py-evm Documentation

Release 0.2.0-alpha.42

Ethereum Foundation

Mar 22, 2019

General

Py-EVM is a new implementation of the Ethereum Virtual Machine (EVM) written in Python.

If none of this makes sense to you yet we recommend to checkout the Ethereum website as well as a higher level description of the Ethereum project.

CHAPTER 1

Goals

The main focus is to enrich the Ethereum ecosystem with a Python implementation that:

- Supports Ethereum 1.0 as well as 2.0 / Serenity
- Is well documented
- Is easy to understand
- Has clear APIs
- Runs fast and resource friendly
- Is highly flexible to support:
  - Public chains
  - Private chains
  - Consortium chains
  - Advanced research

CHAPTER 2

Further reading

Here are a couple more useful links to check out.

- Source Code on GitHub
- Public Gitter Chat
- *Get involved*

CHAPTER 3

Table of contents

3.1 Introduction

Py-EVM is a new implementation of the Ethereum Virtual Machine (EVM) written in Python.

If none of this makes sense to you yet we recommend to checkout the Ethereum website as well as a higher level description of the Ethereum project.

3.1.1 Goals

The main focus is to enrich the Ethereum ecosystem with a Python implementation that:

- Supports Ethereum 1.0 as well as 2.0 / Serenity
- Is well documented
- Is easy to understand
- Has clear APIs
- Runs fast and resource friendly
- Is highly flexible to support:
  - Public chains
  - Private chains
  - Consortium chains
  - Advanced research

3.1.2 Further reading

Here are a couple more useful links to check out.

- Source Code on GitHub

- Public Gitter Chat
- *Get involved*

3.2 Quickstart

3.2.1 Installation

This guide teaches how to use Py-EVM as a library. For contributors, please check out the *Contributing Guide* which explains how to set everything up for development.

Installing on Ubuntu

Py-EVM requires Python 3.6 as well as some tools to compile its dependencies. On Ubuntu, the `python3.6-dev` package contains everything we need. Run the following command to install it.

```
apt-get install python3.6-dev
```

Py-EVM is installed through the pip package manager, if pip isn't available on the system already, we need to install the `python3-pip` package through the following command.

```
apt-get install python3-pip
```

Note: **Optional:** Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

Finally, we can install the `py-evm` package via pip.

```
pip3 install -U trinity
```

Installing on macOS

First, install Python 3 with brew:

```
brew install python3
```

Note: **Optional:** Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

Then, install the `py-evm` package via pip:

```
pip3 install -U py-evm
```

Hint: *Build a first app* on top of Py-EVM in under 5 minutes

3.3 Release notes

3.3.1 Unreleased (latest source)

- #1732: Bugfix: squashed an occasional "mix hash mismatch" while syncing

3.3.2 0.2.0-alpha.42

Released 2019-02-28

- #1719: Implement and activate Petersburg fork (aka Constantinople fixed)
- #1718: Performance: faster account lookups in EVM
- #1670: Performance: lazily look up ancestor block hashes, and cache result, so looking up parent hash in EVM is faster than grand^100 parent

3.3.3 0.2.0-alpha.40

Released Jan 15, 2019

- #1717: Indefinitely postpone the pending Constantinople release
- #1715: Remove Eth2 Beacon code, moving to trinity project

3.4 Cookbook

The Cookbook is a collection of simple recipes that demonstrate good practices to accomplish common tasks. The examples are usually short answers to simple "How do I…" questions that go beyond simple API descriptions but also don't need a full guide to become clear.

3.4.1 Using the Chain object

A "single" blockchain is made by a series of different virtual machines for different spans of blocks. For example, the Ethereum mainnet had one virtual machine for blocks 0 till 1150000 (known as Frontier), and another VM for blocks 1150000 till 1920000 (known as Homestead).

The `Chain` object manages the series of fork rules, after you define the VM ranges. For example, to set up a chain that would track the mainnet Ethereum network until block 1920000, you could create this chain class:

```
>>> from eth import constants, Chain
>>> from eth.vm.forks.frontier import FrontierVM
>>> from eth.vm.forks.homestead import HomesteadVM
>>> from eth.chains.mainnet import HOMESTEAD_MAINNET_BLOCK

>>> chain_class = Chain.configure(
...     __name__='Test Chain',
...     vm_configuration=(
...         (constants.GENESIS_BLOCK_NUMBER, FrontierVM),
...         (HOMESTEAD_MAINNET_BLOCK, HomesteadVM),
...     ),
... )
```

Then to initialize, you can start it up with an in-memory database:

```
>>> from eth.db.atomic import AtomicDB
>>> from eth.chains.mainnet import MAINNET_GENESIS_HEADER

>>> # start a fresh in-memory db

>>> # initialize a fresh chain
>>> chain = chain_class.from_genesis_header(AtomicDB(), MAINNET_GENESIS_HEADER)
```

3.4.2 Creating a chain with custom state

While the previous recipe demos how to create a chain from an existing genesis header, we can also create chains simply by specifing various genesis parameter as well as an optional genesis state.

```
>>> from eth_keys import keys
>>> from eth import constants
>>> from eth.chains.mainnet import MainnetChain
>>> from eth.db.atomic import AtomicDB
>>> from eth_utils import to_wei, encode_hex



>>> # Giving funds to some address
>>> SOME_ADDRESS = b'\x85\x82\xa2\x89V\xb9%\x93M\x03\xdd\xb4Xu\xe1\x8e\x85\x93\x12\xc1
↪'
>>> GENESIS_STATE = {
...     SOME_ADDRESS: {
...         "balance": to_wei(10000, 'ether'),
...         "nonce": 0,
...         "code": b'',
...         "storage": {}
...     }
... }
```

(continues on next page)

(continued from previous page)

```
>>> GENESIS_PARAMS = {
...     'parent_hash': constants.GENESIS_PARENT_HASH,
...     'uncles_hash': constants.EMPTY_UNCLE_HASH,
...     'coinbase': constants.ZERO_ADDRESS,
...     'transaction_root': constants.BLANK_ROOT_HASH,
...     'receipt_root': constants.BLANK_ROOT_HASH,
...     'difficulty': constants.GENESIS_DIFFICULTY,
...     'block_number': constants.GENESIS_BLOCK_NUMBER,
...     'gas_limit': constants.GENESIS_GAS_LIMIT,
...     'extra_data': constants.GENESIS_EXTRA_DATA,
...     'nonce': constants.GENESIS_NONCE
... }

>>> chain = MainnetChain.from_genesis(AtomicDB(), GENESIS_PARAMS, GENESIS_STATE)
```

3.4.3 Getting the balance from an account

Considering our previous example, we can get the balance of our pre-funded account as follows.

```
>>> current_vm = chain.get_vm()
>>> account_db = current_vm.state.account_db
>>> account_db.get_balance(SOME_ADDRESS)
100000000000000000000
```

3.4.4 Building blocks incrementally

The default `Chain` is stateless and thus does not keep a tip block open that would allow us to incrementally build a block. However, we can import the `MiningChain` which does allow exactly that.

```
>>> from eth.chains.base import MiningChain
```

Please check out the *Understanding the mining process* guide for a full example that demonstrates how to use the `MiningChain`.

3.5 Guides

This section aims to provide hands-on guides to demonstrate how to use Py-EVM. If you are looking for detailed API descriptions check out the *API section*.

3.5.1 Building an app that uses Py-EVM

One of the primary use cases of the `Py-EVM` library is to enable developers to build applications that want to interact with the ethereum ecosystem.

In this guide we want to build a very simple script that uses the `Py-EVM` library to create a fresh blockchain with a pre-funded address to simply read the balance of that address through the regular `Py-EVM` APIs. Frankly, not the most exciting application in the world, but the principle of how we use the `Py-EVM` library stays the same for more exciting use cases.

Setting up the application

Let's get started by setting up a new application. Often, that process involves lots of repetitive boilerplate code, so instead of doing it all by hand, let's just clone the Ethereum Python Project Template which contains all the typical things that we want.

To clone this into a new directory `demo-app` run:

```
git clone https://github.com/carver/ethereum-python-project-template.git demo-app
```

Then, change into the directory

```
cd demo-app
```

Add the Py-EVM library as a dependency

To add `Py-EVM` as a dependency, open the `setup.py` file in the root directory of the application and change the `install_requires` section as follows.

```
install_requires=[
    "eth-utils>=1,<2",
    "py-evm==0.2.0a40",
],
```

Warning: Make sure to also change the `name` inside the `setup.py` file to something valid (e.g. `demo-app`) or otherwise, fetching dependencies will fail.

Next, we need to use the `pip` package manager to fetch and install the dependencies of our app.

Note: Optional: Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

To install the dependencies, run:

```
pip install -e .[dev]
```

Congrats! We're now ready to build our application!

Writing the application code

Next, we'll create a new directory `app` and create a file `main.py` inside. Paste in the following content.

Note: The code examples are often written in an interactive session syntax, which is indicated by lines beginning with >>> or .... This enables us to run automatic tests against the examples to ensure they keep working while the library is evolving. When we want to copy and paste example code to play with it, we need to remove these extra characters to get runnable valid Python code.

```
>>> from eth import constants
>>> from eth.chains.mainnet import MainnetChain
>>> from eth.db.atomic import AtomicDB

>>> from eth_utils import to_wei, encode_hex


>>> MOCK_ADDRESS = constants.ZERO_ADDRESS
>>> DEFAULT_INITIAL_BALANCE = to_wei(10000, 'ether')

>>> GENESIS_PARAMS = {
...     'parent_hash': constants.GENESIS_PARENT_HASH,
...     'uncles_hash': constants.EMPTY_UNCLE_HASH,
...     'coinbase': constants.ZERO_ADDRESS,
...     'transaction_root': constants.BLANK_ROOT_HASH,
...     'receipt_root': constants.BLANK_ROOT_HASH,
...     'difficulty': constants.GENESIS_DIFFICULTY,
...     'block_number': constants.GENESIS_BLOCK_NUMBER,
...     'gas_limit': constants.GENESIS_GAS_LIMIT,
...     'extra_data': constants.GENESIS_EXTRA_DATA,
...     'nonce': constants.GENESIS_NONCE
... }

>>> GENESIS_STATE = {
...     MOCK_ADDRESS: {
...         "balance": DEFAULT_INITIAL_BALANCE,
...         "nonce": 0,
...         "code": b'',
...         "storage": {}
...     }
... }

>>> chain = MainnetChain.from_genesis(AtomicDB(), GENESIS_PARAMS, GENESIS_STATE)

>>> mock_address_balance = chain.get_vm().state.account_db.get_balance(MOCK_ADDRESS)

>>> print("The balance of address {} is {} wei".format(
...     encode_hex(MOCK_ADDRESS),
...     mock_address_balance)
... )
The balance of address 0x0000000000000000000000000000000000000000 is␣
↪10000000000000000000 wei
```

Runing the script

Let's run the script by invoking the following command.

```
python app/main.py
```

We should see the following output.

```
The balance of address 0x0000000000000000000000000000000000000000 is␣
↪100000000000000000000 wei
```

3.5.2 Architecture

The primary use case for Py-EVM is supporting the public Ethereum blockchain.

