GraphQL-core 3 Documentation

Release 3.3.0a5

Christoph Zwerschke

Apr 14, 2024

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Introduction

GraphQL-core-3 is a Python port of GraphQL.js, the JavaScript reference implementation for GraphQL, a query language for APIs created by Facebook.

GraphQL consists of three parts:

- A type system that you define
- A query language that you use to query the API
- An execution and validation engine

The reference implementation closely follows the Specification for GraphQL which consists of the following sections:

- Language
- Type System
- Introspection
- Validation
- Execution
- Response

This division into subsections is reflected in the *Sub-Packages* of GraphQL-core 3. Each of these sub-packages implements the aspects specified in one of the sections of the specification.

1.1.1 Getting started

You can install GraphQL-core 3 using pip:

```
pip install graphql-core
```

You can also install GraphQL-core 3 with poetry, if you prefer that:

```
poetry install
```

Now you can start using GraphQL-core 3 by importing from the top-level `graphql` package. Nearly everything defined in the sub-packages can also be imported directly from the top-level package.

For instance, using the types defined in the `graphql.type` package, you can define a GraphQL schema, like this simple one:

```
from graphql import (
    GraphQLSchema, GraphQLObjectType, GraphQLField, GraphQLString)

schema = GraphQLSchema(
    query=GraphQLObjectType(
        name='RootQueryType',
        fields={
            'hello': GraphQLField(
                GraphQLString,
                resolve=lambda obj, info: 'world')
        }))
```

The *graphql.execution* package implements the mechanism for executing GraphQL queries. The top-level *graphql()* and *graphql_sync()* functions also parse and validate queries using the *graphql.language* and *graphql.validation* modules.

So to validate and execute a query against our simple schema, you can do:

```
from graphql import graphql_sync

query = '{ hello }'

print(graphql_sync(schema, query))
```

This will yield the following output:

```
ExecutionResult(data={'hello': 'world'}, errors=None)
```

1.1.2 Reporting Issues and Contributing

Please visit the GitHub repository of GraphQL-core 3 if you're interested in the current development or want to report issues or send pull requests.

1.2 Usage

GraphQL-core provides two important capabilities: building a type schema, and serving queries against that type schema.

1.2.1 Building a Type Schema

Using the classes in the *graphql.type* sub-package as building blocks, you can build a complete GraphQL type schema.

Let's take the following schema as an example, which will allow us to query our favorite heroes from the Star Wars trilogy:

```
enum Episode { NEWHOPE, EMPIRE, JEDI }

interface Character {
  id: String!
```

(continues on next page)

(continued from previous page)

```
  name: String
  friends: [Character]
  appearsIn: [Episode]
}

type Human implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  homePlanet: String
}

type Droid implements Character {
  id: String!
  name: String
  friends: [Character]
  appearsIn: [Episode]
  primaryFunction: String
}

type Query {
  hero(episode: Episode): Character
  human(id: String!): Human
  droid(id: String!): Droid
}
```

We have been using the so called GraphQL schema definition language (SDL) here to describe the schema. While it is also possible to build a schema directly from this notation using GraphQL-core 3, let's first create that schema manually by assembling the types defined here using Python classes, adding resolver functions written in Python for querying the data.

First, we need to import all the building blocks from the *graphql.type* sub-package. Note that you don't need to import from the sub-packages, since nearly everything is also available directly in the top *graphql* package:

```
from graphql import (
    GraphQLArgument, GraphQLEnumType, GraphQLEnumValue,
    GraphQLField, GraphQLInterfaceType, GraphQLList, GraphQLNonNull,
    GraphQLObjectType, GraphQLSchema, GraphQLString)
```

Next, we need to build the enum type `Episode`:

```
episode_enum = GraphQLEnumType('Episode', {
    'NEWHOPE': GraphQLEnumValue(4, description='Released in 1977.'),
    'EMPIRE': GraphQLEnumValue(5, description='Released in 1980.'),
    'JEDI': GraphQLEnumValue(6, description='Released in 1983.')
    }, description='One of the films in the Star Wars Trilogy')
```

If you don't need the descriptions for the enum values, you can also define the enum type like this using an underlying Python `Enum` type:

```
from enum import Enum
```

(continues on next page)

(continued from previous page)

```
class EpisodeEnum(Enum):
    NEWHOPE = 4
    EMPIRE = 5
    JEDI = 6

episode_enum = GraphQLEnumType(
    'Episode', EpisodeEnum,
    description='One of the films in the Star Wars Trilogy')
```

You can also use a Python dictionary instead of a Python `Enum` type to define the GraphQL enum type:

```
episode_enum = GraphQLEnumType(
    'Episode', {'NEWHOPE': 4, 'EMPIRE': 5, 'JEDI': 6},
    description='One of the films in the Star Wars Trilogy')
```

Our schema also contains a `Character` interface. Here is how we build it:

```
character_interface = GraphQLInterfaceType('Character', lambda: {
    'id': GraphQLField(
        GraphQLNonNull(GraphQLString),
        description='The id of the character.'),
    'name': GraphQLField(
        GraphQLString,
        description='The name of the character.'),
    'friends': GraphQLField(
        GraphQLList(character_interface),
        description='The friends of the character,'
                    ' or an empty list if they have none.'),
    'appearsIn': GraphQLField(
        GraphQLList(episode_enum),
        description='Which movies they appear in.'),
    'secretBackstory': GraphQLField(
        GraphQLString,
        description='All secrets about their past.')},
    resolve_type=get_character_type,
    description='A character in the Star Wars Trilogy')
```

Note that we did not pass the dictionary of fields to the `GraphQLInterfaceType` directly, but using a lambda function (a so-called "thunk"). This is necessary because the fields are referring back to the character interface that we are just defining. Whenever you have such recursive definitions in GraphQL-core, you need to use thunks. Otherwise, you can pass everything directly.

Characters in the Star Wars trilogy are either humans or droids. So we define a `Human` and a `Droid` type, which both implement the `Character` interface:

```
human_type = GraphQLObjectType('Human', lambda: {
    'id': GraphQLField(
        GraphQLNonNull(GraphQLString),
        description='The id of the human.'),
    'name': GraphQLField(
        GraphQLString,
        description='The name of the human.'),
    'friends': GraphQLField(
```

(continues on next page)

(continued from previous page)

```
        GraphQLList(character_interface),
        description='The friends of the human,'
                    ' or an empty list if they have none.',
        resolve=get_friends),
    'appearsIn': GraphQLField(
        GraphQLList(episode_enum),
        description='Which movies they appear in.'),
    'homePlanet': GraphQLField(
        GraphQLString,
        description='The home planet of the human, or null if unknown.'),
    'secretBackstory': GraphQLField(
        GraphQLString,
        resolve=get_secret_backstory,
        description='Where are they from'
                    ' and how they came to be who they are.')},
    interfaces=[character_interface],
    description='A humanoid creature in the Star Wars universe.')

droid_type = GraphQLObjectType('Droid', lambda: {
    'id': GraphQLField(
        GraphQLNonNull(GraphQLString),
        description='The id of the droid.'),
    'name': GraphQLField(
        GraphQLString,
        description='The name of the droid.'),
    'friends': GraphQLField(
        GraphQLList(character_interface),
        description='The friends of the droid,'
                    ' or an empty list if they have none.',
        resolve=get_friends,
    ),
    'appearsIn': GraphQLField(
        GraphQLList(episode_enum),
        description='Which movies they appear in.'),
    'secretBackstory': GraphQLField(
        GraphQLString,
        resolve=get_secret_backstory,
        description='Construction date and the name of the designer.'),
    'primaryFunction': GraphQLField(
        GraphQLString,
        description='The primary function of the droid.')
    },
    interfaces=[character_interface],
    description='A mechanical creature in the Star Wars universe.')
```

Now that we have defined all used result types, we can construct the `Query` type for our schema:

```
query_type = GraphQLObjectType('Query', lambda: {
    'hero': GraphQLField(character_interface, args={
        'episode': GraphQLArgument(episode_enum, description=(
            'If omitted, returns the hero of the whole saga.'
            ' If provided, returns the hero of that particular episode.'))},
```

(continues on next page)

(continued from previous page)

```
        resolve=get_hero),
    'human': GraphQLField(human_type, args={
        'id': GraphQLArgument(
            GraphQLNonNull(GraphQLString), description='id of the human')},
        resolve=get_human),
    'droid': GraphQLField(droid_type, args={
        'id': GraphQLArgument(
            GraphQLNonNull(GraphQLString), description='id of the droid')},
        resolve=get_droid)})
```

Using our query type we can define our schema:

```
schema = GraphQLSchema(query_type)
```

Note that you can also pass a mutation type and a subscription type as additional arguments to the *GraphQLSchema*.

1.2.2 Implementing the Resolver Functions

Before we can execute queries against our schema, we also need to define the data (the humans and droids appearing in the Star Wars trilogy) and implement resolver functions that fetch the data (at the beginning of our schema module, because we are referencing them later):

```
luke = dict(
    id='1000', name='Luke Skywalker', homePlanet='Tatooine',
    friends=['1002', '1003', '2000', '2001'], appearsIn=[4, 5, 6])

vader = dict(
    id='1001', name='Darth Vader', homePlanet='Tatooine',
    friends=['1004'], appearsIn=[4, 5, 6])

han = dict(
    id='1002', name='Han Solo', homePlanet=None,
    friends=['1000', '1003', '2001'], appearsIn=[4, 5, 6])

leia = dict(
    id='1003', name='Leia Organa', homePlanet='Alderaan',
    friends=['1000', '1002', '2000', '2001'], appearsIn=[4, 5, 6])

tarkin = dict(
    id='1004', name='Wilhuff Tarkin', homePlanet=None,
    friends=['1001'], appearsIn=[4])

human_data = {
    '1000': luke, '1001': vader, '1002': han, '1003': leia, '1004': tarkin}

threepio = dict(
    id='2000', name='C-3PO', primaryFunction='Protocol',
    friends=['1000', '1002', '1003', '2001'], appearsIn=[4, 5, 6])

artoo = dict(
    id='2001', name='R2-D2', primaryFunction='Astromech',
```

(continues on next page)

(continued from previous page)

```
    friends=['1000', '1002', '1003'], appearsIn=[4, 5, 6])

droid_data = {
    '2000': threepio, '2001': artoo}


def get_character_type(character, _info, _type):
        return 'Droid' if character['id'] in droid_data else 'Human'


def get_character(id):
    """Helper function to get a character by ID."""
    return human_data.get(id) or droid_data.get(id)


def get_friends(character, _info):
    """Allows us to query for a character's friends."""
    return map(get_character, character.friends)


def get_hero(root, _info, episode):
    """Allows us to fetch the undisputed hero of the trilogy, R2-D2."""
    if episode == 5:
        return luke  # Luke is the hero of Episode V
    return artoo  # Artoo is the hero otherwise


def get_human(root, _info, id):
    """Allows us to query for the human with the given id."""
    return human_data.get(id)


def get_droid(root, _info, id):
    """Allows us to query for the droid with the given id."""
    return droid_data.get(id)


def get_secret_backstory(_character, _info):
    """Raise an error when attempting to get the secret backstory."""
    raise RuntimeError('secretBackstory is secret.')
```

Note that the resolver functions get the current object as first argument. For a field on the root Query type this is often not used, but a root object can also be defined when executing the query. As the second argument, they get an object containing execution information, as defined in the *GraphQLResolveInfo* class. This object also has a `context` attribute that can be used to provide every resolver with contextual information like the currently logged in user, or a database session. In our simple example we don't authenticate users and use static data instead of a database, so we don't make use of it here. In addition to these two arguments, resolver functions optionally get the defined for the field in the schema, using the same names (the names are not translated from GraphQL naming conventions to Python naming conventions).

Also note that you don't need to provide resolvers for simple attribute access or for fetching items from Python dictionaries.

Finally, note that our data uses the internal values of the `Episode` enum that we have defined above, not the descriptive

enum names that are used externally. For example, NEWHOPE ("A New Hope") has internally the actual episode number 4 as value.

1.2.3 Executing Queries

Now that we have defined the schema and breathed life into it with our resolver functions, we can execute arbitrary query against the schema.

The *graphql* package provides the *graphql.graphql()* function to execute queries. This is the main feature of GraphQL-core.

Note however that this function is actually a coroutine intended to be used in asynchronous code running in an event loop.

Here is one way to use it:

```
import asyncio
from graphql import graphql

async def query_artoo():
    result = await graphql(schema, """
    {
      droid(id: "2001") {
        name
        primaryFunction
      }
    }
    """)
    print(result)

asyncio.run(query_artoo())
```

In our query, we asked for the droid with the id 2001, which is R2-D2, and its primary function, Astromech. When everything has been implemented correctly as shown above, you should get the expected result:

```
ExecutionResult(
    data={'droid': {'name': 'R2-D2', 'primaryFunction': 'Astromech'}},
    errors=None)
```

The *ExecutionResult* has a data attribute with the actual result, and an errors attribute with a list of errors if there were any.

If all your resolvers work synchronously, as in our case, you can also use the *graphql.graphql_sync()* function to query the result in ordinary synchronous code:

```
from graphql import graphql_sync

result = graphql_sync(schema, """
  query FetchHuman($id: String!) {
    human(id: $id) {
      name
      homePlanet
    }
  }
```

(continues on next page)

(continued from previous page)

```
  """, variable_values={'id': '1000'})
print(result)
```

Here we asked for the human with the id 1000, Luke Skywalker, and his home planet, Tatooine. So the output of the code above is:

```
ExecutionResult(
    data={'human': {'name': 'Luke Skywalker', 'homePlanet': 'Tatooine'}},
    errors=None)
```

Let's see what happens when we make a mistake in the query, by querying a non-existing `homeTown` field:

```
result = graphql_sync(schema, """
    {
      human(id: "1000") {
        name
        homePlace
      }
    }
    """)
print(result)
```

You will get the following result as output:

```
ExecutionResult(data=None, errors=[GraphQLError(
    "Cannot query field 'homePlace' on type 'Human'. Did you mean 'homePlanet'?",
    locations=[SourceLocation(line=5, column=9)])])
```

This is very helpful. Not only do we get the exact location of the mistake in the query, but also a suggestion for correcting the bad field name.

GraphQL also allows to request the meta field `__typename`. We can use this to verify that the hero of "The Empire Strikes Back" episode is Luke Skywalker and that he is in fact a human:

```
result = graphql_sync(schema, """
    {
      hero(episode: EMPIRE) {
        __typename
        name
      }
    }
    """)
print(result)
```

This gives the following output:

```
ExecutionResult(
    data={'hero': {'__typename': 'Human', 'name': 'Luke Skywalker'}},
    errors=None)
```

Finally, let's see what happens when we try to access the secret backstory of our hero:

```
result = graphql_sync(schema, """
    {
```

(continues on next page)

(continued from previous page)

```
      hero(episode: EMPIRE) {
        name
        secretBackstory
      }
    }
    """)
print(result)
```

While we get the name of the hero, the secret backstory fields remains empty, since its resolver function raises an error. However, we get the error that has been raised by the resolver in the `errors` attribute of the result:

```
ExecutionResult(
    data={'hero': {'name': 'Luke Skywalker', 'secretBackstory': None}},
    errors=[GraphQLError('secretBackstory is secret.',
            locations=[SourceLocation(line=5, column=9)],
            path=['hero', 'secretBackstory'])])
```

1.2.4 Using the Schema Definition Language

Above we defined the GraphQL schema as Python code, using the *GraphQLSchema* class and other classes representing the various GraphQL types.

GraphQL-core 3 also provides a language-agnostic way of defining a GraphQL schema using the GraphQL schema definition language (SDL) which is also part of the GraphQL specification. To do this, we simply feed the SDL as a string to the *build_schema()* function in *graphql.utilities*:

```
from graphql import build_schema

schema = build_schema("""

    enum Episode { NEWHOPE, EMPIRE, JEDI }

    interface Character {
      id: String!
      name: String
      friends: [Character]
      appearsIn: [Episode]
    }

    type Human implements Character {
      id: String!
      name: String
      friends: [Character]
      appearsIn: [Episode]
      homePlanet: String
    }

    type Droid implements Character {
      id: String!
      name: String
      friends: [Character]
```

(continues on next page)

(continued from previous page)

```
    appearsIn: [Episode]
    primaryFunction: String
  }

  type Query {
    hero(episode: Episode): Character
    human(id: String!): Human
    droid(id: String!): Droid
  }
  """)
```

The result is a *GraphQLSchema* object just like the one we defined above, except for the resolver functions which cannot be defined in the SDL.

We would need to manually attach these functions to the schema, like so:

```
schema.query_type.fields['hero'].resolve = get_hero
schema.get_type('Character').resolve_type = get_character_type
```

Another problem is that the SDL does not define the server side values of the `Episode` enum type which are returned by the resolver functions and which are different from the names used for the episode.

So we would also need to manually define these values, like so:

```
for name, value in schema.get_type('Episode').values.items():
    value.value = EpisodeEnum[name].value
```

This would allow us to query the schema built from SDL just like the manually assembled schema:

```
from graphql import graphql_sync

result = graphql_sync(schema, """
    {
      hero(episode: EMPIRE) {
        name
        appearsIn
      }
    }
    """)
print(result)
```

And we would get the expected result:

```
ExecutionResult(
    data={'hero': {'name': 'Luke Skywalker',
                   'appearsIn': ['NEWHOPE', 'EMPIRE', 'JEDI']}},
    errors=None)
```

1.2.5 Using resolver methods

Above we have attached resolver functions to the schema only. However, it is also possible to define resolver methods on the resolved objects, starting with the `root_value` object that you can pass to the *graphql()* function when executing a query.

In our case, we could create a `Root` class with three methods as root resolvers, like so:

```
class Root:
    """The root resolvers"""

    def hero(self, info, episode):
        return luke if episode == 5 else artoo

    def human(self, info, id):
        return human_data.get(id)

    def droid(self, info, id):
        return droid_data.get(id)
```

Since we have defined synchronous methods only, we will use the *graphql_sync()* function to execute a query, passing a `Root()` object as the `root_value`:

```
from graphql import graphql_sync

result = graphql_sync(schema, """
    {
      droid(id: "2001") {
        name
        primaryFunction
      }
    }
    """, Root())
print(result)
```

Even if we haven't attached a resolver to the `hero` field as we did above, this would now still resolve and give the following output:

```
ExecutionResult(
    data={'droid': {'name': 'R2-D2', 'primaryFunction': 'Astromech'}},
    errors=None)
```

Of course you can also define asynchronous methods as resolvers, and execute queries asynchronously with *graphql()*.

In a similar vein, you can also attach resolvers as methods to the resolved objects on deeper levels than the root of the query. In that case, instead of resolving to dictionaries with keys for all the fields, as we did above, you would resolve to objects with attributes for all the fields. For instance, you would define a class `Human` with a method `friends()` for resolving the friends of a human. You can also make use of inheritance in this case. The `Human` class and a `Droid` class could inherit from a `Character` class and use its methods as resolvers for common fields.

1.2.6 Using an Introspection Query

A third way of building a schema is using an introspection query on an existing server. This is what GraphiQL uses to get information about the schema on the remote server. You can create an introspection query using GraphQL-core 3 with the *get_introspection_query()* function:

```
from graphql import get_introspection_query

query = get_introspection_query(descriptions=True)
```

This will also yield the descriptions of the introspected schema fields. You can also create a query that omits the descriptions with:

```
query = get_introspection_query(descriptions=False)
```

In practice you would run this query against a remote server, but we can also run it against the schema we have just built above:

```
from graphql import graphql_sync

introspection_query_result = graphql_sync(schema, query)
```

The `data` attribute of the introspection query result now gives us a dictionary, which constitutes a third way of describing a GraphQL schema:

```
{'__schema': {
    'queryType': {'name': 'Query'},
    'mutationType': None, 'subscriptionType': None,
    'types': [
        {'kind': 'OBJECT', 'name': 'Query', 'description': None,
         'fields': [{
            'name': 'hero', 'description': None,
            'args': [{'name': 'episode', 'description': ... }],
            ... }, ... ], ... },
        ... ],
    ... }
}
```

This result contains all the information that is available in the SDL description of the schema, i.e. it does not contain the resolve functions and information on the server-side values of the enum types.

You can convert the introspection result into `GraphQLSchema` with GraphQL-core 3 by using the *build_client_schema()* function:

```
from graphql import build_client_schema

client_schema = build_client_schema(introspection_query_result.data)
```

It is also possible to convert the result to SDL with GraphQL-core 3 by using the *print_schema()* function:

```
from graphql import print_schema

sdl = print_schema(client_schema)
print(sdl)
```

This prints the SDL representation of the schema that we started with.

As you see, it is easy to convert between the three forms of representing a GraphQL schema in GraphQL-core 3 using the *graphql.utilities* module.

1.2.7 Parsing GraphQL Queries and Schema Notation

When executing GraphQL queries, the first step that happens under the hood is parsing the query. But GraphQL-core 3 also exposes the parser for direct usage via the *parse()* function. When you pass this function a GraphQL source code, it will be parsed and returned as a Document, i.e. an abstract syntax tree (AST) of *Node* objects. The root node will be a *DocumentNode*, with child nodes of different kinds corresponding to the GraphQL source. The nodes also carry information on the location in the source code that they correspond to.

Here is an example:

```
from graphql import parse

document = parse("""
    type Query {
      me: User
    }

    type User {
      id: ID
      name: String
    }
""")
```

You can also leave out the information on the location in the source code when creating the AST document:

```
document = parse(..., no_location=True)
```

This will give the same result as manually creating the AST document:

```
from graphql.language.ast import *

document = DocumentNode(definitions=[
    ObjectTypeDefinitionNode(
        name=NameNode(value='Query'),
        fields=[
            FieldDefinitionNode(
                name=NameNode(value='me'),
                type=NamedTypeNode(name=NameNode(value='User')),
                arguments=[], directives=[])
            ], directives=[], interfaces=[]),
    ObjectTypeDefinitionNode(
        name=NameNode(value='User'),
        fields=[
            FieldDefinitionNode(
                name=NameNode(value='id'),
                type=NamedTypeNode(
                    name=NameNode(value='ID')),
                arguments=[], directives=[]),
            FieldDefinitionNode(
                name=NameNode(value='name'),
```

(continues on next page)

(continued from previous page)

```
                type=NamedTypeNode(
                    name=NameNode(value='String')),
                arguments=[], directives=[]),
            ], directives=[], interfaces=[]),
    ])
```

When parsing with `no_location=False` (the default), the AST nodes will also have a `loc` attribute carrying the information on the source code location corresponding to the AST nodes.

When there is a syntax error in the GraphQL source code, then the *parse()* function will raise a *GraphQLSyntaxError*.

