Check the Chain (ctc)

Sep 25, 2022

THE BASICS

1 Features 3

2 Guide 5

3 Datatypes 7

4 Specific Protocols 9

5 External Data Sources 11

Index 53

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

These docs are also a work in progress. Some sections are not yet complete. Feel free to report any documentation-related issues 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 view the `ctc` source code, check out the GitHub Repository.

CHAPTER

THREE

DATATYPES

Datatype	CLI	Python	Source
ABIs	CLI	Python	Source
Addresses	CLI	Python	Source
Binary Data	CLI	Python	Source
Blocks	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	CLI	Python	Source
Balancer	CLI	Python	Source
Chainlink	CLI	Python	Source
Compound	-	Python	Source
Curve	CLI	Python	Source
ENS	CLI	Python	Source
Fei	CLI	Python	Source
Gnosis Safe	CLI	Python	Source
Multicall	CLI	Python	Source
Rari	CLI	Python	Source
Uniswap V2	CLI	Python	Source
Uniswap V3	CLI	Python	Source
Yearn	CLI	Python	Source

CHAPTER

FIVE

EXTERNAL DATA SOURCES

Data Source	CLI	Python	Source
4byte	CLI	Python	Source
CoinGecko	CLI	Python	Source
Defi Llama	CLI	Python	Source
Etherscan	CLI	Python	Source

5.1 Installation

5.1.1 Basic Installation

Installing `ctc` takes 2 steps:

1. `pip install checkthechain`
2. `ctc setup` in the terminal to run the setup wizard (can skip most steps by pressing enter)

See *Configuration* for additional setup options.

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

Installation requires python 3.7 or greater. 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)

If you previously installed `ctc` directly from a git commit, you may need to first `pip uninstall checkthechain` before upgrading.

When upgrading you should also check the *changelog* for

5.1.3 Uninstalling

Fully removing `ctc` from a machine takes three steps:

1. Uninstall the package `pip uninstall checkthechain`
2. Remove the config folder: `rm -rf ~/.config/ctc`
3. Remove the data folder: `rm -rf ~/ctc_data`

You can always check whether a package has been uninstalled from your python environment by attempting to import it in a fresh shell. If you see a `ModuleNotFoundError`, the package has been uninstalled.

```
>>> import ctc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'ctc'
>>> # ctc is not uninstalled
```

5.1.4 Special 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 ./
```

5.1.5 Libraries

On a fresh installation of Ubuntu or Debian, you may need to manually install the `build-essential` and `python-dev` packages. Machines that are used for active python development probably already have these packages installed.

```
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 or greater.

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 these are already installed on your machine. 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 these os dependencies 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 configuration file to control its behavior.

5.3.1 Setting Config Parameters

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

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.

`ctc` can also function under a "no-config" mode, where `ctc setup` does not need to be run. To use this mode, simply set the `ETH_RPC_URL` to an RPC provider url.

5.3.2 Config Parameters

The config file consists of key-value pairs. The keys:

- **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 networks including their names and chain_id's
- **default_network**: default network to use
- **default_providers**: default provider for each network
- **db_configs**: database configuration information
- **log_rpc_calls**: whether to log rpc calls
- **log_sql_queries**: whether to log sql queries

A python type specification for the config can be found in the config typedefs file.