However, it is architected with a strong focus on configurability and extensibility. Use of Py-EVM for alternate use cases such as private chains, consortium chains, or even chains with fundamentally different VM semantics should be possible without any changes to the core library.

The following abstractions are used to represent the full consensus rules for a Py-EVM based blockchain.

- Chain: High level API for interacting with the blockchain.
- VM: High level API for a single fork within a Chain
- VMState: The current state of the VM, transaction execution logic and the state transition function.
- Message: Representation of the portion of the transaction which is relevant to VM execution.
- Computation: The computational state and result of VM execution.
- Opcode: The logic for a single opcode.

The Chain

The term **Chain** is used to encapsulate:

- The state transition function (e.g. VM opcodes and execution logic)
- Protocol rules (e.g. block rewards, header rewards, difficulty calculations, transaction execution)
- The chain data (e.g. **Headers**, **Blocks**, **Transactions** and **Receipts**)
- The state data (e.g. **balance**, **nonce**, **code** and **storage**)
- The chain state (e.g. tracking the chain head, canonical blocks)

Note: While a chain is used to *wrap* these concepts, many of them are actually defined at lower layers such as the underlying **Virtual Machines**.

The `Chain` object itself is largely an interface and orchestration layer. Most of the `Chain` APIs merely serving as a passthrough to the appropriate `VM`.

A chain has one or more underlying **Virtual Machines** or VMs. The chain contains a mapping which defines which VM should be active for which blocks.

The chain for the public mainnet Ethereum blockchain would have a separate VM defined for each fork ruleset (e.g. **Frontier**, **Homestead**, **Tangerine Whistle**, **Spurious Dragon**, **Byzantium**).

The VM

The term **VM** is used to encapsulate:

- The state transition function for a single fork ruleset.
- Orchestration logic for transaction execution.
- Block construction and validation.
- Chain data storage and retrieval APIs

The `VM` object loosely mirrors many of the Chain APIs for retrieval of chain state such as blocks, headers, transactions and receipts. It is also responsible for block level protocol logic such as block creation and validation.

The VMState

The term **VMState** is used to encapsulate:

- Execution context for the VM (e.g. `coinbase` or `gas_limit`)
- The state root defining the current VM state.
- Some block validation

The Message

The term **Message** comes from the yellow paper. It encapsulates the information from the transaction needed to initiate the outermost layer of VM execution.

- Parameters like `sender`, `value`, `to`

The message can be thought of as the VM's internal representation of a transaction.

The Computation

The term **Computation** is used to encapsulate:

- The computational state during VM execution (e.g. memory, stack, gas metering)
- The computational results of VM execution (e.g. return data, gas consumption and refunds, execution errors)

This abstraction is the interface through which opcode logic is implemented.

The Opcode

The term **Opcode** is used to encapsulate:

- A single instruction within the VM such as the `ADD` or `MUL` opcodes.

Opcodes are implemented as TODO

3.5.3 Understanding the mining process

From the *Cookbook* we can already learn how to use the `Chain` class to create a single blockchain as a combination of different virtual machines for different spans of blocks.

In this guide we want to build up on that knowledge and look into the actual mining process.

Note: Mining is an overloaded term and in fact the names of the mentioned APIs are subject to change.

Mining

The term *mining* can refer to different things depending on our point of view. Most of the time when we read about *mining*, we talk about the process where several parties are *competing* to be the first to create a new valid block and pass it on to the network.

In this guide, when we talk about the `mine_block()` API, we are only referring to the part that creates, validates and sets a block as the new canonical head of the chain but not necessarily as part of the mentioned competition to be the first. In fact, the `mine_block()` API is internally also called when we import existing blocks that others created.

Mining an empty block

Usually when we think about creating blocks we naturally think about adding transactions to the block first because, after all, one primary use case for the Ethereum blockchain is to process *transactions* which are wrapped in blocks.

For the sake of simplicity though, we'll mine an empty block as a first example (meaning the block will not contain any transactions)

As a refresher, he's how we create a chain as demonstrated in the *Using the chain object recipe* from the cookbook.

```
from eth.db.atomic import AtomicDB
from eth.chains.mainnet import MAINNET_GENESIS_HEADER

# increase the gas limit
genesis_header = MAINNET_GENESIS_HEADER.copy(gas_limit=3141592)

# initialize a fresh chain
chain = chain_class.from_genesis_header(AtomicDB(), genesis_header)
```

Since we decided to not add any transactions to our block let's just call `mine_block()` and see what happens.

```
# initialize a fresh chain
chain = chain_class.from_genesis_header(AtomicDB(), genesis_header)

chain.mine_block()
```

Aw, snap! We're running into an exception at `check_pow()`. Apparently we are trying to add a block to the chain that doesn't qualify the Proof-of-Work (PoW) rules. The error tells us precisely that the `mix_hash` of our block does not match the expected value.

```
Traceback (most recent call last):
  File "scripts/benchmark/run.py", line 111, in <module>
    run()
  File "scripts/benchmark/run.py", line 52, in run
    block = chain.mine_block()  #**pow_args
```

(continues on next page)

(continued from previous page)

```
  File "/py-evm/eth/chains/base.py", line 545, in mine_block
    self.validate_block(mined_block)
  File "/py-evm/eth/chains/base.py", line 585, in validate_block
    self.validate_seal(block.header)
  File "/py-evm/eth/chains/base.py", line 622, in validate_seal
    header.mix_hash, header.nonce, header.difficulty)
  File "/py-evm/eth/consensus/pow.py", line 70, in check_pow
    encode_hex(mining_output[b'mix digest']), encode_hex(mix_hash)))

eth.exceptions.ValidationError: mix hash mismatch;
0x7a76bbf0c8d0e683fafa2d7cab27f601e19f35e7ecad7e1abb064b6f8f08fe21 !=
0x0000000000000000000000000000000000000000000000000000000000000000
```

Let's lookup how `check_pow()` is implemented.

```
def check_pow(block_number: int,
              mining_hash: Hash32,
              mix_hash: Hash32,
              nonce: bytes,
              difficulty: int) -> None:
    validate_length(mix_hash, 32, title="Mix Hash")
    validate_length(mining_hash, 32, title="Mining Hash")
    validate_length(nonce, 8, title="POW Nonce")
    cache = get_cache(block_number)
    mining_output = hashimoto_light(
        block_number, cache, mining_hash, big_endian_to_int(nonce))
    if mining_output[b'mix digest'] != mix_hash:
        raise ValidationError(
            "mix hash mismatch; expected: {} != actual: {}. "
            "Mix hash calculated from block #{}, mine hash {}, nonce {}, difficulty {}
↪, "
            "cache hash {}".format(
                encode_hex(mining_output[b'mix digest']),
                encode_hex(mix_hash),
                block_number,
                encode_hex(mining_hash),
                encode_hex(nonce),
                difficulty,
                encode_hex(keccak(cache)),
            )
        )
    result = big_endian_to_int(mining_output[b'result'])
    validate_lte(result, 2**256 // difficulty, title="POW Difficulty")
```

Just by looking at the signature of that function we can see that validating the PoW is based on the following parameters:

- `block_number` - the number of the given block
- `difficulty` - the difficulty of the PoW algorithm
- `mining_hash` - hash of the mining header
- `mix_hash` - together with the `nonce` forms the actual proof
- `nonce` - together with the `mix_hash` forms the actual proof

The PoW algorithm checks that all these parameters match correctly, ensuring that only valid blocks can be added to the chain.

In order to produce a valid block, we have to set the correct `mix_hash` and `nonce` in the header. We can pass these as key-value pairs when we call `mine_block()` as seen below.

```
chain.mine_block(nonce=valid_nonce, mix_hash=valid_mix_hash)
```

This call will work just fine assuming we are passing the correct `nonce` and `mix_hash` that corresponds to the block getting mined.

Retrieving a valid nonce and mix hash

Now that we know we can call `mine_block()` with the correct parameters to successfully add a block to our chain, let's briefly go over an example that demonstrates how we can retrieve a matching `nonce` and `mix_hash`.

Note: Py-EVM currently doesn't offer a stable API for actual PoW mining. The following code is for demonstration purpose only.

Mining on the main ethereum chain is a competition done simultanously by many miners, hence the *mining difficulty* is pretty high which means it will take a very long time to find the right `nonce` and `mix_hash` on commodity hardware. In order for us to have something that we can tinker with on a regular laptop, we'll construct a test chain with the `difficulty` set to `1`.

