Check the Chain (ctc)

Fei Labs

May 02, 2022

THE BASICS

Note: `ctc` is in beta, please report bugs to the issue tracker

`ctc` is a tool for historical data analysis of Ethereum and other EVM chains

It can be used as either 1) a cli tool or 2) a python package

CHAPTER

ONE

FEATURES

- **data collection**: collects data from archive nodes robustly and efficiently
- **data storage**: stores collected data on disk so that it only needs to be collected once
- **data coding**: handles data encoding and decoding automatically by default
- **data analysis**: computes derived metrics and other quantitative data summaries
- **data visualization**: plots data to maximize data interpretability and sharing
- **protocol specificity**: includes functionality for protocols like Chainlink and Uniswap
- **command line interface**: performs many block explorer tasks in the terminal

CHAPTER

TWO

GUIDE

- To install `ctc`, see Installation.
- To use `ctc` from the command line, see Command Line Interface.
- To use `ctc` in python, see Python Interface.
- To use `ctc` with specific protocols like Uniswap or Chainlink, see the Specific Protocols (cli) or Specific Protocols (python).
- To see examples of what you can do with `ctc`, see Case Studies.
- To view the `ctc` source code, check out the GitHub Repository.

CHAPTER

THREE

DATATYPES

Datatype	CLI	Python	Source
ABIs	CLI	Python	Source
Binary Data	CLI	Python	Source
Blocks	CLI	Python	Source
Contracts	CLI	Python	Source
Gas	CLI	Python	Source
ERC20s	CLI	Python	Source
ETH Balances	CLI	Python	Source
Events	CLI	Python	Source
Transactions	CLI	Python	Source

CHAPTER

FOUR

SPECIFIC PROTOCOLS

Protocol	CLI	Python	Source
Aave V2	-	Python	Source
Balancer	-	Python	Source
Chainlink	CLI	Python	Source
Compound	-	Python	Source
Curve	-	Python	Source
ENS	CLI	Python	Source
Fei	CLI	Python	Source
Rari	CLI	Python	Source
Uniswap V2	CLI	Python	Source
Uniswap V3	CLI	Python	Source

CHAPTER

FIVE

EXTERNAL DATA SOURCES

Data Source	CLI	Python	Source
4byte	CLI	Python	Source
CoinGecko	CLI	-	Source
Etherscan	-	-	Source

5.1 Installation

5.1.1 Basic Installation

Installing `ctc` takes two steps:

1. `pip install checkthechain`
2. enter `ctc setup` in the terminal to run the setup wizard

See *Configuration* for additional setup options.

If your shell's `PATH` does not include python package scripts you may need to do something like `python3 -m pip ...` and `python3 -m ctc ...`

Installation requires `python >= 3.7`. see *Dependencies* for more information.

5.1.2 Upgrading

Upgrading to a new version of `ctc` takes two steps:

1. `pip install checkthechain -U`
2. Rerun the setup wizard by running `ctc setup` (can skip most steps by pressing enter)

5.1.3 Alternative Installations

Installing from source

If you would like to install the latest version of `ctc` you can clone the repo directly:

```
git clone
cd checkthechain
python -m pip install ./
```

Installing in Develop Mode / Edit Mode

If you would like to make edits to the `ctc` codebase and actively use those edits in your python programs, you can install the package in developer mode with the `-e` flag:

```
git clone
cd checkthechain
python -m pip install -e ./
```

5.1.4 Libraries

On a fresh installation of Ubuntu or Debian you may need to manually install the `build-essential` and `python-dev` packages:

```
PYTHON_VERSION=$(python3 -c "import sys; print('python' + str(sys.version_info.major) +
↪'.' + str(sys.version_info.minor))")

python3 -m pip install $PYTHON_VERSION-dev
sudo apt-get install build-essential
```

5.2 Dependencies

TLDR

This page is only aimed at users that would like know what `ctc` depends on under the hood.

If you just want to install `ctc` then check out the *Installation* docs.

5.2.1 OS Dependencies

Usage of `ctc` requires `python >=3.7, <= 3.10`.

When using a fresh installation of Debian or Ubuntu, you may need to manually install `build-essential` and `python-dev`. These are libraries required by many python packages including `ctc`. If you are an active python user it's likely that you already have these installed. If you are setting up a new machine or environment, you may need to install them according to your operating system and python version:

To install on a fresh Debian / Ubuntu machine, can use the following:

```

PYTHON_VERSION=$(python3 -c "import sys; print('python' + str(sys.version_info.major) +
↪'.' + str(sys.version_info.minor))")

python3 -m pip install $PYTHON_VERSION-dev
sudo apt-get install build-essential
```

5.2.2 Libraries

`ctc` depends on a few different types of external packages:

1. **data science dependencies** include standard python library packages including `numpy`, and `pandas`.
2. **IO dependencies** include packages like `aiohttp` for network communication and `toml` for file io.
3. **toolsuite dependencies** are general python utilities coming the `toolsuite` set of repos. These are written by the same authors as `ctc`.
4. **EVM/Cryptography dependencies** include `pycryptodome`, `rlp`, and `eth_abi`.

Each of these dependencies has its own dependencies.

Reliance on these packages will be minimized in future releases to minimize attack surface and to maximize the number of environments in which `ctc` can be run. Some of the common libraries in the EVM ecosystem have incompatible requirements. For example, `ethereum-etl` requires older versions of `web3py` and `eth_abi`, and so a single environment cannot contain the most recent versions of all of these packages.

5.2.3 Type Checking

`ctc` uses `mypy` for static analysis of type annotations. This helps catch errors before they can appear at runtime. Custom types used by `ctc` can be found in the `ctc.spec` subpackage.

Checks are currently performed using `mypy=0.930`. Future `mypy` versions and features will be used as they become available. End users of `ctc` do not need `mypy` unless they are interested in running these type checks.

New python type annotation features will be used as they become available using `typing_extensions`. By using `typing_extensions` and `from __future__ import annotations`, new typing features can be used as they are released without sacrificing backward compatibility.

5.2.4 Testing

`ctc` tests are run using `pytest 7.0.0` with the `pytest-asyncio` extension. These can be run using `pytest .` in the `/tests` directory.

5.2.5 Documentation

`ctc` documentation is built using `sphinx` 4.5.0. Source files and build instructions can be found in the Documentation Repository.

5.2.6 Databases

`ctc` stores much of the data it downloads in sql databases. Support for sqlite is currently in beta and support for postgresql is coming soon.

For more details see *Data Storage*

5.3 Configuration

TLDR

Run `ctc setup` on the command line to create the config. Run it again to edit the config.

`ctc` uses a simple configuration file to control its behavior.

5.3.1 Config Parameters

The `ctc` config consists of key-value pairs. Keys include:

- **`config_spec_version`**: the `ctc` version used to create the config
- **`data_dir`**: the directory where `ctc` stores its data
- **`providers`**: metadata about RPC providers
- **`networks`**: metadata about custom networks including their names and `chain_id`'s
- **`network_defaults`**: specification of the default provider to use for each network, and the default network
- **`db_configs`**: database configuration information

The main parameters of interest will usually be `providers` and `network defaults`.

An exact specification for the config can be found in the config typedefs file.

5.3.2 Setting Config Parameters

By default ctc will looks for a config file at `~/.config/ctc/config.json`. But if the `CTC_CONFIG_PATH` environment variable is set, it will use that path instead.

Users do not need to directly create or edit `ctc` config files. Instead, all config parameters can be adjusted by using the setup wizard, activated by entering `ctc setup` on the command line. This can be used both for creating new configs and modifying the current config.

5.3.3 Reading Config Parameters

On the command line, using `ctc config` will print information about the config including its location on the filesystem and its current values.