5.3.3 Reading Config Parameters

On the command line, running `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 config data:

```
from ctc import config

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

5.4 Changelog

until version 1.0.0 is reached, will use 0.X.Y versioning where X is for breaking changes / major feature upgrades, and Y is for bug fixes / minor feature upgrades

5.4.1 0.3.0

September 25, 2022

This is a significant release that includes features such as: sql database integration, refactored documentation, streamlined syntax, performance optimizations, and many new types of data queries. This release also includes lots of small bug fixes and quality-of-life improvements not listed below.

DB

- integrate sql db for storing collected data
- create tables for: blocks, contract abis, contract creation blocks, ERC20 metadata, 4byte signatures, and Chainlink feeds
- add flags to functions for whether db should be used
- auto-intake collected data into db by default

Documentation

- create external documentation page https://ctc.readthedocs.io/en/latest

CLI

- add help messages and examples to each subcommand
- add color to cli output
- optimize cli startup times
- allow all cli commands to use ens names in place of addresses
- add many subcommands including
  - storage, limits, encode, proxy, bytecode, chains, selector
  - `abi decompile` command for decoding ABI of solidity and vyper contracts
  - (see `ctc -h` for proper full list)
  - XX do a diff with 3.10?
- add commands for events, candlestick charting
- add `--json` to many cli commands to output data as json

Config

- make configuration file optional by using a default config and looking for RPC provider in `ETH_RPC_URL` env var
- when loading old config versions, attempt to transparently convert it to new config version
- added better config validation
- add shell alias configuration to `ctc setup`

Protocols

- new protocol-specific functionality for Gnosis and Etherscan
- add subcommands to previous covered protocols
- use binary search to implement trade-to-price function for Uniswap V3 and other AMMs

Data Operations

- new transaction and call data decoding system
- automatically query proxy ABI when querying a contract's ABI
  - if a function ABI or event ABI cannot be found, re-query contract proxy to check for ABI updates
- add functionality for fetching all transactions of a given address
- add functionality for predicting block numbers of future timestamps

Testing

- use tox for testing each python version
- create legacy test environment with minimal version of each dependency
- test that all cli commands have examples and test that the examples work
- enforce many coding conventions using tests

Performance

- utilize caches and concurrency when possible
- add appropriate rate limits for etherscan and 4byte for scraping

Python

- upgrade from `setuptools` / `setup.py` to `flit` / `pyproject.toml`
- use black for all py files in repo
- use strict mode for mypy typing annotations
- reduce number of implicit package dependencies by more than 50%
  - fork `eth-abi` package as `eth-abi-lite` to remove dependence on `eth-abi`, `eth-utils`, `toolz` and `cytools`
  - specify min and max version of each dependency to prevent future backwards incompability

Other

- add logging system and allow use of `ctc log` command to follow logs
- populate default data directory with metadata of: 22 networks, >1000 ERC20 tokens, and all Chainlink feeds
- add functions for converting block numbers into timestamps for x-axis labels of plots

Upgrade Guide

1. Run `pip install -U checkthechain`
2. Run `ctc setup`
3. There are some minor api changes (see below)

API Changes

Version `0.3.0` contains some breaking changes to make the API more consistent and intuitive. Care was taken to minimize these breaking changes. Future versions of `ctc` will aim to maximize backward compatilibity as much as possible.

- `config` (running `ctc setup` command will automatically upgrade old config and data directory)
  - new config schema using flat structure instead of nested hierarchy (see `ctc.spec.typedefs.config_types`)
  - new data directory schema that better reflects underlying data relationships (see `ctc.config.upgrade_utils.data_dir_versioning`)
- `directory` deprecated in favor of functions in `config`, `db`, and `evm`
- `evm`
  - `decode_function_output()` arg: `package_named_results` –> `package_named_outputs`
  - `async_get_proxy_address()` –> `async_get_proxy_implementation()`
  - erc20 balance and allowance functions:
    - arg `address` –> `wallet`
    - arg `addresses` –> `wallets`
    - `async_get_erc20_holdings_from_transfers` –> `async_get_erc20_balances_from_transfers`
    - `async_get_block_timestamp()` modes renamed from `before`, `after`, `equal` to <=, >=, ==
    - `async_get_erc20_balance_of` –> `async_get_erc20_balance`
    - `async_get_erc20_balance_of_addresses` –> `async_get_erc20_balances_of_addresses`
    - `async_get_erc20s_balance_of` –> `async_get_erc20s_balances`
    - `async_get_erc20_balance_of_by_block` –> `async_get_erc20_balance_by_block`
    - `async_get_erc20s_allowances_by_address` –> `async_get_erc20s_allowances_of_addresses`
- `protocols`
  - `curve_utils.async_get_pool_addresses` –> `curve_utils.async_get_pool_tokens`
  - `rari_utils.get_pool_tvl_and_tvb` –> `rari_utils.async_get_pool_tvl_and_tvb`
  - use for blockwise functions always use `by_block` rather than `per_block`

  - uniswap_v2_utils.async_get_pool_swaps –> uniswap_v2_utils.async_get_pool_trades
  - functions for querying data from specific DEX's now all use unified a unified DEX syntax and API
- spec
  - ConfigSpec –> Config
  - PartialConfigSpec –> PartialConfig
  - ProviderSpec –> ProviderReference
- toolbox
  - move toolbox.amm_utils, toolbox.twap_utils, and toolbox.lending_utils under toolbox.defi_utils
- cli
  - all commands are standardized on --export rather than --output to specify data export targets
- for functions that print out summary information, instead of using a conventions of print_<X>() and summarize_<X>, use single convention print_X()
- only allow positional arguments for the first two arguments of every function

5.4.2 0.2.10

March 26, 2022

- add functionality for G-Uni Gelato, multicall
- add Fei yield dashboard analytics
- add commands for ABI summarization
