subgrounds Documentation

Release stable

Nov 01, 2022

CONTENTS:

Getting started

Installation

Subgrounds can be installed via `pip` with the following command: `pip install subgrounds`

```{eval-rst} .. Important:: Subgrounds requires ``python>=3.10`` ```

```{eval-rst} .. Note:: If you run into problems during installation, see :ref:`Set up an isolated environment <isolated_environment_setup>`. ```

Simple example ```python >>> from subgrounds import Subgrounds

Initialize Subgrounds >>> sg = Subgrounds()

Load a subgraph >>> aaveV2 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/aave/protocol-v2')

Create a SyntheticField on the Borrow entity type using Python arithmetic operators >>> aaveV2.Borrow.adjusted_amount = aaveV2.Borrow.amount / 10 ** aaveV2.Borrow.reserve.decimals

Create a partial FieldPath representing a selection on the borrows field with arguments >>> last10_borrows = aaveV2.Query.borrows(... orderBy=aaveV2.Borrow.timestamp, ... orderDirection='desc', ... first=10 ...)

Query FieldPaths and format the result into a Pandas DataFrame >>> sg.query_df([... last10_borrows.reserve.symbol, ... last10_borrows.timestamp, ... last10_borrows.adjusted_amount ...])

borrows_reserve_symbol borrows_timestamp borrows_adjusted_amount

0 USDT 1643300294 500000.000000 1 DAI 1643299575 6000.000000 2 USDT 1643298921 900000.000000 3 USDT 1643297685 500000.000000 4 USDC 1643296256 50000.000000 5 PAX 1643295342 4150.000000 6 USDT 1643294783 9000.000000 7 DAI 1643293451 45585.919063 8 UNI 1643289600 50000.000000 9 USDT 1643289117 14000.000000 ```

Subgrounds basics This page was written to provide an overview of the main concepts in Subgrounds and to serve as a quick reference for those concepts.

Subgrounds The *Subgrounds* class provides the toplevel Subgrounds API and most Subgrounds users will be using this class exclusively. This class is used to load (i.e.: introspect) GraphQL APIs (subgraph APIs or vanilla GraphQL APIs) as well as execute querying operations. Moreover, this class is meant to be used as a singleton, i.e.: initialized once and reused throughout a project.

The code cell below demonstrates how to initialize your *Subgrounds* object and load a GraphQL API. ```python >>> from subgrounds import Subgrounds

Initialize Subgrounds >>> sg = Subgrounds()

Load a subgraph using its API URL >>> aaveV2 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/aave/protocol-v2')

Load a vanilla GraphQL API using its API URL >>> snapshot = sg.load_api('https://hub.snapshot.org/graphql') ```

FieldPaths *FieldPaths* are the main building blocks used to construct Subgrounds queries. A *FieldPath* represents a selection path through a GraphQL schema starting from the root *Query* entity (see [The Query and Mutation types](https://graphql.org/learn/schema/#the-query-and-mutation-types)) all the way down to a scalar leaf.

FieldPaths are created by simply selecting attributes starting from the subgraph object returned by the *Subgrounds.load_subgraph* or *Subgrounds.load_api* methods: ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

Create a FieldPath >>> pools_tvl = uniswapV3.Query.pools.totalValueLockedUSD ```

In the example above, the *pools_tvl* variable is a *FieldPath* object representing the following GraphQL query: ```graphql query {

```
pools {
    totalValueLockedUSD
}
```

}

Note that *FieldPaths* don't have to be selected from root to leaf each time and partial *FieldPaths* can be reused: ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

Create a FieldPath >>> pools_tvl = uniswapV3.Query.pools.totalValueLockedUSD

Create a partial FieldPath >>> pools = uniswapV3.Query.pools

This FieldPath is equivalent to the *pools_tvl* FieldPath >>> pools_tvl2 = pools.totalValueLockedUSD ```

FieldPath arguments Field arguments can be specified via *FieldPaths* by "calling" the field in question: ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

Create a partial FieldPath of *pools* created after block 14720000 # ordered by their TVL in descending order >>> new_pools = uniswapV3.Query.pools(… orderBy=uniswapV3.Pool.totalValueLockedUSD, … orderDirection='desc', … where=[… uniswapV3.Pool.createdAtBlockNumber > 14720000 …] …)

Select the *totalValueLockedUSD* of the >>> pools_tvl = new_pools.totalValueLockedUSD ```

