Populus Documentation

Release 6.17.2

The Ethereum Foundation

Apr 17, 2024

INTRO

Important: For **ENS (Ethereum Name Service)** users, web3.py `v6.6.0` introduced ENS name normalization standard ENSIP-15. This update to ENS name validation and normalization won't affect ~99% of names but may prevent invalid names from being created and from interacting with the ENS contracts via web3.py. We feel strongly that this change, though breaking, is in the best interest of our users as it ensures compatibility with the latest ENS standards.

web3.py is a Python library for interacting with Ethereum.

It's commonly found in decentralized apps (dapps) to help with sending transactions, interacting with smart contracts, reading block data, and a variety of other use cases.

The original API was derived from the Web3.js Javascript API, but has since evolved toward the needs and creature comforts of Python developers.

CHAPTER

ONE

GETTING STARTED

Note: Brand new to Ethereum?

0. Don't travel alone! Join the Ethereum Python Community Discord.
1. Read this blog post series for a gentle introduction to Ethereum blockchain concepts.
2. The *Overview* page will give you a quick idea of what else web3.py can do.
3. Try building a little something!

- Ready to code? → *Quickstart*
- Interested in a quick tour? → *Overview*
- Need help debugging? → StackExchange
- Found a bug? → *Contribute*
- Want to chat? → Discord
- Read the source? → Github
- Looking for inspiration? → *Resources and Learning Material*

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Quickstart

Note: All code starting with a `$` is meant to run on your terminal. All code starting with a >>> is meant to run in a python interpreter, like ipython.

2.1.1 Installation

web3.py can be installed (preferably in a *virtualenv*) using `pip` as follows:

```
$ pip install web3
```

Note: If you run into problems during installation, you might have a broken environment. See the troubleshooting guide to *setting up a clean environment*.

2.1.2 Using Web3

This library depends on a connection to an Ethereum node. We call these connections *Providers* and there are several ways to configure them. The full details can be found in the *Providers* documentation. This Quickstart guide will highlight a couple of the most common use cases.

Test Provider

If you're just learning the ropes or doing some quick prototyping, you can use a test provider, eth-tester. This provider includes some accounts prepopulated with test ether and instantly includes each transaction into a block. web3.py makes this test provider available via `EthereumTesterProvider`.

Note: The `EthereumTesterProvider` requires additional dependencies. Install them via `pip install "web3[tester]"`, then import and instantiate the provider as seen below.

```
>>> from web3 import Web3, EthereumTesterProvider
>>> w3 = Web3(EthereumTesterProvider())
>>> w3.is_connected()
True
```

Local Providers

The hardware requirements are steep, but the safest way to interact with Ethereum is to run an Ethereum client on your own hardware. For locally run nodes, an IPC connection is the most secure option, but HTTP and websocket configurations are also available. By default, the popular Geth client exposes port `8545` to serve HTTP requests and `8546` for websocket requests. Connecting to this local node can be done as follows:

```
>>> from web3 import Web3, AsyncWeb3

# IPCProvider:
>>> w3 = Web3(Web3.IPCProvider('./path/to/geth.ipc'))

# HTTPProvider:
>>> w3 = Web3(Web3.HTTPProvider('http://127.0.0.1:8545'))

# WebsocketProvider:
>>> w3 = Web3(Web3.WebsocketProvider('wss://127.0.0.1:8546'))

>>> w3.is_connected()
True

# AsyncHTTPProvider:
>>> w3 = AsyncWeb3(AsyncWeb3.AsyncHTTPProvider('http://127.0.0.1:8545'))

>>> await w3.is_connected()
True
```

Remote Providers

The quickest way to interact with the Ethereum blockchain is to use a remote node provider. You can connect to a remote node by specifying the endpoint, just like the previous local node example:

```
>>> from web3 import Web3, AsyncWeb3

>>> w3 = Web3(Web3.HTTPProvider('https://<your-provider-url>'))

>>> w3 = AsyncWeb3(AsyncWeb3.AsyncHTTPProvider('https://<your-provider-url>'))

>>> w3 = Web3(Web3.WebsocketProvider('wss://<your-provider-url>'))
```

This endpoint is provided by the remote node service, typically after you create an account.

2.1.3 Getting Blockchain Info

It's time to start using web3.py! Once properly configured, the w3 instance will allow you to interact with the Ethereum blockchain. Try getting all the information about the latest block:

```
>>> w3.eth.get_block('latest')
{'difficulty': 1,
 'gasLimit': 6283185,
 'gasUsed': 0,
 'hash': HexBytes('0x53b983fe73e16f6ed8178f6c0e0b91f23dc9dad4cb30d0831f178291ffeb8750'),
 'logsBloom': HexBytes(
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪'),
 'miner': '0x0000000000000000000000000000000000000000',
 'mixHash': HexBytes('0x0000000000000000000000000000000000000000000000000000000000000000
↪'),
 'nonce': HexBytes('0x0000000000000000'),
 'number': 0,
 'parentHash': HexBytes(
↪'0x0000000000000000000000000000000000000000000000000000000000000000'),
 'proofOfAuthorityData': HexBytes(
↪'0x0000000000000000000000000000000000000000000000000000000000000000dddc391ab2bf6701c74d0c8698c2e13355
↪'),
 'receiptsRoot': HexBytes(
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'),
 'sha3Uncles': HexBytes(
↪'0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347'),
 'size': 622,
 'stateRoot': HexBytes(
↪'0x1f5e460eb84dc0606ab74189dbcfe617300549f8f4778c3c9081c119b5b5d1c1'),
 'timestamp': 0,
 'totalDifficulty': 1,
 'transactions': [],
 'transactionsRoot': HexBytes(
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'),
 'uncles': []}
```

web3.py can help you read block data, sign and send transactions, deploy and interact with contracts, and a number of other features.

A few suggestions from here:

- The *Overview* page provides a summary of web3.py's features.
- The `w3.eth` API contains the most frequently used methods.
- A guide to *Contracts* includes deployment and usage examples.
- The nuances of *Sending Transactions* are explained in another guide.
- For other inspiration, see the *Examples*.

Note: It is recommended that your development environment have the `PYTHONWARNINGS=default` environment variable set. Some deprecation warnings will not show up without this variable being set.

2.2 Overview

The purpose of this page is to give you a sense of everything web3.py can do and to serve as a quick reference guide. You'll find a summary of each feature with links to learn more. You may also be interested in the *Examples* page, which demonstrates some of these features in greater detail.

2.2.1 Configuration

After installing web3.py (via `pip install web3`), you'll need to configure a provider endpoint and any middleware you want to use beyond the defaults.

Providers

Providers are how web3.py connects to a blockchain. The library comes with the following built-in providers:

- `IPCProvider` for connecting to ipc socket based JSON-RPC servers.
- `HTTPProvider` for connecting to http and https based JSON-RPC servers.
- `WebsocketProvider` for connecting to ws and wss websocket based JSON-RPC servers.
- `AsyncHTTPProvider` for connecting to http and https based JSON-RPC servers.

Examples

```
>>> from web3 import Web3, AsyncWeb3

# IPCProvider:
>>> w3 = Web3(Web3.IPCProvider('./path/to/geth.ipc'))

# HTTPProvider:
>>> w3 = Web3(Web3.HTTPProvider('http://127.0.0.1:8545'))

# WebsocketProvider:
>>> w3 = Web3(Web3.WebsocketProvider('ws://127.0.0.1:8546'))

>>> w3.is_connected()
```

(continues on next page)

(continued from previous page)

```
True

# AsyncHTTPProvider:
>>> w3 = AsyncWeb3(AsyncWeb3.AsyncHTTPProvider('http://127.0.0.1:8545'))

>>> await w3.is_connected()
True
```

For more context, see the *Providers* documentation.

Middleware

Your web3.py instance may be further configured via *Middleware*.

web3.py middleware is described using an onion metaphor, where each layer of middleware may affect both the incoming request and outgoing response from your provider. The documentation includes a *visualization* of this idea.

Several middleware are *included by default*. You may add to (`add`, `inject`, `replace`) or disable (`remove`, `clear`) any of these middleware.

2.2.2 Accounts and Private Keys

Private keys are required to approve any transaction made on your behalf. The manner in which your key is secured will determine how you create and send transactions in web3.py.

A local node, like Geth, may manage your keys for you. You can reference those keys using the `web3.eth.accounts` property.

A hosted node, like Infura, will have no knowledge of your keys. In this case, you'll need to have your private key available locally for signing transactions.

Full documentation on the distinction between keys can be found *here*. The separate guide to *Sending Transactions* may also help clarify how to manage keys.

2.2.3 Base API

The *Web3* class includes a number of convenient utility functions:

Encoding and Decoding Helpers

- `Web3.is_encodable()`
- `Web3.to_bytes()`
- `Web3.to_hex()`
- `Web3.to_int()`
- `Web3.to_json()`
- `Web3.to_text()`

Address Helpers

- *Web3.is_address()*
- *Web3.is_checksum_address()*
- *Web3.to_checksum_address()*

Currency Conversions

- *Web3.from_wei()*
- *Web3.to_wei()*

Cryptographic Hashing

- *Web3.keccak()*
- *Web3.solidity_keccak()*

2.2.4 web3.eth API

The most commonly used APIs for interacting with Ethereum can be found under the `web3.eth` namespace. As a reminder, the *Examples* page will demonstrate how to use several of these methods.

Fetching Data

Viewing account balances (*get_balance*), transactions (*get_transaction*), and block data (*get_block*) are some of the most common starting points in web3.py.

API

- *web3.eth.get_balance()*
- *web3.eth.get_block()*
- *web3.eth.get_block_transaction_count()*
- *web3.eth.get_code()*
- *web3.eth.get_proof()*
- *web3.eth.get_storage_at()*
- *web3.eth.get_transaction()*
- *web3.eth.get_transaction_by_block()*
- *web3.eth.get_transaction_count()*
- *web3.eth.get_uncle_by_block()*
- *web3.eth.get_uncle_count()*

Sending Transactions

The most common use cases will be satisfied with *send_transaction* or the combination of *sign_transaction* and *send_raw_transaction*. For more context, see the full guide to *Sending Transactions*.

Note: If interacting with a smart contract, a dedicated API exists. See the next section, *Contracts*.

API

- *web3.eth.send_transaction()*
- *web3.eth.sign_transaction()*
- *web3.eth.send_raw_transaction()*
- *web3.eth.replace_transaction()*
- *web3.eth.modify_transaction()*
- *web3.eth.wait_for_transaction_receipt()*
- *web3.eth.get_transaction_receipt()*
- *web3.eth.sign()*
- *web3.eth.sign_typed_data()*
- *web3.eth.estimate_gas()*
- *web3.eth.generate_gas_price()*
- *web3.eth.set_gas_price_strategy()*

Contracts

web3.py can help you deploy, read from, or execute functions on a deployed contract.

Deployment requires that the contract already be compiled, with its bytecode and ABI available. This compilation step can be done within Remix or one of the many contract development frameworks, such as Ape.

Once the contract object is instantiated, calling `transact` on the *constructor* method will deploy an instance of the contract:

```
>>> ExampleContract = w3.eth.contract(abi=abi, bytecode=bytecode)
>>> tx_hash = ExampleContract.constructor().transact()
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> tx_receipt.contractAddress
'0x8a22225eD7eD460D7ee3842bce2402B9deaD23D3'
```

Once a deployed contract is loaded into a Contract object, the functions of that contract are available on the `functions` namespace:

```
>>> deployed_contract = w3.eth.contract(address=tx_receipt.contractAddress, abi=abi)
>>> deployed_contract.functions.myFunction(42).transact()
```

If you want to read data from a contract (or see the result of transaction locally, without executing it on the network), you can use the *ContractFunction.call* method, or the more concise *ContractCaller* syntax:

```
# Using ContractFunction.call
>>> deployed_contract.functions.getMyValue().call()
42

# Using ContractCaller
>>> deployed_contract.caller().getMyValue()
42
```

For more, see the full *Contracts* documentation.

API

- *web3.eth.contract()*
- *Contract.address*
- *Contract.abi*
- *Contract.bytecode*
- *Contract.bytecode_runtime*
- *Contract.functions*
- *Contract.events*
- *Contract.fallback*
- *Contract.constructor()*
- *Contract.encode_abi()*
- *web3.contract.ContractFunction*
- *web3.contract.ContractEvents*

Logs and Filters

If you want to react to new blocks being mined or specific events being emitted by a contract, you can leverage web3.py filters.

```
# Use case: filter for new blocks
>>> new_filter = web3.eth.filter('latest')

# Use case: filter for contract event "MyEvent"
>>> new_filter = deployed_contract.events.MyEvent.create_filter(fromBlock='latest')

# retrieve filter results:
>>> new_filter.get_all_entries()
>>> new_filter.get_new_entries()
```

More complex patterns for creating filters and polling for logs can be found in the *Monitoring Events* documentation.

API

- *web3.eth.filter()*
- *web3.eth.get_filter_changes()*
- *web3.eth.get_filter_logs()*
- *web3.eth.uninstall_filter()*
- *web3.eth.get_logs()*
- *Contract.events.your_event_name.create_filter()*
- *Contract.events.your_event_name.build_filter()*
- *Filter.get_new_entries()*
- *Filter.get_all_entries()*
- *Filter.format_entry()*
- *Filter.is_valid_entry()*

2.2.5 Net API

Some basic network properties are available on the `web3.net` object:

- *web3.net.listening*
- *web3.net.peer_count*
- *web3.net.version*

2.2.6 ethPM

ethPM allows you to package up your contracts for reuse or use contracts from another trusted registry. See the full details *here*.

2.2.7 ENS

Ethereum Name Service (ENS) provides the infrastructure for human-readable addresses. If an address is registered with the ENS registry, the domain name can be used in place of the address itself. For example, the registered domain name `ethereum.eth` will resolve to the address `0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe`. web3.py has support for ENS, documented *here*.

2.3 Release Notes

v6 Breaking Changes Summary
See the *v6 Migration Guide*

2.3.1 web3.py vpatch (2024-04-17)

No significant changes.

2.3.2 web3.py v6.17.2 (2024-04-17)

Bugfixes

- Revert pin of *eth-typing* to utilize *EthPM* types. (#3349)

2.3.3 web3.py v6.17.1 (2024-04-15)

Miscellaneous Changes

- #3339

2.3.4 web3.py v6.17.0 (2024-04-11)

Improved Documentation

- Add feedback survey banner to docs (#3325)
- Fix `eth_createAccessList` docs to reflect the correct behavior. (#3329)

Features

- Add request formatter for `maxFeePerBlobGas` when sending blob transactions. Add formatters for `blobGasPrice` and `blobGasUsed` for *eth_getTransactionReceipt*. (#3323)
- Add formatters to ensure that the result of a `eth_createAccessList` response can be plugged directly into an `accessList` in a transaction. (#3329)
- Add Cancun support to `EthereumTesterProvider` (for supported Python versions greater than 3.7); update Cancun-related fields in some internal types. (#3338)

Internal Changes - for web3.py Contributors

- Use `pre-commit` for linting the v6 branch (#3296)
- Add an upperpin at `eth-typing<4.2.0` due to removal of EthPM types in that lib (#3324)

2.3.5 web3.py v6.16.0 (2024-03-28)

Bugfixes

- Catch all types of `eth-abi` `DecodingError` in `EthereumTesterProvider->_make_request()` (#3267)
- Fix/update methods and decorators in `web3/_utils/abi.py` to address issues raised by `mypy` (#3273)
- Fix `process_log()` when parsing logs for events with indexed and non-indexed inputs. `get_event_data()` now compares log topics and event ABIs as hex values. (#3288)

- Fix `process_log` for `HexStr` inputs. Explicit type coercion of entry `topics` and `data` values. (#3292)
- Fix typing for json data argument to `eth_signTypedData`. (#3311)

Deprecations

- Deprecate Geth miner namespace (#2857)
- Deprecated `Contract.encodeABI()` in favor of `Contract.encode_abi()`. (#3280)

Features

- Implement `state_override` parameter for `eth_estimateGas` method. (#3164)
- Add formatters for new `Cancun` network upgrade block header fields: `blobGasUsed`, `excessBlobGas`, and `parentBeaconBlockRoot`. (#3224)
- Allow for configuring the `request_information_cache_size` for `PersistentConnectionProvider` classes. Issue a warning when the cache is full and unexpected behavior may occur. (#3226)
- Add `user_message` kwarg for human readable `Web3Exception` messages. (#3282)
- Add formatters for type 3 transaction fields `maxFeePerBlobGas` and `blobVersionedHashes`. (#3315)

Internal Changes - for web3.py Contributors

- Fix internal typing for functions used by `process_log`. (#3301)

Performance Improvements

- Utilize `async` functionality when popping responses from request manager cache for persistent connection providers. (#3305)

2.3.6 web3.py v6.15.1 (2024-02-05)

Bugfixes

- Handle new geth errors related to waiting for a transaction receipt while transactions are still being indexed. (#3217)

Improved Documentation

- Remove annual user survey prompt from docs (#3218)

2.4 Your Ethereum Node

2.4.1 Why do I need to connect to a node?

The Ethereum protocol defines a way for people to interact with smart contracts and each other over a network. In order to have up-to-date information about the status of contracts, balances, and new transactions, the protocol requires a connection to nodes on the network. These nodes are constantly sharing new data with each other.

web3.py is a python library for connecting to these nodes. It does not run its own node internally.

2.4.2 How do I choose which node to use?

Due to the nature of Ethereum, this is largely a question of personal preference, but it has significant ramifications on security and usability. Further, node software is evolving quickly, so please do your own research about the current options.

One of the key decisions is whether to use a local node or a hosted node. A quick summary is at *Local vs Hosted Nodes*.

A local node requires less trust than a hosted one. A malicious hosted node can give you incorrect information, log your sent transactions with your IP address, or simply go offline. Incorrect information can cause all kinds of problems, including loss of assets.

On the other hand, with a local node your machine is individually verifying all the transactions on the network, and providing you with the latest state. Unfortunately, this means using up a significant amount of disk space, and sometimes notable bandwidth and computation. Additionally, there is a big up-front time cost for downloading the full blockchain history.

If you want to have your node manage keys for you (a popular option), you must use a local node. Note that even if you run a node on your own machine, you are still trusting the node software with any accounts you create on the node.

You can find a list of node software at ethereum.org.

Some people decide that the time it takes to sync a local node from scratch is too high, especially if they are just exploring Ethereum for the first time. One way to work around this issue is to use a hosted node.

Hosted node options can also be found at ethereum.org. You can connect to a hosted node as if it were a local node, with a few caveats. It cannot (and *should not*) host private keys for you, meaning that some common methods like `w3.eth.send_transaction()` are not directly available. To send transactions to a hosted node, read about *Working with Local Private Keys*.

Once you decide what node option you want, you need to choose which network to connect to. Typically, you are choosing between the main network and one of the available test networks. See *Which network should I connect to?*

Can I use MetaMask as a node?

MetaMask is not a node. It is an interface for interacting with a node. Roughly, it's what you get if you turn web3.py into a browser extension.

By default, MetaMask connects to an Infura node. You can also set up MetaMask to use a node that you run locally.

If you are trying to use accounts that were already created in MetaMask, see *Why isn't my web3 instance connecting to the network?*

2.4.3 Which network should I connect to?

Once you have answered *How do I choose which node to use?* you have to pick which network to connect to. This is easy for some scenarios: if you have ether and you want to spend it, or you want to interact with any production smart contracts, then you connect to the main Ethereum network.

If you want to test these things without using real ether, though, then you need to connect to a test network. There are several test networks to choose from; view the list on ethereum.org.

Each network has its own version of Ether. Main network ether must be purchased, naturally, but test network ether is usually available for free. See *How do I get ether for my test network?*

Once you have decided which network to connect to, and set up your node for that network, you need to decide how to connect to it. There are a handful of options in most nodes. See *Choosing How to Connect to Your Node*.

2.5 Providers

The provider is how web3 talks to the blockchain. Providers take JSON-RPC requests and return the response. This is normally done by submitting the request to an HTTP or IPC socket based server.

Note: web3.py supports one provider per instance. If you have an advanced use case that requires multiple providers, create and configure a new web3 instance per connection.

If you are already happily connected to your Ethereum node, then you can skip the rest of the Providers section.

2.5.1 Choosing How to Connect to Your Node

Most nodes have a variety of ways to connect to them. If you have not decided what kind of node to use, head on over to *How do I choose which node to use?*

The most common ways to connect to your node are:

1. IPC (uses local filesystem: fastest and most secure)
2. Websockets (works remotely, faster than HTTP)
3. HTTP (more nodes support it)

If you're not sure how to decide, choose this way:

- If you have the option of running web3.py on the same machine as the node, choose IPC.
- If you must connect to a node on a different computer, use Websockets.
- If your node does not support Websockets, use HTTP.

Most nodes have a way of "turning off" connection options. We recommend turning off all connection options that you are not using. This provides a safer setup: it reduces the number of ways that malicious hackers can try to steal your ether.

Once you have decided how to connect, you specify the details using a Provider. Providers are web3.py classes that are configured for the kind of connection you want.

See:

- *`IPCProvider`*
- *`WebsocketProvider`*

- *WebsocketProviderV2*
- *HTTPProvider*
- *AsyncHTTPProvider*

Once you have configured your provider, for example:

```
from web3 import Web3
my_provider = Web3.IPCProvider('/my/node/ipc/path')
```

Then you are ready to initialize your Web3 instance, like so:

```
w3 = Web3(my_provider)
```

Finally, you are ready to *get started with web3.py*.

Provider via Environment Variable

Alternatively, you can set the environment variable `WEB3_PROVIDER_URI` before starting your script, and web3 will look for that provider first.

Valid formats for this environment variable are:

- `file:///path/to/node/rpc-json/file.ipc`
- `http://192.168.1.2:8545`
- `https://node.ontheweb.com`
- `ws://127.0.0.1:8546`

2.5.2 Auto-initialization Provider Shortcuts

Geth dev Proof of Authority

To connect to a `geth --dev` Proof of Authority instance with defaults:

```
>>> from web3.auto.gethdev import w3

# confirm that the connection succeeded
>>> w3.is_connected()
True
```

2.5.3 Built In Providers

Web3 ships with the following providers which are appropriate for connecting to local and remote JSON-RPC servers.

HTTPProvider

class web3.providers.rpc.**HTTPProvider**(*endpoint_uri*[, *request_kwargs*, *session*])

This provider handles interactions with an HTTP or HTTPS based JSON-RPC server.

- `endpoint_uri` should be the full URI to the RPC endpoint such as `'https://localhost:8545'`. For RPC servers behind HTTP connections running on port 80 and HTTPS connections running on port 443 the port can be omitted from the URI.
- `request_kwargs` should be a dictionary of keyword arguments which will be passed onto each http/https POST request made to your node.
- `session` allows you to pass a `requests.Session` object initialized as desired.

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545"))
```

Note that you should create only one HTTPProvider with the same provider URL per python process, as the HTTPProvider recycles underlying TCP/IP network connections, for better performance. Multiple HTTP-Providers with different URLs will work as expected.

Under the hood, the `HTTPProvider` uses the python requests library for making requests. If you would like to modify how requests are made, you can use the `request_kwargs` to do so. A common use case for this is increasing the timeout for each request.

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545", request_kwargs={'timeout':␣
→60}))
```

To tune the connection pool size, you can pass your own `requests.Session`.

```
>>> from web3 import Web3
>>> adapter = requests.adapters.HTTPAdapter(pool_connections=20, pool_maxsize=20)
>>> session = requests.Session()
>>> session.mount('http://', adapter)
>>> session.mount('https://', adapter)
>>> w3 = Web3(Web3.HTTPProvider("http://127.0.0.1:8545", session=session))
```

IPCProvider

class web3.providers.ipc.**IPCProvider**(*ipc_path=None*, *testnet=False*, *timeout=10*)

This provider handles interaction with an IPC Socket based JSON-RPC server.

- `ipc_path` is the filesystem path to the IPC socket:

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.IPCProvider("~/Library/Ethereum/geth.ipc"))
```

If no `ipc_path` is specified, it will use a default depending on your operating system.

- On Linux and FreeBSD: `~/.ethereum/geth.ipc`
- On Mac OS: `~/Library/Ethereum/geth.ipc`
- On Windows: `\\.\pipe\geth.ipc`

WebsocketProvider

Note: WebsocketProviderV2 is currently in beta and our goal is to fully replace WebsocketProvider with WebsocketProviderV2 in the next major release of web3.py.

class web3.providers.websocket.**WebsocketProvider**(*endpoint_uri*[, *websocket_timeout*, *websocket_kwargs*])

This provider handles interactions with an WS or WSS based JSON-RPC server.

- `endpoint_uri` should be the full URI to the RPC endpoint such as `'ws://localhost:8546'`.
- `websocket_timeout` is the timeout in seconds, used when receiving or sending data over the connection. Defaults to 10.
- `websocket_kwargs` this should be a dictionary of keyword arguments which will be passed onto the ws/wss websocket connection.

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.WebsocketProvider("ws://127.0.0.1:8546"))
```

Under the hood, the `WebsocketProvider` uses the python websockets library for making requests. If you would like to modify how requests are made, you can use the `websocket_kwargs` to do so. See the websockets documentation for available arguments.

Unlike HTTP connections, the timeout for WS connections is controlled by a separate `websocket_timeout` argument, as shown below.

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.WebsocketProvider("ws://127.0.0.1:8546", websocket_timeout=60))
```

Persistent Connection Providers

class web3.providers.persistent.**PersistentConnectionProvider**(*endpoint_uri: str*, *request_timeout: float = 50.0*, *subscription_response_queue_size: int = 500*, *request_information_cache_size: int = 500*)

This is a base provider class, currently inherited by the `WebsocketProviderV2`. It handles interactions with a persistent connection to a JSON-RPC server. Among its configuration, it houses all of the `RequestProcessor` logic for handling the asynchronous sending and receiving of requests and responses. See the *Request Processing for Persistent Connection Providers* section for more details on the internals of persistent connection providers.

- `request_timeout` is the timeout in seconds, used when sending data over the connection and waiting for a response to be received from the listener task. Defaults to `50.0`.
- `subscription_response_queue_size` is the size of the queue used to store subscription responses, defaults to `500`. While messages are being consumed, this queue should never fill up as it is a transient queue and meant to handle asynchronous receiving and processing of responses. When in sync with the websocket stream, this queue should only ever store 1 to a few messages at a time.
- `request_information_cache_size` is the size of the cache used to store request information so that when a response is received, the provider knows how to process it based on the original request. Defaults to `500`.

WebsocketProviderV2 (beta)

> **Warning:** This provider is still in beta. However, it is being actively developed and supported and is expected to be stable in the next major version of *web3.py* (v7).

class web3.providers.websocket.**WebsocketProviderV2**(*endpoint_uri: str, websocket_kwargs: Dict[str, Any] = {}, silence_listener_task_exceptions: bool = False*)

This provider handles interactions with an WS or WSS based JSON-RPC server.

- `endpoint_uri` should be the full URI to the RPC endpoint such as `'ws://localhost:8546'`.
- `websocket_kwargs` this should be a dictionary of keyword arguments which will be passed onto the ws/wss websocket connection.
- `silence_listener_task_exceptions` is a boolean that determines whether exceptions raised by the listener task are silenced. Defaults to `False`, raising any exceptions that occur in the listener task.

This provider inherits from the *PersistentConnectionProvider* class. Refer to the *PersistentConnectionProvider* documentation for details on additional configuration options available for this provider.

Under the hood, the `WebsocketProviderV2` uses the python websockets library for making requests. If you would like to modify how requests are made, you can use the `websocket_kwargs` to do so. See the websockets documentation for available arguments.

Usage

The `AsyncWeb3` class may be used as a context manager, utilizing the `async with` syntax, when connecting via `persistent_websocket()` using the `WebsocketProviderV2`. This will automatically close the connection when the context manager exits and is the recommended way to initiate a persistent connection to the websocket provider. A similar example, using the `websockets` connection as an asynchronous context manager, can be found in the websockets connection docs.

```
>>> import asyncio
>>> from web3 import AsyncWeb3
>>> from web3.providers import WebsocketProviderV2

>>> LOG = True  # toggle debug logging
>>> if LOG:
...     import logging
...     logger = logging.getLogger("web3.providers.WebsocketProviderV2")
...     logger.setLevel(logging.DEBUG)
...     logger.addHandler(logging.StreamHandler())

>>> async def ws_v2_subscription_context_manager_example():
...     async with AsyncWeb3.persistent_websocket(
...         WebsocketProviderV2(f"ws://127.0.0.1:8546")
...     ) as w3:
...         # subscribe to new block headers
...         subscription_id = await w3.eth.subscribe("newHeads")
...
```

(continues on next page)

(continued from previous page)

```
...         async for response in w3.ws.process_subscriptions():
...             print(f"{response}\n")
...             # handle responses here
...
...             if some_condition:
...                 # unsubscribe from new block headers and break out of
...                 # iterator
...                 await w3.eth.unsubscribe(subscription_id)
...                 break
...
...         # still an open connection, make any other requests and get
...         # responses via send / receive
...         latest_block = await w3.eth.get_block("latest")
...         print(f"Latest block: {latest_block}")
...
...         # the connection closes automatically when exiting the context
...         # manager (the `async with` block)

>>> asyncio.run(ws_v2_subscription_context_manager_example())
```

The AsyncWeb3 class may also be used as an asynchronous iterator, utilizing the `async for` syntax, when connecting via `persistent_websocket()` using the `WebsocketProviderV2`. This may be used to set up an indefinite websocket connection and reconnect automatically if the connection is lost. A similar example, using the `websockets` connection as an asynchronous iterator, can be found in the websockets connection docs.

```
>>> import asyncio
>>> from web3 import AsyncWeb3
>>> from web3.providers import WebsocketProviderV2
>>> import websockets

>>> async def ws_v2_subscription_iterator_example():
...     async for w3 in AsyncWeb3.persistent_websocket(
...         WebsocketProviderV2(f"ws://127.0.0.1:8546")
...     ):
...         try:
...             ...
...         except websockets.ConnectionClosed:
...             continue

# run the example
>>> asyncio.run(ws_v2_subscription_iterator_example())
```

If neither of the two init patterns above work for your application, the `__await__()` method is defined on the `persistent_websocket()` connection in a manner that awaits connecting to the websocket. You may also choose to instantiate and connect via the provider in separate lines. Both of these examples are shown below.

```
>>> async def ws_v2_alternate_init_example_1():
...     # awaiting the persistent connection itself will connect to the websocket
...     w3 = await AsyncWeb3.persistent_websocket(WebsocketProviderV2(f"ws://127.0.0.
↪1:8546"))
...
...     # some code here
```

(continues on next page)

(continued from previous page)

```
...
...     # manual cleanup
...     await w3.provider.disconnect()

# run the example
>>> asyncio.run(ws_v2_alternate_init_example_1)

>>> async def ws_v2_alternate_init_example_2():
...     # instantiation and connection via the provider as separate lines
...     w3 = AsyncWeb3.persistent_websocket(WebsocketProviderV2(f"ws://127.0.0.1:8546"))
...     await w3.provider.connect()
...
...     # some code here
...
...     # manual cleanup
...     await w3.provider.disconnect()

# run the example
>>> asyncio.run(ws_v2_alternate_init_example_2)
```

The `WebsocketProviderV2` class uses the *RequestProcessor* class under the hood to sync up the receiving of responses and response processing for one-to-one and one-to-many request-to-response requests. Refer to the *RequestProcessor* documentation for details.

_PersistentConnectionWeb3 via AsyncWeb3.persistent_websocket()

When an `AsyncWeb3` class is connected to a persistent websocket connection, via the `persistent_websocket()` method, it becomes an instance of the `_PersistentConnectionWeb3` class. This class has a few additional methods and attributes that are not available on the `AsyncWeb3` class.

class `web3.main.`**`_PersistentConnectionWeb3`**

`ws`

The public API for interacting with the websocket connection is available via the `ws` attribute of the `_PersistentConnectionWeb3` class. This attribute is an instance of the *WebsocketConnection* class and is the main interface for interacting with the websocket connection.

Interacting with the Websocket Connection

class `web3.providers.websocket.`**`WebsocketConnection`**

This class handles interactions with a websocket connection. It is available via the `ws` attribute of the `_PersistentConnectionWeb3` class. The `WebsocketConnection` class has the following methods and attributes:

`subscriptions`

This attribute returns the current active subscriptions as a dict mapping the subscription `id` to a dict of metadata about the subscription request.

`process_subscriptions()`

This method is available for listening to websocket subscriptions indefinitely. It is an asynchronous iterator that yields strictly one-to-many (e.g. `eth_subscription` responses) request-to-response messages from the websocket connection. To receive responses for one-to-one request-to-response calls, use the standard API for making requests via the appropriate module (e.g. `block_num = await w3.eth.block_number`)

The responses from this method are formatted by web3.py formatters and run through the middlewares that were present at the time of subscription. An example of its use can be seen above in the *Usage* section.

recv()

The `recv()` method can be used to receive the next message from the websocket. The response from this method is formatted by web3.py formatters and run through the middlewares before being returned. This is not the recommended way to receive a message as the `process_subscriptions()` method is available for listening to websocket subscriptions and the standard API for making requests via the appropriate module (e.g. `block_num = await w3.eth.block_number`) is available for receiving responses for one-to-one request-to-response calls.

send(*method: RPCEndpoint*, *params: Sequence[Any]*)

This method is available strictly for sending raw requests to the websocket, if desired. It is not recommended to use this method directly, as the responses will not be formatted by web3.py formatters or run through the middlewares. Instead, use the methods available on the respective web3 module. For example, use `w3.eth.get_block("latest")` instead of `w3.ws.send("eth_getBlockByNumber", ["latest", True])`.

AutoProvider

`AutoProvider` is the default used when initializing *web3.Web3* without any providers. There's rarely a reason to use it explicitly.

AsyncHTTPProvider

class web3.providers.async_rpc.**AsyncHTTPProvider**(*endpoint_uri*[, *request_kwargs*])

This provider handles interactions with an HTTP or HTTPS based JSON-RPC server asynchronously.

- `endpoint_uri` should be the full URI to the RPC endpoint such as `'https://localhost:8545'`. For RPC servers behind HTTP connections running on port 80 and HTTPS connections running on port 443 the port can be omitted from the URI.
- `request_kwargs` should be a dictionary of keyword arguments which will be passed onto each http/https POST request made to your node.
- the `cache_async_session()` method allows you to use your own `aiohttp.ClientSession` object. This is an async method and not part of the constructor

```
>>> from aiohttp import ClientSession
>>> from web3 import AsyncWeb3, AsyncHTTPProvider

>>> w3 = AsyncWeb3(AsyncHTTPProvider(endpoint_uri))

>>> # If you want to pass in your own session:
>>> custom_session = ClientSession()
>>> await w3.provider.cache_async_session(custom_session) # This method is an async␣
↪method so it needs to be handled accordingly
```

Under the hood, the `AsyncHTTPProvider` uses the python aiohttp library for making requests.

- **Available Middleware** - These middlewares have async versions available:
  - *Attribute Dict Middleware*

EthereumTesterProvider

> **Warning:** Experimental: This provider is experimental. There are still significant gaps in functionality. However it is being actively developed and supported.

class web3.providers.eth_tester.**EthereumTesterProvider**(*eth_tester=None*)

This provider integrates with the `eth-tester` library. The `eth_tester` constructor argument should be an instance of the `EthereumTester` or a subclass of `BaseChainBackend` class provided by the `eth-tester` library. If you would like a custom eth-tester instance to test with, see the `eth-tester` library documentation for details.

```
>>> from web3 import Web3, EthereumTesterProvider
>>> w3 = Web3(EthereumTesterProvider())
```

Note: To install the needed dependencies to use EthereumTesterProvider, you can install the pip extras package that has the correct interoperable versions of the `eth-tester` and `py-evm` dependencies needed to do testing: e.g. `pip install web3[tester]`

2.6 Working with Local Private Keys

2.6.1 Local vs Hosted Nodes

Hosted Node

A hosted node is controlled by someone else. When you connect to Infura, you are connected to a hosted node. See ethereumnodes.com for the list of free and commercial node providers.

Local Node

A local node is started and controlled by you on your computer. For several reasons (e.g., privacy, security), this is the recommended path, but it requires more resources and work to set up and maintain.

2.6.2 Local vs Hosted Keys

An Ethereum private key is a 256-bit (32 bytes) random integer. For each private key, you get one Ethereum address, also known as an Externally Owned Account (EOA).

In Python, the private key is expressed as a 32-byte long Python `bytes` object. When a private key is presented to users in a hexadecimal format, it may or may not contain a starting `0x` hexadecimal prefix.

Local Private Key
A local private key is a locally stored secret you import to your Python application. Please read below how you can create and import a local private key and use it to sign transactions.

Hosted Private Key
This is a legacy way to use accounts when working with unit test backends like `web3.providers.eth_tester.main.EthereumTesterProvider` or Anvil. Calling `web3.eth.accounts` gives you a predefined list of accounts that have been funded with test ETH. You can use any of these accounts with use `send_transaction()` without further configuration.

In the past, around 2015, this was also a way to use private keys in a locally hosted node, but this practice is now discouraged.

Note: Methods like *web3.eth.send_transaction*` do not work with modern node providers, because they relied on a node state and all modern nodes are stateless. You must always use local private keys when working with nodes hosted by someone else.

2.6.3 Some Common Uses for Local Private Keys

A very common reason to work with local private keys is to interact with a hosted node.

Some common things you might want to do with a *Local Private Key* are:

- *Sign a Transaction*
- *Sign a Contract Transaction*
- *Sign a Message*
- *Verify a Message*

Using private keys usually involves `w3.eth.account` in one way or another. Read on for more, or see a full list of things you can do in the docs for `eth_account.Account`.

2.6.4 Creating a Private Key

Each Ethereum address has a matching private key. To create a new Ethereum account you can just generate a random number that acts as a private key.

- A private key is just a random unguessable, or cryptographically safe, 256-bit integer number
- A valid private key is > 0 and < max private key value (a number above the elliptic curve order FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141)
- Private keys do not have checksums.

To create a private key using web3.py and command line you can do:

```
python -c "from web3 import Web3; w3 = Web3(); acc = w3.eth.account.create(); print(f
↪'private key={w3.to_hex(acc.key)}, account={acc.address}')"
```

Which outputs a new private key and an account pair:

```
private key=0x480c4aec9fa..., account=0x9202a9d5D2d129CB400a40e00aC822a53ED81167
```

- *Never store private key with your source*. Use environment variables to store the key. Read more below.
- You can also import the raw hex private key to MetaMask and any other wallet - the private key can be shared between your Python code and any number of wallets.

2.6.5 Funding a New Account

If you create a private key, it comes with its own Ethereum address. By default, the balance of this address is zero. Before you can send any transactions with your account, you need to top up.

- For a local test environment, any environment is bootstrapped with accounts that have ETH on them. Move ETH from default accounts to your newly created account.
- For public mainnet, you need to buy ETH in a cryptocurrency exchange
- For a testnet, you need to [use a testnet faucet](https://faucet.paradigm.xyz/)

2.6.6 Reading a Private Key from an Environment Variable

In this example we pass the private key to our Python application in an environment variable. This private key is then added to the transaction signing keychain with `Signing` middleware.

If unfamiliar, note that you can export your private keys from Metamask and other wallets.

Warning:

- **Never** share your private keys.
- **Never** put your private keys in source code.
- **Never** commit private keys to a Git repository.

Example `account_test_script.py`

```
import os
from eth_account import Account
from eth_account.signers.local import LocalAccount
from web3 import Web3, EthereumTesterProvider
from web3.middleware import construct_sign_and_send_raw_middleware

w3 = Web3(EthereumTesterProvider())

private_key = os.environ.get("PRIVATE_KEY")
assert private_key is not None, "You must set PRIVATE_KEY environment variable"
assert private_key.startswith("0x"), "Private key must start with 0x hex prefix"

account: LocalAccount = Account.from_key(private_key)
```

(continues on next page)

(continued from previous page)

```
w3.middleware_onion.add(construct_sign_and_send_raw_middleware(account))

print(f"Your hot wallet address is {account.address}")

# Now you can use web3.eth.send_transaction(), Contract.functions.xxx.transact()␣
↪functions
# with your local private key through middleware and you no longer get the error
# "ValueError: The method eth_sendTransaction does not exist/is not available
```

Example how to run this in UNIX shell:

```
# Generate a new 256-bit random integer using openssl UNIX command that acts as a␣
↪private key.
# You can also do:
# python -c "from web3 import Web3; w3 = Web3(); acc = w3.eth.account.create(); print(f
↪'private key={w3.to_hex(acc.key)}, account={acc.address}')"
# Store this in a safe place, like in your password manager.
export PRIVATE_KEY=0x`openssl rand -hex 32`

# Run our script
python account_test_script.py
```

This will print:

```
Your hot wallet address is 0x27C8F899bb69E1501BBB96d09d7477a2a7518918
```

2.6.7 Extract private key from geth keyfile

Note: The amount of available ram should be greater than 1GB.

```
with open('~/.ethereum/keystore/UTC--...--5ce9454909639D2D17A3F753ce7d93fa0b9aB12E') as␣
↪keyfile:
    encrypted_key = keyfile.read()
    private_key = w3.eth.account.decrypt(encrypted_key, 'correcthorsebatterystaple')
    # tip: do not save the key or password anywhere, especially into a shared source file
```

2.6.8 Sign a Message

Warning: There is no single message format that is broadly adopted with community consensus. Keep an eye on several options, like EIP-683, EIP-712, and EIP-719. Consider the *w3.eth.sign()* approach be deprecated.

For this example, we will use the same message hashing mechanism that is provided by *w3.eth.sign()*.

```
>>> from web3 import Web3, EthereumTesterProvider
>>> from eth_account.messages import encode_defunct
```

(continues on next page)

(continued from previous page)

```
>>> w3 = Web3(EthereumTesterProvider())
>>> msg = "ISF"
>>> private_key = b"\xb2\\}\xb3\x1f\xee\xd9\x12''\xbf\t9\xdcv\x9a\x96VK-\xe4\xc4rm\x03[6\
↪xec\xf1\xe5\xb3d"
>>> message = encode_defunct(text=msg)
>>> signed_message = w3.eth.account.sign_message(message, private_key=private_key)
>>> signed_message
SignedMessage(messageHash=HexBytes(
↪'0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750'),
 r=104389933075820307925104709181714897380569894203213074526835978196648170704563,
 s=28205917190874851400050446352651915501321657673772411533993420917949420456142,
 v=28,
 signature=HexBytes(
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7680c19ebe
↪'))
```

2.6.9 Verify a Message

With the original message text and a signature:

```
>>> message = encode_defunct(text="ISF")
>>> w3.eth.account.recover_message(message, signature=signed_message.signature)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
```

2.6.10 Prepare message for ecrecover in Solidity

Let's say you want a contract to validate a signed message, like if you're making payment channels, and you want to validate the value in Remix or web3.js.

You might have produced the signed_message locally, as in *Sign a Message*. If so, this will prepare it for Solidity:

```
>>> from web3 import Web3

# ecrecover in Solidity expects v as a native uint8, but r and s as left-padded bytes32
# Remix / web3.js expect r and s to be encoded to hex
# This convenience method will do the pad & hex for us:
>>> def to_32byte_hex(val):
...   return Web3.to_hex(Web3.to_bytes(val).rjust(32, b'\0'))

>>> ec_recover_args = (msghash, v, r, s) = (
...   Web3.to_hex(signed_message.messageHash),
...   signed_message.v,
...   to_32byte_hex(signed_message.r),
...   to_32byte_hex(signed_message.s),
... )
>>> ec_recover_args
('0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750',
 28,
 '0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3',
 '0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce')
```

Instead, you might have received a message and a signature encoded to hex. Then this will prepare it for Solidity:

```
>>> from web3 import Web3
>>> from eth_account.messages import encode_defunct, _hash_eip191_message

>>> hex_message = '0x49e299a55346'
>>> hex_signature =
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7680c19ebeb
↪'

# ecrecover in Solidity expects an encoded version of the message

# - encode the message
>>> message = encode_defunct(hexstr=hex_message)

# - hash the message explicitly
>>> message_hash = _hash_eip191_message(message)

# Remix / web3.js expect the message hash to be encoded to a hex string
>>> hex_message_hash = Web3.to_hex(message_hash)

# ecrecover in Solidity expects the signature to be split into v as a uint8,
#   and r, s as a bytes32
# Remix / web3.js expect r and s to be encoded to hex
>>> sig = Web3.to_bytes(hexstr=hex_signature)
>>> v, hex_r, hex_s = Web3.to_int(sig[-1]), Web3.to_hex(sig[:32]), Web3.to_
↪hex(sig[32:64])

# ecrecover in Solidity takes the arguments in order = (msghash, v, r, s)
>>> ec_recover_args = (hex_message_hash, v, hex_r, hex_s)
>>> ec_recover_args
('0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750',
 28,
 '0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3',
 '0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce')
```

2.6.11 Verify a message with ecrecover in Solidity

Create a simple ecrecover contract in Remix:

```
pragma solidity ^0.4.19;

contract Recover {
  function ecr (bytes32 msgh, uint8 v, bytes32 r, bytes32 s) public pure
  returns (address sender) {
    return ecrecover(msgh, v, r, s);
  }
}
```

Then call `ecr` with these arguments from *Prepare message for ecrecover in Solidity* in Remix, `"0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750", 28, "0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3", "0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce"`

The message is verified, because we get the correct sender of the message back in response: `0x5ce9454909639d2d17a3f753ce7d93fa0b9ab12e`.

2.6.12 Sign a Transaction

Create a transaction, sign it locally, and then send it to your node for broadcasting, with *send_raw_transaction()*.

