

---

# GraphQL-core 3 Documentation

*Release 3.1.7*

Christoph Zwerschke

Sep 24, 2022



# CONTENTS

<b>1</b>	<b>Contents</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Usage . . . . .	2
1.3	Differences from GraphQL.js . . . . .	17
1.4	Reference . . . . .	19
<b>2</b>	<b>Indices and tables</b>	<b>113</b>
	<b>Python Module Index</b>	<b>115</b>
	<b>Index</b>	<b>117</b>



## 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 `pipenv`, if you prefer that:

```
pipenv install graphql-core
```

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 GraphQL 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` sub-package. 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.subscription` module. 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.
- `graphql.subscription`: Subscribe to data updates.

### 1.4.1 Top-Level Functions

```
async graphql.graphql(schema: graphql.type.schema.GraphQLSchema, source: Union[str,
    graphql.language.source.Source], root_value: Any = None, context_value: Any = None,
    variable_values: Optional[Dict[str, Any]] = None, operation_name: Optional[str] =
    None, field_resolver: Optional[Callable[[...], Any]] = None, type_resolver:
    Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo,
    GraphQLAbstractType],
    Optional[Union[Awaitable[Optional[Union[GraphQLObjectType, str]]],
    GraphQLObjectType, str]]] = None, middleware: Optional[Union[Tuple, List,
    graphql.execution.middleware.MiddlewareManager]] = None, execution_context_class:
    Optional[Type[graphql.execution.execute.ExecutionContext]] = None, is_awaitable:
    Optional[Callable[[Any, bool]] = None) →
    graphql.execution.execute.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.

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: graphql.type.schema.GraphQLSchema, source: Union[str,
    graphql.language.source.Source], root_value: Any = None, context_value: Any = None,
    variable_values: Optional[Dict[str, Any]] = None, operation_name: Optional[str] =
    None, field_resolver: Optional[Callable[[...], Any]] = None, type_resolver:
    Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo,
    GraphQLAbstractType],
    Optional[Union[Awaitable[Optional[Union[GraphQLObjectType, str]]],
    GraphQLObjectType, str]]]] = None, middleware: Optional[Union[Tuple, List,
    graphql.execution.middleware.MiddlewareManager]] = None, execution_context_class:
    Optional[Type[graphql.execution.execute.ExecutionContext]] = None, check_sync: bool
    = False) → graphql.execution.execute.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.

```

exception graphql.error.GraphQLError(message: str, nodes: Optional[Union[Collection[Node], Node]] =
    None, source: Optional[Source] = None, positions:
    Optional[Collection[int]] = None, path:
    Optional[Collection[Union[str, int]]] = None, original_error:
    Optional[Exception] = None, extensions: Optional[Dict[str, Any]]
    = 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.

**args**

**extensions:** `Optional[Dict[str, Any]]`

Extension fields to add to the formatted error

**property formatted:** `Dict[str, Any]`

Get error formatted according to the specification.

**locations:** `Optional[List[SourceLocation]]`

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:** `Optional[List[Node]]`

A list of GraphQL AST Nodes corresponding to this error

**original\_error:** `Optional[Exception]`

The original error thrown from a field resolver during execution

**path:** `Optional[List[Union[str, int]]]`

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:** `Optional[Collection[int]]`

Error positions

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

**source:** `Optional[Source]`

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`.

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

Bases: `graphql.error.graphql_error.GraphQLError`

A GraphQLError representing a syntax error.

**args**

**extensions:** `Optional[Dict[str, Any]]`

Extension fields to add to the formatted error

**property formatted:** `Dict[str, Any]`

Get error formatted according to the specification.

**locations:** `Optional[List[SourceLocation]]`

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:** `Optional[List[Node]]`

A list of GraphQL AST Nodes corresponding to this error

**original\_error:** `Optional[Exception]`

The original error thrown from a field resolver during execution

**path:** `Optional[List[Union[str, int]]]`

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:** `Optional[Collection[int]]`

Error positions

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

**source:** `Optional[Source]`

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`.

`graphql.error.format_error(error: graphql.error.graphql_error.GraphQLError) → Dict[str, Any]`

Format a GraphQL error.

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

`graphql.error.located_error(original_error: Exception, nodes: Optional[Union[None, Collection[Node]]], path: Optional[Collection[Union[str, int]]] = None) → graphql.error.graphql_error.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`.

`graphql.error.print_error(error: graphql.error.graphql_error.GraphQLError) → str`

Print a `GraphQLError` to a string.

Represents useful location information about the error’s position in the source.

## Execution

### GraphQL Execution

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

**class** `graphql.execution.ExecutionContext` (*schema: graphql.type.schema.GraphQLSchema, fragments: Dict[str, graphql.language.ast.FragmentDefinitionNode], root\_value: Any, context\_value: Any, operation: graphql.language.ast.OperationDefinitionNode, variable\_values: Dict[str, Any], field\_resolver: Callable[[Any, Any], type\_resolver: Callable[[Any, graphql.type.definition.GraphQLResolveInfo, Union[graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType]], Optional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLOBJECT str]]], graphql.type.definition.GraphQLOBJECT str]]], errors: List[graphql.error.graphql\_error.GraphQLError], middleware\_manager: Optional[graphql.execution.middleware.MiddlewareManager], is\_awaitable: Optional[Callable[[Any], bool]]]*)

Bases: object

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: graphql.type.schema.GraphQLSchema, fragments: Dict[str,
    graphql.language.ast.FragmentDefinitionNode], root_value: Any, context_value: Any, operation:
    graphql.language.ast.OperationDefinitionNode, variable_values: Dict[str, Any], field_resolver:
    Callable[[...], Any], type_resolver: Callable[[Any, graphql.type.definition.GraphQLResolveInfo,
    Union[graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType]],
    Optional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType, str]]],
    graphql.type.definition.GraphQLObjectType, str]]], errors:
    List[graphql.error.graphql_error.GraphQLError], middleware_manager:
    Optional[graphql.execution.middleware.MiddlewareManager], is_awaitable:
    Optional[Callable[[Any], bool]]) → None

classmethod build(schema: graphql.type.schema.GraphQLSchema, document:
    graphql.language.ast.DocumentNode, root_value: Optional[Any] = None,
    context_value: Optional[Any] = None, raw_variable_values: Optional[Dict[str, Any]]
    = None, operation_name: Optional[str] = None, field_resolver:
    Optional[Callable[[...], Any]] = None, type_resolver: Optional[Callable[[Any,
    graphql.type.definition.GraphQLResolveInfo,
    Union[graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType]], Op-
    tional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType,
    str]]], graphql.type.definition.GraphQLObjectType, str]]]] = None, middleware:
    Optional[Union[Tuple, List, graphql.execution.middleware.MiddlewareManager]] =
    None, is_awaitable: Optional[Callable[[Any], bool]] = None) →
    Union[List[graphql.error.graphql_error.GraphQLError],
    graphql.execution.execute.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_resolve_info(field_def: graphql.type.definition.GraphQLField, field_nodes:
    List[graphql.language.ast.FieldNode], parent_type:
    graphql.type.definition.GraphQLObjectType, path: graphql.pyutils.path.Path) →
    graphql.type.definition.GraphQLResolveInfo
```

Build the GraphQLResolveInfo object.

For internal use only.

```
build_response(data: Optional[Union[Awaitable[Optional[Dict[str, Any]]], Dict[str, Any]]]) →
    Union[Awaitable[graphql.execution.execute.ExecutionResult],
    graphql.execution.execute.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*: graphql.type.definition.GraphQLObjectType, *field\_nodes*: List[graphql.language.ast.FieldNode], *path*: graphql.pyutils.path.Path, *result*: Any) → Union[Awaitable[Dict[str, Any]], Dict[str, Any]]

Collect sub-fields to execute to complete this value.

**collect\_fields**(*runtime\_type*: graphql.type.definition.GraphQLObjectType, *selection\_set*: graphql.language.ast.SelectionSetNode, *fields*: Dict[str, List[graphql.language.ast.FieldNode]], *visited\_fragment\_names*: Set[str]) → Dict[str, List[graphql.language.ast.FieldNode]]

Collect fields.

Given a selection\_set, adds all of the fields in that selection to the passed in map of fields, and returns it at the end.

collect\_fields requires the “runtime type” of an object. For a field which returns an Interface or Union type, the “runtime type” will be the actual Object type returned by that field.

For internal use only.

**collect\_subfields**(*return\_type*: graphql.type.definition.GraphQLObjectType, *field\_nodes*: List[graphql.language.ast.FieldNode]) → Dict[str, List[graphql.language.ast.FieldNode]]

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*: Union[graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType], *field\_nodes*: List[graphql.language.ast.FieldNode], *info*: graphql.type.definition.GraphQLResolveInfo, *path*: graphql.pyutils.path.Path, *result*: Any) → Union[Awaitable[Any], 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.

**static complete\_leaf\_value**(*return\_type*: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.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\_value**(*return\_type*: graphql.type.definition.GraphQLList[Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLWrappingType]], *field\_nodes*: List[graphql.language.ast.FieldNode], *info*: graphql.type.definition.GraphQLResolveInfo, *path*: graphql.pyutils.path.Path, *result*: Iterable[Any]) → Union[Awaitable[List[Any]], 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*: graphql.type.definition.GraphQLObjectType, *field\_nodes*: List[graphql.language.ast.FieldNode], *info*: graphql.type.definition.GraphQLResolveInfo, *path*: graphql.pyutils.path.Path, *result*: Any) → Union[Awaitable[Dict[str, Any]], Dict[str, Any]]

Complete an Object value by executing all sub-selections.

**complete\_value**(*return\_type*: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLWrappingType], *field\_nodes*: List[graphql.language.ast.FieldNode], *info*: graphql.type.definition.GraphQLResolveInfo, *path*: graphql.pyutils.path.Path, *result*: Any) → Union[Awaitable[Any], Any]

Complete a value.

Implements the instructions for completeValue as defined in the “Field entries” 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

**does\_fragment\_condition\_match**(*fragment*: Union[graphql.language.ast.FragmentDefinitionNode, graphql.language.ast.InlineFragmentNode], *type\_*: graphql.type.definition.GraphQLObjectType) → bool

Determine if a fragment is applicable to the given type.

**ensure\_valid\_runtime\_type**(*runtime\_type\_or\_name*: Any, *return\_type*: Union[graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType], *field\_nodes*: List[graphql.language.ast.FieldNode], *info*: graphql.type.definition.GraphQLResolveInfo, *result*: Any) → graphql.type.definition.GraphQLObjectType

**errors:** List[[graphql.error.graphql\\_error.GraphQLError](#)]

**execute\_fields**(*parent\_type*: graphql.type.definition.GraphQLObjectType, *source\_value*: Any, *path*: Optional[graphql.pyutils.path.Path], *fields*: Dict[str, List[graphql.language.ast.FieldNode]]) → Union[Awaitable[Dict[str, Any]], Dict[str, Any]]

Execute the given fields concurrently.

Implements the “Evaluating selection sets” section of the spec for “read” mode.

**execute\_fields\_serially**(*parent\_type*: graphql.type.definition.GraphQLObjectType, *source\_value*: Any, *path*: Optional[graphql.pyutils.path.Path], *fields*: Dict[str, List[graphql.language.ast.FieldNode]]) → Union[Awaitable[Dict[str, Any]], Dict[str, Any]]

Execute the given fields serially.

Implements the “Evaluating selection sets” section of the spec for “write” mode.

**execute\_operation**(*operation*: `graphql.language.ast.OperationDefinitionNode`, *root\_value*: *Any*) → *Optional*[*Union*[*Awaitable*[*Any*], *Any*]]

Execute an operation.

Implements the “Evaluating operations” section of the spec.

**field\_resolver**: *Callable*[[...], *Any*]

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

**handle\_field\_error**(*error*: `graphql.error.graphql_error.GraphQLError`, *return\_type*: *Union*[`graphql.type.definition.GraphQLScalarType`, `graphql.type.definition.GraphQLObjectType`, `graphql.type.definition.GraphQLInterfaceType`, `graphql.type.definition.GraphQLUnionType`, `graphql.type.definition.GraphQLEnumType`, `graphql.type.definition.GraphQLWrappingType`]) → *None*

**static is\_awaitable**(*value*: *Any*) → *bool*

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

Instead of testing if the object is an instance of `abc.Awaitable`, it checks the existence of an `__await__` attribute. This is much faster.

**middleware\_manager**: *Optional*[`graphql.execution.middleware.MiddlewareManager`]

**operation**: `graphql.language.ast.OperationDefinitionNode`

**resolve\_field**(*parent\_type*: `graphql.type.definition.GraphQLObjectType`, *source*: *Any*, *field\_nodes*: *List*[`graphql.language.ast.FieldNode`], *path*: `graphql.pyutils.path.Path`) → *Union*[*Awaitable*[*Any*], *Any*]

Resolve the field on the given source object.

In particular, this 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.

**root\_value**: *Any*

**schema**: `graphql.type.schema.GraphQLSchema`

**should\_include\_node**(*node*: *Union*[`graphql.language.ast.FragmentSpreadNode`, `graphql.language.ast.FieldNode`, `graphql.language.ast.InlineFragmentNode`]) → *bool*

Check if node should be included

Determines if a field should be included based on the `@include` and `@skip` directives, where `@skip` has higher precedence than `@include`.

**type\_resolver**: *Callable*[[*Any*, `graphql.type.definition.GraphQLResolveInfo`, *Union*[`graphql.type.definition.GraphQLInterfaceType`, `graphql.type.definition.GraphQLUnionType`]], *Optional*[*Union*[*Awaitable*[*Optional*[*Union*[`graphql.type.definition.GraphQLObjectType`, *str*]]], `graphql.type.definition.GraphQLObjectType`, *str*]]]



**variable\_values:** Dict[str, Any]

**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)

**data:** Optional[Dict[str, Any]]

**errors:** Optional[List[graphql.error.graphql\_error.GraphQLError]]

**extensions:** Optional[Dict[str, Any]]

**property formatted:** Dict[str, Any]

Get execution result formatted according to the specification.

**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)

**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.**default\_field\_resolver**(source: Any, info: graphql.type.definition.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: graphql.type.definition.GraphQLResolveInfo,
                                       abstract_type:
                                         Union[graphql.type.definition.GraphQLInterfaceType,
                                                graphql.type.definition.GraphQLUnionType]) → Op-
                                         tional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType,
                                                                                   str]]], graphql.type.definition.GraphQLObjectType, str]]
```

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.

```
graphql.execution.execute(schema: graphql.type.schema.GraphQLSchema, document:
  graphql.language.ast.DocumentNode, root_value: Optional[Any] = None,
  context_value: Optional[Any] = None, variable_values: Optional[Dict[str, Any]]
  = None, operation_name: Optional[str] = None, field_resolver:
  Optional[Callable[[...], Any]] = None, type_resolver: Optional[Callable[[Any,
  graphql.type.definition.GraphQLResolveInfo,
  Union[graphql.type.definition.GraphQLInterfaceType,
  graphql.type.definition.GraphQLUnionType]], Op-
  tional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType,
  str]]], graphql.type.definition.GraphQLObjectType, str]]]] = None, middleware:
  Optional[Union[Tuple, List, graphql.execution.middleware.MiddlewareManager]]
  = None, execution_context_class:
  Optional[Type[graphql.execution.execute.ExecutionContext]] = None,
  is_awaitable: Optional[Callable[[Any], bool]] = None) →
  Union[Awaitable[graphql.execution.execute.ExecutionResult],
  graphql.execution.execute.ExecutionResult]
```

Execute a GraphQL operation.

Implements the “Evaluating 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.

```
graphql.execution.execute_sync(schema: graphql.type.schema.GraphQLSchema, document:
  graphql.language.ast.DocumentNode, root_value: Optional[Any] = None,
  context_value: Optional[Any] = None, variable_values: Optional[Dict[str,
  Any]] = None, operation_name: Optional[str] = None, field_resolver:
  Optional[Callable[[...], Any]] = None, type_resolver:
  Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo,
  Union[graphql.type.definition.GraphQLInterfaceType,
  graphql.type.definition.GraphQLUnionType]], Op-
  tional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType,
  str]]], graphql.type.definition.GraphQLObjectType, str]]]] = None,
  middleware: Optional[Union[Tuple, List,
  graphql.execution.middleware.MiddlewareManager]] = None,
  execution_context_class:
  Optional[Type[graphql.execution.execute.ExecutionContext]] = None,
  check_sync: bool = False) → graphql.execution.execute.ExecutionResult
```

Execute a GraphQL operation synchronously.

Also implements the “Evaluating 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.get_directive_values(directive_def: graphql.type.directives.GraphQLDirective, node:
    Union[graphql.language.ast.EnumValueDefinitionNode,
    graphql.language.ast.ExecutableDefinitionNode,
    graphql.language.ast.FieldDefinitionNode,
    graphql.language.ast.InputValueDefinitionNode,
    graphql.language.ast.SelectionNode,
    graphql.language.ast.SchemaDefinitionNode,
    graphql.language.ast.TypeDefinitionNode,
    graphql.language.ast.TypeExtensionNode], variable_values:
    Optional[Dict[str, Any]] = None) → Optional[Dict[str, Any]]
```

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`.

## Language

GraphQL Language

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

## AST

```
class graphql.language.Location(start_token: graphql.language.ast.Token, end_token:
    graphql.language.ast.Token, source: graphql.language.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: graphql.language.ast.Token, end_token: graphql.language.ast.Token, source:
    graphql.language.source.Source) → None
```

**end:** int

**end\_token:** `graphql.language.ast.Token`

**source:** `graphql.language.source.Source`

**start:** int

**start\_token:** `graphql.language.ast.Token`

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

Bases: object

AST nodes

```

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

keys = ['loc']

kind: str = 'ast'

loc: Optional[graphql.language.ast.Location]

```

Each kind of AST node has its own class:

```

class graphql.language.ArgumentNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc', 'name', 'value']
    kind: str = 'argument'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode
    value: graphql.language.ast.ValueNode

class graphql.language.BooleanValueNode(**kwargs: Any)
    Bases: graphql.language.ast.ValueNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc', 'value']
    kind: str = 'boolean_value'
    loc: Optional[graphql.language.ast.Location]
    value: bool

class graphql.language.DefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc']
    kind: str = 'definition'
    loc: Optional[graphql.language.ast.Location]

class graphql.language.DirectiveDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeSystemDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

```

```
arguments: graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.InputValueDefinitionNode]

description: Optional[graphql.language.ast.StringValueNode]

keys = ['loc', 'description', 'name', 'arguments', 'repeatable', 'locations']

kind: str = 'directive_definition'

loc: Optional[graphql.language.ast.Location]

locations: graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.NameNode]

name: graphql.language.ast.NameNode

repeatable: bool
```

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

```
    Bases: graphql.language.ast.Node
```

```
    __init__(**kwargs: Any) → None
```

```
        Initialize the node with the given keyword arguments.
```

```
arguments:
```

```
graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.ArgumentNode]
```

```
keys = ['loc', 'name', 'arguments']
```

```
kind: str = 'directive'
```

```
loc: Optional[graphql.language.ast.Location]
```

```
name: graphql.language.ast.NameNode
```

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

```
    Bases: graphql.language.ast.Node
```

```
    __init__(**kwargs: Any) → None
```

```
        Initialize the node with the given keyword arguments.
```

```
definitions:
```

```
graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DefinitionNode]
```

```
keys = ['loc', 'definitions']
```

```
kind: str = 'document'
```

```
loc: Optional[graphql.language.ast.Location]
```

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

```
    Bases: graphql.language.ast.TypeDefinitionNode
```

```
    __init__(**kwargs: Any) → None
```

```
        Initialize the node with the given keyword arguments.
```

```
description: Optional[graphql.language.ast.StringValueNode]
```

```
directives:
```

```
graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]
```

```

    keys = ['loc', 'description', 'name', 'directives', 'values']

    kind: str = 'enum_type_definition'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

    values:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.EnumValueDefinitionNode]

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

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

    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]

    keys = ['loc', 'name', 'directives', 'values']

    kind: str = 'enum_type_extension'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

    values:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.EnumValueDefinitionNode]

class graphql.language.EnumValueDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.DefinitionNode

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

    description: Optional[graphql.language.ast.StringValueNode]

    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]

    keys = ['loc', 'description', 'name', 'directives']

    kind: str = 'enum_value_definition'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

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

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

    keys = ['loc', 'value']

    kind: str = 'enum_value'

```

```
loc: Optional[graphql.language.ast.Location]

value: str

class graphql.language.ExecutableDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.DefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'name', 'directives', 'variable_definitions', 'selection_set']
    kind: str = 'executable_definition'
    loc: Optional[graphql.language.ast.Location]
    name: Optional[graphql.language.ast.NameNode]
    selection_set: graphql.language.ast.SelectionSetNode
    variable_definitions:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.VariableDefinitionNode]

class graphql.language.FieldDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.DefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    arguments: graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.InputValueDefinitionNode]
    description: Optional[graphql.language.ast.StringValueNode]
    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'description', 'name', 'directives', 'arguments', 'type']
    kind: str = 'field_definition'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode
    type: graphql.language.ast.TypeNode

class graphql.language.FieldNode(**kwargs: Any)
    Bases: graphql.language.ast.SelectionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    alias: Optional[graphql.language.ast.NameNode]
```

```

    arguments:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.ArgumentNode]

    directives: graphql.pyutils.frozen_list.FrozenList[DirectiveNode]

    keys = ['loc', 'directives', 'alias', 'name', 'arguments', 'selection_set']

    kind: str = 'field'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

    selection_set: Optional[graphql.language.ast.SelectionSetNode]

class graphql.language.FloatValueNode(**kwargs: Any)
    Bases: graphql.language.ast.ValueNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    keys = ['loc', 'value']

    kind: str = 'float_value'

    loc: Optional[graphql.language.ast.Location]

    value: str

class graphql.language.FragmentDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.ExecutableDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    directives: graphql.pyutils.frozen_list.FrozenList[DirectiveNode]

    keys = ['loc', 'name', 'directives', 'variable_definitions', 'selection_set',
            'type_condition']

    kind: str = 'fragment_definition'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

    selection_set: SelectionSetNode

    type_condition: graphql.language.ast.NamedTypeNode

    variable_definitions:
        graphql.pyutils.frozen_list.FrozenList[VariableDefinitionNode]

class graphql.language.FragmentSpreadNode(**kwargs: Any)
    Bases: graphql.language.ast.SelectionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

```

```
directives: graphql.pyutils.frozen_list.FrozenList[DirectiveNode]
keys = ['loc', 'directives', 'name']
kind: str = 'fragment_spread'
loc: Optional[graphql.language.ast.Location]
name: graphql.language.ast.NameNode

class graphql.language.InlineFragmentNode(**kwargs: Any)
    Bases: graphql.language.ast.SelectionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    directives: graphql.pyutils.frozen_list.FrozenList[DirectiveNode]
    keys = ['loc', 'directives', 'type_condition', 'selection_set']
    kind: str = 'inline_fragment'
    loc: Optional[graphql.language.ast.Location]
    selection_set: graphql.language.ast.SelectionSetNode
    type_condition: graphql.language.ast.NamedTypeNode

class graphql.language.InputObjectTypeDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    description: Optional[graphql.language.ast.StringValueNode]
    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]
    fields: graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.
        InputValueDefinitionNode]
    keys = ['loc', 'description', 'name', 'directives', 'fields']
    kind: str = 'input_object_type_definition'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode

class graphql.language.InputObjectTypeExtensionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeExtensionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]
```



```

    fields: graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.InputValueDefinitionNode]

    keys = ['loc', 'name', 'directives', 'fields']

    kind: str = 'input_object_type_extension'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

class graphql.language.InputValueDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.DefinitionNode

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

    default_value: Optional[graphql.language.ast.ValueNode]

    description: Optional[graphql.language.ast.StringValueNode]

    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]

    keys = ['loc', 'description', 'name', 'directives', 'type', 'default_value']

    kind: str = 'input_value_definition'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

    type: graphql.language.ast.TypeNode

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

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

    keys = ['loc', 'value']

    kind: str = 'int_value'

    loc: Optional[graphql.language.ast.Location]

    value: str

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

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

    description: Optional[graphql.language.ast.StringValueNode]

    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]

```

```
fields:
    graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.FieldDefinitionNode]

interfaces:
    graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.NamedTypeNode]

keys = ['loc', 'description', 'name', 'directives', 'fields', 'interfaces']

kind: str = 'interface_type_definition'

loc: Optional[graphql.language.ast.Location]

name: graphql.language.ast.NameNode

class graphql.language.InterfaceTypeExtensionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeExtensionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]

    fields:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.FieldDefinitionNode]

    interfaces:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.NamedTypeNode]

    keys = ['loc', 'name', 'directives', 'interfaces', 'fields']

    kind: str = 'interface_type_extension'

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

class graphql.language.ListTypeNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    keys = ['loc', 'type']

    kind: str = 'list_type'

    loc: Optional[graphql.language.ast.Location]

    type: graphql.language.ast.TypeNode

class graphql.language.ListValueNode(**kwargs: Any)
    Bases: graphql.language.ast.ValueNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    keys = ['loc', 'values']
```

```

    kind: str = 'list_value'

    loc: Optional[graphql.language.ast.Location]

    values: graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.ValueNode]

class graphql.language.NameNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc', 'value']
    kind: str = 'name'
    loc: Optional[graphql.language.ast.Location]
    value: str

class graphql.language.NamedTypeNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc', 'name']
    kind: str = 'named_type'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode

class graphql.language.NonNullTypeNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc', 'type']
    kind: str = 'non_null_type'
    loc: Optional[graphql.language.ast.Location]
    type: Union[graphql.language.ast.NamedTypeNode, graphql.language.ast.ListTypeNode]

class graphql.language.NullValueNode(**kwargs: Any)
    Bases: graphql.language.ast.ValueNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc']
    kind: str = 'null_value'
    loc: Optional[graphql.language.ast.Location]

```

```
class graphql.language.ObjectFieldNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc', 'name', 'value']
    kind: str = 'object_field'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode
    value: graphql.language.ast.ValueNode

class graphql.language.ObjectTypeDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    description: Optional[graphql.language.ast.StringValueNode]
    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]
    fields:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.FieldDefinitionNode]
    interfaces:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.NamedTypeNode]
    keys = ['loc', 'description', 'name', 'directives', 'interfaces', 'fields']
    kind: str = 'object_type_definition'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode

class graphql.language.ObjectTypeExtensionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeExtensionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]
    fields:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.FieldDefinitionNode]
    interfaces:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.NamedTypeNode]
    keys = ['loc', 'name', 'directives', 'interfaces', 'fields']
    kind: str = 'object_type_extension'
```

```

    loc: Optional[graphql.language.ast.Location]

    name: graphql.language.ast.NameNode

class graphql.language.ObjectValueNode(**kwargs: Any)
    Bases: graphql.language.ast.ValueNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    fields:
        graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.ObjectFieldNode]

    keys = ['loc', 'fields']

    kind: str = 'object_value'

    loc: Optional[graphql.language.ast.Location]

class graphql.language.OperationDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.ExecutableDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    directives: graphql.pyutils.frozen\_list.FrozenList[DirectiveNode]

    keys = ['loc', 'name', 'directives', 'variable_definitions', 'selection_set',
            'operation']

    kind: str = 'operation_definition'

    loc: Optional[graphql.language.ast.Location]

    name: Optional[graphql.language.ast.NameNode]

    operation: graphql.language.ast.OperationType

    selection_set: SelectionSetNode

    variable_definitions:
        graphql.pyutils.frozen\_list.FrozenList[VariableDefinitionNode]

class graphql.language.OperationType(value)
    Bases: enum.Enum
    An enumeration.

    MUTATION = 'mutation'

    QUERY = 'query'

    SUBSCRIPTION = 'subscription'

class graphql.language.OperationTypeDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

```

```
keys = ['loc', 'operation', 'type']

kind: str = 'operation_type_definition'

loc: Optional[graphql.language.ast.Location]

operation: graphql.language.ast.OperationType

type: graphql.language.ast.NamedTypeNode

class graphql.language.ScalarTypeDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    description: Optional[graphql.language.ast.StringValueNode]
    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'description', 'name', 'directives']
    kind: str = 'scalar_type_definition'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode

class graphql.language.ScalarTypeExtensionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeExtensionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'name', 'directives']
    kind: str = 'scalar_type_extension'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode

class graphql.language.SchemaDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeSystemDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    description: Optional[graphql.language.ast.StringValueNode]
    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'description', 'directives', 'operation_types']
```

```

    kind: str = 'schema_definition'

    loc: Optional[graphql.language.ast.Location]

    operation_types: graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.
    OperationTypeDefinitionNode]

class graphql.language.SchemaExtensionNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    directives:
    graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]

    keys = ['loc', 'directives', 'operation_types']

    kind: str = 'schema_extension'

    loc: Optional[graphql.language.ast.Location]

    operation_types: graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.
    OperationTypeDefinitionNode]

class graphql.language.SelectionNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    directives:
    graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]

    keys = ['loc', 'directives']

    kind: str = 'selection'

    loc: Optional[graphql.language.ast.Location]

class graphql.language.SelectionSetNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.

    keys = ['loc', 'selections']

    kind: str = 'selection_set'

    loc: Optional[graphql.language.ast.Location]

    selections:
    graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.SelectionNode]

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