In the example above, the *pools_tvl FieldPath* represents the following GraphQL query: ```graphql query {

pools(
orderBy: totalValueLockedUSD, orderDirection: desc, where: {

createdAtBlockNumber_gt: 14720000

}

) {
totalValueLockedUSD

}

}

Notice that the values for the *orderBy* and *where* arguments are *FieldPath* themselves. This allows users to construct complex queries in pure Python by using the *Subgraph* object returned when loading an API. Note however that the *FieldPaths* used as argument values are *relative FieldPath*, i.e.: they do not start from the root *Query* entity, but rather start from a user defined entity type (in this case the *Pool* entity). It is important to make sure that the relative *FieldPath* used as values for the *orderBy* and *where* arguments match the entity type of the field on which the arguments are applied (in our example, the *pools* field is of type *Pool*). If this is not respected, a type error exception will be thrown.

Argument values can also be supplied in their "raw" form, without the use of relative *FieldPaths*: ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

This partial FieldPath is equivalant to the *new_pools* FieldPath # in the previous example >>> new_pools2 = uniswapV3.Query.pools(… orderBy='totalValueLockedUSD', … orderDirection='desc', … where={ … 'createdAtBlockNumber_gt': 14720000 … } …) ```

Querying data The *Subgrounds* class provides three methods for querying data: 1. *query* 2. *query_df* 3. *query_json*

All three methods take the same arguments (ignoring optional arguments), namely a list of *FieldPaths* and differ only in the way the returned data is formatted. This section will go over each method individually.

Subgrounds.query The *query* method returns the data in its simplest form which, depending on the *FieldPaths* given as argument, will be either: 1) a single value; 2) a list of values; or 3) a tuple containing single values or lists of values. It is important to consider the shape of the queried data (e.g.: single entities, list of entities…) as the shape of the returned data will depend on it.

```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

# 1) Single value: Query the quantity of WETH locked on Uniswap V3 >>> sg.query([ … uniswapV3.Query.token(id='0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2').totalValueLocked … ]) 505722.3421353012

# Partial FieldPath of top 10 most traded tokens >>> most_traded_tokens = uniswapV3.Query.tokens( … orderBy=uniswapV3.Token.volumeUSD, … orderDirection='desc', … first=10 … )

# 2) List of values: Query symbol of top 10 most traded tokens >>> sg.query([ … most_traded_tokens.symbol … ]) ['WETH', 'USDC', 'USDT', 'DAI', 'WBTC', 'FEI', 'UST', 'APE', 'LOOKS', 'HEX']

# 3) Tuple of values: Query symbol and TVL of top 10 most traded tokens >>> sg.query([ … most_traded_tokens.symbol, … most_traded_tokens.totalValueLocked … ]) (['WETH', 'USDC', 'USDT', 'DAI', 'WBTC', 'FEI', 'UST', 'APE', 'LOOKS', 'HEX'],

**[506128.5352471704,**
774159914.44222, 353261712.912211, 384420834.9304443, 11069.25245941, 42232365.752994545, 8030872.929855888, 2604424.599749661, 11610722.749456603, 131046877.18519919])

```

Subgrounds.query_df Subgrounds provides a simple way to query subgraph data directly into a pandas *DataFrame* via the *query_df* method. Just like the *query* method, *query_df* takes as argument a list of *FieldPaths* and returns one or more *DataFrames* depending on the shape of the queried data. *query_df* will attempt to flatten all the data to a single *DataFrame* (effectively mimicking the SQL *JOIN* operation) but when that is not possible, two or more *DataFrames* will be returned.

Example with single DataFrame returned ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

Select top 10 pools by cummulative volume >>> top_10_pools = uniswapV3.Query.pools(… orderBy=uniswapV3.Pool.volumeUSD, … orderDirection='desc', … first=10 …)

Query data flattened to a single DataFrame >>> sg.query_df([… top_10_pools.id, … top_10_pools.token0.symbol, … top_10_pools.token1.symbol, …])

pools_id pools_token0_symbol pools_token1_symbol

0 0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640 USDC WETH 1 0x8ad599c3a0ff1de082011efddc58f1908eb6e6d8 USDC WETH 2 0x11b815efb8f581194ae79006d24e0d814b7697f6 WETH USDT 3 0x4e68ccd3e89f51c3074ca5072bbac773960dfa36 WETH USDT 4 0x60594a405d53811d3bc4766596efd80fd545a270 DAI WETH 5 0x4585fe77225b41b697c938b018e2ac67ac5a20c0 WBTC WETH 6 0x3416cf6c708da44db2624d63ea0aaef7113527c6 USDC USDT 7 0xcbcdf9626bc03e24f779434178a73a0b4bad62ed WBTC WETH 8 0xc2e9f25be6257c210d7adf0d4cd6e3e881ba25f8 DAI WETH 9 0x7858e59e0c01ea06df3af3d20ac7b0003275d4bf USDC USDT ``` #### Example with multiple DataFrames returned ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