The parser can not only be used to parse GraphQL queries, but also to parse the GraphQL schema definition language. This will result in another way of representing a GraphQL schema, as an AST document.

1.2.8 Extending a Schema

With GraphQL-core 3 you can also extend a given schema using type extensions. For example, we might want to add a `lastName` property to our `Human` data type to retrieve only the last name of the person.

This can be achieved with the *extend_schema()* function as follows:

```
from graphql import extend_schema, parse

schema = extend_schema(schema, parse("""
    extend type Human {
     lastName: String
    }
    """))
```

Note that this function expects the extensions as an AST, which we can get using the *parse()* function. Also note that the *extend_schema()* function does not alter the original schema, but returns a new schema object.

We also need to attach a resolver function to the new field:

```
def get_last_name(human, info):
    return human['name'].rsplit(None, 1)[-1]

schema.get_type('Human').fields['lastName'].resolve = get_last_name
```

Now we can query only the last name of a human:

```
from graphql import graphql_sync

result = graphql_sync(schema, """
    {
      human(id: "1000") {
        lastName
        homePlanet
      }
    }
    """)
print(result)
```

This query will give the following result:

```
ExecutionResult(
    data={'human': {'lastName': 'Skywalker', 'homePlanet': 'Tatooine'}},
    errors=None)
```

1.2.9 Validating GraphQL Queries

When executing GraphQL queries, the second step that happens under the hood after parsing the source code is a validation against the given schema using the rules of the GraphQL specification. You can also run the validation step manually by calling the *validate()* function, passing the schema and the AST document:

```
from graphql import parse, validate

errors = validate(schema, parse("""
    {
      human(id: NEWHOPE) {
        name
        homePlace
        friends
      }
    }
    """))
```

As a result, you will get a complete list of all errors that the validators has found. In this case, we will get the following three validation errors:

```
[GraphQLError(
    'String cannot represent a non string value: NEWHOPE',
    locations=[SourceLocation(line=3, column=17)]),
 GraphQLError(
    "Cannot query field 'homePlace' on type 'Human'."
     " Did you mean 'homePlanet'?",
     locations=[SourceLocation(line=5, column=9)]),
 GraphQLError(
    "Field 'friends' of type '[Character]' must have a selection of subfields."
     "  Did you mean 'friends { ... }'?",
     locations=[SourceLocation(line=6, column=9)])]
```

These rules are available in the *specified_rules* list and implemented in the *graphql.validation.rules* subpackage. Instead of the default rules, you can also use a subset or create custom rules. The rules are based on the *ValidationRule* class which is based on the *Visitor* class which provides a way of walking through an AST document using the visitor pattern.

1.2.10 Subscriptions

Sometimes you need to not only query data from a server, but you also want to push data from the server to the client. GraphQL-core 3 has you also covered here, because it implements the "Subscribe" algorithm described in the GraphQL spec. To execute a GraphQL subscription, you must use the *`subscribe()`* method from the *`graphql.execution`* package. Instead of a single *`ExecutionResult`*, this function returns an asynchronous iterator yielding a stream of those, unless there was an immediate error. Of course you will then also need to maintain a persistent channel to the client (often realized via WebSockets) to push these results back.

1.2.11 Other Usages

GraphQL-core 3 provides many more low-level functions that can be used to work with GraphQL schemas and queries. We encourage you to explore the contents of the various *Sub-Packages*, particularly *`graphql.utilities`*, and to look into the source code and tests of GraphQL-core 3 in order to find all the functionality that is provided and understand it in detail.

1.3 Differences from GraphQL.js

The goal of GraphQL-core 3 is to be a faithful replication of GraphQL.js, the JavaScript reference implementation for GraphQL, in Python 3, and to keep it aligned and up to date with the ongoing development of GraphQL.js. Therefore, we strive to be as compatible as possible to the original JavaScript library, sometimes at the cost of being less Pythonic than other libraries written particularly for Python. We also avoid incorporating additional features that do not exist in the JavaScript library, in order to keep the task of maintaining the Python code and keeping it in line with the JavaScript code manageable. The preferred way of getting new features into GraphQL-core is to propose and discuss them on the GraphQL.js issue tracker first, try to get them included into GraphQL.js, and from there ported to GraphQL-core.

Having said this, in a few places we allowed the API to be a bit more Pythonic than the direct equivalent would have been. We also added a few features that do not exist in the JavaScript library, mostly to support existing higher level libraries such as Graphene and the different naming conventions in Python. The most notable differences are the following:

1.3.1 Direct attribute access in GraphQL types

You can access

- the **fields** of GraphQLObjectTypes, GraphQLInterfaceTypes and GraphQLInputObjectTypes,
- the **interfaces** of GraphQLObjectTypes,
- the **types** of GraphQLUnionTypes,
- the **values** of GraphQLEnumTypes and
- the **query**, **mutation**, **subscription** and **type_map** of GraphQLSchemas

directly as attributes, instead of using getters.

For example, to get the fields of a GraphQLObjectType `obj`, you write `obj.fields` instead of `obj.getFields()`.

1.3.2 Arguments, fields and values are dictionaries

- The **arguments** of GraphQLDirectives and GraphQLFields,
- the **fields** of GraphQLObjectTypes, GraphQLInterfaceTypes and GraphQLInputObjectTypes, and
- the **values** of GraphQLEnumTypes

are always Python dictionaries in GraphQL-core, while they are returned as Arrays in GraphQL.js. Also, the values of these dictionaries do not have `name` attributes, since the names are already used as the keys of these dictionaries.

1.3.3 Shorthand notation for creating GraphQL types

The following shorthand notations are possible:

- Where you need to pass a GraphQLArgumentMap, i.e. a dictionary with names as keys and GraphQLArguments as values, you can also pass GraphQLInputTypes as values. The GraphQLInputTypes are then automatically wrapped into GraphQLArguments.
- Where you need to pass a GraphQLFieldMap, i.e. a dictionary with names as keys and GraphQLFields as values, you can also pass GraphQLOutputTypes as values. The GraphQLOutputTypes are then automatically wrapped into GraphQLFields.
- Where you need to pass a GraphQLInputFieldMap, i.e. a dictionary with names as keys and GraphQLInputFields as values, you can also pass GraphQLInputTypes as values. The GraphQLInputTypes are then automatically wrapped into GraphQLInputFields.
- Where you need to pass a GraphQLEnumValueMap, i.e. a dictionary with names as keys and GraphQLEnumValues as values, you can pass any other Python objects as values. These will be automatically wrapped into GraphQLEnumValues. You can also pass a Python Enum type as GraphQLEnumValueMap.

1.3.4 Custom output names of arguments and input fields

You can pass a custom `out_name` argument to *GraphQLArgument* and *GraphQLInputField* that allows using JavaScript naming conventions (camelCase) on ingress and Python naming conventions (snake_case) on egress. This feature is used by Graphene.

1.3.5 Custom output types of input object types

You can also pass a custom `out_type` argument to *GraphQLInputObjectType* that allows conversion to any Python type on egress instead of conversion to a dictionary, which is the default. This is used to support the container feature of Graphene InputObjectTypes.

1.3.6 Custom middleware

The *execute()* function takes an additional `middleware` argument which must be a sequence of middleware functions or a *MiddlewareManager* object. This feature is used by Graphene to affect the evaluation of fields using custom middleware. There has been a request to add this to GraphQL.js as well, but so far this feature only exists in GraphQL-core.

1.3.7 Custom execution context

The *execute()* function takes an additional `execution_context_class` argument which allows specifying a custom execution context class instead of the default *ExecutionContext* used by GraphQL-core.

1.3.8 Registering special types for descriptions

Normally, descriptions for GraphQL types must be strings. However, sometimes you may want to use other kinds of objects which are not strings, but are only resolved to strings at runtime. This is possible if you register the classes of such objects with *pyutils.register_description()*.

If you notice any other important differences, please let us know so that they can be either removed or listed here.

1.4 Reference

GraphQL-core

The primary *graphql* package includes everything you need to define a GraphQL schema and fulfill GraphQL requests.

GraphQL-core provides a reference implementation for the GraphQL specification but is also a useful utility for operating on GraphQL files and building sophisticated tools.

This top-level package exports a general purpose function for fulfilling all steps of the GraphQL specification in a single operation, but also includes utilities for every part of the GraphQL specification:

- Parsing the GraphQL language.
- Building a GraphQL type schema.
- Validating a GraphQL request against a type schema.
- Executing a GraphQL request against a type schema.

This also includes utility functions for operating on GraphQL types and GraphQL documents to facilitate building tools.

You may also import from each sub-package directly. For example, the following two import statements are equivalent:

```
from graphql import parse
from graphql.language import parse
```

The sub-packages of GraphQL-core 3 are:

- *graphql.language*: Parse and operate on the GraphQL language.
- *graphql.type*: Define GraphQL types and schema.
- *graphql.validation*: The Validation phase of fulfilling a GraphQL result.
- *graphql.execution*: The Execution phase of fulfilling a GraphQL request.
- *graphql.error*: Creating and formatting GraphQL errors.
- *graphql.utilities*: Common useful computations upon the GraphQL language and type objects.

1.4.1 Top-Level Functions

async graphql.**graphql**(*schema: GraphQLSchema*, *source: str | Source*, *root_value: Any = None*, *context_value: Any = None*, *variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *middleware: Middleware | None = None*, *execution_context_class: type[ExecutionContext] | None = None*, *is_awaitable: Callable[[Any], bool] | None = None*) → *ExecutionResult*

Execute a GraphQL operation asynchronously.

This is the primary entry point function for fulfilling GraphQL operations by parsing, validating, and executing a GraphQL document along side a GraphQL schema.

More sophisticated GraphQL servers, such as those which persist queries, may wish to separate the validation and execution phases to a static time tooling step, and a server runtime step.

This function does not support incremental delivery (*@defer* and *@stream*).

Accepts the following arguments:

Parameters

- **schema** – The GraphQL type system to use when validating and executing a query.
- **source** – A GraphQL language formatted string representing the requested operation.
- **root_value** – The value provided as the first argument to resolver functions on the top level type (e.g. the query object type).
- **context_value** – The context value is provided as an attribute of the second argument (the resolve info) to resolver functions. It is used to pass shared information useful at any point during query execution, for example the currently logged in user and connections to databases or other services.
- **variable_values** – A mapping of variable name to runtime value to use for all variables defined in the request string.
- **operation_name** – The name of the operation to use if request string contains multiple possible operations. Can be omitted if request string contains only one operation.
- **field_resolver** – A resolver function to use when one is not provided by the schema. If not provided, the default field resolver is used (which looks for a value or method on the source value with the field's name).
- **type_resolver** – A type resolver function to use when none is provided by the schema. If not provided, the default type resolver is used (which looks for a `__typename` field or alternatively calls the *is_type_of()* method).
- **middleware** – The middleware to wrap the resolvers with
- **execution_context_class** – The execution context class to use to build the context
- **is_awaitable** – The predicate to be used for checking whether values are awaitable

graphql.**graphql_sync**(*schema: GraphQLSchema*, *source: str | Source*, *root_value: Any = None*, *context_value: Any = None*, *variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *middleware: Middleware | None = None*, *execution_context_class: type[ExecutionContext] | None = None*, *check_sync: bool = False*) → *ExecutionResult*

Execute a GraphQL operation synchronously.

The graphql_sync function also fulfills GraphQL operations by parsing, validating, and executing a GraphQL document along side a GraphQL schema. However, it guarantees to complete synchronously (or throw an error) assuming that all field resolvers are also synchronous.

Set check_sync to True to still run checks that no awaitable values are returned.

1.4.2 Sub-Packages

Error

GraphQL Errors

The *graphql.error* package is responsible for creating and formatting GraphQL errors.

class graphql.error.**GraphQLError**(*message: str*, *nodes: Collection[*Node*]* | Node | *None* = *None*, *source:* Source | *None* = *None*, *positions: Collection[int]* | *None* = *None*, *path: Collection[str* | *int]* | *None* = *None*, *original_error: Exception* | *None* = *None*, *extensions: GraphQLErrorExtensions* | *None* = *None*)

Bases: `Exception`

GraphQL Error

A GraphQLError describes an Error found during the parse, validate, or execute phases of performing a GraphQL operation. In addition to a message, it also includes information about the locations in a GraphQL document and/or execution result that correspond to the Error.

__init__(*message: str*, *nodes: Collection[*Node*]* | Node | *None* = *None*, *source:* Source | *None* = *None*, *positions: Collection[int]* | *None* = *None*, *path: Collection[str* | *int]* | *None* = *None*, *original_error: Exception* | *None* = *None*, *extensions: GraphQLErrorExtensions* | *None* = *None*) → None

Initialize a GraphQLError.

args

extensions: GraphQLErrorExtensions | None

Extension fields to add to the formatted error

property formatted: *GraphQLFormattedError*

Get error formatted according to the specification.

Given a GraphQLError, format it according to the rules described by the "Response Format, Errors" section of the GraphQL Specification.

locations: list[*SourceLocation*] | None

Source locations

A list of (line, column) locations within the source GraphQL document which correspond to this error.

Errors during validation often contain multiple locations, for example to point out two things with the same name. Errors during execution include a single location, the field which produced the error.

message: str

A message describing the Error for debugging purposes

nodes: list[*Node*] | None

A list of GraphQL AST Nodes corresponding to this error

original_error: Exception | None

The original error thrown from a field resolver during execution

path: list[str | int] | None

A list of field names and array indexes describing the JSON-path into the execution response which corresponds to this error.

Only included for errors during execution.

positions: Collection[int] | None

Error positions

A list of character offsets within the source GraphQL document which correspond to this error.

source: *Source* | None

The source GraphQL document for the first location of this error

Note that if this Error represents more than one node, the source may not represent nodes after the first node.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class graphql.error.**GraphQLSyntaxError**(*source:* Source, *position: int*, *description: str*)

Bases: *GraphQLError*

A GraphQLError representing a syntax error.

__init__(*source:* Source, *position: int*, *description: str*) → None

Initialize the GraphQLSyntaxError

args

extensions: GraphQLErrorExtensions | None

Extension fields to add to the formatted error

property formatted: *GraphQLFormattedError*

Get error formatted according to the specification.

Given a GraphQLError, format it according to the rules described by the "Response Format, Errors" section of the GraphQL Specification.

locations: list[*SourceLocation*] | None

Source locations

A list of (line, column) locations within the source GraphQL document which correspond to this error.

Errors during validation often contain multiple locations, for example to point out two things with the same name. Errors during execution include a single location, the field which produced the error.

message: str

A message describing the Error for debugging purposes

nodes: list[*Node*] | None

A list of GraphQL AST Nodes corresponding to this error

original_error: Exception | None

The original error thrown from a field resolver during execution

path: list[str | int] | None

A list of field names and array indexes describing the JSON-path into the execution response which corresponds to this error.

Only included for errors during execution.

positions: `Collection[int] | None`

Error positions

A list of character offsets within the source GraphQL document which correspond to this error.

source: *Source* `| None`

The source GraphQL document for the first location of this error

Note that if this Error represents more than one node, the source may not represent nodes after the first node.

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class graphql.error.**GraphQLFormattedError**

Bases: `TypedDict`

Formatted GraphQL error

extensions: `Dict[str, Any]`

locations: `list['graphql.language.FormattedSourceLocation']`

message: `str`

path: `list[str | int]`

graphql.error.**located_error**(*original_error: Exception*, *nodes: None | Collection[*Node*] = None*, *path: Collection[str | int] | None = None*) → *GraphQLError*

Located GraphQL Error

Given an arbitrary Exception, presumably thrown while attempting to execute a GraphQL operation, produce a new GraphQLError aware of the location in the document responsible for the original Exception.

Execution

GraphQL Execution

The *graphql.execution* package is responsible for the execution phase of fulfilling a GraphQL request.

graphql.execution.**execute**(*schema:* GraphQLSchema, *document:* DocumentNode, *root_value: Any = None*, *context_value: Any = None*, *variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *subscribe_field_resolver: GraphQLFieldResolver | None = None*, *middleware: Middleware | None = None*, *execution_context_class: type[*ExecutionContext*] | None = None*, *is_awaitable: Callable[[Any], bool] | None = None*) → AwaitableOrValue[*ExecutionResult*]

Execute a GraphQL operation.

Implements the "Executing requests" section of the GraphQL specification.

Returns an ExecutionResult (if all encountered resolvers are synchronous), or a coroutine object eventually yielding an ExecutionResult.

If the arguments to this function do not result in a legal execution context, a GraphQLError will be thrown immediately explaining the invalid input.

This function does not support incremental delivery (*@defer* and *@stream*). If an operation that defers or streams data is executed with this function, it will throw an error instead. Use *experimental_execute_incrementally* if you want to support incremental delivery.

graphql.execution.**experimental_execute_incrementally**(*schema: GraphQLSchema*, *document: DocumentNode*, *root_value: Any = None*, *context_value: Any = None*, *variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *subscribe_field_resolver: GraphQLFieldResolver | None = None*, *middleware: Middleware | None = None*, *execution_context_class: type[ExecutionContext] | None = None*, *is_awaitable: Callable[[Any], bool] | None = None*) → AwaitableOrValue[*ExecutionResult* | *ExperimentalIncrementalExecutionResults*]

Execute GraphQL operation incrementally (internal implementation).

Implements the "Executing requests" section of the GraphQL specification, including *@defer* and *@stream* as proposed in https://github.com/graphql/graphql-spec/pull/742

This function returns an awaitable that is either a single ExecutionResult or an ExperimentalIncrementalExecutionResults object, containing an *initialResult* and a stream of *subsequent_results*.

graphql.execution.**execute_sync**(*schema: GraphQLSchema*, *document: DocumentNode*, *root_value: Any = None*, *context_value: Any = None*, *variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *middleware: Middleware | None = None*, *execution_context_class: type[ExecutionContext] | None = None*, *check_sync: bool = False*) → *ExecutionResult*

Execute a GraphQL operation synchronously.

Also implements the "Executing requests" section of the GraphQL specification.

However, it guarantees to complete synchronously (or throw an error) assuming that all field resolvers are also synchronous.

Set check_sync to True to still run checks that no awaitable values are returned.

graphql.execution.**default_field_resolver**(*source: Any*, *info: GraphQLResolveInfo*, ***args: Any*) → Any

Default field resolver.

If a resolve function is not given, then a default resolve behavior is used which takes the property of the source object of the same name as the field and returns it as the result, or if it's a function, returns the result of calling that function while passing along args and context.

For dictionaries, the field names are used as keys, for all other objects they are used as attribute names.

graphql.execution.**default_type_resolver**(*value: Any*, *info: GraphQLResolveInfo*, *abstract_type: GraphQLAbstractType*) → AwaitableOrValue[str | None]

Default type resolver function.

If a resolve_type function is not given, then a default resolve behavior is used which attempts two strategies:

First, See if the provided value has a `__typename` field defined, if so, use that value as name of the resolved type.

Otherwise, test each possible type for the abstract type by calling `is_type_of()` for the object being coerced, returning the first type that matches.

class graphql.execution.**ExecutionContext**(*schema: GraphQLSchema*, *fragments: dict[str, graphql.language.ast.FragmentDefinitionNode]*, *root_value: Any*, *context_value: Any*, *operation: OperationDefinitionNode*, *variable_values: dict[str, Any]*, *field_resolver: Callable[[...], Any]*, *type_resolver: Callable[[Any, GraphQLResolveInfo, Union[GraphQLInterfaceType, GraphQLUnionType]], Optional[Union[Awaitable[Optional[str]], str]]]*, *subscribe_field_resolver: Callable[[...], Any]*, *subsequent_payloads: dict[Union[graphql.execution.incremental_publisher.DeferredFragmentRecord, graphql.execution.incremental_publisher.StreamItemsRecord], None]*, *errors: list[graphql.error.graphql_error.GraphQLError]*, *middleware_manager: graphql.execution.middleware.MiddlewareManager | None*, *is_awaitable: Optional[Callable[[Any], bool]]*)

Bases: `IncrementalPublisherMixin`

Data that must be available at all points during query execution.

Namely, schema of the type system that is currently executing, and the fragments defined in the query document.

__init__(*schema: GraphQLSchema*, *fragments: dict[str, graphql.language.ast.FragmentDefinitionNode]*, *root_value: Any*, *context_value: Any*, *operation: OperationDefinitionNode*, *variable_values: dict[str, Any]*, *field_resolver: Callable[[...], Any]*, *type_resolver: Callable[[Any, GraphQLResolveInfo, Union[GraphQLInterfaceType, GraphQLUnionType]], Optional[Union[Awaitable[Optional[str]], str]]]*, *subscribe_field_resolver: Callable[[...], Any]*, *subsequent_payloads: dict[Union[graphql.execution.incremental_publisher.DeferredFragmentRecord, graphql.execution.incremental_publisher.StreamItemsRecord], None]*, *errors: list[graphql.error.graphql_error.GraphQLError]*, *middleware_manager: graphql.execution.middleware.MiddlewareManager | None*, *is_awaitable: Optional[Callable[[Any], bool]]*) → None

classmethod build(*schema: GraphQLSchema*, *document: DocumentNode*, *root_value: Any = None*, *context_value: Any = None*, *raw_variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *subscribe_field_resolver: GraphQLFieldResolver | None = None*, *middleware: Middleware | None = None*, *is_awaitable: Callable[[Any], bool] | None = None*) → list[*GraphQLError*] | *ExecutionContext*

Build an execution context

Constructs a ExecutionContext object from the arguments passed to execute, which we will pass throughout the other execution methods.

Throws a GraphQLError if a valid execution context cannot be created.

For internal use only.

build_per_event_execution_context(*payload: Any*) → *ExecutionContext*

Create a copy of the execution context for usage with subscribe events.

build_resolve_info(*field_def:* GraphQLField, *field_group: list[*graphql.language.ast.FieldNode*]*, *parent_type:* GraphQLObjectType, *path:* Path) → *GraphQLResolveInfo*

Build the GraphQLResolveInfo object.

For internal use only.

static build_response(*data: dict[str, Any] | None*, *errors: list[*graphql.error.graphql_error.GraphQLError*]*) → *ExecutionResult*

Build response.

Given a completed execution context and data, build the (data, errors) response defined by the "Response" section of the GraphQL spec.

collect_and_execute_subfields(*return_type:* GraphQLObjectType, *field_group: FieldGroup*, *path:* Path, *result: Any*, *incremental_data_record: IncrementalDataRecord | None*) → AwaitableOrValue[dict[str, Any]]

Collect sub-fields to execute to complete this value.

collect_subfields(*return_type:* GraphQLObjectType, *field_group: list[*graphql.language.ast.FieldNode*]*) → FieldsAndPatches

Collect subfields.