- signficantly improve test coverage

5.4.3 0.2.9

March 18, 2022

- add Uniswap V3 functionliaty
- improve Chainlink functions, commands, and feed registry
- add twap_utils
- add cli aliases
- many small fixes
- handle various types of non-compliant erc20s

5.4.4 0.2.8

March 2, 2022

- fix str processing bug

5.4.5 0.2.7

March 2, 2022

- add robustness and quality-of-life improvements to data cache
- add 4byte functionality
- add Coingecko functionality

5.4.6 0.2.6

February 24, 2022

- fix many typing annotation issues
- add Curve functionality
- add Fei functionality

5.4.7 0.2.5

February 16, 2022

- add ENS functionality
- add hex, ascii, checksum, and lower cli commands
- add Rari lens

5.4.8 0.2.4

February 15, 2022

- python 3.7 compatibility fixes

5.4.9 0.2.3

February 14, 2022

- add many cli commands
- refactor existing cli commands

5.4.10 0.2.2

February 11, 2022

- add python 3.7 and python3.8 compatibility

5.4.11 0.2.1

February 9, 2022

initial public `ctc` release

5.5 FAQ

5.5.1 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.
- **Protocol-specific functionality**: This means having built-in support for popular on-chain protocols.
- **Terminal-based block explorer**: This means supporting as many block explorer tasks as possible from the terminal. And doing so in a way that is faster than can be done with a web browser.
- **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.

5.5.2 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.

5.5.3 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.

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

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

5.6 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*.

5.6.1 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 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*.

5.6.2 Other types of data

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

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

5.7 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 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.

5.7.1 Data Storage Backends

`ctc` uses two main storage backends.

Filesystem

`ctc` stores some files on the filesystem. By default, `ctc` will place its data folder in the user's home directory at `~/ctc_data`. 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`.

Total storage usage of `ctc` on the filesystem can be found by checking the size of the `ctc` data directory.

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.

Total storage usage of `ctc` in the database can be found by running `ctc db -v` in the terminal.

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

5.8 Performance

TLDR

Even in suboptimal conditions, `ctc` uses 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`.

5.8.1 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 much of the data that it retrieves. However, it's possible that alternate settings might be optimal in different contexts. For example if `ctc` is run on the same server as an archive node, then it's possible that certain 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 will make 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 consumes 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`.

5.8.2 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 the 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 -v`.

5.9 Monitoring

5.9.1 Logging

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

Logs are stored in `logs` subdirectory of the `ctc` data dir (default = `~/ctc_data`).

Running `ctc log` in the terminal will begin watching for changes to 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`).

5.9.2 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.

5.10 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. To view the complete list of subcommands use *ctc -h*.

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

5.11 Subcommands

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

5.11.1 Admin Subcommands

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

aliases

chains

config

db

log

setup

5.11.2 Compute Subcommands

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

ascii

checksum

create address

decode

decode call

encode

hex

int

keccak

limits

lower

rlp encode

selector

5.11.3 Data Subcommands

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

abi

abi diff

address

address txs

block

blocks

bytecode

call

call all

calls

chain

decompile

dex chart

dex pool

dex pools

dex trades

erc20

erc20 balance

erc20 balances

erc20 transfers

eth balance

eth balances

events

gas

proxy

proxy register

storage

symbol

timestamp

tx

5.11.4 Protocol Subcommands

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

4byte

4byte build

aave

aave addresses

cg

chainlink

chainlink feeds

curve pools

ens

ens exists

ens hash

ens owner

ens records

ens resolve

ens reverse

etherscan

fei analytics

fei depth

fei dex

fei pcv

fei pcv assets

fei pcv deposits

fei psms

gnosis

llama

llama chain

llama chains

llama pool

llama pools

llama protocol

llama protocols

rari

rari pools

uniswap burns

uniswap chart

uniswap mints

uniswap pool

uniswap swaps

yearn

yearn addresses

5.11.5 Other Subcommands

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

cd

help

version

5.12 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.

Such 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.

The `ctc setup` wizard can add these aliases to your shell configuation.

5.12.1 The Aliases

These aliases are chosen so as not to conflict with any common CLI tools.

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

# data commands
alias abi="ctc abi"
alias address="ctc address"
alias block="ctc block"
alias blocks="ctc blocks"
alias bytecode="ctc bytecode"
alias call="ctc call"
alias calls="ctc calls"
alias dex="ctc dex"
alias erc20="ctc erc20"
alias eth="ctc eth"
alias gas="ctc gas"
alias int="ctc int"
alias rlp="ctc rlp"
alias tx="ctc tx"

# protocol commands
alias 4byte="ctc 4byte"
alias aave="ctc aave"
alias cg="ctc cg"
alias chainlink="ctc chainlink"
alias curve="ctc curve"
alias es="ctc etherscan"
alias ens="ctc ens"
alias fei="ctc fei"
alias gnosis="ctc gnosis"
alias llama="ctc llama"
alias rari="ctc rari"
alias uniswap="ctc uniswap"
alias yearn="ctc yearn"
```