Select WETH-USDC pool >>> weth_usdc = uniswapV3.Query.pool(id='0x88e6a0c2ddd26feeb64f039a2c41296fcb3f5640')

Define FieldPaths for last 10 mints and burns >>> weth_usdc_last10_mints = weth_usdc.mints(… orderBy=uniswapV3.Mint.timestamp, … orderDirection='desc', … first=10 …)

```
>>> weth_usdc_last10_burns = weth_usdc.burns(
...   orderBy=uniswapV3.Burn.timestamp,
...   orderDirection='desc',
...   first=10
... )
```

Query flattened data. Since we are selecting two unnested list fields (i.e.: one is not nested inside the other) we will get back two DataFrames >>> [mints, burns] = sg.query_df([… weth_usdc_last10_mints.timestamp, … weth_usdc_last10_mints.amount0, … weth_usdc_last10_mints.amount1, … weth_usdc_last10_burns.timestamp, … weth_usdc_last10_burns.amount0, … weth_usdc_last10_burns.amount1, …])

```
>>> mints
   pool_mints_timestamp  pool_mints_amount0  pool_mints_amount1
0            1652114746        3.179282e+05          548.114506
1            1652113488        1.194870e+06          400.000000
2            1652113014        1.249154e+06          760.212396
3            1652112429        2.112058e+06         1891.033562
4            1652111846        3.646868e+05          140.593137
5            1652110962        5.740381e+04           23.700000
6            1652110962        3.522151e+05          548.186728
7            1652109116        5.223502e+04           20.894748
8            1652106977        1.204390e+04           10.525402
9            1652106531        7.187740e+02            0.993683
```

```
>>> burns
   pool_burns_timestamp  pool_burns_amount0  pool_burns_amount1
```

(continues on next page)

(continued from previous page)

```
0            1652115279       278038.482233           57.092565
1            1652114890         1754.280892            6.006157
2            1652114731            0.000000          697.289317
3            1652114233            0.000000           19.743883
4            1652113956            0.000000            0.017788
5            1652112925         4743.955507            0.780928
6            1652112719            0.000000          587.274826
7            1652112509            0.000000           23.297411
8            1652112509            0.000000          760.767874
9            1652112459            0.000000          631.697170
```

Subgrounds.query_json Subgrounds allows users to query data in its raw JSON format using the *query_json* method: ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

```
>>> top_2_traded_tokens = uniswapV3.Query.tokens(
...   orderBy=uniswapV3.Token.volumeUSD,
...   orderDirection='desc',
...   first=2
... )
```

```
>>> sg.query_json([
...   top_2_traded_tokens.symbol
... ])
[{'x6e2162a85075a2b3': [{'symbol': 'WETH',
    'id': '0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2',
    'volumeUSD': '456255840800.005036444641069804566 2'},
   {'symbol': 'USDC',
    'id': '0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48',
    'volumeUSD': '324987692399.294355137570564364123 1'}]}]
```

```{eval-rst} .. Note:: Subgrounds uses GraphQL aliases to differentiate queries selecting the same fields but with different arguments, which is why the JSON response data contains the key ``x6e2162a85075a2b3`` instead of the "expected" ``tokens`` key (see Subgrounds query aliases for more information). Subgrounds also selects additional fields automatically due to how automatic pagination is implemented. In the previous example, the ``id`` and ``volumeUSD`` fields were automatically added to the query by Subgrounds. However, these particularities are only apparent when using ``query_json``, hence why using this method is not recommended unless it is absolutely necessary to get the raw JSON data. ```

Combining FieldPaths When passing a list of *FieldPaths* as argument to one of the aforementioned functions, the *FieldPaths* are merged together in a single query **if the `FieldPaths` originate from the same subgraph**: ```python >>> uniswapV3 = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/uniswap/uniswap-v3')

Partial FieldPath selecting the top 4 most traded tokens on Uniswap V3 >>> most_traded_tokens = uniswapV3.Query.tokens(… orderBy=uniswapV3.Token.volumeUSD, … orderDirection='desc', … first=4 …)

Partial FieldPath selecting the top 2 pools by TVL of the top 4 most traded tokens # (notice the FieldPath starts from most_traded_tokens) >>> most_liquid_pairs = most_traded_tokens.whitelistPools(… orderBy=uniswapV3.Pool.totalValueLockedUSD, … orderDirection='desc', … first=2 …)

Querying the symbol of the top 4 most traded tokens, their 2 most liquid # pools' token symbols and their 2 most liquid pool's TVL in USD >>> sg.query_df([… most_traded_tokens.symbol, … most_liquid_pairs.token0.symbol, … most_liquid_pairs.token1.symbol, … most_liquid_pairs.totalValueLockedUSD …])
```