Let's start off by defining the `GENESIS_PARAMS`.

```
from eth import constants

GENESIS_PARAMS = {
      'parent_hash': constants.GENESIS_PARENT_HASH,
      'uncles_hash': constants.EMPTY_UNCLE_HASH,
      'coinbase': constants.ZERO_ADDRESS,
      'transaction_root': constants.BLANK_ROOT_HASH,
      'receipt_root': constants.BLANK_ROOT_HASH,
      'difficulty': 1,
      'block_number': constants.GENESIS_BLOCK_NUMBER,
      'gas_limit': 3141592,
      'timestamp': 1514764800,
      'extra_data': constants.GENESIS_EXTRA_DATA,
      'nonce': constants.GENESIS_NONCE
  }
```

Next, we'll create the chain itself using the defined `GENESIS_PARAMS` and the latest `ByzantiumVM`.

```
from eth import MiningChain
from eth.vm.forks.byzantium import ByzantiumVM
from eth.db.backends.memory import AtomicDB


klass = MiningChain.configure(
    __name__='TestChain',
    vm_configuration=(
        (constants.GENESIS_BLOCK_NUMBER, ByzantiumVM),
    ))
chain = klass.from_genesis(AtomicDB(), GENESIS_PARAMS)
```

Now that we have the building blocks available, let's put it all together and mine a proper block!

```
from eth.consensus.pow import mine_pow_nonce


# We have to finalize the block first in order to be able read the
# attributes that are important for the PoW algorithm
block = chain.get_vm().finalize_block(chain.get_block())

# based on mining_hash, block number and difficulty we can perform
# the actual Proof of Work (PoW) mechanism to mine the correct
# nonce and mix_hash for this block
nonce, mix_hash = mine_pow_nonce(
    block.number,
    block.header.mining_hash,
    block.header.difficulty)

block = chain.mine_block(mix_hash=mix_hash, nonce=nonce)
```

```
>>> print(block)
Block #1
```

Let's take a moment to fully understand what this code does.

1. We call *`finalize_block()`* on the underlying VM in order to retrieve the information that we need to calculate the `nonce` and the `mix_hash`.

2. We then call `mine_pow_nonce()` to retrieve the proper `nonce` and `mix_hash` that we need to mine the block and satisfy the validation.

3. Finally we call `mine_block()` and pass along the `nonce` and the `mix_hash`

Note: The code above will essentially perform `finalize_block` twice. Keep in mind this code is for demonstration purpose only and that Py-EVM will provide a pluggable system in the future to allow PoW mining among other things.

Mining a block with transactions

Now that we've learned the basics of how the mining process works, let's revisited our example and add a transaction before we mine another block. There are a couple of concepts we need to dive into in order to accomplish that goal.

Every transaction goes from a sender `Address` to a receiver `Address`. Each transaction takes some computational power to get processed that is measured in a unit called `gas`.

In practice, we have to pay the miners to put our transaction in a block. However, there is no *technical* reason why we have to pay for the computing power, but only an economical, i.e. in reality we'll usually have trouble finding a miner who's willing to include a transaction that doesn't pay for its computational costs.

In this example, however, **we are the miner** which means we are free to include any transactions we like. In the spirit of this guide, let's start simple and create a transaction that sends zero ether from one address to another address. Keep in mind that even if the value being transferred is zero, there's still a computational cost for the processing but since we are the miner, we'll mine it anyway even if no one is willing to pay for it!

Let's first setup the sender and receiver.

```
from eth_keys import keys
from eth_utils import decode_hex
```

(continues on next page)

(continued from previous page)

```
from eth_typing import Address

SENDER_PRIVATE_KEY = keys.PrivateKey(
  decode_hex('0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8')
)

SENDER = Address(SENDER_PRIVATE_KEY.public_key.to_canonical_address())

RECEIVER = Address(b'\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\x02')
```

One thing that strikes out here is that we only need the plain address for the receiver whereas for the sender we are obtaining an address derived from the `SENDER_PRIVATE_KEY`. That's because we obviously can not send transactions from an address that we don't have the private key to sign it for.

With sender and receiver prepared, let's create the actual transaction.

```
vm = chain.get_vm()
nonce = vm.state.account_db.get_nonce(SENDER)

tx = vm.create_unsigned_transaction(
    nonce=nonce,
    gas_price=0,
    gas=100000,
    to=RECEIVER,
    value=0,
    data=b'',
)
```

Every transaction needs a `nonce` not to be confused with the `nonce` that we previously mined as part of the PoW algorithm. The *transaction nonce* serves as a counter to ensure all transactions from one address are processed in order. We retrieve the current `nonce` by calling `get_nonce(sender)()`.

Once we have the `nonce` we can call *`create_unsigned_transaction()`* and pass the `nonce` among the rest of the transaction attributes as key-value pairs.

- `nonce` - Number of transactions sent by the sender
- `gas_price` - Number of `Wei` to pay per unit of gas
- `gas` - Maximum amount of `gas` the transaction is allowed to consume before it gets rejected
- `to` - Address of transaction recipient
- `value` - Number of `Wei` to be transferred to the recipient

The last step we need to do before we can add the transaction to a block is to sign it with the private key which is as simple as calling *`as_signed_transaction()`* with the `SENDER_PRIVATE_KEY`.

```
signed_tx = tx.as_signed_transaction(SENDER_PRIVATE_KEY)
```

Finally, we can call `apply_transaction()` and pass along the `signed_tx`.

```
chain.apply_transaction(signed_tx)
```

What follows is the complete script that demonstrates how to mine a single block with one simple zero value transfer transaction.

```
>>> from eth_keys import keys
>>> from eth_utils import decode_hex
```

(continues on next page)

(continued from previous page)

```
>>> from eth_typing import Address
>>> from eth import constants
>>> from eth.chains.base import MiningChain
>>> from eth.consensus.pow import mine_pow_nonce
>>> from eth.vm.forks.byzantium import ByzantiumVM
>>> from eth.db.atomic import AtomicDB


>>> GENESIS_PARAMS = {
...     'parent_hash': constants.GENESIS_PARENT_HASH,
...     'uncles_hash': constants.EMPTY_UNCLE_HASH,
...     'coinbase': constants.ZERO_ADDRESS,
...     'transaction_root': constants.BLANK_ROOT_HASH,
...     'receipt_root': constants.BLANK_ROOT_HASH,
...     'difficulty': 1,
...     'block_number': constants.GENESIS_BLOCK_NUMBER,
...     'gas_limit': 3141592,
...     'timestamp': 1514764800,
...     'extra_data': constants.GENESIS_EXTRA_DATA,
...     'nonce': constants.GENESIS_NONCE
... }

>>> SENDER_PRIVATE_KEY = keys.PrivateKey(
...     decode_hex('0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
↪')
... )

>>> SENDER = Address(SENDER_PRIVATE_KEY.public_key.to_canonical_address())

>>> RECEIVER = Address(b'\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0')

>>> klass = MiningChain.configure(
...     __name__='TestChain',
...     vm_configuration=(
...         (constants.GENESIS_BLOCK_NUMBER, ByzantiumVM),
...     ))

>>> chain = klass.from_genesis(AtomicDB(), GENESIS_PARAMS)
>>> vm = chain.get_vm()

>>> nonce = vm.state.account_db.get_nonce(SENDER)

>>> tx = vm.create_unsigned_transaction(
...     nonce=nonce,
...     gas_price=0,
...     gas=100000,
...     to=RECEIVER,
...     value=0,
...     data=b'',
... )

>>> signed_tx = tx.as_signed_transaction(SENDER_PRIVATE_KEY)

>>> chain.apply_transaction(signed_tx)
(<ByzantiumBlock(#Block #1...)
>>> # We have to finalize the block first in order to be able read the
>>> # attributes that are important for the PoW algorithm
```

(continues on next page)

(continued from previous page)

```
>>> block = chain.get_vm().finalize_block(chain.get_block())

>>> # based on mining_hash, block number and difficulty we can perform
>>> # the actual Proof of Work (PoW) mechanism to mine the correct
>>> # nonce and mix_hash for this block
>>> nonce, mix_hash = mine_pow_nonce(
...     block.number,
...     block.header.mining_hash,
...     block.header.difficulty
... )

>>> chain.mine_block(mix_hash=mix_hash, nonce=nonce)
<ByzantiumBlock(#Block #1)>
```

3.5.4 Creating Opcodes

An opcode is just a function which takes a *BaseComputation* instance as it's sole argument. If an opcode function has a return value, this value will be discarded during normal VM execution.

Here are some simple examples.

```
def noop(computation):
    """
    An opcode which does nothing (not even consume gas)
    """
    pass

def burn_5_gas(computation):
    """
    An opcode which simply burns 5 gas
    """
    computation.consume_gas(5, reason='why not?')
```

The `as_opcode()` helper

While these examples are demonstrative of *simple* logic, opcodes will traditionally have an intrinsic gas cost associated with them. Py-EVM offers an abstraction which allows for decoupling of gas consumption from opcode logic which can be convenient for cases where an opcode's gas cost changes between different VM rules but its logic remains constant.

`eth.vm.opcode.`**`as_opcode`**(*logic_fn*, *mnemonic*, *gas_cost*)

- The `logic_fn` argument should be a callable conforming to the opcode API, taking a *~eth.vm.computation.Computation* instance as its sole argument.
- The `mnemonic` is a string such as `'ADD'` or `'MUL'`.
- The `gas_cost` is the gas cost to execute this opcode.

The return value is a function which will consume the `gas_cost` prior to execution of the `logic_fn`.

Usage of the *as_opcode()* helper:

```
def custom_op(computation):
    ... # opcode logic here
```

(continues on next page)

(continued from previous page)

```
class ExampleComputation(BaseComputation):
    opcodes = {
        b'\x01': as_opcode(custom_op, 'CUSTOM_OP', 10),
    }
```

Opcodes as classes

Sometimes it may be helpful to share common logic between similar opcodes, or the same opcode across multiple fork rules. In these cases, implementing opcodes as classes *may* be the right choice. This is as simple as implementing a `__call__` method on your class which conforms to the opcode API, taking a single `Computation` instance as the sole argument.

```
class MyOpcode:
    def initial_logic(self, computation):
        ...

    def main_logic(self, computation):
        ...

    def cleanup_logic(self, computation):
        ...

    def __call__(self, computation):
        self.initial_logic(computation)
        self.main_logic(computation)
        self.cleanup_logic(computation)
```

With this pattern, the overall structure, as well as much of the logic can be re-used while still allowing a mechanism for overriding individual sections of the opcode logic.

3.6 API

This section aims to provide a detailed description of all APIs. If you are looking for something more hands-on or higher-level check out the existing *guides*.