A cached collection of relevant subfields with regard to the return type is kept in the execution context as `_subfields_cache`. This ensures the subfields are not repeatedly calculated, which saves overhead when resolving lists of values.

complete_abstract_value(*return_type: GraphQLAbstractType*, *field_group: FieldGroup*, *info:* GraphQLResolveInfo, *path:* Path, *result: Any*, *incremental_data_record: IncrementalDataRecord | None*) → AwaitableOrValue[Any]

Complete an abstract value.

Complete a value of an abstract type by determining the runtime object type of that value, then complete the value for that type.

async complete_async_iterator_value(*item_type: Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList*]]]*, *field_group: list[*graphql.language.ast.FieldNode*]*, *info:* GraphQLResolveInfo, *path:* Path, *async_iterator: AsyncIterator[Any]*, *incremental_data_record: Optional[Union[DeferredFragmentRecord, StreamItemsRecord]]*) → list[Any]

Complete an async iterator.

Complete an async iterator value by completing the result and calling recursively until all the results are completed.

async complete_awaitable_value(*return_type: Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList*]]]*, *field_group: list[*graphql.language.ast.FieldNode*]*, *info:* GraphQLResolveInfo, *path:* Path, *result: Any*, *incremental_data_record: Optional[Union[DeferredFragmentRecord, StreamItemsRecord]] = None*) → Any

Complete an awaitable value.

static complete_leaf_value(*return_type: Union[GraphQLScalarType, GraphQLEnumType]*, *result: Any*) → Any

Complete a leaf value.

Complete a Scalar or Enum by serializing to a valid value, returning null if serialization is not possible.

complete_list_item_value(*item: Any*, *complete_results: list[Any]*, *item_type: Union[GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull[Union[GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList]]]*, *field_group: list[graphql.language.ast.FieldNode]*, *info: GraphQLResolveInfo*, *item_path: Path*, *incremental_data_record: Optional[Union[DeferredFragmentRecord, StreamItemsRecord]]*) → bool

Complete a list item value by adding it to the completed results.

Returns True if the value is awaitable.

complete_list_value(*return_type: GraphQLList[GraphQLOutputType]*, *field_group: FieldGroup*, *info: GraphQLResolveInfo*, *path: Path*, *result: AsyncIterable[Any] | Iterable[Any]*, *incremental_data_record: IncrementalDataRecord | None*) → AwaitableOrValue[list[Any]]

Complete a list value.

Complete a list value by completing each item in the list with the inner type.

complete_object_value(*return_type: GraphQLObjectType*, *field_group: FieldGroup*, *info: GraphQLResolveInfo*, *path: Path*, *result: Any*, *incremental_data_record: IncrementalDataRecord | None*) → AwaitableOrValue[dict[str, Any]]

Complete an Object value by executing all sub-selections.

complete_value(*return_type: GraphQLOutputType*, *field_group: FieldGroup*, *info: GraphQLResolveInfo*, *path: Path*, *result: Any*, *incremental_data_record: IncrementalDataRecord | None*) → AwaitableOrValue[Any]

Complete a value.

Implements the instructions for completeValue as defined in the "Value completion" section of the spec.

If the field type is Non-Null, then this recursively completes the value for the inner type. It throws a field error if that completion returns null, as per the "Nullability" section of the spec.

If the field type is a List, then this recursively completes the value for the inner type on each item in the list.

If the field type is a Scalar or Enum, ensures the completed value is a legal value of the type by calling the `serialize` method of GraphQL type definition.

If the field is an abstract type, determine the runtime type of the value and then complete based on that type.

Otherwise, the field type expects a sub-selection set, and will complete the value by evaluating all sub-selections.

context_value: Any

ensure_valid_runtime_type(*runtime_type_name: Any*, *return_type: Union[GraphQLInterfaceType, GraphQLUnionType]*, *field_group: list[graphql.language.ast.FieldNode]*, *info: GraphQLResolveInfo*, *result: Any*) → *GraphQLObjectType*

Ensure that the given type is valid at runtime.

errors: list[*graphql.error.graphql_error.GraphQLError*]

execute_deferred_fragment(*parent_type*: GraphQLObjectType, *source_value*: *Any*, *fields*: *dict[str, list[*graphql.language.ast.FieldNode*]]*, *label*: *Optional[str]* = *None*, *path*: *Optional[*Path*]* = *None*, *parent_context*: *Optional[Union[DeferredFragmentRecord, StreamItemsRecord]]* = *None*) → None

Execute deferred fragment.

execute_field(*parent_type*: GraphQLObjectType, *source*: *Any*, *field_group*: *FieldGroup*, *path*: Path, *incremental_data_record*: *IncrementalDataRecord* | *None* = *None*) → AwaitableOrValue[Any]

Resolve the field on the given source object.

Implements the "Executing fields" section of the spec.

In particular, this method figures out the value that the field returns by calling its resolve function, then calls complete_value to await coroutine objects, serialize scalars, or execute the sub-selection-set for objects.

execute_fields(*parent_type*: GraphQLObjectType, *source_value*: *Any*, *path*: Path | *None*, *grouped_field_set*: *GroupedFieldSet*, *incremental_data_record*: *IncrementalDataRecord* | *None* = *None*) → AwaitableOrValue[dict[str, Any]]

Execute the given fields concurrently.

Implements the "Executing selection sets" section of the spec for fields that may be executed in parallel.

execute_fields_serially(*parent_type*: GraphQLObjectType, *source_value*: *Any*, *path*: Path | *None*, *grouped_field_set*: *GroupedFieldSet*) → AwaitableOrValue[dict[str, Any]]

Execute the given fields serially.

Implements the "Executing selection sets" section of the spec for fields that must be executed serially.

execute_operation() → AwaitableOrValue[dict[str, Any]]

Execute an operation.

Implements the "Executing operations" section of the spec.

async execute_stream_async_iterator(*initial_index*: *int*, *async_iterator*: *AsyncIterator[Any]*, *field_group*: *list[*graphql.language.ast.FieldNode*]*, *info*: GraphQLResolveInfo, *item_type*: *Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList*]]]*, *path*: Path, *label*: *Optional[str]* = *None*, *parent_context*: *Optional[Union[DeferredFragmentRecord, StreamItemsRecord]]* = *None*) → None

Execute stream iterator.

async execute_stream_async_iterator_item(*async_iterator: AsyncIterator[Any]*, *field_group: list[*graphql.language.ast.FieldNode*]*, *info:* GraphQLResolveInfo, *item_type: Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList*]]]*, *incremental_data_record: StreamItemsRecord*, *path:* Path, *item_path:* Path) → Any

Execute stream iterator item.

execute_stream_field(*path:* Path, *item_path:* Path, *item: AwaitableOrValue[Any]*, *field_group: FieldGroup*, *info:* GraphQLResolveInfo, *item_type: GraphQLOutputType*, *label: str | None = None*, *parent_context: IncrementalDataRecord | None = None*) → IncrementalDataRecord

Execute stream field.

field_resolver: Callable[[...], Any]

filter_subsequent_payloads(*null_path:* Path, *current_incremental_data_record: IncrementalDataRecord | None = None*) → None

Filter subsequent payloads.

fragments: dict[str, *graphql.language.ast.FragmentDefinitionNode*]

get_completed_incremental_results() → list[Union[*graphql.execution.incremental_publisher.IncrementalDeferResult*, *graphql.execution.incremental_publisher.IncrementalStreamResult*]]

Get completed incremental results.

get_stream_values(*field_group: list[*graphql.language.ast.FieldNode*]*, *path:* Path) → graphql.execution.execute.StreamArguments | None

Get stream values.

Returns an object containing the *@stream* arguments if a field should be streamed based on the experimental flag, stream directive present and not disabled by the "if" argument.

handle_field_error(*raw_error: Exception*, *return_type: Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList*]]]*, *field_group: list[*graphql.language.ast.FieldNode*]*, *path:* Path, *incremental_data_record: Optional[Union[DeferredFragmentRecord, StreamItemsRecord]] = None*) → None

Handle error properly according to the field type.

static is_awaitable(*value: Any*) → TypeGuard[Awaitable]

Return True if object can be passed to an `await` expression.

Instead of testing whether the object is an instance of abc.Awaitable, we check the existence of an *__await__* attribute. This is much faster.

map_source_to_response(*result_or_stream: Union[*ExecutionResult, *AsyncIterable[Any]]*) → Union[AsyncGenerator[*ExecutionResult*, None], *ExecutionResult*]

Map source result to response.

For each payload yielded from a subscription, map it over the normal GraphQL *execute()* function, with `payload` as the `root_value`. This implements the "MapSourceToResponseEvent" algorithm described in the GraphQL specification. The *execute()* function provides the "ExecuteSubscriptionEvent" algorithm, as it is nearly identical to the "ExecuteQuery" algorithm, for which *execute()* is also used.

middleware_manager: *graphql.execution.middleware.MiddlewareManager* | None

operation: *OperationDefinitionNode*

root_value: Any

schema: *GraphQLSchema*

subscribe_field_resolver: Callable[[...], Any]

subsequent_payloads:
dict[Union[graphql.execution.incremental_publisher.DeferredFragmentRecord, graphql.execution.incremental_publisher.StreamItemsRecord], None]

type_resolver: Callable[[Any, *GraphQLResolveInfo*, Union[*GraphQLInterfaceType*, *GraphQLUnionType*]], Optional[Union[Awaitable[Optional[str]], str]]]

variable_values: dict[str, Any]

async yield_subsequent_payloads() → AsyncGenerator[*SubsequentIncrementalExecutionResult*, None]

Yield subsequent payloads.

class graphql.execution.**ExecutionResult**(*data: Optional[dict[str, Any]] = None*, *errors: Optional[list[*graphql.error.graphql_error.GraphQLError*]] = None*, *extensions: Optional[dict[str, Any]] = None*)

Bases: `object`

The result of GraphQL execution.

- `data` is the result of a successful execution of the query.
- `errors` is included when any errors occurred as a non-empty list.
- `extensions` is reserved for adding non-standard properties.

__init__(*data: Optional[dict[str, Any]] = None*, *errors: Optional[list[*graphql.error.graphql_error.GraphQLError*]] = None*, *extensions: Optional[dict[str, Any]] = None*) → None

data: dict[str, Any] | None

errors: list[*graphql.error.graphql_error.GraphQLError*] | None

extensions: dict[str, Any] | None

property formatted: *FormattedExecutionResult*

Get execution result formatted according to the specification.

class graphql.execution.**FormattedExecutionResult**

Bases: `TypedDict`

Formatted execution result

data: dict[str, Any] | None

errors: list[*graphql.error.graphql_error.GraphQLFormattedError*]

extensions: dict[str, Any]

class graphql.execution.**ExperimentalIncrementalExecutionResults**(*initial_result: InitialIncrementalExecutionResult*, *subsequent_results: AsyncGenerator[SubsequentIncrementalExecutionResult, None]*)

Bases: NamedTuple

Execution results when retrieved incrementally.

__init__()

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

initial_result: *InitialIncrementalExecutionResult*

Alias for field number 0

subsequent_results: AsyncGenerator[*SubsequentIncrementalExecutionResult*, None]

Alias for field number 1

class graphql.execution.**InitialIncrementalExecutionResult**(*data: Optional[dict[str, Any]] = None*, *errors: Optional[list[graphql.error.graphql_error.GraphQLError]] = None*, *incremental: Optional[Sequence[Union[IncrementalDeferResult, IncrementalStreamResult]]] = None*, *has_next: bool = False*, *extensions: Optional[dict[str, Any]] = None*)

Bases: object

Initial incremental execution result.

- has_next is True if a future payload is expected.
- incremental is a list of the results from defer/stream directives.

__init__(*data: Optional[dict[str, Any]] = None*, *errors: Optional[list[graphql.error.graphql_error.GraphQLError]] = None*, *incremental: Optional[Sequence[Union[IncrementalDeferResult, IncrementalStreamResult]]] = None*, *has_next: bool = False*, *extensions: Optional[dict[str, Any]] = None*) → None

data: dict[str, Any] | None

errors: list[*graphql.error.graphql_error.GraphQLError*] | None

extensions: dict[str, Any] | None

property formatted: *FormattedInitialIncrementalExecutionResult*

Get execution result formatted according to the specification.

has_next: bool

incremental: Optional[Sequence[Union[*IncrementalDeferResult*, *IncrementalStreamResult*]]]

class graphql.execution.**FormattedInitialIncrementalExecutionResult**

Bases: TypedDict

Formatted initial incremental execution result

data: dict[str, Any] | None

errors: list[*graphql.error.graphql_error.GraphQLFormattedError*]

extensions: dict[str, Any]

hasNext: bool

incremental: list[Union[*graphql.execution.incremental_publisher.FormattedIncrementalDeferResult*, *graphql.execution.incremental_publisher.FormattedIncrementalStreamResult*]]

class graphql.execution.**SubsequentIncrementalExecutionResult**(*incremental: Optional[Sequence[Union[*IncrementalDeferResult, IncrementalStreamResult*]]] = None*, *has_next: bool = False*, *extensions: Optional[dict[str, Any]] = None*)

Bases: object

Subsequent incremental execution result.

- has_next is True if a future payload is expected.
- incremental is a list of the results from defer/stream directives.

__init__(*incremental: Optional[Sequence[Union[*IncrementalDeferResult, IncrementalStreamResult*]]] = None*, *has_next: bool = False*, *extensions: Optional[dict[str, Any]] = None*) → None

extensions: dict[str, Any] | None

property formatted: *FormattedSubsequentIncrementalExecutionResult*

Get execution result formatted according to the specification.

has_next: bool

incremental: Optional[Sequence[Union[*IncrementalDeferResult*, *IncrementalStreamResult*]]]

class graphql.execution.**FormattedSubsequentIncrementalExecutionResult**

Bases: TypedDict

Formatted subsequent incremental execution result

extensions: dict[str, Any]

hasNext: bool

incremental:
list[Union[*graphql.execution.incremental_publisher.FormattedIncrementalDeferResult*,
***graphql.execution.incremental_publisher.FormattedIncrementalStreamResult*]]**

class graphql.execution.**IncrementalDeferResult**(*data: dict[str, Any] | None = None*, *errors: list[*GraphQLError*] | None = None*, *path: list[str | int] | None = None*, *label: str | None = None*, *extensions: dict[str, Any] | None = None*)

Bases: `object`

Incremental deferred execution result

__init__(*data: dict[str, Any] | None = None*, *errors: list[*GraphQLError*] | None = None*, *path: list[str | int] | None = None*, *label: str | None = None*, *extensions: dict[str, Any] | None = None*) → None

data: dict[str, Any] | None

errors: list[*GraphQLError*] | None

extensions: dict[str, Any] | None

property formatted: *FormattedIncrementalDeferResult*

Get execution result formatted according to the specification.

label: str | None

path: list[str | int] | None

class graphql.execution.**FormattedIncrementalDeferResult**

Bases: `TypedDict`

Formatted incremental deferred execution result

data: dict[str, Any] | None

errors: list[*GraphQLFormattedError*]

extensions: dict[str, Any]

label: str

path: list[str | int]

class graphql.execution.**IncrementalStreamResult**(*items: list[Any] | None = None*, *errors: list[*GraphQLError*] | None = None*, *path: list[str | int] | None = None*, *label: str | None = None*, *extensions: dict[str, Any] | None = None*)

Bases: `object`

Incremental streamed execution result

__init__(*items: list[Any] | None = None*, *errors: list[*GraphQLError*] | None = None*, *path: list[str | int] | None = None*, *label: str | None = None*, *extensions: dict[str, Any] | None = None*) → None

errors: list[*GraphQLError*] | None

extensions: dict[str, Any] | None

property formatted: *FormattedIncrementalStreamResult*

Get execution result formatted according to the specification.

items: list[Any] | None

label: str | None

path: list[str | int] | None

class graphql.execution.**FormattedIncrementalStreamResult**

Bases: TypedDict

Formatted incremental stream execution result

errors: list[*GraphQLFormattedError*]

extensions: dict[str, Any]

label: str

path: list[str | int]

graphql.execution.**IncrementalResult**

alias of Union[*IncrementalDeferResult*, *IncrementalStreamResult*]

graphql.execution.**FormattedIncrementalResult**

alias of Union[*FormattedIncrementalDeferResult*, *FormattedIncrementalStreamResult*]

graphql.execution.**subscribe**(*schema:* GraphQLSchema, *document:* DocumentNode, *root_value: Any = None*, *context_value: Any = None*, *variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *subscribe_field_resolver: GraphQLFieldResolver | None = None*, *execution_context_class: type[*ExecutionContext*] | None = None*) → AwaitableOrValue[AsyncIterator[*ExecutionResult*] | *ExecutionResult*]

Create a GraphQL subscription.

Implements the "Subscribe" algorithm described in the GraphQL spec.

Returns a coroutine object which yields either an AsyncIterator (if successful) or an ExecutionResult (client error). The coroutine will raise an exception if a server error occurs.

If the client-provided arguments to this function do not result in a compliant subscription, a GraphQL Response (ExecutionResult) with descriptive errors and no data will be returned.

If the source stream could not be created due to faulty subscription resolver logic or underlying systems, the coroutine object will yield a single ExecutionResult containing `errors` and no `data`.

If the operation succeeded, the coroutine will yield an AsyncIterator, which yields a stream of ExecutionResults representing the response stream.

This function does not support incremental delivery (*@defer* and *@stream*). If an operation that defers or streams data is executed with this function, a field error will be raised at the location of the *@defer* or *@stream* directive.

graphql.execution.**create_source_event_stream**(*schema:* GraphQLSchema, *document:* DocumentNode, *root_value: Any = None*, *context_value: Any = None*, *variable_values: dict[str, Any] | None = None*, *operation_name: str | None = None*, *field_resolver: GraphQLFieldResolver | None = None*, *type_resolver: GraphQLTypeResolver | None = None*, *subscribe_field_resolver: GraphQLFieldResolver | None = None*, *execution_context_class: type[*ExecutionContext*] | None = None*) → AwaitableOrValue[AsyncIterable[Any] | *ExecutionResult*]

Create source event stream

Implements the "CreateSourceEventStream" algorithm described in the GraphQL specification, resolving the subscription source event stream.

Returns a coroutine that yields an AsyncIterable.

If the client-provided arguments to this function do not result in a compliant subscription, a GraphQL Response (ExecutionResult) with descriptive errors and no data will be returned.

If the source stream could not be created due to faulty subscription resolver logic or underlying systems, the coroutine object will yield a single ExecutionResult containing `errors` and no `data`.

A source event stream represents a sequence of events, each of which triggers a GraphQL execution for that event.

This may be useful when hosting the stateful subscription service in a different process or machine than the stateless GraphQL execution engine, or otherwise separating these two steps. For more on this, see the "Supporting Subscriptions at Scale" information in the GraphQL spec.

graphql.execution.**Middleware**

alias of `Optional[Union[Tuple, List, MiddlewareManager]]`

class graphql.execution.**MiddlewareManager**(**middlewares: Any*)

Bases: `object`

Manager for the middleware chain.

This class helps to wrap resolver functions with the provided middleware functions and/or objects. The functions take the next middleware function as first argument. If middleware is provided as an object, it must provide a method `resolve` that is used as the middleware function.

Note that since resolvers return "AwaitableOrValue"s, all middleware functions must be aware of this and check whether values are awaitable before awaiting them.

__init__(**middlewares: Any*) → None

get_field_resolver(*field_resolver: Callable[[...], Any]*) → Callable[[...], Any]

Wrap the provided resolver with the middleware.

Returns a function that chains the middleware functions with the provided resolver function.

middlewares

graphql.execution.**get_directive_values**(*directive_def:* GraphQLDirective, *node: Union[*EnumValueDefinitionNode, ExecutableDefinitionNode, FieldDefinitionNode, InputValueDefinitionNode, SelectionNode, SchemaDefinitionNode, TypeDefinitionNode, TypeExtensionNode*], variable_values: Optional[dict[str, Any]] = None*) → dict[str, Any] | None

Get coerced argument values based on provided nodes.

Prepares a dict of argument values given a directive definition and an AST node which may contain directives. Optionally also accepts a dict of variable values.

If the directive does not exist on the node, returns None.

graphql.execution.**get_variable_values**(*schema:* GraphQLSchema, *var_def_nodes:* *Collection[*VariableDefinitionNode*]*, *inputs: dict[str, Any]*, *max_errors: Optional[int] = None*) → Union[List[*GraphQLError*], Dict[str, Any]]

Get coerced variable values based on provided definitions.

Prepares a dict of variable values of the correct type based on the provided variable definitions and arbitrary input. If the input cannot be parsed to match the variable definitions, a GraphQLError will be raised.

Language

GraphQL Language

The *graphql.language* package is responsible for parsing and operating on the GraphQL language.

AST

class graphql.language.**Location**(*start_token:* Token, *end_token:* Token, *source:* Source)

Bases: object

AST Location

Contains a range of UTF-8 character offsets and token references that identify the region of the source from which the AST derived.