```
>>> transaction = {
...     'to': '0xF0109fC8DF283027b6285cc889F5aA624EaC1F55',
...     'value': 1000000000,
...     'gas': 2000000,
...     'maxFeePerGas': 2000000000,
...     'maxPriorityFeePerGas': 1000000000,
...     'nonce': 0,
...     'chainId': 1,
...     'type': '0x2',  # the type is optional and, if omitted, will be interpreted␣
↪based on the provided transaction parameters
...     'accessList': (  # accessList is optional for dynamic fee transactions
...         {
...             'address': '0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae',
...             'storageKeys': (
...                 '0x0000000000000000000000000000000000000000000000000000000000000003',
...                 '0x0000000000000000000000000000000000000000000000000000000000000007',
...             )
...         },
...         {
...             'address': '0xbb9bc244d798123fde783fcc1c72d3bb8c189413',
...             'storageKeys': ()
...         },
...     )
... }
>>> key = '0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318'
>>> signed = w3.eth.account.sign_transaction(transaction, key)
>>> signed.rawTransaction
HexBytes(
↪'0x02f8e20180843b9aca008477359400831e848094f0109fc8df283027b6285cc889f5aa624eac1f55843b9aca0080f872f8
↪')
>>> signed.hash
HexBytes('0xe85ce7efa52c16cb5c469c7bde54fbd4911639fdfde08003f65525a85076d915')
>>> signed.r
84095564551732371065849105252408326384410939276686534847013731510862163857293
>>> signed.s
32698347985257114675470251181312399332782188326270244072370350491677872459742
>>> signed.v
1

# When you run send_raw_transaction, you get back the hash of the transaction:
>>> w3.eth.send_raw_transaction(signed.rawTransaction)
'0xe85ce7efa52c16cb5c469c7bde54fbd4911639fdfde08003f65525a85076d915'
```

2.6.13 Sign a Contract Transaction

To sign a transaction locally that will invoke a smart contract:

1. Initialize your `Contract` object
2. Build the transaction
3. Sign the transaction, with `w3.eth.account.sign_transaction()`
4. Broadcast the transaction with `send_raw_transaction()`

```
# When running locally, execute the statements found in the file linked below to load␣
↪the EIP20_ABI variable.
# See: https://github.com/carver/ethtoken.py/blob/v0.0.1-alpha.4/ethtoken/abi.py

>>> from web3 import Web3, EthereumTesterProvider
>>> w3 = Web3(EthereumTesterProvider())

>>> unicorns = w3.eth.contract(address="0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359",␣
↪abi=EIP20_ABI)

>>> nonce = w3.eth.get_transaction_count('0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E')

# Build a transaction that invokes this contract's function, called transfer
>>> unicorn_txn = unicorns.functions.transfer(
...     '0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359',
...     1,
... ).build_transaction({
...     'chainId': 1,
...     'gas': 70000,
...     'maxFeePerGas': w3.to_wei('2', 'gwei'),
...     'maxPriorityFeePerGas': w3.to_wei('1', 'gwei'),
...     'nonce': nonce,
... })

>>> unicorn_txn
{'value': 0,
 'chainId': 1,
 'gas': 70000,
 'maxFeePerGas': 2000000000,
 'maxPriorityFeePerGas': 1000000000,
 'nonce': 0,
 'to': '0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359',
 'data':
↪'0xa9059cbb000000000000000000000000fb6916095ca1df60bb79ce92ce3ea74c37c5d359000000000000
↪'}

>>> private_key = b"\xb2\\}\xb3\x1f\xee\xd9\x12''\xbf\t9\xdcv\x9a\x96VK-\xe4\xc4rm\x03[6\
↪xec\xf1\xe5\xb3d"
>>> signed_txn = w3.eth.account.sign_transaction(unicorn_txn, private_key=private_key)
>>> signed_txn.hash
HexBytes('0x748db062639a45e519dba934fce09c367c92043867409160c9989673439dc817')
>>> signed_txn.rawTransaction
HexBytes(
```

(continues on next page)

(continued from previous page)

```
↪'0x02f8b00180843b9aca0084773594008301117094fb6916095ca1df60bb79ce92ce3ea74c37c5d35980b844a9059cbb00000
↪')
>>> signed_txn.r
93522894155654168208483453926995743737629589441154283159505514235904280342434
>>> signed_txn.s
48417310681110102814014302147799665717176259465062324746227758019974374282313
>>> signed_txn.v
1

>>> w3.eth.send_raw_transaction(signed_txn.rawTransaction)

# When you run send_raw_transaction, you get the same result as the hash of the␣
↪transaction:
>>> w3.to_hex(w3.keccak(signed_txn.rawTransaction))
'0x748db062639a45e519dba934fce09c367c92043867409160c9989673439dc817'
```

2.7 Sending Transactions

Note: Prefer to view this code in a Jupyter Notebook? View the repo here.

There are two methods for sending transactions using web3.py: *send_transaction()* and *send_raw_transaction()*. A brief guide:

1. Want to sign a transaction offline or send pre-signed transactions?
   - use `sign_transaction` + *send_raw_transaction()*
2. Are you primarily using the same account for all transactions and would you prefer to save a few lines of code?
   - configure *construct_sign_and_send_raw_middleware()*, then
   - use *send_transaction()*
3. Otherwise:
   - load account via eth-account (`w3.eth.account.from_key(pk)`), then
   - use *send_transaction()*

Interacting with or deploying a contract?

- Option 1: *transact()* uses *send_transaction()* under the hood
- Option 2: *build_transaction()* + `sign_transaction` + *send_raw_transaction()*

An example for each can be found below.

2.7.1 Chapter 0: `w3.eth.send_transaction` with `eth-tester`

Many tutorials use `eth-tester` (via EthereumTesterProvider) for convenience and speed of conveying ideas/building a proof of concept. Transactions sent by test accounts are auto-signed.

```
from web3 import Web3, EthereumTesterProvider

w3 = Web3(EthereumTesterProvider())

# eth-tester populates accounts with test ether:
acct1 = w3.eth.accounts[0]

some_address = "0x0000000000000000000000000000000000000000"

# when using one of its generated test accounts,
# eth-tester signs the tx (under the hood) before sending:
tx_hash = w3.eth.send_transaction({
    "from": acct1,
    "to": some_address,
    "value": 123123123123123
})

tx = w3.eth.get_transaction(tx_hash)
assert tx["from"] == acct1
```

2.7.2 Chapter 1: `w3.eth.send_transaction` + signer middleware

The *send_transaction()* method is convenient and to-the-point. If you want to continue using the pattern after graduating from `eth-tester`, you can utilize web3.py middleware to sign transactions from a particular account:

```
from web3.middleware import construct_sign_and_send_raw_middleware
import os

# Note: Never commit your key in your code! Use env variables instead:
pk = os.environ.get('PRIVATE_KEY')

# Instantiate an Account object from your key:
acct2 = w3.eth.account.from_key(pk)

# For the sake of this example, fund the new account:
w3.eth.send_transaction({
    "from": acct1,
    "value": w3.to_wei(3, 'ether'),
    "to": acct2.address
})

# Add acct2 as auto-signer:
w3.middleware_onion.add(construct_sign_and_send_raw_middleware(acct2))
# pk also works: w3.middleware_onion.add(construct_sign_and_send_raw_middleware(pk))

# Transactions from `acct2` will then be signed, under the hood, in the middleware:
tx_hash = w3.eth.send_transaction({
```

(continues on next page)

(continued from previous page)

```
    "from": acct2.address,
    "value": 3333333333,
    "to": some_address
})

tx = w3.eth.get_transaction(tx_hash)
assert tx["from"] == acct2.address

# Optionally, you can set a default signer as well:
# w3.eth.default_account = acct2.address
# Then, if you omit a "from" key, acct2 will be used.
```

2.7.3 Chapter 2: w3.eth.send_raw_transaction

if you don't opt for the middleware, you'll need to:

- build each transaction,
- sign_transaction, and
- then use *send_raw_transaction()*.

```
# 1. Build a new tx
transaction = {
    'from': acct2.address,
    'to': some_address,
    'value': 1000000000,
    'nonce': w3.eth.get_transaction_count(acct2.address),
    'gas': 200000,
    'maxFeePerGas': 2000000000,
    'maxPriorityFeePerGas': 1000000000,
}

# 2. Sign tx with a private key
signed = w3.eth.account.sign_transaction(transaction, pk)

# 3. Send the signed transaction
tx_hash = w3.eth.send_raw_transaction(signed.rawTransaction)
tx = w3.eth.get_transaction(tx_hash)
assert tx["from"] == acct2.address
```

2.7.4 Chapter 3: Contract transactions

The same concepts apply for contract interactions, at least under the hood.

Executing a function on a smart contract requires sending a transaction, which is typically done in one of two ways:

- executing the *transact()* function, or
- *build_transaction()*, then signing and sending the raw transaction.

```
#########################################
#### SMOL CONTRACT FOR THIS EXAMPLE: ####
#########################################
# // SPDX-License-Identifier: MIT
# pragma solidity 0.8.17;
#
# contract Billboard {
#     string public message;
#
#     constructor(string memory _message) {
#         message = _message;
#     }
#
#     function writeBillboard(string memory _message) public {
#         message = _message;
#     }
# }

# After compiling the contract, initialize the contract factory:
init_bytecode = "6080604052348015620000115760008 0fd5b5060..."
abi = '[{"inputs": [{"internalType": "string","name": "_message",...'
Billboard = w3.eth.contract(bytecode=init_bytecode, abi=abi)

# Deploy a contract using `transact` + the signer middleware:
tx_hash = Billboard.constructor("gm").transact({"from": acct2.address})
receipt = w3.eth.get_transaction_receipt(tx_hash)
deployed_addr = receipt["contractAddress"]

# Reference the deployed contract:
billboard = w3.eth.contract(address=deployed_addr, abi=abi)

# Manually build and sign a transaction:
unsent_billboard_tx = billboard.functions.writeBillboard("gn").build_transaction({
    "from": acct2.address,
    "nonce": w3.eth.get_transaction_count(acct2.address),
})
signed_tx = w3.eth.account.sign_transaction(unsent_billboard_tx, private_key=acct2.key)

# Send the raw transaction:
assert billboard.functions.message().call() == "gm"
tx_hash = w3.eth.send_raw_transaction(signed_tx.rawTransaction)
w3.eth.wait_for_transaction_receipt(tx_hash)
assert billboard.functions.message().call() == "gn"
```

2.8 Monitoring Events

If you're on this page, you're likely looking for an answer to this question: **How do I know when a specific contract is used?** You have at least three options:

1. Query blocks for transactions that include the contract address in the "to" field. This contrived example is searching the latest block for any transactions sent to the WETH contract.

```
WETH_ADDRESS = '0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2'

block = w3.eth.get_block('latest')
for tx_hash in block.transactions:
    tx = w3.eth.get_transaction(tx_hash)
    if tx['to'] == WETH_ADDRESS:
        print(f'Found interaction with WETH contract! {tx}')
```

2. Query for logs emitted by a contract. After instantiating a web3.py Contract object, you can *fetch logs* for any event listed in the ABI. In this example, we query for `Transfer` events in the latest block and log out the results.

```
WETH_ADDRESS = '0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2'
WETH_ABI = '[{"constant":true,"inputs":[],"name":"name","outputs":[{"name":"","type":
↪"string"}],"payable":false,"stateMutability":"view","type":"function"},{"constant
↪":false,"inputs":[{"name":"guy","type":"address"},{"name":"wad","type":"uint256"}],
↪"name":"approve","outputs":[{"name":"","type":"bool"}],"payable":false,"stateMutability
↪":"nonpayable","type":"function"},{"constant":true,"inputs":[],"name":"totalSupply",
↪"outputs":[{"name":"","type":"uint256"}],"payable":false,"stateMutability":"view","type
↪":"function"},{"constant":false,"inputs":[{"name":"src","type":"address"},{"name":"dst
↪","type":"address"},{"name":"wad","type":"uint256"}],"name":"transferFrom","outputs":[{
↪"name":"","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":false,"inputs":[{"name":"wad","type":"uint256"}],"name":
↪"withdraw","outputs":[],"payable":false,"stateMutability":"nonpayable","type":"function
↪"},{"constant":true,"inputs":[],"name":"decimals","outputs":[{"name":"","type":"uint8"}
↪],"payable":false,"stateMutability":"view","type":"function"},{"constant":true,"inputs
↪":[{"name":"","type":"address"}],"name":"balanceOf","outputs":[{"name":"","type":
↪"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{"constant
↪":true,"inputs":[],"name":"symbol","outputs":[{"name":"","type":"string"}],"payable
↪":false,"stateMutability":"view","type":"function"},{"constant":false,"inputs":[{"name
↪":"dst","type":"address"},{"name":"wad","type":"uint256"}],"name":"transfer","outputs
↪":[{"name":"","type":"bool"}],"payable":false,"stateMutability":"nonpayable","type":
↪"function"},{"constant":false,"inputs":[],"name":"deposit","outputs":[],"payable":true,
↪"stateMutability":"payable","type":"function"},{"constant":true,"inputs":[{"name":"",
↪"type":"address"},{"name":"","type":"address"}],"name":"allowance","outputs":[{"name":"
↪","type":"uint256"}],"payable":false,"stateMutability":"view","type":"function"},{
↪"payable":true,"stateMutability":"payable","type":"fallback"},{"anonymous":false,
↪"inputs":[{"indexed":true,"name":"src","type":"address"},{"indexed":true,"name":"guy",
↪"type":"address"},{"indexed":false,"name":"wad","type":"uint256"}],"name":"Approval",
↪"type":"event"},{"anonymous":false,"inputs":[{"indexed":true,"name":"src","type":
↪"address"},{"indexed":true,"name":"dst","type":"address"},{"indexed":false,"name":"wad
↪","type":"uint256"}],"name":"Transfer","type":"event"},{"anonymous":false,"inputs":[{
↪"indexed":true,"name":"dst","type":"address"},{"indexed":false,"name":"wad","type":
↪"uint256"}],"name":"Deposit","type":"event"},{"anonymous":false,"inputs":[{"indexed
↪":true,"name":"src","type":"address"},{"indexed":false,"name":"wad","type":"uint256"}],
↪"name":"Withdrawal","type":"event"}]'
```

(continues on next page)

(continued from previous page)

```
weth_contract = w3.eth.contract(address=WETH_ADDRESS, abi=WETH_ABI)

# fetch transfer events in the last block
logs = weth_contract.events.Transfer().get_logs(fromBlock=w3.eth.block_number)

for log in logs:
   print(f"Transfer of {w3.from_wei(log.args.wad, 'ether')} WETH from {log.args.src} to
↪{log.args.dst}")
```

See an advanced example of fetching log history *here*.

3. Use a filter.

> **Warning:** While filters can be a very convenient way to monitor for blocks, transactions, or events, they are notoriously unreliable. Both remote and locally hosted nodes have a reputation for occasionally dropping filters, and some remote node providers don't support filter-related RPC calls at all.

The `web3.eth.Eth.filter()` method can be used to set up filters for:

- Pending Transactions: `w3.eth.filter("pending")`
- New Blocks `w3.eth.filter("latest")`
- Event Logs

  Through the contract instance api:

  ```
  event_filter = mycontract.events.myEvent.create_filter(fromBlock='latest',␣
  ↪argument_filters={'arg1':10})
  ```

  Or built manually by supplying valid filter params:

  ```
  event_filter = w3.eth.filter({"address": contract_address})
  ```

- Attaching to an existing filter

  ```
  existing_filter = w3.eth.filter(filter_id="0x0")
  ```

Note: Creating event filters requires that your Ethereum node has an API support enabled for filters. Note that Infura support for filters does not offer access to *pending* filters. To get event logs on other stateless nodes please see `web3.contract.ContractEvents`.

2.8.1 Filter Class

class web3.utils.filters.**Filter**(*web3*, *filter_id*)

Filter.**filter_id**

The `filter_id` for this filter as returned by the `eth_newFilter` RPC method when this filter was created.

Filter.**get_new_entries**()

Retrieve new entries for this filter.

Logs will be retrieved using the *web3.eth.Eth.get_filter_changes()* which returns only new entries since the last poll.

Filter.**get_all_entries**()

Retrieve all entries for this filter.

Logs will be retrieved using the *web3.eth.Eth.get_filter_logs()* which returns all entries that match the given filter.

Filter.**format_entry**(*entry*)

Hook for subclasses to modify the format of the log entries this filter returns, or passes to its callback functions.

By default this returns the `entry` parameter umodified.

Filter.**is_valid_entry**(*entry*)

Hook for subclasses to add additional programmatic filtering. The default implementation always returns `True`.

2.8.2 Block and Transaction Filter Classes

class web3.utils.filters.**BlockFilter**(...)

`BlockFilter` is a subclass of *Filter*.

You can setup a filter for new blocks using `web3.eth.filter('latest')` which will return a new *BlockFilter* object.

```
new_block_filter = w3.eth.filter('latest')
new_block_filter.get_new_entries()
```

Note: `"safe"` and `"finalized"` block identifiers are not yet supported for `eth_newBlockFilter`.

class web3.utils.filters.**TransactionFilter**(...)

`TransactionFilter` is a subclass of *Filter*.

You can setup a filter for new blocks using `web3.eth.filter('pending')` which will return a new *TransactionFilter* object.

```
new_transaction_filter = w3.eth.filter('pending')
new_transaction_filter.get_new_entries()
```

2.8.3 Event Log Filters

You can set up a filter for event logs using the web3.py contract api: *web3.contract.Contract.events.your_event_name.create_filter()*, which provides some conveniences for creating event log filters. Refer to the following example:

```
event_filter = myContract.events.<event_name>.create_filter(fromBlock="latest", argument_filters={'arg1':10})
event_filter.get_new_entries()
```

See *web3.contract.Contract.events.your_event_name.create_filter()* documentation for more information.

You can set up an event log filter like the one above with `web3.eth.filter` by supplying a dictionary containing the standard filter parameters. Assuming that `arg1` is indexed, the equivalent filter creation would look like:

```
event_signature_hash = web3.keccak(text="eventName(uint32)").hex()
event_filter = web3.eth.filter({
    "address": myContract_address,
    "topics": [event_signature_hash,
"0x000000000000000000000000000000000000000000000000000000000000000a"],
    })
```

The `topics` argument is order-dependent. For non-anonymous events, the first item in the topic list is always the keccack hash of the event signature. Subsequent topic items are the hex encoded values for indexed event arguments. In the above example, the second item is the `arg1` value `10` encoded to its hex string representation.

In addition to being order-dependent, there are a few more points to recognize when specifying topic filters:

Given a transaction log with topics [A, B], the following topic filters will yield a match:

- [] "anything"
- [A] "A in first position (and anything after)"
- [None, B] "anything in first position AND B in second position (and anything after)"
- [A, B] "A in first position AND B in second position (and anything after)"
- [[A, B], [A, B]] "(A OR B) in first position AND (A OR B) in second position (and anything after)"

See the JSON-RPC documentation for eth_newFilter more information on the standard filter parameters.

> **Note:** Though `"finalized"` and `"safe"` block identifiers are not yet part of the specifications for `eth_newFilter`, they are supported by web3.py and may or may not yield expected results depending on the node being accessed.

Creating a log filter by either of the above methods will return a *LogFilter* instance.

class web3.utils.filters.**LogFilter**(*web3*, *filter_id*, *log_entry_formatter=None*, *data_filter_set=None*)

The *LogFilter* class is a subclass of *Filter*. See the *Filter* documentation for inherited methods.

LogFilter provides the following additional methods:

LogFilter.**set_data_filters**(*data_filter_set*)

Provides a means to filter on the log data, in other words the ability to filter on values from un-indexed event arguments. The parameter `data_filter_set` should be a list or set of 32-byte hex encoded values.

2.8.4 Examples: Listening For Events

Synchronous

```
from web3 import Web3, IPCProvider
import time

# instantiate Web3 instance
w3 = Web3(IPCProvider(...))

def handle_event(event):
    print(event)

def log_loop(event_filter, poll_interval):
    while True:
        for event in event_filter.get_new_entries():
            handle_event(event)
        time.sleep(poll_interval)

def main():
    block_filter = w3.eth.filter('latest')
    log_loop(block_filter, 2)

if __name__ == '__main__':
    main()
```

Asynchronous Filter Polling

Starting with web3 version 4, the `watch` method was taken out of the web3 filter objects. There are many decisions to be made when designing a system regarding threading and concurrency. Rather than force a decision, web3 leaves these choices up to the user. Below are some example implementations of asynchronous filter-event handling that can serve as starting points.

Single threaded concurrency with `async` and `await`

Beginning in python 3.5, the `async` and `await` built-in keywords were added. These provide a shared api for coroutines that can be utilized by modules such as the built-in asyncio. Below is an example event loop using asyncio, that polls multiple web3 filter object, and passes new entries to a handler.

```
from web3 import Web3, IPCProvider
import asyncio

# instantiate Web3 instance
w3 = Web3(IPCProvider(...))

def handle_event(event):
    print(event)
    # and whatever

async def log_loop(event_filter, poll_interval):
    while True:
```

(continues on next page)

(continued from previous page)

```
        for event in event_filter.get_new_entries():
            handle_event(event)
        await asyncio.sleep(poll_interval)

def main():
    block_filter = w3.eth.filter('latest')
    tx_filter = w3.eth.filter('pending')
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(
            asyncio.gather(
                log_loop(block_filter, 2),
                log_loop(tx_filter, 2)))
    finally:
        loop.close()

if __name__ == '__main__':
    main()
```

Read the asyncio documentation for more information.

Running the event loop in a separate thread

Here is an extended version of above example, where the event loop is run in a separate thread, releasing the `main` function for other tasks.

```
from web3 import Web3, IPCProvider
from threading import Thread
import time

# instantiate Web3 instance
w3 = Web3(IPCProvider(...))

def handle_event(event):
    print(event)
    # and whatever


def log_loop(event_filter, poll_interval):
    while True:
        for event in event_filter.get_new_entries():
            handle_event(event)
        time.sleep(poll_interval)


def main():
    block_filter = w3.eth.filter('latest')
    worker = Thread(target=log_loop, args=(block_filter, 5), daemon=True)
    worker.start()
        # .. do some other stuff

```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    main()
```

Here are some other libraries that provide frameworks for writing asynchronous python:

- gevent
- twisted
- celery

2.9 Contracts

Smart contracts are programs deployed to the Ethereum network. See the ethereum.org docs for a proper introduction.

2.9.1 Contract Deployment Example

To run this example, you will need to install a few extra features:

- The sandbox node provided by eth-tester. You can install it with:

```
$ pip install -U "web3[tester]"
```

- `py-solc-x`. This is the supported route to installing the solidity compiler `solc`. You can install it with:

```
$ pip install py-solc-x
```

After `py-solc-x` is installed, you will need to install a version of `solc`. You can install the latest version via a new REPL with:

```
>>> from solcx import install_solc
>>> install_solc(version='latest')
```

You should now be set up to compile and deploy a contract.

The following example runs through these steps: #. Compile Solidity contract into bytecode and an ABI #. Initialize a Contract Web3.py instance #. Deploy the contract using the Contract instance to initiate a transaction #. Interact with the contract functions using the Contract instance

```
>>> from web3 import Web3
>>> from solcx import compile_source

# Solidity source code
>>> compiled_sol = compile_source(
...     '''
...     pragma solidity >0.5.0;
...
...     contract Greeter {
...         string public greeting;
...
...         constructor() public {
...             greeting = 'Hello';
...         }
```

(continues on next page)

(continued from previous page)

```
...
...         function setGreeting(string memory _greeting) public {
...             greeting = _greeting;
...         }
...
...         function greet() view public returns (string memory) {
...             return greeting;
...         }
...     }
...     ''',
...     output_values=['abi', 'bin']
... )

# retrieve the contract interface
>>> contract_id, contract_interface = compiled_sol.popitem()

# get bytecode / bin
>>> bytecode = contract_interface['bin']

# get abi
>>> abi = contract_interface['abi']

# web3.py instance
>>> w3 = Web3(Web3.EthereumTesterProvider())

# set pre-funded account as sender
>>> w3.eth.default_account = w3.eth.accounts[0]

>>> Greeter = w3.eth.contract(abi=abi, bytecode=bytecode)

# Submit the transaction that deploys the contract
>>> tx_hash = Greeter.constructor().transact()

# Wait for the transaction to be mined, and get the transaction receipt
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)

>>> greeter = w3.eth.contract(
...     address=tx_receipt.contractAddress,
...     abi=abi
... )

>>> greeter.functions.greet().call()
'Hello'

>>> tx_hash = greeter.functions.setGreeting('Nihao').transact()
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> greeter.functions.greet().call()
'Nihao'
```

2.9.2 Contract Factories

These factories are not intended to be initialized directly. Instead, create contract objects using the *w3.eth.contract()* method. By default, the contract factory is *Contract*.

class web3.contract.**Contract**(*address*)

Contract provides a default interface for deploying and interacting with Ethereum smart contracts.

The address parameter can be a hex address or an ENS name, like `mycontract.eth`.

2.9.3 Properties

Each Contract Factory exposes the following properties.

Contract.**address**

The hexadecimal encoded 20-byte address of the contract, or an ENS name. May be `None` if not provided during factory creation.

Contract.**abi**

The contract `abi`, or Application Binary Interface, specifies how a contract can be interacted with. Without an `abi`, the contract cannot be decoded. The `abi` enables the Contract instance to expose functions and events as object properties.

For further details, see the Solidity ABI specification.

Contract.**bytecode**

The contract bytecode string. May be `None` if not provided during factory creation.

Contract.**bytecode_runtime**

The runtime part of the contract bytecode string. May be `None` if not provided during factory creation.

Contract.**decode_tuples**

If a Tuple/Struct is returned by a contract function, this flag defines whether to apply the field names from the ABI to the returned data. If False, the returned value will be a normal Python `Tuple`. If True, the returned value will be a Python `NamedTuple` of the class `ABIDecodedNamedTuple`.

NamedTuples have some restrictions regarding field names. web3.py sets `NamedTuple`'s `rename=True`, so disallowed field names may be different than expected. See the Python docs for more information.

Defaults to `False` if not provided during factory creation.

Contract.**functions**

This provides access to contract functions as attributes. For example: `myContract.functions.MyMethod()`. The exposed contract functions are classes of the type *ContractFunction*.

Contract.**events**

This provides access to contract events as attributes. For example: `myContract.events.MyEvent()`. The exposed contract events are classes of the type `ContractEvent`.

2.9.4 Methods

Each Contract Factory exposes the following methods.

classmethod `Contract.constructor`(**args*, ***kwargs).transact(transaction=None)*

Construct and deploy a contract by sending a new public transaction.

If provided `transaction` should be a dictionary conforming to the `web3.eth.send_transaction(transaction)` method. This value may not contain the keys `data` or `to`.

If the contract takes constructor parameters they should be provided as positional arguments or keyword arguments.

If any of the arguments specified in the ABI are an `address` type, they will accept ENS names.

If a `gas` value is not provided, then the `gas` value for the deployment transaction will be created using the `web3.eth.estimate_gas()` method.

Returns the transaction hash for the deploy transaction.

```
>>> deploy_txn = token_contract.constructor(web3.eth.coinbase, 12345).transact()
>>> txn_receipt = web3.eth.get_transaction_receipt(deploy_txn)
>>> txn_receipt['contractAddress']
'0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318'
```

classmethod `Contract.constructor`(**args*, ***kwargs).estimate_gas(transaction=None, block_identifier=None)*

Estimate gas for constructing and deploying the contract.

This method behaves the same as the `Contract.constructor(*args, **kwargs).transact()` method, with transaction details being passed into the end portion of the function call, and function arguments being passed into the first portion.

The `block_identifier` parameter is passed directly to the call at the end portion of the function call.

Returns the amount of gas consumed which can be used as a gas estimate for executing this transaction publicly.

Returns the gas needed to deploy the contract.

```
>>> token_contract.constructor(web3.eth.coinbase, 12345).estimate_gas()
12563
```

classmethod `Contract.constructor`(**args*, ***kwargs).build_transaction(transaction=None)*

Construct the contract deploy transaction bytecode data.

If the contract takes constructor parameters they should be provided as positional arguments or keyword arguments.

If any of the `args` specified in the ABI are an `address` type, they will accept ENS names.

Returns the transaction dictionary that you can pass to send_transaction method.

```
>>> transaction = {
'gasPrice': w3.eth.gas_price,
'chainId': None
}
>>> contract_data = token_contract.constructor(web3.eth.coinbase, 12345).build_
↪transaction(transaction)
>>> web3.eth.send_transaction(contract_data)
```

classmethod `Contract.events.your_event_name.create_filter`(*fromBlock=None*, *toBlock='latest'*, *argument_filters={}*, *topics=[]*)

Creates a new event filter, an instance of `web3.utils.filters.LogFilter`.

- `fromBlock` is a mandatory field. Defines the starting block (exclusive) filter block range. It can be either the starting block number, or 'latest' for the last mined block, or 'pending' for unmined transactions. In the case of `fromBlock`, 'latest' and 'pending' set the 'latest' or 'pending' block as a static value for the starting filter block.
- `toBlock` optional. Defaults to 'latest'. Defines the ending block (inclusive) in the filter block range. Special values 'latest' and 'pending' set a dynamic range that always includes the 'latest' or 'pending' blocks for the filter's upper block range.
- `address` optional. Defaults to the contract address. The filter matches the event logs emanating from `address`.
- `argument_filters`, optional. Expects a dictionary of argument names and values. When provided event logs are filtered for the event argument values. Event arguments can be both indexed or unindexed. Indexed values will be translated to their corresponding topic arguments. Unindexed arguments will be filtered using a regular expression.
- `topics` optional, accepts the standard JSON-RPC topics argument. See the JSON-RPC documentation for eth_newFilter more information on the `topics` parameters.

classmethod `Contract.events.your_event_name.build_filter`()

Creates a EventFilterBuilder instance with the event abi, and the contract address if called from a deployed contract instance. The EventFilterBuilder provides a convenient way to construct the filter parameters with value checking against the event abi. It allows for defining multiple match values or of single values through the match_any and match_single methods.

```
filter_builder = myContract.events.myEvent.build_filter()
filter_builder.fromBlock = "latest"
filter_builder.args.clientID.match_any(1, 2, 3, 4)
filter_builder.args.region.match_single("UK")
filter_instance = filter_builder.deploy()
```

The `deploy` method returns a `web3.utils.filters.LogFilter` instance from the filter parameters generated by the filter builder. Defining multiple match values for array arguments can be accomplished easily with the filter builder:

```
filter_builder = myContract.events.myEvent.build_filter()
filter_builder.args.clientGroups.match_any((1, 3, 5,), (2, 3, 5), (1, 2, 3))
```

The filter builder blocks already defined filter parameters from being changed.

```
filter_builder = myContract.events.myEvent.build_filter()
filter_builder.fromBlock = "latest"
filter_builder.fromBlock = 0  # raises a ValueError
```

classmethod `Contract.encode_abi`(*fn_name*, *args=None*, *kwargs=None*, *data=None*)

Encodes the arguments using the Ethereum ABI for the contract function that matches the given `fn_name` and arguments `args`. The `data` parameter defaults to the function selector.

```
>>> contract.encode_abi(fn_name="register", args=["rainbows", 10])

↪"0xea87152b0000000000000000000000000000000000000000000000000000000000000040000000000000000000000
↪"
```

classmethod Contract.**encodeABI**(*fn_name*, *args=None*, *kwargs=None*, *data=None*)

Deprecated since version 7.0: Use *Contract.encode_abi()* instead.

classmethod Contract.**all_functions**()

Returns a list of all the functions present in a Contract where every function is an instance of *ContractFunction*.

```
>>> contract.all_functions()
[<Function identity(uint256,bool)>, <Function identity(int256,bool)>]
```

classmethod Contract.**get_function_by_signature**(*signature*)

Searches for a distinct function with matching signature. Returns an instance of *ContractFunction* upon finding a match. Raises `ValueError` if no match is found.

```
>>> contract.get_function_by_signature('identity(uint256,bool)')
<Function identity(uint256,bool)>
```

classmethod Contract.**find_functions_by_name**(*name*)

Searches for all function with matching name. Returns a list of matching functions where every function is an instance of *ContractFunction*. Returns an empty list when no match is found.

```
>>> contract.find_functions_by_name('identity')
[<Function identity(uint256,bool)>, <Function identity(int256,bool)>]
```

classmethod Contract.**get_function_by_name**(*name*)

Searches for a distinct function with matching name. Returns an instance of *ContractFunction* upon finding a match. Raises `ValueError` if no match is found or if multiple matches are found.

```
>>> contract.get_function_by_name('unique_name')
<Function unique_name(uint256)>
```

classmethod Contract.**get_function_by_selector**(*selector*)

Searches for a distinct function with matching selector. The selector can be a hexadecimal string, bytes or int. Returns an instance of *ContractFunction* upon finding a match. Raises `ValueError` if no match is found.

```
>>> contract.get_function_by_selector('0xac37eebb')
<Function identity(uint256)'>
>>> contract.get_function_by_selector(b'\xac7\xee\xbb')
<Function identity(uint256)'>
>>> contract.get_function_by_selector(0xac37eebb)
<Function identity(uint256)'>
```

classmethod Contract.**find_functions_by_args**(**args*)

Searches for all function with matching args. Returns a list of matching functions where every function is an instance of *ContractFunction*. Returns an empty list when no match is found.

```
>>> contract.find_functions_by_args(1, True)
[<Function identity(uint256,bool)>, <Function identity(int256,bool)>]
```

classmethod Contract.**get_function_by_args**(**args*)

Searches for a distinct function with matching args. Returns an instance of *ContractFunction* upon finding a match. Raises `ValueError` if no match is found or if multiple matches are found.

```
>>> contract.get_function_by_args(1)
<Function unique_func_with_args(uint256)>
```

Note: `Contract` methods `all_functions`, `get_function_by_signature`, `find_functions_by_name`, `get_function_by_name`, `get_function_by_selector`, `find_functions_by_args` and `get_function_by_args` can only be used when abi is provided to the contract.

Note: web3.py rejects the initialization of contracts that have more than one function with the same selector or signature. eg. `blockHashAddendsInexpansible(uint256)` and `blockHashAskewLimitary(uint256)` have the same selector value equal to `0x00000000`. A contract containing both of these functions will be rejected.

2.9.5 Invoke Ambiguous Contract Functions Example

Below is an example of a contract that has multiple functions of the same name, and the arguments are ambiguous.

```
>>> contract_source_code = """
pragma solidity ^0.4.21;
contract AmbiguousDuo {
  function identity(uint256 input, bool uselessFlag) returns (uint256) {
    return input;
  }
  function identity(int256 input, bool uselessFlag) returns (int256) {
    return input;
  }
}
"""
# fast forward all the steps of compiling and deploying the contract.
>>> ambiguous_contract.functions.identity(1, True) # raises Web3ValidationError

>>> identity_func = ambiguous_contract.get_function_by_signature('identity(uint256,bool)
↪')
>>> identity_func(1, True)
<Function identity(uint256,bool) bound to (1, True)>
>>> identity_func(1, True).call()
1
```

2.9.6 Disabling Strict Checks for Bytes Types

By default, web3 is strict when it comes to hex and bytes values, as of `v6`. If an abi specifies a byte size, but the value that gets passed in is not the specified size, web3 will invalidate the value. For example, if an abi specifies a type of `bytes4`, web3 will invalidate the following values:

Table 1: Invalid byte and hex strings with strict (default) bytes4 type checking

Input	Reason
`''`	Needs to be prefixed with a "0x" to be interpreted as an empty hex string
`2`	Wrong type
`'ah'`	String is not valid hex
`'1234'`	Needs to either be a bytestring (b'1234') or be a hex value of the right size, prefixed with 0x (in this case: '0x31323334')
`b''`	Needs to have exactly 4 bytes
`b'ab'`	Needs to have exactly 4 bytes
`'0xab'`	Needs to have exactly 4 bytes
`'0x6162636464'`	Needs to have exactly 4 bytes

However, you may want to be less strict with acceptable values for bytes types. This may prove useful if you trust that values coming through are what they are meant to be with respect to the ABI. In this case, the automatic padding might be convenient for inferred types. For this, you can set the `w3.strict_bytes_type_checking()` flag to `False`, which is available on the Web3 instance. A Web3 instance which has this flag set to `False` will have a less strict set of rules on which values are accepted. A `bytes` type will allow values as a hex string, a bytestring, or a regular Python string that can be decoded as a hex. 0x-prefixed hex strings are also not required.

- A Python string that is not prefixed with `0x` is valid.
- A bytestring whose length is less than the specified byte size is valid.

Table 2: Valid byte and hex strings for a non-strict bytes4 type

Input	Normalizes to
`''`	`b'\x00\x00\x00\x00'`
`'0x'`	`b'\x00\x00\x00\x00'`
`b''`	`b'\x00\x00\x00\x00'`
`b'ab'`	`b'ab\x00\x00'`
`'0xab'`	`b'\xab\x00\x00\x00'`
`'1234'`	`b'\x124\x00\x00'`
`'0x61626364'`	`b'abcd'`
`'1234'`	`b'1234'`

Taking the following contract code as an example:

```
>>> #  pragma solidity >=0.4.22 <0.6.0;
...
... #   contract ArraysContract {
... #      bytes2[] public bytes2Value;

... #      constructor(bytes2[] memory _bytes2Value) public {
... #          bytes2Value = _bytes2Value;
... #      }

... #      function setBytes2Value(bytes2[] memory _bytes2Value) public {
... #          bytes2Value = _bytes2Value;
... #      }

... #      function getBytes2Value() public view returns (bytes2[] memory) {
```

(continues on next page)

(continued from previous page)

```
... #           return bytes2Value;
... #       }
... #   }

>>> # abi = "..."
>>> # bytecode = "6080..."
```

```
>>> arrays_contract_instance = w3.eth.contract(abi=abi, bytecode=bytecode)

>>> tx_hash = arrays_contract_instance.constructor([b'bb']).transact()
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> arrays_contract = w3.eth.contract(
...     address=tx_receipt.contractAddress,
...     abi=abi
... )
>>> arrays_contract.functions.getBytes2Value().call()
[b'bb']

>>> # set value with appropriate byte size
>>> arrays_contract.functions.setBytes2Value([b'aa']).transact({'gas': 420000,
↪"maxPriorityFeePerGas": 10 ** 9, "maxFeePerGas": 10 ** 9})
HexBytes('0xcb95151142ea56dbf2753d70388aef202a7bb5a1e323d448bc19f1d2e1fe3dc9')
>>> # check value
>>> arrays_contract.functions.getBytes2Value().call()
[b'aa']

>>> # trying to set value without appropriate size (bytes2) is not valid
>>> arrays_contract.functions.setBytes2Value([b'b']).transact()
Traceback (most recent call last):
   ...
web3.exceptions.Web3ValidationError:
Could not identify the intended function with name
>>> # check value is still b'aa'
>>> arrays_contract.functions.getBytes2Value().call()
[b'aa']

>>> # disabling strict byte checking...
>>> w3.strict_bytes_type_checking = False

>>> tx_hash = arrays_contract_instance.constructor([b'b']).transact()
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> arrays_contract = w3.eth.contract(
...     address=tx_receipt.contractAddress,
...     abi=abi
... )
>>> # check value is zero-padded... i.e. b'b\x00'
>>> arrays_contract.functions.getBytes2Value().call()
[b'b\x00']

>>> # set the flag back to True
>>> w3.strict_bytes_type_checking = True
```

(continues on next page)

(continued from previous page)

```
>>> arrays_contract.functions.setBytes2Value([b'a']).transact()
Traceback (most recent call last):
   ...
web3.exceptions.Web3ValidationError:
Could not identify the intended function with name
```

2.9.7 Contract Functions

class web3.contract.**ContractFunction**

The named functions exposed through the *Contract.functions* property are of the ContractFunction type. This class is not to be used directly, but instead through *Contract.functions*.

For example:

```
myContract = web3.eth.contract(address=contract_address, abi=contract_abi)
twentyone = myContract.functions.multiply7(3).call()
```

If you have the function name in a variable, you might prefer this alternative:

```
func_to_call = 'multiply7'
contract_func = myContract.functions[func_to_call]
twentyone = contract_func(3).call()
```

ContractFunction provides methods to interact with contract functions. Positional and keyword arguments supplied to the contract function subclass will be used to find the contract function by signature, and forwarded to the contract function when applicable.

EIP-3668 introduced support for the `OffchainLookup` revert / CCIP Read support. CCIP Read is set to `True` for calls by default, as recommended in EIP-3668. This is done via a global `global_ccip_read_enabled` flag on the provider. If raising the `OffchainLookup` revert is preferred for a specific call, the `ccip_read_enabled` flag on the call may be set to `False`.

```
>>> # raises the revert instead of handling the offchain lookup
>>> myContract.functions.revertsWithOffchainLookup(myData).call(ccip_read_
↪enabled=False)
*** web3.exceptions.OffchainLookup
```

Disabling CCIP Read support can be useful if a transaction needs to be sent to the callback function. In such cases, "preflighting" with an `eth_call`, handling the `OffchainLookup`, and sending the data via a transaction may be necessary. See *CCIP Read support for offchain lookup* in the examples section for how to preflight a transaction with a contract call.

Similarly, if CCIP Read is globally set to `False` via the `global_ccip_read_enabled` flag on the provider, it may be enabled on a per-call basis - overriding the global flag. This ensures only explicitly enabled calls will handle the `OffchainLookup` revert appropriately.

```
>>> # global flag set to `False`
>>> w3.provider.global_ccip_read_enabled = False

>>> # does not raise the revert since explicitly enabled on the call:
>>> response = myContract.functions.revertsWithOffchainLookup(myData).
↪call(ccip_read_enabled=True)
```

If the function called results in a `revert` error, a `ContractLogicError` will be raised. If there is an error message with the error, web3.py attempts to parse the message that comes back and return it to the user as the error string. As of v6.3.0, the raw data is also returned and can be accessed via the `data` attribute on `ContractLogicError`.