5.13 Similar CLI tools

5.13.1 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 stack 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` falls 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

5.13.2 TrueBlocks

TrueBlocks is a tool for managing optimized local indices of EVM chain data. TrueBlocks then makes these local data copies accessible through an enhanced RPC interface. TrueBlocks delivers some of the highest performance ways to query chain data and it excels at tracing and querying all appearances of a given address throughout a chain's history. Since TrueBlocks can provide its data over RPC, it could be used as an ultra high performance RPC provider for `ctc`.

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.

5.13.3 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.

5.14 Basic Usage

The top-level `ctc` module contains functions for generic EVM operations:

Example: Generic EVM Operations

```
import ctc

some_hash = ctc.keccak_text('hello')

encoded_data = ctc.abi_encode_packed((400, 6000), '(int128,int128)')

eth_balance = await ctc.async_get_eth_balance('0x6b175474e89094c44da98b954eedeac495271d0f
↪')

erc20_balance = await ctc.async_get_erc20_balance(
    token='0x6b175474e89094c44da98b954eedeac495271d0f',
    wallet='0x6b175474e89094c44da98b954eedeac495271d0f'],
```

(continues on next page)

(continued from previous page)

```
    block=15000000,
)

events = await ctc.async_get_events(
    '0xcbcdf9626bc03e24f779434178a73a0b4bad62ed',
    event_name='Swap',
)
```

Some points to keep in mind while using `ctc`:

- `ctc` uses functional programming. Instead of custom types or OOP, `ctc` uses simple standard datatypes including python builtins and numpy arrays. There is no need to initialize any objects. Simply `import ctc` and then call functions in the `ctc.*` namespace.
- `ctc` is asynchronous-first, which allows it to efficiently orchestrate large numbers of interdependent queries. *Special consideration* is needed to run code in an asynchronous context.
- `ctc` is designed with historical data analysis in mind. For any query of EVM state, `ctc` aims to support historical versions of that query. Most `ctc` query functions take parameters that can specify a block or block range relevant to the query.

The top-level `ctc` package covers generic EVM operations, which are described in more detail *here*. There are also a few other `ctc` subpackages that are relevant to specific use-cases described below.

5.14.1 RPC Client Subpackage `ctc.rpc`

`ctc.rpc` implements `ctc`'s custom RPC client. This client can be used for fine-grained control over RPC calls. Unless explcitly told not to do so.`ctc` will automatically encode requests to binary and decode requests from binary.

Example: get bytecode for contract, at specific block, using specific provider

```
import ctc.rpc

contract_bytecode = await ctc.rpc.async_eth_get_code(
    '0x6b175474e89094c44da98b954eedeac495271d0f',
    block_number=15000000,
    provider='https://some_rpc_node/',
)
```

5.14.2 Protocol-specific Subpackages `ctc.protocols`

`ctc.protocols` contains functions specific to many different protocols such as Chainlink or Uniswap. See a full list *here*.