__init__(*start_token:* Token, *end_token:* Token, *source:* Source) → None

end: **int**

end_token: ***Token***

source: ***Source***

start: **int**

start_token: ***Token***

class graphql.language.**Node**(***kwargs: Any*)

Bases: object

AST nodes

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: **tuple[str, ...] = ('loc',)**

kind: **str = 'ast'**

loc: ***graphql.language.ast.Location*** **| None**

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

Each kind of AST node has its own class:

class graphql.language.**ArgumentNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'name', 'value')

kind: str = 'argument'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: *ValueNode*

class graphql.language.**BooleanValueNode**(***kwargs: Any*)

Bases: *ValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'value')

kind: str = 'boolean_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: bool

class graphql.language.**ConstArgumentNode**(***kwargs: Any*)

Bases: *ArgumentNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'name', 'value', 'name', 'value')

kind: str = 'argument'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: Union[*IntValueNode*, *FloatValueNode*, *StringValueNode*, *BooleanValueNode*, *NullValueNode*, *EnumValueNode*, *ConstListValueNode*, *ConstObjectValueNode*]

class graphql.language.**ConstDirectiveNode**(***kwargs: Any*)

Bases: *DirectiveNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

arguments: tuple[*graphql.language.ast.ConstArgumentNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'arguments', 'name', 'arguments')

kind: str = 'directive'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**ConstListValueNode**(***kwargs: Any*)

Bases: *ListValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'values', 'values')

kind: str = 'list_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

values: tuple[Union[*graphql.language.ast.IntValueNode*, *graphql.language.ast.FloatValueNode*, *graphql.language.ast.StringValueNode*, *graphql.language.ast.BooleanValueNode*, *graphql.language.ast.NullValueNode*, *graphql.language.ast.EnumValueNode*, *graphql.language.ast.ConstListValueNode*, *graphql.language.ast.ConstObjectValueNode*], ...]

class graphql.language.**ConstObjectFieldNode**(***kwargs: Any*)

Bases: *ObjectFieldNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'name', 'value', 'name', 'value')

kind: str = 'object_field'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: Union[*IntValueNode*, *FloatValueNode*, *StringValueNode*, *BooleanValueNode*, *NullValueNode*, *EnumValueNode*, *ConstListValueNode*, *ConstObjectValueNode*]

class graphql.language.**ConstObjectValueNode**(***kwargs: Any*)

Bases: *ObjectValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

fields: tuple[*graphql.language.ast.ConstObjectFieldNode*, ...]

keys: tuple[str, ...] = ('loc', 'fields', 'fields')

kind: str = 'object_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

graphql.language.**ConstValueNode**

alias of Union[*IntValueNode*, *FloatValueNode*, *StringValueNode*, *BooleanValueNode*, *NullValueNode*, *EnumValueNode*, *ConstListValueNode*, *ConstObjectValueNode*]

class graphql.language.**DefinitionNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc',)

kind: str = 'definition'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**DirectiveDefinitionNode**(***kwargs: Any*)

Bases: *TypeSystemDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

arguments: tuple[*graphql.language.ast.InputValueDefinitionNode*, ...]

description: *graphql.language.ast.StringValueNode* | None

keys: tuple[str, ...] = ('loc', 'description', 'name', 'arguments', 'repeatable', 'locations')

kind: str = 'directive_definition'

loc: *graphql.language.ast.Location* | None

locations: tuple[*graphql.language.ast.NameNode*, ...]

name: *NameNode*

repeatable: bool

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.DirectiveNode(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

arguments: tuple[*graphql.language.ast.ArgumentNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'arguments')

kind: str = 'directive'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.DocumentNode(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

definitions: tuple[*graphql.language.ast.DefinitionNode*, ...]

keys: tuple[str, ...] = ('loc', 'definitions')

kind: str = 'document'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.EnumTypeDefinitionNode(***kwargs: Any*)

Bases: *TypeDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives', 'values')

kind: str = 'enum_type_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

values: tuple[*graphql.language.ast.EnumValueDefinitionNode*, ...]

class graphql.language.**EnumTypeExtensionNode**(***kwargs: Any*)

Bases: *TypeExtensionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives', 'values')

kind: str = 'enum_type_extension'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

values: tuple[*graphql.language.ast.EnumValueDefinitionNode*, ...]

class graphql.language.**EnumValueDefinitionNode**(***kwargs: Any*)

Bases: *DefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives')

kind: str = 'enum_value_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**EnumValueNode**(***kwargs: Any*)

Bases: *ValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'value')

kind: str = 'enum_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: str

class graphql.language.**ErrorBoundaryNode**(***kwargs: Any*)

Bases: *NullabilityAssertionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'nullability_assertion', 'nullability_assertion')

kind: str = 'error_boundary'

loc: *graphql.language.ast.Location* | None

nullability_assertion: *ListNullabilityOperatorNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**ExecutableDefinitionNode**(***kwargs: Any*)

Bases: *DefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.DirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives', 'variable_definitions', 'selection_set')

kind: str = 'executable_definition'

loc: *graphql.language.ast.Location* | None

name: *graphql.language.ast.NameNode* | None

selection_set: *SelectionSetNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

variable_definitions: tuple[*graphql.language.ast.VariableDefinitionNode*, ...]

class graphql.language.**FieldDefinitionNode**(***kwargs: Any*)

Bases: *DefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

arguments: tuple[*graphql.language.ast.InputValueDefinitionNode*, ...]

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives', 'arguments', 'type')

kind: str = 'field_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

type: *TypeNode*

class graphql.language.**FieldNode**(***kwargs: Any*)

Bases: *SelectionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

alias: *graphql.language.ast.NameNode* | None

arguments: tuple[*graphql.language.ast.ArgumentNode*, ...]

directives: tuple[*graphql.language.ast.DirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'directives', 'alias', 'name', 'arguments', 'nullability_assertion', 'selection_set')

kind: str = 'field'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

nullability_assertion: *NullabilityAssertionNode*

selection_set: *graphql.language.ast.SelectionSetNode* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**FloatValueNode**(***kwargs: Any*)

Bases: *ValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'value')

kind: str = 'float_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: str

class graphql.language.**FragmentDefinitionNode**(***kwargs: Any*)

Bases: *ExecutableDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.DirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives', 'variable_definitions', 'selection_set', 'type_condition')

kind: str = 'fragment_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

selection_set: *SelectionSetNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

type_condition: *NamedTypeNode*

variable_definitions: tuple[*graphql.language.ast.VariableDefinitionNode*, ...]

class graphql.language.**FragmentSpreadNode**(***kwargs: Any*)

Bases: *SelectionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.DirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'directives', 'name')

kind: str = 'fragment_spread'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**InlineFragmentNode**(***kwargs: Any*)

Bases: *SelectionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.DirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'directives', 'type_condition', 'selection_set')

kind: str = 'inline_fragment'

loc: *graphql.language.ast.Location* | None

selection_set: *SelectionSetNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

type_condition: *NamedTypeNode*

class graphql.language.**InputObjectTypeDefinitionNode**(***kwargs: Any*)

Bases: *TypeDefinitionNode*

__init__(**kwargs: Any*) → None

Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

fields: tuple[*graphql.language.ast.InputValueDefinitionNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives', 'fields')

kind: str = 'input_object_type_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**InputObjectTypeExtensionNode**(**kwargs: Any*)

Bases: *TypeExtensionNode*

__init__(**kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

fields: tuple[*graphql.language.ast.InputValueDefinitionNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives', 'fields')

kind: str = 'input_object_type_extension'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**InputValueDefinitionNode**(**kwargs: Any*)

Bases: *DefinitionNode*

__init__(**kwargs: Any*) → None

Initialize the node with the given keyword arguments.

default_value: Optional[Union[*IntValueNode*, *FloatValueNode*, *StringValueNode*, *BooleanValueNode*, *NullValueNode*, *EnumValueNode*, *ConstListValueNode*, *ConstObjectValueNode*]]

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives', 'type', 'default_value')

kind: str = 'input_value_definition'

loc: *graphql.language.ast.Location* | **None**

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

type: *TypeNode*

class graphql.language.**IntValueNode**(***kwargs: Any*)

Bases: *ValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: **tuple[str, ...] = ('loc', 'value')**

kind: **str = 'int_value'**

loc: *graphql.language.ast.Location* | **None**

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: **str**

class graphql.language.**InterfaceTypeDefinitionNode**(***kwargs: Any*)

Bases: *TypeDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | **None**

directives: **tuple[***graphql.language.ast.ConstDirectiveNode***, ...]**

fields: **tuple[***graphql.language.ast.FieldDefinitionNode***, ...]**

interfaces: **tuple[***graphql.language.ast.NamedTypeNode***, ...]**

keys: **tuple[str, ...] = ('loc', 'description', 'name', 'directives', 'fields', 'interfaces')**

kind: **str = 'interface_type_definition'**

loc: *graphql.language.ast.Location* | **None**

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**InterfaceTypeExtensionNode**(***kwargs: Any*)

Bases: *TypeExtensionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: **tuple[***graphql.language.ast.ConstDirectiveNode***, ...]**

fields: tuple[*graphql.language.ast.FieldDefinitionNode*, ...]

interfaces: tuple[*graphql.language.ast.NamedTypeNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives', 'interfaces', 'fields')

kind: str = 'interface_type_extension'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**ListNullabilityOperatorNode**(***kwargs: Any*)

Bases: *NullabilityAssertionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'nullability_assertion', 'nullability_assertion')

kind: str = 'list_nullability_operator'

loc: *graphql.language.ast.Location* | None

nullability_assertion: *graphql.language.ast.NullabilityAssertionNode* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**ListTypeNode**(***kwargs: Any*)

Bases: *TypeNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'type')

kind: str = 'list_type'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

type: *TypeNode*

class graphql.language.**ListValueNode**(***kwargs: Any*)

Bases: *ValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'values')

kind: str = 'list_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

values: tuple[*graphql.language.ast.ValueNode*, ...]

class graphql.language.**NameNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'value')

kind: str = 'name'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: str

class graphql.language.**NamedTypeNode**(***kwargs: Any*)

Bases: *TypeNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'name')

kind: str = 'named_type'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**NonNullAssertionNode**(***kwargs: Any*)

Bases: *NullabilityAssertionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'nullability_assertion', 'nullability_assertion')

kind: str = 'non_null_assertion'

loc: *graphql.language.ast.Location* | None

nullability_assertion: *ListNullabilityOperatorNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**NonNullTypeNode**(***kwargs: Any*)

Bases: *TypeNode*

__init__(**kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'type')

kind: str = 'non_null_type'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

type: *graphql.language.ast.NamedTypeNode* | *graphql.language.ast.ListTypeNode*

class graphql.language.**NullabilityAssertionNode**(**kwargs: Any*)

Bases: *Node*

__init__(**kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'nullability_assertion')

kind: str = 'nullability_assertion'

loc: *graphql.language.ast.Location* | None

nullability_assertion: *graphql.language.ast.NullabilityAssertionNode* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**NullValueNode**(**kwargs: Any*)

Bases: *ValueNode*

__init__(**kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc',)

kind: str = 'null_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**ObjectFieldNode**(**kwargs: Any*)

Bases: *Node*

__init__(**kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'name', 'value')

kind: str = 'object_field'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: *ValueNode*

class graphql.language.**ObjectTypeDefinitionNode**(***kwargs: Any*)

Bases: *TypeDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | **None**

directives: **tuple**[*graphql.language.ast.ConstDirectiveNode*, ...]

fields: **tuple**[*graphql.language.ast.FieldDefinitionNode*, ...]

interfaces: **tuple**[*graphql.language.ast.NamedTypeNode*, ...]

keys: **tuple[str, ...] = ('loc', 'description', 'name', 'directives', 'interfaces', 'fields')**

kind: **str = 'object_type_definition'**

loc: *graphql.language.ast.Location* | **None**

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**ObjectTypeExtensionNode**(***kwargs: Any*)

Bases: *TypeExtensionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: **tuple**[*graphql.language.ast.ConstDirectiveNode*, ...]

fields: **tuple**[*graphql.language.ast.FieldDefinitionNode*, ...]

interfaces: **tuple**[*graphql.language.ast.NamedTypeNode*, ...]

keys: **tuple[str, ...] = ('loc', 'name', 'directives', 'interfaces', 'fields')**

kind: **str = 'object_type_extension'**

loc: *graphql.language.ast.Location* | **None**

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**ObjectValueNode**(***kwargs: Any*)

Bases: *ValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

fields: tuple[*graphql.language.ast.ObjectFieldNode*, ...]

keys: tuple[str, ...] = ('loc', 'fields')

kind: str = 'object_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**OperationDefinitionNode**(***kwargs: Any*)

Bases: *ExecutableDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.DirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives', 'variable_definitions', 'selection_set', 'operation')

kind: str = 'operation_definition'

loc: *graphql.language.ast.Location* | None

name: *graphql.language.ast.NameNode* | None

operation: *OperationType*

selection_set: *SelectionSetNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

variable_definitions: tuple[*graphql.language.ast.VariableDefinitionNode*, ...]

class graphql.language.**OperationType**(*value*)

Bases: Enum

An enumeration.

MUTATION = 'mutation'

QUERY = 'query'

SUBSCRIPTION = 'subscription'

class graphql.language.**OperationTypeDefinitionNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc', 'operation', 'type')

kind: str = 'operation_type_definition'

loc: *graphql.language.ast.Location* | None

operation: *OperationType*

to_dict(*locations: bool = False*) → dict
Concert node to a dictionary.

type: *NamedTypeNode*

class graphql.language.ScalarTypeDefinitionNode(***kwargs: Any*)
Bases: *TypeDefinitionNode*

__init__(***kwargs: Any*) → None
Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives')

kind: str = 'scalar_type_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict
Concert node to a dictionary.

class graphql.language.ScalarTypeExtensionNode(***kwargs: Any*)
Bases: *TypeExtensionNode*

__init__(***kwargs: Any*) → None
Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives')

kind: str = 'scalar_type_extension'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict
Concert node to a dictionary.

class graphql.language.SchemaDefinitionNode(***kwargs: Any*)
Bases: *TypeSystemDefinitionNode*

__init__(***kwargs: Any*) → None
Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'directives', 'operation_types')

kind: str = 'schema_definition'

loc: *graphql.language.ast.Location* | **None**

operation_types: **tuple[***graphql.language.ast.OperationTypeDefinitionNode***, ...]**

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**SchemaExtensionNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: **tuple[***graphql.language.ast.ConstDirectiveNode***, ...]**

keys: **tuple[str, ...] = ('loc', 'directives', 'operation_types')**

kind: **str = 'schema_extension'**

loc: *graphql.language.ast.Location* | **None**

operation_types: **tuple[***graphql.language.ast.OperationTypeDefinitionNode***, ...]**

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**SelectionNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: **tuple[***graphql.language.ast.DirectiveNode***, ...]**

keys: **tuple[str, ...] = ('loc', 'directives')**

kind: **str = 'selection'**

loc: *graphql.language.ast.Location* | **None**

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**SelectionSetNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: **tuple[str, ...] = ('loc', 'selections')**

kind: **str = 'selection_set'**

loc: *graphql.language.ast.Location* | **None**

selections: **tuple[***graphql.language.ast.SelectionNode***, ...]**

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**StringValueNode**(***kwargs: Any*)

Bases: *ValueNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

block: bool | None

keys: tuple[str, ...] = ('loc', 'value', 'block')

kind: str = 'string_value'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

value: str

class graphql.language.**TypeDefinitionNode**(***kwargs: Any*)

Bases: *TypeSystemDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.DirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives')

kind: str = 'type_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**TypeExtensionNode**(***kwargs: Any*)

Bases: *TypeSystemDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives')

kind: str = 'type_extension'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**TypeNode**(***kwargs: Any*)

Bases: *Node*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc',)

kind: str = 'type'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

class graphql.language.**TypeSystemDefinitionNode**(***kwargs: Any*)

Bases: *DefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc',)

kind: str = 'type_system_definition'

loc: *graphql.language.ast.Location* | None

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

graphql.language.**TypeSystemExtensionNode**

alias of Union[*SchemaExtensionNode*, *TypeExtensionNode*]

class graphql.language.**UnionTypeDefinitionNode**(***kwargs: Any*)

Bases: *TypeDefinitionNode*

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

description: *graphql.language.ast.StringValueNode* | None

directives: tuple[*graphql.language.ast.ConstDirectiveNode*, ...]

keys: tuple[str, ...] = ('loc', 'description', 'name', 'directives', 'types')

kind: str = 'union_type_definition'

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

types: tuple[*graphql.language.ast.NamedTypeNode*, ...]

class graphql.language.**UnionTypeExtensionNode**(***kwargs: Any*)

Bases: *TypeExtensionNode*

__init__(**kwargs: Any) → None
Initialize the node with the given keyword arguments.

directives: tuple[graphql.language.ast.ConstDirectiveNode, ...]

keys: tuple[str, ...] = ('loc', 'name', 'directives', 'types')

kind: str = 'union_type_extension'

loc: graphql.language.ast.Location | None

name: NameNode

to_dict(locations: bool = False) → dict
Concert node to a dictionary.

types: tuple[graphql.language.ast.NamedTypeNode, ...]

class graphql.language.ValueNode(**kwargs: Any)
Bases: Node

__init__(**kwargs: Any) → None
Initialize the node with the given keyword arguments.

keys: tuple[str, ...] = ('loc',)

kind: str = 'value'

loc: graphql.language.ast.Location | None

to_dict(locations: bool = False) → dict
Concert node to a dictionary.

class graphql.language.VariableDefinitionNode(**kwargs: Any)
Bases: Node

__init__(**kwargs: Any) → None
Initialize the node with the given keyword arguments.

default_value: Optional[Union[IntValueNode, FloatValueNode, StringValueNode, BooleanValueNode, NullValueNode, EnumValueNode, ConstListValueNode, ConstObjectValueNode]]

directives: tuple[graphql.language.ast.ConstDirectiveNode, ...]

keys: tuple[str, ...] = ('loc', 'variable', 'type', 'default_value', 'directives')

kind: str = 'variable_definition'

loc: graphql.language.ast.Location | None

to_dict(locations: bool = False) → dict
Concert node to a dictionary.

type: TypeNode

variable: VariableNode

class graphql.language.VariableNode(**kwargs: Any)
Bases: ValueNode

__init__(***kwargs: Any*) → None

Initialize the node with the given keyword arguments.

keys: `tuple[str, ...] = ('loc', 'name')`

kind: `str = 'variable'`

loc: *graphql.language.ast.Location* | None

name: *NameNode*

to_dict(*locations: bool = False*) → dict

Concert node to a dictionary.

Directive locations are specified using the following enumeration:

class graphql.language.**DirectiveLocation**(*value*)

Bases: Enum

The enum type representing the directive location values.

ARGUMENT_DEFINITION = 'argument definition'

ENUM = 'enum'

ENUM_VALUE = 'enum value'

FIELD = 'field'

FIELD_DEFINITION = 'field definition'

FRAGMENT_DEFINITION = 'fragment definition'

FRAGMENT_SPREAD = 'fragment spread'

INLINE_FRAGMENT = 'inline fragment'

INPUT_FIELD_DEFINITION = 'input field definition'

INPUT_OBJECT = 'input object'

INTERFACE = 'interface'

MUTATION = 'mutation'

OBJECT = 'object'

QUERY = 'query'

SCALAR = 'scalar'

SCHEMA = 'schema'

SUBSCRIPTION = 'subscription'

UNION = 'union'

VARIABLE_DEFINITION = 'variable definition'

You can also check the type of nodes with the following predicates:

graphql.language.**is_definition_node**(*node:* Node) → TypeGuard[*DefinitionNode*]

Check whether the given node represents a definition.

graphql.language.**is_executable_definition_node**(*node:* Node) → TypeGuard[*ExecutableDefinitionNode*]

Check whether the given node represents an executable definition.

graphql.language.**is_selection_node**(*node:* Node) → TypeGuard[*SelectionNode*]

Check whether the given node represents a selection.

graphql.language.**is_value_node**(*node:* Node) → TypeGuard[*ValueNode*]

Check whether the given node represents a value.

graphql.language.**is_const_value_node**(*node:* Node) → TypeGuard[*ValueNode*]

Check whether the given node represents a constant value.

graphql.language.**is_type_node**(*node:* Node) → TypeGuard[*TypeNode*]

Check whether the given node represents a type.

graphql.language.**is_type_system_definition_node**(*node:* Node) → TypeGuard[*TypeSystemDefinitionNode*]

Check whether the given node represents a type system definition.

graphql.language.**is_type_definition_node**(*node:* Node) → TypeGuard[*TypeDefinitionNode*]

Check whether the given node represents a type definition.

graphql.language.**is_type_system_extension_node**(*node:* Node) → TypeGuard[*graphql.language.ast.SchemaExtensionNode* | *graphql.language.ast.TypeExtensionNode*]

Check whether the given node represents a type system extension.

graphql.language.**is_type_extension_node**(*node:* Node) → TypeGuard[*TypeExtensionNode*]

Check whether the given node represents a type extension.

Lexer

class graphql.language.**Lexer**(*source:* Source)

Bases: object

GraphQL Lexer

A Lexer is a stateful stream generator in that every time it is advanced, it returns the next token in the Source. Assuming the source lexes, the final Token emitted by the lexer will be of kind EOF, after which the lexer will repeatedly return the same EOF token whenever called.

__init__(*source:* Source) → None

Given a Source object, initialize a Lexer for that source.

advance() → *Token*

Advance the token stream to the next non-ignored token.

create_token(*kind:* TokenKind, *start: int*, *end: int*, *value: Optional[str] = None*) → *Token*

Create a token with line and column location information.

lookahead() → *Token*

Look ahead and return the next non-ignored token, but do not change state.

print_code_point_at(*location: int*) → str

Print the code point at the given location.

Prints the code point (or end of file reference) at a given location in a source for use in error messages.

Printable ASCII is printed quoted, while other points are printed in Unicode code point form (ie. U+1234).

read_block_string(*start: int*) → *Token*

Read a block string token from the source file.

read_comment(*start: int*) → *Token*

Read a comment token from the source file.

read_digits(*start: int*, *first_char: str*) → int

Return the new position in the source after reading one or more digits.

read_escaped_character(*position: int*) → EscapeSequence

Read escaped character sequence

read_escaped_unicode_fixed_width(*position: int*) → EscapeSequence

Read escaped unicode with fixed width

read_escaped_unicode_variable_width(*position: int*) → EscapeSequence

Read escaped unicode with variable width

read_name(*start: int*) → *Token*

Read an alphanumeric + underscore name from the source.

read_next_token(*start: int*) → *Token*

Get the next token from the source starting at the given position.

This skips over whitespace until it finds the next lexable token, then lexes punctuators immediately or calls the appropriate helper function for more complicated tokens.

read_number(*start: int*, *first_char: str*) → *Token*

Reads a number token from the source file.

This can be either a FloatValue or an IntValue, depending on whether a FractionalPart or ExponentPart is encountered.

read_string(*start: int*) → *Token*

Read a single-quote string token from the source file.

class graphql.language.**TokenKind**(*value*)

Bases: Enum

The different kinds of tokens that the lexer emits

AMP = '&'

AT = '@'

BANG = '!'

BLOCK_STRING = 'BlockString'

BRACE_L = '{'

BRACE_R = '}'

BRACKET_L = '['

BRACKET_R = ']'

COLON = ':'

COMMENT = 'Comment'

DOLLAR = '$'

EOF = '<EOF>'

EQUALS = '='

FLOAT = 'Float'

INT = 'Int'

NAME = 'Name'

PAREN_L = '('

PAREN_R = ')'

PIPE = '|'

QUESTION_MARK = '?'

SOF = '<SOF>'

SPREAD = '...'

STRING = 'String'

class graphql.language.**Token**(*kind: TokenKind*, *start: int*, *end: int*, *line: int*, *column: int*, *value: str | None = None*)

Bases: object

AST Token

Represents a range of characters represented by a lexical token within a Source.

__init__(*kind: TokenKind*, *start: int*, *end: int*, *line: int*, *column: int*, *value: str | None = None*) → None

column: int

property desc: str

A helper property to describe a token as a string for debugging

end: int

kind: *TokenKind*

line: int

next: *Token* | None

prev: *Token* | None

start: int

value: str | None

Location

graphql.language.**get_location**(*source:* Source, *position: int*) → *SourceLocation*

Get the line and column for a character position in the source.

Takes a Source and a UTF-8 character offset, and returns the corresponding line and column as a SourceLocation.

class graphql.language.**SourceLocation**(*line: int*, *column: int*)

Bases: NamedTuple

Represents a location in a Source.