```

```
__init__(**kwargs: Any) → None
    Initialize the node with the given keyword arguments.
block: Optional[bool]
keys = ['loc', 'value', 'block']
kind: str = 'string_value'
loc: Optional[graphql.language.ast.Location]
value: str

class graphql.language.TypeDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeSystemDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    description: Optional[graphql.language.ast.StringValueNode]
    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'description', 'name', 'directives']
    kind: str = 'type_definition'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode

class graphql.language.TypeExtensionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeSystemDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    directives:
graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'name', 'directives']
    kind: str = 'type_extension'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode

class graphql.language.TypeNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc']
    kind: str = 'type'
```



```

    loc: Optional[graphql.language.ast.Location]

class graphql.language.TypeSystemDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.DefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc']
    kind: str = 'type_system_definition'
    loc: Optional[graphql.language.ast.Location]

graphql.language.TypeSystemExtensionNode
    alias of Union[graphql.language.ast.SchemaExtensionNode, graphql.language.ast.TypeExtensionNode]

class graphql.language.UnionTypeDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeDefinitionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    description: Optional[graphql.language.ast.StringValueNode]
    directives:
        graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'description', 'name', 'directives', 'types']
    kind: str = 'union_type_definition'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode
    types: graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.NamedTypeNode]

class graphql.language.UnionTypeExtensionNode(**kwargs: Any)
    Bases: graphql.language.ast.TypeExtensionNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    directives:
        graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'name', 'directives', 'types']
    kind: str = 'union_type_extension'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode
    types: graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.NamedTypeNode]

```

```
class graphql.language.ValueNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc']
    kind: str = 'value'
    loc: Optional[graphql.language.ast.Location]

class graphql.language.VariableDefinitionNode(**kwargs: Any)
    Bases: graphql.language.ast.Node
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    default_value: Optional[graphql.language.ast.ValueNode]
    directives:
        graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.DirectiveNode]
    keys = ['loc', 'variable', 'type', 'default_value', 'directives']
    kind: str = 'variable_definition'
    loc: Optional[graphql.language.ast.Location]
    type: graphql.language.ast.TypeNode
    variable: graphql.language.ast.VariableNode

class graphql.language.VariableNode(**kwargs: Any)
    Bases: graphql.language.ast.ValueNode
    __init__(**kwargs: Any) → None
        Initialize the node with the given keyword arguments.
    keys = ['loc', 'name']
    kind: str = 'variable'
    loc: Optional[graphql.language.ast.Location]
    name: graphql.language.ast.NameNode
```

Directive locations are specified using the following enumeration:

```
class graphql.language.DirectiveLocation(value)
    Bases: enum.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: graphql.language.ast.Node) → bool`

Check whether the given node represents a definition.

`graphql.language.is_executable_definition_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents an executable definition.

`graphql.language.is_selection_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents a selection.

`graphql.language.is_value_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents a value.

`graphql.language.is_type_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents a type.

`graphql.language.is_type_system_definition_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents a type system definition.

`graphql.language.is_type_definition_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents a type definition.

`graphql.language.is_type_system_extension_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents a type system extension.

`graphql.language.is_type_extension_node(node: graphql.language.ast.Node) → bool`

Check whether the given node represents a type extension.

## Lexer

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

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

**advance**() → *graphql.language.ast.Token*

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

**lookahead**() → *graphql.language.ast.Token*

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

**read\_block\_string**(*start*: int, *line*: int, *col*: int, *prev*: Optional[graphql.language.ast.Token]) → *graphql.language.ast.Token*

**read\_comment**(*start*: int, *line*: int, *col*: int, *prev*: Optional[graphql.language.ast.Token]) → *graphql.language.ast.Token*

Read a comment token from the source file.

**read\_digits**(*start*: int, *char*: str) → int

Return the new position in the source after reading digits.

**read\_name**(*start*: int, *line*: int, *col*: int, *prev*: Optional[graphql.language.ast.Token]) → *graphql.language.ast.Token*

Read an alphanumeric + underscore name from the source.

**read\_number**(*start*: int, *char*: str, *line*: int, *col*: int, *prev*: Optional[graphql.language.ast.Token]) → *graphql.language.ast.Token*

Reads a number token from the source file.

Either a float or an int depending on whether a decimal point appears.

**read\_string**(*start*: int, *line*: int, *col*: int, *prev*: Optional[graphql.language.ast.Token]) → *graphql.language.ast.Token*

Read a string token from the source file.

**read\_token**(*prev*: graphql.language.ast.Token) → *graphql.language.ast.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.

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

Bases: enum.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 = '|'
SOF = '<SOF>'
SPREAD = '...'
STRING = 'String'

```

```

class graphql.language.Token(kind: graphql.language.token_kind.TokenKind, start: int, end: int, line: int,
                             column: int, prev: Optional[graphql.language.ast.Token] = None, value:
                             Optional[str] = None)

```

Bases: object

AST Token

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

```

__init__(kind: graphql.language.token_kind.TokenKind, start: int, end: int, line: int, column: int, prev:
          Optional[graphql.language.ast.Token] = None, value: Optional[str] = None) → None

```

**column:** int

**property desc:** str

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

**end:** int

**kind:** [graphql.language.token\\_kind.TokenKind](#)

```
line: int
next: Optional[graphql.language.ast.Token]
prev: Optional[graphql.language.ast.Token]
start: int
value: Optional[str]
```

## Location

`graphql.language.get_location(source: Source, position: int) → graphql.language.location.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: tuple

Represents a location in a Source.

`__init__()`

**property** `column`

Alias for field number 1

`count(value, /)`

Return number of occurrences of value.

**property** `formatted: Dict[str, int]`

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

Return first index of value.

Raises ValueError if the value is not present.

**property** `line`

Alias for field number 0

`graphql.language.print_location(location: graphql.language.ast.Location) → str`

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

## Parser

`graphql.language.parse(source: Union[graphql.language.source.Source, str], no_location: bool = False, experimental_fragment_variables: bool = False) → graphql.language.ast.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.

Experimental features:

If `experimental_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 {
  ...
}
```

`graphql.language.parse_type(source: Union[graphql.language.source.Source, str], no_location: bool = False, experimental_fragment_variables: bool = False) → graphql.language.ast.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[graphql.language.source.Source, str], no_location: bool = False, experimental_fragment_variables: bool = False) → graphql.language.ast.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()`.

## Printer

`graphql.language.print_ast(ast: graphql.language.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: graphql.language.location.SourceLocation = SourceLocation(line=1, column=1))`

Bases: `object`

A representation of source input to GraphQL.

`__init__(body: str, name: str = 'GraphQL request', location_offset: graphql.language.location.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*) → *graphql.language.location.SourceLocation*

**location\_offset**

**name**

**graphql.language.print\_source\_location**(*source: graphql.language.source.Source, source\_location: graphql.language.location.SourceLocation*) → *str*

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

## Visitor

**graphql.language.visit**(*root: graphql.language.ast.Node, visitor: graphql.language.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
```

(continues on next page)



(continued from previous page)

```
# 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 defined 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.

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

**BREAK = True**

**IDLE = None**

**REMOVE = Ellipsis**

**SKIP = False**

`__init__()`

`get_visit_fn(kind: str, is_leaving: bool = False) → Callable`

Get the visit function for the given node kind and direction.

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

Bases: `graphql.language.visitor.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[graphql.language.visitor.Visitor])`

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

```
enter(node: graphql.language.ast.Node, *args: Any) →  
    Optional[graphql.language.visitor.VisitorActionEnum]  
  
get_visit_fn(kind: str, is_leaving: bool = False) → Callable  
    Get the visit function for the given node kind and direction.  
  
leave(node: graphql.language.ast.Node, *args: Any) →  
    Optional[graphql.language.visitor.VisitorActionEnum]
```

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.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: Callable) → None`

A cached property.

A property that is only computed once per instance and then replaces itself with an ordinary attribute. Deleting the attribute resets the property.

`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.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.

**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) →  
graphql.pyutils.simple_pub_sub.SimplePubSubIterator`

**subscribers:** Set[Callable]

**class** `graphql.pyutils.SimplePubSubIterator(pubsub: graphql.pyutils.simple_pub_sub.SimplePubSub,  
transform: Optional[Callable])`

Bases: AsyncIterator

`__init__(pubsub: graphql.pyutils.simple_pub_sub.SimplePubSub, transform: Optional[Callable]) → None`

`async aclose()` → None

`async empty_queue()` → None

`async push_value(event: Any) → None`

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

Return true if a value is a finite number.

`graphql.pyutils.is_integer(value: Any) → bool`

Return true if a value is an integer number.

`graphql.pyutils.AwaitableOrValue`

alias of Union[Awaitable[graphql.pyutils.awaitable\_or\_value.T], graphql.pyutils.awaitable\_or\_value.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.FrozenList(iterable=(), /)`

Bases: `List[graphql.pyutils.frozen_list.T]`

List that can only be read, but not changed.

**class** `graphql.pyutils.FrozenDict`

Bases: `Dict[graphql.pyutils.frozen_dict.KT, graphql.pyutils.frozen_dict.VT]`

Dictionary that can only be read, but not changed.

**class** `graphql.pyutils.Path(prev: Any, key: Union[str, int], typename: Optional[str])`

Bases: `tuple`

A generic path of string or integer indices

`__init__()`

`add_key(key: Union[str, int], typename: Optional[str] = None) → graphql.pyutils.path.Path`

Return a new Path containing the given key.

`as_list() → List[Union[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.

**property** `key`

current index in the path (string or integer)

**property** `prev`

path with the previous indices

**property** `typename`

name of the parent type to avoid path ambiguity

`graphql.pyutils.print_path_list(path: Collection[Union[str, int]]) → str`

Build a string describing the path.

`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.

## Subscription

GraphQL Subscription

The `graphql.subscription` package is responsible for subscribing to updates on specific data.

`async graphql.subscription.subscribe(schema: graphql.type.schema.GraphQLSchema, document: graphql.language.ast.DocumentNode, root_value: Optional[Any] = None, context_value: Optional[Any] = None, variable_values: Optional[Dict[str, Any]] = None, operation_name: Optional[str] = None, field_resolver: Optional[Callable[[...], Any]] = None, subscribe_field_resolver: Optional[Callable[[...], Any]] = None) → Union[AsyncIterator[graphql.execution.execute.ExecutionResult], graphql.execution.execute.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.

## Helpers

`async graphql.subscription.create_source_event_stream(schema: graphql.type.schema.GraphQLSchema, document: graphql.language.ast.DocumentNode, root_value: Optional[Any] = None, context_value: Optional[Any] = None, variable_values: Optional[Dict[str, Any]] = None, operation_name: Optional[str] = None, field_resolver: Optional[Callable[[...], Any]] = None) → Union[AsyncIterable[Any], graphql.execution.execute.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.

```
class graphql.subscription.MapAsyncIterator(iterable: AsyncIterable, callback: Callable,  
                                           reject_callback: Optional[Callable] = None)
```

Bases: object

Map an AsyncIterable over a callback function.

Given an AsyncIterable and a callback function, return an AsyncIterator which produces values mapped via calling the callback function.

When the resulting AsyncIterator is closed, the underlying AsyncIterable will also be closed.

```
__init__(iterable: AsyncIterable, callback: Callable, reject_callback: Optional[Callable] = None) → None
```

```
async aclose() → None
```

Close the iterator.

```
async athrow(type_: Union[BaseException, Type[BaseException]], value: Optional[BaseException] =  
             None, traceback: Optional[types.TracebackType] = None) → None
```

Throw an exception into the asynchronous iterator.

```
property is_closed: bool
```

Check whether the iterator is closed.

## 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) → bool
```

```
graphql.type.is_enum_type(type_: Any) → bool
```

```
graphql.type.is_input_object_type(type_: Any) → bool
```

```
graphql.type.is_input_type(type_: Any) → bool
```

```
graphql.type.is_interface_type(type_: Any) → bool
```

```
graphql.type.is_leaf_type(type_: Any) → bool
graphql.type.is_list_type(type_: Any) → bool
graphql.type.is_named_type(type_: Any) → bool
graphql.type.is_non_null_type(type_: Any) → bool
graphql.type.is_nullable_type(type_: Any) → bool
graphql.type.is_object_type(type_: Any) → bool
graphql.type.is_output_type(type_: Any) → bool
graphql.type.is_scalar_type(type_: Any) → bool
graphql.type.is_type(type_: Any) → bool
graphql.type.is_union_type(type_: Any) → bool
graphql.type.is_wrapping_type(type_: Any) → bool
```

## Assertions

```
graphql.type.assert_abstract_type(type_: Any) → Union[graphql.type.definition.GraphQLInterfaceType,
                                                       graphql.type.definition.GraphQLUnionType]
graphql.type.assert_composite_type(type_: Any) → graphql.type.definition.GraphQLType
graphql.type.assert_enum_type(type_: Any) → graphql.type.definition.GraphQLEnumType
graphql.type.assert_input_object_type(type_: Any) → graphql.type.definition.GraphQLInputObjectType
graphql.type.assert_input_type(type_: Any) → Union[graphql.type.definition.GraphQLScalarType,
                                                    graphql.type.definition.GraphQLEnumType,
                                                    graphql.type.definition.GraphQLInputObjectType,
                                                    graphql.type.definition.GraphQLWrappingType]
graphql.type.assert_interface_type(type_: Any) → graphql.type.definition.GraphQLInterfaceType
graphql.type.assert_leaf_type(type_: Any) → Union[graphql.type.definition.GraphQLScalarType,
                                                  graphql.type.definition.GraphQLEnumType]
graphql.type.assert_list_type(type_: Any) → graphql.type.definition.GraphQLList
graphql.type.assert_named_type(type_: Any) → graphql.type.definition.GraphQLNamedType
graphql.type.assert_non_null_type(type_: Any) → graphql.type.definition.GraphQLNonNull
graphql.type.assert_nullable_type(type_: Any) → Union[graphql.type.definition.GraphQLScalarType,
                                                       graphql.type.definition.GraphQLOBJECTType,
                                                       graphql.type.definition.GraphQLInterfaceType,
                                                       graphql.type.definition.GraphQLUnionType,
                                                       graphql.type.definition.GraphQLEnumType,
                                                       graphql.type.definition.GraphQLInputObjectType,
                                                       graphql.type.definition.GraphQLList]
```

```
graphql.type.assert_object_type(type_: Any) → graphql.type.definition.GraphQLOBJECTType
graphql.type.assert_output_type(type_: Any) → Union[graphql.type.definition.GraphQLScalarType,
                                                    graphql.type.definition.GraphQLOBJECTType,
                                                    graphql.type.definition.GraphQLInterfaceType,
                                                    graphql.type.definition.GraphQLUnionType,
                                                    graphql.type.definition.GraphQLEnumType,
                                                    graphql.type.definition.GraphQLWrappingType]
graphql.type.assert_scalar_type(type_: Any) → graphql.type.definition.GraphQLScalarType
graphql.type.assert_type(type_: Any) → graphql.type.definition.GraphQLType
graphql.type.assert_union_type(type_: Any) → graphql.type.definition.GraphQLUnionType
graphql.type.assert_wrapping_type(type_: Any) → graphql.type.definition.GraphQLWrappingType
```

## Un-modifiers

```
graphql.type.get_nullable_type(type_: None) → None
graphql.type.get_nullable_type(type_: Union[graphql.type.definition.GraphQLScalarType,
                                           graphql.type.definition.GraphQLOBJECTType,
                                           graphql.type.definition.GraphQLInterfaceType,
                                           graphql.type.definition.GraphQLUnionType,
                                           graphql.type.definition.GraphQLEnumType,
                                           graphql.type.definition.GraphQLInputObjectType,
                                           graphql.type.definition.GraphQLList]) →
    Union[graphql.type.definition.GraphQLScalarType,
          graphql.type.definition.GraphQLOBJECTType,
          graphql.type.definition.GraphQLInterfaceType,
          graphql.type.definition.GraphQLUnionType,
          graphql.type.definition.GraphQLEnumType,
          graphql.type.definition.GraphQLInputObjectType,
          graphql.type.definition.GraphQLList]
graphql.type.get_nullable_type(type_: graphql.type.definition.GraphQLNonNull) →
    Union[graphql.type.definition.GraphQLScalarType,
          graphql.type.definition.GraphQLOBJECTType,
          graphql.type.definition.GraphQLInterfaceType,
          graphql.type.definition.GraphQLUnionType,
          graphql.type.definition.GraphQLEnumType,
          graphql.type.definition.GraphQLInputObjectType,
          graphql.type.definition.GraphQLList]
```

Unwrap possible non-null type

```
graphql.type.get_named_type(type_: None) → None
graphql.type.get_named_type(type_: graphql.type.definition.GraphQLType) →
    graphql.type.definition.GraphQLNamedType
```

Unwrap possible wrapping type



## Definitions

```
class graphql.type.GraphQLEnumType(name: str, values: Union[Dict[str,
    graphql.type.definition.GraphQLEnumValue], Dict[str, Any],
    Type[enum.Enum]], description: Optional[str] = None, extensions:
    Optional[Dict[str, Any]] = None, ast_node:
    Optional[graphql.language.ast.EnumTypeDefinitionNode] = None,
    extension_ast_nodes:
    Optional[Collection[graphql.language.ast.EnumTypeExtensionNode]]
    = None)
```

Bases: `graphql.type.definition.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.

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, graphql.type.definition.GraphQLEnumValue], Dict[str, Any],
    Type[enum.Enum]], description: Optional[str] = None, extensions: Optional[Dict[str, Any]] =
    None, ast_node: Optional[graphql.language.ast.EnumTypeDefinitionNode] = None,
    extension_ast_nodes: Optional[Collection[graphql.language.ast.EnumTypeExtensionNode]] =
    None) → None
```

**ast\_node:** Optional[`graphql.language.ast.EnumTypeDefinitionNode`]

**description:** Optional[str]

**extension\_ast\_nodes:** Optional[`graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.EnumTypeExtensionNode]`]

**extensions:** Optional[Dict[str, Any]]

**name:** str

**parse\_literal**(value\_node: graphql.language.ast.ValueNode, \_variables: Optional[Dict[str, Any]] = None) → Any

**parse\_value**(input\_value: str) → Any

**serialize**(output\_value: Any) → str

**to\_kwargs**() → Dict[str, Any]

**values:** Dict[str, [graphql.type.definition.GraphQLEnumValue](#)]

```
class graphql.type.GraphQLInputObjectType(name: str, fields: Union[Callable[[], Dict[str,
    graphql.type.definition.GraphQLInputField]], Dict[str,
    graphql.type.definition.GraphQLInputField]], description:
    Optional[str] = None, out_type: Optional[Callable[[Dict[str,
    Any]], Any]] = None, extensions: Optional[Dict[str, Any]] =
    None, ast_node: Op-
    tional[graphql.language.ast.InputObjectTypeDefinitionNode]
    = None, extension_ast_nodes: Op-
    tional[Collection[graphql.language.ast.InputObjectTypeExtensionNode]]
    = None)
```

Bases: [graphql.type.definition.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[[], Dict[str, graphql.type.definition.GraphQLInputField]],
    Dict[str, graphql.type.definition.GraphQLInputField]], description: Optional[str] = None,
    out_type: Optional[Callable[[Dict[str, Any]], Any]] = None, extensions: Optional[Dict[str, Any]]
    = None, ast_node: Optional[graphql.language.ast.InputObjectTypeDefinitionNode] = None,
    extension_ast_nodes: Optional[Collection[graphql.language.ast.InputObjectTypeExtensionNode]]
    = None) → None
```

**ast\_node:** Optional[[graphql.language.ast.InputObjectTypeDefinitionNode](#)]

**description:** Optional[str]

**extension\_ast\_nodes:** `Optional[graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.InputObjectTypeExtensionNode]]`

**extensions:** `Optional[Dict[str, Any]]`

**fields**

Get provided fields, wrap them as GraphQLInputField if needed.

**name:** `str`

**static out\_type**(*value: Dict[str, Any]*)  $\rightarrow$  Any

Transform outbound values (this is an extension of GraphQL.js).

This default implementation passes values unaltered as dictionaries.

**to\_kwargs**()  $\rightarrow$  Dict[str, Any]

```
class graphql.type.GraphQLInterfaceType(name: str, fields: Optional[Union[Callable[[], Dict[str, graphql.type.definition.GraphQLField]], Dict[str, graphql.type.definition.GraphQLField]] = None, interfaces: Optional[Union[Callable[[], Collection[graphql.type.definition.GraphQLInterfaceType]], Collection[graphql.type.definition.GraphQLInterfaceType]] = None, resolve_type: Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo, Union[graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType]], Optional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType, str]]], graphql.type.definition.GraphQLObjectType, str]]] = None, description: Optional[str] = None, extensions: Optional[Dict[str, Any]] = None, ast_node: Optional[graphql.language.ast.InterfaceTypeDefinitionNode] = None, extension_ast_nodes: Optional[Collection[graphql.language.ast.InterfaceTypeExtensionNode]] = None)
```

Bases: `graphql.type.definition.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: Optional[Union[Callable[[], Dict[str, graphql.type.definition.GraphQLField]], Dict[str, graphql.type.definition.GraphQLField]]] = None, interfaces: Optional[Union[Callable[[], Collection[graphql.type.definition.GraphQLInterfaceType]], Collection[graphql.type.definition.GraphQLInterfaceType]]] = None, resolve_type: Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo, Union[graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType]], Optional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType, str]]], graphql.type.definition.GraphQLObjectType, str]]]] = None, description: Optional[str] = None, extensions: Optional[Dict[str, Any]] = None, ast_node: Optional[graphql.language.ast.InterfaceTypeDefinitionNode] = None, extension_ast_nodes: Optional[Collection[graphql.language.ast.InterfaceTypeExtensionNode]] = None) → None
```

**ast\_node:** Optional[[graphql.language.ast.InterfaceTypeDefinitionNode](#)]

**description:** Optional[str]

**extension\_ast\_nodes:** Optional[[graphql.pyutils.frozen\\_list.FrozenList](#)[[graphql.language.ast.InterfaceTypeExtensionNode](#)]]

**extensions:** Optional[Dict[str, Any]]

**fields**

Get provided fields, wrapping them as GraphQLFields if needed.

**interfaces**

Get provided interfaces.

**name:** str

**resolve\_type:** Optional[Callable[[Any, [graphql.type.definition.GraphQLResolveInfo](#), Union[[graphql.type.definition.GraphQLInterfaceType](#), [graphql.type.definition.GraphQLUnionType](#)]], Optional[Union[Awaitable[Optional[Union[[graphql.type.definition.GraphQLObjectType](#), str]]], [graphql.type.definition.GraphQLObjectType](#), str]]]]]

**to\_kwargs**() → Dict[str, Any]

```
class graphql.type.GraphQLObjectType(name: str, fields: Union[Callable[[], Dict[str, graphql.type.definition.GraphQLField]], Dict[str, graphql.type.definition.GraphQLField]], interfaces: Optional[Union[Callable[[], Collection[graphql.type.definition.GraphQLInterfaceType]], Collection[graphql.type.definition.GraphQLInterfaceType]]] = None, is_type_of: Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo, Union[Awaitable[bool], bool]]] = None, extensions: Optional[Dict[str, Any]] = None, description: Optional[str] = None, ast_node: Optional[graphql.language.ast.ObjectTypeDefinitionNode] = None, extension_ast_nodes: Optional[Collection[graphql.language.ast.ObjectTypeExtensionNode]] = None)
```

Bases: [graphql.type.definition.GraphQLNamedType](#)

Object Type Definition

Almost all of 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[[], Dict[str, graphql.type.definition.GraphQLField]], Dict[str,
    graphql.type.definition.GraphQLField]], interfaces: Optional[Union[Callable[[],
    Collection[graphql.type.definition.GraphQLInterfaceType]],
    Collection[graphql.type.definition.GraphQLInterfaceType]]] = None, is_type_of:
    Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo], Union[Awaitable[bool],
    bool]]] = None, extensions: Optional[Dict[str, Any]] = None, description: Optional[str] = None,
    ast_node: Optional[graphql.language.ast.ObjectTypeDefinitionNode] = None,
    extension_ast_nodes: Optional[Collection[graphql.language.ast.ObjectTypeExtensionNode]] =
    None) → None
```

**ast\_node:** Optional[[graphql.language.ast.ObjectTypeDefinitionNode](#)]

**description:** Optional[str]

**extension\_ast\_nodes:** Optional[[graphql.pyutils.frozen\\_list.FrozenList](#)[[graphql.language.ast.ObjectTypeExtensionNode](#)]]

**extensions:** Optional[Dict[str, Any]]

**fields**

Get provided fields, wrapping them as GraphQLFields if needed.

**interfaces**

Get provided interfaces.

**is\_type\_of:** Optional[Callable[[Any, [graphql.type.definition.GraphQLResolveInfo](#)], Union[Awaitable[bool], bool]]]

**name:** str

**to\_kwargs()** → Dict[str, Any]

```
class graphql.type.GraphQLScalarType(name: str, serialize: Optional[Callable[[Any], Any]] = None,
                                     parse_value: Optional[Callable[[Any], Any]] = None,
                                     parse_literal: Optional[Callable[[graphql.language.ast.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[graphql.language.ast.ScalarTypeDefinitionNode] = None,
                                     extension_ast_nodes: Op-
                                     tional[Collection[graphql.language.ast.ScalarTypeExtensionNode]]
                                     = None)
```

Bases: [graphql.type.definition.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 does not return a value (i.e. it returns `None`), then no error will be included in the response.