Methods

`ContractFunction.transact`(*transaction*)

Execute the specified function by sending a new public transaction.

Refer to the following invocation:

```
myContract.functions.myMethod(*args, **kwargs).transact(transaction)
```

The first portion of the function call `myMethod(*args, **kwargs)` selects the appropriate contract function based on the name and provided argument. Arguments can be provided as positional arguments, keyword arguments, or a mix of the two.

The end portion of this function call `transact(transaction)` takes a single parameter which should be a python dictionary conforming to the same format as the `web3.eth.send_transaction(transaction)` method. This dictionary may not contain the keys `data`.

If any of the `args` or `kwargs` specified in the ABI are an `address` type, they will accept ENS names.

If a `gas` value is not provided, then the `gas` value for the method transaction will be created using the `web3.eth.estimate_gas()` method.

Returns the transaction hash.

```
>>> token_contract.functions.transfer(web3.eth.accounts[1], 12345).transact()
"0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd"
```

`ContractFunction.call`(*transaction*, *block_identifier='latest'*)

Call a contract function, executing the transaction locally using the `eth_call` API. This will not create a new public transaction.

Refer to the following invocation:

```
myContract.functions.myMethod(*args, **kwargs).call(transaction)
```

This method behaves the same as the *`ContractFunction.transact()`* method, with transaction details being passed into the end portion of the function call, and function arguments being passed into the first portion.

Returns the return value of the executed function.

```
>>> my_contract.functions.multiply7(3).call()
21
>>> token_contract.functions.myBalance().call({'from': web3.eth.coinbase})
12345  # the token balance for `web3.eth.coinbase`
>>> token_contract.functions.myBalance().call({'from': web3.eth.accounts[1]})
54321  # the token balance for the account `web3.eth.accounts[1]`
```

You can call the method at a historical block using `block_identifier`. Some examples:

```
# You can call your contract method at a block number:
>>> token_contract.functions.myBalance().call(block_identifier=10)
```

(continues on next page)

(continued from previous page)

```
# or a number of blocks back from pending,
# in this case, the block just before the latest block:
>>> token_contract.functions.myBalance().call(block_identifier=-2)

# or a block hash:
>>> token_contract.functions.myBalance().call(block_identifier=
↪'0x4ff4a38b278ab49f7739d3a4ed4e12714386a9fdf72192f2e8f7da7822f10b4d')
>>> token_contract.functions.myBalance().call(block_identifier=b'O\xf4\xa3\x8b\'\
↪x8a\xb4\x9fw9\xd3\xa4\xedN\x12qC\x86\xa9\xfd\xf7!\x92\xf2\xe8\xf7\xdax"\xf1\x0bM')

# Latest is the default, so this is redundant:
>>> token_contract.functions.myBalance().call(block_identifier='latest')

# You can check the state after your pending transactions (if supported by your␣
↪node):
>>> token_contract.functions.myBalance().call(block_identifier='pending')
```

Passing the `block_identifier` parameter for past block numbers requires that your Ethereum API node is running in the more expensive archive node mode. Normally synced Ethereum nodes will fail with a "missing trie node" error, because Ethereum node may have purged the past state from its database. More information about archival nodes here.

`ContractFunction.`**`estimate_gas`**(*transaction*, *block_identifier=None*)

Call a contract function, executing the transaction locally using the `eth_call` API. This will not create a new public transaction.

Refer to the following invocation:

```
myContract.functions.myMethod(*args, **kwargs).estimate_gas(transaction)
```

This method behaves the same as the *`ContractFunction.transact()`* method, with transaction details being passed into the end portion of the function call, and function arguments being passed into the first portion.

Returns the amount of gas consumed which can be used as a gas estimate for executing this transaction publicly.

```
>>> my_contract.functions.multiply7(3).estimate_gas()
42650
```

Note: The parameter `block_identifier` is not enabled in geth nodes, hence passing a value of `block_identifier` when connected to a geth nodes would result in an error like: `ValueError: {'code': -32602, 'message': 'too many arguments, want at most 1'}`

`ContractFunction.`**`build_transaction`**(*transaction*)

Builds a transaction dictionary based on the contract function call specified.

Refer to the following invocation:

```
myContract.functions.myMethod(*args, **kwargs).build_transaction(transaction)
```

This method behaves the same as the `Contract.transact()` method, with transaction details being passed into the end portion of the function call, and function arguments being passed into the first portion.

Note: *nonce* is not returned as part of the transaction dictionary unless it is specified in the first portion of the function call:

```
>>> math_contract.functions.increment(5).build_transaction({'nonce': 10})
```

You may use `getTransactionCount()` to get the current nonce for an account. Therefore a shortcut for producing a transaction dictionary with nonce included looks like:

```
>>> math_contract.functions.increment(5).build_transaction({'nonce': web3.eth.get_
↪transaction_count('0xF5...')})
```

Returns a transaction dictionary. This transaction dictionary can then be sent using *send_transaction()*.

Additionally, the dictionary may be used for offline transaction signing using `sign_transaction()`.

```
>>> math_contract.functions.increment(5).build_transaction({'maxFeePerGas':␣
↪2000000000, 'maxPriorityFeePerGas': 1000000000})
{
    'to': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
    'data':
↪'0x7cf5dab00000000000000000000000000000000000000000000000000000000000000005',
    'value': 0,
    'gas': 43242,
    'maxFeePerGas': 2000000000,
    'maxPriorityFeePerGas': 1000000000,
    'chainId': 1
}
```

Fallback Function

The Contract Factory also offers an API to interact with the fallback function, which supports four methods like normal functions:

`Contract.fallback.`**`call`**`(`*transaction*`)`

Call fallback function, executing the transaction locally using the `eth_call` API. This will not create a new public transaction.

`Contract.fallback.`**`estimate_gas`**`(`*transaction*`)`

Call fallback function and return the gas estimation.

`Contract.fallback.`**`transact`**`(`*transaction*`)`

Execute fallback function by sending a new public transaction.

`Contract.fallback.`**`build_transaction`**`(`*transaction*`)`

Builds a transaction dictionary based on the contract fallback function call.

2.9.8 Events

class web3.contract.**ContractEvents**

The named events exposed through the *Contract.events* property are of the ContractEvents type. This class is not to be used directly, but instead through *Contract.events*.

For example:

```
myContract = web3.eth.contract(address=contract_address, abi=contract_abi)
tx_hash = myContract.functions.myFunction().transact()
receipt = web3.eth.get_transaction_receipt(tx_hash)
myContract.events.myEvent().process_receipt(receipt)
```

ContractEvent provides methods to interact with contract events. Positional and keyword arguments supplied to the contract event subclass will be used to find the contract event by signature.

ContractEvents.**myEvent**(**args, **kwargs).get_logs(fromBlock=None, toBlock="latest", block_hash=None, argument_filters={}*)

Fetches all logs for a given event within the specified block range or block hash.

argument_filters is an optional dictionary argument that can be used to filter for logs where the event's argument values match the values provided in the dictionary. The keys must match the event argument names as they exist in the ABI. The values can either be a single value or a list of values to match against. If a list is provided, the logs will be filtered for any logs that match any of the values in the list. Indexed arguments are filtered pre-call by building specific topics to filter for. Non-indexed arguments are filtered by the library after the logs are fetched from the node.

```
myContract = web3.eth.contract(address=contract_address, abi=contract_abi)

# get ``myEvent`` logs from block 1337 to block 2337 where the value for the
# event argument "eventArg1" is either 1, 2, or 3
myContract.events.myEvent().get_logs(
    argument_filters={"eventArg1": [1, 2, 3]},
    fromBlock=1337,
    toBlock=2337,
)
```

ContractEvents.**myEvent**(**args, **kwargs).process_receipt(transaction_receipt, errors=WARN*)

Extracts the pertinent logs from a transaction receipt.

If there are no errors, process_receipt returns a tuple of *Event Log Objects*, emitted from the event (e.g. myEvent), with decoded output.

```
>>> tx_hash = contract.functions.myFunction(12345).transact({'to':contract_address})
>>> tx_receipt = w3.eth.get_transaction_receipt(tx_hash)
>>> rich_logs = contract.events.myEvent().process_receipt(tx_receipt)
>>> rich_logs[0]['args']
{'myArg': 12345}
```

If there are errors, the logs will be handled differently depending on the flag that is passed in:

- WARN (default) - logs a warning to the console for the log that has an error, and discards the log. Returns any logs that are able to be processed.
- STRICT - stops all processing and raises the error encountered.

- IGNORE - returns any raw logs that raised an error with an added "errors" field, along with any other logs were able to be processed.
- DISCARD - silently discards any logs that have errors, and returns processed logs that don't have errors.

An event log error flag needs to be imported from web3/logs.py.

```
>>> tx_hash = contract.functions.myFunction(12345).transact({'to':contract_address})
>>> tx_receipt = w3.eth.get_transaction_receipt(tx_hash)
>>> processed_logs = contract.events.myEvent().process_receipt(tx_receipt)
>>> processed_logs
(
   AttributeDict({
       'args': AttributeDict({}),
       'event': 'myEvent',
       'logIndex': 0,
       'transactionIndex': 0,
       'transactionHash': HexBytes(
↪'0xfb95ccb6ab39e19821fb339dee33e7afe2545527725b61c64490a5613f8d11fa'),
       'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
       'blockHash': HexBytes(
↪'0xd74c3e8bdb19337987b987aee0fa48ed43f8f2318edfc84e3a8643e009592a68'),
       'blockNumber': 3
   })
)


# Or, if there were errors encountered during processing:
>>> from web3.logs import STRICT, IGNORE, DISCARD, WARN
>>> processed_logs = contract.events.myEvent().process_receipt(tx_receipt,␣
↪errors=IGNORE)
>>> processed_logs
(
    AttributeDict({
        'type': 'mined',
        'logIndex': 0,
        'transactionIndex': 0,
        'transactionHash': HexBytes(
↪'0x01682095d5abb0270d11a31139b9a1f410b363c84add467004e728ec831bd529'),
        'blockHash': HexBytes(
↪'0x92abf9325a3959a911a2581e9ea36cba3060d8b293b50e5738ff959feb95258a'),
        'blockNumber': 5,
        'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
        'data': '0x0000000000000000000000000000000000000000000000000000000000003039
↪',
        'topics': [
            HexBytes(
↪'0xf70fe689e290d8ce2b2a388ac28db36fbb0e16a6d89c6804c461f65a1b40bb15')
        ],
        'errors': LogTopicError('Expected 1 log topics.  Got 0')})
   })
)
>>> processed_logs = contract.events.myEvent().process_receipt(tx_receipt,␣
↪errors=DISCARD)
```

(continues on next page)

(continued from previous page)

```
>>> assert processed_logs == ()
True
```

ContractEvents.**myEvent**(**args, **kwargs).process_log(log)*

Similar to *process_receipt*, but only processes one log at a time, instead of a whole transaction receipt. Will return a single *Event Log Object* if there are no errors encountered during processing. If an error is encountered during processing, it will be raised.

```
>>> tx_hash = contract.functions.myFunction(12345).transact({'to':contract_address})
>>> tx_receipt = w3.eth.get_transaction_receipt(tx_hash)
>>> log_to_process = tx_receipt['logs'][0]
>>> processed_log = contract.events.myEvent().process_log(log_to_process)
>>> processed_log
AttributeDict({
    'args': AttributeDict({}),
    'event': 'myEvent',
    'logIndex': 0,
    'transactionIndex': 0,
    'transactionHash': HexBytes(
↪'0xfb95ccb6ab39e19821fb339dee33e7afe2545527725b61c64490a5613f8d11fa'),
    'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
    'blockHash': HexBytes(
↪'0xd74c3e8bdb19337987b987aee0fa48ed43f8f2318edfc84e3a8643e009592a68'),
    'blockNumber': 3
})
```

Event Log Object

The Event Log Object is a python dictionary with the following keys:

- `args`: Dictionary - The arguments coming from the event.
- `event`: String - The event name.
- `logIndex`: Number - integer of the log index position in the block.
- `transactionIndex`: Number - integer of the transactions index position log was created from.
- `transactionHash`: String, 32 Bytes - hash of the transactions this log was created from.
- `address`: String, 32 Bytes - address from which this log originated.
- `blockHash`: String, 32 Bytes - hash of the block where this log was in. null when it's pending.
- `blockNumber`: Number - the block number where this log was in. null when it's pending.

```
>>> transfer_filter = my_token_contract.events.Transfer.create_filter(fromBlock="0x0",␣
↪argument_filters={'from': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf'})
>>> transfer_filter.get_new_entries()
[AttributeDict({'args': AttributeDict({'from':
↪'0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'to': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'value': 10}),
 'event': 'Transfer',
 'logIndex': 0,
```

(continues on next page)

(continued from previous page)

```
 'transactionIndex': 0,
 'transactionHash': HexBytes(
↪'0x9da859237e7259832b913d51cb128c8d73d1866056f7a41b52003c953e749678'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 2})]
>>> transfer_filter.get_new_entries()
[]
>>> tx_hash = contract.functions.transfer(alice, 10).transact({'gas': 899000, 'gasPrice
↪': 1000000000})
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> transfer_filter.get_new_entries()
[AttributeDict({'args': AttributeDict({'from':
↪'0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'to': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'value': 10}),
 'event': 'Transfer',
 'logIndex': 0,
 'transactionIndex': 0,
 'transactionHash': HexBytes('...'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 3})]
>>> transfer_filter.get_all_entries()
[AttributeDict({'args': AttributeDict({'from':
↪'0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'to': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'value': 10}),
 'event': 'Transfer',
 'logIndex': 0,
 'transactionIndex': 0,
 'transactionHash': HexBytes('...'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 2}),
 AttributeDict({'args': AttributeDict({'from':
↪'0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'to': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'value': 10}),
 'event': 'Transfer',
 'logIndex': 0,
 'transactionIndex': 0,
 'transactionHash': HexBytes('...'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 3})]
```

2.9.9 Utils

classmethod Contract.**decode_function_input**(*data*)

Decodes the transaction data used to invoke a smart contract function, and returns *ContractFunction* and decoded parameters as dict.

```
>>> transaction = w3.eth.get_transaction(
↪'0x5798fbc45e3b63832abc4984b0f3574a13545f415dd672cd8540cd71f735db56')
>>> transaction.input

↪'0x612e45a3000000000000000000000000b656b2a9c3b2416437a811e07466ca712f5a5b5a0000000000000000000000
↪'
>>> contract.decode_function_input(transaction.input)
(<Function newProposal(address,uint256,string,bytes,uint256,bool)>,
 {'_recipient': '0xB656b2a9c3b2416437A811e07466cA712F5a5b5a',
  '_amount': 0,
  '_description': b'lonely, so lonely',
  '_transactionData': b'',
  '_debatingPeriod': 604800,
  '_newCurator': True})
```

2.9.10 ContractCaller

class web3.contract.**ContractCaller**

The `ContractCaller` class provides an API to call functions in a contract. This class is not to be used directly, but instead through `Contract.caller`.

There are a number of different ways to invoke the `ContractCaller`.

For example:

```
>>> myContract = w3.eth.contract(address=address, abi=ABI)
>>> twentyone = myContract.caller.multiply7(3)
>>> twentyone
21
```

It can also be invoked using parentheses:

```
>>> twentyone = myContract.caller().multiply7(3)
>>> twentyone
21
```

And a transaction dictionary, with or without the `transaction` keyword. You can also optionally include a block identifier. For example:

```
>>> from_address = w3.eth.accounts[1]
>>> twentyone = myContract.caller({'from': from_address}).multiply7(3)
>>> twentyone
21
>>> twentyone = myContract.caller(transaction={'from': from_address}).multiply7(3)
>>> twentyone
21
```

(continues on next page)

(continued from previous page)

```
>>> twentyone = myContract.caller(block_identifier='latest').multiply7(3)
>>> twentyone
21
```

Like `ContractFunction`, `ContractCaller` provides methods to interact with contract functions. Positional and keyword arguments supplied to the contract caller subclass will be used to find the contract function by signature, and forwarded to the contract function when applicable.

2.9.11 Contract FAQs

How do I pass in a struct as a function argument?

web3.py accepts struct arguments as dictionaries. This format also supports nested structs. Let's take a look at a quick example. Given the following Solidity contract:

```
contract Example {
  address addr;

  struct S1 {
    address a1;
    address a2;
  }

  struct S2 {
    bytes32 b1;
    bytes32 b2;
  }

  struct X {
    S1 s1;
    S2 s2;
    address[] users;
  }

  function update(X memory x) public {
    addr = x.s1.a2;
  }

  function retrieve() public view returns (address) {
    return addr;
  }
}
```

You can interact with web3.py contract API as follows:

```
# deploy or lookup the deployed contract, then:

>>> deployed_contract.functions.retrieve().call()
'0x0000000000000000000000000000000000000000'

>>> deployed_contract.functions.update({'s1': [
```

(continues on next page)

(continued from previous page)

```
↪'0x0000000000000000000000000000000000000001',
↪'0x0000000000000000000000000000000000000002'], 's2': [b'0'*32, b'1'*32], 'users': []}).
↪transact()

>>> deployed_contract.functions.retrieve().call()
'0x0000000000000000000000000000000000000002'
```

Where can I find more information about Ethereum Contracts?

Comprehensive documentation for Contracts is available from the Solidity Docs.

2.10 ABI Types

The Web3 library follows the following conventions.

2.10.1 Bytes vs Text

- The term *bytes* is used to refer to the binary representation of a string.
- The term *text* is used to refer to unicode representations of strings.

2.10.2 Hexadecimal Representations

- All hexadecimal values will be returned as text.
- All hexadecimal values will be `0x` prefixed.

2.10.3 Ethereum Addresses

All addresses must be supplied in one of three ways:

- A 20-byte hexadecimal that is checksummed using the EIP-55 spec.
- A 20-byte binary address (python bytes type).
- While connected to an Ethereum Name Service (ENS) supported chain, an ENS name (often in the form `myname.eth`).

2.10.4 Disabling Strict Bytes Type Checking

There is a boolean flag on the `Web3` class and the `ENS` class that will disable strict bytes type checking. This allows bytes values of Python strings and allows byte strings less than the specified byte size, appropriately padding values that need padding. To disable stricter checks, set the `w3.strict_bytes_type_checking` (or `ns.strict_bytes_type_checking`) flag to `False`. This will no longer cause the `Web3` / `ENS` instance to raise an error if a Python string is passed in without a “0x” prefix. It will also render valid byte strings or hex strings that are below the exact number of bytes specified by the ABI type by padding the value appropriately, according to the ABI type. See the *Disabling Strict Checks for Bytes Types* section for an example on using the flag and more details.

Note: If a standalone ENS instance is instantiated from a Web3 instance, i.e. `ns = ENS.from_web3(w3)`, it will inherit the value of the `w3.strict_bytes_type_checking` flag from the Web3 instance at the time of instantiation.

Also of note, all modules on the Web3 class will inherit the value of this flag, since all modules use the parent w3 object reference under the hood. This means that `w3.eth.w3.strict_bytes_type_checking` will always have the same value as `w3.strict_bytes_type_checking`.

For more details on the ABI specification, refer to the Solidity ABI Spec.

2.10.5 Types by Example

Let's use a contrived contract to demonstrate input types in web3.py:

```
contract ManyTypes {
    // booleans
    bool public b;

    // unsigned ints
    uint8 public u8;
    uint256 public u256;
    uint256[] public u256s;

    // signed ints
    int8 public i8;

    // addresses
    address public addr;
    address[] public addrs;

    // bytes
    bytes1 public b1;

    // structs
    struct S {
      address sa;
      bytes32 sb;
    }
    mapping(address => S) addrStructs;

    function updateBool(bool x) public { b = x; }
    function updateUint8(uint8 x) public { u8 = x; }
    function updateUint256(uint256 x) public { u256 = x; }
    function updateUintArray(uint256[] memory x) public { u256s = x; }
    function updateInt8(int8 x) public { i8 = x; }
    function updateAddr(address x) public { addr = x; }
    function updateBytes1(bytes1 x) public { b1 = x; }
    function updateMapping(S memory x) public { addrStructs[x.sa] = x; }
}
```

Booleans

```
contract_instance.functions.updateBool(True).transact()
```

Unsigned Integers

```
contract_instance.functions.updateUint8(255).transact()
contract_instance.functions.updateUint256(2**256 - 1).transact()
contract_instance.functions.updateUintArray([1, 2, 3]).transact()
```

Signed Integers

```
contract_instance.functions.updateInt8(-128).transact()
```

Addresses

```
contract_instance.functions.updateAddr("0x0000000000000000000000000000000000000000").transact()
```

Bytes

```
contract_instance.functions.updateBytes1(HexBytes(255)).transact()
```

Structs

```
contract_instance.functions.updateMapping({"sa": "0x0000000000000000000000000000000000000000", "sb": HexBytes(123)}).transact()
```

2.11 Middleware

Web3 manages layers of middlewares by default. They sit between the public Web3 methods and the *Providers*, which handle native communication with the Ethereum client. Each layer can modify the request and/or response. Some middlewares are enabled by default, and others are available for optional use.

Each middleware layer gets invoked before the request reaches the provider, and then processes the result after the provider returns, in reverse order. However, it is possible for a middleware to return early from a call without the request ever getting to the provider (or even reaching the middlewares that are in deeper layers).

When integrating middleware with your provider, please ensure you're choosing the right version. For `AsyncWeb3` users, select the version prefixed with `async`, such as `async_attrdict_middleware`. On the other hand, `Web3` users should opt for versions lacking the `async` prefix. If an async version isn't listed, it implies it hasn't been made available yet.

More information is available in the "Internals: *Middlewares*" section.

2.11.1 Default Middleware

Middlewares are added by default if you don't add any.

Sync middlewares include:

- `gas_price_strategy`
- `name_to_address`
- `attrdict`
- `validation`
- `abi`
- `gas_estimate`

Async middlewares include:

- `gas_price_strategy`
- `name_to_address`
- `attrdict`
- `validation`
- `gas_estimate`

The defaults are found in `default_middlewares` and `async_default_middlewares` methods in `web3/manager.py`.

AttributeDict

`web3.middleware.attrdict_middleware()`

`web3.middleware.async_attrdict_middleware()`

This middleware recursively converts any dictionary type in the result of a call to an `AttributeDict`. This enables dot-syntax access, like `eth.get_block('latest').number` in addition to `eth.get_block('latest')['number']`.

Note: Accessing a property via attribute breaks type hinting. For this reason, this feature is available as a middleware, which may be removed if desired.

.eth Name Resolution

`web3.middleware.name_to_address_middleware()`

`web3.middleware.async_name_to_address_middleware()`

This middleware converts Ethereum Name Service (ENS) names into the address that the name points to. For example *`w3.eth.send_transaction`* will accept .eth names in the 'from' and 'to' fields.

Note: This middleware only converts ENS names on chains where the proper ENS contracts are deployed to support this functionality. All other cases will result in a `NameNotFound` error.

Gas Price Strategy

`web3.middleware.gas_price_strategy_middleware()`

`web3.middleware.async_gas_price_strategy_middleware()`

> **Warning:** Gas price strategy is only supported for legacy transactions. The London fork introduced `maxFeePerGas` and `maxPriorityFeePerGas` transaction parameters which should be used over `gasPrice` whenever possible.

This adds a `gasPrice` to transactions if applicable and when a gas price strategy has been set. See *Gas Price API* for information about how gas price is derived.

Buffered Gas Estimate

`web3.middleware.buffered_gas_estimate_middleware()`

`web3.middleware.async_buffered_gas_estimate_middleware()`

This adds a gas estimate to transactions if `gas` is not present in the transaction parameters. Sets gas to: `min(w3.eth.estimate_gas + gas_buffer, gas_limit)` where the gas_buffer default is 100,000

HTTPRequestRetry

`web3.middleware.http_retry_request_middleware()`

`web3.middleware.async_http_retry_request_middleware()`

This middleware is a default specifically for HTTPProvider that retries failed requests that return the following errors: `ConnectionError`, `HTTPError`, `Timeout`, `TooManyRedirects`. Additionally there is a whitelist that only allows certain methods to be retried in order to not resend transactions, excluded methods are: `eth_sendTransaction`, `personal_signAndSendTransaction`, `personal_sendTransaction`.

Validation

`web3.middleware.validation_middleware()`

`web3.middleware.async_validation_middleware()`

This middleware includes block and transaction validators which perform validations for transaction parameters.

2.11.2 Configuring Middleware

Middleware can be added, removed, replaced, and cleared at runtime. To make that easier, you can name the middleware for later reference. Alternatively, you can use a reference to the middleware itself.

Middleware Order

Think of the middleware as being layered in an onion, where you initiate a web3.py request at the outermost layer of the onion, and the Ethereum node (like geth) receives and responds to the request inside the innermost layer of the onion. Here is a (simplified) diagram:

```
                            New request from web3.py

                                        |
                                        |
                                        v

                                `````Layer 2``````
                         ```````                  ```````
                    ````` | ````
                 ````   v   ````
              ```   ```
            `.   ````````Layer 1```````   `.`
          ``   ````   `````   .`
        `.   ``` | ```   `.`
       .`   ```   v   ```   `.
     `.          `.`   ```   .`
    ``          .`                  `Layer 0`                  ``   .`
   ``   `.   `````   ``````   .   .`
  `.   ``   ``` | ```   .`   .
  .   ``            `.`               |           ``   .   .
 .   `.   ``   JSON-RPC call   .`            .          .`
 .   .   ``              |             .            ``   .
``   .   .   v   .   .   .
.   .`   .   .   .   ``
.   .   .   Ethereum node   .`   .   .
.   .   .   .   .   .
.   ``   `. | .   .   .
.   .   .`              |              .`   .   .
`.         .`   .`          Response         .`   .`   .
 .   .   `.` | `.`   `.   .
 `.   .   ``` | ````   `.   .
  .   `.   `````   v   ````   `.   ``
   .   .`   ```Layer 0``                  ``   `.
    .   `.                                           `.`           `.
     .   `.                    |                  `.`            `.
      .`   ``` | ```   .`
       `.   ```   v   ````   `.`
         ``   ``````   `````   .`
           ``   `````Layer 1`````   `.`
             ```   ```
               ```` | ```
                  `````   v   ````
                      ``````   `````
                            `````````Layer 2``````````

                                        |
                                        v
```

(continues on next page)

(continued from previous page)

```
                    Returned value in web3.py
```

The middlewares are maintained in `Web3.middleware_onion`. See below for the API.

When specifying middlewares in a list, or retrieving the list of middlewares, they will be returned in the order of outermost layer first and innermost layer last. In the above example, that means that `w3.middleware_onion.middlewares` would return the middlewares in the order of: `[2, 1, 0]`.

See "Internals: *Middlewares*" for a deeper dive to how middlewares work.

Middleware Stack API

To add or remove items in different layers, use the following API:

`Web3.middleware_onion.add`(*middleware*, *name=None*)

Middleware will be added to the outermost layer. That means the new middleware will modify the request first, and the response last. You can optionally name it with any hashable object, typically a string.

```
>>> w3 = Web3(...)
>>> w3.middleware_onion.add(web3.middleware.gas_price_strategy_middleware)
# or
>>> w3.middleware_onion.add(web3.middleware.gas_price_strategy_middleware, 'gas_
↪price_strategy')
# or
>>> async_w3 = AsyncWeb3(...)
>>> async_w3.middleware_onion.add(web3.middleware.async_gas_price_strategy_
↪middleware, 'gas_price_strategy')
```

`Web3.middleware_onion.inject`(*middleware*, *name=None*, *layer=None*)

Inject a named middleware to an arbitrary layer.

The current implementation only supports injection at the innermost or outermost layers. Note that injecting to the outermost layer is equivalent to calling `Web3.middleware_onion.add()` .

```
# Either of these will put the gas_price_strategy middleware at the innermost layer
>>> w3 = Web3(...)
>>> w3.middleware_onion.inject(web3.middleware.gas_price_strategy_middleware,␣
↪layer=0)
# or
>>> w3.middleware_onion.inject(web3.middleware.gas_price_strategy_middleware, 'gas_
↪price_strategy', layer=0)
# or
>>> async_w3 = AsyncWeb3(...)
>>> async_w3.middleware_onion.inject(web3.middleware.async_gas_price_strategy_
↪middleware, 'gas_price_strategy', layer=0)
```

`Web3.middleware_onion.remove`(*middleware*)

Middleware will be removed from whatever layer it was in. If you added the middleware with a name, use the name to remove it. If you added the middleware as an object, use the object again later to remove it:

```
>>> w3 = Web3(...)
>>> w3.middleware_onion.remove(web3.middleware.gas_price_strategy_middleware)
```

(continues on next page)

(continued from previous page)

```
# or
>>> w3.middleware_onion.remove('gas_price_strategy')
```

Web3.middleware_onion.**replace**(*old_middleware*, *new_middleware*)

Middleware will be replaced from whatever layer it was in. If the middleware was named, it will continue to have the same name. If it was un-named, then you will now reference it with the new middleware object.

```
>>> from web3.middleware import gas_price_strategy_middleware, attrdict_middleware
>>> w3 = Web3(...)

>>> w3.middleware_onion.replace(gas_price_strategy_middleware, attrdict_middleware)
# this is now referenced by the new middleware object, so to remove it:
>>> w3.middleware_onion.remove(attrdict_middleware)

# or, if it was named

>>> w3.middleware_onion.replace('gas_price_strategy', attrdict_middleware)
# this is still referenced by the original name, so to remove it:
>>> w3.middleware_onion.remove('gas_price_strategy')
```

Web3.middleware_onion.**clear**()

Empty all the middlewares, including the default ones.

```
>>> w3 = Web3(...)
>>> w3.middleware_onion.clear()
>>> assert len(w3.middleware_onion) == 0
```

Web3.middleware_onion.**middlewares**

Return all the current middlewares for the `Web3` instance in the appropriate order for importing into a new `Web3` instance.

```
>>> w3_1 = Web3(...)
# add uniquely named middleware:
>>> w3_1.middleware_onion.add(web3.middleware.gas_price_strategy_middleware, 'test_
↪middleware')
# export middlewares from first w3 instance
>>> middlewares = w3_1.middleware_onion.middlewares

# import into second instance
>>> w3_2 = Web3(..., middlewares=middlewares)
>>> assert w3_1.middleware_onion.middlewares == w3_2.middleware_onion.middlewares
>>> assert w3_2.middleware_onion.get('test_middleware')
```

2.11.3 Optional Middleware

Web3 ships with non-default middleware, for your custom use. In addition to the other ways of *Configuring Middleware*, you can specify a list of middleware when initializing Web3, with:

```
Web3(middlewares=[my_middleware1, my_middleware2])
```

> **Warning:** This will *replace* the default middlewares. To keep the default functionality, either use `middleware_onion.add()` from above, or add the default middlewares to your list of new middlewares.

Below is a list of available middlewares which are not enabled by default.

Stalecheck

`web3.middleware.`**`make_stalecheck_middleware`**`(`*allowable_delay*`)`

`web3.middleware.`**`async_make_stalecheck_middleware`**`(`*allowable_delay*`)`

This middleware checks how stale the blockchain is, and interrupts calls with a failure if the blockchain is too old.

- `allowable_delay` is the length in seconds that the blockchain is allowed to be behind of `time.time()`

Because this middleware takes an argument, you must create the middleware with a method call.

```
two_day_stalecheck = make_stalecheck_middleware(60 * 60 * 24 * 2)
web3.middleware_onion.add(two_day_stalecheck)
```

If the latest block in the blockchain is older than 2 days in this example, then the middleware will raise a `StaleBlockchain` exception on every call except `web3.eth.get_block()`.

Cache

Simple Cache Middleware

`web3.middleware.`**`construct_simple_cache_middleware`**`(`*cache*, *rpc_whitelist*, *should_cache_fn*`)`

`web3.middleware.`**`async_construct_simple_cache_middleware`**`(`*cache*, *rpc_whitelist*, *should_cache_fn*`)`

These simple cache constructor methods accept the following arguments:

Parameters

- **cache** – Must be an instance of the `web3.utils.caching.SimpleCache` class. If a cache instance is not provided, a new instance will be created.
- **rpc_whitelist** – Must be an iterable, preferably a set, of the RPC methods that may be cached. A default list is used if none is provided.
- **should_cache_fn** – Must be a callable with the signature `fn(method, params, response)` which returns whether the response should be cached.

Constructs a middleware which will cache the return values for any RPC method in the `rpc_whitelist`.

Ready to use versions of this middleware can be found at `web3.middleware.simple_cache_middleware` and `web3.middleware.async_simple_cache_middleware`. These are the equivalent of using the constructor methods with the default arguments.

Time-based Cache Middleware

`web3.middleware.construct_time_based_cache_middleware(cache_class, cache_expire_seconds, rpc_whitelist, should_cache_fn)`

The time-based cache constructor method accepts the following arguments:

Parameters

- **cache_class** – Must be a callable which returns an object which implements the dictionary API.
- **rpc_whitelist** – Must be an iterable, preferably a set, of the RPC methods that may be cached. A default list is used if none is provided.
- **should_cache_fn** – Must be a callable with the signature `fn(method, params, response)` which returns whether the response should be cached.

> **Warning:** The `cache_class` argument is slated to change to the `cache` argument with `web3.utils.caching.SimpleCache` instance in web3.py `v7`, as is the current state of the simple cache middleware above.

Constructs a middleware which will cache the return values for any RPC method in the `rpc_whitelist` for an amount of time defined by `cache_expire_seconds`.

- `cache_expire_seconds` should be the number of seconds a value may remain in the cache before being evicted.

A ready to use version of this middleware can be found at `web3.middleware.time_based_cache_middleware`.

`web3.middleware.construct_latest_block_based_cache_middleware(cache_class, average_block_time_sample_size, default_average_block_time, rpc_whitelist, should_cache_fn)`

The latest-block-based cache constructor method accepts the following arguments:

Parameters

- **cache_class** – Must be a callable which returns an object which implements the dictionary API.
- **rpc_whitelist** – Must be an iterable, preferably a set, of the RPC methods that may be cached. A default list is used if none is provided.
- **should_cache_fn** – Must be a callable with the signature `fn(method, params, response)` which returns whether the response should be cached.

> **Warning:** The `cache_class` argument is slated to change to the `cache` argument with `web3.utils.caching.SimpleCache` instance in web3.py `v7`, as is the current state of the simple cache middleware above.

Constructs a middleware which will cache the return values for any RPC method in the `rpc_whitelist` for the latest block. It avoids re-fetching the current latest block for each request by tracking the current average block time and only requesting a new block when the last seen latest block is older than the average block time.

- `average_block_time_sample_size` The number of blocks which should be sampled to determine the average block time.

- `default_average_block_time` The initial average block time value to use for cases where there is not enough chain history to determine the average block time.

A ready to use version of this middleware can be found at `web3.middleware.latest_block_based_cache_middleware`.

Proof of Authority

`web3.middleware.geth_poa_middleware()`

`web3.middleware.async_geth_poa_middleware()`

Note: It's important to inject the middleware at the 0th layer of the middleware onion: `w3.middleware_onion.inject(geth_poa_middleware, layer=0)`

The `geth_poa_middleware` is required to connect to `geth --dev` or the Goerli public network. It may also be needed for other EVM compatible blockchains like Polygon or BNB Chain (Binance Smart Chain).

If the middleware is not injected at the 0th layer of the middleware onion, you may get errors like the example below when interacting with your EVM node.

```
web3.exceptions.ExtraDataLengthError: The field extraData is 97 bytes, but should be
1.  It is quite likely that you are connected to a POA chain. Refer to
http://web3py.readthedocs.io/en/stable/middleware.html#proof-of-authority
for more details. The full extraData is: HexBytes('...')
```

The easiest way to connect to a default `geth --dev` instance which loads the middleware is:

```
>>> from web3.auto.gethdev import w3

# confirm that the connection succeeded
>>> w3.client_version
'Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9'
```

This example connects to a local `geth --dev` instance on Linux with a unique IPC location and loads the middleware:

```
>>> from web3 import Web3, IPCProvider

# connect to the IPC location started with 'geth --dev --datadir ~/mynode'
>>> w3 = Web3(IPCProvider('~/mynode/geth.ipc'))

>>> from web3.middleware import geth_poa_middleware

# inject the poa compatibility middleware to the innermost layer (0th layer)
>>> w3.middleware_onion.inject(geth_poa_middleware, layer=0)

# confirm that the connection succeeded
>>> w3.client_version
'Geth/v1.7.3-stable-4bb3c89d/linux-amd64/go1.9'
```

Why is geth_poa_middleware necessary?

There is no strong community consensus on a single Proof-of-Authority (PoA) standard yet. Some nodes have successful experiments running though. One is go-ethereum (geth), which uses a prototype PoA for its development mode and the Goerli test network.

Unfortunately, it does deviate from the yellow paper specification, which constrains the `extraData` field in each block to a maximum of 32-bytes. Geth's PoA uses more than 32 bytes, so this middleware modifies the block data a bit before returning it.

Locally Managed Log and Block Filters

web3.middleware.**local_filter_middleware**()

web3.middleware.**async_local_filter_middleware**()

This middleware provides an alternative to ethereum node managed filters. When used, Log and Block filter logic are handled locally while using the same web3 filter api. Filter results are retrieved using JSON-RPC endpoints that don't rely on server state.

```
>>> from web3 import Web3, EthereumTesterProvider
>>> w3 = Web3(EthereumTesterProvider())
>>> from web3.middleware import local_filter_middleware
>>> w3.middleware_onion.add(local_filter_middleware)
```

```
#  Normal block and log filter apis behave as before.
>>> block_filter = w3.eth.filter("latest")

>>> log_filter = myContract.events.myEvent.build_filter().deploy()
```

Signing

web3.middleware.**construct_sign_and_send_raw_middleware**(*private_key_or_account*)

web3.middleware.**async_construct_sign_and_send_raw_middleware**(*private_key_or_account*)

This middleware automatically captures transactions, signs them, and sends them as raw transactions. The `from` field on the transaction, or `w3.eth.default_account` must be set to the address of the private key for this middleware to have any effect.

- `private_key_or_account` A single private key or a tuple, list or set of private keys.

  Keys can be in any of the following formats:

  - An `eth_account.LocalAccount` object
  - An `eth_keys.PrivateKey` object
  - A raw private key as a hex string or byte string

```
>>> from web3 import Web3, EthereumTesterProvider
>>> w3 = Web3(EthereumTesterProvider)
>>> from web3.middleware import construct_sign_and_send_raw_middleware
>>> from eth_account import Account
>>> acct = Account.create('KEYSMASH FJAFJKLDSKF7JKFDJ 1530')
>>> w3.middleware_onion.add(construct_sign_and_send_raw_middleware(acct))
>>> w3.eth.default_account = acct.address
```

Hosted nodes (like Infura or Alchemy) only support signed transactions. This often results in `send_raw_transaction` being used repeatedly. Instead, we can automate this process with `construct_sign_and_send_raw_middleware(private_key_or_account)`.

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.HTTPProvider('HTTP_ENDPOINT'))
>>> from web3.middleware import construct_sign_and_send_raw_middleware
>>> from eth_account import Account
>>> import os
>>> acct = w3.eth.account.from_key(os.environ.get('PRIVATE_KEY'))
>>> w3.middleware_onion.add(construct_sign_and_send_raw_middleware(acct))
>>> w3.eth.default_account = acct.address

>>> # use `eth_sendTransaction` to automatically sign and send the raw transaction
>>> w3.eth.send_transaction(tx_dict)
HexBytes('0x09511acf75918fd03de58141d2fd409af4fd6d3dce48eb3aa1656c8f3c2c5c21')
```

Similarly, with AsyncWeb3:

```
>>> from web3 import AsyncWeb3
>>> async_w3 = AsyncWeb3(AsyncHTTPProvider('HTTP_ENDPOINT'))
>>> from web3.middleware import async_construct_sign_and_send_raw_middleware
>>> from eth_account import Account
>>> import os
>>> acct = async_w3.eth.account.from_key(os.environ.get('PRIVATE_KEY'))
>>> async_w3.middleware_onion.add(await async_construct_sign_and_send_raw_
↪middleware(acct))
>>> async_w3.eth.default_account = acct.address

>>> # use `eth_sendTransaction` to automatically sign and send the raw transaction
>>> await async_w3.eth.send_transaction(tx_dict)
HexBytes('0x09511acf75918fd03de58141d2fd409af4fd6d3dce48eb3aa1656c8f3c2c5c21')
```

Now you can send a transaction from acct.address without having to build and sign each raw transaction.

When making use of this signing middleware, when sending dynamic fee transactions (recommended over legacy transactions), the transaction `type` of `2` (or `'0x2'`) is necessary. This is because transaction signing is validated based on the transaction `type` parameter. This value defaults to `'0x2'` when `maxFeePerGas` and / or `maxPriorityFeePerGas` are present as parameters in the transaction as these params imply a dynamic fee transaction. Since these values effectively replace the legacy `gasPrice` value, do not set a `gasPrice` for dynamic fee transactions. Doing so will lead to validation issues.

```
# dynamic fee transaction, introduced by EIP-1559:
>>> dynamic_fee_transaction = {
...     'type': '0x2',  # optional - defaults to '0x2' when dynamic fee transaction␣
↪params are present
...     'from': acct.address,  # optional if w3.eth.default_account was set with acct.
↪address
...     'to': receiving_account_address,
...     'value': 22,
...     'maxFeePerGas': 2000000000,  # required for dynamic fee transactions
...     'maxPriorityFeePerGas': 1000000000,  # required for dynamic fee transactions
... }
>>> w3.eth.send_transaction(dynamic_fee_transaction)
```

A legacy transaction still works in the same way as it did before EIP-1559 was introduced:

```
>>> legacy_transaction = {
...     'to': receiving_account_address,
...     'value': 22,
...     'gasPrice': 123456,  # optional - if not provided, gas_price_strategy (if␣
↪exists) or eth_gasPrice is used
... }
>>> w3.eth.send_transaction(legacy_transaction)
```

2.12 Web3 Internals

Warning: This section of the documentation is for advanced users. You should probably stay away from these APIs if you don't know what you are doing.

The Web3 library has multiple layers of abstraction between the public api exposed by the web3 object and the backend or node that web3 is connecting to.

- **Providers** are responsible for the actual communication with the blockchain such as sending JSON-RPC requests over HTTP or an IPC socket.
- **Middlewares** provide hooks for monitoring and modifying requests and responses to and from the provider. These can be *global* operating on all providers or specific to one provider.
- **Managers** provide thread safety and primitives to allow for asynchronous usage of web3.

Here are some common things you might want to do with these APIs.

- Redirect certain RPC requests to different providers such as sending all *read* operations to a provider backed by Infura and all *write* operations to a go-ethereum node that you control.
- Transparently intercept transactions sent over `eth_sendTransaction`, sign them locally, and then send them through `eth_sendRawTransaction`.
- Modify the response from an RPC request so that it is returned in different format such as converting all integer values to their hexadecimal representation.
- Validate the inputs to RPC requests

2.12.1 Request Lifecycle

Each web3 RPC call passes through these layers in the following manner.