__init__()

column: int

Alias for field number 1

count(*value*, */*)

Return number of occurrences of value.

property formatted: *graphql.language.FormattedSourceLocation*

Get formatted source location.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

line: int

Alias for field number 0

graphql.language.**print_location**(*location:* Location) → str

Render a helpful description of the location in the GraphQL Source document.

class graphql.language.**FormattedSourceLocation**

Bases: TypedDict

Formatted source location

column: int

line: int

Parser

graphql.language.**parse**(*source: Union[*Source*, str]*, *no_location: bool = False*, *max_tokens: Optional[int] = None*, *allow_legacy_fragment_variables: bool = False*, *experimental_client_controlled_nullability: bool = False*) → *DocumentNode*

Given a GraphQL source, parse it into a Document.

Throws GraphQLError if a syntax error is encountered.

By default, the parser creates AST nodes that know the location in the source that they correspond to. The no_location option disables that behavior for performance or testing.

Parser CPU and memory usage is linear to the number of tokens in a document, however in extreme cases it becomes quadratic due to memory exhaustion. Parsing happens before validation so even invalid queries can burn lots of CPU time and memory. To prevent this you can set a maximum number of tokens allowed within a document.

Legacy feature (will be removed in v3.3):

If `allow_legacy_fragment_variables` is set to `True`, the parser will understand and parse variable definitions contained in a fragment definition. They'll be represented in the *variable_definitions* field of the *FragmentDefinitionNode*.

The syntax is identical to normal, query-defined variables. For example:

```
fragment A($var: Boolean = false) on T  {
  ...
}
```

EXPERIMENTAL:

If enabled, the parser will understand and parse Client Controlled Nullability Designators contained in Fields. They'll be represented in the *nullability_assertion* field of the *FieldNode*.

The syntax looks like the following:

```
 {
   nullableField!
   nonNullableField?
   nonNullableSelectionSet? {
     childField!
  }
}
```

Note: this feature is experimental and may change or be removed in the future.

graphql.language.**parse_type**(*source: Union[Source, str]*, *no_location: bool = False*, *max_tokens: Optional[int] = None*, *allow_legacy_fragment_variables: bool = False*) → *TypeNode*

Parse the AST for a given string containing a GraphQL Type.

Throws GraphQLError if a syntax error is encountered.

This is useful within tools that operate upon GraphQL Types directly and in isolation of complete GraphQL documents.

Consider providing the results to the utility function: *value_from_ast()*.

graphql.language.**parse_value**(*source: Union[Source, str]*, *no_location: bool = False*, *max_tokens: Optional[int] = None*, *allow_legacy_fragment_variables: bool = False*) → *ValueNode*

Parse the AST for a given string containing a GraphQL value.

Throws GraphQLError if a syntax error is encountered.

This is useful within tools that operate upon GraphQL Values directly and in isolation of complete GraphQL documents.

Consider providing the results to the utility function: *value_from_ast()*.

graphql.language.**parse_const_value**(*source: Union[Source, str]*, *no_location: bool = False*, *max_tokens: Optional[int] = None*, *allow_legacy_fragment_variables: bool = False*) → Union[*IntValueNode*, *FloatValueNode*, *StringValueNode*, *BooleanValueNode*, *NullValueNode*, *EnumValueNode*, *ConstListValueNode*, *ConstObjectValueNode*]

Parse the AST for a given string containing a GraphQL constant value.

Similar to parse_value, but raises a arse error if it encounters a variable. The return type will be a constant value.

Printer

graphql.language.**print_ast**(*ast:* Node) → str

Convert an AST into a string.

The conversion is done using a set of reasonable formatting rules.

Source

class graphql.language.**Source**(*body: str*, *name: str = 'GraphQL request'*, *location_offset:* SourceLocation = *SourceLocation(line=1, column=1)*)

Bases: `object`

A representation of source input to GraphQL.

__init__(*body: str*, *name: str = 'GraphQL request'*, *location_offset:* SourceLocation = *SourceLocation(line=1, column=1)*) → None

Initialize source input.

The `name` and `location_offset` parameters are optional, but they are useful for clients who store GraphQL documents in source files. For example, if the GraphQL input starts at line 40 in a file named `Foo.graphql`, it might be useful for `name` to be `"Foo.graphql"` and location to be `(40, 0)`.

The `line` and `column` attributes in `location_offset` are 1-indexed.

body

get_location(*position: int*) → *SourceLocation*

Get source location.

location_offset

name

graphql.language.**print_source_location**(*source:* Source, *source_location:* SourceLocation) → str

Render a helpful description of the location in the GraphQL Source document.

Visitor

graphql.language.**visit**(*root:* Node, *visitor:* Visitor, *visitor_keys: Optional[Dict[str, Tuple[str, ...]]] = None*) → Any

Visit each node in an AST.

visit() will walk through an AST using a depth-first traversal, calling the visitor's enter methods at each node in the traversal, and calling the leave methods after visiting that node and all of its child nodes.

By returning different values from the enter and leave methods, the behavior of the visitor can be altered, including skipping over a sub-tree of the AST (by returning False), editing the AST by returning a value or None to remove the value, or to stop the whole traversal by returning *BREAK*.

When using *visit()* to edit an AST, the original AST will not be modified, and a new version of the AST with the changes applied will be returned from the visit function.

To customize the node attributes to be used for traversal, you can provide a dictionary visitor_keys mapping node kinds to node attributes.

class graphql.language.**Visitor**

Bases: object

Visitor that walks through an AST.

Visitors can define two generic methods "enter" and "leave". The former will be called when a node is entered in the traversal, the latter is called after visiting the node and its child nodes. These methods have the following signature:

```
def enter(self, node, key, parent, path, ancestors):
    # The return value has the following meaning:
    # IDLE (None): no action
    # SKIP: skip visiting this node
    # BREAK: stop visiting altogether
    # REMOVE: delete this node
    # any other value: replace this node with the returned value
    return

def leave(self, node, key, parent, path, ancestors):
    # The return value has the following meaning:
    # IDLE (None) or SKIP: no action
    # BREAK: stop visiting altogether
    # REMOVE: delete this node
    # any other value: replace this node with the returned value
    return
```

The parameters have the following meaning:

Parameters

- **node** – The current node being visiting.
- **key** – The index or key to this node from the parent node or Array.
- **parent** – the parent immediately above this node, which may be an Array.
- **path** – The key path to get to this node from the root node.
- **ancestors** – All nodes and Arrays visited before reaching parent of this node. These correspond to array indices in `path`. Note: ancestors includes arrays which contain the parent of visited node.

You can also define node kind specific methods by suffixing them with an underscore followed by the kind of the node to be visited. For instance, to visit `field` nodes, you would define the methods `enter_field()` and/or `leave_field()`, with the same signature as above. If no kind specific method has been defined for a given node, the generic method is called.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__() → None

enter_leave_map: dict[str, graphql.language.visitor.EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

class graphql.language.**ParallelVisitor**(*visitors: Collection[*Visitor*]*)

Bases: *Visitor*

A Visitor which delegates to many visitors to run in parallel.

Each visitor will be visited for each node before moving on.

If a prior visitor edits a node, no following visitors will see that node.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*visitors: Collection[*Visitor*]*) → None

Create a new visitor from the given list of parallel visitors.

enter_leave_map: dict[str, graphql.language.visitor.EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

The module also exports the following enumeration that can be used as the return type for *Visitor* methods:

class graphql.language.visitor.**VisitorActionEnum**(*value*)

Bases: Enum

Special return values for the visitor methods.

You can also use the values of this enum directly.

BREAK = True

REMOVE = Ellipsis

SKIP = False

The module also exports the values of this enumeration directly. These can be used as return values of *Visitor* methods to signal particular actions:

graphql.language.**BREAK (same as ``True``)**

This return value signals that no further nodes shall be visited.

graphql.language.**SKIP (same as ``False``)**

This return value signals that the current node shall be skipped.

graphql.language.**REMOVE (same as``Ellipsis``)**

This return value signals that the current node shall be deleted.

graphql.language.**IDLE = None**

This return value signals that no additional action shall take place.

PyUtils

Python Utils

This package contains dependency-free Python utility functions used throughout the codebase.

Each utility should belong in its own file and be the default export.

These functions are not part of the module interface and are subject to change.

graphql.pyutils.**camel_to_snake**(*s: str*) → str

Convert from CamelCase to snake_case

graphql.pyutils.**snake_to_camel**(*s: str*, *upper: bool = True*) → str

Convert from snake_case to CamelCase

If upper is set, then convert to upper CamelCase, otherwise the first character keeps its case.

graphql.pyutils.**cached_property**(*func*)

graphql.pyutils.**register_description**(*base: type*) → None

Register a class that shall be accepted as a description.

graphql.pyutils.**unregister_description**(*base: type*) → None

Unregister a class that shall no more be accepted as a description.

graphql.pyutils.**did_you_mean**(*suggestions: Sequence[str]*, *sub_message: Optional[str] = None*) → str

Given [A, B, C] return ' Did you mean A, B, or C?'

graphql.pyutils.**identity_func**(*x: T = Undefined*, **_args: Any*) → T

Return the first received argument.

graphql.pyutils.**inspect**(*value: Any*) → str

Inspect value and a return string representation for error messages.

Used to print values in error messages. We do not use repr() in order to not leak too much of the inner Python representation of unknown objects, and we do not use json.dumps() because not all objects can be serialized as JSON and we want to output strings with single quotes like Python repr() does it.

We also restrict the size of the representation by truncating strings and collections and allowing only a maximum recursion depth.

graphql.pyutils.**is_awaitable**(*value: Any*) → TypeGuard[Awaitable]

Return True if object can be passed to an `await` expression.

Instead of testing whether the object is an instance of abc.Awaitable, we check the existence of an *__await__* attribute. This is much faster.

graphql.pyutils.**is_collection**(*value: Any*) → TypeGuard[Collection]

Check if value is a collection, but not a string or a mapping.

graphql.pyutils.**is_iterable**(*value: Any*) → TypeGuard[Iterable]

Check if value is an iterable, but not a string or a mapping.

graphql.pyutils.**natural_comparison_key**(*key: str*) → tuple

Comparison key function for sorting strings by natural sort order.

See: https://en.wikipedia.org/wiki/Natural_sort_order

graphql.pyutils.**AwaitableOrValue**

alias of `Union[Awaitable[T], T]`

graphql.pyutils.**suggestion_list**(*input_: str*, *options: Collection[str]*) → list[str]

Get list with suggestions for a given input.

Given an invalid input string and list of valid options, returns a filtered list of valid options sorted based on their similarity with the input.

class graphql.pyutils.**FrozenError**

Bases: TypeError

Error when trying to change a frozen (read only) collection.

class graphql.pyutils.**Path**(*prev: Path | None*, *key: str | int*, *typename: str | None*)

Bases: NamedTuple

A generic path of string or integer indices

__init__()

add_key(*key: str | int*, *typename: Optional[str] = None*) → *Path*

Return a new Path containing the given key.

as_list() → list[str | int]

Return a list of the path keys.

count(*value*, */*)

Return number of occurrences of value.

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

key: str | int

current index in the path (string or integer)

prev: *graphql.pyutils.path.Path* | None

path with the previous indices

typename: str | None

name of the parent type to avoid path ambiguity

graphql.pyutils.**print_path_list**(*path: Collection[str | int]*) → str

Build a string describing the path.

class graphql.pyutils.**SimplePubSub**

Bases: object

A very simple publish-subscript system.

Creates an AsyncIterator from an EventEmitter.

Useful for mocking a PubSub system for tests.

__init__() → None

emit(*event: Any*) → bool

Emit an event.

get_subscriber(*transform: Optional[Callable] = None*) → *SimplePubSubIterator*

Return subscriber iterator

subscribers: set[Callable]

class graphql.pyutils.**SimplePubSubIterator**(*pubsub:* SimplePubSub, *transform: Optional[Callable]*)

Bases: AsyncIterator

Async iterator used for subscriptions.

__init__(*pubsub:* SimplePubSub, *transform: Optional[Callable]*) → None

async aclose() → None

Close the iterator.

async empty_queue() → None

Empty the queue.

async push_value(*event: Any*) → None

Push a new value.

graphql.pyutils.**Undefined = Undefined**

Symbol for undefined values

This singleton object is used to describe undefined or invalid values. It can be used in places where you would use undefined in GraphQL.js.

Type

GraphQL Type System

The *graphql.type* package is responsible for defining GraphQL types and schema.

Definition

Predicates

graphql.type.**is_composite_type**(*type_: Any*) → TypeGuard[Union[*GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*]]

Check whether this is a GraphQL composite type.

graphql.type.**is_enum_type**(*type_: Any*) → TypeGuard[*GraphQLEnumType*]

Check whether this is a GraphQL enum type.

graphql.type.**is_input_object_type**(*type_: Any*) → TypeGuard[*GraphQLInputObjectType*]

Check whether this is a GraphQL input type.

graphql.type.**is_input_type**(*type_: Any*) → TypeGuard[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]]]

Check whether this is a GraphQL input type.

graphql.type.**is_interface_type**(*type_: Any*) → TypeGuard[*GraphQLInterfaceType*]

Check whether this is a GraphQL interface type.

graphql.type.**is_leaf_type**(*type_: Any*) → TypeGuard[Union[*GraphQLScalarType*, *GraphQLEnumType*]]

Check whether this is a GraphQL leaf type.

graphql.type.**is_list_type**(*type_: Any*) → TypeGuard[*GraphQLList*]
Check whether this is a GraphQL list type.

graphql.type.**is_named_type**(*type_: Any*) → TypeGuard[*GraphQLNamedType*]
Check whether this is a named GraphQL type.

graphql.type.**is_non_null_type**(*type_: Any*) → TypeGuard[*GraphQLNonNull*]
Check whether this is a non-null GraphQL type.

graphql.type.**is_nullable_type**(*type_: Any*) → TypeGuard[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]
Check whether this is a nullable GraphQL type.

graphql.type.**is_object_type**(*type_: Any*) → TypeGuard[*GraphQLObjectType*]
Check whether this is a graphql object type

graphql.type.**is_output_type**(*type_: Any*) → TypeGuard[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*]]]]
Check whether this is a GraphQL output type.

graphql.type.**is_scalar_type**(*type_: Any*) → TypeGuard[*GraphQLScalarType*]
Check whether this is a GraphQL scalar type.

graphql.type.**is_type**(*type_: Any*) → TypeGuard[*GraphQLType*]
Check whether this is a GraphQL type.

graphql.type.**is_union_type**(*type_: Any*) → TypeGuard[*GraphQLUnionType*]
Check whether this is a GraphQL union type.

graphql.type.**is_wrapping_type**(*type_: Any*) → TypeGuard[*GraphQLWrappingType*]
Check whether this is a GraphQL wrapping type.

Assertions

graphql.type.**assert_abstract_type**(*type_: Any*) → Union[*GraphQLInterfaceType*, *GraphQLUnionType*]
Assert that this is a GraphQL abstract type.

graphql.type.**assert_composite_type**(*type_: Any*) → Union[*GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*]
Assert that this is a GraphQL composite type.

graphql.type.**assert_enum_type**(*type_: Any*) → *GraphQLEnumType*
Assert that this is a GraphQL enum type.

graphql.type.**assert_input_object_type**(*type_: Any*) → *GraphQLInputObjectType*
Assert that this is a GraphQL input type.

graphql.type.**assert_input_type**(*type_: Any*) → Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]]
Assert that this is a GraphQL input type.

graphql.type.**assert_interface_type**(*type_: Any*) → *GraphQLInterfaceType*

Assert that this is a GraphQL interface type.

graphql.type.**assert_leaf_type**(*type_: Any*) → Union[*GraphQLScalarType*, *GraphQLEnumType*]

Assert that this is a GraphQL leaf type.

graphql.type.**assert_list_type**(*type_: Any*) → *GraphQLList*

Assert that this is a GraphQL list type.

graphql.type.**assert_named_type**(*type_: Any*) → *GraphQLNamedType*

Assert that this is a named GraphQL type.

graphql.type.**assert_non_null_type**(*type_: Any*) → *GraphQLNonNull*

Assert that this is a non-null GraphQL type.

graphql.type.**assert_nullable_type**(*type_: Any*) → Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]

Assert that this is a nullable GraphQL type.

graphql.type.**assert_object_type**(*type_: Any*) → *GraphQLObjectType*

Assume that this is a graphql object type

graphql.type.**assert_output_type**(*type_: Any*) → Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*]]]

Assert that this is a GraphQL output type.

graphql.type.**assert_scalar_type**(*type_: Any*) → *GraphQLScalarType*

Assert that this is a GraphQL scalar type.

graphql.type.**assert_type**(*type_: Any*) → *GraphQLType*

Assert that this is a GraphQL type.

graphql.type.**assert_union_type**(*type_: Any*) → *GraphQLUnionType*

Assert that this is a GraphQL union type.

graphql.type.**assert_wrapping_type**(*type_: Any*) → *GraphQLWrappingType*

Assert that this is a GraphQL wrapping type.

Un-modifiers

graphql.type.**get_nullable_type**(*type_: None*) → None

graphql.type.**get_nullable_type**(*type_: Union[*GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList*]*) → Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]

graphql.type.**get_nullable_type**(*type_:* GraphQLNonNull) → Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]

Unwrap possible non-null type

graphql.type.**get_named_type**(*type_: None*) → None

graphql.type.**get_named_type**(*type_:* GraphQLType) → *GraphQLNamedType*

Unwrap possible wrapping type

Definitions

class graphql.type.**GraphQLEnumType**(*name: str*, **_args: Any*, ***_kwargs: Any*)

Bases: *GraphQLNamedType*

Enum Type Definition

Some leaf values of requests and input values are Enums. GraphQL serializes Enum values as strings, however internally Enums can be represented by any kind of type, often integers. They can also be provided as a Python Enum. In this case, the flag *names_as_values* determines what will be used as internal representation. The default value of *False* will use the enum values, the value *True* will use the enum names, and the value *None* will use the members themselves.

Example:

```
RGBType = GraphQLEnumType('RGB', {
    'RED': 0,
    'GREEN': 1,
    'BLUE': 2
})
```

Example using a Python Enum:

```
class RGBEnum(enum.Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

RGBType = GraphQLEnumType('RGB', enum.Enum)
```

Instead of raw values, you can also specify GraphQLEnumValue objects with more detail like description or deprecation information.

Note: If a value is not provided in a definition, the name of the enum value will be used as its internal value when the value is serialized.

__init__(*name: str*, *values: Union[Dict[str,* GraphQLEnumValue*], Mapping[str, Any], type[enum.Enum]]*, *names_as_values: bool | None = False*, *description: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*EnumTypeDefinitionNode*] = None*, *extension_ast_nodes: Optional[Collection[*EnumTypeExtensionNode*]] = None*) → None

ast_node: *graphql.language.ast.EnumTypeDefinitionNode* **| None**

description: **str | None**

extension_ast_nodes: **tuple[***graphql.language.ast.EnumTypeExtensionNode***, ...]**

extensions: **dict[str, Any]**

name: **str**

parse_literal(*value_node:* ValueNode, *_variables: Optional[dict[str, Any]] = None*) → Any

Parse literal value.

parse_value(*input_value: str*) → Any

Parse an enum value.

reserved_types: Mapping[str, *GraphQLNamedType*] = {'Boolean': <GraphQLScalarType 'Boolean'>, 'Float': <GraphQLScalarType 'Float'>, 'ID': <GraphQLScalarType 'ID'>, 'Int': <GraphQLScalarType 'Int'>, 'String': <GraphQLScalarType 'String'>, '__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}

serialize(*output_value: Any*) → str

Serialize an output value.

to_kwargs() → GraphQLEnumTypeKwargs

Get corresponding arguments.

values: Dict[str, *GraphQLEnumValue*]

class graphql.type.**GraphQLInputObjectType**(*name: str*, **_args: Any*, ***_kwargs: Any*)

Bases: *GraphQLNamedType*

Input Object Type Definition

An input object defines a structured collection of fields which may be supplied to a field argument.

Using `NonNull` will ensure that a value must be provided by the query.

Example:

```
NonNullFloat = GraphQLNonNull(GraphQLFloat())

class GeoPoint(GraphQLInputObjectType):
    name = 'GeoPoint'
    fields = {
        'lat': GraphQLInputField(NonNullFloat),
        'lon': GraphQLInputField(NonNullFloat),
        'alt': GraphQLInputField(
                   GraphQLFloat(), default_value=0)
    }
```

The outbound values will be Python dictionaries by default, but you can have them converted to other types by specifying an `out_type` function or class.

__init__(*name: str*, *fields: Union[Callable[[], Mapping[str,* GraphQLInputField*]], Mapping[str,* GraphQLInputField*]]*, *description: Optional[str] = None*, *out_type: Optional[Callable[[Dict[str, Any]], Any]] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*InputObjectTypeDefinitionNode*] = None*, *extension_ast_nodes: Optional[Collection[*InputObjectTypeExtensionNode*]] = None*) → None

ast_node: *graphql.language.ast.InputObjectTypeDefinitionNode* | None

description: str | None

extension_ast_nodes: tuple[*graphql.language.ast.InputObjectTypeExtensionNode*, ...]

extensions: dict[str, Any]

property fields: Dict[str, *GraphQLInputField*]

Get provided fields, wrap them as GraphQLInputField if needed.

name: str

static out_type(*value: dict[str, Any]*) → Any

Transform outbound values (this is an extension of GraphQL.js).

This default implementation passes values unaltered as dictionaries.

reserved_types: Mapping[str, *GraphQLNamedType*] = {'Boolean': <GraphQLScalarType 'Boolean'>, 'Float': <GraphQLScalarType 'Float'>, 'ID': <GraphQLScalarType 'ID'>, 'Int': <GraphQLScalarType 'Int'>, 'String': <GraphQLScalarType 'String'>, '__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}

to_kwargs() → GraphQLInputObjectTypeKwargs

Get corresponding arguments.

class graphql.type.**GraphQLInterfaceType**(*name: str*, **_args: Any*, ***_kwargs: Any*)

Bases: *GraphQLNamedType*

Interface Type Definition

When a field can return one of a heterogeneous set of types, an Interface type is used to describe what types are possible, what fields are in common across all types, as well as a function to determine which type is actually used when the field is resolved.