Warning: We expect each alpha release to have breaking changes to the API.

3.6.1 Chain

BaseChain

class eth.chains.base.**BaseChain**

The base class for all Chain objects

classmethod get_vm_class(*header: eth.rlp.headers.BlockHeader*) → Type[BaseVM]

Returns the VM instance for the given block number.

classmethod get_vm_class_for_block_number(*block_number: New-Type.<locals>.new_type*) → Type[BaseVM]

Returns the VM class for the given block number.

classmethod validate_chain(*root: eth.rlp.headers.BlockHeader, descendants: Tuple[eth.rlp.headers.BlockHeader, ...], seal_check_random_sample_rate: int = 1*) → None

Validate that all of the descendents are valid, given that the root header is valid.

By default, check the seal validity (Proof-of-Work on Ethereum 1.x mainnet) of all headers. This can be expensive. Instead, check a random sample of seals using seal_check_random_sample_rate.

Chain

class eth.chains.base.**Chain**(*base_db: eth.db.backends.base.BaseAtomicDB*)

A Chain is a combination of one or more VM classes. Each VM is associated with a range of blocks. The Chain class acts as a wrapper around these other VM classes, delegating operations to the appropriate VM depending on the current block number.

build_block_with_transactions(*transactions: Tuple[eth.rlp.transactions.BaseTransaction, ...], parent_header: eth.rlp.headers.BlockHeader = None*) → Tuple[eth.rlp.blocks.BaseBlock, Tuple[eth.rlp.receipts.Receipt, ...], Tuple[eth.vm.computation.BaseComputation, ...]]

Generate a block with the provided transactions. This does *not* import that block into your chain. If you want this new block in your chain, run *import_block()* with the result block from this method.

Parameters

- **transactions** – an iterable of transactions to insert to the block
- **parent_header** – parent of the new block – or canonical head if None

Returns (new block, receipts, computations)

chaindb_class

alias of *eth.db.chain.ChainDB*

create_header_from_parent(*parent_header: eth.rlp.headers.BlockHeader*, ***header_params*) → eth.rlp.headers.BlockHeader

Passthrough helper to the VM class of the block descending from the given header.

create_transaction(**args*, ***kwargs*) → eth.rlp.transactions.BaseTransaction

Passthrough helper to the current VM class.

create_unsigned_transaction(**, nonce: int, gas_price: int, gas: int, to: New-Type.<locals>.new_type, value: int, data: bytes*) → eth.rlp.transactions.BaseUnsignedTransaction

Passthrough helper to the current VM class.

ensure_header(*header: eth.rlp.headers.BlockHeader = None*) → eth.rlp.headers.BlockHeader

Return header if it is not None, otherwise return the header of the canonical head.

estimate_gas(*transaction: Union[BaseTransaction, SpoofTransaction], at_header: eth.rlp.headers.BlockHeader = None*) → int

Returns an estimation of the amount of gas the given transaction will use if executed on top of the block specified by the given header.

classmethod from_genesis(*base_db: eth.db.backends.base.BaseAtomicDB, genesis_params: Dict[str, Union[int, None, bytes, NewType.<locals>.new_type, NewType.<locals>.new_type]], genesis_state: Dict[NewType.<locals>.new_type, eth.typing.AccountDetails] = None*) → eth.chains.base.BaseChain
Initializes the Chain from a genesis state.

classmethod from_genesis_header(*base_db: eth.db.backends.base.BaseAtomicDB, genesis_header: eth.rlp.headers.BlockHeader*) → eth.chains.base.BaseChain
Initializes the chain from the genesis header.

get_ancestors(*limit: int*, *header: eth.rlp.headers.BlockHeader*) → Tuple[eth.rlp.blocks.BaseBlock, ...]
Return *limit* number of ancestor blocks from the current canonical head.

get_block() → eth.rlp.blocks.BaseBlock
Returns the current TIP block.

get_block_by_hash(*block_hash: NewType.<locals>.new_type*) → eth.rlp.blocks.BaseBlock
Returns the requested block as specified by block hash.

get_block_by_header(*block_header: eth.rlp.headers.BlockHeader*) → eth.rlp.blocks.BaseBlock
Returns the requested block as specified by the block header.

get_block_header_by_hash(*block_hash: NewType.<locals>.new_type*) → eth.rlp.headers.BlockHeader
Returns the requested block header as specified by block hash.

Raises BlockNotFound if there's no block header with the given hash in the db.

get_canonical_block_by_number(*block_number: NewType.<locals>.new_type*) → eth.rlp.blocks.BaseBlock
Returns the block with the given number in the canonical chain.

Raises BlockNotFound if there's no block with the given number in the canonical chain.

get_canonical_block_hash(*block_number: NewType.<locals>.new_type*) → NewType.<locals>.new_type
Returns the block hash with the given number in the canonical chain.

Raises BlockNotFound if there's no block with the given number in the canonical chain.

get_canonical_head() → eth.rlp.headers.BlockHeader
Returns the block header at the canonical chain head.

Raises CanonicalHeadNotFound if there's no head defined for the canonical chain.

get_canonical_transaction(*transaction_hash: NewType.<locals>.new_type*) → eth.rlp.transactions.BaseTransaction
Returns the requested transaction as specified by the transaction hash from the canonical chain.

Raises TransactionNotFound if no transaction with the specified hash is found in the main chain.

get_score(*block_hash: NewType.<locals>.new_type*) → int
Returns the difficulty score of the block with the given hash.

Raises HeaderNotFound if there is no matching black hash.

get_transaction_result(*transaction: Union[BaseTransaction, SpoofTransaction], at_header: eth.rlp.headers.BlockHeader*) → bytes
Return the result of running the given transaction. This is referred to as a *call()* in web3.

get_vm(*at_header: eth.rlp.headers.BlockHeader = None*) → BaseVM
Returns the VM instance for the given block number.

import_block(*block: eth.rlp.blocks.BaseBlock*, *perform_validation: bool = True*) → Tuple[eth.rlp.blocks.BaseBlock, Tuple[eth.rlp.blocks.BaseBlock, ...], Tuple[eth.rlp.blocks.BaseBlock, ...]]
Imports a complete block and returns a 3-tuple

- the imported block
- a tuple of blocks which are now part of the canonical chain.
- a tuple of blocks which were canonical and now are no longer canonical.

validate_block(*block: eth.rlp.blocks.BaseBlock*) → None
Performs validation on a block that is either being mined or imported.

Since block validation (specifically the uncle validation) must have access to the ancestor blocks, this validation must occur at the Chain level.

Cannot be used to validate genesis block.

validate_gaslimit(*header: eth.rlp.headers.BlockHeader*) → None
Validate the gas limit on the given header.

validate_seal(*header: eth.rlp.headers.BlockHeader*) → None
Validate the seal on the given header.

validate_uncles(*block: eth.rlp.blocks.BaseBlock*) → None
Validate the uncles for the given block.

3.6.2 DataBase

Backends

BaseDB

class eth.db.backends.base.**BaseDB**
This is an abstract key/value lookup with all `bytes` values, with some convenience methods for databases. As much as possible, you can use a DB as if it were a `dict`.

Notable exceptions are that you cannot iterate through all values or get the length. (Unless a subclass explicitly enables it).

All subclasses must implement these methods: __init__, __getitem__, __setitem__, __delitem__

Subclasses may optionally implement an _exists method that is type-checked for key and value.

LevelDB

class eth.db.backends.level.**LevelDB**(*db_path: pathlib.Path = None*, *max_open_files: int = None*)

MemoryDB

class eth.db.backends.memory.**MemoryDB**(*kv_store: Dict[bytes, bytes] = None*)

Account

BaseAccountDB

class eth.db.account.**BaseAccountDB**

make_state_root() → NewType.<locals>.new_type
Generate the state root with all the current changes in AccountDB

Returns the new state root

persist() → None
Send changes to underlying database, including the trie state so that it will forever be possible to read the trie from this checkpoint.

AccountDB

class eth.db.account.**AccountDB**(*db: eth.db.backends.base.BaseDB*, *state_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cxb4!'*)

make_state_root() → NewType.<locals>.new_type
Generate the state root with all the current changes in AccountDB

Returns the new state root

persist() → None
Send changes to underlying database, including the trie state so that it will forever be possible to read the trie from this checkpoint.

Journal

JournalDB

class eth.db.journal.**JournalDB**(*wrapped_db: eth.db.backends.base.BaseDB*)
A wrapper around the basic DB objects that keeps a journal of all changes. Each time a recording is started, the underlying journal creates a new changeset and assigns an id to it. The journal then keeps track of all changes that go into this changeset.

Discarding a changeset simply throws it away inculding all subsequent changesets that may have followed. Commiting a changeset merges the given changeset and all subsequent changesets into the previous changeset giving precidence to later changesets in case of conflicting keys.

Nothing is written to the underlying db until *persist()* is called.

The added memory footprint for a JournalDB is one key/value stored per database key which is changed. Subsequent changes to the same key within the same changeset will not increase the journal size since we only need to track latest value for any given key within any given changeset.

commit(*changeset_id: uuid.UUID*) → None
Commits a given changeset. This merges the given changeset and all subsequent changesets into the previous changeset giving precidence to later changesets in case of any conflicting keys.

If this is the base changeset then all changes will be written to the underlying database and the Journal starts a new recording.

discard(*changeset_id: uuid.UUID*) → None
Throws away all journaled data starting at the given changeset

persist() → None
Persist all changes in underlying db

record() → uuid.UUID
Starts a new recording and returns an id for the associated changeset

reset() → None
Reset the entire journal.

Chain

BaseChainDB

class eth.db.chain.**BaseChainDB**(*db: eth.db.backends.base.BaseAtomicDB*)

ChainDB

class eth.db.chain.**ChainDB**(*db: eth.db.backends.base.BaseAtomicDB*)

add_receipt(*block_header: eth.rlp.headers.BlockHeader*, *index_key: int*, *receipt: eth.rlp.receipts.Receipt*) → NewType.<locals>.new_type
Adds the given receipt to the provided block header.