tokens_symbol tokens_whitelistPools_token0_symbol tokens_whitelistPools_token1_symbol tokens_whitelistPools_totalValueLockedUSD

0 WETH USDC WETH 4.068723e+08 1 WETH WBTC WETH 3.311227e+08 2 USDC DAI USDC 3.284779e+08 3 USDC USDC WETH 4.068723e+08 4 USDT WETH USDT 2.055448e+08 5 USDT USDC USDT 1.980053e+08 6 DAI DAI USDC 3.284779e+08 7 DAI DAI WETH 9.759597e+07 ```

Under the hood, when executing the previous code, Subgrounds will combine the queried *FieldPaths* into the following GraphQL query: ```graphql query {

tokens(first: 4, orderBy: volumeUSD, orderDirection: desc) {
symbol whitelistPools(first: 2, orderBy: totalValueLockedUSD, orderDirection: desc) {

token0 {
symbol

} token1 {

symbol

} totalValueLockedUSD

}

}

CHAPTER

THREE

}

<!– In the cases where the *FieldPaths* originate from different subgraphs, then multiple queries will be executed concurrently: –>

SyntheticFields One of Subgrounds' unique features is the ability to define schema-based (i.e.: pre-querying) transformations using *SyntheticFields*.

SyntheticFields can be created using Python arithmetic operators on relative *FieldPaths* (i.e.: *FieldPaths* starting from an entity and not the root *Query* object) and must be added to the entity which they enhance. Once added to an entity, *SyntheticFields* can be queried just like regular GraphQL fields. The example below demonstrates how to create a simple *SyntheticField* to calculate the swap price of *Swap* events stored on the Sushiswap subgraph: ```python >>> sushiswap = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/sushiswap/exchange')

Define a synthetic field named price1 (the swap price of token1, # in terms of token0) on Swap entities >>> sushiswap.Swap.price1 = abs(sushiswap.Swap.amount0Out - sushiswap.Swap.amount0In) / abs(sushiswap.Swap.amount1Out - sushiswap.Swap.amount1In)

Build query to get the last 10 swaps of the WETH-USDC pair on Sushiswap >>> weth_usdc = sushiswap.Query.pair(id='0x397ff1542f962076d0bfe58ea045ffa2d347aca0')

```
>>> last_10_swaps = weth_usdc.swaps(
...   orderBy=sushiswap.Swap.timestamp,
...   orderDirection='desc',
...   first=10
... )
```

Query swap prices using the SyntheticField price1 just like they were regular fields >>> sg.query_df([… last_10_swaps.timestamp, … last_10_swaps.price1 …])

pair_swaps_timestamp pair_swaps_price1

0 1654267855 1762.526840 1 1654267855 1760.132097 2 1654267766 1747.485689 3 1654267562 1758.131683 4 1654267528 1758.081544 5 1654267493 1742.260312 6 1654267493 1749.891244 7 1654267493 1755.254957 8 1654267364 1748.934985 9 1654267356 1750.284126 ```

SyntheticFields can also be created using the constructor, allowing for much more complex transformations. ```python >>> from datetime import datetime >>> from subgrounds.subgraph import SyntheticField

```
>>> sushiswap = sg.load_subgraph('https://api.thegraph.com/subgraphs/name/sushiswap/
↪exchange')
```