Example:

```
EntityType = GraphQLInterfaceType('Entity', {
        'name': GraphQLField(GraphQLString),
    })
```

__init__(*name: str*, *fields: Union[Callable[[], Mapping[str, GraphQLField]], Mapping[str, GraphQLField]]*, *interfaces: Optional[Union[Callable[[], Collection[GraphQLInterfaceType]], Collection[GraphQLInterfaceType]]] = None*, *resolve_type: Optional[Callable[[Any, GraphQLResolveInfo, Union[GraphQLInterfaceType, GraphQLUnionType]], Optional[Union[Awaitable[Optional[str]], str]]]] = None*, *description: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[InterfaceTypeDefinitionNode] = None*, *extension_ast_nodes: Optional[Collection[InterfaceTypeExtensionNode]] = None*) → None

ast_node: *graphql.language.ast.InterfaceTypeDefinitionNode* | None

description: str | None

extension_ast_nodes: tuple[*graphql.language.ast.InterfaceTypeExtensionNode*, ...]

extensions: dict[str, Any]

property fields: Dict[str, *GraphQLField*]

Get provided fields, wrapping them as GraphQLFields if needed.

property interfaces: tuple[*graphql.type.definition.GraphQLInterfaceType*, ...]

Get provided interfaces.

name: str

reserved_types: Mapping[str, *GraphQLNamedType*] = {'Boolean': <GraphQLScalarType 'Boolean'>, 'Float': <GraphQLScalarType 'Float'>, 'ID': <GraphQLScalarType 'ID'>, 'Int': <GraphQLScalarType 'Int'>, 'String': <GraphQLScalarType 'String'>, '__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}

resolve_type: Optional[Callable[[Any, *GraphQLResolveInfo*, Union[*GraphQLInterfaceType*, *GraphQLUnionType*]], Optional[Union[Awaitable[Optional[str]], str]]]]

to_kwargs() → GraphQLInterfaceTypeKwargs

Get corresponding arguments.

class graphql.type.**GraphQLObjectType**(*name: str*, **_args: Any*, ***_kwargs: Any*)

Bases: *GraphQLNamedType*

Object Type Definition

Almost all the GraphQL types you define will be object types. Object types have a name, but most importantly describe their fields.

Example:

```
AddressType = GraphQLObjectType('Address', {
    'street': GraphQLField(GraphQLString),
    'number': GraphQLField(GraphQLInt),
    'formatted': GraphQLField(GraphQLString,
        lambda obj, info, **args: f'{obj.number} {obj.street}')
})
```

When two types need to refer to each other, or a type needs to refer to itself in a field, you can use a lambda function with no arguments (a so-called "thunk") to supply the fields lazily.

Example:

```
PersonType = GraphQLObjectType('Person', lambda: {
    'name': GraphQLField(GraphQLString),
    'bestFriend': GraphQLField(PersonType)
})
```

__init__(*name: str*, *fields: Union[Callable[[], Mapping[str, GraphQLField]], Mapping[str, GraphQLField]]*, *interfaces: Optional[Union[Callable[[], Collection[GraphQLInterfaceType]], Collection[GraphQLInterfaceType]]] = None*, *is_type_of: Optional[Callable[[Any, GraphQLResolveInfo], Union[Awaitable[bool], bool]]] = None*, *extensions: Optional[dict[str, Any]] = None*, *description: Optional[str] = None*, *ast_node: Optional[ObjectTypeDefinitionNode] = None*, *extension_ast_nodes: Optional[Collection[ObjectTypeExtensionNode]] = None*) → None

ast_node: *graphql.language.ast.ObjectTypeDefinitionNode* | None

description: str | None

extension_ast_nodes: tuple[*graphql.language.ast.ObjectTypeExtensionNode*, ...]

extensions: dict[str, Any]

property fields: Dict[str, *GraphQLField*]

Get provided fields, wrapping them as GraphQLFields if needed.

property interfaces: tuple[*graphql.type.definition.GraphQLInterfaceType*, ...]

Get provided interfaces.

is_type_of: Optional[Callable[[Any, *GraphQLResolveInfo*], Union[Awaitable[bool], bool]]]

name: str

reserved_types: Mapping[str, *GraphQLNamedType*] = {'Boolean': <GraphQLScalarType 'Boolean'>, 'Float': <GraphQLScalarType 'Float'>, 'ID': <GraphQLScalarType 'ID'>, 'Int': <GraphQLScalarType 'Int'>, 'String': <GraphQLScalarType 'String'>, '__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}

to_kwargs() → GraphQLObjectTypeKwargs

Get corresponding arguments.

class graphql.type.**GraphQLScalarType**(*name: str*, **_args: Any*, ***_kwargs: Any*)

Bases: *GraphQLNamedType*

Scalar Type Definition

The leaf values of any request and input values to arguments are Scalars (or Enums) and are defined with a name and a series of functions used to parse input from ast or variables and to ensure validity.

If a type's serialize function returns `None`, then an error will be raised and a `None` value will be returned in the response. It is always better to validate.

Example:

```
def serialize_odd(value: Any) -> int:
    try:
        value = int(value)
    except ValueError:
        raise GraphQLError(
            f"Scalar 'Odd' cannot represent '{value}'"
```

(continues on next page)

(continued from previous page)

```
            " since it is not an integer.")
    if not value % 2:
        raise GraphQLError(
            f"Scalar 'Odd' cannot represent '{value}' since it is even.")
    return value

odd_type = GraphQLScalarType('Odd', serialize=serialize_odd)
```

__init__(*name: str*, *serialize: Optional[Callable[[Any], Any]] = None*, *parse_value: Optional[Callable[[Any], Any]] = None*, *parse_literal: Optional[Callable[[ValueNode, Optional[Dict[str, Any]]], Any]] = None*, *description: Optional[str] = None*, *specified_by_url: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[ScalarTypeDefinitionNode] = None*, *extension_ast_nodes: Optional[Collection[ScalarTypeExtensionNode]] = None*) → None

ast_node: *graphql.language.ast.ScalarTypeDefinitionNode* | None

description: str | None

extension_ast_nodes: tuple[*graphql.language.ast.ScalarTypeExtensionNode*, ...]

extensions: dict[str, Any]

name: str

parse_literal(*node: ValueNode*, *variables: Optional[dict[str, Any]] = None*) → Any

Parses an externally provided literal value to use as an input.

This default method uses the parse_value method and should be replaced with a more specific version when creating a scalar type.

static parse_value(*value: Any*) → Any

Parses an externally provided value to use as an input.

This default method just passes the value through and should be replaced with a more specific version when creating a scalar type.

reserved_types: Mapping[str, *GraphQLNamedType*] = {'Boolean': <GraphQLScalarType 'Boolean'>, 'Float': <GraphQLScalarType 'Float'>, 'ID': <GraphQLScalarType 'ID'>, 'Int': <GraphQLScalarType 'Int'>, 'String': <GraphQLScalarType 'String'>, '__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}

static serialize(*value: Any*) → Any

Serializes an internal value to include in a response.

This default method just passes the value through and should be replaced with a more specific version when creating a scalar type.

specified_by_url: str | None

to_kwargs() → GraphQLScalarTypeKwargs

Get corresponding arguments.

class graphql.type.**GraphQLUnionType**(*name: str*, **_args: Any*, ***_kwargs: Any*)

Bases: *GraphQLNamedType*

Union Type Definition

When a field can return one of a heterogeneous set of types, a Union type is used to describe what types are possible as well as providing a function to determine which type is actually used when the field is resolved.

Example:

```
def resolve_type(obj, _info, _type):
    if isinstance(obj, Dog):
        return DogType()
    if isinstance(obj, Cat):
        return CatType()

PetType = GraphQLUnionType('Pet', [DogType, CatType], resolve_type)
```

__init__(*name: str*, *types: Union[Callable[[], Collection[GraphQLObjectType]], Collection[GraphQLObjectType]]*, *resolve_type: Optional[Callable[[Any, GraphQLResolveInfo, Union[GraphQLInterfaceType, GraphQLUnionType]], Optional[Union[Awaitable[Optional[str]], str]]]] = None*, *description: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[UnionTypeDefinitionNode] = None*, *extension_ast_nodes: Optional[Collection[UnionTypeExtensionNode]] = None*) → None

ast_node: *graphql.language.ast.UnionTypeDefinitionNode* | None

description: str | None

extension_ast_nodes: tuple[*graphql.language.ast.UnionTypeExtensionNode*, ...]

extensions: dict[str, Any]

name: str

reserved_types: Mapping[str, *GraphQLNamedType*] = {'Boolean': <GraphQLScalarType 'Boolean'>, 'Float': <GraphQLScalarType 'Float'>, 'ID': <GraphQLScalarType 'ID'>, 'Int': <GraphQLScalarType 'Int'>, 'String': <GraphQLScalarType 'String'>, '__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}

resolve_type: Optional[Callable[[Any, *GraphQLResolveInfo*, Union[*GraphQLInterfaceType*, *GraphQLUnionType*]], Optional[Union[Awaitable[Optional[str]], str]]]]

to_kwargs() → GraphQLUnionTypeKwargs

Get corresponding arguments.

property **types**: tuple[*graphql.type.definition.GraphQLObjectType*, ...]

Get provided types.

Type Wrappers

class graphql.type.**GraphQLList**(*type_: GT*)

Bases: *GraphQLWrappingType*[GT]

List Type Wrapper

A list is a wrapping type which points to another type. Lists are often created within the context of defining the fields of an object type.

Example:

```
class PersonType(GraphQLObjectType):
    name = 'Person'

    @property
    def fields(self):
        return {
            'parents': GraphQLField(GraphQLList(PersonType())),
            'children': GraphQLField(GraphQLList(PersonType())),
        }
```

__init__(*type_: GT*) → None

of_type: GT

class graphql.type.**GraphQLNonNull**(*type_: GNT*)

Bases: *GraphQLWrappingType*[GNT]

Non-Null Type Wrapper

A non-null is a wrapping type which points to another type. Non-null types enforce that their values are never null and can ensure an error is raised if this ever occurs during a request. It is useful for fields which you can make a strong guarantee on non-nullability, for example usually the id field of a database row will never be null.

Example:

```
class RowType(GraphQLObjectType):
    name = 'Row'
    fields = {
        'id': GraphQLField(GraphQLNonNull(GraphQLString()))
    }
```

Note: the enforcement of non-nullability occurs within the executor.

__init__(*type_: GNT*) → None

of_type: GT

Types

graphql.type.**GraphQLAbstractType**

alias of Union[*GraphQLInterfaceType*, *GraphQLUnionType*]

class graphql.type.**GraphQLArgument**(*type_: Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList*]]], *default_value: Any = Undefined*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *out_name: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*InputValueDefinitionNode*] = None*)

Bases: object

Definition of a GraphQL argument

__init__(*type_: Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList*]]], *default_value: Any = Undefined*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *out_name: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*InputValueDefinitionNode*] = None*) → None

ast_node: *graphql.language.ast.InputValueDefinitionNode* | None

default_value: Any

deprecation_reason: str | None

description: str | None

extensions: dict[str, Any]

out_name: str | None

to_kwargs() → GraphQLArgumentKwargs

Get corresponding arguments.

type: Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]]

graphql.type.**GraphQLArgumentMap**

alias of Dict[str, *GraphQLArgument*]

graphql.type.**GraphQLCompositeType**

alias of Union[*GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*]

class graphql.type.**GraphQLEnumValue**(*value: Optional[Any] = None*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*EnumValueDefinitionNode*] = None*)

Bases: object

A GraphQL enum value.

__init__(*value: Optional[Any] = None*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[EnumValueDefinitionNode] = None*) → None

ast_node: *graphql.language.ast.EnumValueDefinitionNode* | **None**

deprecation_reason: **str | None**

description: **str | None**

extensions: **dict[str, Any]**

to_kwargs() → GraphQLEnumValueKwargs

Get corresponding arguments.

value: **Any**

graphql.type.**GraphQLEnumValueMap**

alias of Dict[str, *GraphQLEnumValue*]

class graphql.type.**GraphQLField**(*type_: Union[GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull[Union[GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList]]]*, *args: Optional[Dict[str, GraphQLArgument]] = None*, *resolve: Optional[Callable[[...], Any]] = None*, *subscribe: Optional[Callable[[...], Any]] = None*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[FieldDefinitionNode] = None*)

Bases: object

Definition of a GraphQL field

__init__(*type_: Union[GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList, GraphQLNonNull[Union[GraphQLScalarType, GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType, GraphQLEnumType, GraphQLList]]]*, *args: Optional[Dict[str, GraphQLArgument]] = None*, *resolve: Optional[Callable[[...], Any]] = None*, *subscribe: Optional[Callable[[...], Any]] = None*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[FieldDefinitionNode] = None*) → None

args: **Dict[str, *GraphQLArgument*]**

ast_node: *graphql.language.ast.FieldDefinitionNode* | **None**

deprecation_reason: **str | None**

description: **str | None**

extensions: **dict[str, Any]**

resolve: **Optional[Callable[[...], Any]]**

subscribe: **Optional[Callable[[...], Any]]**

to_kwargs() → GraphQLFieldKwargs

Get corresponding arguments.

type: Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*]]]

graphql.type.**GraphQLFieldMap**

alias of Dict[str, *GraphQLField*]

class graphql.type.**GraphQLInputField**(*type_: Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList*]]], *default_value: Any = Undefined*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *out_name: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*InputValueDefinitionNode*] = None*)

Bases: object

Definition of a GraphQL input field

__init__(*type_: Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList*]]], *default_value: Any = Undefined*, *description: Optional[str] = None*, *deprecation_reason: Optional[str] = None*, *out_name: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*InputValueDefinitionNode*] = None*) → None

ast_node: *graphql.language.ast.InputValueDefinitionNode* | None

default_value: Any

deprecation_reason: str | None

description: str | None

extensions: dict[str, Any]

out_name: str | None

to_kwargs() → GraphQLInputFieldKwargs

Get corresponding arguments.

type: Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]]

graphql.type.**GraphQLInputFieldMap**

alias of Dict[str, *GraphQLInputField*]

graphql.type.**GraphQLInputType**

alias of Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]]

graphql.type.**GraphQLLeafType**

alias of Union[*GraphQLScalarType*, *GraphQLEnumType*]

class graphql.type.**GraphQLNamedType**(*name: str*, **_args: Any*, ***_kwargs: Any*)

Bases: *GraphQLType*

Base class for all GraphQL named types

__init__(*name: str*, *description: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*TypeDefinitionNode*] = None*, *extension_ast_nodes: Optional[Collection[*TypeExtensionNode*]] = None*) → None

ast_node: *graphql.language.ast.TypeDefinitionNode* | None

description: str | None

extension_ast_nodes: tuple[*graphql.language.ast.TypeExtensionNode*, ...]

extensions: dict[str, Any]

name: str

reserved_types: Mapping[str, *GraphQLNamedType*] = {'Boolean': <GraphQLScalarType 'Boolean'>, 'Float': <GraphQLScalarType 'Float'>, 'ID': <GraphQLScalarType 'ID'>, 'Int': <GraphQLScalarType 'Int'>, 'String': <GraphQLScalarType 'String'>, '__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}

to_kwargs() → GraphQLNamedTypeKwargs

Get corresponding arguments.

graphql.type.**GraphQLNullableType**

alias of Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]

graphql.type.**GraphQLOutputType**

alias of Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*]]]

class graphql.type.**GraphQLType**

Bases: object

Base class for all GraphQL types

__init__()

class graphql.type.**GraphQLWrappingType**(*type_: GT*)

Bases: *GraphQLType*, Generic[GT]

Base class for all GraphQL wrapping types

__init__(*type_: GT*) → None

of_type: GT

graphql.type.**Thunk**

alias of Union[Callable[[], T], T]

graphql.type.**ThunkCollection**

alias of Union[Callable[[], Collection[T]], Collection[T]]

graphql.type.**ThunkMapping**

alias of Union[Callable[[], Mapping[str, T]], Mapping[str, T]]

Resolvers

graphql.type.**GraphQLFieldResolver**

alias of Callable[[...], Any]

graphql.type.**GraphQLIsTypeOfFn**

alias of Callable[[Any, *GraphQLResolveInfo*], Union[Awaitable[bool], bool]]

class graphql.type.**GraphQLResolveInfo**(*field_name: str*, *field_nodes: list[*FieldNode*]*, *return_type: GraphQLOutputType*, *parent_type:* GraphQLObjectType, *path:* Path, *schema:* GraphQLSchema, *fragments: dict[str,* FragmentDefinitionNode*]*, *root_value: Any*, *operation:* OperationDefinitionNode, *variable_values: dict[str, Any]*, *context: Any*, *is_awaitable: Callable[[Any], bool]*)

Bases: NamedTuple

Collection of information passed to the resolvers.

This is always passed as the first argument to the resolvers.

Note that contrary to the JavaScript implementation, the context (commonly used to represent an authenticated user, or request-specific caches) is included here and not passed as an additional argument.

__init__()

context: Any

Alias for field number 10

count(*value*, */*)

Return number of occurrences of value.

field_name: str

Alias for field number 0

field_nodes: list[*FieldNode*]

Alias for field number 1

fragments: dict[str, *FragmentDefinitionNode*]

Alias for field number 6

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

is_awaitable: Callable[[Any], bool]

Alias for field number 11

operation: *OperationDefinitionNode*

Alias for field number 8

parent_type: *GraphQLObjectType*

Alias for field number 3

path: *Path*

Alias for field number 4

return_type: **GraphQLOutputType**

Alias for field number 2

root_value: **Any**

Alias for field number 7

schema: *GraphQLSchema*

Alias for field number 5

variable_values: **dict[str, Any]**

Alias for field number 9

graphql.type.**GraphQLTypeResolver**

alias of `Callable[[Any, GraphQLResolveInfo, GraphQLAbstractType], Optional[Union[Awaitable[Optional[str]], str]]]`

Directives

Predicates

graphql.type.**is_directive**(*directive: Any*) → TypeGuard[*GraphQLDirective*]

Check whether this is a GraphQL directive.

graphql.type.**is_specified_directive**(*directive:* GraphQLDirective) → bool

Check whether the given directive is one of the specified directives.

Definitions

class graphql.type.**GraphQLDirective**(*name: str*, *locations: Collection[*DirectiveLocation*]*, *args: Optional[dict[str,* graphql.type.definition.GraphQLArgument*]] = None*, *is_repeatable: bool = False*, *description: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*DirectiveDefinitionNode*] = None*)

Bases: `object`

GraphQL Directive

Directives are used by the GraphQL runtime as a way of modifying execution behavior. Type system creators will usually not create these directly.

__init__(*name: str*, *locations: Collection[*DirectiveLocation*]*, *args: Optional[dict[str,* graphql.type.definition.GraphQLArgument*]] = None*, *is_repeatable: bool = False*, *description: Optional[str] = None*, *extensions: Optional[dict[str, Any]] = None*, *ast_node: Optional[*DirectiveDefinitionNode*] = None*) → None

args: dict[str, *graphql.type.definition.GraphQLArgument*]

ast_node: *graphql.language.ast.DirectiveDefinitionNode* | None

description: str | None

extensions: dict[str, Any]

is_repeatable: bool

locations: tuple[*graphql.language.directive_locations.DirectiveLocation*, ...]

name: str

to_kwargs() → GraphQLDirectiveKwargs

Get corresponding arguments.

graphql.type.**GraphQLIncludeDirective**

alias of <GraphQLDirective(@include)>

graphql.type.**GraphQLSkipDirective**

alias of <GraphQLDirective(@skip)>

graphql.type.**GraphQLDeferDirective**

alias of <GraphQLDirective(@defer)>

graphql.type.**GraphQLStreamDirective**

alias of <GraphQLDirective(@stream)>

graphql.type.**GraphQLDeprecatedDirective**

alias of <GraphQLDirective(@deprecated)>

graphql.type.**GraphQLSpecifiedByDirective**

alias of <GraphQLDirective(@specifiedBy)>

graphql.type.**specified_directives**

A tuple with all directives from the GraphQL specification

graphql.type.**DEFAULT_DEPRECATION_REASON = 'No longer supported'**

String constant that can be used as the default value for deprecation_reason.

Introspection

Predicates

graphql.type.**is_introspection_type**(*type_:* GraphQLNamedType) → bool

Check whether the given named GraphQL type is an introspection type.

Definitions

class graphql.type.**TypeKind**(*value*)

Bases: Enum

Kinds of types

ENUM = 'enum'

INPUT_OBJECT = 'input object'

INTERFACE = 'interface'

LIST = 'list'

NON_NULL = 'non-null'

OBJECT = 'object'

SCALAR = 'scalar'

UNION = 'union'

graphql.type.**TypeMetaFieldDef**

alias of <GraphQLField <GraphQLObjectType '__Type'>>

graphql.type.**TypeNameMetaFieldDef**

alias of <GraphQLField <GraphQLNonNull <GraphQLScalarType 'String'>>>

graphql.type.**SchemaMetaFieldDef**

alias of <GraphQLField <GraphQLNonNull <GraphQLObjectType '__Schema'>>>

graphql.type.**introspection_types**

This is a mapping containing all introspection types with their names as keys

Scalars

Predicates

graphql.type.**is_specified_scalar_type**(*type_:* GraphQLNamedType) → TypeGuard[*GraphQLScalarType*]

Check whether the given named GraphQL type is a specified scalar type.

Definitions

graphql.type.**GraphQLBoolean**

alias of <GraphQLScalarType 'Boolean'>

graphql.type.**GraphQLFloat**

alias of <GraphQLScalarType 'Float'>

graphql.type.**GraphQLID**

alias of <GraphQLScalarType 'ID'>

graphql.type.**GraphQLInt**

alias of <GraphQLScalarType 'Int'>

graphql.type.**GraphQLString**

alias of <GraphQLScalarType 'String'>

graphql.type.**GRAPHQL_MAX_INT**

Maximum possible Int value as per GraphQL Spec (32-bit signed integer)

graphql.type.**GRAPHQL_MIN_INT**

Minimum possible Int value as per GraphQL Spec (32-bit signed integer)

Schema

Predicates

graphql.type.**is_schema**(*schema: Any*) → TypeGuard[*GraphQLSchema*]

Check whether this is a GraphQL schema.

Definitions

class graphql.type.**GraphQLSchema**(*query:* GraphQLObjectType | *None* = *None*, *mutation:* GraphQLObjectType | *None* = *None*, *subscription:* GraphQLObjectType | *None* = *None*, *types: Collection[*GraphQLNamedType*]* | *None* = *None*, *directives: Collection[*GraphQLDirective*]* | *None* = *None*, *description: str* | *None* = *None*, *extensions: dict[str, Any]* | *None* = *None*, *ast_node:* ast.SchemaDefinitionNode | *None* = *None*, *extension_ast_nodes: Collection[*ast.SchemaExtensionNode*]* | *None* = *None*, *assume_valid: bool* = *False*)

Bases: `object`

Schema Definition

A Schema is created by supplying the root types of each type of operation, query and mutation (optional). A schema definition is then supplied to the validator and executor.

Schemas should be considered immutable once they are created. If you want to modify a schema, modify the result of the `to_kwargs()` method and recreate the schema.