Returns the updated *receipts_root* for updated block header.

add_transaction(*block_header: eth.rlp.headers.BlockHeader*, *index_key: int*, *transaction: BaseTransaction*) → NewType.<locals>.new_type
Adds the given transaction to the provided block header.

Returns the updated *transactions_root* for updated block header.

exists(*key: bytes*) → bool
Returns True if the given key exists in the database.

get(*key: bytes*) → bytes
Return the value for the given key or a KeyError if it doesn't exist in the database.

get_block_transaction_hashes(*block_header: eth.rlp.headers.BlockHeader*) → Iterable[NewType.<locals>.new_type]
Returns an iterable of the transaction hashes from the block specified by the given block header.

get_block_transactions(*header: eth.rlp.headers.BlockHeader, transaction_class: Type[BaseTransaction]*) → Iterable[BaseTransaction]
Returns an iterable of transactions for the block speficied by the given block header.

get_block_uncles(*uncles_hash: NewType.<locals>.new_type*) → List[eth.rlp.headers.BlockHeader]
Returns an iterable of uncle headers specified by the given uncles_hash

get_receipt_by_index(*block_number: NewType.<locals>.new_type*, *receipt_index: int*) → eth.rlp.receipts.Receipt
Returns the Receipt of the transaction at specified index for the block header obtained by the specified block number

get_receipts(*header: eth.rlp.headers.BlockHeader, receipt_class: Type[eth.rlp.receipts.Receipt]*) → Iterable[eth.rlp.receipts.Receipt]
Returns an iterable of receipts for the block specified by the given block header.

get_transaction_by_index(*block_number: NewType.<locals>.new_type, transaction_index: int, transaction_class: Type[BaseTransaction]*) → BaseTransaction
Returns the transaction at the specified *transaction_index* from the block specified by *block_number* from the canonical chain.

Raises TransactionNotFound if no block

get_transaction_index(*transaction_hash: NewType.<locals>.new_type*) → Tuple[NewType.<locals>.new_type, int]
Returns a 2-tuple of (block_number, transaction_index) indicating which block the given transaction can be found in and at what index in the block transactions.

Raises TransactionNotFound if the transaction_hash is not found in the canonical chain.

persist_block(*block: BaseBlock*) → Tuple[Tuple[NewType.<locals>.new_type, ...], Tuple[NewType.<locals>.new_type, ...]]
Persist the given block's header and uncles.

Assumes all block transactions have been persisted already.

persist_trie_data_dict(*trie_data_dict: Dict[NewType.<locals>.new_type, bytes]*) → None
Store raw trie data to db from a dict

persist_uncles(*uncles: Tuple[eth.rlp.headers.BlockHeader]*) → NewType.<locals>.new_type
Persists the list of uncles to the database.

Returns the uncles hash.

3.6.3 Exceptions

exception eth.exceptions.**BlockNotFound**
Raised when the block with the given number/hash does not exist.

exception eth.exceptions.**CanonicalHeadNotFound**
Raised when the chain has no canonical head.

exception eth.exceptions.**ContractCreationCollision**
Raised when there was an address collision during contract creation.

exception eth.exceptions.**FullStack**
Raised when the stack is full.

exception eth.exceptions.**Halt**
Raised when an opcode function halts vm execution.

exception eth.exceptions.**HeaderNotFound**
Raised when a header with the given number/hash does not exist.

exception eth.exceptions.**IncorrectContractCreationAddress**
Raised when the address provided by transaction does not match the calculated contract creation address.

exception eth.exceptions.**InsufficientFunds**
Raised when an account has insufficient funds to transfer the requested value.

exception eth.exceptions.**InsufficientStack**
Raised when the stack is empty.

exception eth.exceptions.**InvalidInstruction**
Raised when an opcode is invalid.

exception eth.exceptions.**InvalidJumpDestination**
Raised when the jump destination for a JUMPDEST operation is invalid.

exception eth.exceptions.**OutOfBoundsRead**
Raised when an attempt was made to read data beyond the boundaries of the buffer (such as with RETURNDATACOPY)

exception eth.exceptions.**OutOfGas**
Raised when a VM execution has run out of gas.

exception eth.exceptions.**ParentNotFound**
Raised when the parent of a given block does not exist.

exception eth.exceptions.**PyEVMError**
Base class for all py-evm errors.

exception eth.exceptions.**ReceiptNotFound**
Raised when the Receipt with the given receipt index does not exist.

exception eth.exceptions.**Revert**
Raised when the REVERT opcode occured

exception eth.exceptions.**StackDepthLimit**
Raised when the call stack has exceeded it's maximum allowed depth.

exception eth.exceptions.**StateRootNotFound**
Raised when the requested state root is not present in our DB.

exception eth.exceptions.**TransactionNotFound**
Raised when the transaction with the given hash or block index does not exist.

exception eth.exceptions.**VMError**
Base class for errors raised during VM execution.

exception eth.exceptions.**VMNotFound**
Raised when no VM is available for the provided block number.

exception eth.exceptions.**WriteProtection**
Raised when an attempt to modify the state database is made while operating inside of a STATICCALL context.

3.6.4 RLP

Accounts

Account

class eth.rlp.accounts.**Account**(*nonce: int = 0*, *balance: int = 0*, *storage_root: bytes = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cxb4!'*, *code_hash: bytes = b"xc5xd2Fx01x86xf7#<x92~}xb2xdcxc7x03xc0xe5x00xb6Sxcax82';{*, ***kwargs*)
RLP object for accounts.

Blocks

BaseBlock

class eth.rlp.blocks.**BaseBlock**(**args*, ***kwargs*)

classmethod from_header(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*) → eth.rlp.blocks.BaseBlock

Returns the block denoted by the given block header.

Headers

BlockHeader

class eth.rlp.headers.**BlockHeader**(*difficulty: int*, *block_number: int*, *gas_limit: int*, *timestamp: int = None*, *coinbase: NewType.<locals>.new_type = b'x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00'*, *parent_hash: NewType.<locals>.new_type = b'x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00*, *uncles_hash: NewType.<locals>.new_type = b'x1dxccMxe8xdexc7]zxabx85xb5gxb6xccxd4x1axd3x12Ex1bx94x8atx13xf0xa1Bxfa*, *state_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cx*, *transaction_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cx*, *receipt_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cx*, *bloom: int = 0*, *gas_used: int = 0*, *extra_data: bytes = b''*, *mix_hash: NewType.<locals>.new_type = b'x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00*, *nonce: bytes = b'x00x00x00x00x00x00x00B'*)

classmethod from_parent(*parent: eth.rlp.headers.BlockHeader*, *gas_limit: int*, *difficulty: int*, *timestamp: int*, *coinbase: NewType.<locals>.new_type = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'*, *nonce: bytes = None*, *extra_data: bytes = None*, *transaction_root: bytes = None*, *receipt_root: bytes = None*) → eth.rlp.headers.BlockHeader

Initialize a new block header with the *parent* header as the block's parent hash.

Logs

Log

class eth.rlp.logs.**Log**(*address: bytes, topics: List[int], data: bytes*)

Receipts

Receipt

class eth.rlp.receipts.**Receipt**(*state_root: bytes, gas_used: int, logs: Iterable[eth.rlp.logs.Log], bloom: int = None*)

Transactions

BaseTransactionMethods

class eth.rlp.transactions.**BaseTransactionMethods**

gas_used_by(*computation: eth.vm.computation.BaseComputation*) → int
Return the gas used by the given computation. In Frontier, for example, this is sum of the intrinsic cost and the gas used during computation.

get_intrinsic_gas() → int
Compute the baseline gas cost for this transaction. This is the amount of gas needed to send this transaction (but that is not actually used for computation).

intrinsic_gas
Convenience property for the return value of *get_intrinsic_gas*

validate() → None
Hook called during instantiation to ensure that all transaction parameters pass validation rules.

BaseTransactionFields

class eth.rlp.transactions.**BaseTransactionFields**(**args*, ***kwargs*)

BaseTransaction

class eth.rlp.transactions.**BaseTransaction**(**args*, ***kwargs*)

check_signature_validity() → None
Checks signature validity, raising a ValidationError if the signature is invalid.

classmethod create_unsigned_transaction(**, nonce: int, gas_price: int, gas: int, to: NewType.<locals>.new_type, value: int, data: bytes*) → eth.rlp.transactions.BaseUnsignedTransaction
Create an unsigned transaction.

get_message_for_signing() → bytes
Return the bytestring that should be signed in order to create a signed transactions

get_sender() → NewType.<locals>.new_type
Get the 20-byte address which sent this transaction.

sender
Convenience property for the return value of *get_sender*

validate() → None
Hook called during instantiation to ensure that all transaction parameters pass validation rules.

BaseUnsignedTransaction

class eth.rlp.transactions.**BaseUnsignedTransaction**(**args*, ***kwargs*)

as_signed_transaction(*private_key: eth_keys.datatypes.PrivateKey*) → eth.rlp.transactions.BaseTransaction
Return a version of this transaction which has been signed using the provided *private_key*

3.6.5 Tools

Builders

Chain Builder

The chain builder utils are intended to reduce common boilerplace for both construction of chain classes as well as building up some desired chain state.

Note: These tools are best used in conjunction with `cytoolz.pipe`.

Constructing Chain Classes

The following utilities are provided to assist with constructing a chain class.

`eth.tools.builder.chain.`**`fork_at`**`()`
Adds the `vm_class` to the chain's `vm_configuration`.

```
from eth.chains.base import MiningChain
from eth.tools.builder.chain import build, fork_at

FrontierOnlyChain = build(MiningChain, fork_at(FrontierVM, 0))

# these two classes are functionally equivalent.
class FrontierOnlyChain(MiningChain):
    vm_configuration = (
        (0, FrontierVM),
    )
```

Note: This function is curriable.

The following pre-curried versions of this function are available as well, one for each mainnet fork.