Create a SyntheticField on the Swap entity called *datetime*, which will format # the timestamp field into something more human readable >>> sushiswap.Swap.datetime = SyntheticField(… lambda timestamp: str(datetime.fromtimestamp(timestamp)), … SyntheticField.STRING, … sushiswap.Swap.timestamp …)

```
>>> last_10_swaps = sushiswap.Query.swaps(
...   orderBy=sushiswap.Swap.timestamp,
...   orderDirection='desc',
...   first=10
... )
```

```
>>> sg.query_df([
...   last_10_swaps.datetime,
...   last_10_swaps.to,
...   last_10_swaps.pair.token0.symbol,
...   last_10_swaps.pair.token1.symbol
... ])
        swaps_datetime                                     swaps_to swaps_pair_token0_
↪symbol swaps_pair_token1_symbol
0  2022-06-03 11:04:25  0x98c3d3183c4b8a650614ad179a1a98be0a8d6b8e                    ␣
↪OOKI                     WETH
1  2022-06-03 11:03:29  0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f                  ␣
↪CHIMP                     WETH
2  2022-06-03 11:03:29  0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f                    ␣
↪WETH                     USDT
3  2022-06-03 11:02:59  0x0eae044f00b0af300500f090ea00027097d03000                    ␣
↪ICHI                     WETH
4  2022-06-03 11:02:59  0xfb3bd022d5dacf95ee28a6b07825d4ff9c5b3814                    ␣
↪USDC                     WETH
5  2022-06-03 11:02:59  0x11111112542d85b3ef69ae05771c2dccff4faa26                    ␣
↪WETH                     USDT
6  2022-06-03 11:02:21  0xd7c09e006a2891880331b0f6224071c1e890a98a                    ␣
↪WETH                     ROOK
7  2022-06-03 11:02:21  0x60c809705e045572ffe3c44badd4d680165960fa                     ␣
↪OHM                      DAI
8  2022-06-03 11:01:00  0xd9e1ce17f2641f24ae83637ab66a2cca9c378b9f                     ␣
↪BIT                     WETH
9  2022-06-03 11:00:54  0xdef171fe48cf0115b1d80b88dc8eab59176fee57                    ␣
↪NEWO                     USDC
```

Advanced topics

Pagination By default, Subgrounds handles GraphQL query pagination automatically. That is, if a query selects more than 1000 entities using the *first* argument (1000 being The Graph's limit to the *first* argument), then Subgrounds will automatically split the query into multiple queries that each query at most 1000 entities.

Pagination is performed by Subgrounds with the use of a pagination strategy: a class that implements the *PaginationStrategy* protocol. Subgrounds provides two pagination strategies out of the box, however, users wishing to implement their own strategy should create a class that implements the aforementioned protocol (see below).

If at some point during the pagination process, an unhandled exception occurs, Subgrounds will raise a *PaginationError* exception containing the initial exception message as well as the *PaginationStrategy* object in the state it was in when the error occured, which, in the case of iterative querying (e.g.: when using *query_df_iter*), could be useful to recover and start pagination from a later stage.

Available pagination strategies Subgrounds provides two pagination strategies out of the box: 1. *LegacyStrategy*: A pagination strategy that implements the pagination algorithm that was used by default prior to this update. This pagination strategy supports pagination on nested fields, but is quite slow. Below is an example of a query for which you should use this strategy:

```graphql query {
    liquidityPools(first: 10) {
        swaps(first: 5000) {
            id
        }
    }
```

2. ***ShallowStrategy*: A new pagination strategy that is faster than the *LegacyStrategy*, but does not paginate on nested list fields. In other words, this strategy is best when nested list fields select fewer than 1000 entities. Below is an example of a query for which you should use this strategy:**
```graphql query {
    liquidityPools(first: 5000) {
        swaps(first: 10) {
            id
        }
    }
```

To use either pagination strategy, set the *pagination_strategy* argument of toplevel querying functions: ```python from subgrounds import Subgrounds from subgrounds.pagination import ShallowStrategy

sg = Subgrounds() subgraph = sg.load_subgraph("https://api.thegraph.com/subgraphs/name/messari/compound-ethereum")

mkt_daily_snapshots = subgraph.Query.marketDailySnapshots(
orderBy='timestamp', orderDirection='desc', first=1000

)

field_paths = [
mkt_daily_snapshots.timestamp, mkt_daily_snapshots.market.inputToken.symbol, mkt_daily_snapshots.rates.rate, mkt_daily_snapshots.rates.side,

]

df = sg.query_df(field_paths, pagination_strategy=ShallowStrategy) ```

Note that pagination can be explicitely disabled by setting *pagination_strategy* to *None*, in which case the query will be executed as-is: `python df = sg.query_df(field_paths, pagination_strategy=ShallowStrategy) `

Custom pagination strategy Subgrounds allows developers to create their own pagination strategy by creating a class that implements the *PaginationStrategy* protocol: ```python class PaginationStrategy(Protocol):

def __init__(
self, schema: SchemaMeta, document: Document

) -> None: ...

def step(
self, page_data: Optional[dict[str, Any]] = None

) -> Tuple[Document, dict[str, Any]]: ...

```

The class's constructor should accept a *SchemaMeta* argument which represents the schema of the subgraph API that the query is directed to and a *Document* argument which represents the query to be paginated on. If no pagination is required for the given document, then the constructor should raise a *SkipPagination* exception.
```

The class's *step* method is where the main logic of the pagination strategy is located. The method accepts a single argument, *page_data* which is a dictionary containing the response data of the previous query (i.e.: the previous page of data). The *step* method should return a tuple *(doc, vars)*, where *doc* is a *Document* representing the query to be made to fetch the next page of data. When pagination is over (e.g.: when all pages of data have been fetched), the *step* method should raise a *StopPagination* exception.