In python, the `ctc.config` module has many functions for getting the current config path and its values:

```
from ctc import config

config_path = config.get_config_path()
data_dir = config.get_data_dir()
providers = config.get_providers()
```

5.3.4 Example Config

```
{
    'config_spec_version': '0.2.10',
    'data_dir': '/home/storm/ctc_data',
    'networks': {},
    'providers': {
        'alchemy_mainnet': {
            'name': 'alchemy_mainnet',
            'network': 'mainnet',
            'protocol': 'http',
            'url': 'https://some-mainnet-rpc-url',
            'session_kwargs': {},
            'chunk_size': None,
        },
        'alchemy_arbitrum': {
            'name': 'alchemy_arbitrum',
            'network': 'arbitrum',
            'protocol': 'http',
            'url': 'https://some-mainnet-rpc-url',
            'session_kwargs': {},
            'chunk_size': None,
        },
    },
    'network_defaults': {
        'default_network': 'mainnet',
        'default_providers': {
            'mainnet': 'alchemy_mainnet',
            'arbitrum': 'alchemy_arbitrum',
        },
    },
    'db_configs': {
        'main': {'dbms': 'sqlite', 'path': '/home/storm/ctc_data/ctc.db'}
    },
}
```

5.4 FAQ

5.4.1 What are the goals of `ctc`?

- **Treat historical data as a first-class feature**: This means having historical data functionality well-integrated into each part of the of the API. It also means optimizing the codebase with historical data workloads in mind.
- **Clean API emphasizing UX**: With `ctc` most data queries can be obtained with a single function call. No need to instantiate objects. RPC inputs/outputs are automatically encoded/decoded by default.
- **Maximize data accessibility**: Blockchains contain vast amounts of data, but accessing this data can require large amounts of time, effort, and expertise. `ctc` aims to lower the barrier to entry on all fronts.

5.4.2 Why use `async`?

`async` is a natural fit for efficiently querying large amounts of data from an archive node. All `ctc` functions that fetch external data use `async`. For tips on using `async` see *this section* in the docs. Future versions of `ctc` will include some wrappers for synchronous code.

5.4.3 Do I need an archive node?

If you want to query historical data, you will need an archive node. You can either run one yourself or use a third-party provider such as Alchemy, Quicknode, or Moralis. You can also use `ctc` to query current (non-historical) data using a non-archive node.

5.4.4 Is `ctc` useful for recent, non-historical data?

Yes, `ctc` has lots of functionality for querying the current state of the chain.

5.5 Obtaining Data

`ctc` collects data from a variety of sources, including RPC nodes, metadata databases, block explorers, and market data aggregators. After initial collection, much of this data is then *stored*.

5.5.1 Sources of Historical Data

`ctc` collects the majority of its data from RPC nodes using the EVM's standard JSON-RPC interface. Collection of historical data (as opposed to recent chain data) requires use of an archive node.

There are 3 main ways to gain access to an RPC node:

1. **Run your own node:** Although this requires more time, effort, and upfront cost than the other methods, it often leads to the best results. Erigon is the most optimized client for running an archive node.
2. **Use a 3rd-party private endpoint:** Private RPC providers (e.g. Alchemy, Quicknode, or Moralis) provide access to archive nodes, either through paid plans or sometimes even through free plans.
3. **Use a 3rd-party public endpoint:** You can query data from public endpoints like Infura. This approach is not recommended for any significant data workloads, as it often suffers from rate-limiting and poor historical data availability.

`ctc`'s RPC config is created and modified by running the *setup wizard*.

5.5.2 Other types of data

Beyond RPC data there are a few other types of data that `ctc` collects, including:

- **ABIs of Contracts, Functions, and Events** from Etherscan and 4byte
- **Market Data** from CoinGecko

5.6 Storing Data

`ctc` places much of the data that it retreives into local storage. This significantly improves the speed at which this data can be retrieved in the future and it also reduces the future load on those data sources.

The default configuration assumes that most data is being queried from a remote RPC node. Some performance-minded setups, such as running `ctc` on the same server as an archive node, might achieve better tradeoffs between speed and storage space by tweaking `ctc`'s local storage features.

5.6.1 Data Storage Backends

`ctc` uses two main storage backends.

Filesystem

`ctc` stores lots of data on the filesystem.

By default, `ctc` will place its data folder in the user's home directory at `~/.ctc`. This is suitable for many setups. However, there are situations where it would be better to store data somewhere else, such as if the home directory is on a drive of limited size, or it the home directory is on a network drive with significant latency. The data directory can be moved by running the setup wizard `ctc setup`.

SQL Databases

`ctc` also stores lots of data in SQL database tables. Schemas for these tables can be found here. `ctc` currently supports sqlite with Postgresql support coming soon.

You can connect to the currently configured database by running `ctc db connect` in the terminal. Don't do this unless you know what you're doing.

5.7 Performance

TLDR

Even in suboptimal conditions, `ctc` uses lots of optimizations that allow running many types of workloads at acceptable levels of performance. This page is for those who wish to squeeze additional performance out of `ctc`.

5.7.1 Optimizing Performance

There are many levers and knobs available for tuning `ctc`'s performance.

RPC Provider

Different 3rd party RPC providers can vary significantly in their reliability and speed. For example, some providers have trouble with large historical queries.

Operations in `ctc` that fetch external data are usually bottlenecked by the RPC provider, specifically the latency to the RPC provider. This latency can be reduced by running `ctc` as closely as possible to the archive node:

- Fastest = running ctc on the same server that is running the archive node
- Fast = running ctc on the same local network as the archive node
- Slower = running ctc in the same geographic region as the archive node
- Slowest = running ctc in a different geographic region than the archive node

If using a 3rd party RPC provider, you should inquire about where their nodes are located and plan accordingly.

`ctc`'s default configuration assumes that the user is querying an RPC node on a remote network. This leads `ctc` to locally store lots of the data that it retrieves. However, it's possible that different tradeoffs are relevant in heavily optimized setups. For example if `ctc` is run on the same server as an archive node, then some caches might hurt more than they help. Cache settings are altered using `ctc setup` on the command line.

Python Versions

More recent versions of python are generally faster. Upgrading to the latest python version is one of the easiest ways to improve code performance. In particular, the upcoming python 3.11 has much faster startup times and shows improvement across many benchmarks. This makes `ctc`'s cli commands feel especially quick and responsive.

Python Packages

By default, `ctc` tries to minimize its dependencies and minimize the number of build steps that happen during installation. This does carry a bit of performance cost. Faster versions of various packages can be installed using:

```
pip install checkthechain[performance]
```

If `ctc` detects that these additional performance packages are installed, it will use those instead of the default packages. This can produce a modest performance increase for some workloads.

Data Storage

`ctc`'s default data directory is `~/.config/ctc/` in the user's home directory. If this directory is on a slow drive (especially a network drive), this will negatively impact performance. To optimize performance, place the data directory on as fast a drive as possible. This can be done by running the setup wizard `ctc setup`.

Data Caching

For tasks that require many RPC requests, or require lots of post-processing (or are demanding in other ways), you should consider caching the result in-memory or on-disk. One way to do this is with the toolcache package. With `toolcache` a simple decorator adds an in-memory or on-disk cache to the expensive function.

For example, if you are using `ctc` to create data payloads for a historical analytics dashboard, you might use a pattern similar to this:

```
import toolcache


async def create_data_payload(timestamps):
    return [
        compute_timestamp_stats(timestamp=timestamp)
        for timestamp in timestamps
    ]


# create an on-disk cache entry for each timestamp
@toolcache.cache('disk')
async def compute_timestamp_stats(timestamp):
    super_expensive_operation()
```

Logging

Logging of RPC requests and SQL queries takes up a non-zero amount of resources. If you don't need logging, disabling it can squeeze out a bit of extra performance. This can be done by running the setup wizard `ctc setup`.

5.7.2 Benchmarking Performance

To truly optimize your environment and implementation, you will need to run your own benchmarks.

Benchmarking Speed

The simplest way to benchmark the speed of a CLI command is `time`. Running `time <command>` will run a given command and report the run time. Benchmarking speed of python code snippets is slightly more complicated but also has many tools available:

1. Synchronous code can be easily profiled used IPython's built-in magics `%timeit`, `%%timeit`, `%prun`, and `%%prun`
2. If using a Jupyter notebook, the Execute Time extension can be extremely useful for getting a crude estimate of how long each code cell takes to run. This works for both synchronous and asynchronous code.
3. For a more programmatic approach you can use python's built-in profilers or 3rd party profilers such as Scalene or pyflame.