- `frontier_at()`
- `homestead_at()`
- `tangerine_whistle_at()`
- `spurious_dragon_at()`
- `byzantium_at()`
- `constantinople_at()`

`eth.tools.builder.chain.`**`dao_fork_at`**`()`
Set the block number on which the DAO fork will happen. Requires that a version of the *HomesteadVM* is present in the chain's `vm_configuration`

`eth.tools.builder.chain.`**`disable_dao_fork`**`()`
Set the `support_dao_fork` flag to `False` on the *HomesteadVM*. Requires that presence of the *HomesteadVM* in the `vm_configuration`

eth.tools.builder.chain.**enable_pow_mining**()
Inject on demand generation of the proof of work mining seal on newly mined blocks into each of the chain's vms.

eth.tools.builder.chain.**disable_pow_check**()
Disable the proof of work validation check for each of the chain's vms. This allows for block mining without generation of the proof of work seal.

Note: blocks mined this way will not be importable on any chain that does not have proof of work disabled.

---

eth.tools.builder.chain.**name**()
Assign the given name to the chain class.

eth.tools.builder.chain.**chain_id**()
Set the `chain_id` for the chain class.

Initializing Chains

The following utilities are provided to assist with initializing a chain into the genesis state.

eth.tools.builder.chain.**genesis**()
Initialize the given chain class with the given genesis header parameters and chain state.

Building Chains

The following utilities are provided to assist with building out chains of blocks.

eth.tools.builder.chain.**copy**()
Make a copy of the chain at the given state. Actions performed on the resulting chain will not affect the original chain.

eth.tools.builder.chain.**import_block**()
Import the provided `block` into the chain.

eth.tools.builder.chain.**import_blocks**(**blocks*) → Callable[eth.chains.base.BaseChain, eth.chains.base.BaseChain]
Variadic argument version of *import_block()*

eth.tools.builder.chain.**mine_block**()
Mine a new block on the chain. Header parameters for the new block can be overridden using keyword arguments.

eth.tools.builder.chain.**mine_blocks**()
Variadic argument version of *mine_block()*

eth.tools.builder.chain.**chain_split**(**splits*) → Callable[eth.chains.base.BaseChain, Iterable[eth.chains.base.BaseChain]]
Construct and execute multiple concurrent forks of the chain.

Any number of forks may be executed. For each fork, provide an iterable of commands.

Returns the resulting chain objects for each fork.

```
chain_a, chain_b = build(
    mining_chain,
    chain_split(
        (mine_block(extra_data=b'chain-a'), mine_block()),
```

(continues on next page)

(continued from previous page)

```
        (mine_block(extra_data=b'chain-b'), mine_block(), mine_block()),
    ),
)
```

eth.tools.builder.chain.**at_block_number**()
Rewind the chain back to the given block number. Calls to things like `get_canonical_head` will still return the canonical head of the chain, however, you can use `mine_block` to mine fork chains.

Builder Tools

The JSON test fillers found in *eth.tools.fixtures* is a set of tools which facilitate creating standard JSON consensus tests as found in the ethereum/tests repository.

Note: Only VM and state tests are supported right now.

State Test Fillers

Tests are generated in two steps.

- First, a *test filler* is written that contains a high level description of the test case.
- Subsequently, the filler is compiled to the actual test in a process called filling, mainly consisting of calculating the resulting state root.

The test builder represents each stage as a nested dictionary. Helper functions are provided to assemble the filler file step by step in the correct format. The `fill_test()` function handles compilation and takes additional parameters that can't be inferred from the filler.

Creating a Filler

Fillers are generated in a functional fashion by piping a dictionary through a sequence of functions.

```
filler = pipe(
    setup_main_filler("test"),
    pre_state(
        (sender, "balance", 1),
        (receiver, "balance", 0),
    ),
    expect(
        networks=["Frontier"],
        transaction={
            "to": receiver,
            "value": 1,
            "secretKey": sender_key,
        },
        post_state=[
            [sender, "balance", 0],
            [receiver, "balance", 1],
        ]
    )
)
```

Note: Note that `setup_filler()` returns a dictionary, whereas all of the following functions such as `pre_state()`, `expect()`, expect to be passed a dictionary as their single argument and return an updated version of the dictionary.

eth.tools.fixtures.fillers.common.**setup_main_filler**(*name: str*, *environment: Dict[Any, Any] = None*) → Dict[str, Dict[str, Any]]

Kick off the filler generation process by creating the general filler scaffold with a test name and general information about the testing environment.

For tests for the main chain, the *environment* parameter is expected to be a dictionary with some or all of the following keys:

key	description
`"currentCoinbase"`	the coinbase address
`"currentNumber"`	the block number
`"previousHash"`	the hash of the parent block
`"currentDifficulty"`	the block's difficulty
`"currentGasLimit"`	the block's gas limit
`"currentTimestamp"`	the timestamp of the block

eth.tools.fixtures.fillers.**pre_state**(**raw_state, filler: Dict[str, Any]*) → None

Specify the state prior to the test execution. Multiple invocations don't override the state but extend it instead.

In general, the elements of *state_definitions* are nested dictionaries of the following form:

```
{
    address: {
        "nonce": <account nonce>,
        "balance": <account balance>,
        "code": <account code>,
        "storage": {
            <storage slot>: <storage value>
        }
    }
}
```

To avoid unnecessary nesting especially if only few fields per account are specified, the following and similar formats are possible as well:

```
(address, "balance", <account balance>)
(address, "storage", <storage slot>, <storage value>)
(address, "storage", {<storage slot>: <storage value>})
(address, {"balance", <account balance>})
```

eth.tools.fixtures.fillers.**execution**()

For VM tests, specify the code that is being run as well as the current state of the EVM. State tests don't support this object. The parameter is a dictionary specifying some or all of the following keys:

key	description
`"address"`	the address of the account executing the code
`"caller"`	the caller address
`"origin"`	the origin address (defaulting to the caller address)
`"value"`	the value of the call
`"data"`	the data passed with the call
`"gasPrice"`	the gas price of the call
`"gas"`	the amount of gas allocated for the call
`"code"`	the bytecode to execute
`"vyperLLLCode"`	the code in Vyper LLL (compiled to bytecode automatically)

eth.tools.fixtures.fillers.**expect**(*post_state: Dict[str, Any] = None*, *networks: Any = None*, *transaction: eth.typing.TransactionDict = None*) → Callable[..., Dict[str, Any]]

Specify the expected result for the test.

For state tests, multiple expectations can be given, differing in the transaction data, gas limit, and value, in the applicable networks, and as a result also in the post state. VM tests support only a single expectation with no specified network and no transaction (here, its role is played by *execution()*).

- `post_state` is a list of state definition in the same form as expected by *pre_state()*. State items that are not set explicitly default to their pre state.
- **networks defines the forks under which the expectation is applicable. It should be a** sublist of the following identifiers (also available in *ALL_FORKS*):
  - `"Frontier"`
  - `"Homestead"`
  - `"EIP150"`
  - `"EIP158"`
  - `"Byzantium"`
- `transaction` is a dictionary coming in two variants. For the main shard:

key	description
`"data"`	the transaction data,
`"gasLimit"`	the transaction gas limit,
`"gasPrice"`	the gas price,
`"nonce"`	the transaction nonce,
`"value"`	the transaction value

In addition, one should specify either the signature itself (via keys `"v"`, `"r"`, and `"s"`) or a private key used for signing (via `"secretKey"`).

3.6.6 Virtual Machine

Computation

BaseComputation

class eth.vm.computation.**BaseComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactionContext*)

The base class for all execution computations.

Note: Each `BaseComputation` class must be configured with:

`opcodes`: A mapping from the opcode integer value to the logic function for the opcode.

`_precompiles`: A mapping of contract address to the precompile function for execution of pre-compiled contracts.

---

apply_child_computation(*child_msg: eth.vm.message.Message*) → eth.vm.computation.BaseComputation
Apply the vm message `child_msg` as a child computation.

classmethod apply_computation(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactionContext*) → eth.vm.computation.BaseComputation
Perform the computation that would be triggered by the VM message.

apply_create_message() → eth.vm.computation.BaseComputation
Execution of a VM message to create a new contract.

apply_message() → eth.vm.computation.BaseComputation
Execution of a VM message.

consume_gas(*amount: int*, *reason: str*) → None
Consume `amount` of gas from the remaining gas. Raise *eth.exceptions.OutOfGas* if there is not enough gas remaining.

extend_memory(*start_position: int*, *size: int*) → None
Extend the size of the memory to be at minimum `start_position + size` bytes in length. Raise *eth.exceptions.OutOfGas* if there is not enough gas to pay for extending the memory.

is_error
Return `True` if the computation resulted in an error.

is_origin_computation
Return `True` if this computation is the outermost computation at `depth == 0`.

is_success
Return `True` if the computation did not result in an error.

memory_read(*start_position: int*, *size: int*) → memoryview
Read and return a view of `size` bytes from memory starting at `start_position`.

memory_read_bytes(*start_position: int*, *size: int*) → bytes
Read and return `size` bytes from memory starting at `start_position`.

memory_write(*start_position: int*, *size: int*, *value: bytes*) → None
Write `value` to memory at `start_position`. Require that `len(value) == size`.

output
Get the return value of the computation.

prepare_child_message(*gas: int, to: NewType.<locals>.new_type, value: int, data: Union[bytes, memoryview], code: bytes, **kwargs*) → eth.vm.message.Message
Helper method for creating a child computation.

raise_if_error() → None
If there was an error during computation, raise it as an exception immediately.

Raises ***VMError*** –

refund_gas(*amount: int*) → None
Add `amount` of gas to the pool of gas marked to be refunded.

return_gas(*amount: int*) → None
Return `amount` of gas to the available gas pool.

should_burn_gas
Return `True` if the remaining gas should be burned.

should_erase_return_data
Return `True` if the return data should be zerod out due to an error.

should_return_gas
Return `True` if the remaining gas should be returned.

stack_dup(*position: int*) → None
Duplicate the stack item at `position` and pushes it onto the stack.

stack_pop(*num_items: int = 1, type_hint: str = None*) → Any
Pop and return a number of items equal to `num_items` from the stack. `type_hint` can be either `'uint256'` or `'bytes'`. The return value will be an `int` or `bytes` type depending on the value provided for the `type_hint`.

Raise *eth.exceptions.InsufficientStack* if there are not enough items on the stack.

stack_push(*value: Union[int, bytes]*) → None
Push `value` onto the stack.