```
  ***********    ************
  | Request |    | Response |
  ***********    ************
      |                ^
      v                |
+-----------------------------+
|            Manager          |
+-----------------------------+
      |                ^
      v                |
```

(continues on next page)

(continued from previous page)

```
+-----------------------------+
|     Global Middlewares      |
+-----------------------------+
      |                ^
      v                |
+-----------------------------+
|    Provider Middlewares     |
+-----------------------------+
      |                ^
      v                |
+-----------------------------+
|          Provider           |
+-----------------------------+
```

You can visualize this relationship like an onion, with the Provider at the center. The request originates from the Manager, outside of the onion, passing down through each layer of the onion until it reaches the Provider at the center. The Provider then handles the request, producing a response which will then pass back out from the center of the onion, through each layer until it is finally returned by the Manager.

2.12.2 Providers

A provider is responsible for all direct blockchain interactions. In most cases this means interacting with the JSON-RPC server for an ethereum node over HTTP or an IPC socket. There is however nothing which requires providers to be RPC based, allowing for providers designed for testing purposes which use an in-memory EVM to fulfill requests.

Writing your own Provider

Writing your own provider requires implementing two required methods as well as setting the middlewares the provider should use.

`BaseProvider.`**`make_request`**`(`*method*, *params*`)`

Each provider class **must** implement this method. This method **should** return a JSON object with either a `'result'` key in the case of success, or an `'error'` key in the case of failure.

- `method` This will be a string representing the JSON-RPC method that is being called such as `'eth_sendTransaction'`.
- `params` This will be a list or other iterable of the parameters for the JSON-RPC method being called.

`BaseProvider.`**`is_connected`**`(`*show_traceback=False*`)`

This function should return `True` or `False` depending on whether the provider should be considered *connected*. For example, an IPC socket based provider should return `True` if the socket is open and `False` if the socket is closed.

If set to `True`, the optional `show_traceback` boolean will raise a `ProviderConnectionError` and provide information on why the provider should not be considered *connected*.

`BaseProvider.`**`middlewares`**

This should be an iterable of middlewares.

You can set a new list of middlewares by assigning to `provider.middlewares`, with the first middleware that processes the request at the beginning of the list.

2.12.3 Middlewares

Note: The Middleware API in web3 borrows heavily from the Django middleware API introduced in version 1.10.0

Middlewares provide a simple yet powerful api for implementing layers of business logic for web3 requests. Writing middleware is simple.

```
def simple_middleware(make_request, w3):
    # do one-time setup operations here

    def middleware(method, params):
        # do pre-processing here

        # perform the RPC request, getting the response
        response = make_request(method, params)

        # do post-processing here

        # finally return the response
        return response
    return middleware
```

It is also possible to implement middlewares as a class.

```
class SimpleMiddleware:
    def __init__(self, make_request, w3):
        self.w3 = w3
        self.make_request = make_request

    def __call__(self, method, params):
        # do pre-processing here

        # perform the RPC request, getting the response
        response = self.make_request(method, params)

        # do post-processing here

        # finally return the response
        return response
```

The `make_request` parameter is a callable which takes two positional arguments, `method` and `params` which correspond to the RPC method that is being called. There is no requirement that the `make_request` function be called. For example, if you were writing a middleware which cached responses for certain methods your middleware would likely not call the `make_request` method, but instead get the response from some local cache.

The `RequestManager` object exposes the `middleware_onion` object to manage middlewares. It is also exposed on the `Web3` object for convenience. That API is detailed in *Configuring Middleware*.

2.12.4 Managers

The Manager acts as a gatekeeper for the request/response lifecycle. It is unlikely that you will need to change the Manager as most functionality can be implemented in the Middleware layer.

2.12.5 Request Processing for Persistent Connection Providers

class `web3.providers.websocket.request_processor.`**`RequestProcessor`**

The `RequestProcessor` class is responsible for the storing and syncing up of asynchronous requests to responses for a `PersistentConnectionProvider`. The best example of one such provider is the *WebsocketProviderV2*. In order to send a websocket message and receive a response to that particular request, `PersistentConnectionProvider` instances have to match request *id* values to response *id* values coming back from the websocket connection. Any provider that does not adhere to the JSON-RPC 2.0 specification in this way will not work with `PersistentConnectionProvider` instances. The specifics of how the request processor handles this are outlined below.

Listening for Responses

Implementations of the `PersistentConnectionProvider` class have a message listener background task that is called when the websocket connection is established. This task is responsible for listening for any and all messages coming in over the websocket connection and storing them in the `RequestProcessor` instance internal to the `PersistentConnectionProvider` instance. The `RequestProcessor` instance is responsible for storing the messages in the correct cache, either the one-to-one cache or the one-to-many (subscriptions) queue, depending on whether the message has a JSON-RPC *id* value or not.

One-To-One Requests

One-to-one requests can be summarized as any request that expects only one response back. An example is using the `eth` module API to request the latest block number.

```
>>> async def wsV2_one_to_one_example():
...     async with AsyncWeb3.persistent_websocket(
...         WebsocketProviderV2(f"ws://127.0.0.1:8546")
...     ) as w3:
...         # make a request and expect a single response returned on the same line
...         latest_block_num = await w3.eth.block_number

>>> asyncio.run(wsV2_one_to_one_example())
```

With websockets we have to call `send()` and asynchronously receive responses via another means, generally by calling `recv()` or by iterating on the websocket connection for messages. As outlined above, the `PersistentConnectionProvider` class has a message listener background task that handles the receiving of messages.

Due to this asynchronous nature of sending and receiving, in order to make one-to-one request-to-response calls work, we have to save the request information somewhere so that, when the response is received, we can match it to the original request that was made (i.e. the request with a matching *id* to the response that was received). The stored request information is then used to process the response when it is received, piping it through the response formatters and middlewares internal to the *web3.py* library.

In order to store the request information, the `RequestProcessor` class has an internal `RequestInformation` cache. The `RequestInformation` class saves important information about a request.

class web3._utils.caching.**RequestInformation**

method

The name of the method - e.g. "eth_subscribe".

params

The params used when the call was made - e.g. ("newPendingTransactions", True).

response_formatters

The formatters that will be used to process the response.

middleware_response_processors

Any middleware that processes responses that is present on the instance at the time of the request is appended here, in order, so the response may be piped through that logic when it comes in.

subscription_id

If the request is an `eth_subscribe` request, rather than popping this information from the cache when the response to the subscription call comes in (i.e. the subscription *id*), we save the subscription id with the request information so that we can correctly process all subscription messages that come in with that subscription *id*. For one-to-one request-to-response calls, this value is always `None`.

One-to-one responses, those that include a JSON-RPC *id* in the response object, are stored in an internal `SimpleCache` class, isolated from any one-to-many responses. When the `PersistentConnectionProvider` is looking for a response internally, it will expect the message listener task to store the response in this cache. Since the request *id* is used in the cache key generation, it will then look for a cache key that matches the response *id* with that of the request *id*. If the cache key is found, the response is processed and returned to the user. If the cache key is not found, the operation will time out and raise a `TimeExhausted` exception. This timeout can be configured by the user when instantiating the `PersistentConnectionProvider` instance via the `response_timeout` keyword argument.

One-To-Many Requests

One-to-many requests can be summarized by any request that expects many responses as a result of the initial request. The only current example is the `eth_subscribe` request. The initial `eth_subscribe` request expects only one response, the subscription *id* value, but it also expects to receive many `eth_subscription` messages if and when the request is successful. For this reason, the original request is considered a one-to-one request so that a subscription *id* can be returned to the user on the same line, but the `process_subscriptions()` method on the *WebsocketConnection* class, the public API for interacting with the active websocket connection, is set up to receive `eth_subscription` responses over an asynchronous interator pattern.

```
>>> async def ws_v2_subscription_example():
...     async with AsyncWeb3.persistent_websocket(
...         WebsocketProviderV2(f"ws://127.0.0.1:8546")
...     ) as w3:
...         # Subscribe to new block headers and receive the subscription_id.
...         # A one-to-one call with a trigger for many responses
...         subscription_id = await w3.eth.subscribe("newHeads")
...
...         # Listen to the websocket for the many responses utilizing the ``w3.ws``
...         # ``WebsocketConnection`` public API method ``process_subscriptions()``
...         async for response in w3.ws.process_subscriptions():
...             # Receive only one-to-many responses here so that we don't
...             # accidentally return the response for a one-to-one request in this
...             # block
...
```

(continues on next page)

(continued from previous page)

```
...             print(f"{response}\n")
...
...             if some_condition:
...                 # unsubscribe from new block headers, another one-to-one request
...                 is_unsubscribed = await w3.eth.unsubscribe(subscription_id)
...                 if is_unsubscribed:
...                     break

>>> asyncio.run(ws_v2_subscription_example())
```

One-to-many responses, those that do not include a JSON-RPC *id* in the response object, are stored in an internal `asyncio.Queue` instance, isolated from any one-to-one responses. When the `PersistentConnectionProvider` is looking for one-to-many responses internally, it will expect the message listener task to store these messages in this queue. Since the order of the messages is important, the queue is a FIFO queue. The `process_subscriptions()` method on the `WebsocketConnection` class is set up to pop messages from this queue as FIFO over an asynchronous iterator pattern.

If the stream of messages from the websocket is not being interrupted by any other tasks, the queue will generally be in sync with the messages coming in over the websocket. That is, the message listener will put a message in the queue and the `process_subscriptions()` method will pop that message from the queue and yield control of the loop back to the listener. This will continue until the websocket connection is closed or the user unsubscribes from the subscription. If the stream of messages lags a bit, or the provider is not consuming messages but has subscribed to a subscription, this internal queue may fill up with messages until it reaches its max size and then trigger a waiting `asyncio.Event` until the provider begins consuming messages from the queue again. For this reason, it's important to begin consuming messages from the queue, via the `process_subscriptions()` method, as soon as a subscription is made.

2.13 ethPM

Warning: The `ethPM` module is no longer being maintained and will be deprecated with `web3.py` version 7.

2.13.1 Overview

This is a Python implementation of the Ethereum Smart Contract Packaging Specification V3, driven by discussions in ERC 190, ERC 1123, ERC 1319.

`Py-EthPM` is being built as a low-level library to help developers leverage the ethPM spec. Including ...

- Parse and validate packages.
- Construct and publish new packages.
- Provide access to contract factory classes.
- Provide access to all of a package's deployments.
- Validate package bytecode matches compilation output.
- Validate deployed bytecode matches compilation output.
- Access to package's dependencies.
- Native integration with compilation metadata.

2.13.2 Package

The `Package` object will function much like the `Contract` class provided by `web3`. Rather than instantiating the base class provided by `ethpm`, you will instead use a `classmethod` which generates a new `Package` class for a given package.

`Package` objects *must* be instantiated with a valid `web3` object.

```
>>> from ethpm import Package, get_ethpm_spec_dir
>>> from web3 import Web3

>>> w3 = Web3(Web3.EthereumTesterProvider())
>>> ethpm_spec_dir = get_ethpm_spec_dir()
>>> owned_manifest_path = ethpm_spec_dir / 'examples' / 'owned' / 'v3.json'
>>> OwnedPackage = Package.from_file(owned_manifest_path, w3)
>>> assert isinstance(OwnedPackage, Package)
```

For a closer look at how to interact with EthPM packages using web3, check out the *examples page*.

Properties

Each `Package` exposes the following properties.

class `ethpm.`**Package**(*manifest: Dict[str, Any], w3:* Web3, *uri: str* | *None = None*)

`__repr__`() → str

String readable representation of the Package.

```
>>> OwnedPackage.__repr__()
'<Package owned==1.0.0>'
```

property name: str

The name of this `Package`.

```
>>> OwnedPackage.name
'owned'
```

property version: str

The package version of a `Package`.

```
>>> OwnedPackage.version
'1.0.0'
```

property manifest_version: str

The manifest version of a `Package`.

```
>>> OwnedPackage.manifest_version
'ethpm/3'
```

property uri: str | None

The uri (local file_path / content-addressed URI) of a `Package`'s manifest.

property contract_types: List[str]

All contract types included in this package.

build_dependencies

Return *Dependencies* instance containing the build dependencies available on this Package.

The Package class should provide access to the full dependency tree.

```
>>> owned_package.build_dependencies['zeppelin']
<ZeppelinPackage>
```

deployments

Returns a Deployments object containing all the deployment data and contract instances of a Package's *contract_types*. Automatically filters deployments to only expose those available on the current Package.w3 instance.

```
package.deployments.get_instance("ContractType")
```

Package.**w3**

The Web3 instance currently set on this Package. The deployments available on a package are automatically filtered to only contain those belonging to the currently set w3 instance.

Package.**manifest**

The manifest dict used to instantiate a Package.

Methods

Each Package exposes the following methods.

class ethpm.**Package**(*manifest: Dict[str, Any], w3: Web3, uri: str | None = None*)

update_w3(*w3: Web3*) → *Package*

Returns a new instance of *Package* containing the same manifest, but connected to a different web3 instance.

```
>>> new_w3 = Web3(Web3.EthereumTesterProvider())
>>> NewPackage = OwnedPackage.update_w3(new_w3)
>>> assert NewPackage.w3 == new_w3
>>> assert OwnedPackage.manifest == NewPackage.manifest
```

classmethod from_file(*file_path: Path, w3: Web3*) → *Package*

Returns a Package instantiated by a manifest located at the provided Path. file_path arg must be a pathlib.Path instance. A valid Web3 instance is required to instantiate a Package.

classmethod from_uri(*uri: URI, w3: Web3*) → *Package*

Returns a Package object instantiated by a manifest located at a content-addressed URI. A valid Web3 instance is also required. URI schemes supported:

- IPFS: *ipfs://Qm...*
- HTTP: *https://api.github.com/repos/:owner/:repo/git/blobs/:file_sha*
- Registry: *erc1319://registry.eth:1/greeter?version=1.0.0*

```
OwnedPackage = Package.from_uri('ipfs://
→QmbeVyFLSuEUxiXKwSsEjef7icpdTdA4kGG9BcrJXKNKUW', w3)  # noqa: E501
```

get_contract_factory(*name: ContractName*) → LinkableContract

Return the contract factory for a given contract type, generated from the data available in Package.manifest. Contract factories are accessible from the package class.

```
Owned = OwnedPackage.get_contract_factory('owned')
```

In cases where a contract uses a library, the contract factory will have unlinked bytecode. The `ethpm` package ships with its own subclass of `web3.contract.Contract`, `ethpm.contract.LinkableContract` with a few extra methods and properties related to bytecode linking.

```
>>> math = owned_package.contract_factories.math
>>> math.needs_bytecode_linking
True
>>> linked_math = math.link_bytecode({'MathLib': '0x1234...'})
>>> linked_math.needs_bytecode_linking
False
```

get_contract_instance(*name: ContractName*, *address: Address*) → Contract

Will return a `Web3.contract` instance generated from the contract type data available in `Package.manifest` and the provided `address`. The provided `address` must be valid on the connected chain available through `Package.w3`.

Validation

The `Package` class currently verifies the following things.

- Manifests used to instantiate a `Package` object conform to the EthPM V3 Manifest Specification and are tightly packed, with keys sorted alphabetically, and no trailing newline.

2.13.3 LinkableContract

Py-EthPM uses a custom subclass of `Web3.contract.Contract` to manage contract factories and instances which might require bytecode linking. To create a deployable contract factory, both the contract type's `abi` and `deploymentBytecode` must be available in the Package's manifest.

```
>>> from eth_utils import is_address
>>> from web3 import Web3
>>> from ethpm import Package, ASSETS_DIR

>>> w3 = Web3(Web3.EthereumTesterProvider())
>>> escrow_manifest_path = ASSETS_DIR / 'escrow' / 'with_bytecode_v3.json'

>>> # Try to deploy from unlinked factory
>>> EscrowPackage = Package.from_file(escrow_manifest_path, w3)
>>> EscrowFactory = EscrowPackage.get_contract_factory("Escrow")
>>> assert EscrowFactory.needs_bytecode_linking
>>> escrow_instance = EscrowFactory.constructor(w3.eth.accounts[0]).transact()
Traceback (most recent call last):
    ...
ethpm.exceptions.BytecodeLinkingError: Contract cannot be deployed until its bytecode is␣
↪linked.

>>> # Deploy SafeSendLib
>>> SafeSendFactory = EscrowPackage.get_contract_factory("SafeSendLib")
>>> safe_send_tx_hash = SafeSendFactory.constructor().transact()
>>> safe_send_tx_receipt = w3.eth.wait_for_transaction_receipt(safe_send_tx_hash)
```

(continues on next page)

(continued from previous page)

```
>>> # Link Escrow factory to deployed SafeSendLib instance
>>> LinkedEscrowFactory = EscrowFactory.link_bytecode({"SafeSendLib": safe_send_tx_
↪receipt.contractAddress})
>>> assert LinkedEscrowFactory.needs_bytecode_linking is False
>>> escrow_tx_hash = LinkedEscrowFactory.constructor(w3.eth.accounts[0]).transact()
>>> escrow_tx_receipt = w3.eth.wait_for_transaction_receipt(escrow_tx_hash)
>>> assert is_address(escrow_tx_receipt.contractAddress)
```

Properties

LinkableContract.**unlinked_references**

A list of link reference data for the deployment bytecode, if present in the manifest data used to generate a `LinkableContract` factory. Deployment bytecode link reference data must be present in a manifest in order to generate a factory for a contract which requires bytecode linking.

LinkableContract.**linked_references**

A list of link reference data for the runtime bytecode, if present in the manifest data used to generate a `LinkableContract` factory. If you want to use the *web3 Deployer* tool for a contract, then runtime bytecode link reference data must be present in a manifest.

LinkableContract.**needs_bytecode_linking**

A boolean attribute used to indicate whether a contract factory has unresolved link references, which must be resolved before a new contract instance can be deployed or instantiated at a given address.

Methods

classmethod LinkableContract.**link_bytecode**(*attr_dict*)

This method returns a newly created contract factory with the applied link references defined in the `attr_dict`. This method expects `attr_dict` to be of the type ``Dict[`contract_name`: `address`]`` for all link references that are unlinked.

2.13.4 URI Schemes and Backends

BaseURIBackend

`Py-EthPM` uses the `BaseURIBackend` as the parent class for all of its URI backends. To write your own backend, it must implement the following methods.

BaseURIBackend.**can_resolve_uri**(*uri*)

Return a bool indicating whether or not this backend is capable of resolving the given URI to a manifest. A content-addressed URI pointing to valid manifest is said to be capable of "resolving".

BaseURIBackend.**can_translate_uri**(*uri*)

Return a bool indicating whether this backend class can translate the given URI to a corresponding content-addressed URI. A registry URI is said to be capable of "translating" if it points to another content-addressed URI in its respective on-chain registry.

BaseURIBackend.**fetch_uri_contents**(*uri*)

Fetch the contents stored at the provided uri, if an available backend is capable of resolving the URI. Validates that contents stored at uri match the content hash suffixing the uri.

IPFS

Py-EthPM has multiple backends available to fetch/pin files to IPFS. The IPFS backends rely on the now-unmaintained ipfshttpclient library. Because of this, they are opt-in and may be installed via the ipfs web3 install extra.

```
$ pip install "web3[ipfs]"
```

The desired backend can be set via the environment variable: ETHPM_IPFS_BACKEND_CLASS.

- **InfuraIPFSBackend (default)**
  - *https://ipfs.infura.io*
- **IPFSGatewayBackend (temporarily deprecated)**
  - *https://ipfs.io/ipfs/*
- **LocalIPFSBacked**
  - Connect to a local IPFS API gateway running on port 5001.
- **DummyIPFSBackend**
  - Won't pin/fetch files to an actual IPFS node, but mocks out this behavior.

BaseIPFSBackend.**pin_assets**(*file_or_directory_path*)

Pin asset(s) found at the given path and returns the pinned asset data.

HTTPS

Py-EthPM offers a backend to fetch files from Github, GithubOverHTTPSBackend.

A valid content-addressed Github URI *must* conform to the following scheme, as described in ERC1319, to be used with this backend.

```
https://api.github.com/repos/:owner/:repo/git/blobs/:file_sha
```

create_content_addressed_github_uri(*uri*)

This util function will return a content-addressed URI, as defined by Github's blob scheme. To generate a content-addressed URI for any manifest stored on github, this function requires accepts a Github API uri that follows the following scheme.

```
https://api.github.com/repos/:owner/:repo/contents/:path/:to/manifest.json
```

```
>>> from ethpm.uri import create_content_addressed_github_uri

>>> owned_github_api_uri = "https://api.github.com/repos/ethpm/ethpm-spec/contents/
↪examples/owned/1.0.0.json"
>>> content_addressed_uri = "https://api.github.com/repos/ethpm/ethpm-spec/git/blobs/
↪8f9dc767d4c8b31fec4a08d9c0858d4f37b83180"

>>> actual_blob_uri = create_content_addressed_github_uri(owned_github_api_uri)
>>> assert actual_blob_uri == content_addressed_uri
```

Registry URIs

The URI to lookup a package from a registry should follow the following format. (subject to change as the Registry Contract Standard makes its way through the EIP process)

```
scheme://address:chain_id/package_name@version
```

- URI must be a string type
- `scheme`: (required) `ethpm` or `erc1319`
- `address`: (required) Must be a valid ENS domain or a valid checksum address pointing towards a registry contract.
- `chain_id`: Chain ID of the chain on which the registry lives. Defaults to Mainnet. Supported chains include…
- 1: Mainnet
- 5: Goerli
- 11155111: Sepolia
- `package-name`: Must conform to the package-name as specified in the EthPM-Spec.
- `version`: The URI escaped version string, *should* conform to the semver version numbering specification.

Examples…

- `ethpm://packages.zeppelinos.eth/owned@1.0.0`
- `ethpm://0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD:1/ethregistrar@1.0.0`

To specify a specific asset within a package, you can namespace the target asset.

- `ethpm://maker.snakecharmers.eth:1/dai-dai@1.0.0/sources/token.sol`
- `ethpm://maker.snakecharmers.eth:1/dai-dai@1.0.0/contractTypes/DSToken/abi`
- `ethpm://maker.snakecharmers.eth:1/dai-dai@1.0.0/deployments/mainnet/dai`

2.13.5 Builder

The manifest Builder is a tool designed to help construct custom manifests. The builder is still under active development, and can only handle simple use-cases for now.

To create a simple manifest

For all manifests, the following ingredients are *required*.

```
build(
    {},
    package_name(str),
    version(str),
    manifest_version(str), ...,
)
# Or
build(
    init_manifest(package_name: str, version: str, manifest_version: str="ethpm/3")
    ...,
)
```

The builder (i.e. `build()`) expects a dict as the first argument. This dict can be empty, or populated if you want to extend an existing manifest.

```
>>> from ethpm.tools.builder import *

>>> expected_manifest = {
...   "name": "owned",
...   "version": "1.0.0",
...   "manifest": "ethpm/3"
... }
>>> base_manifest = {"name": "owned"}
>>> built_manifest = build(
...     {},
...     package_name("owned"),
...     manifest_version("ethpm/3"),
...     version("1.0.0"),
... )
>>> extended_manifest = build(
...     base_manifest,
...     manifest_version("ethpm/3"),
...     version("1.0.0"),
... )
>>> assert built_manifest == expected_manifest
>>> assert extended_manifest == expected_manifest
```

With `init_manifest()`, which populates "manifest" with "ethpm/3" (the only supported EthPM specification version), unless provided with an alternative "version".

```
>>> build(
...     init_manifest("owned", "1.0.0"),
... )
{'name': 'owned', 'version': '1.0.0', 'manifest': 'ethpm/3'}
```

To return a Package

```
build(
    ...,
    as_package(w3: Web3),
)
```

By default, the manifest builder returns a dict representing the manifest. To return a `Package` instance (instantiated with the generated manifest) from the builder, add the `as_package()` builder function with a valid `web3` instance to the end of the builder.

```
>>> from ethpm import Package
>>> from web3 import Web3

>>> w3 = Web3(Web3.EthereumTesterProvider())
>>> built_package = build(
...     {},
...     package_name("owned"),
...     manifest_version("ethpm/3"),
```

(continues on next page)

(continued from previous page)

```
...     version("1.0.0"),
...     as_package(w3),
... )
>>> assert isinstance(built_package, Package)
```

To validate a manifest

```
build(
    ...,
    validate(),
)
```

By default, the manifest builder does *not* perform any validation that the generated fields are correctly formatted. There are two ways to validate that the built manifest conforms to the EthPM V3 Specification.

- Return a Package, which automatically runs validation.
- Add the `validate()` function to the end of the manifest builder.

```
>>> valid_manifest = build(
...     {},
...     package_name("owned"),
...     manifest_version("ethpm/3"),
...     version("1.0.0"),
...     validate(),
... )
>>> assert valid_manifest == {"name": "owned", "manifest": "ethpm/3", "version": "1.0.0"}
>>> invalid_manifest = build(
...     {},
...     package_name("_InvalidPkgName"),
...     manifest_version("ethpm/3"),
...     version("1.0.0"),
...     validate(),
... )
Traceback (most recent call last):
ethpm.exceptions.EthPMValidationError: Manifest invalid for schema version 2. Reason: '_
↪InvalidPkgName' does not match '^[a-z][-a-z0-9]{0,255}$'
```

To write a manifest to disk

```
build(
    ...,
    write_to_disk(
        manifest_root_dir: Optional[Path],
        manifest_name: Optional[str],
        prettify: Optional[bool],
    ),
)
```

Writes the active manifest to disk. Will not overwrite an existing manifest with the same name and root directory.

Defaults - Writes manifest to current working directory (as returned by `os.getcwd()`) unless a `Path` is provided as manifest_root_dir. - Writes manifest with a filename of `<version>.json` unless desired manifest name (which must end in ".json") is provided as manifest_name. - Writes the minified manifest version to disk unless prettify is set to True

```
>>> from pathlib import Path
>>> import tempfile
>>> p = Path(tempfile.mkdtemp("temp"))
>>> build(
...     {},
...     package_name("owned"),
...     manifest_version("ethpm/3"),
...     version("1.0.0"),
...     write_to_disk(manifest_root_dir=p, manifest_name="manifest.json", prettify=True),
... )
{'name': 'owned', 'manifest': 'ethpm/3', 'version': '1.0.0'}
>>> with open(str(p / "manifest.json")) as f:
...     actual_manifest = f.read()
>>> print(actual_manifest)
{
     "manifest": "ethpm/3",
     "name": "owned",
     "version": "1.0.0"
}
```

To pin a manifest to IPFS

```
build(
    ...,
    pin_to_ipfs(
        backend: BaseIPFSBackend,
        prettify: Optional[bool],
    ),
)
```

Pins the active manifest to disk. Must be the concluding function in a builder set since it returns the IPFS pin data rather than returning the manifest for further processing.

To add meta fields

```
build(
    ...,
    description(str),
    license(str),
    authors(*args: str),
    keywords(*args: str),
    links(*kwargs: str),
    ...,
)
```

```
>>> BASE_MANIFEST = {"name": "owned", "manifest": "ethpm/3", "version": "1.0.0"}
>>> expected_manifest = {
...   "name": "owned",
...   "manifest": "ethpm/3",
...   "version": "1.0.0",
...   "meta": {
...     "authors": ["Satoshi", "Nakamoto"],
...     "description": "An awesome package.",
...     "keywords": ["auth"],
...     "license": "MIT",
...     "links": {
...       "documentation": "www.readthedocs.com/...",
...       "repo": "www.github.com/...",
...       "website": "www.website.com",
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     authors("Satoshi", "Nakamoto"),
...     description("An awesome package."),
...     keywords("auth"),
...     license("MIT"),
...     links(documentation="www.readthedocs.com/...", repo="www.github.com/...",␣
↪website="www.website.com"),
... )
>>> assert expected_manifest == built_manifest
```

Compiler Output

To build a more complex manifest for solidity contracts, it is required that you provide standard-json output from the solidity compiler. Or for a more convenient experience, use the EthPM CLI.

Here is an example of how to compile the contracts and generate the standard-json output. More information can be found in the Solidity Compiler docs.

```
solc --allow-paths <path-to-contract-directory> --standard-json < standard-json-input.
↪json > owned_compiler_output.json
```

Sample standard-json-input.json

```
{
    "language": "Solidity",
    "sources": {
        "Contract.sol": {
            "urls": ["<path-to-contract>"]
        }
    },
    "settings": {
        "outputSelection": {
            "*": {
                "*": ["abi", "evm.bytecode.object"]
```

(continues on next page)

(continued from previous page)

```
            }
        }
    }
}
```

The compiler_output as used in the following examples is the entire value of the contracts key of the solc output, which contains compilation data for all compiled contracts.

To add a source

```
# To inline a source
build(
    ...,
    inline_source(
        contract_name: str,
        compiler_output: Dict[str, Any],
        package_root_dir: Optional[Path]
    ),
    ...,
)
# To pin a source
build(
    ...,
    pin_source(
        contract_name: str,
        compiler_output: Dict[str, Any],
        ipfs_backend: BaseIPFSBackend,
        package_root_dir: Optional[Path]
    ),
    ...,
)
```

There are two ways to include a contract source in your manifest.

Both strategies require that either ...

- The current working directory is set to the package root directory or
- The package root directory is provided as an argument (package_root_dir)

To inline the source code directly in the manifest, use inline_source() or source_inliner() (to inline multiple sources from the same compiler_output), which requires the contract name and compiler output as args.

Note: output_v3.json below is expected to be the standard-json output generated by the solidity compiler as described here. The output must contain the abi and bytecode objects from compilation.

```
>>> import json
>>> from ethpm import ASSETS_DIR, get_ethpm_spec_dir
>>> ethpm_spec_dir = get_ethpm_spec_dir()
>>> owned_dir = ethpm_spec_dir / "examples" / "owned" / "contracts"
>>> compiler_output = json.loads((ASSETS_DIR / "owned" / "output_v3.json").read_text())[
↪'contracts']
```

(continues on next page)

(continued from previous page)

```
>>> expected_manifest = {
...   "name": "owned",
...   "version": "1.0.0",
...   "manifest": "ethpm/3",
...   "sources": {
...     "./Owned.sol": {
...       "content": """// SPDX-License-Identifier: MIT\npragma solidity ^0.6.8;\n\
↪ncontract Owned """
...         """{\n    address owner;\n    \n    modifier onlyOwner { require(msg.sender ==␣
↪owner); _; }"""
...         """\n\n    constructor() public {\n        owner = msg.sender;\n    }\n}""",
...       "type": "solidity",
...       "installPath": "./Owned.sol"
...     }
...   }
... }
>>> # With `inline_source()`
>>> built_manifest = build(
...     BASE_MANIFEST,
...     inline_source("Owned", compiler_output, package_root_dir=owned_dir),
... )
>>> assert expected_manifest == built_manifest
>>> # With `source_inliner()` for multiple sources from the same compiler output
>>> inliner = source_inliner(compiler_output, package_root_dir=owned_dir)
>>> built_manifest = build(
...     BASE_MANIFEST,
...     inliner("Owned"),
...     # inliner("other_source"), etc...
... )
>>> assert expected_manifest == built_manifest
```

To include the source as a content-addressed URI, `Py-EthPM` can pin your source via the Infura IPFS API. As well as the contract name and compiler output, this function requires that you provide the desired IPFS backend to pin the contract sources.

```
>>> import json
>>> from ethpm import ASSETS_DIR, get_ethpm_spec_dir
>>> from ethpm.backends.ipfs import get_ipfs_backend
>>> ethpm_spec_dir = get_ethpm_spec_dir()
>>> owned_dir = ethpm_spec_dir / "examples" / "owned" / "contracts"
>>> compiler_output = json.loads((ASSETS_DIR / "owned" / "output_v3.json").read_text())[
↪'contracts']
>>> ipfs_backend = get_ipfs_backend()
>>> expected_manifest = {
...   "name": "owned",
...   "version": "1.0.0",
...   "manifest": "ethpm/3",
...   "sources": {
...     "./Owned.sol": {
...       "installPath": "./Owned.sol",
...       "type": "solidity",
...       "urls": ["ipfs://QmU8QUSt56ZoBDJgjjXvAZEPro9LmK1m2gjVG5Q4s9x29W"]
```

(continues on next page)

(continued from previous page)

```
...     }
...   }
... }
>>> # With `pin_source()`
>>> built_manifest = build(
...     BASE_MANIFEST,
...     pin_source("Owned", compiler_output, ipfs_backend, package_root_dir=owned_dir),
... )
>>> assert expected_manifest == built_manifest
>>> # With `source_pinner()` for multiple sources from the same compiler output
>>> pinner = source_pinner(compiler_output, ipfs_backend, package_root_dir=owned_dir)
>>> built_manifest = build(
...     BASE_MANIFEST,
...     pinner("Owned"),
...     # pinner("other_source"), etc
... )
>>> assert expected_manifest == built_manifest
```

To add a contract type

```
build(
    ...,
    contract_type(
        contract_name: str,
        compiler_output: Dict[str, Any],
        alias: Optional[str],
        abi: Optional[bool],
        compiler: Optional[bool],
        contract_type: Optional[bool],
        deployment_bytecode: Optional[bool],
        devdoc: Optional[bool],
        userdoc: Optional[bool],
        source_id: Optional[bool],
        runtime_bytecode: Optional[bool]
    ),
    ...,
)
```

The default behavior of the manifest builder's `contract_type()` function is to populate the manifest with all of the contract type data found in the `compiler_output`.

```
>>> expected_manifest = {
...   'name': 'owned',
...   'manifest': 'ethpm/3',
...   'version': '1.0.0',
...   'compilers': [
...     {'name': 'solc', 'version': '0.6.8+commit.0bbfe453', 'settings': {'optimize':␣
↪True}, 'contractTypes': ['Owned']}
...   ],
...   'contractTypes': {
...     'Owned': {
```

(continues on next page)

(continued from previous page)

```
...       'abi': [{'inputs': [], 'stateMutability': 'nonpayable', 'type': 'constructor'}
↪],
...       'deploymentBytecode': {
...         'bytecode':
↪'0x6080604052348015600f57600080fd5b5060008054600160016 0a01b03191633179055603f80602f6000396000f3fe60806
↪'
...       },
...       'sourceId': 'Owned.sol',
...       'devdoc': {'methods': {}},
...       'userdoc': {'methods': {}}
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     contract_type("Owned", compiler_output)
... )
>>> assert expected_manifest == built_manifest
```

To select only certain contract type data to be included in your manifest, provide the desired fields as `True` keyword arguments. The following fields can be specified for inclusion in the manifest ...

- `abi`
- `compiler`
- `deployment_bytecode`
- `runtime_bytecode`
- `devdoc`
- `userdoc`
- `source_id`

```
>>> expected_manifest = {
...   'name': 'owned',
...   'manifest': 'ethpm/3',
...   'version': '1.0.0',
...   'contractTypes': {
...     'Owned': {
...       'abi': [{'inputs': [], 'stateMutability': 'nonpayable', 'type': 'constructor'}
↪],
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     contract_type("Owned", compiler_output, abi=True)
... )
>>> assert expected_manifest == built_manifest
```

If you would like to alias your contract type, provide the desired alias as a kwarg. This will automatically include the original contract type in a `contractType` field. Unless specific contract type fields are provided as kwargs, `contractType` will still default to including all available contract type data found in the compiler output.

```
>>> expected_manifest = {
...   'name': 'owned',
...   'manifest': 'ethpm/3',
...   'version': '1.0.0',
...   'contractTypes': {
...     'OwnedAlias': {
...       'abi': [{'inputs': [], 'stateMutability': 'nonpayable', 'type': 'constructor'}
↪],
...       'contractType': 'Owned'
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     contract_type("Owned", compiler_output, alias="OwnedAlias", abi=True)
... )
>>> assert expected_manifest == built_manifest
```

To add a deployment

```
build(
    ...,
    deployment(
        block_uri,
        contract_instance,
        contract_type,
        address,
        transaction=None,
        block=None,
        deployment_bytecode=None,
        runtime_bytecode=None,
        compiler=None,
    ),
    ...,
)
```

There are two strategies for adding a deployment to your manifest.

deployment(*block_uri*, *contract_instance*, *contract_type*, *address*, *transaction=None*, *block=None*, *deployment_bytecode=None*, *runtime_bytecode=None*, *compiler=None*)

This is the simplest builder function for adding a deployment to a manifest. All arguments require keywords. This builder function requires a valid `block_uri`, it's up to the user to be sure that multiple chain URIs representing the same blockchain are not included in the "deployments" object keys.

`runtime_bytecode`, `deployment_bytecode` and `compiler` must all be validly formatted dicts according to the EthPM Spec. If your contract has link dependencies, be sure to include them in the bytecode objects.

```
>>> expected_manifest = {
...   'name': 'owned',
...   'manifest': 'ethpm/3',
...   'version': '1.0.0',
...   'deployments': {
```

(continues on next page)

(continued from previous page)

```
...     'blockchain://1234567890123456789012345678901234567890123456789012345678901234/
↪block/1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef': {
...       'Owned': {
...         'contractType': 'Owned',
...         'address': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
...       }
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     deployment(
...         block_uri='blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
...         contract_instance='Owned',
...         contract_type='Owned',
...         address='0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
...     ),
... )
>>> assert expected_manifest == built_manifest
```

deployment_type(*contract_instance*, *contract_type*, *deployment_bytecode=None*, *runtime_bytecode=None*, *compiler=None*)

This builder function simplifies adding the same contract type deployment across multiple chains. It requires both a `contract_instance` and `contract_type` argument (in many cases these are the same, though `contract_type` *must* always match its correspondent in the manifest's "contract_types") and all arguments require keywords.

`runtime_bytecode`, `deployment_bytecode` and `compiler` must all be validly formatted dicts according to the EthPM Spec. If your contract has link dependencies, be sure to include them in the bytecode objects.

```
owned_type = deployment_type(contract_instance="Owned", contract_type="Owned")
escrow_type = deployment_type(
    contract_instance = "Escrow",
    contract_type = "Escrow",
    deployment_bytecode = {
        "bytecode":
↪"0x608060405234801561001057600080fd5b5060405160208061045383398101604081815291516002819055336000818152
↪"
    },
    runtime_bytecode = {
        "bytecode":
↪"0x6080604052600436106100775763ffffffff7c0100000000000000000000000000000000000000000000000000000000060
↪"
    },
    compiler = {
        "name": "solc",
        "version": "0.4.24+commit.e67f0147.Emscripten.clang",
        "settings": {
            "optimize": True
        }
    }
```

(continues on next page)

(continued from previous page)

```
)
manifest = build(
    package_name("escrow"),
    version("1.0.0"),
    manifest_version("ethpm/3"),
    owned_type(
        block_uri='blockchain://
↪abcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcd/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=owned_testnet_address,
    ),
    owned_type(
        block_uri='blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=owned_mainnet_address,
        transaction=owned_mainnet_transaction,
        block=owned_mainnet_block,
    ),
    escrow_type(
        block_uri='blockchain://
↪abcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcd/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=escrow_testnet_address,
    ),
    escrow_type(
        block_uri='blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=escrow_mainnet_address,
        transaction=escrow_mainnet_transaction,
    ),
)
```

To add a build dependency

```
build(
    ...,
    build_dependency(
        package_name,
        uri,
    ),
    ...,
)
```

build_dependency(*package_name*, *uri*)

To add a build dependency to your manifest, just provide the package's name and a supported, content-addressed URI.

```
>>> expected_manifest = {
...   'name': 'owned',
```

(continues on next page)

(continued from previous page)

```
...   'manifest': 'ethpm/3',
...   'version': '1.0.0',
...   'buildDependencies': {
...     'owned': 'ipfs://QmbeVyFLSuEUxiXKwSsEjef6icpdTdA4kGG9BcrJXKNKUW',
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     build_dependency('owned', 'ipfs://QmbeVyFLSuEUxiXKwSsEjef6icpdTdA4kGG9BcrJXKNKUW
↪'),
... )
>>> assert expected_manifest == built_manifest
```

2.13.6 Checker

The manifest Checker is a tool designed to help validate manifests according to the natural language spec (link).

To validate a manifest

```
>>> from ethpm.tools.checker import check_manifest

>>> basic_manifest = {"name": "example", "version": "1.0.0", "manifest": "ethpm/3"}
>>> check_manifest(basic_manifest)
{'meta': "Manifest missing a suggested 'meta' field.", 'sources': 'Manifest is missing a␣
↪sources field, which defines a source tree that should comprise the full source tree␣
↪necessary to recompile the contracts contained in this release.', 'contractTypes':
↪"Manifest does not contain any 'contractTypes'. Packages should only include contract␣
↪types that can be found in the source files for this package. Packages should not␣
↪include contract types from dependencies. Packages should not include abstract␣
↪contracts in the contract types section of a release.", 'compilers': 'Manifest is␣
↪missing a suggested `compilers` field.'}
```

2.14 Ethereum Name Service (ENS)

The Ethereum Name Service (ENS) is analogous to the Domain Name Service. It enables users and developers to use human-friendly names in place of error-prone hexadecimal addresses, content hashes, and more.

The *ens* module is included with web3.py. It provides an interface to look up domains and addresses, add resolver records, or get and set metadata.

Note: web3.py `v6.6.0` introduced ENS name normalization standard ENSIP-15. This update to ENS name validation and normalization won't affect ~99% of names but may prevent invalid names from being created and from interacting with the ENS contracts via web3.py. We feel strongly that this change, though breaking, is in the best interest of our users as it ensures compatibility with the latest ENS standards.

2.14.1 Setup

Create an ENS object (named ns below) in one of three ways:

1. Automatic detection
2. Specify an instance of a *provider*
3. From an existing *web3.Web3* object

```
# automatic detection
from ens.auto import ns

# or, with a provider
from web3 import IPCProvider
from ens import ENS

provider = IPCProvider(...)
ns = ENS(provider)

# or, with a w3 instance
# Note: This inherits the w3 middlewares from the w3 instance and adds a stalecheck␣
↪middleware to the middleware onion.
# It also inherits the provider and codec from the w3 instance, as well as the ``strict_
↪bytes_type_checking`` flag value.
from ens import ENS
w3 = Web3(...)
ns = ENS.from_web3(w3)
```

Asynchronous support is available via the AsyncENS module:

```
from ens import AsyncENS

ns = AsyncENS(provider)
```

Note that an ens module instance is also available on the w3 instance. The first time it's used, web3.py will create the ens instance using ENS.from_web3(w3) or AsyncENS.from_web3(w3) as appropriate.

```
# instantiate w3 instance
from web3 import Web3, IPCProvider
w3 = Web3(IPCProvider(...))

# use the module
w3.ens.address('ethereum.eth')
```

ens.**strict_bytes_type_checking**

The ENS instance has a strict_bytes_type_checking flag that toggles the flag with the same name on the Web3 instance attached to the ENS instance. You may disable the stricter bytes type checking that is loaded by default using this flag. For more examples, see *Disabling Strict Checks for Bytes Types*

If instantiating a standalone ENS instance using ENS.from_web3(), the ENS instance will inherit the value of the flag on the Web3 instance at time of instantiation.

```
>>> from web3 import Web3, EthereumTesterProvider
>>> from ens import ENS
>>> w3 = Web3(EthereumTesterProvider())
```

(continues on next page)

(continued from previous page)

```
>>> assert w3.strict_bytes_type_checking  # assert strict by default
>>> w3.is_encodable('bytes2', b'1')
False

>>> w3.strict_bytes_type_checking = False
>>> w3.is_encodable('bytes2', b'1')  # zero-padded, so encoded to: b'1\x00'
True

>>> ns = ENS.from_web3(w3)
>>> # assert inherited from w3 at time of instantiation via ENS.from_web3()
>>> assert ns.strict_bytes_type_checking is False
>>> ns.w3.is_encodable('bytes2', b'1')
True

>>> # assert these are now separate instances
>>> ns.strict_bytes_type_checking = True
>>> ns.w3.is_encodable('bytes2', b'1')
False

>>> # assert w3 flag value remains
>>> assert w3.strict_bytes_type_checking is False
>>> w3.is_encodable('bytes2', b'1')
True
```

However, if accessing the `ENS` class via the `Web3` instance as a module (`w3.ens`), since all modules use the same `Web3` object reference under the hood (the parent `w3` object), changing the `strict_bytes_type_checking` flag value on `w3` also changes the flag state for `w3.ens.w3` and all modules.