Example:

```
MyAppSchema = GraphQLSchema(
  query=MyAppQueryRootType,
  mutation=MyAppMutationRootType)
```

Note: When the schema is constructed, by default only the types that are reachable by traversing the root types are included, other types must be explicitly referenced.

Example:

```
character_interface = GraphQLInterfaceType('Character', ...)

human_type = GraphQLObjectType(
    'Human', interfaces=[character_interface], ...)

droid_type = GraphQLObjectType(
```

(continues on next page)

(continued from previous page)

```
    'Droid', interfaces: [character_interface], ...)

schema = GraphQLSchema(
    query=GraphQLObjectType('Query',
        fields={'hero': GraphQLField(character_interface, ....)}),
    ...
    # Since this schema references only the `Character` interface it's
    # necessary to explicitly list the types that implement it if
    # you want them to be included in the final schema.
    types=[human_type, droid_type])
```

Note: If a list of `directives` is provided to GraphQLSchema, that will be the exact list of directives represented and allowed. If `directives` is not provided, then a default set of the specified directives (e.g. @include and @skip) will be used. If you wish to provide *additional* directives to these specified directives, you must explicitly declare them. Example:

```
MyAppSchema = GraphQLSchema(
  ...
  directives=specified_directives + [my_custom_directive])
```

__init__(*query:* GraphQLObjectType | *None* = *None*, *mutation:* GraphQLObjectType | *None* = *None*, *subscription:* GraphQLObjectType | *None* = *None*, *types: Collection[*GraphQLNamedType*]* | *None* = *None*, *directives: Collection[*GraphQLDirective*]* | *None* = *None*, *description: str* | *None* = *None*, *extensions: dict[str, Any]* | *None* = *None*, *ast_node:* ast.SchemaDefinitionNode | *None* = *None*, *extension_ast_nodes: Collection[*ast.SchemaExtensionNode*]* | *None* = *None*, *assume_valid: bool* = *False*) → None

Initialize GraphQL schema.

If this schema was built from a source known to be valid, then it may be marked with `assume_valid` to avoid an additional type system validation.

ast_node: *ast.SchemaDefinitionNode* **| None**

description: **str | None**

directives: **tuple[***GraphQLDirective***, ...]**

extension_ast_nodes: **tuple[***ast.SchemaExtensionNode***, ...]**

extensions: **dict[str, Any]**

get_directive(*name: str*) → *graphql.type.directives.GraphQLDirective* | None

Get the directive with the given name.

get_field(*parent_type: Union[*GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType*]*, *field_name: str*) → *graphql.type.definition.GraphQLField* | None

Get field of a given type with the given name.

This method looks up the field on the given type definition. It has special casing for the three introspection fields, *__schema*, *__type* and *__typename*.

__typename is special because it can always be queried as a field, even in situations where no other fields are allowed, like on a Union.

__schema and *__type* could get automatically added to the query type, but that would require mutating type definitions, which would cause issues.

get_implementations(*interface_type:* GraphQLInterfaceType) → InterfaceImplementations

Get implementations for the given interface type.

get_possible_types(*abstract_type: Union[*GraphQLInterfaceType, GraphQLUnionType*]*) → list[*graphql.type.definition.GraphQLObjectType*]

Get list of all possible concrete types for given abstract type.

get_root_type(*operation:* OperationType) → *GraphQLObjectType* | None

Get the root type.

get_type(*name: str*) → *graphql.type.definition.GraphQLNamedType* | None

Get the type with the given name.

is_sub_type(*abstract_type: Union[*GraphQLInterfaceType, GraphQLUnionType*]*, *maybe_sub_type:* GraphQLNamedType) → bool

Check whether a type is a subtype of a given abstract type.

mutation_type: *GraphQLObjectType* | None

query_type: *GraphQLObjectType* | None

subscription_type: *GraphQLObjectType* | None

to_kwargs() → GraphQLSchemaKwargs

Get corresponding arguments.

type_map: TypeMap

property validation_errors: list[*GraphQLError*] | None

Get validation errors.

Validate

Functions

graphql.type.**validate_schema**(*schema:* GraphQLSchema) → list[*graphql.error.graphql_error.GraphQLError*]

Validate a GraphQL schema.

Implements the "Type Validation" sub-sections of the specification's "Type System" section.

Validation runs synchronously, returning a list of encountered errors, or an empty list if no errors were encountered and the Schema is valid.

Assertions

graphql.type.**assert_valid_schema**(*schema:* GraphQLSchema) → None

Utility function which asserts a schema is valid.

Throws a TypeError if the schema is invalid.

Other

Thunk Handling

graphql.type.**resolve_thunk**(*thunk: Union[Callable[[], T], T]*) → T

Resolve the given thunk.

Used while defining GraphQL types to allow for circular references in otherwise immutable type definitions.

Assertions

graphql.type.**assert_name**(*name: str*) → str

Uphold the spec rules about naming.

graphql.type.**assert_enum_value_name**(*name: str*) → str

Uphold the spec rules about naming enum values.

Utilities

GraphQL Utilities

The *graphql.utilities* package contains common useful computations to use with the GraphQL language and type objects.

The GraphQL query recommended for a full schema introspection:

graphql.utilities.**get_introspection_query**(*descriptions: bool = True*, *specified_by_url: bool = False*, *directive_is_repeatable: bool = False*, *schema_description: bool = False*, *input_value_deprecation: bool = False*) → str

Get a query for introspection.

Optionally, you can exclude descriptions, include specification URLs, include repeatability of directives, and specify whether to include the schema description as well.

class graphql.utilities.**IntrospectionQuery**

Bases: TypedDict

The root typed dictionary for schema introspections.

Get the target Operation from a Document:

graphql.utilities.**get_operation_ast**(*document_ast: DocumentNode*, *operation_name: Optional[str] = None*) → *graphql.language.ast.OperationDefinitionNode* | None

Get operation AST node.

Returns an operation AST given a document AST and optionally an operation name. If a name is not provided, an operation is only returned if only one is provided in the document.

Convert a GraphQLSchema to an IntrospectionQuery:

graphql.utilities.**introspection_from_schema**(*schema: GraphQLSchema*, *descriptions: bool = True*, *specified_by_url: bool = True*, *directive_is_repeatable: bool = True*, *schema_description: bool = True*, *input_value_deprecation: bool = True*) → *IntrospectionQuery*

Build an IntrospectionQuery from a GraphQLSchema

IntrospectionQuery is useful for utilities that care about type and field relationships, but do not need to traverse through those relationships.

This is the inverse of build_client_schema. The primary use case is outside of the server context, for instance when doing schema comparisons.

Build a GraphQLSchema from an introspection result:

graphql.utilities.**build_client_schema**(*introspection:* IntrospectionQuery, *assume_valid: bool = False*) → *GraphQLSchema*

Build a GraphQLSchema for use by client tools.

Given the result of a client running the introspection query, creates and returns a GraphQLSchema instance which can be then used with all GraphQL-core 3 tools, but cannot be used to execute a query, as introspection does not represent the "resolver", "parse" or "serialize" functions or any other server-internal mechanisms.

This function expects a complete introspection result. Don't forget to check the "errors" field of a server response before calling this function.

Build a GraphQLSchema from GraphQL Schema language:

graphql.utilities.**build_ast_schema**(*document_ast:* DocumentNode, *assume_valid: bool = False*, *assume_valid_sdl: bool = False*) → *GraphQLSchema*

Build a GraphQL Schema from a given AST.

This takes the ast of a schema document produced by the parse function in src/language/parser.py.

If no schema definition is provided, then it will look for types named Query, Mutation and Subscription.

Given that AST it constructs a GraphQLSchema. The resulting schema has no resolve methods, so execution will use default resolvers.

When building a schema from a GraphQL service's introspection result, it might be safe to assume the schema is valid. Set `assume_valid` to `True` to assume the produced schema is valid. Set `assume_valid_sdl` to `True` to assume it is already a valid SDL document.

graphql.utilities.**build_schema**(*source: str* | graphql.language.source.Source, *assume_valid: bool = False*, *assume_valid_sdl: bool = False*, *no_location: bool = False*, *allow_legacy_fragment_variables: bool = False*) → *GraphQLSchema*

Build a GraphQLSchema directly from a source document.

Extend an existing GraphQLSchema from a parsed GraphQL Schema language AST:

graphql.utilities.**extend_schema**(*schema:* GraphQLSchema, *document_ast:* DocumentNode, *assume_valid: bool = False*, *assume_valid_sdl: bool = False*) → *GraphQLSchema*

Extend the schema with extensions from a given document.

Produces a new schema given an existing schema and a document which may contain GraphQL type extensions and definitions. The original schema will remain unaltered.

Because a schema represents a graph of references, a schema cannot be extended without effectively making an entire copy. We do not know until it's too late if subgraphs remain unchanged.

This algorithm copies the provided schema, applying extensions while producing the copy. The original schema remains unaltered.

When extending a schema with a known valid extension, it might be safe to assume the schema is valid. Set `assume_valid` to `True` to assume the produced schema is valid. Set `assume_valid_sdl` to `True` to assume it is already a valid SDL document.

Sort a GraphQLSchema:

graphql.utilities.**lexicographic_sort_schema**(*schema:* GraphQLSchema) → *GraphQLSchema*

Sort GraphQLSchema.

This function returns a sorted copy of the given GraphQLSchema.

Print a GraphQLSchema to GraphQL Schema language:

graphql.utilities.**print_schema**(*schema:* GraphQLSchema) → str

Print the given GraphQL schema in SDL format.

graphql.utilities.**print_type**(*type_:* GraphQLNamedType) → str

Print a named GraphQL type.

graphql.utilities.**print_directive**(*directive:* GraphQLDirective) → str

Print a GraphQL directive.

graphql.utilities.**print_introspection_schema**(*schema:* GraphQLSchema) → str

Print the built-in introspection schema in SDL format.

Create a GraphQLType from a GraphQL language AST:

graphql.utilities.**type_from_ast**(*schema:* GraphQLSchema, *type_node:* NamedTypeNode) → *graphql.type.definition.GraphQLNamedType* | None

graphql.utilities.**type_from_ast**(*schema:* GraphQLSchema, *type_node:* ListTypeNode) → *graphql.type.definition.GraphQLList* | None

graphql.utilities.**type_from_ast**(*schema:* GraphQLSchema, *type_node:* NonNullTypeNode) → *graphql.type.definition.GraphQLNonNull* | None

graphql.utilities.**type_from_ast**(*schema:* GraphQLSchema, *type_node:* TypeNode) → *graphql.type.definition.GraphQLType* | None

Get the GraphQL type definition from an AST node.

Given a Schema and an AST node describing a type, return a GraphQLType definition which applies to that type. For example, if provided the parsed AST node for `[User]`, a GraphQLList instance will be returned, containing the type called "User" found in the schema. If a type called "User" is not found in the schema, then None will be returned.

Convert a language AST to a dictionary:

graphql.utilities.**ast_to_dict**(*node:* Node, *locations: bool = False*, *cache: dict[*graphql.language.ast.Node*, Any]* | *None = None*) → dict

graphql.utilities.**ast_to_dict**(*node: Collection[*Node*]*, *locations: bool = False*, *cache: dict[*graphql.language.ast.Node*, Any]* | *None = None*) → list[*graphql.language.ast.Node*]

graphql.utilities.**ast_to_dict**(*node:* OperationType, *locations: bool = False*, *cache: dict[*graphql.language.ast.Node*, Any]* | *None = None*) → str

Convert a language AST to a nested Python dictionary.

Set *location* to True in order to get the locations as well.

Create a Python value from a GraphQL language AST with a type:

graphql.utilities.**value_from_ast**(*value_node:* graphql.language.ast.ValueNode | *None*, *type_: Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList*]]]*, *variables: Optional[dict[str, Any]] = None*) → Any

Produce a Python value given a GraphQL Value AST.

A GraphQL type must be provided, which will be used to interpret different GraphQL Value literals.

Returns `Undefined` when the value could not be validly coerced according to the provided type.

GraphQL Value	JSON Value	Python Value
Input Object	Object	dict
List	Array	list
Boolean	Boolean	bool
String	String	str
Int / Float	Number	int / float
Enum Value	Mixed	Any
NullValue	null	None

Create a Python value from a GraphQL language AST without a type:

graphql.utilities.**value_from_ast_untyped**(*value_node:* ValueNode, *variables: dict[str, Any] | None = None*) → Any

Produce a Python value given a GraphQL Value AST.

Unlike *value_from_ast()*, no type is provided. The resulting Python value will reflect the provided GraphQL value AST.

GraphQL Value	JSON Value	Python Value
Input Object	Object	dict
List	Array	list
Boolean	Boolean	bool
String / Enum	String	str
Int / Float	Number	int / float
Null	null	None

Create a GraphQL language AST from a Python value:

graphql.utilities.**ast_from_value**(*value: Any*, *type_: Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull*[Union[*GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList*]]]*) → Optional[Union[*IntValueNode*, *FloatValueNode*, *StringValueNode*, *BooleanValueNode*, *NullValueNode*, *EnumValueNode*, *ConstListValueNode*, *ConstObjectValueNode*]]

Produce a GraphQL Value AST given a Python object.

This function will match Python/JSON values to GraphQL AST schema format by using the suggested GraphQLInputType. For example:

```
ast_from_value('value', GraphQLString)
```

A GraphQL type must be provided, which will be used to interpret different Python values.

JSON Value	GraphQL Value
Object	Input Object
Array	List
Boolean	Boolean
String	String / Enum Value
Number	Int / Float
Mixed	Enum Value
null	NullValue

A helper to use within recursive-descent visitors which need to be aware of the GraphQL type system:

class graphql.utilities.**TypeInfo**(*schema:* GraphQLSchema, *initial_type: Optional[*GraphQLType*] = None*, *get_field_def_fn: Optional[Callable[[*GraphQLSchema*, Union[*GraphQLObjectType*,* GraphQLInterfaceType*,* GraphQLUnionType*],* FieldNode*], Optional[*GraphQLField*]]] = None*)

Bases: object

Utility class for keeping track of type definitions.

TypeInfo is a utility class which, given a GraphQL schema, can keep track of the current field and type definitions at any point in a GraphQL document AST during a recursive descent by calling *enter(node)* and *leave(node)*.

__init__(*schema:* GraphQLSchema, *initial_type: Optional[*GraphQLType*] = None*, *get_field_def_fn: Optional[Callable[[*GraphQLSchema*, Union[*GraphQLObjectType*,* GraphQLInterfaceType*,* GraphQLUnionType*],* FieldNode*], Optional[*GraphQLField*]]] = None*) → None

Initialize the TypeInfo for the given GraphQL schema.

Initial type may be provided in rare cases to facilitate traversals beginning somewhere other than documents.

The optional last parameter is deprecated and will be removed in v3.3.

enter(*node:* Node) → None

enter_argument(*node:* ArgumentNode) → None

enter_directive(*node:* DirectiveNode) → None

enter_enum_value(*node:* EnumValueNode) → None

enter_field(*node:* FieldNode) → None

enter_fragment_definition(*node:* InlineFragmentNode) → None

enter_inline_fragment(*node:* InlineFragmentNode) → None

enter_list_value(*_node:* ListValueNode) → None

enter_object_field(*node:* ObjectFieldNode) → None

enter_operation_definition(*node:* OperationDefinitionNode) → None

enter_selection_set(*_node:* SelectionSetNode) → None

enter_variable_definition(*node:* VariableDefinitionNode) → None

get_argument() → *graphql.type.definition.GraphQLArgument* | None

get_default_value() → Any

get_directive() → *graphql.type.directives.GraphQLDirective* | None

get_enum_value() → *graphql.type.definition.GraphQLEnumValue* | None

get_field_def() → *graphql.type.definition.GraphQLField* | None

get_input_type() → Optional[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]]]

get_parent_input_type() → Optional[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLEnumType*, *GraphQLInputObjectType*, *GraphQLList*]]]]

get_parent_type() → Optional[Union[*GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*]]

get_type() → Optional[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*, *GraphQLNonNull*[Union[*GraphQLScalarType*, *GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*, *GraphQLEnumType*, *GraphQLList*]]]]

leave(*node:* Node) → None

leave_argument() → None

leave_directive() → None

leave_enum_value() → None

leave_field() → None

leave_fragment_definition() → None

leave_inline_fragment() → None

leave_list_value() → None

leave_object_field() → None

leave_operation_definition() → None

leave_selection_set() → None

leave_variable_definition() → None

class graphql.utilities.**TypeInfoVisitor**(*type_info:* TypeInfo, *visitor:* Visitor)

Bases: *Visitor*

A visitor which maintains a provided TypeInfo.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*type_info:* TypeInfo, *visitor:* Visitor) → None

enter(*node:* Node, **args: Any*) → Any

enter_leave_map: dict[str, EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

leave(*node:* Node, **args: Any*) → Any

Coerce a Python value to a GraphQL type, or produce errors:

graphql.utilities.**coerce_input_value**(*input_value: ~typing.Any, type_:*
~typing.Union[~graphql.type.definition.GraphQLScalarType,
~graphql.type.definition.GraphQLEnumType,
~graphql.type.definition.GraphQLInputObjectType,
~graphql.type.definition.GraphQLList,
~graphql.type.definition.GraphQLNonNull[~typing.Union[~graphql.type.definition.Gra
~graphql.type.definition.GraphQLEnumType,
~graphql.type.definition.GraphQLInputObjectType,
~graphql.type.definition.GraphQLList]]], on_error:
~typing.Callable[[~typing.List[~typing.Union[str, int]],
~typing.Any, ~graphql.error.graphql_error.GraphQLError], None]
= <function default_on_error>, path:
~typing.Optional[~graphql.pyutils.path.Path] = None) → Any

Coerce a Python value given a GraphQL Input Type.

Concatenate multiple ASTs together:

graphql.utilities.**concat_ast**(*asts: Collection[*DocumentNode*]*) → *DocumentNode*

Concat ASTs.

Provided a collection of ASTs, presumably each from different files, concatenate the ASTs together into batched AST, useful for validating many GraphQL source files which together represent one conceptual application.

Separate an AST into an AST per Operation:

graphql.utilities.**separate_operations**(*document_ast:* DocumentNode) → dict[str,
graphql.language.ast.DocumentNode]

Separate operations in a given AST document.

This function accepts a single AST document which may contain many operations and fragments and returns a collection of AST documents each of which contains a single operation as well the fragment definitions it refers to.

Strip characters that are not significant to the validity or execution of a GraphQL document:

graphql.utilities.**strip_ignored_characters**(*source: str* | graphql.language.source.Source) → str

Strip characters that are ignored anyway.

Strips characters that are not significant to the validity or execution of a GraphQL document:

- UnicodeBOM
- WhiteSpace
- LineTerminator

- Comment
- Comma
- BlockString indentation

Note: It is required to have a delimiter character between neighboring non-punctuator tokes and this function always uses single space as delimiter.

It is guaranteed that both input and output documents if parsed would result in the exact same AST except for nodes location.

Warning: It is guaranteed that this function will always produce stable results. However, it's not guaranteed that it will stay the same between different releases due to bugfixes or changes in the GraphQL specification.

Query example:

```
query SomeQuery($foo: String!, $bar: String) {
  someField(foo: $foo, bar: $bar) {
    a
    b {
      c
      d
    }
  }
}
```

Becomes:

```
query SomeQuery($foo:String!$bar:String){someField(foo:$foo bar:$bar){a b{c d}}}
```

SDL example:

```
"""
Type description
"""
type Foo {
  """
  Field description
  """
  bar: String
}
```

Becomes:

```
"""Type description""" type Foo{"""Field description""" bar:String}
```

Comparators for types:

graphql.utilities.**is_equal_type**(*type_a:* GraphQLType, *type_b:* GraphQLType) → bool

Check whether two types are equal.

Provided two types, return true if the types are equal (invariant).

graphql.utilities.**is_type_sub_type_of**(*schema:* GraphQLSchema, *maybe_subtype:* GraphQLType, *super_type:* GraphQLType) → bool

Check whether a type is subtype of another type in a given schema.

Provided a type and a super type, return true if the first type is either equal or a subset of the second super type (covariant).

graphql.utilities.**do_types_overlap**(*schema:* GraphQLSchema, *type_a: Union[*GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType*]*, *type_b:* *Union[*GraphQLObjectType, GraphQLInterfaceType, GraphQLUnionType*]*) → bool

Check whether two types overlap in a given schema.

Provided two composite types, determine if they "overlap". Two composite types overlap when the Sets of possible concrete types for each intersect.

This is often used to determine if a fragment of a given type could possibly be visited in a context of another type.

This function is commutative.

Compare two GraphQLSchemas and detect breaking changes:

graphql.utilities.**find_breaking_changes**(*old_schema:* GraphQLSchema, *new_schema:* GraphQLSchema) → list[*graphql.utilities.find_breaking_changes.BreakingChange*]

Find breaking changes.

Given two schemas, returns a list containing descriptions of all the types of breaking changes covered by the other functions down below.

graphql.utilities.**find_dangerous_changes**(*old_schema:* GraphQLSchema, *new_schema:* GraphQLSchema) → list[*graphql.utilities.find_breaking_changes.DangerousChange*]

Find dangerous changes.

Given two schemas, returns a list containing descriptions of all the types of potentially dangerous changes covered by the other functions down below.

class graphql.utilities.**BreakingChange**(*type:* BreakingChangeType, *description: str*)

Bases: NamedTuple

Type and description of a breaking change

__init__()

count(*value*, */*)

Return number of occurrences of value.

description: str

Alias for field number 1

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

type: *BreakingChangeType*

Alias for field number 0

class graphql.utilities.**BreakingChangeType**(*value*)

Bases: Enum

Types of breaking changes

ARG_CHANGED_KIND = 42

ARG_REMOVED = 41

DIRECTIVE_ARG_REMOVED = 51

DIRECTIVE_LOCATION_REMOVED = 54

DIRECTIVE_REMOVED = 50

DIRECTIVE_REPEATABLE_REMOVED = 53

FIELD_CHANGED_KIND = 31

FIELD_REMOVED = 30

IMPLEMENTED_INTERFACE_REMOVED = 23

REQUIRED_ARG_ADDED = 40

REQUIRED_DIRECTIVE_ARG_ADDED = 52

REQUIRED_INPUT_FIELD_ADDED = 22

TYPE_CHANGED_KIND = 11

TYPE_REMOVED = 10

TYPE_REMOVED_FROM_UNION = 20

VALUE_REMOVED_FROM_ENUM = 21

class graphql.utilities.**DangerousChange**(*type:* DangerousChangeType, *description: str*)

Bases: NamedTuple

Type and description of a dangerous change

__init__()

count(*value*, */*)

Return number of occurrences of value.

description: str

Alias for field number 1

index(*value*, *start=0*, *stop=9223372036854775807*, */*)

Return first index of value.