Example:

```
def serialize_odd(value):
    if value % 2 == 1:
        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[[graphql.language.ast.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[graphql.language.ast.ScalarTypeDefinitionNode] = None, extension_ast_nodes:
Optional[Collection[graphql.language.ast.ScalarTypeExtensionNode]] = None) → None
```

**ast\_node:** Optional[[graphql.language.ast.ScalarTypeDefinitionNode](#)]

**description:** Optional[str]

**extension\_ast\_nodes:** Optional[[graphql.pyutils.frozen\\_list.FrozenList](#)[[graphql.language.ast.ScalarTypeExtensionNode](#)]]

**extensions:** Optional[Dict[str, Any]]

**name:** str

**parse\_literal**(node: [graphql.language.ast.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.

**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:** *Optional[str]*

**to\_kwargs**() → *Dict[str, Any]*

```
class graphql.type.GraphQLUnionType(name: str, types: Union[Callable[],
    Collection[graphql.type.definition.GraphQLObjectType]],
    Collection[graphql.type.definition.GraphQLObjectType]],
    resolve_type: Optional[Callable[[Any,
    graphql.type.definition.GraphQLResolveInfo,
    Union[graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType]], Op-
    tional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType,
    str]]], graphql.type.definition.GraphQLObjectType, str]]] = None,
    description: Optional[str] = None, extensions: Optional[Dict[str,
    Any]] = None, ast_node:
    Optional[graphql.language.ast.UnionTypeDefinitionNode] = None,
    extension_ast_nodes: Op-
    tional[Collection[graphql.language.ast.UnionTypeExtensionNode]] =
    None)
```

Bases: *graphql.type.definition.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[graphql.type.definition.GraphQLObjectType]],
    Collection[graphql.type.definition.GraphQLObjectType]], resolve_type: Optional[Callable[[Any,
    graphql.type.definition.GraphQLResolveInfo,
    Union[graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType]],
    Optional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType, str]]],
    graphql.type.definition.GraphQLObjectType, str]]] = None, description: Optional[str] = None,
    extensions: Optional[Dict[str, Any]] = None, ast_node:
    Optional[graphql.language.ast.UnionTypeDefinitionNode] = None, extension_ast_nodes:
    Optional[Collection[graphql.language.ast.UnionTypeExtensionNode]] = None) → None
```

**ast\_node:** *Optional[graphql.language.ast.UnionTypeDefinitionNode]*

**description:** *Optional[str]*



```
extension_ast_nodes: Optional[graphql.pyutils.frozen_list.FrozenList[graphql.language.ast.UnionTypeExtensionNode]]

extensions: Optional[Dict[str, Any]]

name: str

resolve_type: Optional[Callable[[Any, graphql.type.definition.GraphQLResolveInfo,
Union[graphql.type.definition.GraphQLInterfaceType,
graphql.type.definition.GraphQLUnionType]],
Optional[Union[Awaitable[Optional[Union[graphql.type.definition.GraphQLObjectType,
str]]], graphql.type.definition.GraphQLObjectType, str]]]]

to_kwargs() → Dict[str, Any]

types
    Get provided types.
```

## Type Wrappers

**class** `graphql.type.GraphQLList`(*type\_*: *graphql.type.definition.GT*)

Bases: `Generic[graphql.type.definition.GT]`, `graphql.type.definition.GraphQLWrappingType[graphql.type.definition.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\_*: *graphql.type.definition.GT*) → None

**of\_type**: `graphql.type.definition.GT`

**class** `graphql.type.GraphQLNonNull`(*type\_*: *graphql.type.definition.GNT*)

Bases: `graphql.type.definition.GraphQLWrappingType[graphql.type.definition.GNT]`, `Generic[graphql.type.definition.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_: graphql.type.definition.GNT)
```

```
of_type: graphql.type.definition.GT
```

## Types

graphql.type.GraphQLAbstractType

alias of Union[graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType]

```
class graphql.type.GraphQLArgument(type_: Union[graphql.type.definition.GraphQLScalarType,
        graphql.type.definition.GraphQLEnumType,
        graphql.type.definition.GraphQLInputObjectType,
        graphql.type.definition.GraphQLWrappingType], 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[graphql.language.ast.InputValueDefinitionNode] = None)
```

Bases: object

Definition of a GraphQL argument

```
__init__(type_: Union[graphql.type.definition.GraphQLScalarType,
        graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType,
        graphql.type.definition.GraphQLWrappingType], 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[graphql.language.ast.InputValueDefinitionNode] = None) → None
```

```
ast_node: Optional[graphql.language.ast.InputValueDefinitionNode]
```

```
default_value: Any
```

```
deprecation_reason: Optional[str]
```

```
description: Optional[str]
```

```
extensions: Optional[Dict[str, Any]]
```

```
out_name: Optional[str]
```

```
to_kwargs() → Dict[str, Any]
```

```
type: Union[graphql.type.definition.GraphQLScalarType,
        graphql.type.definition.GraphQLEnumType,
        graphql.type.definition.GraphQLInputObjectType,
        graphql.type.definition.GraphQLWrappingType]
```

`graphql.type.GraphQLArgumentMap`

alias of `Dict[str, GraphQLArgument]`

`graphql.type.GraphQLCompositeType`

alias of `Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.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[graphql.language.ast.EnumValueDefinitionNode] = None)
```

Bases: `object`

```
__init__(value: Optional[Any] = None, description: Optional[str] = None, deprecation_reason:
Optional[str] = None, extensions: Optional[Dict[str, Any]] = None, ast_node:
Optional[graphql.language.ast.EnumValueDefinitionNode] = None) → None
```

**ast\_node:** `Optional[graphql.language.ast.EnumValueDefinitionNode]`

**deprecation\_reason:** `Optional[str]`

**description:** `Optional[str]`

**extensions:** `Optional[Dict[str, Any]]`

**property is\_deprecated:** `bool`

**to\_kwargs()** → `Dict[str, Any]`

**value:** `Any`

`graphql.type.GraphQLEnumValueMap`

alias of `Dict[str, GraphQLEnumValue]`

```
class graphql.type.GraphQLField(type_: Union[graphql.type.definition.GraphQLScalarType,
graphql.type.definition.GraphQLObjectType,
graphql.type.definition.GraphQLInterfaceType,
graphql.type.definition.GraphQLUnionType,
graphql.type.definition.GraphQLEnumType,
graphql.type.definition.GraphQLWrappingType], args: Optional[Dict[str,
graphql.type.definition.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[graphql.language.ast.FieldDefinitionNode] = None)
```

Bases: `object`

Definition of a GraphQL field

```
__init__(type_: Union[graphql.type.definition.GraphQLScalarType,
graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType,
graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType,
graphql.type.definition.GraphQLWrappingType], args: Optional[Dict[str,
graphql.type.definition.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[graphql.language.ast.FieldDefinitionNode] = None) → None
```

```

args: Dict[str, graphql.type.definition.GraphQLArgument]
ast_node: Optional[graphql.language.ast.FieldDefinitionNode]
deprecation_reason: Optional[str]
description: Optional[str]
extensions: Optional[Dict[str, Any]]
property is_deprecated: bool
resolve: Optional[Callable[[...], Any]]
subscribe: Optional[Callable[[...], Any]]
to_kwargs() → Dict[str, Any]

type: Union[graphql.type.definition.GraphQLScalarType,
graphql.type.definition.GraphQLObjectType,
graphql.type.definition.GraphQLInterfaceType,
graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType,
graphql.type.definition.GraphQLWrappingType]

```

`graphql.type.GraphQLFieldMap`

alias of Dict[str, *graphql.type.definition.GraphQLField*]

```

class graphql.type.GraphQLInputField(type_: Union[graphql.type.definition.GraphQLScalarType,
graphql.type.definition.GraphQLEnumType,
graphql.type.definition.GraphQLInputObjectType,
graphql.type.definition.GraphQLWrappingType], 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[graphql.language.ast.InputValueDefinitionNode] = None)

```

Bases: object

Definition of a GraphQL input field

```

__init__(type_: Union[graphql.type.definition.GraphQLScalarType,
graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType,
graphql.type.definition.GraphQLWrappingType], 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[graphql.language.ast.InputValueDefinitionNode] = None) → None

```

**ast\_node:** Optional[*graphql.language.ast.InputValueDefinitionNode*]

**default\_value:** Any

**deprecation\_reason:** Optional[str]

**description:** Optional[str]

**extensions:** Optional[Dict[str, Any]]

**out\_name:** Optional[str]

`to_kwargs() → Dict[str, Any]`

`type: Union[graphql.type.definition.GraphQLScalarType,  
graphql.type.definition.GraphQLEnumType,  
graphql.type.definition.GraphQLInputObjectType,  
graphql.type.definition.GraphQLWrappingType]`

`graphql.type.GraphQLInputFieldMap`

alias of `Dict[str, GraphQLInputField]`

`graphql.type.GraphQLInputType`

alias of `Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType]`

`graphql.type.GraphQLLeafType`

alias of `Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType]`

`class graphql.type.GraphQLNamedType(name: str, description: Optional[str] = None, extensions: Optional[Dict[str, Any]] = None, ast_node: Optional[graphql.language.ast.TypeDefinitionNode] = None, extension_ast_nodes: Optional[Collection[graphql.language.ast.TypeExtensionNode]] = None)`

Bases: `graphql.type.definition.GraphQLType`

Base class for all GraphQL named types

`__init__(name: str, description: Optional[str] = None, extensions: Optional[Dict[str, Any]] = None, ast_node: Optional[graphql.language.ast.TypeDefinitionNode] = None, extension_ast_nodes: Optional[Collection[graphql.language.ast.TypeExtensionNode]] = None) → None`

`ast_node: Optional[graphql.language.ast.TypeDefinitionNode]`

`description: Optional[str]`

`extension_ast_nodes: Optional[graphql.pyutils.frozen\_list.FrozenList[graphql.language.ast.TypeExtensionNode]]`

`extensions: Optional[Dict[str, Any]]`

`name: str`

`to_kwargs() → Dict[str, Any]`

`graphql.type.GraphQLNullableType`

alias of `Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLList]`

`graphql.type.GraphQLOutputType`

alias of `Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLWrappingType]`

**class** `graphql.type.GraphQLType`

Bases: `object`

Base class for all GraphQL types

`__init__()`

**class** `graphql.type.GraphQLWrappingType`(*type\_*: `graphql.type.definition.GT`)

Bases: `graphql.type.definition.GraphQLType`, `Generic[graphql.type.definition.GT]`

Base class for all GraphQL wrapping types

`__init__(type_: graphql.type.definition.GT) → None`

**of\_type:** `graphql.type.definition.GT`

`graphql.type.Thunk`

alias of `Union[Callable[[], graphql.type.definition.T], graphql.type.definition.T]`

## Resolvers

`graphql.type.GraphQLFieldResolver`

alias of `Callable[[...], Any]`

`graphql.type.GraphQLIsTypeOfFn`

alias of `Callable[[Any, graphql.type.definition.GraphQLResolveInfo], Union[Awaitable[bool], bool]]`

**class** `graphql.type.GraphQLResolveInfo`(*field\_name*: `str`, *field\_nodes*: `List[graphql.language.ast.FieldNode]`, *return\_type*: `GraphQLObjectType`, *parent\_type*: `GraphQLObjectType`, *path*: `graphql.pyutils.path.Path`, *schema*: `GraphQLSchema`, *fragments*: `Dict[str, graphql.language.ast.FragmentDefinitionNode]`, *root\_value*: `Any`, *operation*: `graphql.language.ast.OperationDefinitionNode`, *variable\_values*: `Dict[str, Any]`, *context*: `Any`, *is\_awaitable*: `Callable[[Any], bool]`)

Bases: `tuple`

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__()`

**property context**

Alias for field number 10

**count**(*value*, /)

Return number of occurrences of value.

**property field\_name**

Alias for field number 0

**property field\_nodes**

Alias for field number 1

**property fragments**

Alias for field number 6

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

Return first index of value.

Raises ValueError if the value is not present.

**property is\_awaitable**

Alias for field number 11

**property operation**

Alias for field number 8

**property parent\_type**

Alias for field number 3

**property path**

Alias for field number 4

**property return\_type**

Alias for field number 2

**property root\_value**

Alias for field number 7

**property schema**

Alias for field number 5

**property variable\_values**

Alias for field number 9

graphql.type.GraphQLTypeResolver

alias of Callable[[Any, [graphql.type.definition.GraphQLResolveInfo](#), [GraphQLAbstractType](#)],  
Optional[Union[Awaitable[Optional[Union[[GraphQLObjectType](#), str]]], [GraphQLObjectType](#), str]]]

## Directives

### Predicates

graphql.type.is\_directive(directive: Any) → bool

Test if the given value is a GraphQL directive.

graphql.type.is\_specified\_directive(directive: [graphql.type.directives.GraphQLDirective](#)) → bool

Check whether the given directive is one of the specified directives.

## Definitions

```
class graphql.type.GraphQLDirective(name: str, locations:
    Collection[graphql.language.directive_locations.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[graphql.language.ast.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[graphql.language.directive_locations.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[graphql.language.ast.DirectiveDefinitionNode] = None) → None
```

**args:** Dict[str, [graphql.type.definition.GraphQLArgument](#)]

**ast\_node:** Optional[[graphql.language.ast.DirectiveDefinitionNode](#)]

**description:** Optional[str]

**extensions:** Optional[Dict[str, Any]]

**is\_repeatable:** bool

**locations:** List[[graphql.language.directive\\_locations.DirectiveLocation](#)]

**name:** str

**to\_kwargs()** → Dict[str, Any]

**graphql.type.GraphQLIncludeDirective**

alias of <GraphQLDirective(@include)>

**graphql.type.GraphQLSkipDirective**

alias of <GraphQLDirective(@skip)>

**graphql.type.GraphQLDeprecatedDirective**

alias of <GraphQLDirective(@deprecated)>

```
graphql.type.specified_directives = [<GraphQLDirective(@include)>,
<GraphQLDirective(@skip)>, <GraphQLDirective(@deprecated)>,
<GraphQLDirective(@specifiedBy)>]
```

The full list of specified directives.

**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_: graphql.type.definition.GraphQLNamedType) → bool`

Check whether the given named GraphQL type is an introspection type.

### Definitions

`class graphql.type.TypeKind(value)`

Bases: `enum.Enum`

An enumeration.

`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 = {'__Directive': <GraphQLObjectType '__Directive'>, '__DirectiveLocation': <GraphQLEnumType '__DirectiveLocation'>, '__EnumValue': <GraphQLObjectType '__EnumValue'>, '__Field': <GraphQLObjectType '__Field'>, '__InputValue': <GraphQLObjectType '__InputValue'>, '__Schema': <GraphQLObjectType '__Schema'>, '__Type': <GraphQLObjectType '__Type'>, '__TypeKind': <GraphQLEnumType '__TypeKind'>}`

A dictionary containing all introspection types.



## Scalars

### Predicates

`graphql.type.is_specified_scalar_type(type_: graphql.type.definition.GraphQLNamedType) → bool`  
 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'>

The list of all specified directives is available as [`specified\_directives`](#).

## Schema

### Predicates

`graphql.type.is_schema(schema: Any) → bool`  
 Test if the given value is a GraphQL schema.

### Definitions

```
class graphql.type.GraphQLSchema(query: Optional[graphql.type.definition.GraphQLObjectType] = None,
                                mutation: Optional[graphql.type.definition.GraphQLObjectType] =
                                None, subscription:
                                Optional[graphql.type.definition.GraphQLObjectType] = None, types:
                                Optional[Collection[graphql.type.definition.GraphQLNamedType]] =
                                None, directives:
                                Optional[Collection[graphql.type.directives.GraphQLDirective]] =
                                None, description: Optional[str] = None, extensions: Optional[Dict[str,
                                Any]] = None, ast_node:
                                Optional[graphql.language.ast.SchemaDefinitionNode] = None,
                                extension_ast_nodes:
                                Optional[Collection[graphql.language.ast.SchemaExtensionNode]] =
                                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.

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(
    '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: Optional[graphql.type.definition.GraphQLObjectType] = None, mutation:
    Optional[graphql.type.definition.GraphQLObjectType] = None, subscription:
    Optional[graphql.type.definition.GraphQLObjectType] = None, types:
    Optional[Collection[graphql.type.definition.GraphQLNamedType]] = None, directives:
    Optional[Collection[graphql.type.directives.GraphQLDirective]] = None, description:
    Optional[str] = None, extensions: Optional[Dict[str, Any]] = None, ast_node:
    Optional[graphql.language.ast.SchemaDefinitionNode] = None, extension_ast_nodes:
    Optional[Collection[graphql.language.ast.SchemaExtensionNode]] = 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:** Optional[`graphql.language.ast.SchemaDefinitionNode`]

```

description: Optional[str]

directives:
    graphql.pyutils.frozen_list.FrozenList[graphql.type.directives.GraphQLDirective]

extension_ast_nodes: Optional[graphql.pyutils.frozen_list.FrozenList[graphql.
    language.ast.SchemaExtensionNode]]

extensions: Optional[Dict[str, Any]]

get_directive(name: str) → Optional[graphql.type.directives.GraphQLDirective]

get_implementations(interface_type: graphql.type.definition.GraphQLInterfaceType) →
    graphql.type.schema.InterfaceImplementations

get_possible_types(abstract_type: Union[graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType]) →
    List[graphql.type.definition.GraphQLObjectType]
    Get list of all possible concrete types for given abstract type.

get_type(name: str) → Optional[graphql.type.definition.GraphQLNamedType]

is_possible_type(abstract_type: Union[graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType], possible_type:
    graphql.type.definition.GraphQLObjectType) → bool
    Check whether a concrete type is possible for an abstract type.
    Deprecated: Use is_sub_type() instead.

is_sub_type(abstract_type: Union[graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType], maybe_sub_type:
    graphql.type.definition.GraphQLNamedType) → bool
    Check whether a type is a subtype of a given abstract type.

mutation_type: Optional[graphql.type.definition.GraphQLObjectType]

query_type: Optional[graphql.type.definition.GraphQLObjectType]

subscription_type: Optional[graphql.type.definition.GraphQLObjectType]

to_kwargs() → Dict[str, Any]

type_map: Dict[str, graphql.type.definition.GraphQLNamedType]

property validation_errors:
    Optional[List[graphql.error.graphql_error.GraphQLError]]

```

## Validate

### Functions:

```

graphql.type.validate_schema(schema: graphql.type.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: graphql.type.schema.GraphQLSchema) → None`

Utility function which asserts a schema is valid.

Throws a `TypeError` if the schema is invalid.

## 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.

Get the target Operation from a Document:

```
graphql.utilities.get_operation_ast(document_ast: graphql.language.ast.DocumentNode,
                                   operation_name: Optional[str] = None) →
                                   Optional[graphql.language.ast.OperationDefinitionNode]
```

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.

Get the Type for the target Operation AST:

```
graphql.utilities.get_operation_root_type(schema: graphql.type.schema.GraphQLSchema, operation:
                                         Union[graphql.language.ast.OperationDefinitionNode,
                                         graphql.language.ast.OperationTypeDefinitionNode]) →
                                         graphql.type.definition.GraphQLOBJECTType
```

Extract the root type of the operation from the schema.

Convert a GraphQLSchema to an IntrospectionQuery:

```
graphql.utilities.introspection_from_schema(schema: graphql.type.schema.GraphQLSchema,
                                           descriptions: bool = True, specified_by_url: bool = True,
                                           directive_is_repeatable: bool = True, schema_description:
                                           bool = True, input_value_deprecation: bool = True) →
                                           Dict[str, Any]
```

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: Dict, assume_valid: bool = False) →  
    graphql.type.schema.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: graphql.language.ast.DocumentNode, assume_valid:  
    bool = False, assume_valid_sdl: bool = False) →  
    graphql.type.schema.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 and Mutation.

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: Union[str, graphql.language.source.Source], assume_valid: bool =  
    False, assume_valid_sdl: bool = False, no_location: bool = False,  
    experimental_fragment_variables: bool = False) →  
    graphql.type.schema.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: graphql.type.schema.GraphQLSchema, document_ast:  
    graphql.language.ast.DocumentNode, assume_valid: bool = False,  
    assume_valid_sdl: bool = False) →  
    graphql.type.schema.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: graphql.type.schema.GraphQLSchema) → graphql.type.schema.GraphQLSchema`

Sort GraphQLSchema.

This function returns a sorted copy of the given GraphQLSchema.

Print a GraphQLSchema to GraphQL Schema language:

`graphql.utilities.print_introspection_schema(schema: graphql.type.schema.GraphQLSchema) → str`

`graphql.utilities.print_schema(schema: graphql.type.schema.GraphQLSchema) → str`

`graphql.utilities.print_type(type_: graphql.type.definition.GraphQLNamedType) → str`

`graphql.utilities.print_value(value: Any, type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType]) → str`

@deprecated: Convenience function for printing a Python value

Create a GraphQLType from a GraphQL language AST:

`graphql.utilities.type_from_ast(schema: graphql.type.schema.GraphQLSchema, type_node: graphql.language.ast.NamedTypeNode) → Optional[graphql.type.definition.GraphQLNamedType]`

`graphql.utilities.type_from_ast(schema: graphql.type.schema.GraphQLSchema, type_node: graphql.language.ast.ListTypeNode) → Optional[graphql.type.definition.GraphQLList]`

`graphql.utilities.type_from_ast(schema: graphql.type.schema.GraphQLSchema, type_node: graphql.language.ast.NonNullTypeNode) → Optional[graphql.type.definition.GraphQLNonNull]`

`graphql.utilities.type_from_ast(schema: graphql.type.schema.GraphQLSchema, type_node: graphql.language.ast.TypeNode) → Optional[graphql.type.definition.GraphQLType]`

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.

Create a Python value from a GraphQL language AST with a type:

`graphql.utilities.value_from_ast(value_node: Optional[graphql.language.ast.ValueNode], type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType], 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: graphql.language.ast.ValueNode, variables:
Optional[Dict[str, Any]] = 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[graphql.type.definition.GraphQLScalarType,
graphql.type.definition.GraphQLEnumType,
graphql.type.definition.GraphQLInputObjectType,
graphql.type.definition.GraphQLWrappingType]) →
Optional[graphql.language.ast.ValueNode]
```

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: graphql.type.schema.GraphQLSchema, get_field_def_fn:
    Optional[Callable[[graphql.type.schema.GraphQLSchema,
    graphql.type.definition.GraphQLType, graphql.language.ast.FieldNode],
    Optional[graphql.type.definition.GraphQLField]]] = None, initial_type:
    Optional[graphql.type.definition.GraphQLType] = 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: graphql.type.schema.GraphQLSchema, get_field_def_fn:
    Optional[Callable[[graphql.type.schema.GraphQLSchema, graphql.type.definition.GraphQLType,
    graphql.language.ast.FieldNode], Optional[graphql.type.definition.GraphQLField]]] = None,
    initial_type: Optional[graphql.type.definition.GraphQLType] = None) → None
```

Initialize the TypeInfo for the given GraphQL schema.

The experimental optional second parameter is only needed in order to support non-spec-compliant code bases. You should never need to use it. It may disappear in the future.

Initial type may be provided in rare cases to facilitate traversals beginning somewhere other than documents.

**enter**(node: graphql.language.ast.Node) → None

**enter\_argument**(node: graphql.language.ast.ArgumentNode) → None

**enter\_directive**(node: graphql.language.ast.DirectiveNode) → None

**enter\_enum\_value**(node: graphql.language.ast.EnumValueNode) → None

**enter\_field**(node: graphql.language.ast.FieldNode) → None

**enter\_fragment\_definition**(node: graphql.language.ast.InlineFragmentNode) → None

**enter\_inline\_fragment**(node: graphql.language.ast.InlineFragmentNode) → None

**enter\_list\_value**(node: graphql.language.ast.ListValueNode) → None

**enter\_object\_field**(node: graphql.language.ast.ObjectFieldNode) → None

**enter\_operation\_definition**(node: graphql.language.ast.OperationDefinitionNode) → None

**enter\_selection\_set**(node: graphql.language.ast.SelectionSetNode) → None

**enter\_variable\_definition**(node: graphql.language.ast.VariableDefinitionNode) → None

**get\_argument**() → Optional[graphql.type.definition.GraphQLArgument]

**get\_default\_value**() → Any

**get\_directive**() → Optional[graphql.type.directives.GraphQLDirective]

**get\_enum\_value**() → Optional[graphql.type.definition.GraphQLEnumValue]

**get\_field\_def**() → Optional[graphql.type.definition.GraphQLField]



**get\_input\_type()** → Optional[Union[*graphql.type.definition.GraphQLScalarType*,  
*graphql.type.definition.GraphQLEnumType*,  
*graphql.type.definition.GraphQLInputObjectType*,  
*graphql.type.definition.GraphQLWrappingType*]]

**get\_parent\_input\_type()** → Optional[Union[*graphql.type.definition.GraphQLScalarType*,  
*graphql.type.definition.GraphQLEnumType*,  
*graphql.type.definition.GraphQLInputObjectType*,  
*graphql.type.definition.GraphQLWrappingType*]]

**get\_parent\_type()** → Optional[Union[*graphql.type.definition.GraphQLObjectType*,  
*graphql.type.definition.GraphQLInterfaceType*,  
*graphql.type.definition.GraphQLUnionType*]]

**get\_type()** → Optional[Union[*graphql.type.definition.GraphQLScalarType*,  
*graphql.type.definition.GraphQLObjectType*, *graphql.type.definition.GraphQLInterfaceType*,  
*graphql.type.definition.GraphQLUnionType*, *graphql.type.definition.GraphQLEnumType*,  
*graphql.type.definition.GraphQLWrappingType*]]

**leave**(*node*: *graphql.language.ast.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*: *graphql.utilities.type\_info.TypeInfo*, *visitor*:  
*graphql.language.visitor.Visitor*)

Bases: *graphql.language.visitor.Visitor*

A visitor which maintains a provided TypeInfo.

**BREAK** = True

**IDLE** = None

**REMOVE** = Ellipsis

**SKIP** = False

**\_\_init\_\_**(*type\_info*: *graphql.utilities.type\_info.TypeInfo*, *visitor*: *graphql.language.visitor.Visitor*)

**enter**(node: [graphql.language.ast.Node](#), \*args: Any) → Any

**get\_visit\_fn**(kind: str, is\_leaving: bool = False) → Callable

Get the visit function for the given node kind and direction.

**leave**(node: [graphql.language.ast.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.GraphQLWrappingType], 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[graphql.language.ast.DocumentNode]) →
    graphql.language.ast.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: graphql.language.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: Union[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 tokens 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: graphql.type.definition.GraphQLType, type_b: graphql.type.definition.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: graphql.type.schema.GraphQLSchema, maybe_subtype: graphql.type.definition.GraphQLType, super_type: graphql.type.definition.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: graphql.type.schema.GraphQLSchema, type_a: Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.GraphQLUnionType], type_b: Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType, graphql.type.definition.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.

Assert that a string is a valid GraphQL name:

```
graphql.utilities.assert_valid_name(name: str) → str
```

Uphold the spec rules about naming.

```
graphql.utilities.is_valid_name_error(name: str) →  
Optional[graphql.error.graphql_error.GraphQLError]
```

Return an Error if a name is invalid.

Compare two GraphQLSchemas and detect breaking changes:

```
graphql.utilities.find_breaking_changes(old_schema: graphql.type.schema.GraphQLSchema,  
                                       new_schema: graphql.type.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: graphql.type.schema.GraphQLSchema,  
                                       new_schema: graphql.type.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, description)
```

Bases: tuple

```
__init__()
```

```
count(value, /)
```

Return number of occurrences of value.

**property description**

Alias for field number 1

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

Return first index of value.

Raises ValueError if the value is not present.

**property type**

Alias for field number 0

```
class graphql.utilities.BreakingChangeType(value)
```

Bases: enum.Enum

An enumeration.

```

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, description)
```

Bases: tuple

```
__init__()
```

```
count(value, /)
```

Return number of occurrences of value.

**property description**

Alias for field number 1

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

Return first index of value.

Raises ValueError if the value is not present.

**property type**

Alias for field number 0

```
class graphql.utilities.DangerousChangeType(value)
```

Bases: enum.Enum

An enumeration.

```
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

Report all deprecated usages within a GraphQL document:

```
graphql.utilities.find_deprecated_usages(schema: graphql.type.schema.GraphQLSchema, ast:
    graphql.language.ast.DocumentNode) →
    List[graphql.error.graphql_error.GraphQLError]
```

Get a list of GraphQLError instances describing each deprecated use.

Deprecated since version 3.1.3.

Please use `validate` with `NoDeprecatedCustomRule` instead:

```
from graphql import validate, NoDeprecatedCustomRule

errors = validate(schema, document, [NoDeprecatedCustomRule])
```

## Validation

### GraphQL Validation

The `graphql.validation` package fulfills the Validation phase of fulfilling a GraphQL result.

```
graphql.validation.validate(schema: graphql.type.schema.GraphQLSchema, document_ast:
    graphql.language.ast.DocumentNode, rules:
    Optional[Collection[Type[graphql.validation.rules.ASTValidationRule]]] =
    None, type_info: Optional[graphql.utilities.type_info.TypeInfo] = None,
    max_errors: Optional[int] = None) →
    List[graphql.error.graphql_error.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.

Optionally a custom `TypeInfo` instance may be provided. If not provided, one will be created from the provided schema.

```
class graphql.validation.ASTValidationContext(ast: graphql.language.ast.DocumentNode, on_error:
    Callable[[graphql.error.graphql_error.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: graphql.language.ast.DocumentNode, on_error:
    Callable[[graphql.error.graphql_error.GraphQLError], None]) → None

document: graphql.language.ast.DocumentNode

get_fragment(name: str) → Optional[graphql.language.ast.FragmentDefinitionNode]

get_fragment_spreads(node: graphql.language.ast.SelectionSetNode) →
    List[graphql.language.ast.FragmentSpreadNode]

get_recursively_referenced_fragments(operation: graphql.language.ast.OperationDefinitionNode) →
    List[graphql.language.ast.FragmentDefinitionNode]

on_error(error: graphql.error.graphql_error.GraphQLError) → None

report_error(error: graphql.error.graphql_error.GraphQLError) → None

class graphql.validation.ASTValidationRule(context:
    graphql.validation.validation_context.ASTValidationContext)

    Bases: graphql.language.visitor.Visitor
    Visitor for validation of an AST.
    BREAK = True
    IDLE = None
    REMOVE = Ellipsis
    SKIP = False

    __init__(context: graphql.validation.validation_context.ASTValidationContext)

    context: graphql.validation.validation_context.ASTValidationContext

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.

    report_error(error: graphql.error.graphql_error.GraphQLError) → None

class graphql.validation.SDLValidationContext(ast: graphql.language.ast.DocumentNode, schema:
    Optional[graphql.type.schema.GraphQLSchema],
    on_error:
    Callable[[graphql.error.graphql_error.GraphQLError],
    None])

    Bases: graphql.validation.validation_context.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: graphql.language.ast.DocumentNode, schema:
        Optional[graphql.type.schema.GraphQLSchema], on_error:
        Callable[[graphql.error.graphql_error.GraphQLError], None]) → None

    document: graphql.language.ast.DocumentNode

    get_fragment(name: str) → Optional[graphql.language.ast.FragmentDefinitionNode]

```

```
get_fragment_spreads(node: graphql.language.ast.SelectionSetNode) →  
    List[graphql.language.ast.FragmentSpreadNode]  
  
get_recursively_referenced_fragments(operation: graphql.language.ast.OperationDefinitionNode) →  
    List[graphql.language.ast.FragmentDefinitionNode]  
  
on_error(error: graphql.error.graphql_error.GraphQLError) → None  
  
report_error(error: graphql.error.graphql_error.GraphQLError) → None  
  
schema: Optional[graphql.type.schema.GraphQLSchema]  
  
class graphql.validation.SDLValidationRule(context:  
    graphql.validation.validation_context.SDLValidationContext)  
  
    Bases: graphql.validation.rules.ASTValidationRule  
    Visitor for validation of an SDL AST.  
  