```
>>> from web3 import Web3, EthereumTesterProvider
>>> w3 = Web3(EthereumTesterProvider())

>>> assert w3.strict_bytes_type_checking  # assert strict by default
>>> w3.is_encodable('bytes2', b'1')
False

>>> w3.strict_bytes_type_checking = False
>>> w3.is_encodable('bytes2', b'1')  # zero-padded, so encoded to: b'1\x00'
True

>>> assert w3 == w3.ens.w3  # assert same object
>>> assert not w3.ens.w3.strict_bytes_type_checking
>>> w3.ens.w3.is_encodable('bytes2', b'1')
True

>>> # sanity check on eth module as well
>>> assert not w3.eth.w3.strict_bytes_type_checking
>>> w3.eth.w3.is_encodable('bytes2', b'1')
True
```

2.14.2 Usage

Name Info

Get the Address for an ENS Name

```
from ens.auto import ns
eth_address = ns.address('ens.eth')
assert eth_address == '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7'
```

The ENS module has no opinion as to which **TLD (Top Level Domain)** you can use, but will not infer a TLD if it is not provided with the name.

Multichain Address Resolution

ENSIP-9 introduced multichain address resolution, allowing users to resolve addresses from different chains, specified by the coin type index from SLIP44. The address() method on the ENS class supports multichain address resolution via the coin_type keyword argument.

```
from ens.auto import ns
eth_address = ns.address('ens.eth', coin_type=60)  # ETH is coin_type 60
assert eth_address == '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7'
```

Get the ENS Name for an Address

```
domain = ns.name('0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7')

# name() also accepts the bytes version of the address
assert ns.name(b'\xfe\x89\xccz\xbb,A\x83h:\xb7\x16S\xc4\xcd\xc9\xb0-D\xb7') == domain

# confirm that the name resolves back to the address that you looked up:
assert ns.address(domain) == '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7'
```

Note: For accuracy, and as a recommendation from the ENS documentation on reverse resolution, the ENS module now verifies that the forward resolution matches the address with every call to get the name() for an address. This is the only sure way to know whether the reverse resolution is correct. Anyone can claim any name, only forward resolution implies that the owner of the name gave their stamp of approval.

Get the Owner of a Name

```
eth_address = ns.owner('exchange.eth')
```

Set Up Your Name and Address

Link a Name to an Address

You can set up your name so that `address()` will show the address it points to. In order to do so, you must already be the owner of the domain (or its parent).

```
ns.setup_address('ens.eth', '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7')
```

In the common case where you want to point the name to the owning address, you can skip the address.

```
ns.setup_address('ens.eth')
```

You can claim arbitrarily deep subdomains.

```
ns.setup_address('supreme.executive.power.derives.from.a.mandate.from.the.masses.ens.eth
↪')

# wait for the transaction to be mined, then:
assert (
    ns.address('supreme.executive.power.derives.from.a.mandate.from.the.masses.ens.eth')
    == '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7'
)
```

Warning: Gas costs scale up with the number of subdomains!

Multichain Address Support

ENSIP-9 introduced multichain address resolution, allowing users to resolve addresses from different chains, specified by the coin type index from SLIP44. The `setup_address()` method on the `ENS` class supports multichain address setup via the `coin_type` keyword argument.

```
from ens.auto import ns
ns.setup_address('ens.eth', coin_type=60)  # ETH is coin_type 60
assert ns.address('ens.eth', coin_type=60) == '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7
↪'
```

Link an Address to a Name

You can set up your address so that `name()` will show the name that points to it.

This is like Caller ID. It enables you and others to take an account and determine what name points to it. Sometimes this is referred to as "reverse" resolution. The ENS Reverse Resolver is used for this functionality.

```
ns.setup_name('ens.eth', '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7')
```

If you don't supply the address, `setup_name()` will assume you want the address returned by `address()`.

```
ns.setup_name('ens.eth')
```

If the name doesn't already point to an address, `setup_name()` will call `setup_address()` for you.

Wait for the transaction to be mined, then:

```
assert ns.name('0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7') == 'ens.eth'
```

Text Records

Set Text Metadata for an ENS Record

As the owner of an ENS record, you can add text metadata. A list of supported fields can be found in the ENS documentation. You'll need to setup the address first, and then the text can be set:

```
ns.setup_address('ens.eth', '0xFe89cc7aBB2C4183683ab71653C4cdc9B02D44b7')
ns.set_text('ens.eth', 'url', 'https://example.com')
```

A transaction dictionary can be passed as the last argument if desired:

```
transaction_dict = {'from': '0x123...'}
ns.set_text('ens.eth', 'url', 'https://example.com', transaction_dict)
```

If the transaction dictionary is not passed, sensible defaults will be used, and if a transaction dictionary is passed but does not have a `from` value, the default will be the `owner`.

Read Text Metadata for an ENS Record

Anyone can read the data from an ENS Record:

```
url = ns.get_text('ens.eth', 'url')
assert url == 'https://example.com'
```

Working With Resolvers

Get the Resolver for an ENS Record

You can get the resolver for an ENS name via the `resolver()` method.

```
>>> resolver = ns.resolver('ens.eth')
>>> resolver.address
'0x5B2063246F2191f18F2675ceDB8b28102e957458'
```

2.14.3 Wildcard Resolution Support

The ENS module supports Wildcard Resolution for resolvers that implement the ExtendedResolver interface as described in ENSIP-10. Resolvers that implement the extended resolver interface should return True when calling the supportsInterface() function with the extended resolver interface id "0x9061b923" and should resolve subdomains to a unique address.

2.15 Examples

Here are some common things you might want to do with web3.

2.15.1 Looking up blocks

Blocks can be looked up by either their number or hash using the `web3.eth.get_block` API. Block hashes should be in their hexadecimal representation. Block numbers

```
# get a block by number
>>> web3.eth.get_block(12345)
{
    'author': '0xad5C1768e5974C231b2148169da064e61910f31a',
    'difficulty': 735512610763,
    'extraData': '0x476574682f76312e302e302f6c696e75782f676f312e342e32',
    'gasLimit': 5000,
    'gasUsed': 0,
    'hash': '0x767c2bfb3bdee3f78676c1285cd757bcd5d8c272cef2eb30d9733800a78c0b6d',
    'logsBloom':
↪'0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
    'miner': '0xad5C1768e5974C231b2148169da064e61910f31a',
    'mixHash': '0x31d9ec7e3855aeba37fd92aa1639845e70b360a60f77f12eff530429ef8cfcba',
    'nonce': '0x549f882c5f356f85',
    'number': 12345,
    'parentHash': '0x4b3c1d7e65a507b62734feca1ee9f27a5379e318bd52ae62de7ba67dbeac66a3',
    'receiptsRoot': '0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421',
    'sealFields': ['0x31d9ec7e3855aeba37fd92aa1639845e70b360a60f77f12eff530429ef8cfcba',
    '0x549f882c5f356f85'],
    'sha3Uncles': '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
    'size': 539,
    'stateRoot': '0xca495e22ed6b88c61714d129dbc8c94f5bf966ac581c09a57c0a72d0e55e7286',
    'timestamp': 1438367030,
    'totalDifficulty': 3862140487204603,
    'transactions': [],
    'transactionsRoot':
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421',
    'uncles': [],
}

# get a block by it's hash
>>> web3.eth.get_block(
↪'0x767c2bfb3bdee3f78676c1285cd757bcd5d8c272cef2eb30d9733800a78c0b6d')
{...}
```

2.15.2 Getting the latest block

You can also retrieve the latest block using the string `'latest'` in the `web3.eth.get_block` API.

```
>>> web3.eth.get_block('latest')
{...}
```

If you want to know the latest block number you can use the `web3.eth.block_number` property.

```
>>> web3.eth.block_number
4194803
```

2.15.3 Checking the balance of an account

To find the amount of ether owned by an account, use the `get_balance()` method. At the time of writing, the account with the most ether has a public address of *0x742d35Cc6634C0532925a3b844Bc454e4438f44e*.

```
>>> web3.eth.get_balance('0x742d35Cc6634C0532925a3b844Bc454e4438f44e')
3841357360894980500000001
```

Note that this number is not denominated in ether, but instead in the smallest unit of value in Ethereum, wei. Read on to learn how to convert that number to ether.

2.15.4 Converting currency denominations

Web3 can help you convert between denominations. The following denominations are supported.

denomination	amount in wei
wei	1
kwei	1000
babbage	1000
femtoether	1000
mwei	1000000
lovelace	1000000
picoether	1000000
gwei	1000000000
shannon	1000000000
nanoether	1000000000
nano	1000000000
szabo	1000000000000
microether	1000000000000
micro	1000000000000
finney	1000000000000000
milliether	1000000000000000
milli	1000000000000000
ether	1000000000000000000
kether	1000000000000000000000
grand	1000000000000000000000
mether	1000000000000000000000000
gether	1000000000000000000000000000
tether	1000000000000000000000000000000

Picking up from the previous example, the largest account contained 3841357360894980500000001 wei. You can use the `from_wei()` method to convert that balance to ether (or another denomination).

```
>>> web3.from_wei(3841357360894980500000001, 'ether')
Decimal('3841357.360894980500000001')
```

To convert back to wei, you can use the inverse function, `to_wei()`. Note that Python's default floating point precision is insufficient for this use case, so it's necessary to cast the value to a Decimal if it isn't already.

```
>>> from decimal import Decimal
>>> web3.to_wei(Decimal('3841357.360894980500000001'), 'ether')
3841357360894980500000001
```

Best practice: If you need to work with multiple currency denominations, default to wei. A typical workflow may require a conversion from some denomination to wei, then from wei to whatever you need.

```
>>> web3.to_wei(Decimal('0.000000005'), 'ether')
5000000000
>>> web3.from_wei(5000000000, 'gwei')
Decimal('5')
```

2.15.5 Sending transactions

There are a few options for sending transactions:

- *send_transaction()*
- *send_raw_transaction()*
- Calling *transact()* on a contract function
- Utilizing *construct_sign_and_send_raw_middleware()*

For more context, see the *Sending Transactions* Guide.

2.15.6 Looking up transactions

You can look up transactions using the `web3.eth.get_transaction` function.

```
>>> web3.eth.get_transaction(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
{
    'blockHash': '0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd',
    'blockNumber': 46147,
    'condition': None,
    'creates': None,
    'from': '0xA1E4380A3B1f749673E270229993eE55F35663b4',
    'gas': 21000,
    'gasPrice': None,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'maxFeePerGas': 2000000000,
    'maxPriorityFeePerGas': 1000000000,
    'networkId': None,
    'nonce': 0,
    'publicKey':
↪'0x376fc429acc35e610f75b14bc96242b13623833569a5bb3d72c17be7e51da0bb58e48e2462a59897cead8ab88e78709f9d
↪',
    'r': '0x88ff6cf0fefd94db46111149ae4bfc179e9b94721fffd821d38d16464b3f71d0',
    'raw':
↪'0xf86780862d79883d2000825208945df9b87991262f6ba471f09758cde1c0fc1de734827a69801ca088ff6cf0fefd94db46
↪',
    's': '0x45e0aff800961cfce805daef7016b9b675c137a6a41a548f7b60a3484c06a33a',
    'standardV': '0x1',
    'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
    'transactionIndex': 0,
    'v': '0x1c',
```

(continues on next page)

(continued from previous page)

```
    'value': 31337,
}
```

If no transaction for the given hash can be found, this method will throw `web3.exceptions.TransactionNotFound`.

2.15.7 Looking up receipts

Transaction receipts can be retrieved using the `web3.eth.get_transaction_receipt` API.

```
>>> web3.eth.get_transaction_receipt(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
{
    'blockHash': '0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd',
    'blockNumber': 46147,
    'contractAddress': None,
    'cumulativeGasUsed': 21000,
    'gasUsed': 21000,
    'logs': [],
    'logsBloom':
↪'0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
    'root': '0x96a8e009d2b88b1483e6941e6812e32263b05683fac202abc622a3e31aed1957',
    'transactionHash':
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'transactionIndex': 0,
}
```

If no transaction for the given hash can be found, this method will throw `web3.exceptions.TransactionNotFound`.

2.15.8 Working with Contracts

Interacting with existing contracts

In order to use an existing contract, you'll need its deployed address and its ABI. Both can be found using block explorers, like Etherscan. Once you instantiate a contract instance, you can read data and execute transactions.

```
# Configure w3, e.g., w3 = Web3(...)
address = '0x1f9840a85d5aF5bf1D1762F925BDADdC4201F988'
abi = '[{"inputs":[{"internalType":"address","name":"account","type":"address"},{
↪"internalType":"address","name":"minter_","type":"address"},...'
contract_instance = w3.eth.contract(address=address, abi=abi)

# read state:
contract_instance.functions.storedValue().call()
# 42

# update state:
tx_hash = contract_instance.functions.updateValue(43).transact()
```

Deploying new contracts

Given the following solidity source file stored at `contract.sol`.

```
contract StoreVar {

    uint8 public _myVar;
    event MyEvent(uint indexed _var);

    function setVar(uint8 _var) public {
        _myVar = _var;
        emit MyEvent(_var);
    }

    function getVar() public view returns (uint8) {
        return _myVar;
    }

}
```

The following example demonstrates a few things:

- Compiling a contract from a sol file.
- Estimating gas costs of a transaction.
- Transacting with a contract function.
- Waiting for a transaction receipt to be mined.

```
import sys
import time
import pprint

from web3.providers.eth_tester import EthereumTesterProvider
from web3 import Web3
from eth_tester import PyEVMBackend
from solcx import compile_source

def compile_source_file(file_path):
   with open(file_path, 'r') as f:
      source = f.read()

   return compile_source(source,output_values=['abi','bin'])


def deploy_contract(w3, contract_interface):
    tx_hash = w3.eth.contract(
        abi=contract_interface['abi'],
        bytecode=contract_interface['bin']).constructor().transact()

    address = w3.eth.get_transaction_receipt(tx_hash)['contractAddress']
    return address


w3 = Web3(EthereumTesterProvider(PyEVMBackend()))
```

(continues on next page)

(continued from previous page)

```
contract_source_path = 'contract.sol'
compiled_sol = compile_source_file('contract.sol')

contract_id, contract_interface = compiled_sol.popitem()

address = deploy_contract(w3, contract_interface)
print(f'Deployed {contract_id} to: {address}\n')

store_var_contract = w3.eth.contract(address=address, abi=contract_interface["abi"])

gas_estimate = store_var_contract.functions.setVar(255).estimate_gas()
print(f'Gas estimate to transact with setVar: {gas_estimate}')

if gas_estimate < 100000:
    print("Sending transaction to setVar(255)\n")
    tx_hash = store_var_contract.functions.setVar(255).transact()
    receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
    print("Transaction receipt mined:")
    pprint.pprint(dict(receipt))
    print("\nWas transaction successful?")
    pprint.pprint(receipt["status"])
else:
    print("Gas cost exceeds 100000")
```

Output:

```
Deployed <stdin>:StoreVar to: 0xF2E246BB76DF876Cef8b38ae84130F4F55De395b

Gas estimate to transact with setVar: 45535

Sending transaction to setVar(255)

Transaction receipt mined:
{'blockHash': HexBytes(
↪'0x837609ad0a404718c131ac5157373662944b778250a507783349d4e78bd8ac84'),
 'blockNumber': 2,
 'contractAddress': None,
 'cumulativeGasUsed': 43488,
 'gasUsed': 43488,
 'logs': [AttributeDict({'type': 'mined', 'logIndex': 0, 'transactionIndex': 0,
↪'transactionHash': HexBytes(
↪'0x50aa3ba0673243f1e60f546a12ab364fc2f6603b1654052ebec2b83d4524c6d0'), 'blockHash':␣
↪HexBytes('0x837609ad0a404718c131ac5157373662944b778250a507783349d4e78bd8ac84'),
↪'blockNumber': 2, 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b', 'data': '0x
↪', 'topics': [HexBytes(
↪'0x6c2b4666ba8da5a95717621d879a77de725f3d816709b9cbe9f059b8f875e284'), HexBytes(
↪'0x00000000000000000000000000000000000000000000000000000000000000ff')]})],
 'status': 1,
 'transactionHash': HexBytes(
↪'0x50aa3ba0673243f1e60f546a12ab364fc2f6603b1654052ebec2b83d4524c6d0'),
 'transactionIndex': 0}
```

(continues on next page)

(continued from previous page)

```
Was transaction successful?
1
```

2.15.9 Working with Contracts via ethPM

Warning: The `ethPM` module is no longer being maintained and will be deprecated with `web3.py` version 7.

ethPM packages contain configured contracts ready for use. Web3's `ethpm` module (`web3.pm`) extends Web3's native `Contract` module, with a few modifications for how you instantiate `Contract` factories and instances.

All you need is the package name, version and ethPM registry address for the package you wish to use. An ethPM registry is an on-chain datastore for the release data associated with an ethPM package. You can find some sample registries to explore in the ethPM registry. Remember, you should only use packages from registries whose maintainer you trust not to inject malicious code!

In this example we will use the `ethregistrar@3.0.0` package sourced from the `ens.snakecharmers.eth` registry.

`web3.pm` uses the `Package` class to represent an ethPM package. This object houses all of the contract assets within a package, and exposes them via an API. So, before we can interact with our package, we need to generate it as a `Package` instance.

```
# Note. To use the web3.pm module, you will need to instantiate your w3 instance
# with a web3 provider connected to the chain on which your registry lives.
from web3 import Web3, IPCProvider
w3 = Web3(IPCProvider(...))

# The ethPM module is still experimental and subject to change,
# so for now we need to enable it via a temporary flag.
w3.enable_unstable_package_management_api()

# Then we need to set the registry address that we want to use.
# This should be an ENS address, but can also be a checksummed contract address.
w3.pm.set_registry("ens.snakecharmers.eth")

# This generates a Package instance of the target ethPM package.
ens_package = w3.pm.get_package("ethregistrar", "3.0.0")
```

Now that we have a `Package` representation of our target ethPM package, we can generate contract factories and instances from this `Package`. However, it's important to note that some packages might be missing the necessary contract assets needed to generate an instance or a factory. You can use the ethPM CLI to figure out the available contract types and deployments within an ethPM package.

```
# To interact with a deployment located in an ethPM package.
# Note. This will only expose deployments located on the
# chain of the connected provider (in this example, mainnet)
mainnet_registrar = ens_package.deployments.get_instance("BaseRegistrarImplementation")

# Now you can treat mainnet_registrar like any other Web3 Contract instance!
mainnet_registrar.caller.balanceOf("0x123...")
> 0
```

(continues on next page)

(continued from previous page)

```
mainnet_registrar.functions.approve("0x123", 100000).transact()
> 0x123abc...  # tx_hash

# To create a contract factory from a contract type located in an ethPM package.
registrar_factory = ens_package.get_contract_factory("BaseRegistrarImplementation")

# Now you can treat registrar_factory like any other Web3 Contract factory to deploy new␣
↪instances!
# Note. This will deploy new instances to the chain of the connected provider (in this␣
↪example, mainnet)
registrar_factory.constructor(...).transact()
> 0x456def...  # tx_hash

# To connect your Package to a new chain - simply pass it a new Web3 instance
# connected to your provider of choice. Now your factories will automatically
# deploy to this new chain, and the deployments available on a package will
# be automatically filtered to those located on the new chain.
goerli_registrar = ens_package.update_w3(goerli_w3_instance)
```

2.15.10 Working with an ERC20 Token Contract

Most fungible tokens on the Ethereum blockchain conform to the ERC20 standard. This section of the guide covers interacting with an existing token contract which conforms to this standard.

In this guide we will interact with an existing token contract that we have already deployed to a local testing chain. This guide assumes:

1. An existing token contract at a known address.
2. Access to the proper `ABI` for the given contract.
3. A `web3.main.Web3` instance connected to a provider with an unlocked account which can send transactions.

Creating the contract factory

First we need to create a contract instance with the address of our token contract and the `ERC20` ABI.

```
>>> contract = w3.eth.contract(contract_address, abi=ABI)
>>> contract.address
'0xF2E246BB76DF876Cef8b38ae84130F4F55De395b'
```

Querying token metadata

Each token will have a total supply which represents the total number of tokens in circulation. In this example we've initialized the token contract to have 1 million tokens. Since this token contract is setup to have 18 decimal places, the raw total supply returned by the contract is going to have 18 additional decimal places.

```
>>> contract.functions.name().call()
'TestToken'
>>> contract.functions.symbol().call()
```

(continues on next page)

(continued from previous page)

```
'TEST'
>>> decimals = contract.functions.decimals().call()
>>> decimals
18
>>> DECIMALS = 10 ** decimals
>>> contract.functions.totalSupply().call() // DECIMALS
1000000
```

Query account balances

Next we can query some account balances using the contract's `balanceOf` function. The token contract we are using starts with a single account which we'll refer to as `alice` holding all of the tokens.

```
>>> alice = '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf'
>>> bob = '0x2B5AD5c4795c026514f8317c7a215E218DcCD6cF'
>>> raw_balance = contract.functions.balanceOf(alice).call()
>>> raw_balance
1000000000000000000000000
>>> raw_balance // DECIMALS
1000000
>>> contract.functions.balanceOf(bob).call()
0
```

Sending tokens

Next we can transfer some tokens from `alice` to `bob` using the contract's `transfer` function.

```
>>> tx_hash = contract.functions.transfer(bob, 100).transact({'from': alice})
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> contract.functions.balanceOf(alice).call()
999999999999999999999900
>>> contract.functions.balanceOf(bob).call()
100
```

Creating an approval for external transfers

Alice could also *approve* someone else to spend tokens from her account using the `approve` function. We can also query how many tokens we're approved to spend using the `allowance` function.

```
>>> contract.functions.allowance(alice, bob).call()
0
>>> tx_hash = contract.functions.approve(bob, 200).transact({'from': alice})
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> contract.functions.allowance(alice, bob).call()
200
```

Performing an external transfer

When someone has an allowance they can transfer those tokens using the `transferFrom` function.

```
>>> contract.functions.allowance(alice, bob).call()
200
>>> contract.functions.balanceOf(bob).call()
100
>>> tx_hash = contract.functions.transferFrom(alice, bob, 75).transact({'from': bob})
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
>>> contract.functions.allowance(alice, bob).call()
125
>>> contract.functions.balanceOf(bob).call()
175
```

2.15.11 CCIP Read support for offchain lookup

Contract calls support CCIP Read by default, via a `ccip_read_enabled` flag on the call and, more globally, a `global_ccip_read_enabled` flag on the provider. The following should work by default without raising an `OffchainLookup` and instead handling it appropriately as per the specification outlined in EIP-3668.

```
myContract.functions.revertsWithOffchainLookup(myData).call()
```

If the offchain lookup requires the user to send a transaction rather than make a call, this may be handled appropriately in the following way:

```
from web3 import Web3, WebsocketProvider
from web3.utils import handle_offchain_lookup

w3 = Web3(WebsocketProvider(...))

myContract = w3.eth.contract(address=...)
myData = b'data for offchain lookup function call'

# preflight with an `eth_call` and handle the exception
try:
    myContract.functions.revertsWithOffchainLookup(myData).call(ccip_read_enabled=False)
except OffchainLookup as ocl:
    tx = {'to': myContract.address, 'from': my_account}
    data_for_callback_function = handle_offchain_lookup(ocl.payload)
    tx['data'] = data_for_callback_function

    # send the built transaction with `eth_sendTransaction` or sign and send with `eth_
↪sendRawTransaction`
    tx_hash = w3.eth.send_transaction(tx)
```

2.15.12 Contract Unit Tests in Python

Here is an example of how one can use the pytest framework in python, web3.py, eth-tester, and PyEVM to perform unit tests entirely in python without any additional need for a full featured ethereum node/client. To install needed dependencies you can use the pinned extra for eth_tester in web3 and pytest:

```
$ pip install web3[tester] pytest
```

Once you have an environment set up for testing, you can then write your tests like so:

```
# of how to write unit tests with web3.py
import pytest

import pytest_asyncio

from web3 import (
    EthereumTesterProvider,
    Web3,
)
from web3.eth import (
    AsyncEth,
)
from web3.providers.eth_tester.main import (
    AsyncEthereumTesterProvider,
)


@pytest.fixture
def tester_provider():
    return EthereumTesterProvider()


@pytest.fixture
def eth_tester(tester_provider):
    return tester_provider.ethereum_tester


@pytest.fixture
def w3(tester_provider):
    return Web3(tester_provider)


@pytest.fixture
def foo_contract(eth_tester, w3):
    # For simplicity of this example we statically define the
    # contract code here. You might read your contracts from a
    # file, or something else to test with in your own code
    #
    # pragma solidity^0.5.3;
    #
    # contract Foo {
    #
    #     string public bar;
    #     event barred(string _bar);
```

(continues on next page)

(continued from previous page)

```
    #
    #     constructor() public {
    #         bar = "hello world";
    #     }
    #
    #     function setBar(string memory _bar) public {
    #         bar = _bar;
    #         emit barred(_bar);
    #     }
    #
    # }

    deploy_address = eth_tester.get_accounts()[0]

    abi = """[{"anonymous":false,"inputs":[{"indexed":false,"name":"_bar","type":"string
↪"}],"name":"barred","type":"event"},{"constant":false,"inputs":[{"name":"_bar","type":
↪"string"}],"name":"setBar","outputs":[],"payable":false,"stateMutability":"nonpayable",
↪"type":"function"},{"inputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"constructor"},{"constant":true,"inputs":[],"name":"bar","outputs":[{"name":"","type":
↪"string"}],"payable":false,"stateMutability":"view","type":"function"}]"""  # noqa:␣
↪E501
    # This bytecode is the output of compiling with
    # solc version:0.5.3+commit.10d17f24.Emscripten.clang
    bytecode = ""
↪"608060405234801561001057600080fd5b5060408051908101604052806000b81526020017f68656c6c6f20776f726c640000
↪"""  # noqa: E501

    # Create our contract class.
    FooContract = w3.eth.contract(abi=abi, bytecode=bytecode)
    # issue a transaction to deploy the contract.
    tx_hash = FooContract.constructor().transact(
        {
            "from": deploy_address,
        }
    )
    # wait for the transaction to be mined
    tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash, 180)
    # instantiate and return an instance of our contract.
    return FooContract(tx_receipt.contractAddress)


def test_initial_greeting(foo_contract):
    hw = foo_contract.caller.bar()
    assert hw == "hello world"


def test_can_update_greeting(w3, foo_contract):
    # send transaction that updates the greeting
    tx_hash = foo_contract.functions.setBar("testing contracts is easy").transact(
        {
            "from": w3.eth.accounts[1],
        }
```

(continues on next page)

(continued from previous page)

```
    )
    w3.eth.wait_for_transaction_receipt(tx_hash, 180)

    # verify that the contract is now using the updated greeting
    hw = foo_contract.caller.bar()
    assert hw == "testing contracts is easy"


def test_updating_greeting_emits_event(w3, foo_contract):
    # send transaction that updates the greeting
    tx_hash = foo_contract.functions.setBar("testing contracts is easy").transact(
        {
            "from": w3.eth.accounts[1],
        }
    )
    receipt = w3.eth.wait_for_transaction_receipt(tx_hash, 180)

    # get all of the `barred` logs for the contract
    logs = foo_contract.events.barred.get_logs()
    assert len(logs) == 1

    # verify that the log's data matches the expected value
    event = logs[0]
    assert event.blockHash == receipt.blockHash
    assert event.args._bar == "testing contracts is easy"


@pytest.fixture
def async_eth_tester():
    return AsyncEthereumTesterProvider().ethereum_tester


@pytest_asyncio.fixture()
async def async_w3():
    provider = AsyncEthereumTesterProvider()
    w3 = Web3(provider, modules={"eth": [AsyncEth]}, middlewares=provider.middlewares)
    w3.eth.default_account = await w3.eth.coinbase
    return w3


@pytest_asyncio.fixture()
async def async_foo_contract(async_w3):
    # For simplicity of this example we statically define the
    # contract code here. You might read your contracts from a
    # file, or something else to test with in your own code
    #
    # pragma solidity^0.5.3;
    #
    # contract Foo {
    #
    #     string public bar;
    #     event barred(string _bar);
```

(continues on next page)

(continued from previous page)

```
    #
    #     constructor() public {
    #         bar = "hello world";
    #     }
    #
    #     function setBar(string memory _bar) public {
    #         bar = _bar;
    #         emit barred(_bar);
    #     }
    #
    # }

    async_eth_tester_accounts = await async_w3.eth.accounts
    deploy_address = async_eth_tester_accounts[0]

    abi = """[{"anonymous":false,"inputs":[{"indexed":false,"name":"_bar","type":"string
↪"}],"name":"barred","type":"event"},{"constant":false,"inputs":[{"name":"_bar","type":
↪"string"}],"name":"setBar","outputs":[],"payable":false,"stateMutability":"nonpayable",
↪"type":"function"},{"inputs":[],"payable":false,"stateMutability":"nonpayable","type":
↪"constructor"},{"constant":true,"inputs":[],"name":"bar","outputs":[{"name":"","type":
↪"string"}],"payable":false,"stateMutability":"view","type":"function"}]"""  # noqa:␣
↪E501
    # This bytecode is the output of compiling with
    # solc version:0.5.3+commit.10d17f24.Emscripten.clang
    bytecode = ""
↪"608060405234801561001057600080fd5b5060408051908101604052806000b81526020017f68656c6c6f20776f726c6400000
↪"""  # noqa: E501

    # Create our contract class.
    FooContract = async_w3.eth.contract(abi=abi, bytecode=bytecode)
    # issue a transaction to deploy the contract.
    tx_hash = await FooContract.constructor().transact(
        {
            "from": deploy_address,
        }
    )
    # wait for the transaction to be mined
    tx_receipt = await async_w3.eth.wait_for_transaction_receipt(tx_hash, 180)
    # instantiate and return an instance of our contract.
    return FooContract(tx_receipt.contractAddress)


@pytest.mark.asyncio
async def test_async_initial_greeting(async_foo_contract):
    hw = await async_foo_contract.caller.bar()
    assert hw == "hello world"


@pytest.mark.asyncio
async def test_async_can_update_greeting(async_w3, async_foo_contract):
    async_eth_tester_accounts = await async_w3.eth.accounts
    # send transaction that updates the greeting
```

(continues on next page)

(continued from previous page)

```
    tx_hash = await async_foo_contract.functions.setBar(
        "testing contracts is easy",
    ).transact(
        {
            "from": async_eth_tester_accounts[1],
        }
    )
    await async_w3.eth.wait_for_transaction_receipt(tx_hash, 180)

    # verify that the contract is now using the updated greeting
    hw = await async_foo_contract.caller.bar()
    assert hw == "testing contracts is easy"


@pytest.mark.asyncio
async def test_async_updating_greeting_emits_event(async_w3, async_foo_contract):
    async_eth_tester_accounts = await async_w3.eth.accounts
    # send transaction that updates the greeting
    tx_hash = await async_foo_contract.functions.setBar(
        "testing contracts is easy",
    ).transact(
        {
            "from": async_eth_tester_accounts[1],
        }
    )
    receipt = await async_w3.eth.wait_for_transaction_receipt(tx_hash, 180)

    # get all of the `barred` logs for the contract
    logs = await async_foo_contract.events.barred.get_logs()
    assert len(logs) == 1

    # verify that the log's data matches the expected value
    event = logs[0]
    assert event.blockHash == receipt.blockHash
    assert event.args._bar == "testing contracts is easy"
```

2.15.13 Using Infura Goerli Node

Import your required libraries

```
from web3 import Web3, HTTPProvider
```

Initialize a web3 instance with an Infura node

```
w3 = Web3(Web3.HTTPProvider("https://goerli.infura.io/v3/YOUR_INFURA_KEY"))
```

Inject the middleware into the middleware onion

```
from web3.middleware import geth_poa_middleware
w3.middleware_onion.inject(geth_poa_middleware, layer=0)
```

Just remember that you have to sign all transactions locally, as infura does not handle any keys from your wallet (refer to this)

```
transaction = contract.functions.function_Name(params).build_transaction()
transaction.update({ 'gas' : appropriate_gas_amount })
transaction.update({ 'nonce' : w3.eth.get_transaction_count('Your_Wallet_Address') })
signed_tx = w3.eth.account.sign_transaction(transaction, private_key)
```

P.S : the two updates are done to the transaction dictionary, since a raw transaction might not contain gas & nonce amounts, so you have to add them manually.

And finally, send the transaction

```
txn_hash = w3.eth.send_raw_transaction(signed_tx.rawTransaction)
txn_receipt = w3.eth.wait_for_transaction_receipt(txn_hash)
```

Tip : afterwards you can use the value stored in `txn_hash`, in an explorer like etherscan to view the transaction's details

2.15.14 Adjusting log levels

web3.py internally uses Python logging subsystem.

If you want to run your application logging in debug mode, below is an example of how to make some JSON-RPC traffic quieter.

```
import logging
import coloredlogs

def setup_logging(log_level=logging.DEBUG):
    """Setup root logger and quiet some levels."""
    logger = logging.getLogger()

    # Set log format to display the logger name to hunt down verbose logging modules
    fmt = "%(name)-25s %(levelname)-8s %(message)s"

    # Use colored logging output for console with the coloredlogs package
    # https://pypi.org/project/coloredlogs/
    coloredlogs.install(level=log_level, fmt=fmt, logger=logger)

    # Disable logging of JSON-RPC requests and replies
    logging.getLogger("web3.RequestManager").setLevel(logging.WARNING)
    logging.getLogger("web3.providers.HTTPProvider").setLevel(logging.WARNING)
    # logging.getLogger("web3.RequestManager").propagate = False

    # Disable all internal debug logging of requests and urllib3
    # E.g. HTTP traffic
    logging.getLogger("requests").setLevel(logging.WARNING)
    logging.getLogger("urllib3").setLevel(logging.WARNING)

    return logger
```

2.15.15 Advanced example: Fetching all token transfer events

In this example, we show how to fetch all events of a certain event type from the Ethereum blockchain. There are three challenges when working with a large set of events:

- How to incrementally update an existing database of fetched events
- How to deal with interruptions in long running processes
- How to deal with *eth_getLogs* JSON-RPC call query limitations
- How to handle Ethereum minor chain reorganisations in (near) real-time data

eth_getLogs limitations

Ethereum JSON-RPC API servers, like Geth, do not provide easy to paginate over events, only over blocks. There's no request that can find the first block with an event or how many events occur within a range of blocks. The only feedback the JSON-RPC service will give you is whether the *eth_getLogs* call failed.

In this example script, we provide two kinds of heuristics to deal with this issue. The script scans events in a chunk of blocks (start block number - end block number). Then it uses two methods to find how many events there are likely to be in a block window:

- Dynamically set the block range window size, while never exceeding a threshold (e.g., 10,000 blocks).
- In the case *eth_getLogs* JSON-PRC call gives a timeout error, decrease the end block number and try again with a smaller block range window.

Example code

The following example code is divided into a reusable `EventScanner` class and then a demo script that:

- fetches all transfer events for RCC token,
- can incrementally run again to check if there are new events,
- handles interruptions (e.g., CTRL+C abort) gracefully,
- writes all `Transfer` events in a single file JSON database, so that other process can consume them,
- uses the tqdm library for progress bar output in a console,
- only supports `HTTPS` providers, because JSON-RPC retry logic depends on the implementation details of the underlying protocol,
- disables the standard `http_retry_request_middleware` because it does not know how to handle the shrinking block range window for `eth_getLogs`, and
- consumes around 20k JSON-RPC API calls.

The script can be run with: `python ./eventscanner.py <your JSON-RPC API URL>`.

```
"""A stateful event scanner for Ethereum-based blockchains using web3.py.

With the stateful mechanism, you can do one batch scan or incremental scans,
where events are added wherever the scanner left off.
"""

import datetime
import time
```

(continues on next page)

(continued from previous page)

```
import logging
from abc import ABC, abstractmethod
from typing import Tuple, Optional, Callable, List, Iterable, Dict, Any

from web3 import Web3
from web3.contract import Contract
from web3.datastructures import AttributeDict
from web3.exceptions import BlockNotFound
from eth_abi.codec import ABICodec

# Currently this method is not exposed over official web3 API,
# but we need it to construct eth_getLogs parameters
from web3._utils.filters import construct_event_filter_params
from web3._utils.events import get_event_data


logger = logging.getLogger(__name__)


class EventScannerState(ABC):
    """Application state that remembers what blocks we have scanned in the case of crash.
    """

    @abstractmethod
    def get_last_scanned_block(self) -> int:
        """Number of the last block we have scanned on the previous cycle.

        :return: 0 if no blocks scanned yet
        """

    @abstractmethod
    def start_chunk(self, block_number: int):
        """Scanner is about to ask data of multiple blocks over JSON-RPC.

        Start a database session if needed.
        """

    @abstractmethod
    def end_chunk(self, block_number: int):
        """Scanner finished a number of blocks.

        Persistent any data in your state now.
        """

    @abstractmethod
    def process_event(self, block_when: datetime.datetime, event: AttributeDict) ->␣
↪object:
        """Process incoming events.

        This function takes raw events from Web3, transforms them to your application␣
↪internal
        format, then saves them in a database or some other state.
```

(continues on next page)

(continued from previous page)

```
        :param block_when: When this block was mined

        :param event: Symbolic dictionary of the event data

        :return: Internal state structure that is the result of event transformation.
        """

    @abstractmethod
    def delete_data(self, since_block: int) -> int:
        """Delete any data since this block was scanned.

        Purges any potential minor reorg data.
        """


class EventScanner:
    """Scan blockchain for events and try not to abuse JSON-RPC API too much.

    Can be used for real-time scans, as it detects minor chain reorganisation and␣
↪rescans.
    Unlike the easy web3.contract.Contract, this scanner can scan events from multiple␣
↪contracts at once.
    For example, you can get all transfers from all tokens in the same scan.

    You *should* disable the default `http_retry_request_middleware` on your provider for␣
↪Web3,
    because it cannot correctly throttle and decrease the `eth_getLogs` block number␣
↪range.
    """

    def __init__(self, w3: Web3, contract: Contract, state: EventScannerState, events:␣
↪List, filters: Dict[str, Any],
                 max_chunk_scan_size: int = 10000, max_request_retries: int = 30,␣
↪request_retry_seconds: float = 3.0):
        """
        :param contract: Contract
        :param events: List of web3 Event we scan
        :param filters: Filters passed to get_logs
        :param max_chunk_scan_size: JSON-RPC API limit in the number of blocks we query.␣
↪(Recommendation: 10,000 for mainnet, 500,000 for testnets)
        :param max_request_retries: How many times we try to reattempt a failed JSON-RPC␣
↪call
        :param request_retry_seconds: Delay between failed requests to let JSON-RPC␣
↪server to recover
        """

        self.logger = logger
        self.contract = contract
        self.w3 = w3
        self.state = state
        self.events = events
```

(continues on next page)

(continued from previous page)

```
        self.filters = filters

        # Our JSON-RPC throttling parameters
        self.min_scan_chunk_size = 10  # 12 s/block = 120 seconds period
        self.max_scan_chunk_size = max_chunk_scan_size
        self.max_request_retries = max_request_retries
        self.request_retry_seconds = request_retry_seconds

        # Factor how fast we increase the chunk size if results are found
        # # (slow down scan after starting to get hits)
        self.chunk_size_decrease = 0.5

        # Factor how fast we increase chunk size if no results found
        self.chunk_size_increase = 2.0

    @property
    def address(self):
        return self.token_address

    def get_block_timestamp(self, block_num) -> datetime.datetime:
        """Get Ethereum block timestamp"""
        try:
            block_info = self.w3.eth.get_block(block_num)
        except BlockNotFound:
            # Block was not mined yet,
            # minor chain reorganisation?
            return None
        last_time = block_info["timestamp"]
        return datetime.datetime.utcfromtimestamp(last_time)

    def get_suggested_scan_start_block(self):
        """Get where we should start to scan for new token events.

        If there are no prior scans, start from block 1.
        Otherwise, start from the last end block minus ten blocks.
        We rescan the last ten scanned blocks in the case there were forks to avoid
        misaccounting due to minor single block works (happens once in an hour in␣
↪Ethereum).
        These heuristics could be made more robust, but this is for the sake of simple␣
↪reference implementation.
        """

        end_block = self.get_last_scanned_block()
        if end_block:
            return max(1, end_block - self.NUM_BLOCKS_RESCAN_FOR_FORKS)
        return 1

    def get_suggested_scan_end_block(self):
        """Get the last mined block on Ethereum chain we are following."""

        # Do not scan all the way to the final block, as this
        # block might not be mined yet
```

(continues on next page)

(continued from previous page)

```
        return self.w3.eth.block_number - 1

    def get_last_scanned_block(self) -> int:
        return self.state.get_last_scanned_block()

    def delete_potentially_forked_block_data(self, after_block: int):
        """Purge old data in the case of blockchain reorganisation."""
        self.state.delete_data(after_block)

    def scan_chunk(self, start_block, end_block) -> Tuple[int, datetime.datetime, list]:
        """Read and process events between to block numbers.

        Dynamically decrease the size of the chunk if the case JSON-RPC server pukes out.

        :return: tuple(actual end block number, when this block was mined, processed␣
↪events)
        """

        block_timestamps = {}
        get_block_timestamp = self.get_block_timestamp

        # Cache block timestamps to reduce some RPC overhead
        # Real solution might include smarter models around block
        def get_block_when(block_num):
            if block_num not in block_timestamps:
                block_timestamps[block_num] = get_block_timestamp(block_num)
            return block_timestamps[block_num]

        all_processed = []

        for event_type in self.events:

            # Callable that takes care of the underlying web3 call
            def _fetch_events(_start_block, _end_block):
                return _fetch_events_for_all_contracts(self.w3,
                                                       event_type,
                                                       self.filters,
                                                       from_block=_start_block,
                                                       to_block=_end_block)

            # Do `n` retries on `eth_getLogs`,
            # throttle down block range if needed
            end_block, events = _retry_web3_call(
                _fetch_events,
                start_block=start_block,
                end_block=end_block,
                retries=self.max_request_retries,
                delay=self.request_retry_seconds)

            for evt in events:
                idx = evt["logIndex"]  # Integer of the log index position in the block,␣
↪null when its pending
```

(continues on next page)

(continued from previous page)

```
                # We cannot avoid minor chain reorganisations, but
                # at least we must avoid blocks that are not mined yet
                assert idx is not None, "Somehow tried to scan a pending block"

                block_number = evt["blockNumber"]

                # Get UTC time when this event happened (block mined timestamp)
                # from our in-memory cache
                block_when = get_block_when(block_number)

                logger.debug(f"Processing event {evt['event']}, block: {evt['blockNumber
↪']} count: {evt['blockNumber']}")
                processed = self.state.process_event(block_when, evt)
                all_processed.append(processed)

        end_block_timestamp = get_block_when(end_block)
        return end_block, end_block_timestamp, all_processed

    def estimate_next_chunk_size(self, current_chuck_size: int, event_found_count: int):
        """Try to figure out optimal chunk size

        Our scanner might need to scan the whole blockchain for all events

        * We want to minimize API calls over empty blocks

        * We want to make sure that one scan chunk does not try to process too many␣
↪entries once, as we try to control commit buffer size and potentially asynchronous␣
↪busy loop

        * Do not overload node serving JSON-RPC API by asking data for too many events␣
↪at a time

        Currently Ethereum JSON-API does not have an API to tell when a first event␣
↪occurred in a blockchain
        and our heuristics try to accelerate block fetching (chunk size) until we see␣
↪the first event.

        These heuristics exponentially increase the scan chunk size depending on if we␣
↪are seeing events or not.
        When any transfers are encountered, we are back to scanning only a few blocks at␣
↪a time.
        It does not make sense to do a full chain scan starting from block 1, doing one␣
↪JSON-RPC call per 20 blocks.
        """

        if event_found_count > 0:
            # When we encounter first events, reset the chunk size window
            current_chuck_size = self.min_scan_chunk_size
        else:
            current_chuck_size *= self.chunk_size_increase
```

(continues on next page)

(continued from previous page)

```
        current_chuck_size = max(self.min_scan_chunk_size, current_chuck_size)
        current_chuck_size = min(self.max_scan_chunk_size, current_chuck_size)
        return int(current_chuck_size)

    def scan(self, start_block, end_block, start_chunk_size=20, progress_
↪callback=Optional[Callable]) -> Tuple[
        list, int]:
        """Perform a token balances scan.