Measuring Storage Usage

It is also valuable to measure `ctc`'s storage usage to check whether it falls into an acceptable range for a given hardware setup. Storage usage in the `ctc` data folder can be found by running a storage profiling command like `du -h` or `dust`. Storage usage in databases can be found by running `ctc db storage`.

5.8 Monitoring

5.8.1 Logging

`ctc` can log outgoing RPC requests and SQL queries. This functionality can be enabled or disabled using `ctc setup`.

Logs are stored in the `ctc` data dir:

- `./logs/rpc_requests.md`
- `./logs/sql_queries.md`

Running `ctc log` in the terminal will start a watching script of the log files. This provides a detailed view of external queries as they happen, which can be useful for debugging and ensuring that external calls are happening as expected.

Logs are written to disk using a non-blocking queue, making it suitable for async applications and imparting minimal impact on performance. These logs are also rotated once they reach a certain size (default = `10MB`). However, being non-blocking also means that the timestamps in the logs lose a bit of temporal precision, and so they do not provide a precise picture of event timing.

Logs are managed by the Loguru package. Loguru must be installed for logging to be enabled (`pip install loguru`).

5.8.2 Other monitoring

Beyond the built-in logging, the best way to monitor `ctc` is through standard 3rd party tools.

Recommended utilities for profiling resource usage:

- **CPU Usage**: htop, btop
- **Storage IO**: iotop, btop
- **Storage Capacity**: du, dust, btop
- **Network Usage**: nethogs, btop

If your situation calls for a more programmatic monitoring approach, then you probably already know what tools you need.

5.9 Basic Usage

The `ctc` cli command performs operations related to processing EVM data, especially operations related to historical data analysis. Many different EVM datasets can be generated by individual calls to `ctc`.

Typical usage is `ctc <subcommand> [options]`, using Subcommands.

Most of the cli documentation pages are copied from `ctc`'s in-terminal help messages.

5.10 Subcommands

Note: Click on a subcommand to view its documentation page.

5.10.1 Admin Subcommands

Note: Click on a subcommand to view its documentation page.

config

config path

download-proxy-abi

log

rechunk-events

setup

5.10.2 Compute Subcommands

Note: Click on a subcommand to view its documentation page.

ascii

checksum

decode

hex

keccak

lower

5.10.3 Data Subcommands

Note: Click on a subcommand to view its documentation page.

abi

address

address transactions

block

blocks

call

calls

db connect

db create tables

decompile

erc20 balance

erc20 balances

erc20 transfers

eth balance

eth balances

events

find

gas

transaction

5.10.4 Protocol Subcommands

Note: Click on a subcommand to view its documentation page.

4byte

4byte build

4byte path

cg

chainlink

chainlink ls

curve pools

ens

ens exists

ens hash

ens owner

ens records

ens resolve

ens reverse

fei analytics

fei depth

fei pcv

fei pcv assets

fei pcv deposits

rari

rari pools

uniswap burns

uniswap chart

uniswap mints

uniswap pool

uniswap swaps

5.10.5 Other Subcommands

Note: Click on a subcommand to view its documentation page.

cd

help

record help

version

5.11 Useful Aliases

`ctc` makes it simple to perform many tasks from the command line. However, `ctc` can be made even more simple by using shell aliases that reduce the number of required keystrokes that must be typed.

The `ctc` codebase includes an optional set of cli aliases for this purpose. These aliases can be found in a script in the repo, but they are also pasted below for convenience.

These aliases make it so you do not need to type the `"ctc"` before a subcommand name. For example, instead of typing `ctc keccak <address>`, you just type `keccak <address>`. Instead of typing `ctc 4byte <query>`, you just type `4byte <query>`. And so on, for many different `ctc` subcommands.

To use these aliases, you need to include them in your shell config file (e.g. `~/.profile`). You can either copy paste the alias commands directly, or you can add a single line: `source <PATH_TO_ALIAS_FILE>`.

5.11.1 The Aliases File

```
#!/bin/sh

# these commands allow you to call ctc subcommands without typing the "ctc" part

# to use, either:
# 1. add the contents of this file to the end of your terminal config file
# 2. add `source PATH_TO_THIS_FILE` to the end of your terminal config file

# to learn more about these commands, run `ctc help`

# depending on preference can use all of these aliases or just a subset of them:

# compute commands
alias ascii="ctc ascii"
alias hex="ctc hex"
alias keccak="ctc keccack"
alias lower="ctc lower"
```

(continues on next page)

(continued from previous page)

```
# data commands
alias abi="ctc abi"
alias address="ctc address"
alias block="ctc block"
alias blocks="ctc blocks"
alias call="ctc call"
alias calls="ctc calls"
alias erc20="ctc erc20"
alias eth="ctc eth"
alias gas="ctc gas"
alias transaction="ctc transaction"

# protocol commands
alias 4byte="ctc 4byte"
alias cg="ctc cg"
alias chainlink="ctc chainlink"
alias curve="ctc curve"
alias ens="ctc ens"
alias fei="ctc fei"
alias rari="ctc rari"
```

5.12 Similar CLI tools

5.12.1 ethereum-etl

ethereum-etl is a tool for collecting raw historical data from EVM chains, including blocks, transactions, erc20 transfers, and internal traces. Along with the rest of the blockchain-etl stack, it powers the popular BigQuery blockchain datasets. The primary use case of `ethereum-etl` and its associated repos is to index a significant portion of a chain's history in preparation for large scale data analysis.

Prior to creating `ctc`, `ethereum-etl` was the primary data collection tool used by `ctc`'s authors. It was extensive use of `ethereum-etl` that inspired much of `ctc`'s design. Compared to `ethereum-etl`, `ctc` aims fall closer to the porcelain end of the plumbing-vs-porcelain spectrum, with goals such as:

- create more diverse datasets, such as datasets that rely on `eth_call`
- create more targeted datasets, such as datasets focused on specific protocols like Chainlink or Uniswap
- create tighter integration with the python ecosystem
- go beyond data collection by creating a data analysis toolkit that serves each stage of the data analysis lifecycle
- implement quality-of-life improvements for the lazy
  - store and manage metadata such as addresses of tokens, oracles, and pools
  - automate tasks such as data encoding/decoding

5.12.2 TrueBlocks

TrueBlocks is a tool for managing optimized local copies of EVM chain data. TrueBlocks then makes these local data copies accessible through an enhanced RPC interface. TrueBlocks excels at tracing and querying all appearances of a given address throughout a chain's history.

There's a decent amount of overlap between `ethereum-etl`, TrueBlocks, and `ctc`. Relatively speaking, `ethereum-etl` is plumbing, TrueBlocks is mostly plumbing with some porcelain, and `ctc` is mostly porcelain with some plumbing.

5.12.3 ethereal, seth, and cast

ethereal (go), seth (dapptools, bash+javascript), and cast (foundry, rust) are powerful command line utilities that each perform a wide range of EVM-related tasks.

`ctc` has lots of overlapping functionality with each. Where they differ is their focus. These other tools are more aimed at smart contract development, whereas `ctc` is more aimed at data collection and analysis. Compared to these tools, `ctc`'s biggest disadvantage is that it is limited to read-only operations. On the other hand `ctc`'s biggest advantage is its treatment of historical data as a first class feature.

5.13 Codebase Tour

5.13.1 Architecture and API

- `ctc` does not use any custom types or OOP. Instead it emphasizes simple standard datatypes including python builtins, numpy arrays, and the occassional pandas dataframe. Effort is made to ensure that the users stay "close" to the data with minimal implicit magic going on.
- `ctc` is *asynchronous-first*, which allows it to efficiently orchestrate large numbers of interdependent queries.
- `ctc` is designed with historical data in mind. Throughout its API many functions take parameters such as `block`, `blocks`, `start_block`, or `end_block` to specify the relevant block ranges for each query.