    BREAK = True  
  
    IDLE = None  
  
    REMOVE = Ellipsis  
  
    SKIP = False  
  
    __init__(context: graphql.validation.validation_context.SDLValidationContext) → None  
  
    context: graphql.validation.validation_context.SDLValidationContext  
  
    get_visit_fn(kind: str, is_leaving: bool = False) → Callable  
        Get the visit function for the given node kind and direction.  
  
    report_error(error: graphql.error.graphql_error.GraphQLError) → None  
  
class graphql.validation.ValidationContext(schema: graphql.type.schema.GraphQLSchema, ast:  
    graphql.language.ast.DocumentNode, type_info:  
    graphql.utilities.type_info.TypeInfo, on_error:  
    Callable[[graphql.error.graphql_error.GraphQLError],  
    None])  
  
    Bases: graphql.validation.validation_context.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: graphql.type.schema.GraphQLSchema, ast: graphql.language.ast.DocumentNode,  
        type_info: graphql.utilities.type_info.TypeInfo, on_error:  
        Callable[[graphql.error.graphql_error.GraphQLError], None]) → None  
  
    document: graphql.language.ast.DocumentNode  
  
    get_argument() → Optional[graphql.type.definition.GraphQLArgument]  
  
    get_directive() → Optional[graphql.type.directives.GraphQLDirective]  
  
    get_enum_value() → Optional[graphql.type.definition.GraphQLEnumValue]
```



```

get_field_def() → Optional[graphql.type.definition.GraphQLField]

get_fragment(name: str) → Optional[graphql.language.ast.FragmentDefinitionNode]

get_fragment_spreads(node: graphql.language.ast.SelectionSetNode) →
    List[graphql.language.ast.FragmentSpreadNode]

get_input_type() → Optional[Union[graphql.type.definition.GraphQLScalarType,
    graphql.type.definition.GraphQLEnumType,
    graphql.type.definition.GraphQLInputObjectType,
    graphql.type.definition.GraphQLWrappingType]]

get_parent_input_type() → Optional[Union[graphql.type.definition.GraphQLScalarType,
    graphql.type.definition.GraphQLEnumType,
    graphql.type.definition.GraphQLInputObjectType,
    graphql.type.definition.GraphQLWrappingType]]

get_parent_type() → Optional[Union[graphql.type.definition.GraphQLObjectType,
    graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType]]

get_recursive_variable_usages(operation: graphql.language.ast.OperationDefinitionNode) →
    List[graphql.validation.validation_context.VariableUsage]

get_recursively_referenced_fragments(operation: graphql.language.ast.OperationDefinitionNode) →
    List[graphql.language.ast.FragmentDefinitionNode]

get_type() → Optional[Union[graphql.type.definition.GraphQLScalarType,
    graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType,
    graphql.type.definition.GraphQLUnionType, graphql.type.definition.GraphQLEnumType,
    graphql.type.definition.GraphQLWrappingType]]

get_variable_usages(node: Union[graphql.language.ast.OperationDefinitionNode,
    graphql.language.ast.FragmentDefinitionNode]) →
    List[graphql.validation.validation_context.VariableUsage]

on_error(error: graphql.error.graphql_error.GraphQLError) → None

report_error(error: graphql.error.graphql_error.GraphQLError) → None

schema: graphql.type.schema.GraphQLSchema

class graphql.validation.ValidationRule(context:
    graphql.validation.validation_context.ValidationContext)
    Bases: graphql.validation.rules.ASTValidationRule
    Visitor for validation using a GraphQL schema.
    BREAK = True
    IDLE = None
    REMOVE = Ellipsis
    SKIP = False
    __init__(context: graphql.validation.validation_context.ValidationContext) → None

```

**context:** `graphql.validation.validation_context.ValidationContext`

**get\_visit\_fn**(*kind: str, is\_leaving: bool = False*) → Callable

Get the visit function for the given node kind and direction.

**report\_error**(*error: graphql.error.graphql\_error.GraphQLError*) → None

## Rules

`graphql.validation.specified_rules = FrozenList([...])`

This list includes all validation rules defined by the GraphQL spec.

The order of the rules in this list has been adjusted to lead to the most clear output when encountering multiple validation errors.

### Spec Section: “Executable Definitions”

**class** `graphql.validation.ExecutableDefinitionsRule`(*context:*  
`graphql.validation.validation_context.ASTValidationContext`)

Bases: `graphql.validation.rules.ASTValidationRule`

Executable definitions

A GraphQL document is only valid for execution if all definitions are either operation or fragment definitions.

**BREAK** = True

**IDLE** = None

**REMOVE** = Ellipsis

**SKIP** = False

**\_\_init\_\_**(*context: graphql.validation.validation\_context.ASTValidationContext*)

**context:** `graphql.validation.validation_context.ASTValidationContext`

**enter\_document**(*node: graphql.language.ast.DocumentNode, \*\_args: Any*) →  
Optional[`graphql.language.visitor.VisitorActionEnum`]

**get\_visit\_fn**(*kind: str, is\_leaving: bool = False*) → Callable

Get the visit function for the given node kind and direction.

**report\_error**(*error: graphql.error.graphql\_error.GraphQLError*) → None

### Spec Section: “Field Selections on Objects, Interfaces, and Unions Types”

**class** `graphql.validation.FieldsOnCorrectTypeRule`(*context:*  
`graphql.validation.validation_context.ValidationContext`)

Bases: `graphql.validation.rules.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`.

**BREAK** = True

**IDLE** = None

**REMOVE = Ellipsis****SKIP = False****\_\_init\_\_**(context: graphql.validation.validation\_context.ValidationContext) → None**context:** *graphql.validation.validation\_context.ValidationContext***enter\_field**(node: graphql.language.ast.FieldNode, \*\_args: Any) → None**get\_visit\_fn**(kind: str, is\_leaving: bool = False) → Callable

Get the visit function for the given node kind and direction.

**report\_error**(error: graphql.error.graphql\_error.GraphQLError) → None**Spec Section: “Fragments on Composite Types”****class** graphql.validation.FragmentsOnCompositeTypesRule(context:  
graphql.validation.validation\_context.ValidationContext)Bases: *graphql.validation.rules.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.

**BREAK = True****IDLE = None****REMOVE = Ellipsis****SKIP = False****\_\_init\_\_**(context: graphql.validation.validation\_context.ValidationContext) → None**context:** *graphql.validation.validation\_context.ValidationContext***enter\_fragment\_definition**(node: graphql.language.ast.FragmentDefinitionNode, \*\_args: Any) → None**enter\_inline\_fragment**(node: graphql.language.ast.InlineFragmentNode, \*\_args: Any) → None**get\_visit\_fn**(kind: str, is\_leaving: bool = False) → Callable

Get the visit function for the given node kind and direction.

**report\_error**(error: graphql.error.graphql\_error.GraphQLError) → None**Spec Section: “Argument Names”****class** graphql.validation.KnownArgumentNamesRule(context:  
graphql.validation.validation\_context.ValidationContext)Bases: *graphql.validation.rules.known\_argument\_names.KnownArgumentNamesOnDirectivesRule*

Known argument names

A GraphQL field is only valid if all supplied arguments are defined by that field.

**BREAK = True****IDLE = None**

```
REMOVE = Ellipsis

SKIP = False

__init__(context: graphql.validation.validation_context.ValidationContext)

context: graphql.validation.validation_context.ValidationContext

enter_argument(arg_node: graphql.language.ast.ArgumentNode, *args: Any) → None

enter_directive(directive_node: graphql.language.ast.DirectiveNode, *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

#### Spec Section: “Directives Are Defined”

```
class graphql.validation.KnownDirectivesRule(context:
    Union[graphql.validation.validation_context.ValidationContext,
    graphql.validation.validation_context.SDLValidationContext])

Bases: graphql.validation.rules.ASTValidationRule

Known directives

A GraphQL document is only valid if all @directives are known by the schema and legally positioned.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(context: Union[graphql.validation.validation_context.ValidationContext,
    graphql.validation.validation_context.SDLValidationContext])

context: Union[graphql.validation.validation_context.ValidationContext,
graphql.validation.validation_context.SDLValidationContext]

enter_directive(node: graphql.language.ast.DirectiveNode, _key: Any, _parent: Any, _path: Any,
    ancestors: List[graphql.language.ast.Node]) → None

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

#### Spec Section: “Fragment spread target defined”

```
class graphql.validation.KnownFragmentNamesRule(context:
    graphql.validation.validation_context.ValidationContext)

Bases: graphql.validation.rules.ValidationRule

Known fragment names

A GraphQL document is only valid if all ...Fragment fragment spreads refer to fragments defined in the same document.
```

```

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(context: graphql.validation.validation_context.ValidationContext) → None

context: graphql.validation.validation_context.ValidationContext

enter_fragment_spread(node: graphql.language.ast.FragmentSpreadNode, *_args: Any) → None

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None

```

#### Spec Section: “Fragment Spread Type Existence”

```

class graphql.validation.KnownTypeNamesRule(context:
    Union[graphql.validation.validation_context.ValidationContext,
    graphql.validation.validation_context.SDLValidationContext])

Bases: graphql.validation.rules.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.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(context: Union[graphql.validation.validation_context.ValidationContext,
    graphql.validation.validation_context.SDLValidationContext])

context: graphql.validation.validation_context.ASTValidationContext

enter_named_type(node: graphql.language.ast.NamedTypeNode, _key: Any, parent:
    graphql.language.ast.Node, _path: Any, ancestors: List[graphql.language.ast.Node]) →
    None

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None

```

#### Spec Section: “Lone Anonymous Operation”

```

class graphql.validation.LoneAnonymousOperationRule(context:
    graphql.validation.validation_context.ASTValidationContext)

Bases: graphql.validation.rules.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.

**BREAK** = True

**IDLE** = None

**REMOVE** = Ellipsis

**SKIP** = False

**\_\_init\_\_**(*context*: graphql.validation.validation\_context.ASTValidationContext)

**context**: *graphql.validation.validation\_context.ASTValidationContext*

**enter\_document**(*node*: graphql.language.ast.DocumentNode, \*\_args: Any) → None

**enter\_operation\_definition**(*node*: graphql.language.ast.OperationDefinitionNode, \*\_args: Any) → None

**get\_visit\_fn**(*kind*: str, *is\_leaving*: bool = False) → Callable

Get the visit function for the given node kind and direction.

**report\_error**(*error*: graphql.error.graphql\_error.GraphQLError) → None

Spec Section: “Fragments must not form cycles”

**class** graphql.validation.NoFragmentCyclesRule(*context*:  
graphql.validation.validation\_context.ASTValidationContext)

Bases: *graphql.validation.rules.ASTValidationRule*

No fragment cycles

**BREAK** = True

**IDLE** = None

**REMOVE** = Ellipsis

**SKIP** = False

**\_\_init\_\_**(*context*: graphql.validation.validation\_context.ASTValidationContext)

**context**: *graphql.validation.validation\_context.ASTValidationContext*

**detect\_cycle\_recursive**(*fragment*: graphql.language.ast.FragmentDefinitionNode) → None

**enter\_fragment\_definition**(*node*: graphql.language.ast.FragmentDefinitionNode, \*\_args: Any) →  
Optional[graphql.language.visitor.VisitorActionEnum]

**static enter\_operation\_definition**(\*\_args: Any) →  
Optional[graphql.language.visitor.VisitorActionEnum]

**get\_visit\_fn**(*kind*: str, *is\_leaving*: bool = False) → Callable

Get the visit function for the given node kind and direction.

**report\_error**(*error*: graphql.error.graphql\_error.GraphQLError) → None

Spec Section: “All Variable Used Defined”

```
class graphql.validation.NoUndefinedVariablesRule(context:
    graphql.validation.validation_context.ValidationContext)

    Bases: graphql.validation.rules.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.

    BREAK = True

    IDLE = None

    REMOVE = Ellipsis

    SKIP = False

    __init__(context: graphql.validation.validation_context.ValidationContext)

    context: graphql.validation.validation_context.ValidationContext

    enter_operation_definition(*_args: Any) → None

    enter_variable_definition(node: graphql.language.ast.VariableDefinitionNode, *_args: Any) → None

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.

    leave_operation_definition(operation: graphql.language.ast.OperationDefinitionNode, *_args: Any)
        → None

    report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

#### Spec Section: “Fragments must be used”

```
class graphql.validation.NoUnusedFragmentsRule(context:
    graphql.validation.validation_context.ASTValidationContext)

    Bases: graphql.validation.rules.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.

    BREAK = True

    IDLE = None

    REMOVE = Ellipsis

    SKIP = False

    __init__(context: graphql.validation.validation_context.ASTValidationContext)

    context: graphql.validation.validation_context.ASTValidationContext

    enter_fragment_definition(node: graphql.language.ast.FragmentDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    enter_operation_definition(node: graphql.language.ast.OperationDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]
```

```
get_visit_fn(kind: str, is_leaving: bool = False) → Callable
```

Get the visit function for the given node kind and direction.

```
leave_document(*_args: Any) → None
```

```
report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

### Spec Section: “All Variables Used”

```
class graphql.validation.NoUnusedVariablesRule(context:
    graphql.validation.validation_context.ValidationContext)
```

Bases: `graphql.validation.rules.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.

**BREAK = True**

**IDLE = None**

REMOVE = Ellipsis

**SKIP = False**

```
__init__(context: graphql.validation.validation_context.ValidationContext)
```

**context:** `graphql.validation.validation_context.ValidationContext`

```
enter_operation_definition(*_args: Any) → None
```

```
enter_variable_definition(definition: graphql.language.ast.VariableDefinitionNode, *_args: Any) → None
```

```
get_visit_fn(kind: str, is_leaving: bool = False) → Callable
```

Get the visit function for the given node kind and direction.

```
leave_operation_definition(operation: graphql.language.ast.OperationDefinitionNode, *_args: Any)
    → None
```

```
report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

### Spec Section: “Field Selection Merging”

```
class graphql.validation.OverlappingFieldsCanBeMergedRule(context:
    graphql.validation.validation_context.ValidationContext)
```

Bases: `graphql.validation.rules.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.

**BREAK = True**

**IDLE = None**

**REMOVE = Ellipsis**

**SKIP = False**



```

__init__(context: graphql.validation.validation_context.ValidationContext)

context:  graphql.validation.validation_context.ValidationContext

enter_selection_set(selection_set: graphql.language.ast.SelectionSetNode, *_args: Any) → None

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None

```

#### Spec Section: “Fragment spread is possible”

```

class graphql.validation.PossibleFragmentSpreadsRule(context:
                                                    graphql.validation.validation_context.ValidationContext)

    Bases:  graphql.validation.rules.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 inter-
    section of the possible parent types, and possible types which pass the type condition.

    BREAK = True

    IDLE = None

    REMOVE = Ellipsis

    SKIP = False

    __init__(context: graphql.validation.validation_context.ValidationContext) → None

    context:  graphql.validation.validation_context.ValidationContext

    enter_fragment_spread(node: graphql.language.ast.FragmentSpreadNode, *_args: Any) → None

    enter_inline_fragment(node: graphql.language.ast.InlineFragmentNode, *_args: Any) → None

    get_fragment_type(name: str) → Optional[Union[graphql.type.definition.GraphQLOBJECTType,
                                                  graphql.type.definition.GraphQLInterfaceType,
                                                  graphql.type.definition.GraphQLUnionType]]

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.

    report_error(error: graphql.error.graphql_error.GraphQLError) → None

```

#### Spec Section: “Argument Optionality”

```

class graphql.validation.ProvidedRequiredArgumentsRule(context:
                                                       graphql.validation.validation_context.ValidationContext)

    Bases:  graphql.validation.rules.provided_required_arguments.
            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: graphql.validation.validation_context.ValidationContext)

context: graphql.validation.validation_context.ValidationContext

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

leave_directive(directive_node: graphql.language.ast.DirectiveNode, *_args: Any) → None

leave_field(field_node: graphql.language.ast.FieldNode, *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

report_error(error: graphql.error.graphql_error.GraphQLError) → None

```

### Spec Section: “Leaf Field Selections”

```

class graphql.validation.ScalarLeafsRule(context:
                                graphql.validation.validation_context.ValidationContext)

Bases: graphql.validation.rules.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: graphql.validation.validation_context.ValidationContext) → None

context: graphql.validation.validation_context.ValidationContext

enter_field(node: graphql.language.ast.FieldNode, *_args: Any) → None

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None

```

**Spec Section: “Subscriptions with Single Root Field”**

```
class graphql.validation.SingleFieldSubscriptionsRule(context:
                                                    graphql.validation.validation_context.ASTValidationContext)

Bases: graphql.validation.rules.ASTValidationRule

Subscriptions must only include one field.

A GraphQL subscription is valid only if it contains a single root.

BREAK = True
```

IDLE = None

REMOVE = Ellipsis

SKIP = False

`__init__(context: graphql.validation.validation_context.ASTValidationContext)`

**context:** `graphql.validation.validation_context.ASTValidationContext`

**enter\_operation\_definition**(node: graphql.language.ast.OperationDefinitionNode, \*\_args: Any) → None

**get\_visit\_fn**(kind: str, is\_leaving: bool = False) → Callable  
Get the visit function for the given node kind and direction.

**report\_error**(error: graphql.error.graphql\_error.GraphQLError) → None

#### Spec Section: “Argument Uniqueness”

**class** `graphql.validation.UniqueArgumentNamesRule`(context: `graphql.validation.validation_context.ASTValidationContext`)

Bases: `graphql.validation.rules.ASTValidationRule`

Unique argument names

A GraphQL field or directive is only valid if all supplied arguments are uniquely named.

BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

`__init__(context: graphql.validation.validation_context.ASTValidationContext)`

**context:** `graphql.validation.validation_context.ASTValidationContext`

**enter\_argument**(node: graphql.language.ast.ArgumentNode, \*\_args: Any) → Optional[`graphql.language.visitor.VisitorActionEnum`]

**enter\_directive**(\*\_args: Any) → None

**enter\_field**(\*\_args: Any) → None

**get\_visit\_fn**(kind: str, is\_leaving: bool = False) → Callable  
Get the visit function for the given node kind and direction.

**report\_error**(error: graphql.error.graphql\_error.GraphQLError) → None

#### Spec Section: “Directives Are Unique Per Location”

**class** `graphql.validation.UniqueDirectivesPerLocationRule`(context: `Union[graphql.validation.validation_context.ValidationContext, graphql.validation.validation_context.SDLValidationContext]`)

Bases: `graphql.validation.rules.ASTValidationRule`

Unique directive names per location

A GraphQL document is only valid if all non-repeatable directives at a given location are uniquely named.

```
BREAK = True

IDLE = None

REMOVE = Ellipsis

SKIP = False

__init__(context: Union[graphql.validation.validation_context.ValidationContext,
                        graphql.validation.validation_context.SDLValidationContext])

context: Union[graphql.validation.validation_context.ValidationContext,
               graphql.validation.validation_context.SDLValidationContext]

enter(node: graphql.language.ast.Node, *_args: Any) → None

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

#### Spec Section: “Fragment Name Uniqueness”

```
class graphql.validation.UniqueFragmentNamesRule(context:
                                                graphql.validation.validation_context.ASTValidationContext)

    Bases: graphql.validation.rules.ASTValidationRule

    Unique fragment names

    A GraphQL document is only valid if all defined fragments have unique names.

    BREAK = True

    IDLE = None

    REMOVE = Ellipsis

    SKIP = False

    __init__(context: graphql.validation.validation_context.ASTValidationContext)

    context: graphql.validation.validation_context.ASTValidationContext

    enter_fragment_definition(node: graphql.language.ast.FragmentDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    static enter_operation_definition(*_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.

    report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

#### Spec Section: “Input Object Field Uniqueness”

```
class graphql.validation.UniqueInputFieldNamesRule(context:
    graphql.validation.validation_context.ASTValidationContext)

    Bases: graphql.validation.rules.ASTValidationRule

    Unique input field names

    A GraphQL input object value is only valid if all supplied fields are uniquely named.

    BREAK = True

    IDLE = None

    REMOVE = Ellipsis

    SKIP = False

    __init__(context: graphql.validation.validation_context.ASTValidationContext)

    context: graphql.validation.validation_context.ASTValidationContext

    enter_object_field(node: graphql.language.ast.ObjectFieldNode, *_args: Any) → None

    enter_object_value(*_args: Any) → None

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.

    leave_object_value(*_args: Any) → None

    report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

#### Spec Section: “Operation Name Uniqueness”

```
class graphql.validation.UniqueOperationNamesRule(context:
    graphql.validation.validation_context.ASTValidationContext)

    Bases: graphql.validation.rules.ASTValidationRule

    Unique operation names

    A GraphQL document is only valid if all defined operations have unique names.

    BREAK = True

    IDLE = None

    REMOVE = Ellipsis

    SKIP = False

    __init__(context: graphql.validation.validation_context.ASTValidationContext)

    context: graphql.validation.validation_context.ASTValidationContext

    static enter_fragment_definition(*_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    enter_operation_definition(node: graphql.language.ast.OperationDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.
```

**report\_error**(*error*: graphql.error.graphql\_error.GraphQLError) → None

**Spec Section: “Variable Uniqueness”**

```
class graphql.validation.UniqueVariableNamesRule(context:
                                                    graphql.validation.validation_context.ASTValidationContext)

    Bases: graphql.validation.rules.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: graphql.validation.validation_context.ASTValidationContext)

    context: graphql.validation.validation_context.ASTValidationContext

    enter_operation_definition(*_args: Any) → None

    enter_variable_definition(node: graphql.language.ast.VariableDefinitionNode, *_args: Any) → None

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.

    report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

**Spec Section: “Value Type Correctness”**

```
class graphql.validation.ValuesOfCorrectTypeRule(context:
                                                    graphql.validation.validation_context.ValidationContext)

    Bases: graphql.validation.rules.ValidationRule

    Value literals of correct type

    A GraphQL document is only valid if all value literals are of the type expected at their position.

    BREAK = True

    IDLE = None

    REMOVE = Ellipsis

    SKIP = False

    __init__(context: graphql.validation.validation_context.ValidationContext) → None

    context: graphql.validation.validation_context.ValidationContext

    enter_boolean_value(node: graphql.language.ast.BooleanValueNode, *_args: Any) → None

    enter_enum_value(node: graphql.language.ast.EnumValueNode, *_args: Any) → None

    enter_float_value(node: graphql.language.ast.FloatValueNode, *_args: Any) → None

    enter_int_value(node: graphql.language.ast.IntValueNode, *_args: Any) → None
```

**enter\_list\_value**(node: graphql.language.ast.ListValueNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**enter\_null\_value**(node: graphql.language.ast.NullValueNode, \*\_args: Any) → None

**enter\_object\_field**(node: graphql.language.ast.ObjectFieldNode, \*\_args: Any) → None

**enter\_object\_value**(node: graphql.language.ast.ObjectValueNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**enter\_string\_value**(node: graphql.language.ast.StringValueNode, \*\_args: Any) → None

**get\_visit\_fn**(kind: str, is\_leaving: bool = False) → Callable  
Get the visit function for the given node kind and direction.

**is\_valid\_value\_node**(node: graphql.language.ast.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: graphql.error.graphql\_error.GraphQLError) → None

#### Spec Section: “Variables are Input Types”

**class** graphql.validation.VariablesAreInputTypesRule(context: graphql.validation.validation\_context.ValidationContext)

Bases: [graphql.validation.rules.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).

**BREAK** = True

**IDLE** = None

**REMOVE** = Ellipsis

**SKIP** = False

**\_\_init\_\_**(context: graphql.validation.validation\_context.ValidationContext) → None

**context**: [graphql.validation.validation\\_context.ValidationContext](#)

**enter\_variable\_definition**(node: graphql.language.ast.VariableDefinitionNode, \*\_args: Any) → None

**get\_visit\_fn**(kind: str, is\_leaving: bool = False) → Callable  
Get the visit function for the given node kind and direction.

**report\_error**(error: graphql.error.graphql\_error.GraphQLError) → None

#### Spec Section: “All Variable Usages Are Allowed”

**class** graphql.validation.VariablesInAllowedPositionRule(context: graphql.validation.validation\_context.ValidationContext)

Bases: [graphql.validation.rules.ValidationRule](#)

Variables passed to field arguments conform to type

**BREAK** = True

```
IDLE = None
REMOVE = Ellipsis
SKIP = False

__init__(context: graphql.validation.validation_context.ValidationContext)
context: graphql.validation.validation_context.ValidationContext

enter_operation_definition(*_args: Any) → None
enter_variable_definition(node: graphql.language.ast.VariableDefinitionNode, *_args: Any) → None
get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.
leave_operation_definition(operation: graphql.language.ast.OperationDefinitionNode, *_args: Any)
    → None

report_error(error: graphql.error.graphql_error.GraphQLError) → None
```

#### SDL-specific validation rules

```
class graphql.validation.LoneSchemaDefinitionRule(context:
                                                    graphql.validation.validation_context.SDLValidationContext)
    Bases: graphql.validation.rules.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: graphql.validation.validation_context.SDLValidationContext)
    context: graphql.validation.validation_context.SDLValidationContext

    enter_schema_definition(node: graphql.language.ast.SchemaDefinitionNode, *_args: Any) → None
    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.
    report_error(error: graphql.error.graphql_error.GraphQLError) → None

class graphql.validation.UniqueOperationTypesRule(context:
                                                    graphql.validation.validation_context.SDLValidationContext)
    Bases: graphql.validation.rules.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: graphql.validation.validation_context.SDLValidationContext)

check_operation_types(node: Union[graphql.language.ast.SchemaDefinitionNode,
                                  graphql.language.ast.SchemaExtensionNode], *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

context: graphql.validation.validation_context.SDLValidationContext

enter_schema_definition(node: Union[graphql.language.ast.SchemaDefinitionNode,
                                    graphql.language.ast.SchemaExtensionNode], *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

enter_schema_extension(node: Union[graphql.language.ast.SchemaDefinitionNode,
                                   graphql.language.ast.SchemaExtensionNode], *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None

class graphql.validation.UniqueTypeNamesRule(context:
                                             graphql.validation.validation_context.SDLValidationContext)

    Bases: graphql.validation.rules.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: graphql.validation.validation_context.SDLValidationContext)

    check_type_name(node: graphql.language.ast.TypeDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    context: graphql.validation.validation_context.SDLValidationContext

    enter_enum_type_definition(node: graphql.language.ast.TypeDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    enter_input_object_type_definition(node: graphql.language.ast.TypeDefinitionNode, *_args: Any)
        → Optional[graphql.language.visitor.VisitorActionEnum]

    enter_interface_type_definition(node: graphql.language.ast.TypeDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

```

**enter\_object\_type\_definition**(*node*: graphql.language.ast.TypeDefinitionNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**enter\_scalar\_type\_definition**(*node*: graphql.language.ast.TypeDefinitionNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**enter\_union\_type\_definition**(*node*: graphql.language.ast.TypeDefinitionNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**get\_visit\_fn**(*kind*: str, *is\_leaving*: bool = False) → Callable  
Get the visit function for the given node kind and direction.

**report\_error**(*error*: graphql.error.graphql\_error.GraphQLError) → None

**class** graphql.validation.UniqueEnumValueNamesRule(*context*:  
graphql.validation.validation\_context.SDLValidationContext)

Bases: [graphql.validation.rules.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*: graphql.validation.validation\_context.SDLValidationContext)

**check\_value\_uniqueness**(*node*: graphql.language.ast.EnumTypeDefinitionNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**context**: [graphql.validation.validation\\_context.SDLValidationContext](#)

**enter\_enum\_type\_definition**(*node*: graphql.language.ast.EnumTypeDefinitionNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**enter\_enum\_type\_extension**(*node*: graphql.language.ast.EnumTypeDefinitionNode, \*\_args: Any) → Optional[graphql.language.visitor.VisitorActionEnum]

**get\_visit\_fn**(*kind*: str, *is\_leaving*: bool = False) → Callable  
Get the visit function for the given node kind and direction.

**report\_error**(*error*: graphql.error.graphql\_error.GraphQLError) → None

**class** graphql.validation.UniqueFieldDefinitionNamesRule(*context*:  
graphql.validation.validation\_context.SDLValidationContext)

Bases: [graphql.validation.rules.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: graphql.validation.validation_context.SDLValidationContext)

check_field_uniqueness(node: graphql.language.ast.ObjectTypeDefinitionNode, *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

context: graphql.validation.validation_context.SDLValidationContext

enter_input_object_type_definition(node: graphql.language.ast.ObjectTypeDefinitionNode, *_args:
    Any) → Optional[graphql.language.visitor.VisitorActionEnum]

enter_input_object_type_extension(node: graphql.language.ast.ObjectTypeDefinitionNode, *_args:
    Any) → Optional[graphql.language.visitor.VisitorActionEnum]

enter_interface_type_definition(node: graphql.language.ast.ObjectTypeDefinitionNode, *_args:
    Any) → Optional[graphql.language.visitor.VisitorActionEnum]

enter_interface_type_extension(node: graphql.language.ast.ObjectTypeDefinitionNode, *_args: Any)
    → Optional[graphql.language.visitor.VisitorActionEnum]

enter_object_type_definition(node: graphql.language.ast.ObjectTypeDefinitionNode, *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

enter_object_type_extension(node: graphql.language.ast.ObjectTypeDefinitionNode, *_args: Any) →
    Optional[graphql.language.visitor.VisitorActionEnum]

get_visit_fn(kind: str, is_leaving: bool = False) → Callable
    Get the visit function for the given node kind and direction.

report_error(error: graphql.error.graphql_error.GraphQLError) → None

class graphql.validation.UniqueDirectiveNamesRule(context:
    graphql.validation.validation_context.SDLValidationContext)

    Bases: graphql.validation.rules.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: graphql.validation.validation_context.SDLValidationContext)

    context: graphql.validation.validation_context.SDLValidationContext

    enter_directive_definition(node: graphql.language.ast.DirectiveDefinitionNode, *_args: Any) →
        Optional[graphql.language.visitor.VisitorActionEnum]

    get_visit_fn(kind: str, is_leaving: bool = False) → Callable
        Get the visit function for the given node kind and direction.