Below is the algorithm used by Subgrounds to paginate over a query document given a pagination strategy: ```python def paginate(

schema: SchemaMeta, doc: Document, pagination_strategy: Type[PaginationStrategy]

) **-> dict[str, Any]:**

try:

Initialize the strategy strategy = pagination_strategy(schema, doc)

data: dict[str, Any] = {}

Compute the query document and variables to get the first page of data next_page_doc, variables = strategy.step(page_data=None)

while True:

try:

Fetch a data page page_data = client.query(

url=next_page_doc.url, query_str=next_page_doc.graphql, variables=next_page_doc.variables | variables

)

Merge the page with the data blob data = merge(data, page_data)

Compute the query document and variables to get the next page of data next_page_doc, variables = strategy.step(page_data=page_data)

except StopPagination:

break

except Exception as exn:

raise PaginationError(exn.args[0], strategy)

return data

except SkipPagination:

Excecute the query document as is if *SkipPagination* is raised return client.query(doc.url, doc.graphql, variables=doc.variables)

```
```

CHAPTER

FOUR

SUBGROUNDS

4.1 subgrounds package

4.1.1 Subpackages

subgrounds.pagination package

Submodules

subgrounds.pagination.pagination module

subgrounds.pagination.preprocess module

subgrounds.pagination.strategies module

subgrounds.pagination.utils module

Module contents

subgrounds.subgraph package

Submodules

subgrounds.subgraph.fieldpath module

subgrounds.subgraph.filter module

subgrounds.subgraph.object module

subgrounds.subgraph.subgraph module

Module contents

4.1.2 Submodules

subgrounds.client module

subgrounds.dash_wrappers module

subgrounds.dataframe_utils module

subgrounds.plotly_wrappers module

subgrounds.query module

subgrounds.schema module

subgrounds.subgrounds module

subgrounds.transform module

subgrounds.utils module

4.1.3 Module contents

Troubleshooting

(isolated_environment_setup)= ## Set up an isolated environment (recommended setup) ### Pyenv Pyenv is a tool used to manage different versions of Python on a single operating system (follow this [guide](https://github.com/pyenv/pyenv#installation) for installation instructions). If you are having problems getting Python 3.10 to work, we recommend you use Pyenv to set it up.

Once pyenv is installed, navigate to your project's directory and run the following commands: ` pyenv install 3.10.4 pyenv local 3.10.4 `

The first command downloads Python version *3.10.4* on your operating system. The second command create a *.python-version* file in your current directory which will be used by pyenv so set the active Python version when you are working from that directory.

To test that the correct Python version has been selected, you can run the following command: `bash pyenv version ` Which should output something like: ` 3.10.4 (set by /my/project/path/.python-version) `

Note: At time of writing, *3.10.4* is the latest Python version.

Poetry Poetry is the recommended Python package manager to use with Subgrounds (follow this [guide](https://python-poetry.org/docs/master/#installing-with-the-official-installer) for installation instructions).

Once you have Poetry installed and your active Python version is *3.10.x*, initialize your project and install Subgrounds by running the following commands: ` poetry init poetry add subgrounds ipykernel `

Note: The *ipykernel* might be needed to run Jupyter notebooks with Python *3.10.x*.

Now you're all set!

To run a Subgrounds powered Python program, you can use either of the following methods: ` poetry run python my_subgrounds_project.py ` Or ` poetry shell python my_subgrounds_project.py `

Examples ## Community ### [Votium Bribes Analysis (credit: RplusT)](https://github.com/RplusT/on-chain/blob/main/Votium%20Bribes%20Analysis.ipynb)

[Aave V2 ETH and stETH analysis (credit: GivenAppimate)](https://github.com/GivenAppimate/web3-Data-analytics/blob/c6706d0a98e91dc1767c60e081e0b10b54e5869d/aaveM2.ipynb)

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search