5.13.2 Subpackages

There are a few subpackages that you should acquiant yourself with:

ctc.evm

`ctc.evm` defines high-level functions for working with EVM data, and it contains most of the functions that users will need on a day-to-day basis. These are the "porcelain" functions, whereas the other `ctc` subpackages are the plumbing.

ctc.protocols

`ctc.protocols` contains functions specific to many different protocols such as Chainlink or Uniswap.

ctc.spec

`ctc.spec` is where most annotation types are defined.

Other Subpackages

- `ctc.binary`: utilities for hashing and abi encoding/decoding
- `ctc.cli`: ctc command line implementation
- `ctc.config`: config loading and management
- `ctc.directory`: token and contract addresses as well as chain_id's
- `ctc.rpc`: utilities for communicating over rpc
- `ctc.toolbox`: miscellaneous python utilities

5.14 RPC Client

`ctc.rpc` is a low-level asynchronous RPC client that implements the EVM JSON-RPC standard. This standard consists of many methods such as `eth_call` and `eth_getCode` that query current and historical states of an EVM chain.

5.14.1 Implementation of Methods

For every method specified by the EVM JSON-RPC standard, `ctc.rpc` implements five python functions:

1. **constructor function**: create method requests
2. **digestor function**: process method responses
3. **executor function**: perform construction, , and digestion all in one step
4. **batch construct**: create method requests in bulk
5. **batch execute**: execute method requests in bulk

(there are no batch digestor functions because they compose naturally from the scalar digestor functions)

5.14.2 RPC Providers

Unless otherwise specified, requests will be sent to the default RPC provider in `ctc`'s config. Functions in `ctc.rpc` that send RPC requests also take an optional `provider` argument that can be used to specify other RPC providers.

For more details, see the RPC Provider section on the *Data Sources* page.

5.14.3 Typical RPC Request Lifecycle in `ctc`

1. a constructor function encodes request metadata and parameters into a `RpcRequest` `dict`
2. the request is dispatched to an rpc provider using `rpc.async_send_http()`
3. the client `await`s until the rpc provider returns a response
4. a digestor function decodes the response

For requests that execute contract code (like `eth_call`) or retrieve events (like `getLogs`), `ctc` will encode/decode inputs/outputs using the relevant function abi's and event abi's.

5.15 Asynchronous Code

`ctc` uses `async` functions for network calls and database calls. This allows for high levels of concurrency and makes it easy to dispatch large numbers of complex interdependent queries.

`async` is an intermediate-level python topic with a bit of a learning curve. If you've never used `async` before, you should probably read a tutorial or two before trying to use it in `ctc`.

To use `async` functions, they must be run from an event loop. These functions can be called from synchronous code as follows:

```
import asyncio

result = asyncio.run(some_async_function(input1, input2))
```

If you are using IPython or Jupyter notebooks, you can directly `await` the `async` functions inside code cells without using `asyncio.run()`:

```
result = await some_async_function(input1, input2)
```

If your code opens up network connections, you should also close those connections at the end of your scipt. For example:

```
from ctc import rpc

await rpc.async_close_http_session(provider=provider)
```

5.16 Datatypes

Datatype	Examples	Reference
ABIs	Examples	Reference
Binary	Examples	Reference
Blocks	Examples	Reference
Contracts	Examples	Reference
Gas	Examples	Reference
ERC20s	Examples	Reference
ETH	Examples	Reference
Events	Examples	Reference
Transactions	Examples	Reference

5.16.1 ABIs

Examples

Reference

ctc.binary.**get_event_hash**(*event_abi*)

compute event hash from event's abi

Parameters **event_abi** (*<class 'EventABI'>*) –

Return type <class 'str'>

ctc.binary.**get_event_indexed_names**(*event_abi*)

get list of indexed names in signature of event

ctc.binary.**get_event_indexed_types**(*event_abi*)

get list of indexed types in signature of event

ctc.binary.**get_event_signature**(*event_abi*)

Parameters **event_abi** (*<class 'EventABI'>*) –

Return type <class 'str'>

ctc.binary.**get_event_unindexed_names**(*event_abi*)

get list of data names in signature of event

ctc.binary.**get_event_unindexed_types**(*event_abi*)

get list of data types in signature of event

ctc.binary.**get_function_output_names**(*function_abi*, *human_readable=False*)

ctc.binary.**get_function_output_types**(*function_abi*)

ctc.binary.**get_function_parameter_types**(*function_abi=None*, *function_signature=None*)

Parameters

- **function_abi** (*FunctionABI | None*) –
- **function_signature** (*Optional[str]*) –

Return type list[ABIDatumType]

ctc.binary.**get_function_selector**(*function_abi=None*, *function_signature=None*)

Parameters

- **function_abi** (*Union[FunctionABI, NoneType]*) –
- **function_signature** (*Union[str, NoneType]*) –

Return type <class 'str'>

ctc.binary.**get_function_signature**(*function_abi=None*, *parameter_types=None*, *function_name=None*, *include_names=False*)

Parameters

- **function_abi** (*FunctionABI | None*) –
- **parameter_types** (*Optional[list[str]]*) –

- **function_name** (*Optional[str]*) –
- **include_names** (*bool*) –

Return type str

async ctc.evm.**async_decompile_function_abis**(*bytecode*, *sort=None*)

Parameters

- **bytecode** (*str*) –
- **sort** (*str | None*) –

Return type Sequence[Mapping]

async ctc.evm.**async_get_contract_abi**(***query*)

Parameters **query** (*Any*) –

Return type List[Union[FunctionABI, EventABI, ErrorABI]]

async ctc.evm.**async_get_event_abi**(*contract_abi=None*, *contract_address=None*, *event_name=None*, *event_hash=None*, *event_abi=None*, *network=None*)

Parameters

- **contract_abi** (*Union[List[Union[FunctionABI, EventABI, ErrorABI]], NoneType]*) –
- **contract_address** (*Union[str, NoneType]*) –
- **event_name** (*Union[str, NoneType]*) –
- **event_hash** (*Union[str, NoneType]*) –
- **event_abi** (*Union[EventABI, NoneType]*) –
- **network** (*Union[int, str, NoneType]*) –

Return type <class 'EventABI'>

async ctc.evm.**async_get_function_abi**(*function_name=None*, *contract_abi=None*, *contract_address=None*, *n_parameters=None*, *parameter_types=None*, *function_selector=None*, *network=None*)

5.16.2 Binary Data

Note: By default ctc will perform end-to-end encoding/decoding of many operations. The low-level functions listed here are only needed if you need to work directly with raw binary data.

Examples

Reference

ctc.binary.**decode_call_data**(*call_data*, *function_abi=None*, *contract_abi=None*)

Parameters

- **call_data** (*Union[bytes, str]*) –
- **function_abi** (*Union[FunctionABI, NoneType]*) –
- **contract_abi** (*Union[List[Union[FunctionABI, EventABI, ErrorABI]], NoneType]*) –

Return type <class 'DecodedCallData'>

ctc.binary.**decode_function_output**(*encoded_output*, *output_types=None*, *function_abi=None*, *delist_single_outputs=True*, *package_named_outputs=False*)

ctc.binary.**decode_types**(*data*, *types*)

Parameters

- **data** (*<class 'bytes'>*) –
- **types** (*<class 'str'>*) –

Return type Any

ctc.binary.**encode_call_data**(*, *function_selector=None*, *parameter_types=None*, *parameters=None*, *encoded_parameters=None*, *function_abi=None*)

Parameters

- **function_selector** (*Union[str, NoneType]*) –
- **parameter_types** (*Union[str, NoneType]*) –
- **parameters** (*Union[Sequence[Any], Mapping[str, Any], NoneType]*) –
- **encoded_parameters** (*Union[bytes, str, NoneType]*) –
- **function_abi** (*Union[FunctionABI, NoneType]*) –

Return type <class 'str'>

ctc.binary.**encode_types**(*data*, *types*)