```

**report\_error**(*error*: graphql.error.graphql\_error.GraphQLError) → None

**class** graphql.validation.**PossibleTypeExtensionsRule**(*context*:  
graphql.validation.validation\_context.SDLValidationContext)

Bases: *graphql.validation.rules.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*: graphql.validation.validation\_context.SDLValidationContext)

**check\_extension**(*node*: graphql.language.ast.TypeExtensionNode, \*\_args: Any) → None

**context**: *graphql.validation.validation\_context.SDLValidationContext*

**enter\_enum\_type\_extension**(*node*: graphql.language.ast.TypeExtensionNode, \*\_args: Any) → None

**enter\_input\_object\_type\_extension**(*node*: graphql.language.ast.TypeExtensionNode, \*\_args: Any) →  
None

**enter\_interface\_type\_extension**(*node*: graphql.language.ast.TypeExtensionNode, \*\_args: Any) →  
None

**enter\_object\_type\_extension**(*node*: graphql.language.ast.TypeExtensionNode, \*\_args: Any) → None

**enter\_scalar\_type\_extension**(*node*: graphql.language.ast.TypeExtensionNode, \*\_args: Any) → None

**enter\_union\_type\_extension**(*node*: graphql.language.ast.TypeExtensionNode, \*\_args: Any) → None

**get\_visit\_fn**(*kind*: str, *is\_leaving*: bool = False) → Callable

Get the visit function for the given node kind and direction.

**report\_error**(*error*: graphql.error.graphql\_error.GraphQLError) → None

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### g

- `graphql`, 19
- `graphql.error`, 21
- `graphql.execution`, 23
- `graphql.language`, 30
- `graphql.pyutils`, 54
- `graphql.subscription`, 57
- `graphql.type`, 58
- `graphql.utilities`, 80
- `graphql.validation`, 90
- `graphql.validation.rules`, 94





## Symbols

- `__init__()` (*graphql.execution.ExecutionContext method*), 24
- `__init__()` (*graphql.execution.ExecutionResult method*), 28
- `__init__()` (*graphql.execution.MiddlewareManager method*), 28
- `__init__()` (*graphql.language.ArgumentNode method*), 31
- `__init__()` (*graphql.language.BooleanValueNode method*), 31
- `__init__()` (*graphql.language.DefinitionNode method*), 31
- `__init__()` (*graphql.language.DirectiveDefinitionNode method*), 31
- `__init__()` (*graphql.language.DirectiveNode method*), 32
- `__init__()` (*graphql.language.DocumentNode method*), 32
- `__init__()` (*graphql.language.EnumTypeDefinitionNode method*), 32
- `__init__()` (*graphql.language.EnumTypeExtensionNode method*), 33
- `__init__()` (*graphql.language.EnumValueDefinitionNode method*), 33
- `__init__()` (*graphql.language.EnumValueNode method*), 33
- `__init__()` (*graphql.language.ExecutableDefinitionNode method*), 34
- `__init__()` (*graphql.language.FieldDefinitionNode method*), 34
- `__init__()` (*graphql.language.FieldNode method*), 34
- `__init__()` (*graphql.language.FloatValueNode method*), 35
- `__init__()` (*graphql.language.FragmentDefinitionNode method*), 35
- `__init__()` (*graphql.language.FragmentSpreadNode method*), 35
- `__init__()` (*graphql.language.InlineFragmentNode method*), 36
- `__init__()` (*graphql.language.InputObjectTypeDefinitionNode method*), 36
- `__init__()` (*graphql.language.InputObjectTypeExtensionNode method*), 36
- `__init__()` (*graphql.language.InputValueDefinitionNode method*), 37
- `__init__()` (*graphql.language.IntValueNode method*), 37
- `__init__()` (*graphql.language.InterfaceTypeDefinitionNode method*), 37
- `__init__()` (*graphql.language.InterfaceTypeExtensionNode method*), 38
- `__init__()` (*graphql.language.Lexer method*), 48
- `__init__()` (*graphql.language.ListTypeNode method*), 38
- `__init__()` (*graphql.language.ListValueNode method*), 38
- `__init__()` (*graphql.language.Location method*), 30
- `__init__()` (*graphql.language.NameNode method*), 39
- `__init__()` (*graphql.language.NamedTypeNode method*), 39
- `__init__()` (*graphql.language.Node method*), 30
- `__init__()` (*graphql.language.NonNullTypeNode method*), 39
- `__init__()` (*graphql.language.NullValueNode method*), 39
- `__init__()` (*graphql.language.ObjectFieldNode method*), 40
- `__init__()` (*graphql.language.ObjectTypeDefinitionNode method*), 40
- `__init__()` (*graphql.language.ObjectTypeExtensionNode method*), 40
- `__init__()` (*graphql.language.ObjectValueNode method*), 41
- `__init__()` (*graphql.language.OperationDefinitionNode method*), 41
- `__init__()` (*graphql.language.OperationTypeDefinitionNode method*), 41
- `__init__()` (*graphql.language.ParallelVisitor method*), 53
- `__init__()` (*graphql.language.ScalarTypeDefinitionNode method*), 42
- `__init__()` (*graphql.language.ScalarTypeExtensionNode method*), 42

```

__init__() (graphql.language.SchemaDefinitionNode
method), 42
__init__() (graphql.language.SchemaExtensionNode
method), 43
__init__() (graphql.language.SelectionNode method),
43
__init__() (graphql.language.SelectionSetNode
method), 43
__init__() (graphql.language.Source method), 51
__init__() (graphql.language.SourceLocation
method), 50
__init__() (graphql.language.StringValueNode
method), 43
__init__() (graphql.language.Token method), 49
__init__() (graphql.language.TypeDefinitionNode
method), 44
__init__() (graphql.language.TypeExtensionNode
method), 44
__init__() (graphql.language.TypeNode method), 44
__init__() (graphql.language.TypeSystemDefinitionNode
method), 45
__init__() (graphql.language.UnionTypeDefinitionNode
method), 45
__init__() (graphql.language.UnionTypeExtensionNode
method), 45
__init__() (graphql.language.ValueNode method), 46
__init__() (graphql.language.VariableDefinitionNode
method), 46
__init__() (graphql.language.VariableNode method),
46
__init__() (graphql.language.Visitor method), 53
__init__() (graphql.pyutils.Path method), 56
__init__() (graphql.pyutils.SimplePubSub method), 55
__init__() (graphql.pyutils.SimplePubSubIterator
method), 55
__init__() (graphql.subscription.MapAsyncIterator
method), 58
__init__() (graphql.type.GraphQLArgument method),
69
__init__() (graphql.type.GraphQLDirective method),
75
__init__() (graphql.type.GraphQLEnumType method),
61
__init__() (graphql.type.GraphQLEnumValue
method), 70
__init__() (graphql.type.GraphQLField method), 70
__init__() (graphql.type.GraphQLInputField method),
71
__init__() (graphql.type.GraphQLInputObjectType
method), 62
__init__() (graphql.type.GraphQLInterfaceType
method), 63
__init__() (graphql.type.GraphQLList method), 68
__init__() (graphql.type.GraphQLNamedType
method), 72
__init__() (graphql.type.GraphQLNonNull method),
69
__init__() (graphql.type.GraphQLObjectType
method), 65
__init__() (graphql.type.GraphQLResolveInfo
method), 73
__init__() (graphql.type.GraphQLScalarType
method), 66
__init__() (graphql.type.GraphQLSchema method), 78
__init__() (graphql.type.GraphQLType method), 73
__init__() (graphql.type.GraphQLUnionType method),
67
__init__() (graphql.type.GraphQLWrappingType
method), 73
__init__() (graphql.utilities.BreakingChange method),
88
__init__() (graphql.utilities.DangerousChange
method), 89
__init__() (graphql.utilities.TypeInfo method), 84
__init__() (graphql.utilities.TypeInfoVisitor method),
85
__init__() (graphql.validation.ASTValidationContext
method), 90
__init__() (graphql.validation.ASTValidationRule
method), 91
__init__() (graphql.validation.ExecutableDefinitionsRule
method), 94
__init__() (graphql.validation.FieldsOnCorrectTypeRule
method), 95
__init__() (graphql.validation.FragmentsOnCompositeTypesRule
method), 95
__init__() (graphql.validation.KnownArgumentNamesRule
method), 96
__init__() (graphql.validation.KnownDirectivesRule
method), 96
__init__() (graphql.validation.KnownFragmentNamesRule
method), 97
__init__() (graphql.validation.KnownTypeNamesRule
method), 97
__init__() (graphql.validation.LoneAnonymousOperationRule
method), 98
__init__() (graphql.validation.LoneSchemaDefinitionRule
method), 108
__init__() (graphql.validation.NoFragmentCyclesRule
method), 98
__init__() (graphql.validation.NoUndefinedVariablesRule
method), 99
__init__() (graphql.validation.NoUnusedFragmentsRule
method), 99
__init__() (graphql.validation.NoUnusedVariablesRule
method), 100
__init__() (graphql.validation.OverlappingFieldsCanBeMergedRule
method), 100

```

[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.PossibleFragmentSpreadsRule* attribute), 34  
[method](#)), 101  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.PossibleTypeExtensionsRule* attribute), 34  
[method](#)), 112  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.ProvidedRequiredArgumentsRule* attribute), 34  
[method](#)), 102  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.SDLValidationContext* attribute), 34  
[method](#)), 91  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.SDLValidationRule* attribute), 34  
[method](#)), 92  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.ScalarLeafsRule* attribute), 34  
[method](#)), 102  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.SingleFieldSubscriptionsRule* attribute), 34  
[method](#)), 103  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueArgumentNamesRule* attribute), 34  
[method](#)), 103  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueDirectiveNamesRule* attribute), 34  
[method](#)), 111  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueDirectivesPerLocationRule* attribute), 34  
[method](#)), 104  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueEnumValueNamesRule* attribute), 34  
[method](#)), 110  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueFieldDefinitionNamesRule* attribute), 34  
[method](#)), 111  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueFragmentNamesRule* attribute), 34  
[method](#)), 104  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueInputFieldNamesRule* attribute), 34  
[method](#)), 105  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueOperationNamesRule* attribute), 34  
[method](#)), 105  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueOperationTypesRule* attribute), 34  
[method](#)), 109  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueTypeNamesRule* attribute), 34  
[method](#)), 109  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.UniqueVariableNamesRule* attribute), 34  
[method](#)), 106  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.ValidationContext* attribute), 34  
[method](#)), 92  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.ValidationRule* attribute), 34  
[method](#)), 93  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.ValuesOfCorrectTypeRule* attribute), 34  
[method](#)), 106  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.VariablesAreInputTypesRule* attribute), 34  
[method](#)), 107  
[\\_\\_init\\_\\_\(\)](#) (*graphql.validation.VariablesInAllowedPositionsRule* attribute), 34  
[method](#)), 108

**A**  
[aclose\(\)](#) (*graphql.pyutils.SimplePubSubIterator* attribute), 55  
[method](#)), 55  
[aclose\(\)](#) (*graphql.subscription.MapAsyncIterator* attribute), 55  
[method](#)), 58  
[add\\_key\(\)](#) (*graphql.pyutils.Path* attribute), 56  
[method](#)), 56  
[advance\(\)](#) (*graphql.language.Lexer* attribute), 48  
[method](#)), 48  
[alias\(\)](#) (*graphql.language.FieldNode* attribute), 34  
[method](#)), 34  
[AMP](#) (*graphql.language.TokenKind* attribute), 48  
[ARG\\_CHANGED\\_KIND](#) (*graphql.utilities.BreakingChangeType* attribute), 88  
[ARG\\_DEFAULT\\_VALUE\\_CHANGE](#) (*graphql.utilities.DangerousChangeType* attribute), 89  
[ARG\\_REMOVED](#) (*graphql.utilities.BreakingChangeType* attribute), 89  
[args](#) (*graphql.error.GraphQLError* attribute), 21  
[args](#) (*graphql.error.GraphQLError* attribute), 22  
[args](#) (*graphql.type.GraphQLDirective* attribute), 75  
[args](#) (*graphql.type.GraphQLField* attribute), 70  
[ARGUMENT\\_DEFINITION](#) (*graphql.language.DirectiveLocation* attribute), 46  
[ArgumentNode](#) (class in *graphql.language*), 31  
[arguments](#) (*graphql.language.DirectiveDefinitionNode* attribute), 31  
[arguments](#) (*graphql.language.DirectiveNode* attribute), 32  
[arguments](#) (*graphql.language.FieldDefinitionNode* attribute), 34  
[arguments](#) (*graphql.language.FieldNode* attribute), 34  
[as\\_list\(\)](#) (*graphql.pyutils.Path* attribute), 56  
[assert\\_abstract\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_composite\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_enum\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_input\\_object\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_input\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_interface\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_leaf\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_list\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_named\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_non\\_null\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_nullable\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_object\\_type\(\)](#) (in module *graphql.type*), 59  
[assert\\_output\\_type\(\)](#) (in module *graphql.type*), 60  
[assert\\_scalar\\_type\(\)](#) (in module *graphql.type*), 60  
[assert\\_type\(\)](#) (in module *graphql.type*), 60  
[assert\\_union\\_type\(\)](#) (in module *graphql.type*), 60  
[assert\\_valid\\_name\(\)](#) (in module *graphql.utilities*), 88  
[assert\\_valid\\_schema\(\)](#) (in module *graphql.type*), 80  
[assert\\_wrapping\\_type\(\)](#) (in module *graphql.type*), 60  
[ast\\_from\\_value\(\)](#) (in module *graphql.utilities*), 83  
[ast\\_node](#) (*graphql.type.GraphQLArgument* attribute), 69  
[ast\\_node](#) (*graphql.type.GraphQLDirective* attribute), 75  
[ast\\_node](#) (*graphql.type.GraphQLEnumType* attribute), 61

`ast_node` (`graphql.type.GraphQLEnumValue` attribute), 70

`ast_node` (`graphql.type.GraphQLField` attribute), 71

`ast_node` (`graphql.type.GraphQLInputField` attribute), 71

`ast_node` (`graphql.type.GraphQLInputObjectType` attribute), 62

`ast_node` (`graphql.type.GraphQLInterfaceType` attribute), 64

`ast_node` (`graphql.type.GraphQLNamedType` attribute), 72

`ast_node` (`graphql.type.GraphQLObjectType` attribute), 65

`ast_node` (`graphql.type.GraphQLScalarType` attribute), 66

`ast_node` (`graphql.type.GraphQLSchema` attribute), 78

`ast_node` (`graphql.type.GraphQLUnionType` attribute), 67

`ASTValidationContext` (class in `graphql.validation`), 90

`ASTValidationRule` (class in `graphql.validation`), 91

`AT` (`graphql.language.TokenKind` attribute), 48

`athrow()` (`graphql.subscription.MapAsyncIterator` method), 58

`AwaitableOrValue` (in module `graphql.pyutils`), 56

## B

`BANG` (`graphql.language.TokenKind` attribute), 48

`block` (`graphql.language.StringValueNode` attribute), 44

`BLOCK_STRING` (`graphql.language.TokenKind` attribute), 49

`body` (`graphql.language.Source` attribute), 52

`BooleanValueNode` (class in `graphql.language`), 31

`BRACE_L` (`graphql.language.TokenKind` attribute), 49

`BRACE_R` (`graphql.language.TokenKind` attribute), 49

`BRACKET_L` (`graphql.language.TokenKind` attribute), 49

`BRACKET_R` (`graphql.language.TokenKind` attribute), 49

`BREAK` (`graphql.language.ParallelVisitor` attribute), 53

`BREAK` (`graphql.language.Visitor` attribute), 53

`BREAK` (`graphql.language.visitor.VisitorActionEnum` attribute), 54

`BREAK` (`graphql.utilities.TypeInfoVisitor` attribute), 85

`BREAK` (`graphql.validation.ASTValidationRule` attribute), 91

`BREAK` (`graphql.validation.ExecutableDefinitionsRule` attribute), 94

`BREAK` (`graphql.validation.FieldsOnCorrectTypeRule` attribute), 94

`BREAK` (`graphql.validation.FragmentsOnCompositeTypesRule` attribute), 95

`BREAK` (`graphql.validation.KnownArgumentNamesRule` attribute), 95

`BREAK` (`graphql.validation.KnownDirectivesRule` attribute), 96

`BREAK` (`graphql.validation.KnownFragmentNamesRule` attribute), 96

`BREAK` (`graphql.validation.KnownTypeNamesRule` attribute), 97

`BREAK` (`graphql.validation.LoneAnonymousOperationRule` attribute), 98

`BREAK` (`graphql.validation.LoneSchemaDefinitionRule` attribute), 108

`BREAK` (`graphql.validation.NoFragmentCyclesRule` attribute), 98

`BREAK` (`graphql.validation.NoUndefinedVariablesRule` attribute), 99

`BREAK` (`graphql.validation.NoUnusedFragmentsRule` attribute), 99

`BREAK` (`graphql.validation.NoUnusedVariablesRule` attribute), 100

`BREAK` (`graphql.validation.OverlappingFieldsCanBeMergedRule` attribute), 100

`BREAK` (`graphql.validation.PossibleFragmentSpreadsRule` attribute), 101

`BREAK` (`graphql.validation.PossibleTypeExtensionsRule` attribute), 112

`BREAK` (`graphql.validation.ProvidedRequiredArgumentsRule` attribute), 101

`BREAK` (`graphql.validation.ScalarLeafsRule` attribute), 102

`BREAK` (`graphql.validation.SDLValidationRule` attribute), 92

`BREAK` (`graphql.validation.SingleFieldSubscriptionsRule` attribute), 102

`BREAK` (`graphql.validation.UniqueArgumentNamesRule` attribute), 103

`BREAK` (`graphql.validation.UniqueDirectiveNamesRule` attribute), 111

`BREAK` (`graphql.validation.UniqueDirectivesPerLocationRule` attribute), 103

`BREAK` (`graphql.validation.UniqueEnumValueNamesRule` attribute), 110

`BREAK` (`graphql.validation.UniqueFieldDefinitionNamesRule` attribute), 110

`BREAK` (`graphql.validation.UniqueFragmentNamesRule` attribute), 104

`BREAK` (`graphql.validation.UniqueInputFieldNamesRule` attribute), 105

`BREAK` (`graphql.validation.UniqueOperationNamesRule` attribute), 105

`BREAK` (`graphql.validation.UniqueOperationTypesRule` attribute), 108

`BREAK` (`graphql.validation.UniqueTypeNamesRule` attribute), 109

`BREAK` (`graphql.validation.UniqueVariableNamesRule` attribute), 106

`BREAK` (`graphql.validation.ValidationRule` attribute), 93

`BREAK` (`graphql.validation.ValuesOfCorrectTypeRule` attribute), 93



- tribute*), 106
- BREAK (*graphql.validation.VariablesAreInputTypesRule* attribute), 107
- BREAK (*graphql.validation.VariablesInAllowedPositionRule* attribute), 107
- BREAK (in module *graphql.language*), 54
- BreakingChange (class in *graphql.utilities*), 88
- BreakingChangeType (class in *graphql.utilities*), 88
- build() (*graphql.execution.ExecutionContext* class method), 24
- build\_ast\_schema() (in module *graphql.utilities*), 81
- build\_client\_schema() (in module *graphql.utilities*), 81
- build\_resolve\_info() (*graphql.execution.ExecutionContext* method), 24
- build\_response() (*graphql.execution.ExecutionContext* method), 24
- build\_schema() (in module *graphql.utilities*), 81
- C**
- cached\_property() (in module *graphql.pyutils*), 54
- camel\_to\_snake() (in module *graphql.pyutils*), 54
- check\_extension() (*graphql.validation.PossibleTypeExtensionsRule* method), 112
- check\_field\_uniqueness() (*graphql.validation.UniqueFieldDefinitionNamesRule* method), 111
- check\_operation\_types() (*graphql.validation.UniqueOperationTypesRule* method), 109
- check\_type\_name() (*graphql.validation.UniqueTypeNamesRule* method), 109
- check\_value\_uniqueness() (*graphql.validation.UniqueEnumValueNamesRule* method), 110
- coerce\_input\_value() (in module *graphql.utilities*), 86
- collect\_and\_execute\_subfields() (*graphql.execution.ExecutionContext* method), 24
- collect\_fields() (*graphql.execution.ExecutionContext* method), 25
- collect\_subfields() (*graphql.execution.ExecutionContext* method), 25
- COLON (*graphql.language.TokenKind* attribute), 49
- column (*graphql.language.SourceLocation* property), 50
- column (*graphql.language.Token* attribute), 49
- COMMENT (*graphql.language.TokenKind* attribute), 49
- complete\_abstract\_value() (*graphql.execution.ExecutionContext* method), 25
- complete\_leaf\_value() (*graphql.execution.ExecutionContext* static method), 25
- complete\_list\_value() (*graphql.execution.ExecutionContext* method), 25
- complete\_object\_value() (*graphql.execution.ExecutionContext* method), 25
- complete\_value() (*graphql.execution.ExecutionContext* method), 26
- concat\_ast() (in module *graphql.utilities*), 86
- context (*graphql.type.GraphQLResolveInfo* property), 73
- context (*graphql.validation.ASTValidationRule* attribute), 91
- context (*graphql.validation.ExecutableDefinitionsRule* attribute), 94
- context (*graphql.validation.FieldsOnCorrectTypeRule* attribute), 95
- context (*graphql.validation.FragmentsOnCompositeTypesRule* attribute), 95
- context (*graphql.validation.KnownArgumentNamesRule* attribute), 96
- context (*graphql.validation.KnownDirectivesRule* attribute), 96
- context (*graphql.validation.KnownFragmentNamesRule* attribute), 97
- context (*graphql.validation.KnownTypeNamesRule* attribute), 97
- context (*graphql.validation.LoneAnonymousOperationRule* attribute), 98
- context (*graphql.validation.LoneSchemaDefinitionRule* attribute), 108
- context (*graphql.validation.NoFragmentCyclesRule* attribute), 98
- context (*graphql.validation.NoUndefinedVariablesRule* attribute), 99
- context (*graphql.validation.NoUnusedFragmentsRule* attribute), 99
- context (*graphql.validation.NoUnusedVariablesRule* attribute), 100
- context (*graphql.validation.OverlappingFieldsCanBeMergedRule* attribute), 101
- context (*graphql.validation.PossibleFragmentSpreadsRule* attribute), 101
- context (*graphql.validation.PossibleTypeExtensionsRule* attribute), 112
- context (*graphql.validation.ProvidedRequiredArgumentsRule* attribute), 102
- context (*graphql.validation.ScalarLeafsRule* attribute), 102
- context (*graphql.validation.SDLValidationRule* attribute), 92

`context` (`graphql.validation.SingleFieldSubscriptionsRule` attribute), 103  
`context` (`graphql.validation.UniqueArgumentNamesRule` attribute), 103  
`context` (`graphql.validation.UniqueDirectiveNamesRule` attribute), 111  
`context` (`graphql.validation.UniqueDirectivesPerLocationRule` attribute), 104  
`context` (`graphql.validation.UniqueEnumValueNamesRule` attribute), 110  
`context` (`graphql.validation.UniqueFieldDefinitionNamesRule` attribute), 111  
`context` (`graphql.validation.UniqueFragmentNamesRule` attribute), 104  
`context` (`graphql.validation.UniqueInputFieldNamesRule` attribute), 105  
`context` (`graphql.validation.UniqueOperationNamesRule` attribute), 105  
`context` (`graphql.validation.UniqueOperationTypesRule` attribute), 109  
`context` (`graphql.validation.UniqueTypeNamesRule` attribute), 109  
`context` (`graphql.validation.UniqueVariableNamesRule` attribute), 106  
`context` (`graphql.validation.ValidationRule` attribute), 93  
`context` (`graphql.validation.ValuesOfCorrectTypeRule` attribute), 106  
`context` (`graphql.validation.VariablesAreInputTypesRule` attribute), 107  
`context` (`graphql.validation.VariablesInAllowedPositionRule` attribute), 108  
`context_value` (`graphql.execution.ExecutionContext` attribute), 26  
`count()` (`graphql.language.SourceLocation` method), 50  
`count()` (`graphql.pyutils.Path` method), 56  
`count()` (`graphql.type.GraphQLResolveInfo` method), 73  
`count()` (`graphql.utilities.BreakingChange` method), 88  
`count()` (`graphql.utilities.DangerousChange` method), 89  
`create_source_event_stream()` (in module `graphql.subscription`), 57  
**D**  
`DangerousChange` (class in `graphql.utilities`), 89  
`DangerousChangeType` (class in `graphql.utilities`), 89  
`data` (`graphql.execution.ExecutionResult` attribute), 28  
`DEFAULT_DEPRECATION_REASON` (in module `graphql.type`), 75  
`default_field_resolver()` (in module `graphql.execution`), 28  
`default_type_resolver()` (in module `graphql.execution`), 28  
`default_value` (`graphql.language.InputValueDefinitionNode` attribute), 37  
`default_value` (`graphql.language.VariableDefinitionNode` attribute), 46  
`default_value` (`graphql.type.GraphQLArgument` attribute), 69  
`default_value` (`graphql.type.GraphQLInputField` attribute), 71  
`DefinitionNode` (class in `graphql.language`), 31  
`definitions` (`graphql.language.DocumentNode` attribute), 32  
`deprecation_reason` (`graphql.type.GraphQLArgument` attribute), 69  
`deprecation_reason` (`graphql.type.GraphQLEnumValue` attribute), 70  
`deprecation_reason` (`graphql.type.GraphQLField` attribute), 71  
`deprecation_reason` (`graphql.type.GraphQLInputField` attribute), 71  
`desc` (`graphql.language.Token` property), 49  
`description` (`graphql.language.DirectiveDefinitionNode` attribute), 32  
`description` (`graphql.language.EnumTypeDefinitionNode` attribute), 32  
`description` (`graphql.language.EnumValueDefinitionNode` attribute), 33  
`description` (`graphql.language.FieldDefinitionNode` attribute), 34  
`description` (`graphql.language.InputObjectTypeDefinitionNode` attribute), 36  
`description` (`graphql.language.InputValueDefinitionNode` attribute), 37  
`description` (`graphql.language.InterfaceTypeDefinitionNode` attribute), 37  
`description` (`graphql.language.ObjectTypeDefinitionNode` attribute), 40  
`description` (`graphql.language.ScalarTypeDefinitionNode` attribute), 42  
`description` (`graphql.language.SchemaDefinitionNode` attribute), 42  
`description` (`graphql.language.TypeDefinitionNode` attribute), 44  
`description` (`graphql.language.UnionTypeDefinitionNode` attribute), 45  
`description` (`graphql.type.GraphQLArgument` attribute), 69  
`description` (`graphql.type.GraphQLDirective` attribute), 75  
`description` (`graphql.type.GraphQLEnumType` attribute), 61  
`description` (`graphql.type.GraphQLEnumValue` attribute), 70  
`description` (`graphql.type.GraphQLField` attribute), 71  
`description` (`graphql.type.GraphQLInputField` attribute), 71