Example: gather complete historical data for Chainlink's RAI-USD feed

```
from ctc.protocols import chainlink_utils

feed_data = await chainlink_utils.async_get_feed_data('RAI_USD')
```

5.14.3 Other Subpackages

End users of `ctc` probably won't need to use any of these directly.

- `ctc.cli`: command line interface
- `ctc.config`: configuration utilities
- `ctc.db`: local cache database
- `ctc.spec`: ctc specifications, mainly types for type annotations
- `ctc.toolbox`: miscellaneous python utilities

5.15 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.

5.15.1 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, dispatching, 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)

5.15.2 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.

5.15.3 Typical RPC Request Lifecycle in `ctc`

1. a constructor function encodes request metadata and parameters into a `RpcRequest` python `dict`
2. the request is dispatched to an rpc provider using `rpc.async_send_http()`
3. the client `awaits` 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 `eth_getLogs`), `ctc` will encode/decode inputs/outputs using the relevant function abi's and event abi's.

5.16 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))
```

Inside of IPython or Jupyter notebooks, `await` can be used directly, without `asyncio.run()`. Many of the code examples in these docs assume this is the context and omit `asyncio.run()`.

```
# no asyncio.run() necessary inside of IPython / Jupyter
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 script. For example:

```
from ctc import rpc

await rpc.async_close_http_session()
```

5.17 Datatypes

`ctc` has functions for collecting and analyzing many different EVM datatypes

Datatype	Examples	Reference
ABIs	Examples	Reference
Addresses	Examples	Reference
Binary	Examples	Reference
Blocks	Examples	Reference
ERC20s	Examples	Reference
ETH	Examples	Reference
Events	Examples	Reference
Transactions	Examples	Reference

5.17.1 ABIs

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

decompile solidity-style function ABI's from contract bytecode

Return type

Sequence[*Mapping*[str, *Any*]]

async ctc.evm.**async_get_contract_abi**(*contract_address*, *, *network=None*, *provider=None*, *use_db=True*, *db_query=None*, *db_intake=None*, *block=None*, *proxy_implementation=None*, *verbose=True*)

retrieve abi of contract either from local database or block explorer

for addresses that change ABI's over time, use db_query=False to skip cache

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

get event ABI from local database or block explorer

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*)

get function ABI from local database or block explorer

ctc.binary.get_event_hash ctc.binary.get_event_indexed_names ctc.binary.get_event_indexed_types ctc.binary.get_event_signature ctc.binary.get_event_unindexed_names ctc.binary.get_event_unindexed_types ctc.binary.get_function_output_names ctc.binary.get_function_output_types ctc.binary.get_function_parameter_names ctc.binary.get_function_parameter_types ctc.binary.get_function_selector ctc.binary.get_function_signature

5.17.2 Contracts

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

get block number of when contract was created

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

5.17.3 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

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

[none]

.. autofunction:: ctc.binary.convert .. autofunction:: ctc.binary.decode_call_data .. autofunction:: ctc.binary.decode_function_output .. autofunction:: ctc.binary.decode_types .. autofunction:: ctc.binary.encode_call_data .. autofunction:: ctc.binary.encode_types .. autofunction:: ctc.binary.keccak .. autofunction:: ctc.binary.keccak_text

5.17.4 Blocks

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

get block from local database or from RPC node

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*, *use_db_assist=True*, *mode='>='*)

search for the block that corresponds to a given timestamp

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

get blocks from local database or from RPC node

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=True*, *mode='>='*)

search for blocks corresponding to list of timestamps

5.17.5 ERC20s

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

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

async ctc.evm.**async_get_erc20_balance**(*wallet*, *token*, ***, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

get ERC20 balance

async ctc.evm.**async_get_erc20_balances_of_addresses**(*wallets*, *token*, ***, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

get ERC20 balance of multiple addresses

async ctc.evm.**async_get_erc20_balance_by_block**(*wallet*, *token*, ***, *blocks*, *normalize=True*, *provider=None*, *empty_token=0*, ***rpc_kwargs*)

get historical ERC20 balance over multiple blocks

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

get decimals of an erc20

async ctc.evm.**async_get_erc20_balances_from_transfers**(*transfers*, ***, *block=None*, *dtype=None*, *normalize=False*)