        Assumes all balances in the database are valid before start_block (no forks␣
↪sneaked in).

        :param start_block: The first block included in the scan

        :param end_block: The last block included in the scan

        :param start_chunk_size: How many blocks we try to fetch over JSON-RPC on the␣
↪first attempt

        :param progress_callback: If this is an UI application, update the progress of␣
↪the scan

        :return: [All processed events, number of chunks used]
        """

        assert start_block <= end_block

        current_block = start_block

        # Scan in chunks, commit between
        chunk_size = start_chunk_size
        last_scan_duration = last_logs_found = 0
        total_chunks_scanned = 0

        # All processed entries we got on this scan cycle
        all_processed = []

        while current_block <= end_block:

            self.state.start_chunk(current_block, chunk_size)

            # Print some diagnostics to logs to try to fiddle with real world JSON-RPC␣
↪API performance
            estimated_end_block = current_block + chunk_size
            logger.debug(
                f"Scanning token transfers for blocks: {current_block} - {estimated_end_
↪block}, chunk size {chunk_size}, last chunk scan took {last_scan_duration}, last logs␣
↪found {last_logs_found}"
            )

            start = time.time()
            actual_end_block, end_block_timestamp, new_entries = self.scan_chunk(current_
```

(continues on next page)

(continued from previous page)

```
↪block, estimated_end_block)

            # Where does our current chunk scan ends - are we out of chain yet?
            current_end = actual_end_block

            last_scan_duration = time.time() - start
            all_processed += new_entries

            # Print progress bar
            if progress_callback:
                progress_callback(start_block, end_block, current_block, end_block_
↪timestamp, chunk_size, len(new_entries))

            # Try to guess how many blocks to fetch over `eth_getLogs` API next time
            chunk_size = self.estimate_next_chunk_size(chunk_size, len(new_entries))

            # Set where the next chunk starts
            current_block = current_end + 1
            total_chunks_scanned += 1
            self.state.end_chunk(current_end)

        return all_processed, total_chunks_scanned


def _retry_web3_call(func, start_block, end_block, retries, delay) -> Tuple[int, list]:
    """A custom retry loop to throttle down block range.

    If our JSON-RPC server cannot serve all incoming `eth_getLogs` in a single request,
    we retry and throttle down block range for every retry.

    For example, Go Ethereum does not indicate what is an acceptable response size.
    It just fails on the server-side with a "context was cancelled" warning.

    :param func: A callable that triggers Ethereum JSON-RPC, as func(start_block, end_
↪block)
    :param start_block: The initial start block of the block range
    :param end_block: The initial start block of the block range
    :param retries: How many times we retry
    :param delay: Time to sleep between retries
    """
    for i in range(retries):
        try:
            return end_block, func(start_block, end_block)
        except Exception as e:
            # Assume this is HTTPConnectionPool(host='localhost', port=8545): Read timed␣
↪out. (read timeout=10)
            # from Go Ethereum. This translates to the error "context was cancelled" on␣
↪the server side:
            # https://github.com/ethereum/go-ethereum/issues/20426
            if i < retries - 1:
                # Give some more verbose info than the default middleware
                logger.warning(
```

(continues on next page)

(continued from previous page)

```
                    f"Retrying events for block range {start_block} - {end_block} ({end_
↪block-start_block}) failed with {e} , retrying in {delay} seconds")
                # Decrease the `eth_getBlocks` range
                end_block = start_block + ((end_block - start_block) // 2)
                # Let the JSON-RPC to recover e.g. from restart
                time.sleep(delay)
                continue
            else:
                logger.warning("Out of retries")
                raise


def _fetch_events_for_all_contracts(
        w3,
        event,
        argument_filters: Dict[str, Any],
        from_block: int,
        to_block: int) -> Iterable:
    """Get events using eth_getLogs API.

    This method is detached from any contract instance.

    This is a stateless method, as opposed to create_filter.
    It can be safely called against nodes which do not provide `eth_newFilter` API, like␣
↪Infura.
    """

    if from_block is None:
        raise TypeError("Missing mandatory keyword argument to get_logs: from_block")

    # Currently no way to poke this using a public web3.py API.
    # This will return raw underlying ABI JSON object for the event
    abi = event._get_event_abi()

    # Depending on the Solidity version used to compile
    # the contract that uses the ABI,
    # it might have Solidity ABI encoding v1 or v2.
    # We just assume the default that you set on Web3 object here.
    # More information here https://eth-abi.readthedocs.io/en/latest/index.html
    codec: ABICodec = w3.codec

    # Here we need to poke a bit into Web3 internals, as this
    # functionality is not exposed by default.
    # Construct JSON-RPC raw filter presentation based on human readable Python␣
↪descriptions
    # Namely, convert event names to their keccak signatures
    # More information here:
    # https://github.com/ethereum/web3.py/blob/e176ce0793dafdd0573acc8d4b76425b6eb604ca/
↪web3/_utils/filters.py#L71
    data_filter_set, event_filter_params = construct_event_filter_params(
        abi,
        codec,
```

(continues on next page)

(continued from previous page)

```
        address=argument_filters.get("address"),
        argument_filters=argument_filters,
        fromBlock=from_block,
        toBlock=to_block
    )

    logger.debug(f"Querying eth_getLogs with the following parameters: {event_filter_
↪params}")

    # Call JSON-RPC API on your Ethereum node.
    # get_logs() returns raw AttributedDict entries
    logs = w3.eth.get_logs(event_filter_params)

    # Convert raw binary data to Python proxy objects as described by ABI
    all_events = []
    for log in logs:
        # Convert raw JSON-RPC log result to human readable event by using ABI data
        # More information how process_log works here
        # https://github.com/ethereum/web3.py/blob/
↪fbaf1ad11b0c7fac09ba34baff2c256cffe0a148/web3/_utils/events.py#L200
        evt = get_event_data(codec, abi, log)
        # Note: This was originally yield,
        # but deferring the timeout exception caused the throttle logic not to work
        all_events.append(evt)
    return all_events


if __name__ == "__main__":
    # Simple demo that scans all the token transfers of RCC token (11k).
    # The demo supports persistent state by using a JSON file.
    # You will need an Ethereum node for this.
    # Running this script will consume around 20k JSON-RPC calls.
    # With locally running Geth, the script takes 10 minutes.
    # The resulting JSON state file is 2.9 MB.
    import sys
    import json
    from web3.providers.rpc import HTTPProvider

    # We use tqdm library to render a nice progress bar in the console
    # https://pypi.org/project/tqdm/
    from tqdm import tqdm

    # RCC has around 11k Transfer events
    # https://etherscan.io/token/0x9b6443b0fb9c241a7fdac375595cea13e6b7807a
    RCC_ADDRESS = "0x9b6443b0fb9c241a7fdac375595cea13e6b7807a"

    # Reduced ERC-20 ABI, only Transfer event
    ABI = """[
        {
            "anonymous": false,
            "inputs": [
                {
```

(continues on next page)

(continued from previous page)

```
                "indexed": true,
                "name": "from",
                "type": "address"
            },
            {
                "indexed": true,
                "name": "to",
                "type": "address"
            },
            {
                "indexed": false,
                "name": "value",
                "type": "uint256"
            }
        ],
        "name": "Transfer",
        "type": "event"
    }
]
"""

class JSONifiedState(EventScannerState):
    """Store the state of scanned blocks and all events.

    All state is an in-memory dict.
    Simple load/store massive JSON on start up.
    """

    def __init__(self):
        self.state = None
        self.fname = "test-state.json"
        # How many second ago we saved the JSON file
        self.last_save = 0

    def reset(self):
        """Create initial state of nothing scanned."""
        self.state = {
            "last_scanned_block": 0,
            "blocks": {},
        }

    def restore(self):
        """Restore the last scan state from a file."""
        try:
            self.state = json.load(open(self.fname, "rt"))
            print(f"Restored the state, previously {self.state['last_scanned_block']}
↪ blocks have been scanned")
        except (IOError, json.decoder.JSONDecodeError):
            print("State starting from scratch")
            self.reset()

    def save(self):
```

(continues on next page)

(continued from previous page)

```
        """Save everything we have scanned so far in a file."""
        with open(self.fname, "wt") as f:
            json.dump(self.state, f)
        self.last_save = time.time()

    #
    # EventScannerState methods implemented below
    #

    def get_last_scanned_block(self):
        """The number of the last block we have stored."""
        return self.state["last_scanned_block"]

    def delete_data(self, since_block):
        """Remove potentially reorganised blocks from the scan data."""
        for block_num in range(since_block, self.get_last_scanned_block()):
            if block_num in self.state["blocks"]:
                del self.state["blocks"][block_num]

    def start_chunk(self, block_number, chunk_size):
        pass

    def end_chunk(self, block_number):
        """Save at the end of each block, so we can resume in the case of a crash or␣
↪CTRL+C"""
        # Next time the scanner is started we will resume from this block
        self.state["last_scanned_block"] = block_number

        # Save the database file for every minute
        if time.time() - self.last_save > 60:
            self.save()

    def process_event(self, block_when: datetime.datetime, event: AttributeDict) ->␣
↪str:
        """Record a ERC-20 transfer in our database."""
        # Events are keyed by their transaction hash and log index
        # One transaction may contain multiple events
        # and each one of those gets their own log index

        # event_name = event.event # "Transfer"
        log_index = event.logIndex  # Log index within the block
        # transaction_index = event.transactionIndex  # Transaction index within the␣
↪block
        txhash = event.transactionHash.hex()  # Transaction hash
        block_number = event.blockNumber

        # Convert ERC-20 Transfer event to our internal format
        args = event["args"]
        transfer = {
            "from": args["from"],
            "to": args.to,
            "value": args.value,
```

(continues on next page)

(continued from previous page)

```
                "timestamp": block_when.isoformat(),
            }

            # Create empty dict as the block that contains all transactions by txhash
            if block_number not in self.state["blocks"]:
                self.state["blocks"][block_number] = {}

            block = self.state["blocks"][block_number]
            if txhash not in block:
                # We have not yet recorded any transfers in this transaction
                # (One transaction may contain multiple events if executed by a smart␣
↪contract).
                # Create a tx entry that contains all events by a log index
                self.state["blocks"][block_number][txhash] = {}

            # Record ERC-20 transfer in our database
            self.state["blocks"][block_number][txhash][log_index] = transfer

            # Return a pointer that allows us to look up this event later if needed
            return f"{block_number}-{txhash}-{log_index}"

    def run():

        if len(sys.argv) < 2:
            print("Usage: eventscanner.py http://your-node-url")
            sys.exit(1)

        api_url = sys.argv[1]

        # Enable logs to the stdout.
        # DEBUG is very verbose level
        logging.basicConfig(level=logging.INFO)

        provider = HTTPProvider(api_url)

        # Remove the default JSON-RPC retry middleware
        # as it correctly cannot handle eth_getLogs block range
        # throttle down.
        provider.middlewares.clear()

        w3 = Web3(provider)

        # Prepare stub ERC-20 contract object
        abi = json.loads(ABI)
        ERC20 = w3.eth.contract(abi=abi)

        # Restore/create our persistent state
        state = JSONifiedState()
        state.restore()

        # chain_id: int, w3: Web3, abi: Dict, state: EventScannerState, events: List,␣
↪filters: Dict, max_chunk_scan_size: int=10000
```

(continues on next page)

(continued from previous page)

```
        scanner = EventScanner(
            w3=w3,
            contract=ERC20,
            state=state,
            events=[ERC20.events.Transfer],
            filters={"address": RCC_ADDRESS},
            # How many maximum blocks at the time we request from JSON-RPC
            # and we are unlikely to exceed the response size limit of the JSON-RPC␣
↪server
            max_chunk_scan_size=10000
        )

        # Assume we might have scanned the blocks all the way to the last Ethereum block
        # that mined a few seconds before the previous scan run ended.
        # Because there might have been a minor Ethereum chain reorganisations
        # since the last scan ended, we need to discard
        # the last few blocks from the previous scan results.
        chain_reorg_safety_blocks = 10
        scanner.delete_potentially_forked_block_data(state.get_last_scanned_block() -␣
↪chain_reorg_safety_blocks)

        # Scan from [last block scanned] - [latest ethereum block]
        # Note that our chain reorg safety blocks cannot go negative
        start_block = max(state.get_last_scanned_block() - chain_reorg_safety_blocks, 0)
        end_block = scanner.get_suggested_scan_end_block()
        blocks_to_scan = end_block - start_block

        print(f"Scanning events from blocks {start_block} - {end_block}")

        # Render a progress bar in the console
        start = time.time()
        with tqdm(total=blocks_to_scan) as progress_bar:
            def _update_progress(start, end, current, current_block_timestamp, chunk_
↪size, events_count):
                if current_block_timestamp:
                    formatted_time = current_block_timestamp.strftime("%d-%m-%Y")
                else:
                    formatted_time = "no block time available"
                progress_bar.set_description(f"Current block: {current} ({formatted_time}
↪), blocks in a scan batch: {chunk_size}, events processed in a batch {events_count}")
                progress_bar.update(chunk_size)

            # Run the scan
            result, total_chunks_scanned = scanner.scan(start_block, end_block, progress_
↪callback=_update_progress)

        state.save()
        duration = time.time() - start
        print(f"Scanned total {len(result)} Transfer events, in {duration} seconds,␣
↪total {total_chunks_scanned} chunk scans performed")

    run()
```

2.16 Troubleshooting

2.16.1 Set up a clean environment

Many things can cause a broken environment. You might be on an unsupported version of Python. Another package might be installed that has a name or version conflict. Often, the best way to guarantee a correct environment is with `virtualenv`, like:

```
# Install pip if it is not available:
$ which pip || curl https://bootstrap.pypa.io/get-pip.py | python

# Install virtualenv if it is not available:
$ which virtualenv || pip install --upgrade virtualenv

# *If* the above command displays an error, you can try installing as root:
$ sudo pip install virtualenv

# Create a virtual environment:
$ virtualenv -p python3 ~/.venv-py3

# Activate your new virtual environment:
$ source ~/.venv-py3/bin/activate

# With virtualenv active, make sure you have the latest packaging tools
$ pip install --upgrade pip setuptools

# Now we can install web3.py...
$ pip install --upgrade web3
```

Note: Remember that each new terminal session requires you to reactivate your virtualenv, like: `$ source ~/.venv-py3/bin/activate`

2.16.2 Why can't I use a particular function?

Note that a web3.py instance must be configured before you can use most of its capabilities. One symptom of not configuring the instance first is an error that looks something like this: `AttributeError: type object 'Web3' has no attribute 'eth'`.

To properly configure your web3.py instance, specify which provider you're using to connect to the Ethereum network. An example configuration, if you're connecting to a locally run node, might be:

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.HTTPProvider('http://localhost:8545'))

# now `w3` is available to use:
>>> w3.is_connected()
True
>>> w3.eth.send_transaction(...)
```

Refer to the *Providers* documentation for further help with configuration.

2.16.3 Why isn't my web3 instance connecting to the network?

You can check that your instance is connected via the `is_connected` method:

```
>>> w3.is_connected()
False
```

There are a variety of explanations for why you may see `False` here. To help you diagnose the problem, `is_connected` has an optional `show_traceback` argument:

```
>>> w3.is_connected(show_traceback=True)
# this is an example, your error may differ

# <long stack trace output>
ProviderConnectionError: Problem connecting to provider with error: <class
↪'FileNotFoundError'>: cannot connect to IPC socket at path: None
```

If you're running a local node, such as Geth, double-check that you've indeed started the binary and that you've started it from the intended directory - particularly if you've specified a relative path to its ipc file.

If that does not address your issue, it's probable that you still have a Provider configuration issue. There are several options for configuring a Provider, detailed *here*.

2.16.4 How do I use my MetaMask accounts from web3.py?

Often you don't need to do this, just make a new account in web3.py, and transfer funds from your MetaMask account into it. But if you must...

Export your private key from MetaMask, and use the local private key tools in web3.py to sign and send transactions.

See how to export your private key and *Working with Local Private Keys*.

2.16.5 How do I get ether for my test network?

Test networks usually have something called a "faucet" to help get test ether to people who want to use it. The faucet simply sends you test ether when you visit a web page, or ping a chat bot, etc.

Each test network has its own version of test ether, so each one must maintain its own faucet. If you're not sure which test network to use, see *Which network should I connect to?*

Faucet mechanisms tend to come and go, so if any information here is out of date, try the Ethereum Stackexchange. Here are some links to testnet ether instructions (in no particular order):

- Goerli (different faucet links on top menu bar)
- Sepolia

2.16.6 Why can't I create an account?

If you're seeing the error `The method personal_newAccount does not exist/is not available`, you may be trying to create an account while connected to a remote node provider, like Infura. As a matter of security, remote nodes cannot create accounts.

If you are in fact running a local node, make sure that it's properly configured to accept `personal` methods. For Geth, that looks something like: `--http.api personal,eth,<etc>` or `--ws.api personal,eth,<etc>` depending on your configuration. Note that the IPC configuration is most secure and includes the `personal` API by default.

In general, your options for accounts are:

- Run a node (e.g., Geth) locally, connect to it via the local port, then use the `personal` API.
- Import a keystore file for an account and *extract the private key*.
- Create an account via the *eth-account* API, e.g., `new_acct = w3.eth.account.create()`.
- Use an external service (e.g., MyCrypto) to generate a new account, then securely import its private key.

Warning: Don't store real value in an account until you are familiar with security best practices. If you lose your private key, you lose your account!

2.16.7 Making Ethereum JSON-RPC API access faster

Your Ethereum node JSON-RPC API might be slow when fetching multiple and large requests, especially when running batch jobs. Here are some tips for how to speed up your web3.py application.

- Run your client locally, e.g., Go Ethereum or TurboGeth. The network latency and speed are the major limiting factors for fast API access.
- Use IPC communication instead of HTTP/WebSockets. See *Choosing How to Connect to Your Node*.
- Use an optimised JSON decoder. A future iteration of web3.py may change the default decoder or provide an API to configure one, but for now, you may patch the provider class to use ujson.

```
"""JSON-RPC decoding optimised for web3.py"""

from typing import cast

import ujson

from web3.providers import JSONBaseProvider
from web3.types import RPCResponse


def _fast_decode_rpc_response(raw_response: bytes) -> RPCResponse:
    decoded = ujson.loads(raw_response)
    return cast(RPCResponse, decoded)


def patch_provider(provider: JSONBaseProvider):
    """Monkey-patch web3.py provider for faster JSON decoding.

    Call this on your provider after construction.
```

(continues on next page)

(continued from previous page)

```
    This greatly improves JSON-RPC API access speeds, when fetching
    multiple and large responses.
    """
    provider.decode_rpc_response = _fast_decode_rpc_response
```

2.16.8 Why am I getting Visual C++ or Cython not installed error?

Some Windows users that do not have Microsoft Visual C++ version 14.0 or greater installed may see an error message when installing web3.py as shown below:

```
error: Microsoft Visual C++ 14.0 or greater is required. Get it with "Microsoft C++␣
↪Build Tools": https://visualstudio.microsoft.com/visual-cpp-build-tools/
```

To fix this error, download and install Microsoft Visual C++ from here :

Microsoft Visual C++ Redistributable for Visual Studio

- x64 Visual C++
- x86 Visual C++
- ARM64 Visual C++

2.17 Migrating your code from v5 to v6

web3.py follows Semantic Versioning, which means that version 6 introduced backwards-incompatible changes. If your project depends on web3.py v6, then you'll probably need to make some changes.

Breaking Changes:

2.17.1 Strict Bytes Checking by Default

web3.py v6 moved to requiring strict bytes checking by default. This means that if an ABI specifies a `bytes4` argument, web3.py will invalidate any entry that is not encodable as a bytes type with length of 4. This means only 0x-prefixed hex strings with a length of 4 and bytes types with a length of 4 will be considered valid. This removes doubt that comes from inferring values and assuming they should be padded.

This behavior was previously available in via the `w3.enable_strict_bytes_checking()` method. This is now, however, a toggleable flag on the `Web3` instance via the `w3.strict_bytes_type_checking` property. As outlined above, this property is set to `True` by default but can be toggled on and off via the property's setter (e.g. `w3.strict_bytes_type_checking = False`).

2.17.2 Snake Case

web3.py v6 moved to the more Pythonic convention of snake_casing wherever possible. There are some exceptions to this pattern:

- Contract methods and events use whatever is listed in the ABI. If the smart contract convention is to use camelCase for method and event names, web3.py won't do any magic to convert it to snake_casing.
- Arguments to JSON-RPC methods. For example: transaction and filter parameters still use camelCasing. The reason for this is primarily due to error messaging. It would be confusing to pass in a snake_cased parameter and get an error message with a camelCased parameter.
- Data that is returned from JSON-RPC methods. For example: The keys in a transaction receipt will still be returned as camelCase.

2.17.3 Python 3.10 and 3.11 Support

Support for Python 3.10 and 3.11 is here. In order to support Python 3.10, we had to update the Websockets dependency to v10+.

2.17.4 Exceptions

Exceptions inherit from a base class

In v5, some web3.py exceptions inherited from `AttributeError`, namely:

- `NoABIFunctionsFound`
- `NoABIFound`
- `NoABIEventsFound`

Others inherited from `ValueError`, namely:

- `InvalidAddress`
- `NameNotFound`
- `LogTopicError`
- `InvalidEventABI`

Now web3.py exceptions inherit from the same base `Web3Exception`.

As such, any code that was expecting a `ValueError` or an `AttributeError` from web3.py must update to expecting one of the exceptions listed above, or `Web3Exception`.

Similarly, exceptions raised in the EthPM and ENS modules inherit from the base `EthPMException` and `ENSException`, respectively.

ValidationError

The Python dev tooling ecosystem is moving towards standardizing `ValidationError`, so users know that they're catching the correct `ValidationError`. The base `ValidationError` is imported from `eth_utils`. However, we also wanted to empower users to catch all errors emitted by a particular module. So we now have a `Web3ValidationError`, `EthPMValidationError`, and an `ENSValidationError` that all inherit from the generic `eth_utils.exceptions.ValidationError`.

Web3 class split into Web3 and AsyncWeb3

The *Web3* class previously contained both sync and async methods. We've separated *Web3* and *AsyncWeb3* functionality to tighten up typing. For example:

```
from web3 import Web3, AsyncWeb3

w3 = Web3(Web3.HTTPProvider(<provider.url>))
async_w3 = AsyncWeb3(AsyncWeb3.AsyncHTTPProvider(<provider.url>))
```

dict to *AttributeDict* conversion moved to middleware

Eth module data returned as key-value pairs was previously automatically converted to an *AttributeDict* by result formatters, which could cause problems with typing. This conversion has been moved to a default *attrdict_middleware* where it can be easily removed if necessary. See the Eth module docs for more detail.

Other Misc Changes

- `InfuraKeyNotFound` exception has been changed to `InfuraProjectIdNotFound`
- `SolidityError` has been removed in favor of `ContractLogicError`
- When a method is unavailable from a node provider (i.e. a response error code of -32601 is returned), a `MethodUnavailable` error is now raised instead of `ValueError`
- Logs' *data* field was previously formatted with *to_ascii_if_bytes*, now formatted to *HexBytes*
- Receipts' *type* field was previously not formatted, now formatted with *to_integer_if_hex*

2.17.5 Removals

- Removed unused IBAN module
- Removed `WEB3_INFURA_API_KEY` environment variable in favor of `WEB3_INFURA_PROJECT_ID`
- Removed Kovan auto provider
- Removed deprecated `sha3` and `soliditySha3` methods in favor of `keccak` and `solidityKeccak`
- Remove Parity Module and References

2.17.6 Other notable changes

- The `ipfshttpclient` library is now opt-in via a web3 install extra. This only affects the ethpm ipfs backends, which rely on the library.

2.18 Migrating your code from v4 to v5

Web3.py follows Semantic Versioning, which means that version 5 introduced backwards-incompatible changes. If your project depends on Web3.py v4, then you'll probably need to make some changes.

Here are the most common required updates:

2.18.1 Python 3.5 no longer supported

You will need to upgrade to either Python 3.6 or 3.7

2.18.2 `eth-abi` v1 no longer supported

You will need to upgrade the `eth-abi` dependency to v2

2.18.3 Changes to base API

JSON-RPC Updates

In v4, JSON-RPC calls that looked up transactions or blocks and didn't find them, returned `None`. Now if a transaction or block is not found, a `BlockNotFound` or a `TransactionNotFound` error will be thrown as appropriate. This applies to the following web3 methods:

- `getTransaction()` will throw a `TransactionNotFound` error
- `getTransactionReceipt()` will throw a `TransactionNotFound` error
- `getTransactionByBlock()` will throw a `TransactionNotFound` error
- `getTransactionCount()` will throw a `BlockNotFound` error
- `getBlock()` will throw a `BlockNotFound` error
- `getUncleCount()` will throw a `BlockNotFound` error
- `getUncleByBlock()` will throw a `BlockNotFound` error

Removed Methods

- `contract.buildTransaction` was removed for `contract.functions.buildTransaction.<method name>`
- `contract.deploy` was removed for `contract.constructor.transact`
- `contract.estimateGas` was removed for `contract.functions.<method name>.estimateGas`
- `contract.call` was removed for `contract.<functions/events>.<method name>.call`
- `contract.transact` was removed for `contract.<functions/events>.<method name>.transact`

- `contract.eventFilter` was removed for `contract.events.<event name>.createFilter`
- `middleware_stack` was renamed to `middleware_onion()`
- `web3.miner.hashrate` was a duplicate of *hashrate()* and was removed.
- `web3.version.network` was a duplicate of `version()` and was removed.
- `web3.providers.tester.EthereumTesterProvider` and `web3.providers.tester.TestRPCProvider` have been removed for *EthereumTesterProvider()*
- `web3.eth.enableUnauditedFeatures` was removed
- `web3.txpool` was moved to *txpool()*
- `web3.version.node` was removed for `web3.clientVersion`
- `web3.version.ethereum` was removed for `protocolVersion()`
- Relocated personal RPC endpoints to reflect Parity and Geth implementations:
  - `web3.personal.listAccounts` was removed for `listAccounts()` or `listAccounts()`
  - `web3.personal.importRawKey` was removed for `importRawKey()` or `importRawKey()`
  - `web3.personal.newAccount` was removed for `newAccount()` or `newAccount()`
  - `web3.personal.lockAccount` was removed for `lockAccount()`
  - `web3.personal.unlockAccount` was removed for `unlockAccount()` or `unlockAccount()`
  - `web3.personal.sendTransaction` was removed for `sendTransaction()` or `sendTransaction()`
- Relocated `web3.admin` module to `web3.geth` namespace
- Relocated `web3.miner` module to `web3.geth` namespace

Deprecated Methods

Expect the following methods to be removed in v6:

- `web3.sha3` was deprecated for `keccak()`
- `web3.soliditySha3` was deprecated for `solidityKeccak()`
- `chainId()` was deprecated for `chainId()`. Follow issue #1293 for details
- `web3.eth.getCompilers()` was deprecated and will not be replaced
- `getTransactionFromBlock()` was deprecated for `getTransactionByBlock()`

Deprecated ConciseContract and ImplicitContract

The ConciseContract and ImplicitContract have been deprecated and will be removed in v6.

ImplicitContract instances will need to use the verbose syntax. For example:

`contract.functions.<function name>.transact({})`

ConciseContract has been replaced with the ContractCaller API. Instead of using the ConciseContract factory, you can now use:

`contract.caller.<function_name>`

or the classic contract syntax:

`contract.functions.<function name>.call()`.

Some more concrete examples can be found in the ContractCaller docs

Manager Provider

In v5, only a single provider will be allowed. While allowing multiple providers is a feature we'd like to support in the future, the way that multiple providers was handled in v4 wasn't ideal. The only thing they could do was fall back. There was no mechanism for any round robin, nor was there any control around which provider was chosen. Eventually, the idea is to expand the Manager API to support injecting custom logic into the provider selection process.

For now, `manager.providers` has changed to `manager.provider`. Similarly, instances of `web3.providers` have been changed to `web3.provider`.

Testnet Changes

Web3.py will no longer automatically look up a testnet connection in IPCProvider.

2.18.4 ENS

Web3.py has stopped inferring the `.eth` TLD on domain names. If a domain name is used instead of an address, you'll need to specify the TLD. An `InvalidTLD` error will be thrown if the TLD is missing.

2.18.5 Required Infura API Key

In order to interact with Infura after March 27, 2019, you'll need to set an environment variable called `WEB3_INFURA_PROJECT_ID`. You can get a project id by visiting https://infura.io/register.

2.19 Migrating your code from v3 to v4

Web3.py follows Semantic Versioning, which means that version 4 introduced backwards-incompatible changes. If your project depends on Web3.py v3, then you'll probably need to make some changes.

Here are the most common required updates:

2.19.1 Python 2 to Python 3

Only Python 3 is supported in v4. If you are running in Python 2, it's time to upgrade. We recommend using *2to3* which can make most of your code compatible with Python 3, automatically.

The most important update, relevant to Web3.py, is the new `bytes` type. It is used regularly, throughout the library, whenever dealing with data that is not guaranteed to be text.

Many different methods in Web3.py accept text or binary data, like contract methods, transaction details, and cryptographic functions. The following example uses `sha3()`, but the same pattern applies elsewhere.

In v3 & Python 2, you might have calculated the hash of binary data this way:

```
>>> Web3.sha3('I\xe2\x99\xa5SF')
'0x50a826df121f4d076a3686d74558f40082a8e70b3469d8e9a16ceb2a79102e5e'
```

Or, you might have calculated the hash of text data this way:

```
>>> Web3.sha3(text=u'ISF')
'0x50a826df121f4d076a3686d74558f40082a8e70b3469d8e9a16ceb2a79102e5e'
```

After switching to Python 3, these would instead be executed as:

```
>>> Web3.sha3(b'I\xe2\x99\xa5SF')
HexBytes('0x50a826df121f4d076a3686d74558f40082a8e70b3469d8e9a16ceb2a79102e5e')

>>> Web3.sha3(text='ISF')
HexBytes('0x50a826df121f4d076a3686d74558f40082a8e70b3469d8e9a16ceb2a79102e5e')
```

Note that the return value is different too: you can treat `hexbytes.main.HexBytes` like any other bytes value, but the representation on the console shows you the hex encoding of those bytes, for easier visual comparison.

It takes a little getting used to, but the new py3 types are much better. We promise.

2.19.2 Filters

Filters usually don't work quite the way that people want them to.

The first step toward fixing them was to simplify them by removing the polling logic. Now, you must request an update on your filters explicitly. That means that any exceptions during the request will bubble up into your code.

In v3, those exceptions (like "filter is not found") were swallowed silently in the automated polling logic. Here was the invocation for printing out new block hashes as they appear:

```
>>> def new_block_callback(block_hash):
...     print(f"New Block: {block_hash}")
...
>>> new_block_filter = web3.eth.filter('latest')
>>> new_block_filter.watch(new_block_callback)
```

In v4, that same logic:

```
>>> new_block_filter = web3.eth.filter('latest')
>>> for block_hash in new_block_filter.get_new_entries():
...     print(f"New Block: {block_hash}")
```

The caller is responsible for polling the results from `get_new_entries()`. See *Asynchronous Filter Polling* for examples of filter-event handling with web3 v4.

2.19.3 TestRPCProvider and EthereumTesterProvider

These providers are fairly uncommon. If you don't recognize the names, you can probably skip the section.

However, if you were using web3.py for testing contracts, you might have been using TestRPCProvider or EthereumTesterProvider.

In v4 there is a new `EthereumTesterProvider`, and the old v3 implementation has been removed. Web3.py v4 uses `eth_tester.main.EthereumTester` under the hood, instead of eth-testrpc. While `eth-tester` is still in beta, many parts are already in better shape than testrpc, so we decided to replace it in v4.

If you were using TestRPC, or were explicitly importing EthereumTesterProvider, like: `from web3.providers.tester import EthereumTesterProvider`, then you will need to update.

With v4 you should import with `from web3 import EthereumTesterProvider`. As before, you'll need to install Web3.py with the `tester` extra to get these features, like:

```
$ pip install web3[tester]
```

2.19.4 Changes to base API convenience methods

Web3.toDecimal()

In v4 `Web3.toDecimal()` is renamed: `toInt()` for improved clarity. It does not return a `decimal.Decimal`, it returns an `int`.

Removed Methods

- `Web3.toUtf8` was removed for `toText()`.
- `Web3.fromUtf8` was removed for `toHex()`.
- `Web3.toAscii` was removed for `toBytes()`.
- `Web3.fromAscii` was removed for `toHex()`.
- `Web3.fromDecimal` was removed for `toHex()`.

Provider Access

In v4, `w3.currentProvider` was removed, in favor of `w3.providers`.

Disambiguating String Inputs

There are a number of places where an arbitrary string input might be either a byte-string that has been hex-encoded, or unicode characters in text. These are named `hexstr` and `text` in Web3.py. You specify which kind of `str` you have by using the appropriate keyword argument. See examples in *Encoding and Decoding Helpers*.

In v3, some methods accepted a `str` as the first positional argument. In v4, you must pass strings as one of `hexstr` or `text` keyword arguments.

Notable methods that no longer accept ambiguous strings:

- `sha3()`
- `toBytes()`

2.19.5 Contracts

- When a contract returns the ABI type `string`, Web3.py v4 now returns a `str` value by decoding the underlying bytes using UTF-8.
- When a contract returns the ABI type `bytes` (of any length), Web3.py v4 now returns a `bytes` value

2.19.6 Personal API

`w3.personal.signAndSendTransaction` is no longer available. Use `w3.personal.sendTransaction()` instead.

2.20 Web3 API

class `web3.`**`Web3`**(*provider*)

Each `Web3` instance exposes the following APIs.

2.20.1 Providers

`Web3.`**`HTTPProvider`**

Convenience API to access *`web3.providers.rpc.HTTPProvider`*

`Web3.`**`IPCProvider`**

Convenience API to access *`web3.providers.ipc.IPCProvider`*

2.20.2 Attributes

`Web3.`**`api`**

Returns the current Web3 version.

```
>>> web3.api
"4.7.0"
```

`Web3.`**`client_version`**

- Delegates to `web3_clientVersion` RPC Method

Returns the current client version.

```
>>> web3.client_version
'Geth/v1.4.11-stable-fed692f6/darwin/go1.7'
```

2.20.3 Encoding and Decoding Helpers

`Web3.to_hex(primitive=None, hexstr=None, text=None)`

Takes a variety of inputs and returns it in its hexadecimal representation. It follows the rules for converting to hex in the JSON-RPC spec

```
>>> Web3.to_hex(0)
'0x0'
>>> Web3.to_hex(1)
'0x1'
>>> Web3.to_hex(0x0)
'0x0'
>>> Web3.to_hex(0x000F)
'0xf'
>>> Web3.to_hex(b'')
'0x'
>>> Web3.to_hex(b'\x00\x0F')
'0x000f'
>>> Web3.to_hex(False)
'0x0'
>>> Web3.to_hex(True)
'0x1'
>>> Web3.to_hex(hexstr='0x000F')
'0x000f'
>>> Web3.to_hex(hexstr='000F')
'0x000f'
>>> Web3.to_hex(text='')
'0x'
>>> Web3.to_hex(text='cowmö')
'0x636f776dc3b6'
```

`Web3.to_text(primitive=None, hexstr=None, text=None)`

Takes a variety of inputs and returns its string equivalent. Text gets decoded as UTF-8.

```
>>> Web3.to_text(0x636f776dc3b6)
'cowmö'
>>> Web3.to_text(b'cowm\xc3\xb6')
'cowmö'
>>> Web3.to_text(hexstr='0x636f776dc3b6')
'cowmö'
>>> Web3.to_text(hexstr='636f776dc3b6')
'cowmö'
>>> Web3.to_text(text='cowmö')
'cowmö'
```

`Web3.to_bytes(primitive=None, hexstr=None, text=None)`

Takes a variety of inputs and returns its bytes equivalent. Text gets encoded as UTF-8.

```
>>> Web3.to_bytes(0)
b'\x00'
>>> Web3.to_bytes(0x000F)
b'\x0f'
>>> Web3.to_bytes(b'')
```

(continues on next page)

(continued from previous page)

```
b''
>>> Web3.to_bytes(b'\x00\x0F')
b'\x00\x0f'
>>> Web3.to_bytes(False)
b'\x00'
>>> Web3.to_bytes(True)
b'\x01'
>>> Web3.to_bytes(hexstr='0x000F')
b'\x00\x0f'
>>> Web3.to_bytes(hexstr='000F')
b'\x00\x0f'
>>> Web3.to_bytes(text='')
b''
>>> Web3.to_bytes(text='cowmö')
b'cowm\xc3\xb6'
```

`Web3.to_int`(*primitive=None*, *hexstr=None*, *text=None*)

Takes a variety of inputs and returns its integer equivalent.

```
>>> Web3.to_int(0)
0
>>> Web3.to_int(0x000F)
15
>>> Web3.to_int(b'\x00\x0F')
15
>>> Web3.to_int(False)
0
>>> Web3.to_int(True)
1
>>> Web3.to_int(hexstr='0x000F')
15
>>> Web3.to_int(hexstr='000F')
15
```

`Web3.to_json`(*obj*)

Takes a variety of inputs and returns its JSON equivalent.

```
>>> Web3.to_json(3)
'3'
>>> Web3.to_json({'one': 1})
'{"one": 1}'
```

2.20.4 Currency Conversions

Web3.**to_wei**(*value*, *currency*)

Returns the value in the denomination specified by the currency argument converted to wei.

```
>>> Web3.to_wei(1, 'ether')
1000000000000000000
```

Web3.**from_wei**(*value*, *currency*)

Returns the value in wei converted to the given currency. The value is returned as a Decimal to ensure precision down to the wei.

```
>>> Web3.from_wei(1000000000000000000, 'ether')
Decimal('1')
```

2.20.5 Addresses

Web3.**is_address**(*value*)

Returns True if the value is one of the recognized address formats.

- Allows for both 0x prefixed and non-prefixed values.
- If the address contains mixed upper and lower cased characters this function also checks if the address checksum is valid according to EIP55

```
>>> Web3.is_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
```

Web3.**is_checksum_address**(*value*)

Returns True if the value is a valid EIP55 checksummed address

```
>>> Web3.is_checksum_address('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
>>> Web3.is_checksum_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
False
```

Web3.**to_checksum_address**(*value*)

Returns the given address with an EIP55 checksum.

```
>>> Web3.to_checksum_address('0xd3cda913deb6f67967b99d67acdfa1712c293601')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

2.20.6 Cryptographic Hashing

classmethod Web3.**keccak**(*primitive=None*, *hexstr=None*, *text=None*)

Returns the Keccak-256 of the given value. Text is encoded to UTF-8 before computing the hash, just like Solidity. Any of the following are valid and equivalent:

```
>>> Web3.keccak(0x747874)
>>> Web3.keccak(b'\x74\x78\x74')
>>> Web3.keccak(hexstr='0x747874')
```

(continues on next page)

(continued from previous page)

```
>>> Web3.keccak(hexstr='747874')
>>> Web3.keccak(text='txt')
HexBytes('0xd7278090a36507640ea6b7a0034b69b0d240766fa3f98e3722be93c613b29d2e')
```

classmethod Web3.**solidity_keccak**(*abi_types*, *value*)

Returns the Keccak-256 as it would be computed by the solidity `keccak` function on a *packed* ABI encoding of the `value` list contents. The `abi_types` argument should be a list of solidity type strings which correspond to each of the provided values.

```
>>> Web3.solidity_keccak(['bool'], [True])
HexBytes("0x5fe7f977e71dba2ea1a68e21057beebb9be2ac30c6410aa38d4f3fbe41dcffd2")

>>> Web3.solidity_keccak(['uint8', 'uint8', 'uint8'], [97, 98, 99])
HexBytes("0x4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45")

>>> Web3.solidity_keccak(['uint8[]'], [[97, 98, 99]])
HexBytes("0x233002c671295529bcc50b76a2ef2b0de2dac2d93945fca745255de1a9e4017e")

>>> Web3.solidity_keccak(['address'], ["0x49EdDD3769c0712032808D86597B84ac5c2F5614
↪"])
HexBytes("0x2ff37b5607484cd4eecf6d13292e22bd6e5401eaffcc07e279583bc742c68882")

>>> Web3.solidity_keccak(['address'], ["ethereumfoundation.eth"])
HexBytes("0x913c99ea930c78868f1535d34cd705ab85929b2eaaf70fcd09677ecd6e5d75e9")
```

Comparable solidity usage:

```
bytes32 data1 = keccak256(abi.encodePacked(true));
assert(data1 == hex"5fe7f977e71dba2ea1a68e21057beebb9be2ac30c6410aa38d4f3fbe41dcffd2
↪");
bytes32 data2 = keccak256(abi.encodePacked(uint8(97), uint8(98), uint8(99)));
assert(data2 == hex"4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45
↪");
```

2.20.7 Check Encodability

w3.**is_encodable**(*_type*, *value*)

Returns `True` if a value can be encoded as the given type. Otherwise returns `False`.

```
>>> from web3.auto.gethdev import w3
>>> w3.is_encodable('bytes2', b'12')
True
>>> w3.is_encodable('bytes2', '0x1234')
True
>>> w3.is_encodable('bytes2', '1234')  # not 0x-prefixed, no assumptions␣
↪will be made
False
>>> w3.is_encodable('bytes2', b'1')  # does not match specified bytes size
False
>>> w3.is_encodable('bytes2', b'123')  # does not match specified bytes size
False
```

w3.**strict_bytes_type_checking**

Disable the stricter bytes type checking that is loaded by default. For more examples, see *Disabling Strict Checks for Bytes Types*

```
>>> from web3.auto.gethdev import w3

>>> w3.is_encodable('bytes2', b'12')
True

>>>  # not of exact size bytes2
>>> w3.is_encodable('bytes2', b'1')
False

>>> w3.strict_bytes_type_checking = False

>>> # zero-padded, so encoded to: b'1\x00'
>>> w3.is_encodable('bytes2', b'1')
True

>>> # re-enable it
>>> w3.strict_bytes_type_checking = True
>>> w3.is_encodable('bytes2', b'1')
False
```

2.20.8 RPC API Modules

Each `Web3` instance also exposes these namespaced API modules.

Web3.**eth**

See *web3.eth API*

Web3.**miner**

See *Miner API*

Web3.**pm**

See *Package Manager API*

Web3.**geth**

See *Geth API*

These internal modules inherit from the `web3.module.Module` class which give them some configurations internal to the web3.py library.

2.20.9 Custom Methods

You may add or overwrite methods within any module using the `attach_methods` function. To create a property instead, set `is_property` to `True`.

