity

Default: 0

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

license
ISC

PYTHON MODULE INDEX

S

INDEX

Symbols

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

U

V