- tribute*), 71
  - `description` (`graphql.type.GraphQLInputObjectType` attribute), 62
  - `description` (`graphql.type.GraphQLInterfaceType` attribute), 64
  - `description` (`graphql.type.GraphQLNamedType` attribute), 72
  - `description` (`graphql.type.GraphQLObjectType` attribute), 65
  - `description` (`graphql.type.GraphQLScalarType` attribute), 66
  - `description` (`graphql.type.GraphQLSchema` attribute), 78
  - `description` (`graphql.type.GraphQLUnionType` attribute), 67
  - `description` (`graphql.utilities.BreakingChange` property), 88
  - `description` (`graphql.utilities.DangerousChange` property), 89
  - `detect_cycle_recursive()` (`graphql.validation.NoFragmentCyclesRule` method), 98
  - `did_you_mean()` (in module `graphql.pyutils`), 55
  - `DIRECTIVE_ARG_REMOVED` (`graphql.utilities.BreakingChangeType` attribute), 89
  - `DIRECTIVE_LOCATION_REMOVED` (`graphql.utilities.BreakingChangeType` attribute), 89
  - `DIRECTIVE_REMOVED` (`graphql.utilities.BreakingChangeType` attribute), 89
  - `DIRECTIVE_REPEATABLE_REMOVED` (`graphql.utilities.BreakingChangeType` attribute), 89
  - `DirectiveDefinitionNode` (class in `graphql.language`), 31
  - `DirectiveLocation` (class in `graphql.language`), 46
  - `DirectiveNode` (class in `graphql.language`), 32
  - `directives` (`graphql.language.EnumTypeDefinitionNode` attribute), 32
  - `directives` (`graphql.language.EnumTypeExtensionNode` attribute), 33
  - `directives` (`graphql.language.EnumValueDefinitionNode` attribute), 33
  - `directives` (`graphql.language.ExecutableDefinitionNode` attribute), 34
  - `directives` (`graphql.language.FieldDefinitionNode` attribute), 34
  - `directives` (`graphql.language.FieldNode` attribute), 35
  - `directives` (`graphql.language.FragmentDefinitionNode` attribute), 35
  - `directives` (`graphql.language.FragmentSpreadNode` attribute), 35
  - `directives` (`graphql.language.InlineFragmentNode` attribute), 36
  - `directives` (`graphql.language.InputObjectTypeDefinitionNode` attribute), 36
  - `directives` (`graphql.language.InputObjectTypeExtensionNode` attribute), 36
  - `directives` (`graphql.language.InputValueDefinitionNode` attribute), 37
  - `directives` (`graphql.language.InterfaceTypeDefinitionNode` attribute), 37
  - `directives` (`graphql.language.InterfaceTypeExtensionNode` attribute), 38
  - `directives` (`graphql.language.ObjectTypeDefinitionNode` attribute), 40
  - `directives` (`graphql.language.ObjectTypeExtensionNode` attribute), 40
  - `directives` (`graphql.language.OperationDefinitionNode` attribute), 41
  - `directives` (`graphql.language.ScalarTypeDefinitionNode` attribute), 42
  - `directives` (`graphql.language.ScalarTypeExtensionNode` attribute), 42
  - `directives` (`graphql.language.SchemaDefinitionNode` attribute), 42
  - `directives` (`graphql.language.SchemaExtensionNode` attribute), 43
  - `directives` (`graphql.language.SelectionNode` attribute), 43
  - `directives` (`graphql.language.TypeDefinitionNode` attribute), 44
  - `directives` (`graphql.language.TypeExtensionNode` attribute), 44
  - `directives` (`graphql.language.UnionTypeDefinitionNode` attribute), 45
  - `directives` (`graphql.language.UnionTypeExtensionNode` attribute), 45
  - `directives` (`graphql.language.VariableDefinitionNode` attribute), 46
  - `directives` (`graphql.type.GraphQLSchema` attribute), 79
  - `do_types_overlap()` (in module `graphql.utilities`), 87
  - `document` (`graphql.validation.ASTValidationContext` attribute), 91
  - `document` (`graphql.validation.SDLValidationContext` attribute), 91
  - `document` (`graphql.validation.ValidationContext` attribute), 92
  - `DocumentNode` (class in `graphql.language`), 32
  - `does_fragment_condition_match()` (`graphql.execution.ExecutionContext` method), 26
  - `DOLLAR` (`graphql.language.TokenKind` attribute), 49
- ## E
- `emit()` (`graphql.pyutils.SimplePubSub` method), 55

`empty_queue()` (`graphql.pyutils.SimplePubSubIterator` method), 55

`end` (`graphql.language.Location` attribute), 30

`end` (`graphql.language.Token` attribute), 49

`end_token` (`graphql.language.Location` attribute), 30

`ensure_valid_runtime_type()` (`graphql.execution.ExecutionContext` method), 26

`enter()` (`graphql.language.ParallelVisitor` method), 53

`enter()` (`graphql.utilities.TypeInfo` method), 84

`enter()` (`graphql.utilities.TypeInfoVisitor` method), 85

`enter()` (`graphql.validation.UniqueDirectivesPerLocationRule` method), 104

`enter_argument()` (`graphql.utilities.TypeInfo` method), 84

`enter_argument()` (`graphql.validation.KnownArgumentNamesRule` method), 96

`enter_argument()` (`graphql.validation.UniqueArgumentNamesRule` method), 103

`enter_boolean_value()` (`graphql.validation.ValuesOfCorrectTypeRule` method), 106

`enter_directive()` (`graphql.utilities.TypeInfo` method), 84

`enter_directive()` (`graphql.validation.KnownArgumentNamesRule` method), 96

`enter_directive()` (`graphql.validation.KnownDirectivesRule` method), 96

`enter_directive()` (`graphql.validation.UniqueArgumentNamesRule` method), 103

`enter_directive_definition()` (`graphql.validation.UniqueDirectiveNamesRule` method), 111

`enter_document()` (`graphql.validation.ExecutableDefinitionsRule` method), 94

`enter_document()` (`graphql.validation.LoneAnonymousOperationRule` method), 98

`enter_enum_type_definition()` (`graphql.validation.UniqueEnumValueNamesRule` method), 110

`enter_enum_type_definition()` (`graphql.validation.UniqueTypeNamesRule` method), 109

`enter_enum_type_extension()` (`graphql.validation.PossibleTypeExtensionsRule` method), 112

`enter_enum_type_extension()` (`graphql.validation.UniqueEnumValueNamesRule` method), 110

`enter_enum_value()` (`graphql.utilities.TypeInfo` method), 84

`enter_enum_value()` (`graphql.validation.ValuesOfCorrectTypeRule` method), 106

`enter_field()` (`graphql.utilities.TypeInfo` method), 84

`enter_field()` (`graphql.validation.FieldsOnCorrectTypeRule` method), 95

`enter_field()` (`graphql.validation.ScalarLeafsRule` method), 102

`enter_field()` (`graphql.validation.UniqueArgumentNamesRule` method), 103

`enter_float_value()` (`graphql.validation.ValuesOfCorrectTypeRule` method), 106

`enter_fragment_definition()` (`graphql.utilities.TypeInfo` method), 84

`enter_fragment_definition()` (`graphql.validation.FragmentsOnCompositeTypesRule` method), 95

`enter_fragment_definition()` (`graphql.validation.NoFragmentCyclesRule` method), 98

`enter_fragment_definition()` (`graphql.validation.NoUnusedFragmentsRule` method), 99

`enter_fragment_definition()` (`graphql.validation.UniqueFragmentNamesRule` method), 104

`enter_fragment_definition()` (`graphql.validation.UniqueOperationNamesRule` static method), 105

`enter_fragment_spread()` (`graphql.validation.KnownFragmentNamesRule` method), 97

`enter_fragment_spread()` (`graphql.validation.PossibleFragmentSpreadsRule` method), 101

`enter_inline_fragment()` (`graphql.utilities.TypeInfo` method), 84

`enter_inline_fragment()` (`graphql.validation.FragmentsOnCompositeTypesRule` method), 95

`enter_inline_fragment()` (`graphql.validation.PossibleFragmentSpreadsRule` method), 101

`enter_input_object_type_definition()` (`graphql.validation.UniqueFieldDefinitionNamesRule` method), 111

`enter_input_object_type_definition()` (`graphql.validation.UniqueTypeNamesRule` method), 109

`enter_input_object_type_extension()` (`graphql.validation.PossibleTypeExtensionsRule` method), 112

`enter_input_object_type_extension()` (`graphql.validation.UniqueFieldDefinitionNamesRule` method), 111

`enter_int_value()` (`graphql.validation.ValuesOfCorrectTypeRule` method), 106



<code>enter_interface_type_definition()</code>	<code>enter_operation_definition()</code>
( <code>graphql.validation.UniqueFieldDefinitionNamesRule</code>	( <code>graphql.validation.NoUndefinedVariablesRule</code>
<code>method</code> ), 111	<code>method</code> ), 99
<code>enter_interface_type_definition()</code>	<code>enter_operation_definition()</code>
( <code>graphql.validation.UniqueTypeNamesRule</code>	( <code>graphql.validation.NoUnusedFragmentsRule</code>
<code>method</code> ), 109	<code>method</code> ), 99
<code>enter_interface_type_extension()</code>	<code>enter_operation_definition()</code>
( <code>graphql.validation.PossibleTypeExtensionsRule</code>	( <code>graphql.validation.NoUnusedVariablesRule</code>
<code>method</code> ), 112	<code>method</code> ), 100
<code>enter_interface_type_extension()</code>	<code>enter_operation_definition()</code>
( <code>graphql.validation.UniqueFieldDefinitionNamesRule</code>	( <code>graphql.validation.SingleFieldSubscriptionsRule</code>
<code>method</code> ), 111	<code>method</code> ), 103
<code>enter_list_value()</code> ( <code>graphql.utilities.TypeInfo</code>	<code>enter_operation_definition()</code>
<code>method</code> ), 84	( <code>graphql.validation.UniqueFragmentNamesRule</code>
<code>enter_list_value()</code> ( <code>graphql.validation.ValuesOfCorrectTypeRule</code>	<code>static method</code> ), 104
<code>method</code> ), 106	<code>enter_operation_definition()</code>
<code>enter_named_type()</code> ( <code>graphql.validation.KnownTypeNamesRule</code>	( <code>graphql.validation.UniqueOperationNamesRule</code>
<code>method</code> ), 97	<code>method</code> ), 105
<code>enter_null_value()</code> ( <code>graphql.validation.ValuesOfCorrectTypeRule</code>	<code>enter_operation_definition()</code>
<code>method</code> ), 107	( <code>graphql.validation.UniqueVariableNamesRule</code>
<code>enter_object_field()</code> ( <code>graphql.utilities.TypeInfo</code>	<code>method</code> ), 106
<code>method</code> ), 84	<code>enter_operation_definition()</code>
<code>enter_object_field()</code>	( <code>graphql.validation.VariablesInAllowedPositionRule</code>
( <code>graphql.validation.UniqueInputFieldNamesRule</code>	<code>method</code> ), 108
<code>method</code> ), 105	<code>enter_scalar_type_definition()</code>
<code>enter_object_field()</code>	( <code>graphql.validation.UniqueTypeNamesRule</code>
( <code>graphql.validation.ValuesOfCorrectTypeRule</code>	<code>method</code> ), 110
<code>method</code> ), 107	<code>enter_scalar_type_extension()</code>
<code>enter_object_type_definition()</code>	( <code>graphql.validation.PossibleTypeExtensionsRule</code>
( <code>graphql.validation.UniqueFieldDefinitionNamesRule</code>	<code>method</code> ), 112
<code>method</code> ), 111	<code>enter_schema_definition()</code>
<code>enter_object_type_definition()</code>	( <code>graphql.validation.LoneSchemaDefinitionRule</code>
( <code>graphql.validation.UniqueTypeNamesRule</code>	<code>method</code> ), 108
<code>method</code> ), 109	<code>enter_schema_definition()</code>
<code>enter_object_type_extension()</code>	( <code>graphql.validation.UniqueOperationTypesRule</code>
( <code>graphql.validation.PossibleTypeExtensionsRule</code>	<code>method</code> ), 109
<code>method</code> ), 112	<code>enter_schema_extension()</code>
<code>enter_object_type_extension()</code>	( <code>graphql.validation.UniqueOperationTypesRule</code>
( <code>graphql.validation.UniqueFieldDefinitionNamesRule</code>	<code>method</code> ), 109
<code>method</code> ), 111	<code>enter_selection_set()</code> ( <code>graphql.utilities.TypeInfo</code>
<code>enter_object_value()</code>	<code>method</code> ), 84
( <code>graphql.validation.UniqueInputFieldNamesRule</code>	<code>enter_selection_set()</code>
<code>method</code> ), 105	( <code>graphql.validation.OverlappingFieldsCanBeMergedRule</code>
<code>enter_object_value()</code>	<code>method</code> ), 101
( <code>graphql.validation.ValuesOfCorrectTypeRule</code>	<code>enter_string_value()</code>
<code>method</code> ), 107	( <code>graphql.validation.ValuesOfCorrectTypeRule</code>
<code>enter_operation_definition()</code>	<code>method</code> ), 107
( <code>graphql.utilities.TypeInfo</code> <code>method</code> ), 84	<code>enter_union_type_definition()</code>
<code>enter_operation_definition()</code>	( <code>graphql.validation.UniqueTypeNamesRule</code>
( <code>graphql.validation.LoneAnonymousOperationRule</code>	<code>method</code> ), 110
<code>method</code> ), 98	<code>enter_union_type_extension()</code>
<code>enter_operation_definition()</code>	( <code>graphql.validation.PossibleTypeExtensionsRule</code>
( <code>graphql.validation.NoFragmentCyclesRule</code>	<code>method</code> ), 112
<code>static method</code> ), 98	<code>enter_variable_definition()</code>

(*graphql.utilities.TypeInfo* method), 84  
*enter\_variable\_definition()*  
 (*graphql.validation.NoUndefinedVariablesRule* method), 99  
*enter\_variable\_definition()*  
 (*graphql.validation.NoUnusedVariablesRule* method), 100  
*enter\_variable\_definition()*  
 (*graphql.validation.UniqueVariableNamesRule* method), 106  
*enter\_variable\_definition()*  
 (*graphql.validation.VariablesAreInputTypesRule* method), 107  
*enter\_variable\_definition()*  
 (*graphql.validation.VariablesInAllowedPositionRule* method), 108  
*ENUM* (*graphql.language.DirectiveLocation* attribute), 46  
*ENUM* (*graphql.type.TypeKind* attribute), 76  
*ENUM\_VALUE* (*graphql.language.DirectiveLocation* attribute), 46  
*EnumTypeDefinitionNode* (class in *graphql.language*), 32  
*EnumTypeExtensionNode* (class in *graphql.language*), 33  
*EnumValueDefinitionNode* (class in *graphql.language*), 33  
*EnumValueNode* (class in *graphql.language*), 33  
*EOF* (*graphql.language.TokenKind* attribute), 49  
*EQUALS* (*graphql.language.TokenKind* attribute), 49  
*errors* (*graphql.execution.ExecutionContext* attribute), 26  
*errors* (*graphql.execution.ExecutionResult* attribute), 28  
*ExecutableDefinitionNode* (class in *graphql.language*), 34  
*ExecutableDefinitionsRule* (class in *graphql.validation*), 94  
*execute()* (in module *graphql.execution*), 29  
*execute\_fields()* (*graphql.execution.ExecutionContext* method), 26  
*execute\_fields\_serially()*  
 (*graphql.execution.ExecutionContext* method), 26  
*execute\_operation()*  
 (*graphql.execution.ExecutionContext* method), 27  
*execute\_sync()* (in module *graphql.execution*), 29  
*ExecutionContext* (class in *graphql.execution*), 23  
*ExecutionResult* (class in *graphql.execution*), 28  
*extend\_schema()* (in module *graphql.utilities*), 81  
*extension\_ast\_nodes*  
 (*graphql.type.GraphQLEnumType* attribute), 61  
*extension\_ast\_nodes*  
 (*graphql.type.GraphQLInputObjectType* attribute), 62  
*extension\_ast\_nodes*  
 (*graphql.type.GraphQLInterfaceType* attribute), 64  
*extension\_ast\_nodes*  
 (*graphql.type.GraphQLNamedType* attribute), 72  
*extension\_ast\_nodes*  
 (*graphql.type.GraphQLObjectType* attribute), 65  
*extension\_ast\_nodes*  
 (*graphql.type.GraphQLScalarType* attribute), 66  
*extension\_ast\_nodes* (*graphql.type.GraphQLSchema* attribute), 79  
*extension\_ast\_nodes*  
 (*graphql.type.GraphQLUnionType* attribute), 67  
*extensions* (*graphql.error.GraphQLError* attribute), 21  
*extensions* (*graphql.error.GraphQLSyntaxError* attribute), 22  
*extensions* (*graphql.execution.ExecutionResult* attribute), 28  
*extensions* (*graphql.type.GraphQLArgument* attribute), 69  
*extensions* (*graphql.type.GraphQLDirective* attribute), 75  
*extensions* (*graphql.type.GraphQLEnumType* attribute), 61  
*extensions* (*graphql.type.GraphQLEnumValue* attribute), 70  
*extensions* (*graphql.type.GraphQLField* attribute), 71  
*extensions* (*graphql.type.GraphQLInputField* attribute), 71  
*extensions* (*graphql.type.GraphQLInputObjectType* attribute), 63  
*extensions* (*graphql.type.GraphQLInterfaceType* attribute), 64  
*extensions* (*graphql.type.GraphQLNamedType* attribute), 72  
*extensions* (*graphql.type.GraphQLObjectType* attribute), 65  
*extensions* (*graphql.type.GraphQLScalarType* attribute), 66  
*extensions* (*graphql.type.GraphQLSchema* attribute), 79  
*extensions* (*graphql.type.GraphQLUnionType* attribute), 68  
**F**  
*FIELD* (*graphql.language.DirectiveLocation* attribute), 46  
*FIELD\_CHANGED\_KIND* (*graphql.utilities.BreakingChangeType* attribute), 89

- FIELD\_DEFINITION (*graphql.language.DirectiveLocation* attribute), 47
- field\_name (*graphql.type.GraphQLResolveInfo* property), 73
- field\_nodes (*graphql.type.GraphQLResolveInfo* property), 73
- FIELD\_REMOVED (*graphql.utilities.BreakingChangeType* attribute), 89
- field\_resolver (*graphql.execution.ExecutionContext* attribute), 27
- FieldDefinitionNode (class in *graphql.language*), 34
- FieldNode (class in *graphql.language*), 34
- fields (*graphql.language.InputObjectTypeDefinitionNode* attribute), 36
- fields (*graphql.language.InputObjectTypeExtensionNode* attribute), 36
- fields (*graphql.language.InterfaceTypeDefinitionNode* attribute), 37
- fields (*graphql.language.InterfaceTypeExtensionNode* attribute), 38
- fields (*graphql.language.ObjectTypeDefinitionNode* attribute), 40
- fields (*graphql.language.ObjectTypeExtensionNode* attribute), 40
- fields (*graphql.language.ObjectValueNode* attribute), 41
- fields (*graphql.type.GraphQLInputObjectType* attribute), 63
- fields (*graphql.type.GraphQLInterfaceType* attribute), 64
- fields (*graphql.type.GraphQLObjType* attribute), 65
- FieldsOnCorrectTypeRule (class in *graphql.validation*), 94
- find\_breaking\_changes() (in module *graphql.utilities*), 88
- find\_dangerous\_changes() (in module *graphql.utilities*), 88
- find\_deprecated\_usages() (in module *graphql.utilities*), 90
- FLOAT (*graphql.language.TokenKind* attribute), 49
- FloatValueNode (class in *graphql.language*), 35
- format\_error() (in module *graphql.error*), 23
- formatted (*graphql.error.GraphQLError* property), 21
- formatted (*graphql.error.GraphQLSyntaxError* property), 22
- formatted (*graphql.execution.ExecutionResult* property), 28
- formatted (*graphql.language.SourceLocation* property), 50
- FRAGMENT\_DEFINITION (*graphql.language.DirectiveLocation* attribute), 47
- FRAGMENT\_SPREAD (*graphql.language.DirectiveLocation* attribute), 47
- FragmentDefinitionNode (class in *graphql.language*), 35
- fragments (*graphql.execution.ExecutionContext* attribute), 27
- fragments (*graphql.type.GraphQLResolveInfo* property), 73
- FragmentsOnCompositeTypesRule (class in *graphql.validation*), 95
- FragmentSpreadNode (class in *graphql.language*), 35
- FrozenDict (class in *graphql.pyutils*), 56
- FrozenError (class in *graphql.pyutils*), 56
- FrozenList (class in *graphql.pyutils*), 56
- ## G
- get\_argument() (*graphql.utilities.TypeInfo* method), 84
- get\_argument() (*graphql.validation.ValidationContext* method), 92
- get\_default\_value() (*graphql.utilities.TypeInfo* method), 84
- get\_directive() (*graphql.type.GraphQLSchema* method), 79
- get\_directive() (*graphql.utilities.TypeInfo* method), 84
- get\_directive() (*graphql.validation.ValidationContext* method), 92
- get\_directive\_values() (in module *graphql.execution*), 30
- get\_enum\_value() (*graphql.utilities.TypeInfo* method), 84
- get\_enum\_value() (*graphql.validation.ValidationContext* method), 92
- get\_field\_def() (*graphql.utilities.TypeInfo* method), 84
- get\_field\_def() (*graphql.validation.ValidationContext* method), 92
- get\_field\_resolver() (*graphql.execution.MiddlewareManager* method), 28
- get\_fragment() (*graphql.validation.ASTValidationContext* method), 91
- get\_fragment() (*graphql.validation.SDLValidationContext* method), 91
- get\_fragment() (*graphql.validation.ValidationContext* method), 93
- get\_fragment\_spreads() (*graphql.validation.ASTValidationContext* method), 91
- get\_fragment\_spreads() (*graphql.validation.SDLValidationContext* method), 91
- get\_fragment\_spreads() (*graphql.validation.ValidationContext* method), 93

`get_fragment_type()`  
(*graphql.validation.PossibleFragmentSpreadsRule* method), 101

`get_implementations()`  
(*graphql.type.GraphQLSchema* method), 79

`get_input_type()` (*graphql.utilities.TypeInfo* method), 84

`get_input_type()` (*graphql.validation.ValidationContext* method), 93

`get_introspection_query()` (in module *graphql.utilities*), 80

`get_location()` (*graphql.language.Source* method), 52

`get_location()` (in module *graphql.language*), 50

`get_named_type()` (in module *graphql.type*), 60

`get_nullable_type()` (in module *graphql.type*), 60

`get_operation_ast()` (in module *graphql.utilities*), 80

`get_operation_root_type()` (in module *graphql.utilities*), 80

`get_parent_input_type()` (*graphql.utilities.TypeInfo* method), 85

`get_parent_input_type()`  
(*graphql.validation.ValidationContext* method), 93

`get_parent_type()` (*graphql.utilities.TypeInfo* method), 85

`get_parent_type()` (*graphql.validation.ValidationContext* method), 93

`get_possible_types()`  
(*graphql.type.GraphQLSchema* method), 79

`get_recursive_variable_usages()`  
(*graphql.validation.ValidationContext* method), 93

`get_recursively_referenced_fragments()`  
(*graphql.validation.ASTValidationContext* method), 91

`get_recursively_referenced_fragments()`  
(*graphql.validation.SDLValidationContext* method), 92

`get_recursively_referenced_fragments()`  
(*graphql.validation.ValidationContext* method), 93

`get_subscriber()` (*graphql.pyutils.SimplePubSub* method), 55

`get_type()` (*graphql.type.GraphQLSchema* method), 79

`get_type()` (*graphql.utilities.TypeInfo* method), 85

`get_type()` (*graphql.validation.ValidationContext* method), 93

`get_variable_usages()`  
(*graphql.validation.ValidationContext* method), 93

`get_visit_fn()` (*graphql.language.ParallelVisitor* method), 54

`get_visit_fn()` (*graphql.language.Visitor* method), 53

`get_visit_fn()` (*graphql.utilities.TypeInfoVisitor* method), 86

`get_visit_fn()` (*graphql.validation.ASTValidationRule* method), 91

`get_visit_fn()` (*graphql.validation.ExecutableDefinitionsRule* method), 94

`get_visit_fn()` (*graphql.validation.FieldsOnCorrectTypeRule* method), 95

`get_visit_fn()` (*graphql.validation.FragmentsOnCompositeTypesRule* method), 95

`get_visit_fn()` (*graphql.validation.KnownArgumentNamesRule* method), 96

`get_visit_fn()` (*graphql.validation.KnownDirectivesRule* method), 96

`get_visit_fn()` (*graphql.validation.KnownFragmentNamesRule* method), 97

`get_visit_fn()` (*graphql.validation.KnownTypeNamesRule* method), 97

`get_visit_fn()` (*graphql.validation.LoneAnonymousOperationRule* method), 98

`get_visit_fn()` (*graphql.validation.LoneSchemaDefinitionRule* method), 108

`get_visit_fn()` (*graphql.validation.NoFragmentCyclesRule* method), 98

`get_visit_fn()` (*graphql.validation.NoUndefinedVariablesRule* method), 99

`get_visit_fn()` (*graphql.validation.NoUnusedFragmentsRule* method), 99

`get_visit_fn()` (*graphql.validation.NoUnusedVariablesRule* method), 100

`get_visit_fn()` (*graphql.validation.OverlappingFieldsCanBeMergedRule* method), 101

`get_visit_fn()` (*graphql.validation.PossibleFragmentSpreadsRule* method), 101

`get_visit_fn()` (*graphql.validation.PossibleTypeExtensionsRule* method), 112

`get_visit_fn()` (*graphql.validation.ProvidedRequiredArgumentsRule* method), 102

`get_visit_fn()` (*graphql.validation.ScalarLeafsRule* method), 102

`get_visit_fn()` (*graphql.validation.SDLValidationRule* method), 92

`get_visit_fn()` (*graphql.validation.SingleFieldSubscriptionsRule* method), 103

`get_visit_fn()` (*graphql.validation.UniqueArgumentNamesRule* method), 103

`get_visit_fn()` (*graphql.validation.UniqueDirectiveNamesRule* method), 111

`get_visit_fn()` (*graphql.validation.UniqueDirectivesPerLocationRule* method), 104

`get_visit_fn()` (*graphql.validation.UniqueEnumValueNamesRule* method), 110

`get_visit_fn()` (*graphql.validation.UniqueFieldDefinitionNamesRule*



- method*), 111
  - `get_visit_fn()` (*graphql.validation.UniqueFragmentNameRule* *method*), 104
  - `get_visit_fn()` (*graphql.validation.UniqueInputFieldNameRule* *method*), 105
  - `get_visit_fn()` (*graphql.validation.UniqueOperationNameRule* *method*), 105
  - `get_visit_fn()` (*graphql.validation.UniqueOperationTypesRule* *method*), 109
  - `get_visit_fn()` (*graphql.validation.UniqueTypeNamesRule* *method*), 110
  - `get_visit_fn()` (*graphql.validation.UniqueVariableNamesRule* *method*), 106
  - `get_visit_fn()` (*graphql.validation.ValidationRule* *method*), 94
  - `get_visit_fn()` (*graphql.validation.ValuesOfCorrectTypeRule* *method*), 107
  - `get_visit_fn()` (*graphql.validation.VariablesAreInputTypesRule* *method*), 107
  - `get_visit_fn()` (*graphql.validation.VariablesInAllowedPositionsRule* *method*), 108
  - `graphql`
    - module, 19
  - `graphql()` (*in module graphql*), 20
  - `graphql.error`
    - module, 21
  - `graphql.execution`
    - module, 23
  - `graphql.language`
    - module, 30
  - `graphql.pyutils`
    - module, 54
  - `graphql.subscription`
    - module, 57
  - `graphql.type`
    - module, 58
  - `graphql.utilities`
    - module, 80
  - `graphql.validation`
    - module, 90
  - `graphql.validation.rules`
    - module, 94
  - `graphql_sync()` (*in module graphql*), 20
  - `GraphQLAbstractType` (*in module graphql.type*), 69
  - `GraphQLArgument` (*class in graphql.type*), 69
  - `GraphQLArgumentMap` (*in module graphql.type*), 69
  - `GraphQLBoolean` (*in module graphql.type*), 77
  - `GraphQLCompositeType` (*in module graphql.type*), 70
  - `GraphQLDeprecatedDirective` (*in module graphql.type*), 75
  - `GraphQLDirective` (*class in graphql.type*), 75
  - `GraphQLEnumType` (*class in graphql.type*), 61
  - `GraphQLEnumValue` (*class in graphql.type*), 70
  - `GraphQLEnumValueMap` (*in module graphql.type*), 70
  - `GraphQLError`, 21
  - `GraphQLField` (*class in graphql.type*), 70
  - `GraphQLFieldMap` (*in module graphql.type*), 71
  - `GraphQLFieldResolver` (*in module graphql.type*), 73
  - `GraphQLFloat` (*in module graphql.type*), 77
  - `GraphQLID` (*in module graphql.type*), 77
  - `GraphQLIncludeDirective` (*in module graphql.type*), 75
  - `GraphQLInputField` (*class in graphql.type*), 71
  - `GraphQLInputFieldMap` (*in module graphql.type*), 72
  - `GraphQLInputObjectType` (*class in graphql.type*), 62
  - `GraphQLInputType` (*in module graphql.type*), 72
  - `GraphQLInt` (*in module graphql.type*), 77
  - `GraphQLInterfaceType` (*class in graphql.type*), 63
  - `GraphQLIsTypeOfFn` (*in module graphql.type*), 73
  - `GraphQLLeafType` (*in module graphql.type*), 72
  - `GraphQLList` (*class in graphql.type*), 68
  - `GraphQLNamedType` (*class in graphql.type*), 72
  - `GraphQLNonNull` (*class in graphql.type*), 68
  - `GraphQLNullableType` (*in module graphql.type*), 72
  - `GraphQLObjectType` (*class in graphql.type*), 64
  - `GraphQLOutputType` (*in module graphql.type*), 72
  - `GraphQLResolveInfo` (*class in graphql.type*), 73
  - `GraphQLScalarType` (*class in graphql.type*), 65
  - `GraphQLSchema` (*class in graphql.type*), 77
  - `GraphQLSkipDirective` (*in module graphql.type*), 75
  - `GraphQLString` (*in module graphql.type*), 77
  - `GraphQLSyntaxError`, 22
  - `GraphQLType` (*class in graphql.type*), 72
  - `GraphQLTypeResolver` (*in module graphql.type*), 74
  - `GraphQLUnionType` (*class in graphql.type*), 67
  - `GraphQLWrappingType` (*class in graphql.type*), 73
- ## H
- `handle_field_error()`
    - (*graphql.execution.ExecutionContext* *method*), 27
- ## I
- `identity_func()` (*in module graphql.pyutils*), 55
  - `IDLE` (*graphql.language.ParallelVisitor* *attribute*), 53
  - `IDLE` (*graphql.language.Visitor* *attribute*), 53
  - `IDLE` (*graphql.utilities.TypeInfoVisitor* *attribute*), 85
  - `IDLE` (*graphql.validation.ASTValidationRule* *attribute*), 91
  - `IDLE` (*graphql.validation.ExecutableDefinitionsRule* *attribute*), 94
  - `IDLE` (*graphql.validation.FieldsOnCorrectTypeRule* *attribute*), 94
  - `IDLE` (*graphql.validation.FragmentsOnCompositeTypesRule* *attribute*), 95
  - `IDLE` (*graphql.validation.KnownArgumentNamesRule* *attribute*), 95