```
>>> w3.eth.attach_methods({
...    'example_method': Method(
...      'eth_example',
...       mungers=[...],
...       request_formatters=[...],
```

(continues on next page)

(continued from previous page)

```
...         result_formatters=[...],
...         is_property=False,
...     ),
... })
>>> w3.eth.example_method()
```

2.20.10 External Modules

External modules can be used to introduce custom or third-party APIs to your `Web3` instance. External modules are simply classes whose methods and properties can be made available within the `Web3` instance. Optionally, the external module may make use of the parent `Web3` instance by accepting it as the first argument within the `__init__` function:

```
>>> class ExampleModule:
...     def __init__(self, w3):
...         self.w3 = w3
...
...     def print_balance_of_shaq(self):
...         print(self.w3.eth.get_balance('shaq.eth'))
```

> **Warning:** Given the flexibility of external modules, use caution and only import modules from trusted third parties and open source code you've vetted!

Configuring external modules can occur either at instantiation of the `Web3` instance or by making use of the `attach_modules()` method. To instantiate the `Web3` instance with external modules use the `external_modules` keyword argument:

```
>>> from web3 import Web3, HTTPProvider
>>> from external_module_library import (
...     ModuleClass1,
...     ModuleClass2,
...     ModuleClass3,
...     ModuleClass4,
...     ModuleClass5,
... )
>>> w3 = Web3(
...     HTTPProvider(provider_uri),
...     external_modules={
...         'module1': ModuleClass1,
...         'module2': (ModuleClass2, {
...             'submodule1': ModuleClass3,
...             'submodule2': (ModuleClass4, {
...                 'submodule2a': ModuleClass5,  # submodule children may be nested␣
↪further if necessary
...             })
...         })
...     }
... )

# `return_zero`, in this case, is an example attribute of the `ModuleClass1` object
```

(continues on next page)

(continued from previous page)

```
>>> w3.module1.return_zero()
0
>>> w3.module2.submodule1.return_one()
1
>>> w3.module2.submodule2.submodule2a.return_two()
2
```

w3.**attach_modules**(*modules*)

The `attach_modules()` method can be used to attach external modules after the `Web3` instance has been instantiated.

Modules are attached via a *dict* with module names as the keys. The values can either be the module classes themselves, if there are no submodules, or two-item tuples with the module class as the 0th index and a similarly built *dict* containing the submodule information as the 1st index. This pattern may be repeated as necessary.

```
>>> from web3 import Web3, HTTPProvider
>>> from external_module_library import (
...     ModuleClass1,
...     ModuleClass2,
...     ModuleClass3,
...     ModuleClass4,
...     ModuleClass5,
... )
>>> w3 = Web3(HTTPProvider(provider_uri))

>>> w3.attach_modules({
...     'module1': ModuleClass1,  # the module class itself may be used for a␣
↪single module with no submodules
...     'module2': (ModuleClass2, {  # a tuple with module class and corresponding␣
↪submodule dict may be used for modules with submodules
...         'submodule1': ModuleClass3,
...         'submodule2': (ModuleClass4, {  # this pattern may be repeated as␣
↪necessary
...             'submodule2a': ModuleClass5,
...         })
...     })
... })
>>> w3.module1.return_zero()
0
>>> w3.module2.submodule1.return_one()
1
>>> w3.module2.submodule2.submodule2a.return_two()
2
```

2.21 web3.eth API

Warning: Whoa there, Binance Smart Chain user! web3.py is an Ethereum-specific library, which now defaults to "type 2" transactions as of the London network upgrade. BSC apparently does not support these newer transaction types.

From issues opened, it seems BSC transactions must include `gasPrice`, but not `type`, `maxFeePerGas`, or `maxPriorityFeePerGas`. If you have trouble beyond that, please find an appropriate BSC forum to raise your question.

class `web3.eth.`**`Eth`**

The `web3.eth` object exposes the following properties and methods to interact with the RPC APIs under the `eth_` namespace.

By default, when a property or method returns a mapping of keys to values, it will return an `AttributeDict` which acts like a `dict` but you can access the keys as attributes and cannot modify its fields. For example, you can find the latest block number in these two ways:

```
>>> block = web3.eth.get_block('latest')
AttributeDict({
  'hash': '0xe8ad537a261e6fff80d551d8d087ee0f2202da9b09b64d172a5f45e818eb472a',
  'number': 4022281,
  # ... etc ...
})

>>> block['number']
4022281
>>> block.number
4022281

>>> block.number = 4022282
Traceback # ... etc ...
TypeError: This data is immutable -- create a copy instead of modifying
```

This feature is available via the `attrdict_middleware` which is a default middleware.

Note: Accessing an `AttributeDict` property via attribute will break type hinting. If typing is crucial for your application, accessing via key / value, as well as removing the `attrdict_middleware` altogether, may be desired.

2.21.1 Properties

The following properties are available on the `web3.eth` namespace.

`Eth.`**`default_account`**

The ethereum address that will be used as the default `from` address for all transactions. Defaults to empty.

`Eth.`**`default_block`**

The default block number that will be used for any RPC methods that accept a block identifier. Defaults to `'latest'`.

`Eth.syncing`

- Delegates to `eth_syncing` RPC Method

Returns either `False` if the node is not syncing or a dictionary showing sync status.

```
>>> web3.eth.syncing
AttributeDict({
    'currentBlock': 2177557,
    'highestBlock': 2211611,
    'knownStates': 0,
    'pulledStates': 0,
    'startingBlock': 2177365,
})
```

`Eth.coinbase`

- Delegates to `eth_coinbase` RPC Method

Returns the current *Coinbase* address.

```
>>> web3.eth.coinbase
'0xC014BA5EC014ba5ec014Ba5EC014ba5Ec014bA5E'
```

`Eth.mining`

- Delegates to `eth_mining` RPC Method

Returns boolean as to whether the node is currently mining.

```
>>> web3.eth.mining
False
```

`Eth.hashrate`

- Delegates to `eth_hashrate` RPC Method

Returns the current number of hashes per second the node is mining with.

```
>>> web3.eth.hashrate
906
```

`Eth.max_priority_fee`

- Delegates to `eth_maxPriorityFeePerGas` RPC Method

Returns a suggestion for a max priority fee for dynamic fee transactions in Wei.

```
>>> web3.eth.max_priority_fee
2000000000
```

`Eth.gas_price`

- Delegates to `eth_gasPrice` RPC Method

Returns the current gas price in Wei.

```
>>> web3.eth.gas_price
20000000000
```

`Eth.accounts`

- Delegates to `eth_accounts` RPC Method

Returns the list of known accounts.

```
>>> web3.eth.accounts
['0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD']
```

`Eth.block_number`

- Delegates to `eth_blockNumber` RPC Method

Returns the number of the most recent block

Alias for *get_block_number()*

```
>>> web3.eth.block_number
2206939
```

`Eth.chain_id`

- Delegates to `eth_chainId` RPC Method

Returns an integer value for the currently configured "Chain Id" value introduced in EIP-155. Returns `None` if no Chain Id is available.

```
>>> web3.eth.chain_id
61
```

Note: This property gets called frequently in validation middleware, but *chain_id* is added to the `simple_cache_middleware` by default. Add the *simple_cache_middleware* to the `middleware_onion` to increase performance:

```
>>> from web3.middleware import simple_cache_middleware
>>> w3.middleware_onion.add(simple_cache_middleware)
```

2.21.2 Methods

The following methods are available on the `web3.eth` namespace.

`Eth.get_balance(account, block_identifier=eth.default_block)`

- Delegates to `eth_getBalance` RPC Method

Returns the balance of the given `account` at the block specified by `block_identifier`.

`account` may be a checksum address or an ENS name

```
>>> web3.eth.get_balance('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
77320681768999138915
```

`Eth.get_block_number()`

- Delegates to `eth_blockNumber` RPC Method

Returns the number of the most recent block.

```
>>> web3.eth.get_block_number()
2206939
```

`Eth.get_storage_at`(*account*, *position*, *block_identifier=eth.default_block*)

- Delegates to `eth_getStorageAt` RPC Method

Returns the value from a storage position for the given `account` at the block specified by `block_identifier`.

`account` may be a checksum address or an ENS name

```
>>> web3.eth.get_storage_at('0x6C8f2A135f6ed072DE4503Bd7C4999a1a17F824B', 0)
'0x00000000000000000000000000000000000000000000000000120a0b063499d4'
```

`Eth.get_proof`(*account*, *positions*, *block_identifier=eth.default_block*)

- Delegates to `eth_getProof` RPC Method

Returns the values from an array of storage positions for the given `account` at the block specified by `block_identifier`.

`account` may be a checksum address or an ENS name

```
>>> web3.eth.get_proof('0x6C8f2A135f6ed072DE4503Bd7C4999a1a17F824B', [0], 3391)
AttributeDict({
    'address': '0x4CB06C43fcdABeA22541fcF1F856A6a296448B6c',
    'accountProof': [
↪'0xf90211a03841a7ddd65c70c94b8efa79190d00f0ab134b26f18dcad508f60a7e74559d0ba0464b07429a05039e2293
↪',

↪'0xf90151a097b17a89fd2c03ee98cb6459c08f51b269da5cee46650e84470f62bf83b43efe80a03b269d284a4c3cf8f8
↪',

↪'0xf869a020d13b52a61d3c1325ce3626a51418adebd6323d4840f1bdd93906359d11c933b846f8440180a01ab7c0b0a2
↪'
                    ],
    'balance': 0,
    'codeHash': '0x551332d96d085185ab4019ad8bcf89c45321e136c261eb6271e574a2edf1461f
↪',
    'nonce': 1,
    'storageHash':
↪'0x1ab7c0b0a2a4bbb5a1495da8c142150891fc64e0c321e1feb70bd5f881951f7e',
    'storageProof': [
        AttributeDict({
            'key': '0x00',
            'value':
↪'0x48656c6c6f00000000000000000000000000000000000000000000000000000a',
            'proof': [
↪'0xf9019180a01ace80e7bed79fbadbe390876bd1a7d9770edf9462049ef8f4b555d05715d53ea049347a3c2eac6525a3
↪',

↪'0xf891808080a0c7d094301e0c54da37b696d85f72de5520b224ab2cf4f045d8db1a3374caf0488080a0fc5581783bfe
↪',

↪'0xf843a0200decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563a1a048656c6c6f000000000
↪'
```

(continues on next page)

(continued from previous page)

```
                    ]
            })
        ]
})
```

- Merkle proof verification using py-trie.

The following example verifies that the values returned in the `AttributeDict` are included in the state of given trie `root`.

```
from eth_utils import (
    keccak,
)
import rlp
from rlp.sedes import (
    Binary,
    big_endian_int,
)
from trie import (
    HexaryTrie,
)
from web3._utils.encoding import (
    pad_bytes,
)

def format_proof_nodes(proof):
    trie_proof = []
    for rlp_node in proof:
        trie_proof.append(rlp.decode(bytes(rlp_node)))
    return trie_proof

def verify_eth_get_proof(proof, root):
    trie_root = Binary.fixed_length(32, allow_empty=True)
    hash32 = Binary.fixed_length(32)

    class _Account(rlp.Serializable):
        fields = [
                    ('nonce', big_endian_int),
                    ('balance', big_endian_int),
                    ('storage', trie_root),
                    ('code_hash', hash32)
                ]
    acc = _Account(
        proof.nonce, proof.balance, proof.storageHash, proof.codeHash
    )
    rlp_account = rlp.encode(acc)
    trie_key = keccak(bytes.fromhex(proof.address[2:]))

    assert rlp_account == HexaryTrie.get_from_proof(
        root, trie_key, format_proof_nodes(proof.accountProof)
    ), f"Failed to verify account proof {proof.address}"
```

(continues on next page)

(continued from previous page)

```
    for storage_proof in proof.storageProof:
        trie_key = keccak(pad_bytes(b'\x00', 32, storage_proof.key))
        root = proof.storageHash
        if storage_proof.value == b'\x00':
            rlp_value = b''
        else:
            rlp_value = rlp.encode(storage_proof.value)

        assert rlp_value == HexaryTrie.get_from_proof(
            root, trie_key, format_proof_nodes(storage_proof.proof)
        ), f"Failed to verify storage proof {storage_proof.key}"

    return True

block = w3.eth.get_block(3391)
proof = w3.eth.get_proof('0x6C8f2A135f6ed072DE4503Bd7C4999a1a17F824B', [0, 1], 3391)
assert verify_eth_get_proof(proof, block.stateRoot)
```

`Eth.get_code`(*account*, *block_identifier=eth.default_block*)

- Delegates to `eth_getCode` RPC Method

Returns the bytecode for the given `account` at the block specified by `block_identifier`.

`account` may be a checksum address or an ENS name

```
# For a contract address.
>>> web3.eth.get_code('0x6C8f2A135f6ed072DE4503Bd7C4999a1a17F824B')
'0x606060405236156100276c5760e060020a60003504630199.....'
# For a private key address.
>>> web3.eth.get_code('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
'0x'
```

`Eth.get_block`(*block_identifier=eth.default_block*, *full_transactions=False*)

- Delegates to `eth_getBlockByNumber` or `eth_getBlockByHash` RPC Methods

Returns the block specified by `block_identifier`. Delegates to `eth_getBlockByNumber` if `block_identifier` is an integer or one of the predefined block parameters `'latest'`, `'earliest'`, `'pending'`, `'safe'`, `'finalized'` - otherwise delegates to `eth_getBlockByHash`. Throws `BlockNotFound` error if the block is not found.

If `full_transactions` is `True` then the `'transactions'` key will contain full transactions objects. Otherwise it will be an array of transaction hashes.

```
>>> web3.eth.get_block(2000000)
AttributeDict({
    'difficulty': 49824742724615,
    'extraData': '0xe4b883e5bda9e7a59ee4bb99e9b1bc',
    'gasLimit': 4712388,
    'gasUsed': 21000,
    'hash': '0xc0f4906fea23cf6f3cce98cb44e8e1449e455b28d684dfa9ff65426495584de6',
    'logsBloom':
↪'0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
```

(continues on next page)

(continued from previous page)

```
    'miner': '0x61c808d82a3ac53231750dadc13c777b59310bd9',
    'nonce': '0x3b05c6d5524209f1',
    'number': 2000000,
    'parentHash':
↪'0x57ebf07eb9ed1137d41447020a25e51d30a0c272b5896571499c82c33ecb7288',
    'receiptsRoot':
↪'0x84aea4a7aad5c5899bd5cfc7f309cc379009d30179316a2a7baa4a2ea4a438ac',
    'sha3Uncles':
↪'0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
    'size': 650,
    'stateRoot': '0x96dbad955b166f5119793815c36f11ffa909859bbfeb64b735cca37cbf10bef1
↪',
    'timestamp': 1470173578,
    'totalDifficulty': 44010101827705409388,
    'transactions': [
↪'0xc55e2b90168af6972193c1f86fa4d7d7b31a29c156665d15b9cd48618b5177ef'],
    'transactionsRoot':
↪'0xb31f174d27b99cdae8e746bd138a01ce60d8dd7b224f7c60845914def05ecc58',
    'uncles': [],
})
```

`Eth.get_block_transaction_count`(*block_identifier*)

- Delegates to `eth_getBlockTransactionCountByNumber` or `eth_getBlockTransactionCountByHash` RPC Methods

Returns the number of transactions in the block specified by `block_identifier`. Delegates to `eth_getBlockTransactionCountByNumber` if `block_identifier` is an integer or one of the predefined block parameters `'latest'`, `'earliest'`, `'pending'`, `'safe'`, `'finalized'`, otherwise delegates to `eth_getBlockTransactionCountByHash`. Throws `BlockNotFoundError` if transactions are not found.

```
>>> web3.eth.get_block_transaction_count(46147)
1
>>> web3.eth.get_block_transaction_count(
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd')  # block␣
↪46147
1
```

`Eth.get_uncle_by_block`(*block_identifier*, *uncle_index*)

- Delegates to `eth_getUncleByBlockHashAndIndex` or `eth_getUncleByBlockNumberAndIndex` RPC methods

Returns the uncle at the index specified by `uncle_index` from the block specified by `block_identifier`. Delegates to `eth_getUncleByBlockNumberAndIndex` if `block_identifier` is an integer or one of the predefined block parameters `'latest'`, `'earliest'`, `'pending'`, otherwise delegates to `eth_getUncleByBlockHashAndIndex`. Throws `BlockNotFound` if the block is not found.

```
>>> web3.eth.get_uncle_by_block(56160, 0)
AttributeDict({
  'author': '0xbe4532e1b1db5c913cf553be76180c1777055403',
  'difficulty': '0x17dd9ca0afe',
  'extraData': '0x476574682f686261722f76312e302e312f6c696e75782f676f312e342e32',
  'gasLimit': '0x2fefd8',
```

(continues on next page)

(continued from previous page)

```
  'gasUsed': '0x0',
  'hash': '0xc78c35720d930f9ef34b4e6fb9d02ffec936f9b02a8f0fa858456e4afd4d5614',
  'logsBloom':
↪'0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
  'miner': '0xbe4532e1b1db5c913cf553be76180c1777055403',
  'mixHash': '0x041e14603f35a82f6023802fec96ef760433292434a39787514f140950597e5e',
  'nonce': '0x5d2b7e3f1af09995',
  'number': '0xdb5e',
  'parentHash': '0xcc30e8a9b15c548d5bf113c834143a8f0e1909fbfea96b2a208dc154293a78cf
↪',
  'receiptsRoot':
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421',
  'sealFields': [
↪'0xa0041e14603f35a82f6023802fec96ef760433292434a39787514f140950597e5e',
↪'0x885d2b7e3f1af09995'],
  'sha3Uncles': '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
↪',
  'size': None, 'stateRoot':
↪'0x8ce2b1bf8e25a06a8ca34c647ff5fd0fa48ac725cc07f657ae1645ab8ef68c91',
  'timestamp': '0x55c6a972',
  'totalDifficulty': '0xce4c4f0a0b810b',
  'transactions': [],
  'transactionsRoot':
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421',
  'uncles': []
})

# You can also refer to the block by hash:
>>> web3.eth.get_uncle_by_block(
↪'0x685b2226cbf6e1f890211010aa192bf16f0a0cba9534264a033b023d7367b845', 0)
AttributeDict({
    ...
})
```

Eth.**get_uncle_count**(*block_identifier*)

- Delegates to `eth_getUncleCountByBlockHash` or `eth_getUncleCountByBlockNumber` RPC methods

Returns the (integer) number of uncles associated with the block specified by `block_identifier`. Delegates to `eth_getUncleCountByBlockNumber` if `block_identifier` is an integer or one of the predefined block parameters `'latest', 'earliest', 'pending'`, otherwise delegates to `eth_getUncleCountByBlockHash`. Throws `BlockNotFound` if the block is not found.

```
>>> web3.eth.get_uncle_count(56160)
1

# You can also refer to the block by hash:
>>> web3.eth.get_uncle_count(
↪'0x685b2226cbf6e1f890211010aa192bf16f0a0cba9534264a033b023d7367b845')
1
```

Eth.**get_transaction**(*transaction_hash*)

- Delegates to `eth_getTransactionByHash` RPC Method

Returns the transaction specified by `transaction_hash`. If the transaction cannot be found throws `web3.exceptions.TransactionNotFound`.

```
>>> web3.eth.get_transaction(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
AttributeDict({'blockHash': HexBytes(
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd'),
    'blockNumber': 46147,
    'from': '0xA1E4380A3B1f749673E270229993eE55F35663b4',
    'gas': 21000,
    'gasPrice': 50000000000000,
    'hash': HexBytes(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060'),
    'input': HexBytes('0x'),
    'nonce': 0,
     'r': HexBytes(
↪'0x88ff6cf0fefd94db46111149ae4bfc179e9b94721fffd821d38d16464b3f71d0'),
     's': HexBytes(
↪'0x45e0aff800961cfce805daef7016b9b675c137a6a41a548f7b60a3484c06a33a'),
     'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
     'transactionIndex': 0,
     'type': 0,
     'v': 28,
     'value': 31337
})
```

`Eth.get_raw_transaction`(*transaction_hash*)

- Delegates to `eth_getRawTransactionByHash` RPC Method

Returns the raw form of transaction specified by `transaction_hash`.

If no transaction is found, `TransactionNotFound` is raised.

```
>>> web3.eth.get_raw_transaction(
↪'0x86fbfe56cce542ff0a2a2716c31675a0c9c43701725c4a751d20ee2ddf8a733d')
HexBytes(
↪'0xf86907843b9aca0082520894dc544d1aa88ff8bbd2f2aec754b1f1e99e1812fd018086eecac466e115a0f9db4e2548
↪')
```

`Eth.get_transaction_by_block`(*block_identifier*, *transaction_index*)

- Delegates to `eth_getTransactionByBlockNumberAndIndex` or `eth_getTransactionByBlockHashAndIndex` RPC Methods

Returns the transaction at the index specified by `transaction_index` from the block specified by `block_identifier`. Delegates to `eth_getTransactionByBlockNumberAndIndex` if `block_identifier` is an integer or one of the predefined block parameters `'latest'`, `'earliest'`, `'pending'`, `'safe'`, `'finalized'`, otherwise delegates to `eth_getTransactionByBlockHashAndIndex`. If a transaction is not found at specified arguments, throws `web3.exceptions.TransactionNotFound`.

```
>>> web3.eth.get_transaction_by_block(46147, 0)
AttributeDict({
    'blockHash': '0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd
```

(continues on next page)

(continued from previous page)

```
↪',
    'blockNumber': 46147,
    'from': '0xA1E4380A3B1f749673E270229993eE55F35663b4',
    'gas': 21000,
    'gasPrice': None,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'maxFeePerGas': 2000000000,
    'maxPriorityFeePerGas': 1000000000,
    'nonce': 0,
    'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
    'transactionIndex': 0,
    'value': 31337,
})
>>> web3.eth.get_transaction_by_block(
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd', 0)
AttributeDict({
    'blockHash': '0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd
↪',
    'blockNumber': 46147,
    'from': '0xA1E4380A3B1f749673E270229993eE55F35663b4',
    'gas': 21000,
    'gasPrice': None,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'maxFeePerGas': 2000000000,
    'maxPriorityFeePerGas': 1000000000,
    'nonce': 0,
    'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
    'transactionIndex': 0,
    'value': 31337,
})
```

`Eth.get_raw_transaction_by_block(`*block_identifier*, *transaction_index*`)`

- Delegates to `eth_getRawTransactionByBlockNumberAndIndex` or `eth_getRawTransactionByBlockHashAndIndex` RPC Methods

Returns the raw transaction at the index specified by `transaction_index` from the block specified by `block_identifier`. Delegates to `eth_getRawTransactionByBlockNumberAndIndex` if `block_identifier` is an integer or one of the predefined block parameters `'latest'`, `'earliest'`, `'pending'`, `'safe'`, `'finalized'`, otherwise delegates to `eth_getRawTransactionByBlockHashAndIndex`. If a transaction is not found at specified arguments, throws `web3.exceptions.TransactionNotFound`.

```
>>> web3.eth.get_raw_transaction_by_block('latest', 0)
HexBytes(
↪'0x02f87582053901843b9aca00843b9aca008301d8a894e2dfcfa89a45abdc3de91f7a2844b276b8451d2e888ac72304
↪')
>>> web3.eth.get_raw_transaction_by_block(2, 0)
HexBytes(
↪'0x02f87582053901843b9aca00843b9aca008301d8a894e2dfcfa89a45abdc3de91f7a2844b276b8451d2e888ac72304
↪')
```

(continues on next page)

(continued from previous page)

```
>>> web3.eth.get_raw_transaction_by_block(
↪'0xca609fb606a04ce6aaec76415cd0b9d8c2bc83ad2a4d17db7fd403ee7d97bf40', 0)
HexBytes(
↪'0x02f87582053901843b9aca00843b9aca008301d8a894e2dfcfa89a45abdc3de91f7a2844b276b8451d2e888ac7230
↪')
```

Eth.**wait_for_transaction_receipt**(*transaction_hash*, *timeout=120*, *poll_latency=0.1*)

Waits for the transaction specified by `transaction_hash` to be included in a block, then returns its transaction receipt.

Optionally, specify a `timeout` in seconds. If timeout elapses before the transaction is added to a block, then *wait_for_transaction_receipt()* raises a `web3.exceptions.TimeExhausted` exception.

```
>>> web3.eth.wait_for_transaction_receipt(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
# If transaction is not yet in a block, time passes, while the thread sleeps...
# ...
# Then when the transaction is added to a block, its receipt is returned:
AttributeDict({
    'blockHash': '0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd
↪',
    'blockNumber': 46147,
    'contractAddress': None,
    'cumulativeGasUsed': 21000,
    'from': '0xA1E4380A3B1f749673E270229993eE55F35663b4',
    'gasUsed': 21000,
    'logs': [],
    'logsBloom': '0x0000000000000000000000000000000000000000000000000...0000',
    'status': 1,
    'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
    'transactionHash':
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'transactionIndex': 0,
})
```

Eth.**get_transaction_receipt**(*transaction_hash*)

- Delegates to `eth_getTransactionReceipt` RPC Method

Returns the transaction receipt specified by `transaction_hash`. If the transaction cannot be found throws `web3.exceptions.TransactionNotFound`.

If `status` in response equals 1 the transaction was successful. If it is equals 0 the transaction was reverted by EVM.

```
>>> web3.eth.get_transaction_receipt(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')  # not yet␣
↪mined
Traceback # ... etc ...
TransactionNotFound: Transaction with hash:␣
↪0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060 not found.

# wait for it to be mined....
>>> web3.eth.get_transaction_receipt(
```

(continues on next page)

(continued from previous page)

```
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
AttributeDict({
    'blockHash': '0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd
↪',
    'blockNumber': 46147,
    'contractAddress': None,
    'cumulativeGasUsed': 21000,
    'from': '0xA1E4380A3B1f749673E270229993eE55F35663b4',
    'gasUsed': 21000,
    'logs': [],
    'logsBloom': '0x000000000000000000000000000000000000000000000000...0000',
    'status': 1, # 0 or 1
    'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
    'transactionHash':
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'transactionIndex': 0,
})
```

`Eth.`**`get_transaction_count`**(*account*, *block_identifier=web3.eth.default_block*)

- Delegates to `eth_getTransactionCount` RPC Method

Returns the number of transactions that have been sent from `account` as of the block specified by `block_identifier`.

`account` may be a checksum address or an ENS name

```
>>> web3.eth.get_transaction_count('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
340
```

`Eth.`**`send_transaction`**(*transaction*)

- Delegates to `eth_sendTransaction` RPC Method

Signs and sends the given `transaction`

The `transaction` parameter should be a dictionary with the following fields.

- `from`: `bytes or text`, checksum address or ENS name - (optional, default: `web3.eth.defaultAccount`) The address the transaction is sent from.
- `to`: `bytes or text`, checksum address or ENS name - (optional when creating new contract) The address the transaction is directed to.
- `gas`: `integer` - (optional) Integer of the gas provided for the transaction execution. It will return unused gas.
- `maxFeePerGas`: `integer or hex` - (optional) maximum amount you're willing to pay, inclusive of `baseFeePerGas` and `maxPriorityFeePerGas`. The difference between `maxFeePerGas` and `baseFeePerGas + maxPriorityFeePerGas` is refunded to the user.
- `maxPriorityFeePerGas`: `integer or hex` - (optional) the part of the fee that goes to the miner
- `gasPrice`: `integer` - Integer of the gasPrice used for each paid gas **LEGACY** - unless you have a good reason to use `gasPrice`, use `maxFeePerGas` and `maxPriorityFeePerGas` instead.
- `value`: `integer` - (optional) Integer of the value send with this transaction
- `data`: `bytes or text` - The compiled code of a contract OR the hash of the invoked method signature and encoded parameters. For details see Ethereum Contract ABI.

- nonce: integer - (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.

If the transaction specifies a data value but does not specify gas then the gas value will be populated using the estimate_gas() function with an additional buffer of 100000 gas up to the gasLimit of the latest block. In the event that the value returned by estimate_gas() method is greater than the gasLimit a ValueError will be raised.

```
# simple example (web3.py and / or client determines gas and fees, typically␣
↪defaults to a dynamic fee transaction post London fork)
>>> web3.eth.send_transaction({
  'to': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
  'from': web3.eth.coinbase,
  'value': 12345
})

# Dynamic fee transaction, introduced by EIP-1559:
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
>>> web3.eth.send_transaction({
  'to': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
  'from': web3.eth.coinbase,
  'value': 12345,
  'gas': 21000,
  'maxFeePerGas': web3.to_wei(250, 'gwei'),
  'maxPriorityFeePerGas': web3.to_wei(2, 'gwei'),
})
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')

# Legacy transaction (less efficient)
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
>>> web3.eth.send_transaction({
  'to': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
  'from': web3.eth.coinbase,
  'value': 12345,
  'gas': 21000,
  'gasPrice': web3.to_wei(50, 'gwei'),
})
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
```

Eth.**sign_transaction**(*transaction*)

- Delegates to eth_signTransaction RPC Method.

Returns a transaction that's been signed by the node's private key, but not yet submitted. The signed tx can be submitted with Eth.send_raw_transaction

```
>>> signed_txn = w3.eth.sign_transaction(dict(
    nonce=w3.eth.get_transaction_count(w3.eth.coinbase),
    maxFeePerGas=2000000000,
    maxPriorityFeePerGas=1000000000,
    gas=100000,
    to='0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
    value=1,
    data=b'',
    )
```

(continues on next page)

(continued from previous page)

```
)
b"\xf8d\x80\x85\x040\xe24\x00\x82R\x08\x94\xdcTM\x1a\xa8\x8f\xf8\xbb\xd2\xf2\xae\
↪xc7T\xb1\xf1\xe9\x9e\x18\x12\xfd\x01\x80\x1b\xa0\x11\r\x8f\xee\x1d\xe5=\xf0\x87\
↪x0en\xb5\x99\xed;\xf6\x8f\xb3\xf1\xe6,\x82\xdf\xe5\x97lF|\x97%;\x15\xa04P\xb7=*\
↪xef \t\xf0&\xbc\xbf\tz%z\xe7\xa3~\xb5\xd3\xb7=\xc0v\n\xef\xad+\x98\xe3'"  # noqa:␣
↪E501
```

Eth.**send_raw_transaction**(*raw_transaction*)

- Delegates to `eth_sendRawTransaction` RPC Method

Sends a signed and serialized transaction. Returns the transaction hash as a HexBytes object.

```
>>> signed_txn = w3.eth.account.sign_transaction(dict(
    nonce=w3.eth.get_transaction_count(public_address_of_senders_account),
    maxFeePerGas=3000000000,
    maxPriorityFeePerGas=2000000000,
    gas=100000,
    to='0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
    value=12345,
    data=b'',
    type=2,  # (optional) the type is now implicitly set based on appropriate␣
↪transaction params
    chainId=1,
  ),
  private_key_for_senders_account,
)
>>> w3.eth.send_raw_transaction(signed_txn.rawTransaction)
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
```

Eth.**replace_transaction**(*transaction_hash*, *new_transaction*)

- Delegates to `eth_sendTransaction` RPC Method

Sends a transaction that replaces the transaction with `transaction_hash`.

The `transaction_hash` must be the hash of a pending transaction.

The `new_transaction` parameter should be a dictionary with transaction fields as required by *send_transaction()*. It will be used to entirely replace the transaction of `transaction_hash` without using any of the pending transaction's values.

If the `new_transaction` specifies a `nonce` value, it must match the pending transaction's nonce.

If the `new_transaction` specifies `maxFeePerGas` and `maxPriorityFeePerGas` values, they must be greater than the pending transaction's values for each field, respectively.

- Legacy Transaction Support (Less Efficient - Not Recommended)

If the pending transaction specified a `gasPrice` value (legacy transaction), the `gasPrice` value for the `new_transaction` must be greater than the pending transaction's `gasPrice`.

If the `new_transaction` does not specify any of `gasPrice`, `maxFeePerGas`, or `maxPriorityFeePerGas` values, one of the following will happen:

- If the pending transaction has a `gasPrice` value, this value will be used with a multiplier of 1.125 - This is typically the minimum `gasPrice` increase a node requires before it accepts a replacement transaction.

- If a gas price strategy is set, the `gasPrice` value from the gas price strategy(See *Gas Price API*) will be used.
- If none of the above, the client will ultimately decide appropriate values for `maxFeePerGas` and `maxPriorityFeePerGas`. These will likely be default values and may result in an unsuccessful replacement of the pending transaction.

This method returns the transaction hash of the replacement transaction as a HexBytes object.

```
>>> tx = web3.eth.send_transaction({
        'to': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
        'from': web3.eth.coinbase,
        'value': 1000
    })
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
>>> web3.eth.replace_transaction(
↪'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331', {
        'to': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
        'from': web3.eth.coinbase,
        'value': 2000
    })
HexBytes('0x4177e670ec6431606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1528989')
```

`Eth.`**`modify_transaction`**(*transaction_hash*, ***transaction_params*)

- Delegates to `eth_sendTransaction` RPC Method

Sends a transaction that modifies the transaction with `transaction_hash`.

`transaction_params` are keyword arguments that correspond to valid transaction parameters as required by *`send_transaction()`*. The parameter values will override the pending transaction's values to create the replacement transaction to send.

The same validation and defaulting rules of *`replace_transaction()`* apply.

This method returns the transaction hash of the newly modified transaction as a HexBytes object.

```
>>> tx = web3.eth.send_transaction({
        'to': '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
        'from': web3.eth.coinbase,
        'value': 1000
    })
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
>>> web3.eth.modify_transaction(
↪'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331', value=2000)
HexBytes('0xec6434e6701771606e55d6b4ca35a1a6b75ee3d73315145a921026d15299d05')
```

`Eth.`**`sign`**(*account*, *data=None*, *hexstr=None*, *text=None*)

- Delegates to `eth_sign` RPC Method

Caller must specify exactly one of: `data`, `hexstr`, or `text`.

Signs the given data with the private key of the given `account`. The account must be unlocked.

`account` may be a checksum address or an ENS name

```
>>> web3.eth.sign(
      '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
```

(continues on next page)

(continued from previous page)

```
    text='some-text-tö-sign')

↪'0x1a8bbe6eab8c72a219385681efefe565afd3accee35f516f8edf5ae82208fbd45a58f9f9116d8d88ba40fcd29076d6
↪'

>>> web3.eth.sign(
    '0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
    data=b'some-text-t\xc3\xb6-sign')

↪'0x1a8bbe6eab8c72a219385681efefe565afd3accee35f516f8edf5ae82208fbd45a58f9f9116d8d88ba40fcd29076d6
↪'

>>> web3.eth.sign(
    '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
    hexstr='0x736f6d652d746578742d74c3b62d7369676e')

↪'0x1a8bbe6eab8c72a219385681efefe565afd3accee35f516f8edf5ae82208fbd45a58f9f9116d8d88ba40fcd29076d6
↪'
```

`Eth.`**`sign_typed_data`**(*account*, *jsonMessage*)

- Delegates to `eth_signTypedData` RPC Method

Note: `eth_signTypedData` is not currently supported by any major client (Besu, Erigon, Geth, or Nethermind)

Please note that the `jsonMessage` argument is the loaded JSON Object and **NOT** the JSON String itself.

Signs the `Structured Data` (or `Typed Data`) with the private key of the given `account`. The account must be unlocked.

`account` may be a checksum address or an ENS name

`Eth.`**`call`**(*transaction*, *block_identifier=web3.eth.default_block*, *state_override=None*, *ccip_read_enabled=True*)

- Delegates to `eth_call` RPC Method

Executes the given transaction locally without creating a new transaction on the blockchain. Returns the return value of the executed contract.

The `transaction` parameter is handled in the same manner as the *`send_transaction()`* method.

```
>>> myContract.functions.setVar(1).transact()
HexBytes('0x79af0c7688afba7588c32a61565fd488c422da7b5773f95b242ea66d3d20afda')
>>> myContract.functions.getVar().call()
1
# The above call equivalent to the raw call:
>>> web3.eth.call({'value': 0, 'gas': 21736, 'maxFeePerGas': 2000000000,
↪'maxPriorityFeePerGas': 1000000000, 'to':
↪'0xc305c901078781C232A2a521C2aF7980f8385ee9', 'data': '0x477a5c98'})
HexBytes('0x0000000000000000000000000000000000000000000000000000000000000001')
```

In most cases it is better to make contract function call through the *`web3.contract.Contract`* interface.

Overriding state is a debugging feature available in Geth clients. View their usage documentation for a list of possible parameters.

EIP-3668 introduced support for the `OffchainLookup` revert / CCIP Read support. In order to properly handle a call to a contract function that reverts with an `OffchainLookup` error for offchain data retrieval, the `ccip_read_enabled` flag has been added to the `eth_call` method. `ccip_read_enabled` is optional, yielding the default value for CCIP Read on calls to a global `global_ccip_read_enabled` flag on the provider which is set to `True` by default. This means CCIP Read is enabled by default for calls, as is recommended in EIP-3668. Therefore, calls to contract functions that revert with an `OffchainLookup` will be handled appropriately by default.

The `ccip_read_enabled` flag on the call will always override the value of the global flag on the provider for explicit control over specific calls. If the flag on the call is set to `False`, the call will raise the `OffchainLookup` instead of properly handling the exception according to EIP-3668. This may be useful for "preflighting" a transaction with a call (see *CCIP Read support for offchain lookup* within the examples section).

If the function called results in a `revert` error, a `ContractLogicError` will be raised. If there is an error message with the error, web3.py attempts to parse the message that comes back and return it to the user as the error string. As of v6.3.0, the raw data is also returned and can be accessed via the `data` attribute on `ContractLogicError`.

`Eth.create_access_list(`*transaction*, *block_identifier=web3.eth.default_block*`)`

- Delegates to `eth_createAccessList` RPC Method

This method creates an EIP-2930 type `accessList` based on a given `transaction`. The `accessList` contains all storage slots and addresses read and written by the transaction, except for the sender account and the precompiles. This method uses the same `transaction` call object and `block_identifier` object as `call()`. An `accessList` can be used to access contracts that became inaccessible due to gas cost increases.

The `transaction` parameter is handled in the same manner as the `send_transaction()` method. The optional `block_identifier` parameter is a block_number or `latest` or `pending`. Default is `latest`.

```
>>> w3.eth.create_access_list(
...     {
...         "to": to_checksum_address("0xF0109fC8DF283027b6285cc889F5aA624EaC1F55"),
...         "gasPrice": 10**11,
...         "value": 0,
...         "data": "0x608060806080608155",
...     },
...     "pending",
... )
AttributeDict({
    'accessList': [
        AttributeDict({
            'address': '0xde0B295669a9FD93d5F28D9Ec85E40f4cb697BAe',
            'storageKeys': [
                HexBytes(
↪'0x0000000000000000000000000000000000000000000000000000000000000003'),
                HexBytes(
↪'0x0000000000000000000000000000000000000000000000000000000000000007'),
            ]
        }),
        AttributeDict({
            'address': '0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413',
            'storageKeys': []
        }),
    ],
    "gasUsed": 21000
```

(continues on next page)

(continued from previous page)

```
})
```

The method `eth_createAccessList` returns a list of addresses and storage keys used by the transaction, plus the gas consumed when the `accessList` is included. Like `eth_estimateGas`, this is an estimation; the list could change when the transaction is actually finalized. Adding an `accessList` to your transaction does not necessarily result in lower gas usage compared to a transaction without an `accessList`.

`Eth.fee_history`(*block_count*, *newest_block*, *reward_percentiles=None*)

- Delegates to `eth_feeHistory` RPC Method

Returns transaction fee data for up to 1,024 blocks.

Parameters

- **block_count** (`int or hexstring`) – The number of blocks in the requested range. Depending on the client, this value should be either a `int` between 1 and 1024 or a hexstring. Less than requested may be returned if not all blocks are available.
- **newest_block** (`int or BlockParams`) – The newest, highest-numbered, block in the requested range. This value may be an `int` or one of the predefined block parameters `'latest'`, `'earliest'`, or `'pending'`.
- **reward_percentiles** (`List[float] or None`) – *(optional)* A monotonically increasing list of percentile `float` values to sample from each block's effective priority fees per gas in ascending order, weighted by gas used.

Returns

An `AttributeDict` containing the following keys:

- **oldestBlock** *(int)* – The oldest, lowest-numbered, block in the range requested as a `BlockNumber` type with `int` value.
- **baseFeePerGas** *(List[Wei])* – An array of block base fees per gas. This includes the next block after the newest of the returned range, because this value can be derived from the newest block. Zeroes are returned for pre-EIP-1559 blocks.
- **gasUsedRatio** *(List[float])* – An array of `gasUsed/gasLimit` float values for the requested blocks.
- **reward** *(List[List[Wei]])* – *(optional)* A two-dimensional array of effective priority fees per gas at the requested block percentiles.

```
>>> w3.eth.fee_history(4, 'latest', [10, 90])
AttributeDict({
    'oldestBlock': 3,
    'reward': [[220, 7145389], [1000000, 6000213], [550, 550], [125, 12345678]],
    'baseFeePerGas': [202583058, 177634473, 155594425, 136217133, 119442408],
    'gasUsedRatio': [0.007390479689642084, 0.0036988514889990873, 0.
↪0018512333048507866, 0.00741217041320997]
})
```

`Eth.estimate_gas`(*transaction*, *block_identifier=None*, *state_override=None*)

- Delegates to `eth_estimateGas` RPC Method

Executes the given transaction locally without creating a new transaction on the blockchain. Returns amount of gas consumed by execution which can be used as a gas estimate.

The `transaction` and `block_identifier` parameters are handled in the same manner as the *send_transaction()* method.

The `state_override` is useful when there is a chain of transaction calls. It overrides state so that the gas estimate of a transaction is accurate in cases where prior calls produce side effects.

```
>>> web3.eth.estimate_gas({'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601', 'from
↪':web3.eth.coinbase, 'value': 12345})
21000
```

`Eth.generate_gas_price(transaction_params=None)`

Uses the selected gas price strategy to calculate a gas price. This method returns the gas price denominated in wei.

The `transaction_params` argument is optional however some gas price strategies may require it to be able to produce a gas price.

```
>>> web3.eth.generate_gas_price()
20000000000
```

Note: For information about how gas price can be customized in web3 see *Gas Price API*.

`Eth.set_gas_price_strategy(gas_price_strategy)`

Set the selected gas price strategy. It must be a method of the signature `(web3, transaction_params)` and return a gas price denominated in wei.

2.21.3 Filters

The following methods are available on the `web3.eth` object for interacting with the filtering API.

`Eth.filter(filter_params)`

- Delegates to `eth_newFilter`, `eth_newBlockFilter`, and `eth_newPendingTransactionFilter` RPC Methods.

This method delegates to one of three RPC methods depending on the value of `filter_params`.

- If `filter_params` is the string `'pending'` then a new filter is registered using the `eth_newPendingTransactionFilter` RPC method. This will create a new filter that will be called for each new unmined transaction that the node receives.
- If `filter_params` is the string `'latest'` then a new filter is registered using the `eth_newBlockFilter` RPC method. This will create a new filter that will be called each time the node receives a new block.
- If `filter_params` is a dictionary then a new filter is registered using the `eth_newFilter` RPC method. This will create a new filter that will be called for all log entries that match the provided `filter_params`.

This method returns a `web3.utils.filters.Filter` object which can then be used to either directly fetch the results of the filter or to register callbacks which will be called with each result of the filter.

When creating a new log filter, the `filter_params` should be a dictionary with the following keys.

- `fromBlock`: `integer/tag` - (optional, default: "latest") Integer block number, or one of predefined block identifiers "latest", "pending", "earliest", "safe", or "finalized".
- `toBlock`: `integer/tag` - (optional, default: "latest") Integer block number, or one of predefined block identifiers "latest", "pending", "earliest", "safe", or "finalized".

- address: string or list of strings, each 20 Bytes - (optional) Contract address or a list of addresses from which logs should originate.
- topics: list of 32 byte strings or null - (optional) Array of topics that should be used for filtering. Topics are order-dependent. This parameter can also be a list of topic lists in which case filtering will match any of the provided topic arrays.

Note: Though "latest" and "safe" block identifiers are not yet part of the specifications for eth_newFilter, they are supported by web3.py and may or may not yield expected results depending on the node being accessed.

See *Monitoring Events* for more information about filtering.