Raise *eth.exceptions.StackDepthLimit* if the stack is full.

stack_swap(*position: int*) → None
Swap the item on the top of the stack with the item at `position`.

CodeStream

class eth.vm.code_stream.**CodeStream**(*code_bytes: bytes*)

ExecutionContext

class eth.vm.execution_context.**ExecutionContext**(*coinbase: NewType.<locals>.new_type, timestamp: int, block_number: int, difficulty: int, gas_limit: int, prev_hashes: Iterable[NewType.<locals>.new_type]*)

GasMeter

class eth.vm.gas_meter.**GasMeter**(*start_gas: int, refund_strategy: Callable[[int, int], int] = <function default_refund_strategy>*)

Memory

class eth.vm.memory.**Memory**
VM Memory

read(*start_position: int*, *size: int*) → memoryview
Return a view into the memory

read_bytes(*start_position: int*, *size: int*) → bytes
Read a value from memory and return a fresh bytes instance

write(*start_position: int*, *size: int*, *value: bytes*) → None
Write *value* into memory.

Message

class eth.vm.message.**Message**(*gas: int, to: NewType.<locals>.new_type, sender: NewType.<locals>.new_type, value: int, data: Union[bytes, memoryview], code: bytes, depth: int = 0, create_address: NewType.<locals>.new_type = None, code_address: NewType.<locals>.new_type = None, should_transfer_value: bool = True, is_static: bool = False*)
A message for VM computation.

Opcode

class eth.vm.opcode.**Opcode**

classmethod as_opcode(*logic_fn: Callable[..., Any], mnemonic: str, gas_cost: int*) → Type[T]
Class factory method for turning vanilla functions into Opcode classes.

VM

BaseVM

class eth.vm.base.**BaseVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

classmethod compute_difficulty(*parent_header: eth.rlp.headers.BlockHeader*, *timestamp: int*) → int
Compute the difficulty for a block header.

Parameters

- **parent_header** – the parent header
- **timestamp** – the timestamp of the child header

configure_header(***header_params*) → eth.rlp.headers.BlockHeader
Setup the current header with the provided parameters. This can be used to set fields like the gas limit or timestamp to value different than their computed defaults.

classmethod create_header_from_parent(*parent_header: eth.rlp.headers.BlockHeader*, ***header_params*) → eth.rlp.headers.BlockHeader
Creates and initializes a new block header from the provided *parent_header*.

static get_block_reward() → int
Return the amount in **wei** that should be given to a miner as a reward for this block.

> **Note:** This is an abstract method that must be implemented in subclasses

classmethod get_nephew_reward() → int
Return the reward which should be given to the miner of the given *nephew*.

> **Note:** This is an abstract method that must be implemented in subclasses

static get_uncle_reward(*block_number: int*, *uncle: eth.rlp.blocks.BaseBlock*) → int
Return the reward which should be given to the miner of the given *uncle*.

> **Note:** This is an abstract method that must be implemented in subclasses

make_receipt(*base_header: eth.rlp.headers.BlockHeader*, *transaction: eth.rlp.transactions.BaseTransaction*, *computation: eth.vm.computation.BaseComputation*, *state: eth.vm.state.BaseState*) → eth.rlp.receipts.Receipt
Generate the receipt resulting from applying the transaction.

Parameters

- **base_header** – the header of the block before the transaction was applied.
- **transaction** – the transaction used to generate the receipt
- **computation** – the result of running the transaction computation
- **state** – the resulting state, after executing the computation

Returns receipt

validate_transaction_against_header(*base_header: eth.rlp.headers.BlockHeader*, *transaction: eth.rlp.transactions.BaseTransaction*) → None
Validate that the given transaction is valid to apply to the given header.

Parameters

- **base_header** – header before applying the transaction
- **transaction** – the transaction to validate

Raises ValidationError if the transaction is not valid to apply

VM

class eth.vm.base.**VM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)
The *BaseVM* class represents the Chain rules for a specific protocol definition such as the Frontier or Homestead network.

> **Note:** Each *BaseVM* class must be configured with:

- `block_class`: The `Block` class for blocks in this VM ruleset.
- `_state_class`: The `State` class used by this VM for execution.

---

apply_all_transactions(*transactions: Tuple[eth.rlp.transactions.BaseTransaction, ...], base_header: eth.rlp.headers.BlockHeader*) → Tuple[eth.rlp.headers.BlockHeader, Tuple[eth.rlp.receipts.Receipt, ...], Tuple[eth.vm.computation.BaseComputation, ...]]

Determine the results of applying all transactions to the base header. This does *not* update the current block or header of the VM.

Parameters

- **transactions** – an iterable of all transactions to apply
- **base_header** – the starting header to apply transactions to

Returns the final header, the receipts of each transaction, and the computations

apply_transaction(*header: eth.rlp.headers.BlockHeader, transaction: eth.rlp.transactions.BaseTransaction*) → Tuple[eth.rlp.headers.BlockHeader, eth.rlp.receipts.Receipt, eth.vm.computation.BaseComputation]

Apply the transaction to the current block. This is a wrapper around `apply_transaction()` with some extra orchestration logic.

Parameters

- **header** – header of the block before application
- **transaction** – to apply

create_transaction(**args*, ***kwargs*) → eth.rlp.transactions.BaseTransaction

Proxy for instantiating a signed transaction for this VM.

classmethod create_unsigned_transaction(**, nonce: int, gas_price: int, gas: int, to: NewType.<locals>.new_type, value: int, data: bytes*) → eth.rlp.transactions.BaseUnsignedTransaction

Proxy for instantiating an unsigned transaction for this VM.

execute_bytecode(*origin: NewType.<locals>.new_type, gas_price: int, gas: int, to: NewType.<locals>.new_type, sender: NewType.<locals>.new_type, value: int, data: bytes, code: bytes, code_address: NewType.<locals>.new_type = None*) → eth.vm.computation.BaseComputation

Execute raw bytecode in the context of the current state of the virtual machine.

finalize_block(*block: eth.rlp.blocks.BaseBlock*) → eth.rlp.blocks.BaseBlock

Perform any finalization steps like awarding the block mining reward.

classmethod generate_block_from_parent_header_and_coinbase(*parent_header: eth.rlp.headers.BlockHeader, coinbase: NewType.<locals>.new_type*) → eth.rlp.blocks.BaseBlock

Generate block from parent header and coinbase.

classmethod get_block_class() → Type[eth.rlp.blocks.BaseBlock]

Return the `Block` class that this VM uses for blocks.

classmethod get_state_class() → Type[eth.vm.state.BaseState]

Return the class that this VM uses for states.

classmethod get_transaction_class() → Type[eth.rlp.transactions.BaseTransaction]
Return the class that this VM uses for transactions.

import_block(*block: eth.rlp.blocks.BaseBlock*) → eth.rlp.blocks.BaseBlock
Import the given block to the chain.

mine_block(**args*, ***kwargs*) → eth.rlp.blocks.BaseBlock
Mine the current block. Proxies to self.pack_block method.

pack_block(*block: eth.rlp.blocks.BaseBlock*, **args*, ***kwargs*) → eth.rlp.blocks.BaseBlock
Pack block for mining.

Parameters

- **coinbase** (*bytes*) – 20-byte public address to receive block reward
- **uncles_hash** (*bytes*) – 32 bytes
- **state_root** (*bytes*) – 32 bytes
- **transaction_root** (*bytes*) – 32 bytes
- **receipt_root** (*bytes*) – 32 bytes
- **bloom** (*int*) –
- **gas_used** (*int*) –
- **extra_data** (*bytes*) – 32 bytes
- **mix_hash** (*bytes*) – 32 bytes
- **nonce** (*bytes*) – 8 bytes

previous_hashes
Convenience API for accessing the previous 255 block hashes.

validate_block(*block: eth.rlp.blocks.BaseBlock*) → None
Validate the the given block.

classmethod validate_header(*header: eth.rlp.headers.BlockHeader*, *parent_header: eth.rlp.headers.BlockHeader*, *check_seal: bool = True*) → None

Raises **eth.exceptions.ValidationError** – if the header is not valid

classmethod validate_seal(*header: eth.rlp.headers.BlockHeader*) → None
Validate the seal on the given header.

classmethod validate_uncle(*block: eth.rlp.blocks.BaseBlock*, *uncle: eth.rlp.blocks.BaseBlock*, *uncle_parent: eth.rlp.blocks.BaseBlock*) → None
Validate the given uncle in the context of the given block.

Stack

class eth.vm.stack.**Stack**
VM Stack

dup(*position: int*) → None
Perform a DUP operation on the stack.

pop(*num_items: int*, *type_hint: str*) → Union[int, bytes, Tuple[Union[int, bytes], ...]]
Pop an item off the stack.

Note: This function is optimized for speed over readability.

push(*value: Union[int, bytes]*) → None
Push an item onto the stack.

swap(*position: int*) → None
Perform a SWAP operation on the stack.

State

BaseState

class eth.vm.state.**BaseState**(*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionContext*, *state_root: bytes*)
The base class that encapsulates all of the various moving parts related to the state of the VM during execution. Each *BaseVM* must be configured with a subclass of the *BaseState*.

Note: Each *BaseState* class must be configured with:

- computation_class: The *BaseComputation* class for vm execution.
- transaction_context_class: The TransactionContext class for vm execution.