Raises ValueError if the value is not present.

type: *DangerousChangeType*

Alias for field number 0

class graphql.utilities.**DangerousChangeType**(*value*)

Bases: Enum

Types of dangerous changes

ARG_DEFAULT_VALUE_CHANGE = 65

IMPLEMENTED_INTERFACE_ADDED = 64

OPTIONAL_ARG_ADDED = 63

OPTIONAL_INPUT_FIELD_ADDED = 62

TYPE_ADDED_TO_UNION = 61

VALUE_ADDED_TO_ENUM = 60

Validation

GraphQL Validation

The *graphql.validation* package fulfills the Validation phase of fulfilling a GraphQL result.

graphql.validation.**validate**(*schema:* GraphQLSchema, *document_ast:* DocumentNode, *rules: Collection[type[*ASTValidationRule*]] | None = None*, *max_errors: int | None = None*, *type_info:* TypeInfo | *None = None*) → list[*GraphQLError*]

Implements the "Validation" section of the spec.

Validation runs synchronously, returning a list of encountered errors, or an empty list if no errors were encountered and the document is valid.

A list of specific validation rules may be provided. If not provided, the default list of rules defined by the GraphQL specification will be used.

Each validation rule is a ValidationRule object which is a visitor object that holds a ValidationContext (see the language/visitor API). Visitor methods are expected to return GraphQLErrors, or lists of GraphQLErrors when invalid.

Validate will stop validation after a `max_errors` limit has been reached. Attackers can send pathologically invalid queries to induce a DoS attack, so by default `max_errors` set to 100 errors.

Providing a custom TypeInfo instance is deprecated and will be removed in v3.3.

class graphql.validation.**ASTValidationContext**(*ast:* DocumentNode, *on_error: Callable[[*GraphQLError*], None]*)

Bases: `object`

Utility class providing a context for validation of an AST.

An instance of this class is passed as the context attribute to all Validators, allowing access to commonly useful contextual information from within a validation rule.

__init__(*ast:* DocumentNode, *on_error: Callable[[*GraphQLError*], None]*) → None

document: *DocumentNode*

get_fragment(*name: str*) → *graphql.language.ast.FragmentDefinitionNode* | None

get_fragment_spreads(*node:* SelectionSetNode) → list[*graphql.language.ast.FragmentSpreadNode*]

get_recursively_referenced_fragments(*operation:* OperationDefinitionNode) → list[*graphql.language.ast.FragmentDefinitionNode*]

on_error(*error:* GraphQLError) → None

report_error(*error:* GraphQLError) → None

class graphql.validation.**ASTValidationRule**(*context:* ASTValidationContext)

Bases: *Visitor*

Visitor for validation of an AST.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**SDLValidationContext**(*ast:* DocumentNode, *schema:* GraphQLSchema | *None*, *on_error: Callable[[*GraphQLError*], None]*)

Bases: *ASTValidationContext*

Utility class providing a context for validation of an SDL AST.

An instance of this class is passed as the context attribute to all Validators, allowing access to commonly useful contextual information from within a validation rule.

__init__(*ast:* DocumentNode, *schema:* GraphQLSchema | *None*, *on_error: Callable[[*GraphQLError*], None]*) → None

document: *DocumentNode*

get_fragment(*name: str*) → *graphql.language.ast.FragmentDefinitionNode* | None

get_fragment_spreads(*node:* SelectionSetNode) → list[*graphql.language.ast.FragmentSpreadNode*]

get_recursively_referenced_fragments(*operation:* OperationDefinitionNode) → list[*graphql.language.ast.FragmentDefinitionNode*]

on_error(*error:* GraphQLError) → None

report_error(*error:* GraphQLError) → None

schema: *GraphQLSchema* | **None**

class graphql.validation.**SDLValidationRule**(*context:* SDLValidationContext)

Bases: *ASTValidationRule*

Visitor for validation of an SDL AST.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

context: *SDLValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**ValidationContext**(*schema:* GraphQLSchema, *ast:* DocumentNode, *type_info:* TypeInfo, *on_error: Callable[[*GraphQLError*], None]*)

Bases: *ASTValidationContext*

Utility class providing a context for validation using a GraphQL schema.

An instance of this class is passed as the context attribute to all Validators, allowing access to commonly useful contextual information from within a validation rule.

__init__(*schema:* GraphQLSchema, *ast:* DocumentNode, *type_info:* TypeInfo, *on_error: Callable[[*GraphQLError*], None]*) → None

document: *DocumentNode*

get_argument() → *GraphQLArgument* | None

get_directive() → *GraphQLDirective* | None

get_enum_value() → *GraphQLEnumValue* | None

get_field_def() → *GraphQLField* | None

get_fragment(*name: str*) → *graphql.language.ast.FragmentDefinitionNode* | None

get_fragment_spreads(*node:* SelectionSetNode) → list[*graphql.language.ast.FragmentSpreadNode*]

get_input_type() → GraphQLInputType | None

get_parent_input_type() → GraphQLInputType | None

get_parent_type() → GraphQLCompositeType | None

get_recursive_variable_usages(*operation:* OperationDefinitionNode) → list[graphql.validation.validation_context.VariableUsage]

get_recursively_referenced_fragments(*operation:* OperationDefinitionNode) → list[*graphql.language.ast.FragmentDefinitionNode*]

get_type() → GraphQLOutputType | None

get_variable_usages(*node: Union[*OperationDefinitionNode, FragmentDefinitionNode*]*) → list[graphql.validation.validation_context.VariableUsage]

on_error(*error:* GraphQLError) → None

report_error(*error:* GraphQLError) → None

schema: *GraphQLSchema*

class graphql.validation.**ValidationRule**(*context:* ValidationContext)

Bases: *ASTValidationRule*

Visitor for validation using a GraphQL schema.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Rules

graphql.validation.**specified_rules**

A tuple with all validation rules defined by the GraphQL specification

Spec Section: "Executable Definitions"

class graphql.validation.**ExecutableDefinitionsRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

Executable definitions

A GraphQL document is only valid for execution if all definitions are either operation or fragment definitions.

See https://spec.graphql.org/draft/#sec-Executable-Definitions

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

enter_document(*node:* DocumentNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Field Selections on Objects, Interfaces, and Unions Types"

class graphql.validation.**FieldsOnCorrectTypeRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Fields on correct type

A GraphQL document is only valid if all fields selected are defined by the parent type, or are an allowed meta field such as `__typename`.

See https://spec.graphql.org/draft/#sec-Field-Selections

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_field(*node:* FieldNode, **_args: Any*) → None

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Fragments on Composite Types"

class graphql.validation.**FragmentsOnCompositeTypesRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Fragments on composite type

Fragments use a type condition to determine if they apply, since fragments can only be spread into a composite type (object, interface, or union), the type condition must also be a composite type.

See https://spec.graphql.org/draft/#sec-Fragments-On-Composite-Types

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_fragment_definition(*node:* FragmentDefinitionNode, **_args: Any*) → None

enter_inline_fragment(*node:* InlineFragmentNode, **_args: Any*) → None

enter_leave_map: dict[str, EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Argument Names"

class graphql.validation.**KnownArgumentNamesRule**(*context:* ValidationContext)

Bases: KnownArgumentNamesOnDirectivesRule

Known argument names

A GraphQL field is only valid if all supplied arguments are defined by that field.

See https://spec.graphql.org/draft/#sec-Argument-Names See https://spec.graphql.org/draft/#sec-Directives-Are-In-Valid-Locations

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_argument(*arg_node:* ArgumentNode, **args: Any*) → None

enter_directive(*directive_node:* DirectiveNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: dict[str, EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Directives Are Defined"

class graphql.validation.**KnownDirectivesRule**(*context:* graphql.validation.validation_context.ValidationContext | graphql.validation.validation_context.SDLValidationContext)

Bases: *ASTValidationRule*

Known directives

A GraphQL document is only valid if all `@directives` are known by the schema and legally positioned.

See https://spec.graphql.org/draft/#sec-Directives-Are-Defined

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* graphql.validation.validation_context.ValidationContext | graphql.validation.validation_context.SDLValidationContext) → None

context: *ValidationContext | SDLValidationContext*

enter_directive(*node:* DirectiveNode, *_key: Any*, *_parent: Any*, *_path: Any*, *ancestors: list[*graphql.language.ast.Node*]*) → None

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Fragment spread target defined"

class graphql.validation.**KnownFragmentNamesRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Known fragment names

A GraphQL document is only valid if all `...Fragment` fragment spreads refer to fragments defined in the same document.

See https://spec.graphql.org/draft/#sec-Fragment-spread-target-defined

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_fragment_spread(*node:* FragmentSpreadNode, **_args: Any*) → None

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Fragment Spread Type Existence"

class graphql.validation.**KnownTypeNamesRule**(*context:* graphql.validation.validation_context.ValidationContext | graphql.validation.validation_context.SDLValidationContext)

Bases: *ASTValidationRule*

Known type names

A GraphQL document is only valid if referenced types (specifically variable definitions and fragment conditions) are defined by the type schema.

See https://spec.graphql.org/draft/#sec-Fragment-Spread-Type-Existence

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* graphql.validation.validation_context.ValidationContext | graphql.validation.validation_context.SDLValidationContext) → None

context: *ASTValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_named_type(*node:* NamedTypeNode, *_key: Any*, *parent:* Node, *_path: Any*, *ancestors:* *list[*graphql.language.ast.Node*]*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Lone Anonymous Operation"

class graphql.validation.**LoneAnonymousOperationRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

Lone anonymous operation

A GraphQL document is only valid if when it contains an anonymous operation (the query short-hand) that it contains only that one operation definition.

See https://spec.graphql.org/draft/#sec-Lone-Anonymous-Operation

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

enter_document(*node:* DocumentNode, **_args: Any*) → None

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_operation_definition(*node:* OperationDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Fragments must not form cycles"

class graphql.validation.**NoFragmentCyclesRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

No fragment cycles

The graph of fragment spreads must not form any cycles including spreading itself. Otherwise an operation could infinitely spread or infinitely execute on cycles in the underlying data.

See https://spec.graphql.org/draft/#sec-Fragment-spreads-must-not-form-cycles

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

detect_cycle_recursive(*fragment:* FragmentDefinitionNode) → None

enter_fragment_definition(*node:* FragmentDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: dict[str, EnterLeaveVisitor]

static enter_operation_definition(**_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "All Variable Used Defined"

class graphql.validation.**NoUndefinedVariablesRule**(*context:* ValidationContext)

Bases: *ValidationRule*

No undefined variables

A GraphQL operation is only valid if all variables encountered, both directly and via fragment spreads, are defined by that operation.

See https://spec.graphql.org/draft/#sec-All-Variable-Uses-Defined

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_operation_definition(**_args: Any*) → None

enter_variable_definition(*node:* VariableDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

leave_operation_definition(*operation:* OperationDefinitionNode, **_args: Any*) → None

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Fragments must be used"

class graphql.validation.**NoUnusedFragmentsRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

No unused fragments

A GraphQL document is only valid if all fragment definitions are spread within operations, or spread within other fragments spread within operations.

See https://spec.graphql.org/draft/#sec-Fragments-Must-Be-Used

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

enter_fragment_definition(*node:* FragmentDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_operation_definition(*node:* OperationDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

leave_document(**_args: Any*) → None

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "All Variables Used"

class graphql.validation.**NoUnusedVariablesRule**(*context:* ValidationContext)

Bases: *ValidationRule*

No unused variables

A GraphQL operation is only valid if all variables defined by an operation are used, either directly or within a spread fragment.

See https://spec.graphql.org/draft/#sec-All-Variables-Used

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_operation_definition(**_args: Any*) → None

enter_variable_definition(*definition:* VariableDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

leave_operation_definition(*operation:* OperationDefinitionNode, **_args: Any*) → None

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Field Selection Merging"

class graphql.validation.**OverlappingFieldsCanBeMergedRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Overlapping fields can be merged

A selection set is only valid if all fields (including spreading any fragments) either correspond to distinct response names or can be merged without ambiguity.

See https://spec.graphql.org/draft/#sec-Field-Selection-Merging

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_selection_set(*selection_set:* SelectionSetNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Fragment spread is possible"

class graphql.validation.**PossibleFragmentSpreadsRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Possible fragment spread

A fragment spread is only valid if the type condition could ever possibly be true: if there is a non-empty intersection of the possible parent types, and possible types which pass the type condition.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_fragment_spread(*node:* FragmentSpreadNode, **_args: Any*) → None

enter_inline_fragment(*node:* InlineFragmentNode, **_args: Any*) → None

enter_leave_map: dict[str, EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

get_fragment_type(*name: str*) → Optional[Union[*GraphQLObjectType*, *GraphQLInterfaceType*, *GraphQLUnionType*]]

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Argument Optionality"

class graphql.validation.**ProvidedRequiredArgumentsRule**(*context:* ValidationContext)

Bases: ProvidedRequiredArgumentsOnDirectivesRule

Provided required arguments

A field or directive is only valid if all required (non-null without a default value) field arguments have been provided.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor
Given a node kind, return the EnterLeaveVisitor for that kind.

leave_directive(*directive_node:* DirectiveNode, **_args: Any*) → None

leave_field(*field_node:* FieldNode, **_args: Any*) → Optional[*VisitorActionEnum*]

report_error(*error:* GraphQLError) → None
Report a GraphQL error.

Spec Section: "Leaf Field Selections"

class graphql.validation.**ScalarLeafsRule**(*context:* ValidationContext)
Bases: *ValidationRule*

Scalar leafs

A GraphQL document is valid only if all leaf fields (fields without sub selections) are of scalar or enum types.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_field(*node:* FieldNode, **_args: Any*) → None

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor
Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None
Report a GraphQL error.

Spec Section: "Subscriptions with Single Root Field"

class graphql.validation.**SingleFieldSubscriptionsRule**(*context:* ValidationContext)
Bases: *ValidationRule*

Subscriptions must only include a single non-introspection field.

A GraphQL subscription is valid only if it contains a single root field and that root field is not an introspection field.

See https://spec.graphql.org/draft/#sec-Single-root-field

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_operation_definition(*node:* OperationDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Argument Uniqueness"

class graphql.validation.**UniqueArgumentNamesRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

Unique argument names

A GraphQL field or directive is only valid if all supplied arguments are uniquely named.

See https://spec.graphql.org/draft/#sec-Argument-Names

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

check_arg_uniqueness(*argument_nodes: Collection[*ArgumentNode*]*) → None

context: *ASTValidationContext*

enter_directive(*node:* DirectiveNode, **_args: Any*) → None

enter_field(*node:* FieldNode, **_args: Any*) → None

enter_leave_map: dict[str, EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Directives Are Unique Per Location"

class graphql.validation.**UniqueDirectivesPerLocationRule**(*context:* graphql.validation.validation_context.ValidationContext | graphql.validation.validation_context.SDLValidationContext)

Bases: *ASTValidationRule*

Unique directive names per location

A GraphQL document is only valid if all non-repeatable directives at a given location are uniquely named.

See https://spec.graphql.org/draft/#sec-Directives-Are-Unique-Per-Location

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* graphql.validation.validation_context.ValidationContext | graphql.validation.validation_context.SDLValidationContext) → None

context: *ValidationContext* | *SDLValidationContext*

enter(*node:* Node, **_args: Any*) → None

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Fragment Name Uniqueness"

class graphql.validation.**UniqueFragmentNamesRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

Unique fragment names

A GraphQL document is only valid if all defined fragments have unique names.

See https://spec.graphql.org/draft/#sec-Fragment-Name-Uniqueness

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

enter_fragment_definition(*node:* FragmentDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: dict[str, EnterLeaveVisitor]

static enter_operation_definition(**_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Input Object Field Uniqueness"

class graphql.validation.**UniqueInputFieldNamesRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

Unique input field names

A GraphQL input object value is only valid if all supplied fields are uniquely named.

See https://spec.graphql.org/draft/#sec-Input-Object-Field-Uniqueness

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_object_field(*node:* ObjectFieldNode, **_args: Any*) → None

enter_object_value(**_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

leave_object_value(**_args: Any*) → None

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Operation Name Uniqueness"

class graphql.validation.**UniqueOperationNamesRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

Unique operation names

A GraphQL document is only valid if all defined operations have unique names.

See https://spec.graphql.org/draft/#sec-Operation-Name-Uniqueness

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

static enter_fragment_definition(**_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_operation_definition(*node:* OperationDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Variable Uniqueness"

class graphql.validation.**UniqueVariableNamesRule**(*context:* ASTValidationContext)

Bases: *ASTValidationRule*

Unique variable names

A GraphQL operation is only valid if all its variables are uniquely named.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ASTValidationContext) → None

context: *ASTValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_operation_definition(*node:* OperationDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Value Type Correctness"

class graphql.validation.**ValuesOfCorrectTypeRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Value literals of correct type

A GraphQL document is only valid if all value literals are of the type expected at their position.

See https://spec.graphql.org/draft/#sec-Values-of-Correct-Type

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_boolean_value(*node:* BooleanValueNode, **_args: Any*) → None

enter_enum_value(*node:* EnumValueNode, **_args: Any*) → None

enter_float_value(*node:* FloatValueNode, **_args: Any*) → None

enter_int_value(*node:* IntValueNode, **_args: Any*) → None

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_list_value(*node:* ListValueNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_null_value(*node:* NullValueNode, **_args: Any*) → None

enter_object_field(*node:* ObjectFieldNode, **_args: Any*) → None

enter_object_value(*node:* ObjectValueNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_string_value(*node:* StringValueNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

is_valid_value_node(*node:* ValueNode) → None

Check whether this is a valid value node.

Any value literal may be a valid representation of a Scalar, depending on that scalar type.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "Variables are Input Types"

class graphql.validation.**VariablesAreInputTypesRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Variables are input types

A GraphQL operation is only valid if all the variables it defines are of input types (scalar, enum, or input object).

See https://spec.graphql.org/draft/#sec-Variables-Are-Input-Types

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_variable_definition(*node:* VariableDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

Spec Section: "All Variable Usages Are Allowed"

class graphql.validation.**VariablesInAllowedPositionRule**(*context:* ValidationContext)

Bases: *ValidationRule*

Variables in allowed position

Variable usages must be compatible with the arguments they are passed to.

See https://spec.graphql.org/draft/#sec-All-Variable-Usages-are-Allowed

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* ValidationContext) → None

context: *ValidationContext*

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_operation_definition(**_args: Any*) → None

enter_variable_definition(*node:* VariableDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

leave_operation_definition(*operation:* OperationDefinitionNode, **_args: Any*) → None

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

SDL-specific validation rules

class graphql.validation.**LoneSchemaDefinitionRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Lone Schema definition

A GraphQL document is only valid if it contains only one schema definition.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

context: *SDLValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_schema_definition(*node:* SchemaDefinitionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**UniqueOperationTypesRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Unique operation types

A GraphQL document is only valid if it has only one type per operation.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

check_operation_types(*node:* graphql.language.ast.SchemaDefinitionNode | graphql.language.ast.SchemaExtensionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

context: *SDLValidationContext*

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_schema_definition(*node:* graphql.language.ast.SchemaDefinitionNode | graphql.language.ast.SchemaExtensionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_schema_extension(*node:* graphql.language.ast.SchemaDefinitionNode | graphql.language.ast.SchemaExtensionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**UniqueTypeNamesRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Unique type names

A GraphQL document is only valid if all defined types have unique names.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

check_type_name(*node:* TypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

context: *SDLValidationContext*

enter_enum_type_definition(*node:* TypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_input_object_type_definition(*node:* TypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_interface_type_definition(*node:* TypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_object_type_definition(*node:* TypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_scalar_type_definition(*node:* TypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_union_type_definition(*node:* TypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**UniqueEnumValueNamesRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Unique enum value names

A GraphQL enum type is only valid if all its values are uniquely named.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

check_value_uniqueness(*node:* EnumTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

context: *SDLValidationContext*

enter_enum_type_definition(*node:* EnumTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_enum_type_extension(*node:* EnumTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: **dict[str, EnterLeaveVisitor]**

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**UniqueFieldDefinitionNamesRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Unique field definition names

A GraphQL complex type is only valid if all its fields are uniquely named.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

check_field_uniqueness(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

context: *SDLValidationContext*

enter_input_object_type_definition(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_input_object_type_extension(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_interface_type_definition(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_interface_type_extension(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: **dict[str, EnterLeaveVisitor]**

enter_object_type_definition(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_object_type_extension(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**UniqueArgumentDefinitionNamesRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Unique argument definition names

A GraphQL Object or Interface type is only valid if all its fields have uniquely named arguments. A GraphQL Directive is only valid if all its arguments are uniquely named.

See https://spec.graphql.org/draft/#sec-Argument-Uniqueness

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

check_arg_uniqueness(*parent_name: str*, *argument_nodes: Collection[*InputValueDefinitionNode*]*) → Optional[*VisitorActionEnum*]

check_arg_uniqueness_per_field(*name:* NameNode, *fields: Collection[*FieldDefinitionNode*]*) → Optional[*VisitorActionEnum*]

context: *SDLValidationContext*

enter_directive_definition(*node:* DirectiveDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_interface_type_definition(*node:* InterfaceTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_interface_type_extension(*node:* InterfaceTypeExtensionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_object_type_definition(*node:* ObjectTypeDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_object_type_extension(*node:* ObjectTypeExtensionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**UniqueDirectiveNamesRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Unique directive names

A GraphQL document is only valid if all defined directives have unique names.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

context: *SDLValidationContext*

enter_directive_definition(*node:* DirectiveDefinitionNode, **_args: Any*) → Optional[*VisitorActionEnum*]

enter_leave_map: dict[str, EnterLeaveVisitor]

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

class graphql.validation.**PossibleTypeExtensionsRule**(*context:* SDLValidationContext)

Bases: *SDLValidationRule*

Possible type extension

A type extension is only valid if the type is defined and has the same kind.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(*context:* SDLValidationContext) → None

check_extension(*node:* TypeExtensionNode, **_args: Any*) → None

context: *SDLValidationContext*

enter_enum_type_extension(*node:* TypeExtensionNode, **_args: Any*) → None

enter_input_object_type_extension(*node:* TypeExtensionNode, **_args: Any*) → None

enter_interface_type_extension(*node:* TypeExtensionNode, **_args: Any*) → None

enter_leave_map: dict[str, EnterLeaveVisitor]

enter_object_type_extension(*node:* TypeExtensionNode, **_args: Any*) → None

enter_scalar_type_extension(*node:* TypeExtensionNode, **_args: Any*) → None

enter_union_type_extension(*node:* TypeExtensionNode, **_args: Any*) → None

get_enter_leave_for_kind(*kind: str*) → EnterLeaveVisitor

Given a node kind, return the EnterLeaveVisitor for that kind.

report_error(*error:* GraphQLError) → None

Report a GraphQL error.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

T

U

V

W

Y