compute ERC20 balance of each wallet using Transfer events

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

get name of an erc20

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

get symbol of an erc20

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

get total supply of ERC20

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

get historical total supply of ERC20 across multiple blocks

async ctc.evm.**async_get_erc20_transfers**(*token*, *, *start_block=None*, *end_block=None*, *start_time=None*, *end_time=None*, *include_timestamps=False*, *normalize=True*, *convert_from_str=True*, *verbose=False*, *provider=None*, ***event_kwargs*)

get transfer events of ERC20 token

async ctc.evm.**async_get_erc20s_balances**(*wallet*, *tokens*, *, *block=None*, *normalize=True*, *provider=None*, ***rpc_kwargs*)

get ERC20 balance of wallet for multiple tokens

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*)

get total supplies of ERC20s

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

normalize ERC20 quantites by adjusting radix by (10 ** decimals)

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)

5.17.6 ETH Balances

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

5.17.7 Events

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

async `ctc.evm.`**`async_get_events`**(*contract_address*, *, *start_block=None*, *end_block=None*, *start_time=None*, *end_time=None*, *include_timestamps=False*, *backend_order=None*, *keep_multiindex=True*, *verbose=True*, *provider=None*, ***query*)

get events matching given inputs

5.17.8 Transactions

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

get transaction

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

get transaction count of address

5.18 Specific Protocols

`ctc` has functions for collecting and analyzing data from many on-chain and off-chain sources

5.18.1 4byte

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

async ctc.protocols.fourbyte_utils.**async_build_event_signatures_dataset**(*signature_data=None*)

Return type
None

async ctc.protocols.fourbyte_utils.**async_build_function_signatures_dataset**(*signature_data=None*)

Return type
None

async ctc.protocols.fourbyte_utils.**async_query_event_signatures**(*hex_signature=None*, *, *id=None*, *bytes_signature=None*, *text_signature=None*, *use_local=True*, *use_remote=True*)

Return type
`Sequence[Entry]`

async ctc.protocols.fourbyte_utils.**async_query_function_signatures**(*hex_signature=None*, *, *id=None*, *bytes_signature=None*, *text_signature=None*, *use_local=True*, *use_remote=True*)

Return type
`Sequence[Entry]`

5.18.2 Aave V2

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

async ctc.protocols.aave_v2_utils.**async_get_deposits**(**, start_block=None, end_block=None, start_time=None, end_time=None, include_timestamps=False, provider=None*)

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

async ctc.protocols.aave_v2_utils.**async_get_interest_rates_by_block**(*token, blocks, *, reserve_data_by_block=None*)

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

async ctc.protocols.aave_v2_utils.**async_get_reserve_data_by_block**(*asset, blocks, *, provider=None*)

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

async ctc.protocols.aave_v2_utils.**async_get_withdrawals**(**, start_block=None, end_block=None, start_time=None, end_time=None, include_timestamps=False, provider=None*)

5.18.3 Balancer

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

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

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

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

async ctc.protocols.balancer_utils.**async_get_pool_swaps**(*pool_address=None, *, start_block=None, end_block=None, start_time=None, end_time=None, include_timestamps=False*) async ctc.protocols.balancer_utils.**async_get_pool_tokens**(*, *pool_address=None*, *pool_id=None*, *block=None*)

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'*)

5.18.4 Chainlink

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

async ctc.protocols.chainlink_utils.**async_print_feed_summary**(*feed*, *, *start_block=None*, *n_recent=None*, *verbose=False*)

5.18.5 Compound

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

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*)

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

5.18.6 Gnosis Safe

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

5.18.7 Curve

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

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

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

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

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

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

5.18.8 Gnosis Safe

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

5.18.9 ENS

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

Return type
`int`

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

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

Return type
`int`

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

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

Return type
dict[str, str]

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

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

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

5.18.10 Fei

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

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

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

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

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

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

5.18.11 Gnosis Safe

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

5.18.12 Defi Llama

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

5.18.13 Multicall

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

5.18.14 Rari

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

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*)

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

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*)

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

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

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

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

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

5.18.15 Uniswap V2

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

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

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

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

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

async ctc.protocols.uniswap_v2_utils.**async_get_pool_swaps**(*pool*, *, *start_block=None*, *end_block=None*, *start_time=None*, *end_time=None*, *include_timestamps=False*, *include_prices=False*, *include_volumes=False*, *label='index'*, *normalize=True*, *provider=None*, *verbose=False*)

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

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

5.18.16 Uniswap V3

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

Reference

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

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

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*)

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*)

5.18.17 Yearn

Examples

Note

These examples are crafted as a Jupyter notebook. You can download the original notebook file here.