Parameters

- **data** (*Any*) –
- **types** (*<class 'str'>*) –

Return type <class 'bytes'>

5.16.3 Blocks

Examples

Reference

async ctc.evm.**async_get_block**(*block*, *include_full_transactions=False*, *provider=None*)

Parameters

- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]*) –
- **include_full_transactions** (*<class 'bool'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –

Return type <class 'Block'>

async ctc.evm.**async_get_block_of_timestamp**(*timestamp*, *nary=None*, *cache=None*, *block_timestamps=None*, *block_timestamp_array=None*, *block_number_array=None*, *verbose=False*, *provider=None*, *use_db=True*)

Parameters

- **timestamp** (*Union[int, float, str, datetime]*) –
- **nary** (*Union[int, NoneType]*) –
- **cache** (*Union[BlockTimestampSearchCache, NoneType]*) –
- **block_timestamps** (*Union[Mapping[int, int], NoneType]*) –
- **block_timestamp_array** (*Union[Any, NoneType]*) –
- **block_number_array** (*Union[Any, NoneType]*) –
- **verbose** (*<class 'bool'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **use_db** (*<class 'bool'>*) –

Return type <class 'int'>

async ctc.evm.**async_get_blocks**(*blocks*, *include_full_transactions=False*, *chunk_size=500*, *provider=None*)

async ctc.evm.**async_get_blocks_of_timestamps**(*timestamps*, *block_timestamps=None*, *block_number_array=None*, *block_timestamp_array=None*, *nary=None*, *cache=None*, *provider=None*, *use_db=None*)

once parallel node search created, use that

Parameters

- **timestamps** (*Sequence[int]*) –
- **block_timestamps** (*Optional[Mapping[int, int]]*) –
- **block_number_array** (*Optional[NumpyArray]*) –
- **block_timestamp_array** (*Optional[NumpyArray]*) –
- **nary** (*Optional[int]*) –

- **cache** (*Optional[BlockTimestampSearchCache]*) –
- **provider** (*ProviderSpec*) –
- **use_db** (*bool | None*) –

Return type list[int]

5.16.4 Contracts

Examples

Reference

async ctc.evm.**async_get_contract_creation_block**(*contract_address*, *provider=None*, ***search_kwargs*)

get block number of when contract was created

- caches result in local database
- behavior is undefined for functions that have undergone SELF-DESTRUCT(S)

Parameters

- **contract_address** (*Address*) –
- **provider** (*ProviderSpec*) –
- **search_kwargs** (*Any*) –

Return type int | None

5.16.5 Gas

Examples

Note: These examples are a Jupyter notebook. Grab the original notebook here

Reference

async ctc.evm.**async_get_block_gas_stats**(*block*, *normalize=True*, *provider=None*)

get gas statistics for a given block

Parameters

- **block** (*BlockNumberReference | Block*) –
- **normalize** (*bool*) –
- **provider** (*ProviderSpec*) –

Return type *BlockGasStats*

async ctc.evm.**async_get_blocks_gas_stats**(*blocks=None*, *start_block=None*, *end_block=None*, *normalize=True*, *provider=None*)

get gas statistics aggregated over multiple blocks

Parameters

- **blocks** (*Sequence[BlockNumberReference] | None*) –
- **start_block** (*BlockNumberReference | None*) –
- **end_block** (*BlockNumberReference | None*) –
- **normalize** (*bool*) –
- **provider** (*ProviderSpec*) –

Return type *BlocksGasStats*

class ctc.evm.**BlockGasStats**(***kwargs*)

base_fee: int | float | None

gas_limit: int

gas_used: int

max_gas_price: int | float | None

mean_gas_price: float | None

median_gas_price: int | float | None

min_gas_price: int | float | None

n_transactions: int

class ctc.evm.**BlocksGasStats**(***kwargs*)

max_base_fee: int | float | None

max_gas_limit: int | float | None

max_gas_price: int | float | None

max_gas_used: int | float | None

max_mean_gas_price: int | float | None

max_median_gas_price: int | float | None

max_min_gas_price: int | float | None

max_n_transactions: int | float | None

mean_base_fee: int | float | None

mean_gas_limit: int | float | None

mean_gas_price: int | float | None

mean_gas_used: int | float | None

```
mean_max_gas_price: int | float | None
mean_median_gas_price: int | float | None
mean_min_gas_price: int | float | None
mean_n_transactions: int | float | None
median_base_fee: int | float | None
median_gas_limit: int | float | None
median_gas_used: int | float | None
median_max_gas_price: int | float | None
median_mean_gas_price: int | float | None
median_median_gas_price: int | float | None
median_min_gas_price: int | float | None
median_n_transactions: int | float | None
min_base_fee: int | float | None
min_gas_limit: int | float | None
min_gas_price: int | float | None
min_gas_used: int | float | None
min_max_gas_price: int | float | None
min_mean_gas_price: int | float | None
min_median_gas_price: int | float | None
min_n_transactions: int | float | None
n_blocks: int
```

5.16.6 ERC20s

Note: functions that require multiple RPC calls will attempt to do so concurrently for maximum efficiency

Examples

Reference

async ctc.evm.**async_get_erc20_balance_of**(*address*, *token*, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

Parameters

- **address** (*<class 'str'>*) –
- **token** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **normalize** (*<class 'bool'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **rpc_kwargs** (*Any*) –

Return type Union[int, float]

async ctc.evm.**async_get_erc20_balance_of_addresses**(*addresses*, *token*, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

async ctc.evm.**async_get_erc20_balance_of_by_block**(*address*, *token*, *blocks*, *normalize=True*, *provider=None*, *empty_token=0*, ***rpc_kwargs*)

async ctc.evm.**async_get_erc20_decimals**(*token*, *block=None*, ***rpc_kwargs*)

get decimals of an erc20

Parameters

- **token** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **rpc_kwargs** (*Any*) –

Return type <class 'int'>

async ctc.evm.**async_get_erc20_holdings_from_transfers**(*transfers*, *block=None*, *dtype=None*, *normalize=False*)

Parameters

- **transfers** (*Any*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **dtype** (*Union[Type[int], Type[float], NoneType]*) –
- **normalize** (*<class 'bool'>*) –

Return type Any

async ctc.evm.**async_get_erc20_name**(*token*, *block=None*, ***rpc_kwargs*)

get name of an erc20

Parameters

- **token** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –

- **rpc_kwargs** (*Any*) –

Return type <class 'str'>

async ctc.evm.**async_get_erc20_symbol**(*token*, *block=None*, ***rpc_kwargs*)

get symbol of an erc20

Parameters

- **token** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **rpc_kwargs** (*Any*) –

Return type <class 'str'>

async ctc.evm.**async_get_erc20_total_supply**(*token*, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

Parameters

- **token** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **normalize** (*<class 'bool'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **rpc_kwargs** (*Any*) –

Return type Union[int, float]

async ctc.evm.**async_get_erc20_total_supply_by_block**(*token*, *blocks*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

async ctc.evm.**async_get_erc20_transfers**(*token_address*, *start_block=None*, *end_block=None*, *normalize=True*, *convert_from_str=True*, ***event_kwargs*)

Parameters

- **token_address** (*<class 'str'>*) –
- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **normalize** (*<class 'bool'>*) –
- **convert_from_str** (*<class 'bool'>*) –
- **event_kwargs** (*Any*) –

Return type Any

async ctc.evm.**async_get_erc20s_balance_of**(*address*, *tokens*, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

async ctc.evm.**async_get_erc20s_decimals**(*tokens*, *block=None*, ***rpc_kwargs*)

get decimals of multiple erc20s

async ctc.evm.**async_get_erc20s_names**(*tokens*, *block=None*, ***rpc_kwargs*)

get name of multiple erc20s

async ctc.evm.**async_get_erc20s_symbols**(*tokens*, *block=None*, ***rpc_kwargs*)

get symbol of multiple erc20s

async ctc.evm.**async_get_erc20s_total_supplies**(*tokens*, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

async ctc.evm.**async_normalize_erc20_quantities**(*quantities*, *token=None*, *provider=None*, *decimals=None*, *block=None*)