IDLE ( <i>graphql.validation.KnownDirectivesRule</i> attribute), 96	IDLE ( <i>graphql.validation.ValuesOfCorrectTypeRule</i> attribute), 106
IDLE ( <i>graphql.validation.KnownFragmentNamesRule</i> attribute), 97	IDLE ( <i>graphql.validation.VariablesAreInputTypesRule</i> attribute), 107
IDLE ( <i>graphql.validation.KnownTypeNamesRule</i> attribute), 97	IDLE ( <i>graphql.validation.VariablesInAllowedPositionRule</i> attribute), 107
IDLE ( <i>graphql.validation.LoneAnonymousOperationRule</i> attribute), 98	IDLE (in module <i>graphql.language</i> ), 54
IDLE ( <i>graphql.validation.LoneSchemaDefinitionRule</i> attribute), 108	IMPLEMENTED_INTERFACE_ADDED ( <i>graphql.utilities.DangerousChangeType</i> attribute), 89
IDLE ( <i>graphql.validation.NoFragmentCyclesRule</i> attribute), 98	IMPLEMENTED_INTERFACE_REMOVED ( <i>graphql.utilities.BreakingChangeType</i> attribute), 89
IDLE ( <i>graphql.validation.NoUndefinedVariablesRule</i> attribute), 99	index() ( <i>graphql.language.SourceLocation</i> method), 50
IDLE ( <i>graphql.validation.NoUnusedFragmentsRule</i> attribute), 99	index() ( <i>graphql.pyutils.Path</i> method), 56
IDLE ( <i>graphql.validation.NoUnusedVariablesRule</i> attribute), 100	index() ( <i>graphql.type.GraphQLResolveInfo</i> method), 74
IDLE ( <i>graphql.validation.OverlappingFieldsCanBeMergedRule</i> attribute), 100	index() ( <i>graphql.utilities.BreakingChange</i> method), 88
IDLE ( <i>graphql.validation.PossibleFragmentSpreadsRule</i> attribute), 101	index() ( <i>graphql.utilities.DangerousChange</i> method), 89
IDLE ( <i>graphql.validation.PossibleTypeExtensionsRule</i> attribute), 112	INLINE_FRAGMENT ( <i>graphql.language.DirectiveLocation</i> attribute), 47
IDLE ( <i>graphql.validation.ProvidedRequiredArgumentsRule</i> attribute), 101	InlineFragmentNode (class in <i>graphql.language</i> ), 36
IDLE ( <i>graphql.validation.ScalarLeafsRule</i> attribute), 102	INPUT_FIELD_DEFINITION ( <i>graphql.language.DirectiveLocation</i> attribute), 47
IDLE ( <i>graphql.validation.SDLValidationRule</i> attribute), 92	INPUT_OBJECT ( <i>graphql.language.DirectiveLocation</i> attribute), 47
IDLE ( <i>graphql.validation.SingleFieldSubscriptionsRule</i> attribute), 102	INPUT_OBJECT ( <i>graphql.type.TypeKind</i> attribute), 76
IDLE ( <i>graphql.validation.UniqueArgumentNamesRule</i> attribute), 103	InputObjectTypeDefinitionNode (class in <i>graphql.language</i> ), 36
IDLE ( <i>graphql.validation.UniqueDirectiveNamesRule</i> attribute), 111	InputObjectTypeExtensionNode (class in <i>graphql.language</i> ), 36
IDLE ( <i>graphql.validation.UniqueDirectivesPerLocationRule</i> attribute), 104	InputValueDefinitionNode (class in <i>graphql.language</i> ), 37
IDLE ( <i>graphql.validation.UniqueEnumValueNamesRule</i> attribute), 110	inspect() (in module <i>graphql.pyutils</i> ), 55
IDLE ( <i>graphql.validation.UniqueFieldDefinitionNamesRule</i> attribute), 110	INT ( <i>graphql.language.TokenKind</i> attribute), 49
IDLE ( <i>graphql.validation.UniqueFragmentNamesRule</i> attribute), 104	INTERFACE ( <i>graphql.language.DirectiveLocation</i> attribute), 47
IDLE ( <i>graphql.validation.UniqueInputFieldNamesRule</i> attribute), 105	INTERFACE ( <i>graphql.type.TypeKind</i> attribute), 76
IDLE ( <i>graphql.validation.UniqueOperationNamesRule</i> attribute), 105	interfaces ( <i>graphql.language.InterfaceTypeDefinitionNode</i> attribute), 38
IDLE ( <i>graphql.validation.UniqueOperationTypesRule</i> attribute), 108	interfaces ( <i>graphql.language.InterfaceTypeExtensionNode</i> attribute), 38
IDLE ( <i>graphql.validation.UniqueTypeNamesRule</i> attribute), 109	interfaces ( <i>graphql.language.ObjectTypeDefinitionNode</i> attribute), 40
IDLE ( <i>graphql.validation.UniqueVariableNamesRule</i> attribute), 106	interfaces ( <i>graphql.language.ObjectTypeExtensionNode</i> attribute), 40
IDLE ( <i>graphql.validation.ValidationRule</i> attribute), 93	interfaces ( <i>graphql.type.GraphQLInterfaceType</i> attribute), 64
	interfaces ( <i>graphql.type.GraphQLOBJECTType</i> attribute), 65
	InterfaceTypeDefinitionNode (class in <i>graphql.language</i> ), 37
	InterfaceTypeExtensionNode (class in <i>graphql.language</i> ), 37

- `graphql.language)`, 38
  - `introspection_from_schema()` (in module `graphql.utilities`), 80
  - `introspection_types` (in module `graphql.type`), 76
  - `IntValueNode` (class in `graphql.language`), 37
  - `is_awaitable` (`graphql.type.GraphQLResolveInfo` property), 74
  - `is_awaitable()` (`graphql.execution.ExecutionContext` static method), 27
  - `is_closed` (`graphql.subscription.MapAsyncIterator` property), 58
  - `is_composite_type()` (in module `graphql.type`), 58
  - `is_definition_node()` (in module `graphql.language`), 47
  - `is_deprecated` (`graphql.type.GraphQLEnumValue` property), 70
  - `is_deprecated` (`graphql.type.GraphQLField` property), 71
  - `is_directive()` (in module `graphql.type`), 74
  - `is_enum_type()` (in module `graphql.type`), 58
  - `is_equal_type()` (in module `graphql.utilities`), 87
  - `is_executable_definition_node()` (in module `graphql.language`), 47
  - `is_finite()` (in module `graphql.pyutils`), 55
  - `is_input_object_type()` (in module `graphql.type`), 58
  - `is_input_type()` (in module `graphql.type`), 58
  - `is_integer()` (in module `graphql.pyutils`), 56
  - `is_interface_type()` (in module `graphql.type`), 58
  - `is_introspection_type()` (in module `graphql.type`), 76
  - `is_leaf_type()` (in module `graphql.type`), 58
  - `is_list_type()` (in module `graphql.type`), 59
  - `is_named_type()` (in module `graphql.type`), 59
  - `is_non_null_type()` (in module `graphql.type`), 59
  - `is_nullable_type()` (in module `graphql.type`), 59
  - `is_object_type()` (in module `graphql.type`), 59
  - `is_output_type()` (in module `graphql.type`), 59
  - `is_possible_type()` (`graphql.type.GraphQLSchema` method), 79
  - `is_repeatable` (`graphql.type.GraphQLDirective` attribute), 75
  - `is_scalar_type()` (in module `graphql.type`), 59
  - `is_schema()` (in module `graphql.type`), 77
  - `is_selection_node()` (in module `graphql.language`), 47
  - `is_specified_directive()` (in module `graphql.type`), 74
  - `is_specified_scalar_type()` (in module `graphql.type`), 77
  - `is_sub_type()` (`graphql.type.GraphQLSchema` method), 79
  - `is_type()` (in module `graphql.type`), 59
  - `is_type_definition_node()` (in module `graphql.language`), 47
  - `is_type_extension_node()` (in module `graphql.language`), 47
  - `is_type_node()` (in module `graphql.language`), 47
  - `is_type_of` (`graphql.type.GraphQLObjectType` attribute), 65
  - `is_type_sub_type_of()` (in module `graphql.utilities`), 87
  - `is_type_system_definition_node()` (in module `graphql.language`), 47
  - `is_type_system_extension_node()` (in module `graphql.language`), 47
  - `is_union_type()` (in module `graphql.type`), 59
  - `is_valid_name_error()` (in module `graphql.utilities`), 88
  - `is_valid_value_node()` (`graphql.validation.ValuesOfCorrectTypeRule` method), 107
  - `is_value_node()` (in module `graphql.language`), 47
  - `is_wrapping_type()` (in module `graphql.type`), 59
- ## K
- `key` (`graphql.pyutils.Path` property), 56
  - `keys` (`graphql.language.ArgumentNode` attribute), 31
  - `keys` (`graphql.language.BooleanValueNode` attribute), 31
  - `keys` (`graphql.language.DefinitionNode` attribute), 31
  - `keys` (`graphql.language.DirectiveDefinitionNode` attribute), 32
  - `keys` (`graphql.language.DirectiveNode` attribute), 32
  - `keys` (`graphql.language.DocumentNode` attribute), 32
  - `keys` (`graphql.language.EnumTypeDefinitionNode` attribute), 32
  - `keys` (`graphql.language.EnumTypeExtensionNode` attribute), 33
  - `keys` (`graphql.language.EnumValueDefinitionNode` attribute), 33
  - `keys` (`graphql.language.EnumValueNode` attribute), 33
  - `keys` (`graphql.language.ExecutableDefinitionNode` attribute), 34
  - `keys` (`graphql.language.FieldDefinitionNode` attribute), 34
  - `keys` (`graphql.language.FieldNode` attribute), 35
  - `keys` (`graphql.language.FloatValueNode` attribute), 35
  - `keys` (`graphql.language.FragmentDefinitionNode` attribute), 35
  - `keys` (`graphql.language.FragmentSpreadNode` attribute), 36
  - `keys` (`graphql.language.InlineFragmentNode` attribute), 36
  - `keys` (`graphql.language.InputObjectTypeDefinitionNode` attribute), 36
  - `keys` (`graphql.language.InputObjectTypeExtensionNode` attribute), 37
  - `keys` (`graphql.language.InputValueDefinitionNode` attribute), 37

`keys` (`graphql.language.InterfaceTypeDefinitionNode` attribute), 38

`keys` (`graphql.language.InterfaceTypeExtensionNode` attribute), 38

`keys` (`graphql.language.IntValueNode` attribute), 37

`keys` (`graphql.language.ListTypeNode` attribute), 38

`keys` (`graphql.language.ListValueNode` attribute), 38

`keys` (`graphql.language.NamedTypeNode` attribute), 39

`keys` (`graphql.language.NameNode` attribute), 39

`keys` (`graphql.language.Node` attribute), 31

`keys` (`graphql.language.NonNullTypeNode` attribute), 39

`keys` (`graphql.language.NullValueNode` attribute), 39

`keys` (`graphql.language.ObjectFieldNode` attribute), 40

`keys` (`graphql.language.ObjectTypeDefinitionNode` attribute), 40

`keys` (`graphql.language.ObjectTypeExtensionNode` attribute), 40

`keys` (`graphql.language.ObjectValueNode` attribute), 41

`keys` (`graphql.language.OperationDefinitionNode` attribute), 41

`keys` (`graphql.language.OperationTypeDefinitionNode` attribute), 41

`keys` (`graphql.language.ScalarTypeDefinitionNode` attribute), 42

`keys` (`graphql.language.ScalarTypeExtensionNode` attribute), 42

`keys` (`graphql.language.SchemaDefinitionNode` attribute), 42

`keys` (`graphql.language.SchemaExtensionNode` attribute), 43

`keys` (`graphql.language.SelectionNode` attribute), 43

`keys` (`graphql.language.SelectionSetNode` attribute), 43

`keys` (`graphql.language.StringValueNode` attribute), 44

`keys` (`graphql.language.TypeDefinitionNode` attribute), 44

`keys` (`graphql.language.TypeExtensionNode` attribute), 44

`keys` (`graphql.language.TypeNode` attribute), 44

`keys` (`graphql.language.TypeSystemDefinitionNode` attribute), 45

`keys` (`graphql.language.UnionTypeDefinitionNode` attribute), 45

`keys` (`graphql.language.UnionTypeExtensionNode` attribute), 45

`keys` (`graphql.language.ValueNode` attribute), 46

`keys` (`graphql.language.VariableDefinitionNode` attribute), 46

`keys` (`graphql.language.VariableNode` attribute), 46

`kind` (`graphql.language.ArgumentNode` attribute), 31

`kind` (`graphql.language.BooleanValueNode` attribute), 31

`kind` (`graphql.language.DefinitionNode` attribute), 31

`kind` (`graphql.language.DirectiveDefinitionNode` attribute), 32

`kind` (`graphql.language.DirectiveNode` attribute), 32

`kind` (`graphql.language.DocumentNode` attribute), 32

`kind` (`graphql.language.EnumTypeDefinitionNode` attribute), 33

`kind` (`graphql.language.EnumTypeExtensionNode` attribute), 33

`kind` (`graphql.language.EnumValueDefinitionNode` attribute), 33

`kind` (`graphql.language.EnumValueNode` attribute), 33

`kind` (`graphql.language.ExecutableDefinitionNode` attribute), 34

`kind` (`graphql.language.FieldDefinitionNode` attribute), 34

`kind` (`graphql.language.FieldNode` attribute), 35

`kind` (`graphql.language.FloatValueNode` attribute), 35

`kind` (`graphql.language.FragmentDefinitionNode` attribute), 35

`kind` (`graphql.language.FragmentSpreadNode` attribute), 36

`kind` (`graphql.language.InlineFragmentNode` attribute), 36

`kind` (`graphql.language.InputObjectTypeDefinitionNode` attribute), 36

`kind` (`graphql.language.InputObjectTypeExtensionNode` attribute), 37

`kind` (`graphql.language.InputValueDefinitionNode` attribute), 37

`kind` (`graphql.language.InterfaceTypeDefinitionNode` attribute), 38

`kind` (`graphql.language.InterfaceTypeExtensionNode` attribute), 38

`kind` (`graphql.language.IntValueNode` attribute), 37

`kind` (`graphql.language.ListTypeNode` attribute), 38

`kind` (`graphql.language.ListValueNode` attribute), 38

`kind` (`graphql.language.NamedTypeNode` attribute), 39

`kind` (`graphql.language.NameNode` attribute), 39

`kind` (`graphql.language.Node` attribute), 31

`kind` (`graphql.language.NonNullTypeNode` attribute), 39

`kind` (`graphql.language.NullValueNode` attribute), 39

`kind` (`graphql.language.ObjectFieldNode` attribute), 40

`kind` (`graphql.language.ObjectTypeDefinitionNode` attribute), 40

`kind` (`graphql.language.ObjectTypeExtensionNode` attribute), 40

`kind` (`graphql.language.ObjectValueNode` attribute), 41

`kind` (`graphql.language.OperationDefinitionNode` attribute), 41

`kind` (`graphql.language.OperationTypeDefinitionNode` attribute), 42

`kind` (`graphql.language.ScalarTypeDefinitionNode` attribute), 42

`kind` (`graphql.language.ScalarTypeExtensionNode` attribute), 42

`kind` (`graphql.language.SchemaDefinitionNode` attribute), 42



- `kind` (`graphql.language.SchemaExtensionNode` attribute), 43
- `kind` (`graphql.language.SelectionNode` attribute), 43
- `kind` (`graphql.language.SelectionSetNode` attribute), 43
- `kind` (`graphql.language.StringValueNode` attribute), 44
- `kind` (`graphql.language.Token` attribute), 49
- `kind` (`graphql.language.TypeDefinitionNode` attribute), 44
- `kind` (`graphql.language.TypeExtensionNode` attribute), 44
- `kind` (`graphql.language.TypeNode` attribute), 44
- `kind` (`graphql.language.TypeSystemDefinitionNode` attribute), 45
- `kind` (`graphql.language.UnionTypeDefinitionNode` attribute), 45
- `kind` (`graphql.language.UnionTypeExtensionNode` attribute), 45
- `kind` (`graphql.language.ValueNode` attribute), 46
- `kind` (`graphql.language.VariableDefinitionNode` attribute), 46
- `kind` (`graphql.language.VariableNode` attribute), 46
- `KnownArgumentNamesRule` (class in `graphql.validation`), 95
- `KnownDirectivesRule` (class in `graphql.validation`), 96
- `KnownFragmentNamesRule` (class in `graphql.validation`), 96
- `KnownTypeNamesRule` (class in `graphql.validation`), 97
- L**
- `leave()` (`graphql.language.ParallelVisitor` method), 54
- `leave()` (`graphql.utilities.TypeInfo` method), 85
- `leave()` (`graphql.utilities.TypeInfoVisitor` method), 86
- `leave_argument()` (`graphql.utilities.TypeInfo` method), 85
- `leave_directive()` (`graphql.utilities.TypeInfo` method), 85
- `leave_directive()` (`graphql.validation.ProvidedRequiredArgumentsRule` method), 102
- `leave_document()` (`graphql.validation.NoUnusedFragmentsRule` method), 100
- `leave_enum_value()` (`graphql.utilities.TypeInfo` method), 85
- `leave_field()` (`graphql.utilities.TypeInfo` method), 85
- `leave_field()` (`graphql.validation.ProvidedRequiredArgumentsRule` method), 102
- `leave_fragment_definition()` (`graphql.utilities.TypeInfo` method), 85
- `leave_inline_fragment()` (`graphql.utilities.TypeInfo` method), 85
- `leave_list_value()` (`graphql.utilities.TypeInfo` method), 85
- `leave_object_field()` (`graphql.utilities.TypeInfo` method), 85
- `leave_object_value()` (`graphql.validation.UniqueInputFieldNamesRule` method), 105
- `leave_operation_definition()` (`graphql.utilities.TypeInfo` method), 85
- `leave_operation_definition()` (`graphql.validation.NoUndefinedVariablesRule` method), 99
- `leave_operation_definition()` (`graphql.validation.NoUnusedVariablesRule` method), 100
- `leave_operation_definition()` (`graphql.validation.VariablesInAllowedPositionRule` method), 108
- `leave_selection_set()` (`graphql.utilities.TypeInfo` method), 85
- `leave_variable_definition()` (`graphql.utilities.TypeInfo` method), 85
- `Lexer` (class in `graphql.language`), 48
- `lexicographic_sort_schema()` (in module `graphql.utilities`), 81
- `line` (`graphql.language.SourceLocation` property), 50
- `line` (`graphql.language.Token` attribute), 49
- `LIST` (`graphql.type.TypeKind` attribute), 76
- `ListTypeNode` (class in `graphql.language`), 38
- `ListValueNode` (class in `graphql.language`), 38
- `loc` (`graphql.language.ArgumentNode` attribute), 31
- `loc` (`graphql.language.BooleanValueNode` attribute), 31
- `loc` (`graphql.language.DefinitionNode` attribute), 31
- `loc` (`graphql.language.DirectiveDefinitionNode` attribute), 32
- `loc` (`graphql.language.DirectiveNode` attribute), 32
- `loc` (`graphql.language.DocumentNode` attribute), 32
- `loc` (`graphql.language.EnumTypeDefinitionNode` attribute), 33
- `loc` (`graphql.language.EnumTypeExtensionNode` attribute), 33
- `loc` (`graphql.language.EnumValueDefinitionNode` attribute), 33
- `loc` (`graphql.language.EnumValueNode` attribute), 33
- `loc` (`graphql.language.ExecutableDefinitionNode` attribute), 34
- `loc` (`graphql.language.FieldDefinitionNode` attribute), 34
- `loc` (`graphql.language.FieldNode` attribute), 35
- `loc` (`graphql.language.FloatValueNode` attribute), 35
- `loc` (`graphql.language.FragmentDefinitionNode` attribute), 35
- `loc` (`graphql.language.FragmentSpreadNode` attribute), 36
- `loc` (`graphql.language.InlineFragmentNode` attribute), 36
- `loc` (`graphql.language.InputObjectTypeDefinitionNode` attribute), 36
- `loc` (`graphql.language.InputObjectTypeExtensionNode` attribute), 36

attribute), 37  
 loc (graphql.language.InputValueDefinitionNode attribute), 37  
 loc (graphql.language.InterfaceTypeDefinitionNode attribute), 38  
 loc (graphql.language.InterfaceTypeExtensionNode attribute), 38  
 loc (graphql.language.IntValueNode attribute), 37  
 loc (graphql.language.ListTypeNode attribute), 38  
 loc (graphql.language.ListValueNode attribute), 39  
 loc (graphql.language.NamedTypeNode attribute), 39  
 loc (graphql.language.NameNode attribute), 39  
 loc (graphql.language.Node attribute), 31  
 loc (graphql.language.NonNullTypeNode attribute), 39  
 loc (graphql.language.NullValueNode attribute), 39  
 loc (graphql.language.ObjectFieldNode attribute), 40  
 loc (graphql.language.ObjectTypeDefinitionNode attribute), 40  
 loc (graphql.language.ObjectTypeExtensionNode attribute), 40  
 loc (graphql.language.ObjectValueNode attribute), 41  
 loc (graphql.language.OperationDefinitionNode attribute), 41  
 loc (graphql.language.OperationTypeDefinitionNode attribute), 42  
 loc (graphql.language.ScalarTypeDefinitionNode attribute), 42  
 loc (graphql.language.ScalarTypeExtensionNode attribute), 42  
 loc (graphql.language.SchemaDefinitionNode attribute), 43  
 loc (graphql.language.SchemaExtensionNode attribute), 43  
 loc (graphql.language.SelectionNode attribute), 43  
 loc (graphql.language.SelectionSetNode attribute), 43  
 loc (graphql.language.StringValueNode attribute), 44  
 loc (graphql.language.TypeDefinitionNode attribute), 44  
 loc (graphql.language.TypeExtensionNode attribute), 44  
 loc (graphql.language.TypeNode attribute), 44  
 loc (graphql.language.TypeSystemDefinitionNode attribute), 45  
 loc (graphql.language.UnionTypeDefinitionNode attribute), 45  
 loc (graphql.language.UnionTypeExtensionNode attribute), 45  
 loc (graphql.language.ValueNode attribute), 46  
 loc (graphql.language.VariableDefinitionNode attribute), 46  
 loc (graphql.language.VariableNode attribute), 46  
 located\_error() (in module graphql.error), 23  
 Location (class in graphql.language), 30  
 location\_offset (graphql.language.Source attribute), 52  
 locations (graphql.error.GraphQLError attribute), 21

locations (graphql.error.GraphQLError attribute), 22  
 locations (graphql.language.DirectiveDefinitionNode attribute), 32  
 locations (graphql.type.GraphQLDirective attribute), 75  
 LoneAnonymousOperationRule (class in graphql.validation), 97  
 LoneSchemaDefinitionRule (class in graphql.validation), 108  
 lookahead() (graphql.language.Lexer method), 48

## M

MapAsyncIterator (class in graphql.subscription), 58  
 message (graphql.error.GraphQLError attribute), 21  
 message (graphql.error.GraphQLError attribute), 22  
 middleware\_manager (graphql.execution.ExecutionContext attribute), 27  
 MiddlewareManager (class in graphql.execution), 28  
 middlewares (graphql.execution.MiddlewareManager attribute), 28  
 module  
   graphql, 19  
   graphql.error, 21  
   graphql.execution, 23  
   graphql.language, 30  
   graphql.pyutils, 54  
   graphql.subscription, 57  
   graphql.type, 58  
   graphql.utilities, 80  
   graphql.validation, 90  
   graphql.validation.rules, 94  
 MUTATION (graphql.language.DirectiveLocation attribute), 47  
 MUTATION (graphql.language.OperationType attribute), 41  
 mutation\_type (graphql.type.GraphQLSchema attribute), 79

## N

name (graphql.language.ArgumentNode attribute), 31  
 name (graphql.language.DirectiveDefinitionNode attribute), 32  
 name (graphql.language.DirectiveNode attribute), 32  
 name (graphql.language.EnumTypeDefinitionNode attribute), 33  
 name (graphql.language.EnumTypeExtensionNode attribute), 33  
 name (graphql.language.EnumValueDefinitionNode attribute), 33  
 name (graphql.language.ExecutableDefinitionNode attribute), 34

- `name` (`graphql.language.FieldDefinitionNode` attribute), 34
  - `name` (`graphql.language.FieldNode` attribute), 35
  - `name` (`graphql.language.FragmentDefinitionNode` attribute), 35
  - `name` (`graphql.language.FragmentSpreadNode` attribute), 36
  - `name` (`graphql.language.InputObjectTypeDefinitionNode` attribute), 36
  - `name` (`graphql.language.InputObjectTypeExtensionNode` attribute), 37
  - `name` (`graphql.language.InputValueDefinitionNode` attribute), 37
  - `name` (`graphql.language.InterfaceTypeDefinitionNode` attribute), 38
  - `name` (`graphql.language.InterfaceTypeExtensionNode` attribute), 38
  - `name` (`graphql.language.NamedTypeNode` attribute), 39
  - `name` (`graphql.language.ObjectFieldNode` attribute), 40
  - `name` (`graphql.language.ObjectTypeDefinitionNode` attribute), 40
  - `name` (`graphql.language.ObjectTypeExtensionNode` attribute), 41
  - `name` (`graphql.language.OperationDefinitionNode` attribute), 41
  - `name` (`graphql.language.ScalarTypeDefinitionNode` attribute), 42
  - `name` (`graphql.language.ScalarTypeExtensionNode` attribute), 42
  - `name` (`graphql.language.Source` attribute), 52
  - `NAME` (`graphql.language.TokenKind` attribute), 49
  - `name` (`graphql.language.TypeDefinitionNode` attribute), 44
  - `name` (`graphql.language.TypeExtensionNode` attribute), 44
  - `name` (`graphql.language.UnionTypeDefinitionNode` attribute), 45
  - `name` (`graphql.language.UnionTypeExtensionNode` attribute), 45
  - `name` (`graphql.language.VariableNode` attribute), 46
  - `name` (`graphql.type.GraphQLDirective` attribute), 75
  - `name` (`graphql.type.GraphQLEnumType` attribute), 61
  - `name` (`graphql.type.GraphQLInputObjectType` attribute), 63
  - `name` (`graphql.type.GraphQLInterfaceType` attribute), 64
  - `name` (`graphql.type.GraphQLNamedType` attribute), 72
  - `name` (`graphql.type.GraphQLObjectType` attribute), 65
  - `name` (`graphql.type.GraphQLScalarType` attribute), 66
  - `name` (`graphql.type.GraphQLUnionType` attribute), 68
  - `NamedTypeNode` (class in `graphql.language`), 39
  - `NameNode` (class in `graphql.language`), 39
  - `next` (`graphql.language.Token` attribute), 50
  - `Node` (class in `graphql.language`), 30
  - `nodes` (`graphql.error.GraphQLError` attribute), 22
  - `nodes` (`graphql.error.GraphQLError` attribute), 22
  - `NoFragmentCyclesRule` (class in `graphql.validation`), 98
  - `NON_NULL` (`graphql.type.TypeKind` attribute), 76
  - `NonNullTypeNode` (class in `graphql.language`), 39
  - `NoUndefinedVariablesRule` (class in `graphql.validation`), 98
  - `NoUnusedFragmentsRule` (class in `graphql.validation`), 99
  - `NoUnusedVariablesRule` (class in `graphql.validation`), 100
  - `NullValueNode` (class in `graphql.language`), 39
- ## O
- `OBJECT` (`graphql.language.DirectiveLocation` attribute), 47
  - `OBJECT` (`graphql.type.TypeKind` attribute), 76
  - `ObjectFieldNode` (class in `graphql.language`), 39
  - `ObjectTypeDefinitionNode` (class in `graphql.language`), 40
  - `ObjectTypeExtensionNode` (class in `graphql.language`), 40
  - `ObjectValueNode` (class in `graphql.language`), 41
  - `of_type` (`graphql.type.GraphQLList` attribute), 68
  - `of_type` (`graphql.type.GraphQLNonNull` attribute), 69
  - `of_type` (`graphql.type.GraphQLWrappingType` attribute), 73
  - `on_error()` (`graphql.validation.ASTValidationContext` method), 91
  - `on_error()` (`graphql.validation.SDLValidationContext` method), 92
  - `on_error()` (`graphql.validation.ValidationContext` method), 93
  - `operation` (`graphql.execution.ExecutionContext` attribute), 27
  - `operation` (`graphql.language.OperationDefinitionNode` attribute), 41
  - `operation` (`graphql.language.OperationTypeDefinitionNode` attribute), 42
  - `operation` (`graphql.type.GraphQLResolveInfo` property), 74
  - `operation_types` (`graphql.language.SchemaDefinitionNode` attribute), 43
  - `operation_types` (`graphql.language.SchemaExtensionNode` attribute), 43
  - `OperationDefinitionNode` (class in `graphql.language`), 41
  - `OperationType` (class in `graphql.language`), 41
  - `OperationTypeDefinitionNode` (class in `graphql.language`), 41
  - `OPTIONAL_ARG_ADDED` (`graphql.utilities.DangerousChangeType` attribute), 89
  - `OPTIONAL_INPUT_FIELD_ADDED` (`graphql.utilities.DangerousChangeType` attribute), 89