Also note that inside Jupyter notebooks, `await` can be used freely outside of `asyncio.run()`.

5.18.18 On-chain Protocols

Protocol	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
Gnosis Safe	Examples	Reference	Source
Multicall	Examples	Reference	Source
Rari	Examples	Reference	Source
Uniswap V2	Examples	Reference	Source
Uniswap V3	Examples	Reference	Source
Yearn	Examples	Reference	Source

5.18.19 External Data Sources

Protocol	Examples	Reference	Source
4byte	Examples	Reference	Source
Coingecko	Examples	Reference	Source
Defi Llama	Examples	Reference	Source
Etherscan	Examples	Reference	Source

5.19 Similar Python Tools

5.19.1 web3.py

web3.py is a general purpose EVM library that is created 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.

5.19.2 ape

ape is another general purpose EVM library that aims to improve upon web3.py in a variety of areas. Ape features direct integrations with many tools for both the development and deployment of smart contracts. Ape has plugins for many popular languages and tools including vyper, solidity, foundry, and hardhat.

5.19.3 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.

INDEX

A

async_build_event_signatures_dataset() (*in module ctc.protocols.fourbyte_utils*), 41
async_build_function_signatures_dataset() (*in module ctc.protocols.fourbyte_utils*), 41
async_create_payload() (*in module ctc.protocols.fei_utils*), 46
async_decompile_function_abis() (*in module ctc.evm*), 36
async_get_all_pools() (*in module ctc.protocols.rari_utils*), 47
async_get_base_pools() (*in module ctc.protocols.curve_utils*), 44
async_get_block() (*in module ctc.evm*), 37
async_get_block_of_timestamp() (*in module ctc.evm*), 37
async_get_blocks() (*in module ctc.evm*), 37
async_get_blocks_of_timestamps() (*in module ctc.evm*), 38
async_get_borrow_apy() (*in module ctc.protocols.compound_utils*), 43
async_get_borrow_apy_by_block() (*in module ctc.protocols.compound_utils*), 43
async_get_contract_abi() (*in module ctc.evm*), 36
async_get_contract_creation_block() (*in module ctc.evm*), 36
async_get_ctoken_state() (*in module ctc.protocols.rari_utils*), 47
async_get_ctoken_state_by_block() (*in module ctc.protocols.rari_utils*), 47
async_get_deposits() (*in module ctc.protocols.aave_v2_utils*), 42
async_get_erc20_balance() (*in module ctc.evm*), 38
async_get_erc20_balance_by_block() (*in module ctc.evm*), 38
async_get_erc20_balances_from_transfers() (*in module ctc.evm*), 38
async_get_erc20_balances_of_addresses() (*in module ctc.evm*), 38
async_get_erc20_decimals() (*in module ctc.evm*), 38
async_get_erc20_name() (*in module ctc.evm*), 38
async_get_erc20_symbol() (*in module ctc.evm*), 38
async_get_erc20_total_supply() (*in module ctc.evm*), 39
async_get_erc20_total_supply_by_block() (*in module ctc.evm*), 39
async_get_erc20_transfers() (*in module ctc.evm*), 39
async_get_erc20s_balances() (*in module ctc.evm*), 39
async_get_erc20s_decimals() (*in module ctc.evm*), 39
async_get_erc20s_names() (*in module ctc.evm*), 39
async_get_erc20s_symbols() (*in module ctc.evm*), 39
async_get_erc20s_total_supplies() (*in module ctc.evm*), 39
async_get_event_abi() (*in module ctc.evm*), 36
async_get_events() (*in module ctc.evm*), 40
async_get_expiration() (*in module ctc.protocols.ens_utils*), 45
async_get_function_abi() (*in module ctc.evm*), 36
async_get_interest_rates() (*in module ctc.protocols.aave_v2_utils*), 42
async_get_interest_rates_by_block() (*in module ctc.protocols.aave_v2_utils*), 42
async_get_meta_pools() (*in module ctc.protocols.curve_utils*), 44
async_get_owner() (*in module ctc.protocols.ens_utils*), 45
async_get_pcv_stats() (*in module ctc.protocols.fei_utils*), 46
async_get_pcv_stats_by_block() (*in module ctc.protocols.fei_utils*), 46
async_get_plain_pools() (*in module ctc.protocols.curve_utils*), 44
async_get_pool_address() (*in module ctc.protocols.balancer_utils*), 42
async_get_pool_balances() (*in module ctc.protocols.balancer_utils*), 42
async_get_pool_burns() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_ctokens() (*in module*