Parameters

- **quantities** (*Sequence[SupportsInt] | Series*) –
- **token** (*ERC20Address | None*) –
- **provider** (*ProviderSpec*) –
- **decimals** (*Optional[SupportsInt]*) –
- **block** (*Optional[BlockNumberReference]*) –

Return type list[float]

async ctc.evm.**async_normalize_erc20_quantity**(*quantity*, *token=None*, *provider=None*, *decimals=None*, *block=None*)

convert raw erc20 quantity by adjusting radix by (10 ** decimals)

Parameters

- **quantity** (*<class 'SupportsFloat'>*) –
- **token** (*Union[str, NoneType]*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **decimals** (*Union[SupportsInt, NoneType]*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –

Return type <class 'float'>

5.16.7 ETH Balances

Examples

Reference

5.16.8 Events

Examples

Reference

async ctc.evm.**async_get_events**(*, *contract_address*, *start_block=None*, *end_block=None*, *backend_order=None*, *keep_multiindex=True*, ***query*)

Parameters

- **contract_address** (*Address*) –
- **start_block** (*BlockNumberReference | None*) –
- **end_block** (*BlockNumberReference | None*) –
- **backend_order** (*Sequence[str] | None*) –
- **keep_multiindex** (*bool*) –
- **query** (*Any*) –

Return type DataFrame

5.16.9 Transactions

Examples

Reference

async ctc.evm.**async_get_transaction**(*transaction_hash*)

Parameters **transaction_hash** (*<class 'str'>*) –

Return type <class 'Transaction'>

async ctc.evm.**async_get_transaction_count**(*address*)

Parameters **address** (*<class 'str'>*) –

Return type <class 'int'>

5.17 Specific Protocols

5.17.1 4byte

Examples

Reference

async ctc.protocols.fourbyte_utils.**async_build_event_signatures_dataset**()

Return type <class 'NoneType'>

async ctc.protocols.fourbyte_utils.**async_build_function_signatures_dataset**()

Return type <class 'NoneType'>

async ctc.protocols.fourbyte_utils.**async_query_event_signature**(*hex_signature=None*, *, *id=None*, *bytes_signature=None*, *text_signature=None*, *source='local'*)

async ctc.protocols.fourbyte_utils.**async_query_function_signature**(*hex_signature=None*, *, *id=None*, *bytes_signature=None*, *text_signature=None*, *source='local'*)

ctc.protocols.fourbyte_utils.**local_event_signatures_exist**()

Return type <class 'bool'>

ctc.protocols.fourbyte_utils.**local_function_signatures_exist**()

Return type <class 'bool'>

5.17.2 Aave V2

Examples

Reference

async ctc.protocols.aave_v2_utils.**async_get_deposits**(*start_block=None*, *end_block=None*, *provider=None*)

Parameters

- **start_block** (*BlockNumberReference | None*) –
- **end_block** (*BlockNumberReference | None*) –
- **provider** (*ProviderSpec*) –

Return type DataFrame

async ctc.protocols.aave_v2_utils.**async_get_interest_rates**(*token*, *block=None*)

Parameters

- **token** (*Address*) –
- **block** (*BlockNumberReference | None*) –

Return type dict[str, float]

async ctc.protocols.aave_v2_utils.**async_get_interest_rates_by_block**(*token*, *blocks*)

async ctc.protocols.aave_v2_utils.**async_get_reserve_data**(*asset*, *block=None*)

Parameters

- **asset** (*Address*) –
- **block** (*BlockNumberReference | None*) –

Return type AaveV2ReserveData

async ctc.protocols.aave_v2_utils.**async_get_reserve_data_by_block**(*asset*, *blocks*)

Parameters

- **asset** (*<class 'str'>*) –
- **blocks** (*Sequence[Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]]*) –

Return type <class 'aave_v2AaveV2ReserveListData'>

async ctc.protocols.aave_v2_utils.**async_get_underlying_asset**(*pool_token*, *provider=None*)

Parameters

- **pool_token** (*<class 'str'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –

Return type <class 'str'>

async ctc.protocols.aave_v2_utils.**async_get_withdrawals**(*start_block=None*, *end_block=None*, *provider=None*)

Parameters

- **start_block** (*BlockNumberReference | None*) –
- **end_block** (*BlockNumberReference | None*) –
- **provider** (*ProviderSpec*) –

Return type DataFrame

5.17.3 Balancer

Examples

Reference

async ctc.protocols.balancer_utils.**async_get_pool_address**(*pool_id*, *block=None*, *vault=None*)

Parameters

- **pool_id** (*str*) –
- **block** (*BlockNumberReference | None*) –
- **vault** (*Address | None*) –

Return type Address

async ctc.protocols.balancer_utils.**async_get_pool_balances**(*, *pool_address=None*, *pool_id=None*, *block='latest'*, *vault=None*, *normalize=True*)

async ctc.protocols.balancer_utils.**async_get_pool_fees**(*pool_address*, *block='latest'*, *normalize=True*)

Parameters

- **pool_address** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]*) –
- **normalize** (*<class 'bool'>*) –

Return type Union[int, float]

async ctc.protocols.balancer_utils.**async_get_pool_id**(*pool_address*, *block=None*)

Parameters

- **pool_address** (*Address*) –
- **block** (*BlockNumberReference | None*) –

Return type str

async ctc.protocols.balancer_utils.**async_get_pool_swaps**(*pool_address=None*, *start_block=None*, *end_block=None*, *vault=None*)

Parameters

- **pool_address** (*Union[str, NoneType]*) –
- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **vault** (*Union[str, NoneType]*) –

Return type Any

async ctc.protocols.balancer_utils.**async_get_pool_tokens**(*, *pool_address=None*, *pool_id=None*, *block=None*, *vault=None*)

Parameters

- **pool_address** (*Address | None*) –
- **pool_id** (*str | None*) –
- **block** (*BlockNumberReference | None*) –
- **vault** (*Address | None*) –

Return type list[Address]

async ctc.protocols.balancer_utils.**async_get_pool_weights**(*pool_address*, *block='latest'*, *normalize=True*)

async ctc.protocols.balancer_utils.**async_get_pool_weights_by_block**(*pool_address*, *blocks*, *normalize=True*)

async ctc.protocols.balancer_utils.**async_summarize_pool_state**(*pool_address*, *block='latest'*)

Parameters

- **pool_address** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]*) –

Return type <class 'BalancerPoolState'>

5.17.4 Chainlink

Examples

Reference

async ctc.protocols.chainlink_utils.**async_summarize_feed**(*feed*, *n_recent=10*)

Parameters

- **feed** (*<class 'str'>*) –
- **n_recent** (*<class 'int'>*) –

Return type <class 'NoneType'>

5.17.5 Compound

Examples

Reference

async ctc.protocols.compound_utils.**async_get_borrow_apy**(*ctoken*, *block=None*)

Parameters

- **ctoken** (*Address*) –
- **block** (*BlockNumberReference | None*) –

Return type float

async ctc.protocols.compound_utils.**async_get_borrow_apy_by_block**(*ctoken*, *blocks*)

async ctc.protocols.compound_utils.**async_get_supply_apy**(*ctoken*, *block=None*)

Parameters

- **ctoken** (*Address*) –
- **block** (*BlockNumberReference | None*) –

Return type float

async ctc.protocols.compound_utils.**async_get_supply_apy_by_block**(*ctoken*, *blocks*)

5.17.6 Curve

Examples

Reference

async ctc.protocols.curve_utils.**async_get_base_pools**(*start_block=None*, *end_block=None*, *provider=None*, *verbose=False*)

Parameters

- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –

- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **verbose** (*<class 'bool'>*) –

Return type Any

async ctc.protocols.curve_utils.**async_get_meta_pools**(*start_block=None*, *end_block=None*, *provider=None*, *verbose=False*)

Parameters

- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **verbose** (*<class 'bool'>*) –

Return type Any

async ctc.protocols.curve_utils.**async_get_plain_pools**(*start_block=None*, *end_block=None*, *provider=None*, *verbose=False*)