*attribute*), 89  
original\_error (*graphql.error.GraphQLError attribute*), 22  
original\_error (*graphql.error.GraphQLError attribute*), 22  
out\_name (*graphql.type.GraphQLArgument attribute*), 69  
out\_name (*graphql.type.GraphQLInputField attribute*), 71  
out\_type() (*graphql.type.GraphQLInputObjectType static method*), 63  
OverlappingFieldsCanBeMergedRule (*class in graphql.validation*), 100

## P

ParallelVisitor (*class in graphql.language*), 53  
PAREN\_L (*graphql.language.TokenKind attribute*), 49  
PAREN\_R (*graphql.language.TokenKind attribute*), 49  
parent\_type (*graphql.type.GraphQLResolveInfo property*), 74  
parse() (*in module graphql.language*), 50  
parse\_literal() (*graphql.type.GraphQLEnumType method*), 62  
parse\_literal() (*graphql.type.GraphQLScalarType method*), 66  
parse\_type() (*in module graphql.language*), 51  
parse\_value() (*graphql.type.GraphQLEnumType method*), 62  
parse\_value() (*graphql.type.GraphQLScalarType static method*), 66  
parse\_value() (*in module graphql.language*), 51  
Path (*class in graphql.pyutils*), 56  
path (*graphql.error.GraphQLError attribute*), 22  
path (*graphql.error.GraphQLError attribute*), 22  
path (*graphql.type.GraphQLResolveInfo property*), 74  
PIPE (*graphql.language.TokenKind attribute*), 49  
positions (*graphql.error.GraphQLError attribute*), 22  
positions (*graphql.error.GraphQLError attribute*), 23  
PossibleFragmentSpreadsRule (*class in graphql.validation*), 101  
PossibleTypeExtensionsRule (*class in graphql.validation*), 112  
prev (*graphql.language.Token attribute*), 50  
prev (*graphql.pyutils.Path property*), 56  
print\_ast() (*in module graphql.language*), 51  
print\_error() (*in module graphql.error*), 23  
print\_introspection\_schema() (*in module graphql.utilities*), 82  
print\_location() (*in module graphql.language*), 50  
print\_path\_list() (*in module graphql.pyutils*), 56  
print\_schema() (*in module graphql.utilities*), 82  
print\_source\_location() (*in module graphql.language*), 52

print\_type() (*in module graphql.utilities*), 82  
print\_value() (*in module graphql.utilities*), 82  
ProvidedRequiredArgumentsRule (*class in graphql.validation*), 101  
push\_value() (*graphql.pyutils.SimplePubSubIterator method*), 55

## Q

QUERY (*graphql.language.DirectiveLocation attribute*), 47  
QUERY (*graphql.language.OperationType attribute*), 41  
query\_type (*graphql.type.GraphQLSchema attribute*), 79

## R

read\_block\_string() (*graphql.language.Lexer method*), 48  
read\_comment() (*graphql.language.Lexer method*), 48  
read\_digits() (*graphql.language.Lexer method*), 48  
read\_name() (*graphql.language.Lexer method*), 48  
read\_number() (*graphql.language.Lexer method*), 48  
read\_string() (*graphql.language.Lexer method*), 48  
read\_token() (*graphql.language.Lexer method*), 48  
register\_description() (*in module graphql.pyutils*), 55  
REMOVE (*graphql.language.ParallelVisitor attribute*), 53  
REMOVE (*graphql.language.Visitor attribute*), 53  
REMOVE (*graphql.language.visitor.VisitorActionEnum attribute*), 54  
REMOVE (*graphql.utilities.TypeInfoVisitor attribute*), 85  
REMOVE (*graphql.validation.ASTValidationRule attribute*), 91  
REMOVE (*graphql.validation.ExecutableDefinitionsRule attribute*), 94  
REMOVE (*graphql.validation.FieldsOnCorrectTypeRule attribute*), 94  
REMOVE (*graphql.validation.FragmentsOnCompositeTypesRule attribute*), 95  
REMOVE (*graphql.validation.KnownArgumentNamesRule attribute*), 95  
REMOVE (*graphql.validation.KnownDirectivesRule attribute*), 96  
REMOVE (*graphql.validation.KnownFragmentNamesRule attribute*), 97  
REMOVE (*graphql.validation.KnownTypeNamesRule attribute*), 97  
REMOVE (*graphql.validation.LoneAnonymousOperationRule attribute*), 98  
REMOVE (*graphql.validation.LoneSchemaDefinitionRule attribute*), 108  
REMOVE (*graphql.validation.NoFragmentCyclesRule attribute*), 98  
REMOVE (*graphql.validation.NoUndefinedVariablesRule attribute*), 99



REMOVE ( <i>graphql.validation.NoUnusedFragmentsRule</i> attribute), 99	report_error() ( <i>graphql.validation.ExecutableDefinitionsRule</i> method), 94
REMOVE ( <i>graphql.validation.NoUnusedVariablesRule</i> attribute), 100	report_error() ( <i>graphql.validation.FieldsOnCorrectTypeRule</i> method), 95
REMOVE ( <i>graphql.validation.OverlappingFieldsCanBeMergedRule</i> attribute), 100	report_error() ( <i>graphql.validation.FragmentsOnCompositeTypesRule</i> method), 95
REMOVE ( <i>graphql.validation.PossibleFragmentSpreadsRule</i> attribute), 101	report_error() ( <i>graphql.validation.KnownArgumentNamesRule</i> method), 96
REMOVE ( <i>graphql.validation.PossibleTypeExtensionsRule</i> attribute), 112	report_error() ( <i>graphql.validation.KnownDirectivesRule</i> method), 96
REMOVE ( <i>graphql.validation.ProvidedRequiredArgumentsRule</i> attribute), 102	report_error() ( <i>graphql.validation.KnownFragmentNamesRule</i> method), 97
REMOVE ( <i>graphql.validation.ScalarLeafsRule</i> attribute), 102	report_error() ( <i>graphql.validation.KnownTypeNamesRule</i> method), 97
REMOVE ( <i>graphql.validation.SDLValidationRule</i> attribute), 92	report_error() ( <i>graphql.validation.LoneAnonymousOperationRule</i> method), 98
REMOVE ( <i>graphql.validation.SingleFieldSubscriptionsRule</i> attribute), 103	report_error() ( <i>graphql.validation.LoneSchemaDefinitionRule</i> method), 108
REMOVE ( <i>graphql.validation.UniqueArgumentNamesRule</i> attribute), 103	report_error() ( <i>graphql.validation.NoFragmentCyclesRule</i> method), 98
REMOVE ( <i>graphql.validation.UniqueDirectiveNamesRule</i> attribute), 111	report_error() ( <i>graphql.validation.NoUndefinedVariablesRule</i> method), 99
REMOVE ( <i>graphql.validation.UniqueDirectivesPerLocationRule</i> attribute), 104	report_error() ( <i>graphql.validation.NoUnusedFragmentsRule</i> method), 100
REMOVE ( <i>graphql.validation.UniqueEnumValueNamesRule</i> attribute), 110	report_error() ( <i>graphql.validation.NoUnusedVariablesRule</i> method), 100
REMOVE ( <i>graphql.validation.UniqueFieldDefinitionNamesRule</i> attribute), 110	report_error() ( <i>graphql.validation.OverlappingFieldsCanBeMergedRule</i> method), 101
REMOVE ( <i>graphql.validation.UniqueFragmentNamesRule</i> attribute), 104	report_error() ( <i>graphql.validation.PossibleFragmentSpreadsRule</i> method), 101
REMOVE ( <i>graphql.validation.UniqueInputFieldNamesRule</i> attribute), 105	report_error() ( <i>graphql.validation.PossibleTypeExtensionsRule</i> method), 112
REMOVE ( <i>graphql.validation.UniqueOperationNamesRule</i> attribute), 105	report_error() ( <i>graphql.validation.ProvidedRequiredArgumentsRule</i> method), 102
REMOVE ( <i>graphql.validation.UniqueOperationTypesRule</i> attribute), 109	report_error() ( <i>graphql.validation.ScalarLeafsRule</i> method), 102
REMOVE ( <i>graphql.validation.UniqueTypeNamesRule</i> attribute), 109	report_error() ( <i>graphql.validation.SDLValidationContext</i> method), 92
REMOVE ( <i>graphql.validation.UniqueVariableNamesRule</i> attribute), 106	report_error() ( <i>graphql.validation.SDLValidationRule</i> method), 92
REMOVE ( <i>graphql.validation.ValidationRule</i> attribute), 93	report_error() ( <i>graphql.validation.SingleFieldSubscriptionsRule</i> method), 103
REMOVE ( <i>graphql.validation.ValuesOfCorrectTypeRule</i> attribute), 106	report_error() ( <i>graphql.validation.UniqueArgumentNamesRule</i> method), 103
REMOVE ( <i>graphql.validation.VariablesAreInputTypesRule</i> attribute), 107	report_error() ( <i>graphql.validation.UniqueDirectiveNamesRule</i> method), 111
REMOVE ( <i>graphql.validation.VariablesInAllowedPositionRule</i> attribute), 108	report_error() ( <i>graphql.validation.UniqueDirectivesPerLocationRule</i> method), 104
REMOVE ( <i>in module graphql.language</i> ), 54	report_error() ( <i>graphql.validation.UniqueEnumValueNamesRule</i> method), 110
repeatable ( <i>graphql.language.DirectiveDefinitionNode</i> attribute), 32	report_error() ( <i>graphql.validation.UniqueFieldDefinitionNamesRule</i> method), 111
report_error() ( <i>graphql.validation.ASTValidationContext</i> method), 91	report_error() ( <i>graphql.validation.UniqueFragmentNamesRule</i> method), 104
report_error() ( <i>graphql.validation.ASTValidationRule</i> method), 91	

[report\\_error\(\)](#) (*graphql.validation.UniqueInputFieldNamesRule* attribute), 105  
[report\\_error\(\)](#) (*graphql.validation.UniqueOperationNamesRule* attribute), 105  
[report\\_error\(\)](#) (*graphql.validation.UniqueOperationTypesRule* attribute), 109  
[report\\_error\(\)](#) (*graphql.validation.UniqueTypeNamesRule* attribute), 110  
[report\\_error\(\)](#) (*graphql.validation.UniqueVariableNamesRule* attribute), 106  
[report\\_error\(\)](#) (*graphql.validation.ValidationContext* attribute), 93  
[report\\_error\(\)](#) (*graphql.validation.ValidationRule* attribute), 94  
[report\\_error\(\)](#) (*graphql.validation.ValuesOfCorrectTypeRule* attribute), 107  
[report\\_error\(\)](#) (*graphql.validation.VariablesAreInputTypesRule* attribute), 107  
[report\\_error\(\)](#) (*graphql.validation.VariablesInAllowedPositionRule* attribute), 108  
[REQUIRED\\_ARG\\_ADDED](#) (*graphql.utilities.BreakingChangeType* attribute), 89  
[REQUIRED\\_DIRECTIVE\\_ARG\\_ADDED](#) (*graphql.utilities.BreakingChangeType* attribute), 89  
[REQUIRED\\_INPUT\\_FIELD\\_ADDED](#) (*graphql.utilities.BreakingChangeType* attribute), 89  
[resolve](#) (*graphql.type.GraphQLField* attribute), 71  
[resolve\\_field\(\)](#) (*graphql.execution.ExecutionContext* attribute), 27  
[resolve\\_type](#) (*graphql.type.GraphQLInterfaceType* attribute), 64  
[resolve\\_type](#) (*graphql.type.GraphQLUnionType* attribute), 68  
[return\\_type](#) (*graphql.type.GraphQLResolveInfo* property), 74  
[root\\_value](#) (*graphql.execution.ExecutionContext* attribute), 27  
[root\\_value](#) (*graphql.type.GraphQLResolveInfo* property), 74

## S

[SCALAR](#) (*graphql.language.DirectiveLocation* attribute), 47  
[SCALAR](#) (*graphql.type.TypeKind* attribute), 76  
[ScalarLeafsRule](#) (class in *graphql.validation*), 102  
[ScalarTypeDefinitionNode](#) (class in *graphql.language*), 42  
[ScalarTypeExtensionNode](#) (class in *graphql.language*), 42  
[schema](#) (*graphql.execution.ExecutionContext* attribute), 27  
[schema](#) (*graphql.language.DirectiveLocation* attribute), 47  
[schema](#) (*graphql.type.GraphQLResolveInfo* property), 74  
[schema](#) (*graphql.validation.SDLValidationContext* attribute), 92  
[schema](#) (*graphql.validation.ValidationContext* attribute), 93  
[SchemaDefinitionNode](#) (class in *graphql.language*), 42  
[SchemaExtensionNode](#) (class in *graphql.language*), 43  
[SchemaMetaFieldDef](#) (in module *graphql.type*), 76  
[SDLValidationContext](#) (class in *graphql.validation*), 91  
[SDLValidationRule](#) (class in *graphql.validation*), 92  
[selection\\_set](#) (*graphql.language.ExecutableDefinitionNode* attribute), 34  
[selection\\_set](#) (*graphql.language.FieldNode* attribute), 35  
[selection\\_set](#) (*graphql.language.FragmentDefinitionNode* attribute), 35  
[selection\\_set](#) (*graphql.language.InlineFragmentNode* attribute), 35  
[selection\\_set](#) (*graphql.language.OperationDefinitionNode* attribute), 41  
[SelectionNode](#) (class in *graphql.language*), 43  
[selections](#) (*graphql.language.SelectionSetNode* attribute), 43  
[SelectionSetNode](#) (class in *graphql.language*), 43  
[separate\\_operations\(\)](#) (in module *graphql.utilities*), 86  
[serialize\(\)](#) (*graphql.type.GraphQLEnumType* attribute), 62  
[serialize\(\)](#) (*graphql.type.GraphQLScalarType* attribute), 66  
[should\\_include\\_node\(\)](#) (*graphql.execution.ExecutionContext* attribute), 27  
[SimplePubSub](#) (class in *graphql.pyutils*), 55  
[SimplePubSubIterator](#) (class in *graphql.pyutils*), 55  
[SingleFieldSubscriptionsRule](#) (class in *graphql.validation*), 102  
[SKIP](#) (*graphql.language.ParallelVisitor* attribute), 53  
[SKIP](#) (*graphql.language.Visitor* attribute), 53  
[SKIP](#) (*graphql.language.visitor.VisitorActionEnum* attribute), 54  
[SKIP](#) (*graphql.utilities.TypeInfoVisitor* attribute), 85  
[SKIP](#) (*graphql.validation.ASTValidationRule* attribute), 91  
[SKIP](#) (*graphql.validation.ExecutableDefinitionsRule* attribute), 94  
[SKIP](#) (*graphql.validation.FieldsOnCorrectTypeRule* attribute), 95  
[SKIP](#) (*graphql.validation.FragmentsOnCompositeTypesRule* attribute), 95  
[SKIP](#) (*graphql.validation.KnownArgumentNamesRule* attribute), 95

- tribute), 96
  - SKIP (*graphql.validation.KnownDirectivesRule* attribute), 96
  - SKIP (*graphql.validation.KnownFragmentNamesRule* attribute), 97
  - SKIP (*graphql.validation.KnownTypeNamesRule* attribute), 97
  - SKIP (*graphql.validation.LoneAnonymousOperationRule* attribute), 98
  - SKIP (*graphql.validation.LoneSchemaDefinitionRule* attribute), 108
  - SKIP (*graphql.validation.NoFragmentCyclesRule* attribute), 98
  - SKIP (*graphql.validation.NoUndefinedVariablesRule* attribute), 99
  - SKIP (*graphql.validation.NoUnusedFragmentsRule* attribute), 99
  - SKIP (*graphql.validation.NoUnusedVariablesRule* attribute), 100
  - SKIP (*graphql.validation.OverlappingFieldsCanBeMergedRule* attribute), 100
  - SKIP (*graphql.validation.PossibleFragmentSpreadsRule* attribute), 101
  - SKIP (*graphql.validation.PossibleTypeExtensionsRule* attribute), 112
  - SKIP (*graphql.validation.ProvidedRequiredArgumentsRule* attribute), 102
  - SKIP (*graphql.validation.ScalarLeafsRule* attribute), 102
  - SKIP (*graphql.validation.SDLValidationRule* attribute), 92
  - SKIP (*graphql.validation.SingleFieldSubscriptionsRule* attribute), 103
  - SKIP (*graphql.validation.UniqueArgumentNamesRule* attribute), 103
  - SKIP (*graphql.validation.UniqueDirectiveNamesRule* attribute), 111
  - SKIP (*graphql.validation.UniqueDirectivesPerLocationRule* attribute), 104
  - SKIP (*graphql.validation.UniqueEnumValueNamesRule* attribute), 110
  - SKIP (*graphql.validation.UniqueFieldDefinitionNamesRule* attribute), 111
  - SKIP (*graphql.validation.UniqueFragmentNamesRule* attribute), 104
  - SKIP (*graphql.validation.UniqueInputFieldNamesRule* attribute), 105
  - SKIP (*graphql.validation.UniqueOperationNamesRule* attribute), 105
  - SKIP (*graphql.validation.UniqueOperationTypesRule* attribute), 109
  - SKIP (*graphql.validation.UniqueTypeNamesRule* attribute), 109
  - SKIP (*graphql.validation.UniqueVariableNamesRule* attribute), 106
  - SKIP (*graphql.validation.ValidationRule* attribute), 93
  - SKIP (*graphql.validation.ValuesOfCorrectTypeRule* attribute), 106
  - SKIP (*graphql.validation.VariablesAreInputTypesRule* attribute), 107
  - SKIP (*graphql.validation.VariablesInAllowedPositionRule* attribute), 108
  - SKIP (in module *graphql.language*), 54
  - snake\_to\_camel() (in module *graphql.pyutils*), 54
  - SOF (*graphql.language.TokenKind* attribute), 49
  - Source (class in *graphql.language*), 51
  - source (*graphql.error.GraphQLError* attribute), 22
  - source (*graphql.error.GraphQLSyntaxError* attribute), 23
  - source (*graphql.language.Location* attribute), 30
  - SourceLocation (class in *graphql.language*), 50
  - specified\_by\_url (*graphql.type.GraphQLScalarType* attribute), 67
  - specified\_directives (in module *graphql.type*), 75
  - specified\_rules (in module *graphql.validation*), 94
  - SPREAD (*graphql.language.TokenKind* attribute), 49
  - start (*graphql.language.Location* attribute), 30
  - start (*graphql.language.Token* attribute), 50
  - start\_token (*graphql.language.Location* attribute), 30
  - STRING (*graphql.language.TokenKind* attribute), 49
  - StringValueNode (class in *graphql.language*), 43
  - strip\_ignored\_characters() (in module *graphql.utilities*), 86
  - subscribe (*graphql.type.GraphQLField* attribute), 71
  - subscribe() (in module *graphql.subscription*), 57
  - subscribers (*graphql.pyutils.SimplePubSub* attribute), 55
  - SUBSCRIPTION (*graphql.language.DirectiveLocation* attribute), 47
  - SUBSCRIPTION (*graphql.language.OperationType* attribute), 41
  - subscription\_type (*graphql.type.GraphQLSchema* attribute), 79
  - suggestion\_list() (in module *graphql.pyutils*), 56
- ## T
- Thunk (in module *graphql.type*), 73
  - to\_kwargs() (*graphql.type.GraphQLArgument* method), 69
  - to\_kwargs() (*graphql.type.GraphQLDirective* method), 75
  - to\_kwargs() (*graphql.type.GraphQLEnumType* method), 62
  - to\_kwargs() (*graphql.type.GraphQLEnumValue* method), 70
  - to\_kwargs() (*graphql.type.GraphQLField* method), 71
  - to\_kwargs() (*graphql.type.GraphQLInputField* method), 71

- `to_kwargs()` (*graphql.type.GraphQLInputObjectType* method), 63
  - `to_kwargs()` (*graphql.type.GraphQLInterfaceType* method), 64
  - `to_kwargs()` (*graphql.type.GraphQLNamedType* method), 72
  - `to_kwargs()` (*graphql.type.GraphQLOBJECTType* method), 65
  - `to_kwargs()` (*graphql.type.GraphQLScalarType* method), 67
  - `to_kwargs()` (*graphql.type.GraphQLSchema* method), 79
  - `to_kwargs()` (*graphql.type.GraphQLUnionType* method), 68
  - `Token` (class in *graphql.language*), 49
  - `TokenKind` (class in *graphql.language*), 48
  - `type` (*graphql.language.FieldDefinitionNode* attribute), 34
  - `type` (*graphql.language.InputValueDefinitionNode* attribute), 37
  - `type` (*graphql.language.ListTypeNode* attribute), 38
  - `type` (*graphql.language.NonNullTypeNode* attribute), 39
  - `type` (*graphql.language.OperationTypeDefinitionNode* attribute), 42
  - `type` (*graphql.language.VariableDefinitionNode* attribute), 46
  - `type` (*graphql.type.GraphQLArgument* attribute), 69
  - `type` (*graphql.type.GraphQLField* attribute), 71
  - `type` (*graphql.type.GraphQLInputField* attribute), 72
  - `type` (*graphql.utilities.BreakingChange* property), 88
  - `type` (*graphql.utilities.DangerousChange* property), 89
  - `TYPE_ADDED_TO_UNION` (*graphql.utilities.DangerousChangeType* attribute), 90
  - `TYPE_CHANGED_KIND` (*graphql.utilities.BreakingChangeType* attribute), 89
  - `type_condition` (*graphql.language.FragmentDefinitionNode* attribute), 35
  - `type_condition` (*graphql.language.InlineFragmentNode* attribute), 36
  - `type_from_ast()` (in module *graphql.utilities*), 82
  - `type_map` (*graphql.type.GraphQLSchema* attribute), 79
  - `TYPE_REMOVED` (*graphql.utilities.BreakingChangeType* attribute), 89
  - `TYPE_REMOVED_FROM_UNION` (*graphql.utilities.BreakingChangeType* attribute), 89
  - `type_resolver` (*graphql.execution.ExecutionContext* attribute), 27
  - `TypeDefinitionNode` (class in *graphql.language*), 44
  - `TypeExtensionNode` (class in *graphql.language*), 44
  - `TypeInfo` (class in *graphql.utilities*), 83
  - `TypeInfoVisitor` (class in *graphql.utilities*), 85
  - `TypeKind` (class in *graphql.type*), 76
  - `TypeMetaFieldDef` (in module *graphql.type*), 76
  - `typename` (*graphql.pyutils.Path* property), 56
  - `TypeNameMetaFieldDef` (in module *graphql.type*), 76
  - `TypeNode` (class in *graphql.language*), 44
  - `types` (*graphql.language.UnionTypeDefinitionNode* attribute), 45
  - `types` (*graphql.language.UnionTypeExtensionNode* attribute), 45
  - `types` (*graphql.type.GraphQLUnionType* attribute), 68
  - `TypeDefintionNode` (class in *graphql.language*), 45
  - `TypeSystemExtensionNode` (in module *graphql.language*), 45
- ## U
- `Undefined` (in module *graphql.pyutils*), 57
  - `UNION` (*graphql.language.DirectiveLocation* attribute), 47
  - `UNION` (*graphql.type.TypeKind* attribute), 76
  - `UnionTypeDefinitionNode` (class in *graphql.language*), 45
  - `UnionTypeExtensionNode` (class in *graphql.language*), 45
  - `UniqueArgumentNamesRule` (class in *graphql.validation*), 103
  - `UniqueDirectiveNamesRule` (class in *graphql.validation*), 111
  - `UniqueDirectivesPerLocationRule` (class in *graphql.validation*), 103
  - `UniqueEnumValueNamesRule` (class in *graphql.validation*), 110
  - `UniqueFieldDefinitionNamesRule` (class in *graphql.validation*), 110
  - `UniqueFragmentNamesRule` (class in *graphql.validation*), 104
  - `UniqueInputFieldNamesRule` (class in *graphql.validation*), 104
  - `UniqueOperationNamesRule` (class in *graphql.validation*), 105
  - `UniqueOperationTypesRule` (class in *graphql.validation*), 108
  - `UniqueTypeNamesRule` (class in *graphql.validation*), 109
  - `UniqueVariableNamesRule` (class in *graphql.validation*), 106
  - `unregister_description()` (in module *graphql.pyutils*), 55
- ## V
- `validate()` (in module *graphql.validation*), 90
  - `validate_schema()` (in module *graphql.type*), 79
  - `validation_errors` (*graphql.type.GraphQLSchema* property), 79
  - `ValidationContext` (class in *graphql.validation*), 92



[ValidationRule](#) (class in [graphql.validation](#)), 93  
[value](#) ([graphql.language.ArgumentNode](#) attribute), 31  
[value](#) ([graphql.language.BooleanValueNode](#) attribute), 31  
[value](#) ([graphql.language.EnumValueNode](#) attribute), 34  
[value](#) ([graphql.language.FloatValueNode](#) attribute), 35  
[value](#) ([graphql.language.IntValueNode](#) attribute), 37  
[value](#) ([graphql.language.NameNode](#) attribute), 39  
[value](#) ([graphql.language.ObjectFieldNode](#) attribute), 40  
[value](#) ([graphql.language.StringValueNode](#) attribute), 44  
[value](#) ([graphql.language.Token](#) attribute), 50  
[value](#) ([graphql.type.GraphQLEnumValue](#) attribute), 70  
[VALUE\\_ADDED\\_TO\\_ENUM](#)  
 ([graphql.utilities.DangerousChangeType](#) attribute), 90  
[value\\_from\\_ast\(\)](#) (in module [graphql.utilities](#)), 82  
[value\\_from\\_ast\\_untyped\(\)](#) (in module [graphql.utilities](#)), 83  
[VALUE\\_REMOVED\\_FROM\\_ENUM](#)  
 ([graphql.utilities.BreakingChangeType](#) attribute), 89  
[ValueNode](#) (class in [graphql.language](#)), 45  
[values](#) ([graphql.language.EnumTypeDefinitionNode](#) attribute), 33  
[values](#) ([graphql.language.EnumTypeExtensionNode](#) attribute), 33  
[values](#) ([graphql.language.ListValueNode](#) attribute), 39  
[values](#) ([graphql.type.GraphQLEnumType](#) attribute), 62  
[ValuesOfCorrectTypeRule](#) (class in [graphql.validation](#)), 106  
[variable](#) ([graphql.language.VariableDefinitionNode](#) attribute), 46  
[VARIABLE\\_DEFINITION](#)  
 ([graphql.language.DirectiveLocation](#) attribute), 47  
[variable\\_definitions](#)  
 ([graphql.language.ExecutableDefinitionNode](#) attribute), 34  
[variable\\_definitions](#)  
 ([graphql.language.FragmentDefinitionNode](#) attribute), 35  
[variable\\_definitions](#)  
 ([graphql.language.OperationDefinitionNode](#) attribute), 41  
[variable\\_values](#) ([graphql.execution.ExecutionContext](#) attribute), 27  
[variable\\_values](#) ([graphql.type.GraphQLResolveInfo](#) property), 74  
[VariableDefinitionNode](#) (class in [graphql.language](#)), 46  
[VariableNode](#) (class in [graphql.language](#)), 46  
[VariablesAreInputTypesRule](#) (class in [graphql.validation](#)), 107  
[VariablesInAllowedPositionRule](#) (class in [graphql.validation](#)), 107  
[visit\(\)](#) (in module [graphql.language](#)), 52  
[Visitor](#) (class in [graphql.language](#)), 52  
[VisitorActionEnum](#) (class in [graphql.language.visitor](#)), 54

## W

[with\\_traceback\(\)](#) ([graphql.error.GraphQLError](#) method), 22  
[with\\_traceback\(\)](#) ([graphql.error.GraphQLSyntaxError](#) method), 23