---

apply_transaction(*transaction: BaseTransaction*) → Tuple[bytes, BaseComputation]
Apply transaction to the vm state

Parameters **transaction** – the transaction to apply

Returns the new state root, and the computation

block_number
Return the current block_number from the current execution_context

coinbase
Return the current coinbase from the current execution_context

commit(*snapshot: Tuple[bytes, Tuple[uuid.UUID, uuid.UUID]]*) → None
Commit the journal to the point where the snapshot was taken. This will merge in any changesets that were recorded *after* the snapshot changeset.

difficulty
Return the current difficulty from the current execution_context

gas_limit
Return the current gas_limit from the current transaction_context

classmethod get_account_db_class() → Type[eth.db.account.BaseAccountDB]
Return the *BaseAccountDB* class that the state class uses.

get_ancestor_hash(*block_number: int*) → NewType.<locals>.new_type
Return the hash for the ancestor block with number block_number. Return the empty bytestring b'' if the block number is outside of the range of available block numbers (typically the last 255 blocks).

get_computation(*message: eth.vm.message.Message*, *transaction_context: BaseTransactionContext*) → BaseComputation
Return a computation instance for the given *message* and *transaction_context*

classmethod get_transaction_context_class() → Type[BaseTransactionContext]
Return the *BaseTransactionContext* class that the state class uses.

revert(*snapshot: Tuple[bytes, Tuple[uuid.UUID, uuid.UUID]]*) → None
Revert the VM to the state at the snapshot

snapshot() → Tuple[bytes, Tuple[uuid.UUID, uuid.UUID]]
Perform a full snapshot of the current state.

Snapshots are a combination of the *state_root* at the time of the snapshot and the id of the changeset from the journaled DB.

state_root
Return the current state_root from the underlying database

timestamp
Return the current timestamp from the current execution_context

BaseTransactionExecutor

class eth.vm.state.**BaseTransactionExecutor**(*vm_state: eth.vm.state.BaseState*)

BaseTransactionContext

class eth.vm.transaction_context.**BaseTransactionContext**(*gas_price: int*, *origin: NewType.<locals>.new_type*)
This immutable object houses information that remains constant for the entire context of the VM execution.

Forks

Frontier

FrontierVM

class eth.vm.forks.frontier.**FrontierVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

block_class
alias of eth.vm.forks.frontier.blocks.FrontierBlock

static compute_difficulty(*parent_header: eth.rlp.headers.BlockHeader*, *timestamp: int*) → int
Computes the difficulty for a frontier block based on the parent block.

static get_block_reward() → int
Return the amount in **wei** that should be given to a miner as a reward for this block.

> **Note:** This is an abstract method that must be implemented in subclasses

classmethod get_nephew_reward() → int
Return the reward which should be given to the miner of the given *nephew*.

> **Note:** This is an abstract method that must be implemented in subclasses

static get_uncle_reward(*block_number: int*, *uncle: eth.rlp.blocks.BaseBlock*) → int
Return the reward which should be given to the miner of the given *uncle*.

Note: This is an abstract method that must be implemented in subclasses

FrontierState

class eth.vm.forks.frontier.state.**FrontierState**(*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionContext*, *state_root: bytes*)

account_db_class
alias of *eth.db.account.AccountDB*

computation_class
alias of *eth.vm.forks.frontier.computation.FrontierComputation*

transaction_context_class
alias of eth.vm.forks.frontier.transaction_context.FrontierTransactionContext

transaction_executor
alias of FrontierTransactionExecutor

FrontierComputation

class eth.vm.forks.frontier.computation.**FrontierComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactionC*

A class for all execution computations in the Frontier fork. Inherits from *BaseComputation*

apply_create_message() → eth.vm.computation.BaseComputation
Execution of a VM message to create a new contract.

apply_message() → eth.vm.computation.BaseComputation
Execution of a VM message.

Homestead

HomesteadVM

class eth.vm.forks.homestead.**HomesteadVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

block_class
alias of eth.vm.forks.homestead.blocks.HomesteadBlock

static compute_difficulty(*parent_header: eth.rlp.headers.BlockHeader*, *timestamp: int*) → int
Computes the difficulty for a homestead block based on the parent block.

HomesteadState

class eth.vm.forks.homestead.state.**HomesteadState**(*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionContext*, *state_root: bytes*)

computation_class
alias of *eth.vm.forks.homestead.computation.HomesteadComputation*

HomesteadComputation

class eth.vm.forks.homestead.computation.**HomesteadComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactio*

A class for all execution computations in the `Frontier` fork. Inherits from *FrontierComputation*

apply_create_message() → eth.vm.computation.BaseComputation
Execution of a VM message to create a new contract.

TangerineWhistle

TangerineWhistleVM

class eth.vm.forks.tangerine_whistle.**TangerineWhistleVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

TangerineWhistleState

class eth.vm.forks.tangerine_whistle.state.**TangerineWhistleState**(*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionC state_root: bytes*)

computation_class
alias of *eth.vm.forks.tangerine_whistle.computation.TangerineWhistleComputation*

TangerineWhistleComputation

class eth.vm.forks.tangerine_whistle.computation.**TangerineWhistleComputation**(*state: eth.vm.state.BaseSta mes- sage: eth.vm.message.Mes trans- ac- tion_context: eth.vm.transaction_*

A class for all execution computations in the TangerineWhistle fork. Inherits from *HomesteadComputation*

SpuriousDragon

SpuriousDragonVM

class eth.vm.forks.spurious_dragon.**SpuriousDragonVM**(*header: eth.rlp.headers.BlockHeader, chaindb: eth.db.chain.BaseChainDB*)

block_class
alias of eth.vm.forks.spurious_dragon.blocks.SpuriousDragonBlock

SpuriousDragonState

class eth.vm.forks.spurious_dragon.state.**SpuriousDragonState**(*db: eth.db.backends.base.BaseDB, execution_context: eth.vm.execution_context.ExecutionContext, state_root: bytes*)

computation_class
alias of *eth.vm.forks.spurious_dragon.computation.SpuriousDragonComputation*

transaction_executor
alias of SpuriousDragonTransactionExecutor

SpuriousDragonComputation

class eth.vm.forks.spurious_dragon.computation.**SpuriousDragonComputation**(*state: eth.vm.state.BaseState, mes- sage: eth.vm.message.Message, trans- ac- tion_context: eth.vm.transaction_contex*

A class for all execution computations in the SpuriousDragon fork. Inherits from

HomesteadComputation

apply_create_message() → eth.vm.computation.BaseComputation
Execution of a VM message to create a new contract.

Byzantium

ByzantiumVM

class eth.vm.forks.byzantium.**ByzantiumVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

block_class
alias of eth.vm.forks.byzantium.blocks.ByzantiumBlock

compute_difficulty
https://github.com/ethereum/EIPs/issues/100

static get_block_reward() → int
Return the amount in **wei** that should be given to a miner as a reward for this block.

> **Note:** This is an abstract method that must be implemented in subclasses

ByzantiumState

class eth.vm.forks.byzantium.state.**ByzantiumState**(*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionContext*, *state_root: bytes*)

computation_class
alias of *eth.vm.forks.byzantium.computation.ByzantiumComputation*

ByzantiumComputation

class eth.vm.forks.byzantium.computation.**ByzantiumComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactio*

A class for all execution computations in the Byzantium fork. Inherits from *SpuriousDragonComputation*

3.7 Contributing

Thank you for your interest in contributing! We welcome all contributions no matter their size. Please read along to learn how to get started. If you get stuck, feel free to reach for help in our Gitter channel.

3.7.1 Setting the stage

First we need to clone the Py-EVM repository. Py-EVM depends on a submodule of the common tests across all clients, so we need to clone the repo with the `--recursive` flag. Example:

```
$ git clone --recursive https://github.com/ethereum/py-evm.git
```

Optional: Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

After we have activated our virtual environment, installing all dependencies that are needed to run, develop and test all code in this repository is as easy as:

```
pip install -e .[dev]
```

3.7.2 Running the tests

A great way to explore the code base is to run the tests.

We can run all tests with:

```
pytest
```

However, running the entire test suite does take a very long time so often we just want to run a subset instead, like:

```
pytest tests/core/padding-utils/test_padding.py
```

We can also install `tox` to run the full test suite which also covers things like testing the code against different Python versions, linting etc.

It is important to understand that each Pull Request must pass the full test suite as part of the CI check, hence it is often convenient to have `tox` installed locally as well.

3.7.3 Code Style

When multiple people are working on the same body of code, it is important that they write code that conforms to a similar style. It often doesn't matter as much which style, but rather that they conform to one style.

To ensure your contribution conforms to the style being used in this project, we encourage you to read our style guide.

3.7.4 Type Hints

The code bases is transitioning to use type hints. Type hints make it easy to prevent certain types of bugs, enable richer tooling and enhance the documentation, making the code easier to follow.

All new code is required to land with type hints with the exception of test code that is not expected to use type hints.

All parameters as well as the return type of defs are expected to be typed with the exception of `self` and `cls` as seen in the following example.

```
def __init__(self, wrapped_db: BaseDB) -> None:
    self.wrapped_db = wrapped_db
    self.reset()
```

3.7.5 Documentation

Good documentation will lead to quicker adoption and happier users. Please check out our guide on how to create documentation for the Python Ethereum ecosystem.

3.7.6 Pull Requests

It's a good idea to make pull requests early on. A pull request represents the start of a discussion, and doesn't necessarily need to be the final, finished submission.

GitHub's documentation for working on pull requests is available here.

Once you've made a pull request take a look at the Circle CI build status in the GitHub interface and make sure all tests are passing. In general pull requests that do not pass the CI build yet won't get reviewed unless explicitly requested.

3.7.7 Releasing

One time setup

Pandoc is required for transforming the markdown README to the proper format to render correctly on pypi.

For Debian-like systems:

```
apt install pandoc
```

Or on OSX:

```
brew install pandoc
```

Final test before each release

Before releasing a new version, build and test the package that will be released:

```
git checkout master && git pull

make package
```

Push the release to github & pypi

After confirming that the release package looks okay, release a new version:

```
make release bump=$$VERSION_PART_TO_BUMP$$
```

Which version part to bump

The version format for this repo is `{major}.{minor}.{patch}` for stable, and `{major}.{minor}.{patch}-{stage}.{devnum}` for unstable (`stage` can be alpha or beta).

During a release, specify which part to bump, like `make release bump=minor` or `make release bump=devnum`.

If you are in a beta version, `make release bump=stage` will switch to a stable.

To issue an unstable version when the current version is stable, specify the new version explicitly, like `make release bump="--new-version 4.0.0-alpha.1 devnum"`

3.8 Code of Conduct

3.8.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

3.8.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

3.8.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

3.8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

3.8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at piper@pipermerriam.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

3.8.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

Python Module Index

e

Index

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

V

W