Parameters

- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **verbose** (*<class 'bool'>*) –

Return type Any

async ctc.protocols.curve_utils.**async_get_pool_metadata**(*pool*, *n_tokens=None*, *provider=None*)

Parameters

- **pool** (*Address*) –
- **n_tokens** (*int | None*) –
- **provider** (*ProviderSpec*) –

Return type CurvePoolMetadata

async ctc.protocols.curve_utils.**async_get_pool_state**(*pool*, *n_tokens=None*, *block=None*, *provider=None*, *normalize=True*)

Parameters

- **pool** (*Address*) –
- **n_tokens** (*int | None*) –
- **block** (*BlockNumberReference | None*) –
- **provider** (*ProviderSpec*) –

- **normalize** (*bool*) –

Return type dict

async ctc.protocols.curve_utils.**async_get_virtual_price**(*pool*, *provider=None*, *block=None*)

Parameters

- **pool** (*Address*) –
- **provider** (*ProviderSpec*) –
- **block** (*BlockNumberReference | None*) –

Return type int

5.17.7 ENS

Examples

Reference

async ctc.protocols.ens_utils.**async_get_expiration**(*name*)

Parameters **name** (*<class 'str'>*) –

Return type <class 'int'>

async ctc.protocols.ens_utils.**async_get_owner**(*name*, *provider=None*, *block=None*)

Parameters

- **name** (*str*) –
- **provider** (*ProviderSpec*) –
- **block** (*BlockNumberReference | None*) –

Return type str

async ctc.protocols.ens_utils.**async_get_registration_block**(*name*)

Parameters **name** (*<class 'str'>*) –

Return type <class 'int'>

async ctc.protocols.ens_utils.**async_get_registrations**()

Return type Any

async ctc.protocols.ens_utils.**async_get_text_records**(*name=None*, *node=None*, *keys=None*)

https://docs.ens.domains/ens-improvement-proposals/ensip-5-text-records

Parameters

- **name** (*str | None*) –
- **node** (*str | None*) –
- **keys** (*Sequence[str] | None*) –

Return type dict[str, str]

async ctc.protocols.ens_utils.**async_record_exists**(*name*, *provider=None*, *block=None*)

Parameters

- **name** (*str*) –
- **provider** (*ProviderSpec*) –
- **block** (*BlockNumberReference | None*) –

Return type bool

async ctc.protocols.ens_utils.**async_reverse_lookup**(*address*, *provider=None*, *block=None*)

Parameters

- **address** (*Address*) –
- **provider** (*ProviderSpec*) –
- **block** (*BlockNumberReference | None*) –

Return type str

ctc.protocols.ens_utils.**hash_name**(*name*)

Parameters **name** (*<class 'str'>*) –

Return type <class 'str'>

5.17.8 Fei

Examples

Reference

async ctc.protocols.fei_utils.**async_create_payload**(*, *blocks=None*, *timestamps=None*, *timescale=None*, *end_time=None*, *window_size=None*, *interval_size=None*, *provider=None*)

create data payload from scratch

Parameters

- **blocks** (*Sequence[BlockNumberReference] | None*) –
- **timestamps** (*Sequence[int] | None*) –
- **timescale** (*TimescaleSpec | None*) –
- **end_time** (*Timestamp | None*) –
- **window_size** (*str | None*) –
- **interval_size** (*str | None*) –
- **provider** (*ProviderSpec*) –

Return type AnalyticsPayload

async ctc.protocols.fei_utils.**async_get_pcv_stats**(*block=None*, *wrapper=False*, *provider=None*)

Parameters

- **block** (*BlockNumberReference | None*) –
- **wrapper** (*bool*) –
- **provider** (*ProviderSpec*) –

Return type FeiPcvStats

async ctc.protocols.fei_utils.**async_get_pcv_stats_by_block**(*blocks*, *wrapper=False*, *provider=None*)

Parameters

- **blocks** (*Sequence[Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]]*) –
- **wrapper** (*<class 'bool'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –

Return type Any

async ctc.protocols.fei_utils.**async_print_pcv_assets**(*block=None*)

Parameters **block** (*BlockNumberReference | None*) –

Return type None

async ctc.protocols.fei_utils.**async_print_pcv_deposits**(*block=None*)

Parameters **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –

Return type <class 'NoneType'>

5.17.9 Rari

Examples

Reference

async ctc.protocols.rari_utils.**async_get_all_pools**(*block=None*, *provider=None*)

async ctc.protocols.rari_utils.**async_get_ctoken_state**(*ctoken*, *block='latest'*, *metrics=None*, *eth_price=None*, *in_usd=True*)

Parameters

- **ctoken** (*Address*) –
- **block** (*BlockNumberReference*) –
- **metrics** (*CTokenMetricSpec | None*) –
- **eth_price** (*Number | None*) –
- **in_usd** (*bool*) –

Return type CTokenMetrics

async ctc.protocols.rari_utils.**async_get_ctoken_state_by_block**(*ctoken*, *blocks*, *metrics=None*, *eth_price=None*, *in_usd=True*)

Parameters

- **ctoken** (*Address*) –
- **blocks** (*Sequence[BlockNumberReference]*) –
- **metrics** (*CTokenMetricSpec | None*) –
- **eth_price** (*Number | None*) –
- **in_usd** (*bool*) –

Return type CTokenMetricsByBlock

async ctc.protocols.rari_utils.**async_get_pool_ctokens**(*comptroller*, *block='latest'*)

async ctc.protocols.rari_utils.**async_get_pool_prices**(*, *oracle=None*, *ctokens=None*, *comptroller=None*, *block='latest'*, *to_usd=True*)

Parameters

- **oracle** (*Address | None*) –
- **ctokens** (*Sequence[Address] | None*) –
- **comptroller** (*Address | None*) –
- **block** (*BlockNumberReference*) –
- **to_usd** (*bool*) –

Return type dict[Address, Number]

async ctc.protocols.rari_utils.**async_get_pool_tvl_and_tvb**(*, *comptroller=None*, *ctokens=None*, *oracle=None*, *block='latest'*)

Parameters

- **comptroller** (*Address | None*) –
- **ctokens** (*Sequence[Address] | None*) –
- **oracle** (*Address | None*) –
- **block** (*BlockNumberReference*) –

Return type dict[str, Number]

async ctc.protocols.rari_utils.**async_get_pool_underlying_tokens**(*, *ctokens=None*, *comptroller=None*, *block='latest'*)

Parameters

- **ctokens** (*Sequence[Address] | None*) –
- **comptroller** (*Address | None*) –
- **block** (*BlockNumberReference*) –

Return type dict[Address, Address]

async ctc.protocols.rari_utils.**async_get_token_multipool_stats**(*token*, *block='latest'*, *in_usd=True*)

Parameters

- **token** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]*) –
- **in_usd** (*<class 'bool'>*) –

Return type <class 'dict'>

async ctc.protocols.rari_utils.**async_print_all_pool_summary**(*block='latest'*, *n_display=15*)

Parameters

- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]*) –
- **n_display** (*<class 'int'>*) –

Return type <class 'NoneType'>

async ctc.protocols.rari_utils.**async_print_fuse_token_summary**(*token*, *block='latest'*, *in_usd=True*)

Parameters

- **token** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]*) –
- **in_usd** (*<class 'bool'>*) –

Return type <class 'NoneType'>

5.17.10 Uniswap V2

Examples

Reference

async ctc.protocols.uniswap_v2_utils.**async_get_pool_burns**(*pool_address*, *start_block=None*, *end_block=None*, *replace_symbols=False*, *normalize=True*, *provider=None*, *verbose=False*)

Parameters

- **pool_address** (*<class 'str'>*) –
- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **replace_symbols** (*<class 'bool'>*) –
- **normalize** (*<class 'bool'>*) –

- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **verbose** (*<class 'bool'>*) –

Return type Any

async ctc.protocols.uniswap_v2_utils.**async_get_pool_decimals**(*pool=None*, ***, *x_address=None*, *y_address=None*, *provider=None*)