ctc.protocols.rari_utils), 47
async_get_pool_decimals() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_fees() (*in module ctc.protocols.balancer_utils*), 42
async_get_pool_id() (*in module ctc.protocols.balancer_utils*), 42
async_get_pool_metadata() (*in module ctc.protocols.curve_utils*), 44
async_get_pool_metadata() (*in module ctc.protocols.uniswap_v3_utils*), 49
async_get_pool_mints() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_prices() (*in module ctc.protocols.rari_utils*), 47
async_get_pool_state() (*in module ctc.protocols.curve_utils*), 44
async_get_pool_state() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_state_by_block() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_swaps() (*in module ctc.protocols.balancer_utils*), 42
async_get_pool_swaps() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_swaps() (*in module ctc.protocols.uniswap_v3_utils*), 49
async_get_pool_symbols() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_tokens() (*in module ctc.protocols.balancer_utils*), 42
async_get_pool_tokens() (*in module ctc.protocols.uniswap_v2_utils*), 48
async_get_pool_tvl_and_tvb() (*in module ctc.protocols.rari_utils*), 47
async_get_pool_underlying_tokens() (*in module ctc.protocols.rari_utils*), 47
async_get_pool_weights() (*in module ctc.protocols.balancer_utils*), 43
async_get_pool_weights_by_block() (*in module ctc.protocols.balancer_utils*), 43
async_get_registration_block() (*in module ctc.protocols.ens_utils*), 45
async_get_registrations() (*in module ctc.protocols.ens_utils*), 45
async_get_reserve_data() (*in module ctc.protocols.aave_v2_utils*), 42
async_get_reserve_data_by_block() (*in module ctc.protocols.aave_v2_utils*), 42
async_get_supply_apy() (*in module ctc.protocols.compound_utils*), 43
async_get_supply_apy_by_block() (*in module ctc.protocols.compound_utils*), 43
async_get_text_records() (*in module ctc.protocols.ens_utils*), 45
async_get_token_multipool_stats() (*in module ctc.protocols.rari_utils*), 47
async_get_transaction() (*in module ctc.evm*), 40
async_get_transaction_count() (*in module ctc.evm*), 40
async_get_underlying_asset() (*in module ctc.protocols.aave_v2_utils*), 42
async_get_virtual_price() (*in module ctc.protocols.curve_utils*), 44
async_get_withdrawals() (*in module ctc.protocols.aave_v2_utils*), 42
async_normalize_erc20_quantities() (*in module ctc.evm*), 39
async_normalize_erc20_quantity() (*in module ctc.evm*), 39
async_print_all_pool_summary() (*in module ctc.protocols.rari_utils*), 47
async_print_feed_summary() (*in module ctc.protocols.chainlink_utils*), 43
async_print_fuse_token_summary() (*in module ctc.protocols.rari_utils*), 47
async_print_pcv_assets() (*in module ctc.protocols.fei_utils*), 46
async_print_pcv_deposits() (*in module ctc.protocols.fei_utils*), 46
async_query_event_signatures() (*in module ctc.protocols.fourbyte_utils*), 41
async_query_function_signatures() (*in module ctc.protocols.fourbyte_utils*), 41
async_quote_exact_input_single() (*in module ctc.protocols.uniswap_v3_utils*), 49
async_quote_exact_output_single() (*in module ctc.protocols.uniswap_v3_utils*), 49
async_record_exists() (*in module ctc.protocols.ens_utils*), 45
async_reverse_lookup() (*in module ctc.protocols.ens_utils*), 45
async_summarize_pool_state() (*in module ctc.protocols.balancer_utils*), 43

H

hash_name() (*in module ctc.protocols.ens_utils*), 45