Parameters

- **pool** (*Address | None*) –
- **x_address** (*Address | None*) –
- **y_address** (*Address | None*) –
- **provider** (*ProviderSpec*) –

Return type list[int]

async ctc.protocols.uniswap_v2_utils.**async_get_pool_mints**(*pool_address*, *start_block=None*, *end_block=None*, *replace_symbols=False*, *normalize=True*, *provider=None*, *verbose=False*)

Parameters

- **pool_address** (*<class 'str'>*) –
- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **replace_symbols** (*<class 'bool'>*) –
- **normalize** (*<class 'bool'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **verbose** (*<class 'bool'>*) –

Return type Any

async ctc.protocols.uniswap_v2_utils.**async_get_pool_state**(*pool*, ***, *block='latest'*, *provider=None*, *normalize=True*, *fill_empty=True*)

Parameters

- **pool** (*<class 'str'>*) –
- **block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **normalize** (*<class 'bool'>*) –
- **fill_empty** (*<class 'bool'>*) –

Return type <class 'uniswap_v2uniswap_v2PoolState'>

async ctc.protocols.uniswap_v2_utils.**async_get_pool_state_by_block**(*pool*, *, *blocks*, *provider=None*, *normalize=True*)

Parameters

- **pool** (*<class 'str'>*) –
- **blocks** (*Sequence[Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending']]]*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **normalize** (*<class 'bool'>*) –

Return type <class 'uniswap_v2uniswap_v2PoolStateByBlock'>

async ctc.protocols.uniswap_v2_utils.**async_get_pool_swaps**(*pool_address*, *start_block=None*, *end_block=None*, *replace_symbols=False*, *normalize=True*, *provider=None*, *verbose=False*)

Parameters

- **pool_address** (*<class 'str'>*) –
- **start_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **end_block** (*Union[SupportsRound, str, int, Literal['latest'], Literal['earliest'], Literal['pending'], NoneType]*) –
- **replace_symbols** (*<class 'bool'>*) –
- **normalize** (*<class 'bool'>*) –
- **provider** (*Union[str, PartialProvider, Provider, NoneType]*) –
- **verbose** (*<class 'bool'>*) –

Return type Any

async ctc.protocols.uniswap_v2_utils.**async_get_pool_symbols**(*pool=None*, *, *x_address=None*, *y_address=None*, *provider=None*)

Parameters

- **pool** (*Address | None*) –
- **x_address** (*Address | None*) –
- **y_address** (*Address | None*) –
- **provider** (*ProviderSpec*) –

Return type list[str]

async ctc.protocols.uniswap_v2_utils.**async_get_pool_tokens**(*pool*, *provider=None*)

ctc.toolbox.amm_utils.cpmm.**print_pool_summary**(*x_reserves*, *y_reserves*, *lp_total_supply=None*, *x_name=None*, *y_name=None*, *fee_rate=None*, *indent=None*, *depths=None*)

Parameters

- **x_reserves** (*int | float*) –
- **y_reserves** (*int | float*) –
- **lp_total_supply** (*int | float | None*) –
- **x_name** (*str | None*) –
- **y_name** (*str | None*) –
- **fee_rate** (*float | None*) –
- **indent** (*int | str | None*) –
- **depths** (*Sequence[float] | None*) –

Return type None

ctc.toolbox.amm_utils.cpmm.**print_trade_summary**(*x_name=None*, *y_name=None*, *x_holdings_before=None*, *y_holdings_before=None*, *indent=None*, ***trade_kwargs*)

Parameters

- **x_name** (*str | None*) –
- **y_name** (*str | None*) –
- **x_holdings_before** (*int | float | None*) –
- **y_holdings_before** (*int | float | None*) –
- **indent** (*int | str | None*) –
- **trade_kwargs** (*Any*) –

Return type None

ctc.toolbox.amm_utils.cpmm.**trade**(*x_reserves*, *y_reserves*, *x_sold=None*, *x_bought=None*, *y_sold=None*, *y_bought=None*, *new_x_reserves=None*, *new_y_reserves=None*, *fee_rate=None*)

perform trade with AMM

Input Requirements - all input values must be positive - must always specify both x_reserves and y_reserves - must specify exactly one of:

- x_sold
- x_bought
- y_sold
- y_bought
- new_x_reserves
- new_y_reserves
- values in this list can be scalars or numpy arrays

Parameters

- **x_reserves** (*int | float*) –
- **y_reserves** (*int | float*) –
- **x_sold** (*int | float | None*) –

- **x_bought** (*int | float | None*) –
- **y_sold** (*int | float | None*) –
- **y_bought** (*int | float | None*) –
- **new_x_reserves** (*int | float | None*) –
- **new_y_reserves** (*int | float | None*) –
- **fee_rate** (*int | float | None*) –

Return type Trade

5.17.11 Uniswap V3

Examples

Reference

async ctc.protocols.uniswap_v3_utils.**async_get_pool_metadata**(*pool_address*, ***rpc_kwargs*)

Parameters

- **pool_address** (*<class 'str'>*) –
- **rpc_kwargs** (*Any*) –

Return type <class 'uniswap_v3uniswap_v3UniswapV3PoolMetadata'>

async ctc.protocols.uniswap_v3_utils.**async_get_pool_swaps**(*pool_address*, *start_block=None*, *end_block=None*, *replace_symbols=False*, *normalize=True*)

Parameters

- **pool_address** (*Address*) –
- **start_block** (*BlockNumberReference | None*) –
- **end_block** (*BlockNumberReference | None*) –
- **replace_symbols** (*bool*) –
- **normalize** (*bool*) –

Return type DataFrame

async ctc.protocols.uniswap_v3_utils.**async_quote_exact_input_single**(*token_in*, *token_out*, *fee*, *amount_in*, *sqrt_price_limit_x96=0*, *provider=None*, *block=None*)

Parameters

- **token_in** (*Address*) –
- **token_out** (*Address*) –
- **fee** (*int*) –
- **amount_in** (*int*) –

- **sqrt_price_limit_x96** (*int*) –
- **provider** (*ProviderSpec*) –
- **block** (*BlockNumberReference | None*) –

Return type int

async ctc.protocols.uniswap_v3_utils.**async_quote_exact_output_single**(*token_in*, *token_out*, *fee*, *amount_out*, *sqrt_price_limit_x96=0*, *provider=None*, *block=None*)

Parameters

- **token_in** (*Address*) –
- **token_out** (*Address*) –
- **fee** (*int*) –
- **amount_out** (*int*) –
- **sqrt_price_limit_x96** (*int*) –
- **provider** (*ProviderSpec*) –
- **block** (*BlockNumberReference | None*) –

Return type int

Protocol	Examples	Reference	Source
4byte	Examples	Reference	Source
Aave V2	Examples	Reference	Source
Balancer	Examples	Reference	Source
Chainlink	Examples	Reference	Source
Compound	Examples	Reference	Source
Curve	Examples	Reference	Source
ENS	Examples	Reference	Source
Fei	Examples	Reference	Source
Rari	Examples	Reference	Source
Uniswap V2	Examples	Reference	Source
Uniswap V3	Examples	Reference	Source

5.18 Similar Python Tools

5.18.1 web3.py

web3.py is a python library that is created, hosted, and maintained by the Ethereum Foundation. Although web3.py and `ctc` have some overlapping functionality, they focus on different things. Web3.py supports full wallet functionality, whereas `ctc` is currently limited to read-only operations. Web3.py also supports a greater variety of communication protocols including websockets.

On the other hand, `ctc` is primarily aimed at historical data analysis. It contains more functions for aggregating historical datasets from various on-chain protocols. Additionally, web3.py is primarily synchronous, whereas `ctc` is primarily asynchronous.

5.18.2 ethtx

ethtx is a library for decoding and summarizing individual transactions. You can see it in action at https://ethtx.info/. Although `ctc` has its own transaction summarizing capabilities, it is currently much more limited than ethtx when it comes to tracing internal transactions and revealing the resultant state changes. These types of features may come to `ctc` in a future release.

5.19 [Under Construction]

come back later

INDEX

A

B

D

T