```
>>> web3.eth.filter('latest')
<BlockFilter at 0x10b72dc28>
>>> web3.eth.filter('pending')
<TransactionFilter at 0x10b780340>
>>> web3.eth.filter({'fromBlock': 1000000, 'toBlock': 1000100, 'address':
↪'0x6C8f2A135f6ed072DE4503Bd7C4999a1a17F824B'})
<LogFilter at 0x10b7803d8>
```

Eth.**get_filter_changes**(*self*, *filter_id*)

- Delegates to eth_getFilterChanges RPC Method.

Returns all new entries which occurred since the last call to this method for the given filter_id

```
>>> filter = web3.eth.filter()
>>> web3.eth.get_filter_changes(filter.filter_id)
[
    {
        'address': '0xDc3A9Db694BCdd55EBaE4A89B22aC6D12b3F0c24',
        'blockHash':
↪'0xb72256286ca528e09022ffd408856a73ef90e7216ac560187c6e43b4c4efd2f0',
        'blockNumber': 2217196,
        'data': '0x0000000000000000000000000000000000000000000000000000000000000001
↪',
        'logIndex': 0,
        'topics': [
↪'0xe65b00b698ba37c614af350761c735c5f4a82b4ab365a1f1022d49d9dfc8e930',
        '0x000000000000000000000000754c50465885f1ed1fa1a55b95ee8ecf3f1f4324',
        '0x296c7fb6ccafa3e689950b947c2895b07357c95b066d5cdccd58c301f41359a3'],
        'transactionHash':
↪'0xfe1289fd3915794b99702202f65eea2e424b2f083a12749d29b4dd51f6dce40d',
        'transactionIndex': 1,
    },
    ...
]
```

Eth.**get_filter_logs**(*self*, *filter_id*)

- Delegates to eth_getFilterLogs RPC Method.

Returns all entries for the given filter_id

```
>>> filter = web3.eth.filter()
>>> web3.eth.get_filter_logs(filter.filter_id)
[
    {
        'address': '0xDc3A9Db694BCdd55EBaE4A89B22aC6D12b3F0c24',
        'blockHash':
↪'0xb72256286ca528e09022ffd408856a73ef90e7216ac560187c6e43b4c4efd2f0',
        'blockNumber': 2217196,
        'data': '0x0000000000000000000000000000000000000000000000000000000000000001
↪',
        'logIndex': 0,
        'topics': [
↪'0xe65b00b698ba37c614af350761c735c5f4a82b4ab365a1f1022d49d9dfc8e930',
        '0x000000000000000000000000754c50465885f1ed1fa1a55b95ee8ecf3f1f4324',
        '0x296c7fb6ccafa3e689950b947c2895b07357c95b066d5cdccd58c301f41359a3'],
        'transactionHash':
↪'0xfe1289fd3915794b99702202f65eea2e424b2f083a12749d29b4dd51f6dce40d',
        'transactionIndex': 1,
    },
    ...
]
```

Eth.**uninstall_filter**(*self*, *filter_id*)

- Delegates to `eth_uninstallFilter` RPC Method.

Uninstalls the filter specified by the given `filter_id`. Returns boolean as to whether the filter was successfully uninstalled.

```
>>> filter = web3.eth.filter()
>>> web3.eth.uninstall_filter(filter.filter_id)
True
>>> web3.eth.uninstall_filter(filter.filter_id)
False  # already uninstalled.
```

Eth.**get_logs**(*filter_params*)

This is the equivalent of: creating a new filter, running *get_filter_logs()*, and then uninstalling the filter. See *filter()* for details on allowed filter parameters.

Eth.**submit_hashrate**(*hashrate*, *nodeid*)

- Delegates to `eth_submitHashrate` RPC Method

```
>>> node_id = '59daa26581d0acd1fce254fb7e85952f4c09d0915afd33d3886cd914bc7d283c'
>>> web3.eth.submit_hashrate(5000, node_id)
True
```

Eth.**submit_work**(*nonce*, *pow_hash*, *mix_digest*)

- Delegates to `eth_submitWork` RPC Method.

```
>>> web3.eth.submit_work(
        1,
        '0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        '0xD1FE5700000000000000000000000000D1FE5700000000000000000000000000',
```

(continues on next page)

(continued from previous page)

```
    )
True
```

2.21.4 Contracts

`Eth.contract`(*address=None*, *contract_name=None*, *ContractFactoryClass=Contract*, ***contract_factory_kwargs*)

If `address` is provided, then this method will return an instance of the contract defined by `abi`. The address may be a checksum string, or an ENS name like `'mycontract.eth'`.

```
from web3 import Web3

w3 = Web3(...)

contract = w3.eth.contract(address='0x000000000000000000000000000000000000dEaD',␣
↪abi=...)

# alternatively:
contract = w3.eth.contract(address='mycontract.eth', abi=...)
```

Note: If you use an ENS name to initialize a contract, the contract will be looked up by name on each use. If the name could ever change maliciously, first *Get the Address for an ENS Name*, and then create the contract with the checksum address.

If `address` is *not* provided, the newly created contract class will be returned. That class will then be initialized by supplying the address.

```
from web3 import Web3

w3 = Web3(...)

Contract = w3.eth.contract(abi=...)

# later, initialize contracts with the same metadata at different addresses:
contract1 = Contract(address='0x000000000000000000000000000000000000dEaD')
contract2 = Contract(address='mycontract.eth')
```

`contract_name` will be used as the name of the contract class. If it is `None` then the name of the `ContractFactoryClass` will be used.

`ContractFactoryClass` will be used as the base Contract class.

The following arguments are accepted for contract class creation.

Parameters

- **abi** (*ABI*) – Application Binary Interface. Usually provided since an `abi` is required to interact with any contract.
- **asm** – Asssembly code generated by the compiler
- **ast** – Abstract Syntax Tree of the contract generated by the compiler

- **bytecode** – Bytecode of the contract generated by the compiler
- **bytecode_runtime** – Bytecode stored at the contract address, excludes the constructor and initialization code
- **clone_bin** –
- **dev_doc** –
- **decode_tuples** – Optionally convert tuples/structs to named tuples
- **interface** –
- **metadata** – Contract Metadata generated by the compiler
- **opcodes** – Opcodes for the contract generated by the compiler
- **src_map** –
- **src_map_runtime** –
- **user_doc** –

Returns
Instance of the contract

Return type
Contract

Raises

- `TypeError` – If the address is not provided
- `AttributeError` – If the contract class is not initialized

See the *Contracts* documentation for more information about Contracts.

`Eth.`**`set_contract_factory`**(*contractFactoryClass*)

Modify the default contract factory from `Contract` to `contractFactoryClass`. Future calls to `Eth.contract()` will then default to `contractFactoryClass`.

2.22 Beacon API

Warning: This API Is experimental. Client support is incomplete and the API itself is still evolving.

To use this API, you'll need a beacon node running locally or remotely. To set that up, refer to the documentation of your specific client.

Once you have a running beacon node, import and configure your beacon instance:

```
>>> from web3.beacon import Beacon
>>> beacon = Beacon("http://localhost:5051")
```

2.22.1 Methods

Beacon.**get_genesis**()

```
>>> beacon.get_genesis()
{
  'data': {
    'genesis_time': '1605700807',
    'genesis_validators_root':
↪'0x9436e8a630e3162b7ed4f449b12b8a5a368a4b95bc46b941ae65c11613bfa4c1',
    'genesis_fork_version': '0x00002009'
  }
}
```

Beacon.**get_hash_root**(*state_id='head'*)

```
>>> beacon.get_hash_root()
{
  "data": {
    "root":"0xbb399fda70617a6f198b3d9f1c1cdbd70077677231b84f34e58568c9dc903558"
  }
}
```

Beacon.**get_fork_data**(*state_id='head'*)

```
>>> beacon.get_fork_data()
{
  'data': {
    'previous_version': '0x00002009',
    'current_version': '0x00002009',
    'epoch': '0'
  }
}
```

Beacon.**get_finality_checkpoint**(*state_id='head'*)

```
>>> beacon.get_finality_checkpoint()
{
  'data': {
    'previous_justified': {
      'epoch': '5024',
      'root': '0x499ba555e8e8be639dd84be1be6d54409738facefc662f37d97065aa91a1a8d4'
    },
    'current_justified': {
      'epoch': '5025',
      'root': '0x34e8a230f11536ab2ec56a0956e1f3b3fd703861f96d4695877eaa48fbacc241'
    },
    'finalized': {
      'epoch': '5024',
      'root': '0x499ba555e8e8be639dd84be1be6d54409738facefc662f37d97065aa91a1a8d4'
    }
  }
}
```

Beacon.**get_validators**(*state_id='head'*)

```
>>> beacon.get_validators()
 {
   'data': [
     {
       'index': '110280',
       'balance': '32000000000',
       'status': 'pending_queued',
       'validator': {
         'pubkey':
↪'0x99d37d1f7dd15859995330f75c158346f86d298e2ffeedfbf1b38dcf3df89a7dbd1b34815f3bcd1b2a5588592a35b7
↪',
         'withdrawal_credentials':
↪'0x00f338cfdb0c22bb85beed9042bd19fff58ad6421c8a833f8bc902b7cca06f5f',
         'effective_balance': '32000000000',
         'slashed': False,
         'activation_eligibility_epoch': '5029',
         'activation_epoch': '18446744073709551615',
         'exit_epoch': '18446744073709551615',
         'withdrawable_epoch': '18446744073709551615'
       }
     },
     ...
   ]
 }
```

Beacon.**get_validator**(*validator_id*, *state_id='head'*)

```
>>> beacon.get_validator(110280)
{
  'data': {
    'index': '110280',
    'balance': '32000000000',
    'status': 'pending_queued',
    'validator': {
      'pubkey':
↪'0x99d37d1f7dd15859995330f75c158346f86d298e2ffeedfbf1b38dcf3df89a7dbd1b34815f3bcd1b2a5588592a35b7
↪',
      'withdrawal_credentials':
↪'0x00f338cfdb0c22bb85beed9042bd19fff58ad6421c8a833f8bc902b7cca06f5f',
      'effective_balance': '32000000000',
      'slashed': False,
      'activation_eligibility_epoch': '5029',
      'activation_epoch': '18446744073709551615',
      'exit_epoch': '18446744073709551615',
      'withdrawable_epoch': '18446744073709551615'
    }
  }
}
```

Beacon.**get_validator_balances**(*state_id='head'*)

```
>>> beacon.get_validator_balances()
{
  'data': [
    {
      'index': '110278',
      'balance': '32000000000'
    },
    ...
  ]
}
```

Beacon.**get_epoch_committees**(*state_id='head'*)

```
>>> beacon.get_epoch_committees()
{
  'data': [
    {
      'slot': '162367',
      'index': '25',
      'validators': ['50233', '36829', '84635', ...],
    },
    ...
  ]
}
```

Beacon.**get_block_headers**()

```
>>> beacon.get_block_headers()
{
  'data': [
    {
      'root': '0xa3873e7b1e0bcc7c59013340cfea59dff16e42e79825e7b8ab6c243dbafd4fe0',
      'canonical': True,
      'header': {
        'message': {
          'slot': '163587',
          'proposer_index': '69198',
          'parent_root':
↪'0xc32558881dbb791ef045c48e3709a0978dc445abee4ae34d30df600eb5fbbb3d',
          'state_root':
↪'0x4dc0a72959803a84ee0231160b05dda76a91b8f8b77220b4cfc7db160840b8a8',
          'body_root':
↪'0xa3873e7b1e0bcc7c59013340cfea59dff16e42e79825e7b8ab6c243dbafd4fe0'
        },
        'signature':
↪'0x87b549448d36e5e8b1783944b5511a05f34bb78ad3fcbf71a1adb346eed363d46e50d51ac53cd23bd03d0107d064e0
↪'
      }
    }
  ]
}
```

Beacon.**get_block_header**(*block_id*)

```
>>> beacon.get_block_header(1)
{
  'data': {
    root': '0x30c04689dd4f6cd4d56eb78f72727d2d16d8b6346724e4a88f546875f11b750d',
    'canonical': True,
    'header': {
      'message': {
        'slot': '1',
        'proposer_index': '61090',
        'parent_root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87',
        'state_root':
↪'0x7773ed5a7e944c6238cd0a5c32170663ef2be9efc594fb43ad0f07ecf4c09d2b',
        'body_root':
↪'0x30c04689dd4f6cd4d56eb78f72727d2d16d8b6346724e4a88f546875f11b750d'
      },
      'signature':
↪'0xa30d70b3e62ff776fe97f7f8b3472194af66849238a958880510e698ec3b8a470916680b1a82f9d4753c023153fbe
↪'
    }
  }
}
```

Beacon.**get_block**(*block_id*)

```
>>> beacon.get_block(1)
{
  'data': {
    'message': {
      'slot': '1',
      'proposer_index': '61090',
      'parent_root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87',
      'state_root':
↪'0x7773ed5a7e944c6238cd0a5c32170663ef2be9efc594fb43ad0f07ecf4c09d2b',
      'body': {
        'randao_reveal':
↪'0x8e245a52a0a680fcfe789013e123880c321f237de10cad108dc55dd47290d7cfe50cdaa003c6f783405efdac48cef4
↪',
        'eth1_data': {
          'deposit_root':
↪'0x4e910ac762815c13e316e72506141f5b6b441d58af8e0a049cd3341c25728752',
          'deposit_count': '100596',
          'block_hash':
↪'0x89cb78044843805fb4dab8abd743fc96c2b8e955c58f9b7224d468d85ef57130'
        },
        'graffiti':
↪'0x74656b752f76302e31322e31342b34342d673863656562663600000000000000',
        'proposer_slashings': [],
        'attester_slashings': [],
        'attestations': [
          {
            'aggregation_bits': '0x0080020004000000008208000102000905',
```

(continues on next page)

(continued from previous page)

```
            'data': {
              'slot': '0',
              'index': '7',
              'beacon_block_root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87',
              'source': {
                'epoch': '0',
                'root':
↪'0x0000000000000000000000000000000000000000000000000000000000000000'
              },
              'target': {
                'epoch': '0',
                'root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87'
              }
            },
            'signature':
↪'0x967dd2946358db7e426ed19d4576bc75123520ef6a489ca50002222070ee4611f9cef394e5e3071236a93b825f18a4
↪'
          }
        ],
        'deposits': [],
        'voluntary_exits': []
      }
    },
    'signature':
↪'0xa30d70b3e62ff776fe97f7f8b3472194af66849238a958880510e698ec3b8a470916680b1a82f9d4753c023153fbe6
↪'
  }
}
```

Beacon.**get_block_root**(*block_id*)

```
>>> beacon.get_block_root(1)
{
  'data': {
    'root': '0x30c04689dd4f6cd4d56eb78f72727d2d16d8b6346724e4a88f546875f11b750d'
  }
}
```

Beacon.**get_block_attestations**(*block_id*)

```
>>> beacon.get_block_attestations(1)
{
  'data': [
    {
      'aggregation_bits': '0x00800200040000000082080001020009 05',
      'data': {
        'slot': '0',
        'index': '7',
        'beacon_block_root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87',
```

(continues on next page)

(continued from previous page)

```
        'source': {
          'epoch': '0',
          'root':
↪'0x0000000000000000000000000000000000000000000000000000000000000000'
        },
        'target': {
          'epoch': '0',
          'root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87'
        }
      },
      'signature':
↪'0x967dd2946358db7e426ed19d4576bc75123520ef6a489ca50002222070ee4611f9cef394e5e3071236a93b825f18a4
↪'
    },
    ...
  ]
}
```

Beacon.**get_attestations**()

```
>>> beacon.get_attestations()
{'data': []}
```

Beacon.**get_attester_slashings**()

```
>>> beacon.get_attester_slashings()
{'data': []}
```

Beacon.**get_proposer_slashings**()

```
>>> beacon.get_proposer_slashings()
{'data': []}
```

Beacon.**get_voluntary_exits**()

```
>>> beacon.get_voluntary_exits()
{'data': []}
```

Beacon.**get_fork_schedule**()

```
>>> beacon.get_fork_schedule()
{
  'data': [
    {
      'previous_version': '0x00002009',
      'current_version': '0x00002009',
      'epoch': '0'
    }
  ]
}
```

Beacon.**get_spec**()

```
>>> beacon.get_spec()
{
  'data': {
    'DEPOSIT_CONTRACT_ADDRESS': '0x8c5fecdC472E27Bc447696F431E425D02dd46a8c',
    'MIN_ATTESTATION_INCLUSION_DELAY': '1',
    'SLOTS_PER_EPOCH': '32',
    'SHUFFLE_ROUND_COUNT': '90',
    'MAX_EFFECTIVE_BALANCE': '32000000000',
    'DOMAIN_BEACON_PROPOSER': '0x00000000',
    'MAX_ATTESTER_SLASHINGS': '2',
    'DOMAIN_SELECTION_PROOF': '0x05000000',
    ...
  }
}
```

Beacon.**get_deposit_contract**()

```
>>> beacon.get_deposit_contract()
{
  'data': {
    'chain_id': '5',
    'address': '0x8c5fecdC472E27Bc447696F431E425D02dd46a8c'
  }
}
```

Beacon.**get_beacon_state**(*state_id='head'*)

```
>>> beacon.get_beacon_state()
{
  'data': {
    'genesis_time': '1',
    'genesis_validators_root':
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2',
    'slot': '1',
    'fork': {
      'previous_version': '0x00000000',
      'current_version': '0x00000000',
      'epoch': '1'
    },
    'latest_block_header': {
      'slot': '1',
      'proposer_index': '1',
      'parent_root':
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2',
      'state_root':
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2',
      'body_root':
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2'
    },
    'block_roots': [
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2'],
```

(continues on next page)

(continued from previous page)

```
    'state_roots': [
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2'],
    'historical_roots': [
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2'],
    'eth1_data': {
      'deposit_root':
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2',
      'deposit_count': '1',
      'block_hash':
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2'
    },
    'eth1_data_votes': [...],
    'eth1_deposit_index': '1',
    'validators': [...],
    'balances': [...],
    'randao_mixes': [...],
    'slashings': [...],
    'previous_epoch_attestations': [...],
    'current_epoch_attestations': [...],
    'justification_bits': '0x0f',
    'previous_justified_checkpoint': {
      'epoch': '5736',
      'root': '0xec7ef54f1fd81bada8170dd0cb6be8216f8ee2f445e6936f95f5c6894a4a3b38'
    },
    'current_justified_checkpoint': {
      'epoch': '5737',
      'root': '0x781f0166e34c361ce2c88070c1389145abba2836edcb446338a2ca2b0054826e'
    },
    'finalized_checkpoint': {
      'epoch': '5736',
      'root': '0xec7ef54f1fd81bada8170dd0cb6be8216f8ee2f445e6936f95f5c6894a4a3b38'
    }
  }
}
```

Beacon.**get_beacon_heads**()

```
>>> beacon.get_beacon_heads()
{
  'data': [
    {
      'slot': '221600',
      'root': '0x9987754077fe6100a60c75d81a51b1ef457d019404d1546a66f4f5d6c23fae45'
    }
  ]
}
```

Beacon.**get_node_identity**()

```
>>> beacon.get_node_identity()
{
  'data': {
```

(continues on next page)

(continued from previous page)

```
    'peer_id': '16Uiu2HAmLZ1CYVFKpa3wwn4cnknZqosum8HX3GHDhUpEULQc9ixE',
    'enr': 'enr:-KG4QCIp6eCZ6hG_
↪fd93qsw12qmbfsl2rUTfQvwVP4FOTlWeNXYo0Gg9y3WVYIdF6FQC6R0E8CbK0Ywq_
↪6TKMx1BpGlAhGV0aDKQOwiHlQAAIAn__________4JpZIJ2NIJpcIR_
↪AAABiXNlY3AyNTZrMaEDdVT4g1gw86BfbrtLCq2fRBlG0AnMxsXtAQgA327S5FeDdGNwgiMog3VkcIIjKA
↪',
    'p2p_addresses': ['/ip4/127.0.0.1/tcp/9000/p2p/
↪16Uiu2HAmLZ1CYVFKpa3wwn4cnknZqosum8HX3GHDhUpEULQc9ixE'],
    'discovery_addresses': ['/ip4/127.0.0.1/udp/9000/p2p/
↪16Uiu2HAmLZ1CYVFKpa3wwn4cnknZqosum8HX3GHDhUpEULQc9ixE'],
    'metadata': {'seq_number': '0', 'attnets': '0x0000000000000000'}
  }
}
```

Beacon.**get_peers**()

```
>>> beacon.get_peers()
{
  'data': [
    {
      'peer_id': '16Uiu2HAkw1yVqF3RtMCBHMbkLZbNhfGcTUdD6Uo4X3wfzPhGVnqv',
      'address': '/ip4/3.127.23.51/tcp/9000',
      'state': 'connected',
      'direction': 'outbound'
    },
    {
      'peer_id': '16Uiu2HAmEJHiCzgS8GwiEYLyM3d148mzvZ9iZzsz8yqayWVPANMG',
      'address': '/ip4/3.88.7.240/tcp/9000',
      'state': 'connected',
      'direction': 'outbound'
    }
  ]
}
```

Beacon.**get_peer**(*peer_id*)

```
>>> beacon.get_peer('16Uiu2HAkw1yVqF3RtMCBHMbkLZbNhfGcTUdD6Uo4X3wfzPhGVnqv')
{
  'data': {
    'peer_id': '16Uiu2HAkw1yVqF3RtMCBHMbkLZbNhfGcTUdD6Uo4X3wfzPhGVnqv',
    'address': '/ip4/3.127.23.51/tcp/9000',
    'state': 'connected',
    'direction': 'outbound'
  }
}
```

Beacon.**get_health**()

```
>>> beacon.get_health()
200
```

Beacon.**get_version**()

```
>>> beacon.get_version()
{
  'data': {
    'version': 'teku/v20.12.0+9-g9392008/osx-x86_64/adoptopenjdk-java-15'
  }
}
```

Beacon.**get_syncing**()

```
>>> beacon.get_syncing()
{
  'data': {
    'head_slot': '222270',
    'sync_distance': '190861'
  }
}
```

2.23 Package Manager API

The `web3.pm` object exposes methods to interact with Packages as defined by ERC 1123.

- To learn more about the EthPM spec, visit the spec or the documentation.

Warning: The `web3.pm` API is still under development and likely to change quickly.

Now is a great time to get familiar with the API, and test out writing code that uses some of the great upcoming features.

By default, access to this module has been turned off in the stable version of web3.py:

```
>>> from web3 import Web3, IPCProvider
>>> w3 = Web3(IPCProvider(...))
>>> w3.pm
...
AttributeError: The Package Management feature is disabled by default ...
```

In order to access these features, you can turn it on with…

```
>>> w3.enable_unstable_package_management_api()
>>> w3.pm
<web3.pm.PM at 0x....>
```

2.23.1 Methods

The following methods are available on the `web3.pm` namespace.

class `web3.pm.PM`(*w3: AsyncWeb3* | Web3)

The PM module will work with any subclass of `ERC1319Registry`, tailored to a particular implementation of ERC1319, set as its `registry` attribute.

get_package_from_manifest(*manifest: Manifest*) → *Package*

Returns a Package # noqa: E501 instance built with the given manifest.

- **Parameters:**
  - `manifest`: A dict representing a valid manifest

get_package_from_uri(*manifest_uri: URI*) → *Package*

Returns a Package # noqa: E501 instance built with the Manifest stored at the URI. If you want to use a specific IPFS backend, set `ETHPM_IPFS_BACKEND_CLASS` to your desired backend. Defaults to Infura IPFS backend.

- **Parameters:**
  - `uri`: Must be a valid content-addressed URI

get_local_package(*package_name: str, ethpm_dir: Path* | *None = None*) → *Package*

Returns a Package # noqa: E501 instance built with the Manifest found at the package name in your local ethpm_dir.

- **Parameters:**
  - `package_name`: Must be the name of a package installed locally.
  - `ethpm_dir`: Path pointing to the target ethpm directory (optional).

set_registry(*address: Address* | *ChecksumAddress* | ENS) → None

Sets the current registry used in `web3.pm` functions that read/write to an on-chain registry. This method accepts checksummed/canonical addresses or ENS names. Addresses must point to an on-chain instance of an ERC1319 registry implementation.

To use an ENS domain as the address, make sure a valid ENS instance set as `web3.ens`.

- **Parameters:**
  - `address`: Address of on-chain Registry.

deploy_and_set_registry() → ChecksumAddress

Returns the address of a freshly deployed instance of *SimpleRegistry* and sets the newly deployed registry as the active registry on `web3.pm.registry`.

To tie your registry to an ENS name, use web3's ENS module, ie.

```
w3.ens.setup_address(ens_name, w3.pm.registry.address)
```

release_package(*package_name: str, version: str, manifest_uri: URI*) → bytes

Returns the release id generated by releasing a package on the current registry. Requires `web3.PM` to have a registry set. Requires `web3.eth.default_account` to be the registry owner.

- **Parameters:**
  - **`package_name`: Must be a valid package name, matching the**
    given manifest.

  - `version`: Must be a valid package version, matching the given manifest.
  - **`manifest_uri`: Must be a valid content-addressed URI. Currently,** only IPFS and Github content-addressed URIs are supported.

get_all_package_names() → Iterable[str]

Returns a tuple containing all the package names available on the current registry.

get_package_count() → int

Returns the number of packages available on the current registry.

get_release_count(*package_name: str*) → int

Returns the number of releases of the given package name available on the current registry.

get_release_id(*package_name: str*, *version: str*) → bytes

Returns the 32 byte identifier of a release for the given package name and version, if they are available on the current registry.

get_all_package_releases(*package_name: str*) → Iterable[Tuple[str, str]]

Returns a tuple of release data (version, manifest_ur) for every release of the given package name available on the current registry.

get_release_id_data(*release_id: bytes*) → ReleaseData

Returns `(package_name, version, manifest_uri)` associated with the given release id, *if* it is available on the current registry.

- **Parameters:**
  - `release_id`: 32 byte release identifier

get_release_data(*package_name: str*, *version: str*) → ReleaseData

Returns `(package_name, version, manifest_uri)` associated with the given package name and version, *if* they are published to the currently set registry.

- **Parameters:**
  - `name`: Must be a valid package name.
  - `version`: Must be a valid package version.

get_package(*package_name: str*, *version: str*) → *Package*

Returns a `Package` instance, generated by the `manifest_uri` associated with the given package name and version, if they are published to the currently set registry.

- **Parameters:**
  - `name`: Must be a valid package name.
  - `version`: Must be a valid package version.

class web3.pm.**ERC1319Registry**(*address: Address*, *w3: Web3*)

The ERC1319Registry class is a base class for all registry implementations to inherit from. It defines the methods specified in ERC 1319. All of these methods are prefixed with an underscore, since they are not intended to be accessed directly, but rather through the methods on `web3.pm`. They are unlikely to change, but must be implemented in a *ERC1319Registry* subclass in order to be compatible with the *PM* module. Any custom methods (eg. not defined in ERC1319) in a subclass should *not* be prefixed with an underscore.

All of these methods must be implemented in any subclass in order to work with *web3.pm.PM*. Any implementation specific logic should be handled in a subclass.

abstract `__init__`(*address: Address*, *w3: Web3*) → None

Initializes the class with the on-chain address of the registry, and a web3 instance connected to the chain where the registry can be found.

Must set the following properties...

- `self.registry`: A *web3.contract* instance of the target registry.
- `self.address`: The address of the target registry.
- **`self.w3`: The *web3* instance connected to the chain where the**
  registry can be found.

abstract `_release`(*package_name: str*, *version: str*, *manifest_uri: str*) → bytes

Returns the releaseId created by successfully adding a release to the registry.

- **Parameters:**
  - `package_name`: Valid package name according the spec.
  - **`version`: Version identifier string, can conform to**
    any versioning scheme.
  - **`manifest_uri`: URI location of a manifest which details the**
    release contents

abstract `_get_package_name`(*package_id: bytes*) → str

Returns the package name associated with the given package id, if the package id exists on the connected registry.

- **Parameters:**
  - `package_id`: 32 byte package identifier.

abstract `_get_all_package_ids`() → Iterable[bytes]

Returns a tuple containing all of the package ids found on the connected registry.

abstract `_get_release_id`(*package_name: str*, *version: str*) → bytes

Returns the 32 bytes release id associated with the given package name and version, if the release exists on the connected registry.

- **Parameters:**
  - `package_name`: Valid package name according the spec.
  - **`version`: Version identifier string, can conform to**
    any versioning scheme.

abstract `_get_all_release_ids`(*package_name: str*) → Iterable[bytes]

Returns a tuple containing all of the release ids belonging to the given package name, if the package has releases on the connected registry.

- **Parameters:**
  - `package_name`: Valid package name according the spec.

abstract `_get_release_data`(*release_id: bytes*) → ReleaseData

Returns a tuple containing (package_name, version, manifest_uri) for the given release id, if the release exists on the connected registry.

- **Parameters:**
  - `release_id`: 32 byte release identifier.

abstract _generate_release_id(*package_name: str*, *version: str*) → bytes

Returns the 32 byte release identifier that *would* be associated with the given package name and version according to the registry's hashing mechanism. The release *does not* have to exist on the connected registry.

- **Parameters:**
  - `package_name`: Valid package name according the spec.
  - **`version`: Version identifier string, can conform to** any versioning scheme.

abstract _num_package_ids() → int

Returns the number of packages that exist on the connected registry.

abstract _num_release_ids(*package_name: str*) → int

Returns the number of releases found on the connected registry, that belong to the given package name.

- **Parameters:**
  - `package_name`: Valid package name according the spec.

2.23.2 Creating your own Registry class

If you want to implement your own registry and use it with `web3.pm`, you must create a subclass that inherits from `ERC1319Registry`, and implements all the ERC 1319 standard methods prefixed with an underscore in `ERC1319Registry`. Then, you have to manually set it as the `registry` attribute on `web3.pm`.

```
custom_registry = CustomRegistryClass(address, w3)
w3.pm.registry = custom_registry
```

One reason a user might want to create their own Registry class is if they build a custom Package Registry smart contract that has features beyond those specified in ERC 1319. For example, the ability to delete a release or some micropayment feature. Rather than accessing those functions directly on the contract instance, they can create a custom `ERC1319Registry` subclass to easily call both the standard & custom methods.

2.24 Net API

The `web3.net` object exposes methods to interact with the RPC APIs under the `net_` namespace.

2.24.1 Properties

The following properties are available on the `web3.net` namespace.

`web3.net.`**`listening`**`()`

..py:property::

- Delegates to `net_listening` RPC method

Returns true if client is actively listening for network connections.

```
>>> web3.net.listening
True
```

`web3.net.`**`peer_count`**`()`

..py:property::

- Delegates to `net_peerCount` RPC method

Returns number of peers currently connected to the client.

```
>>> web3.net.peer_count
1
```

`web3.net.version()`

..py:property::

- Delegates to `net_version` RPC Method

Returns the current network id.

```
>>> web3.net.version
'8996'
```

2.25 Miner API

The `web3.geth.miner` object exposes methods to interact with the RPC APIs under the `miner_` namespace that are supported by the Geth client.

> **Warning:** Deprecated: The Geth client deprecated the miner namespace because mining was switched off during the proof-of-stake transition.

2.25.1 Methods

The following methods are available on the `web3.geth.miner` namespace.

`GethMiner.make_dag(number)`

- Delegates to `miner_makeDag` RPC Method

Generate the DAG for the given block number.

```
>>> web3.geth.miner.make_dag(10000)
```

`GethMiner.set_extra(extra)`

- Delegates to `miner_setExtra` RPC Method

Set the 32 byte value `extra` as the extra data that will be included when this node mines a block.

```
>>> web3.geth.miner.set_extra('abcdefghijklmnopqrstuvwxyzABCDEF')
```

`GethMiner.set_gas_price(gas_price)`

- Delegates to `miner_setGasPrice` RPC Method

Sets the minimum accepted gas price that this node will accept when mining transactions. Any transactions with a gas price below this value will be ignored.

```
>>> web3.geth.miner.set_gas_price(19999999999)
```

GethMiner.**start**(*num_threads*)

- Delegates to `miner_start` RPC Method

Start the CPU mining process using the given number of threads.

```
>>> web3.geth.miner.start(2)
```

GethMiner.**stop**()

- Delegates to `miner_stop` RPC Method

Stop the CPU mining operation

```
>>> web3.geth.miner.stop()
```

GethMiner.**start_auto_dag**()

- Delegates to `miner_startAutoDag` RPC Method

Enable automatic DAG generation.

```
>>> web3.geth.miner.start_auto_dag()
```

GethMiner.**stop_auto_dag**()

- Delegates to `miner_stopAutoDag` RPC Method

Disable automatic DAG generation.

```
>>> web3.geth.miner.stop_auto_dag()
```

2.26 Geth API

The `web3.geth` object exposes modules that enable you to interact with the JSON-RPC endpoints supported by Geth that are not defined in the standard set of Ethereum JSONRPC endpoints according to EIP 1474.

2.26.1 GethAdmin API

The following methods are available on the `web3.geth.admin` namespace.

The `web3.geth.admin` object exposes methods to interact with the RPC APIs under the `admin_` namespace that are supported by the Geth client.

web3.geth.admin.**datadir**()

- Delegates to `admin_datadir` RPC Method

Returns the system path of the node's data directory.

```
>>> web3.geth.admin.datadir()
'/Users/snakecharmers/Library/Ethereum'
```

web3.geth.admin.**node_info**()

- Delegates to `admin_nodeInfo` RPC Method

Returns information about the currently running node.

```
>>> web3.geth.admin.node_info()
{
    'enode': 'enode://
↪e54eebad24dce1f6d246bea455ffa756d97801582420b9ed681a2ea84bf376d0bd87ae8dd6dc06cdb862a2ca89ecabe1b
↪',
    'id':
↪'e54eebad24dce1f6d246bea455ffa756d97801582420b9ed681a2ea84bf376d0bd87ae8dd6dc06cdb862a2ca89ecabe1
↪',
    'ip': '::',
    'listenAddr': '[::]:30303',
    'name': 'Geth/v1.4.11-stable-fed692f6/darwin/go1.7',
    'ports': {'discovery': 30303, 'listener': 30303},
    'protocols': {
        'eth': {
            'difficulty': 57631175724744612603,
            'genesis':
↪'0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3',
            'head':
↪'0xaaef6b9dd0d34088915f4c62b6c166379da2ad250a88f76955508f7cc81fb796',
            'network': 1,
        },
    },
}
```

web3.geth.admin.**peers**()

- Delegates to `admin_peers` RPC Method

Returns the current peers the node is connected to.

```
>>> web3.geth.admin.peers()
[
    {
        'caps': ['eth/63'],
        'id':
↪'146e8e3e2460f1e18939a5da37c4a79f149c8b9837240d49c7d94c122f30064e07e4a42ae2c2992d0f8e7e6f68a30e7e
↪',
        'name': 'Geth/v1.4.10-stable/windows/go1.6.2',
        'network': {
            'localAddress': '10.0.3.115:64478',
            'remoteAddress': '72.208.167.127:30303',
        },
        'protocols': {
            'eth': {
                'difficulty': 17179869184,
                'head':
↪'0xd4e56740f876aef8c010b86a40d5f56745a118d0906a34e69aec8c0db1cb8fa3',
                'version': 63,
            },
```

(continues on next page)

(continued from previous page)

```
        }
    },
    {
        'caps': ['eth/62', 'eth/63'],
        'id':
↪'76cb6cd3354be081923a90dfd4cda40aa78b307cc3cf4d5733dc32cc171d00f7c08356e9eb2ea47eab5aad7a15a3419b
↪',
        'name': 'Geth/v1.4.10-stable-5f55d95a/linux/go1.5.1',
        'network': {
            'localAddress': '10.0.3.115:64784',
            'remoteAddress': '60.205.92.119:30303',
        },
        'protocols': {
            'eth': {
                'difficulty': 57631175724744612603,
                'head':
↪'0xaaef6b9dd0d34088915f4c62b6c166379da2ad250a88f76955508f7cc81fb796',
                'version': 63,
            },
        },
    },
    ...
]
```

web3.geth.admin.**add_peer**(*node_url*)

- Delegates to `admin_addPeer` RPC Method

Requests adding a new remote node to the list of tracked static nodes.

```
>>> web3.geth.admin.add_peer('enode://
↪e54eebad24dce1f6d246bea455ffa756d97801582420b9ed681a2ea84bf376d0bd87ae8dd6dc06cdb862a2ca89ecabe1b
↪71.255.237:30303')
True
```

web3.geth.admin.**start_http**(*host='localhost'*, *port=8545*, *cors=''*, *apis='eth,net,web3'*)

- Delegates to `admin_startHTTP` RPC Method

Starts the HTTP based JSON RPC API webserver on the specified `host` and `port`, with the `rpccorsdomain` set to the provided `cors` value and with the APIs specified by `apis` enabled. Returns boolean as to whether the server was successfully started.

```
>>> web3.geth.admin.start_http()
True
```

web3.geth.admin.**start_ws**(*host='localhost'*, *port=8546*, *cors=''*, *apis='eth,net,web3'*)

- Delegates to `admin_startWS` RPC Method

Starts the Websocket based JSON RPC API webserver on the specified `host` and `port`, with the `rpccorsdomain` set to the provided `cors` value and with the APIs specified by `apis` enabled. Returns boolean as to whether the server was successfully started.

```
>>> web3.geth.admin.start_ws()
True
```

web3.geth.admin.**stop_http**()

- Delegates to `admin_stopHTTP` RPC Method

Stops the HTTP based JSON RPC server.

```
>>> web3.geth.admin.stop_http()
True
```

web3.geth.admin.**stop_ws**()

- Delegates to `admin_stopWS` RPC Method

Stops the Websocket based JSON RPC server.

```
>>> web3.geth.admin.stop_ws()
True
```

2.26.2 GethPersonal API

The following methods are available on the `web3.geth.personal` namespace.

web3.geth.personal.**ec_recover**(*message*, *signature*)

- Delegates to `personal_ecRecover` RPC Method

Returns the address associated with a signature created with `personal.sign`.

```
>>> web3.geth.personal.sign('snakesnax', '0x9ad3c920dce5cea9a31d69467bb8d7c954e5acff
↪', '')

↪'0x8eb502165dec388af1c45c4bc835fd1852eaf358316ae5d248a40af8cd8dd7dc6373a6e606d8b411f788718b8b09a
↪'
>>> web3.geth.personal.ec_recover('snakesnax',
↪'0x8eb502165dec388af1c45c4bc835fd1852eaf358316ae5d248a40af8cd8dd7dc6373a6e606d8b411f788718b8b09a
↪')
'0x9ad3c920dce5cea9a31d69467bb8d7c954e5acff'
```

web3.geth.personal.**import_raw_key**(*private_key*, *passphrase*)

- Delegates to `personal_importRawKey` RPC Method

Adds the given `private_key` to the node's keychain, encrypted with the given `passphrase`. Returns the address of the imported account.

```
>>> web3.geth.personal.import_raw_key(some_private_key, 'the-passphrase')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

web3.geth.personal.**list_accounts**()

- Delegates to `personal_listAccounts` RPC Method

Returns the list of known accounts.

```
>>> web3.geth.personal.list_accounts()
['0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD']
```

`web3.geth.personal.`**`list_wallets`**`()`

- Delegates to `personal_listWallets` RPC Method

Returns the list of wallets managed by Geth.

```
>>> web3.geth.personal.list_wallets()
[{
    accounts: [{
        address: "0x44f705f3c31017856777f2931c2f09f240dd800b",
        url: "keystore:///path/to/keystore/UTC--2020-03-30T23-24-43.133883000Z--
↪44f705f3c31017856777f2931c2f09f240dd800b"
    }],
    status: "Unlocked",
    url: "keystore:///path/to/keystore/UTC--2020-03-30T23-24-43.133883000Z--
↪44f705f3c31017856777f2931c2f09f240dd800b"
}]
```

`web3.geth.personal.`**`lock_account`**`(`*account*`)`

- Delegates to `personal_lockAccount` RPC Method

Locks the given `account`.

```
>>> web3.geth.personal.lock_account('0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD')
True
```

`web3.geth.personal.`**`new_account`**`(`*passphrase*`)`

- Delegates to `personal_newAccount` RPC Method

Generates a new account in the node's keychain encrypted with the given `passphrase`. Returns the address of the created account.

```
>>> web3.geth.personal.new_account('the-passphrase')
'0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD'
```

`web3.geth.personal.`**`send_transaction`**`(`*transaction*, *passphrase*`)`

- Delegates to `personal_sendTransaction` RPC Method

Sends the transaction.

`web3.geth.personal.`**`sign`**`(`*message*, *account*, *passphrase*`)`

- Delegates to `personal_sign` RPC Method

Generates an Ethereum-specific signature for `keccak256("\x19Ethereum Signed Message:\n" + len(message) + message))`

```
>>> web3.geth.personal.sign('snakesnax', '0x9ad3c920dce5cea9a31d69467bb8d7c954e5acff
↪', '')

↪'0x8eb502165dec388af1c45c4bc835fd1852eaf358316ae5d248a40af8cd8dd7dc6373a6e606d8b411f788718b8b09a6
↪'
```

`web3.geth.personal.unlock_account(account, passphrase, duration=None)`

- Delegates to `personal_unlockAccount` RPC Method

Unlocks the given `account` for `duration` seconds. If `duration` is `None`, then the account will remain unlocked for the current default duration set by Geth. If `duration` is set to `0`, the account will remain unlocked indefinitely. Returns a `boolean` signifying whether the account was unlocked successfully.

```
>>> web3.geth.personal.unlock_account('0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
↪'wrong-passphrase')
False
>>> web3.geth.personal.unlock_account('0x582AC4D8929f58c217d4a52aDD361AE470a8a4cD',
↪'the-passphrase')
True
```

2.26.3 GethTxPool API

The `web3.geth.txpool` object exposes methods to interact with the RPC APIs under the `txpool_` namespace. These methods are only exposed under the `geth` namespace since they are not standard.

The following methods are available on the `web3.geth.txpool` namespace.

`TxPool.inspect()`

- Delegates to `txpool_inspect` RPC Method

Returns a textual summary of all transactions currently pending for inclusion in the next block(s) as well as ones that are scheduled for future execution.

```
>>> web3.geth.txpool.inspect()
{
    'pending': {
        '0x26588a9301b0428d95e6Fc3A5024fcE8BEc12D51': {
          31813: ["0x3375Ee30428b2A71c428afa5E89e427905F95F7e: 0 wei + 500000 ×␣
↪20000000000 gas"]
        },
        '0x2a65Aca4D5fC5B5C859090a6c34d164135398226': {
          563662: ["0x958c1Fa64B34db746925c6F8a3Dd81128e40355E: 1051546810000000000␣
↪wei + 90000 × 20000000000 gas"],
          563663: ["0x77517B1491a0299A44d668473411676f94e97E34: 1051190740000000000␣
↪wei + 90000 × 20000000000 gas"],
          563664: ["0x3E2A7Fe169c8F8eee251BB00d9fb6d304cE07d3A: 1050828950000000000␣
↪wei + 90000 × 20000000000 gas"],
          563665: ["0xAF6c4695da477F8C663eA2D8B768Ad82Cb6A8522: 1050544770000000000␣
↪wei + 90000 × 20000000000 gas"],
          563666: ["0x139B148094C50F4d20b01cAf21B85eDb711574dB: 1048598530000000000␣
↪wei + 90000 × 20000000000 gas"],
          563667: ["0x48B3Bd66770b0D1EeceFCe090daFeE36257538aE: 1048367260000000000␣
↪wei + 90000 × 20000000000 gas"],
          563668: ["0x468569500925D53e06Dd0993014aD166fD7Dd381: 1048126690000000000␣
↪wei + 90000 × 20000000000 gas"],
          563669: ["0x3DcB4C90477a4b8Ff7190b79b524773CbE3bE661: 1047965690000000000␣
↪wei + 90000 × 20000000000 gas"],
          563670: ["0x6DfeF5BC94b031407FFe71ae8076CA0FbF190963: 1047859050000000000␣
↪wei + 90000 × 20000000000 gas"]
```

(continues on next page)

(continued from previous page)

```
        },
        '0x9174E688d7dE157C5C0583Df424EAAB2676aC162': {
          3: ["0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413: 30000000000000000000 wei␣
↪+ 85000 × 21000000000 gas"]
        },
        '0xb18F9d01323e150096650ab989CfecD39D757Aec': {
          777: ["0xcD79c72690750F079ae6AB6ccd7e7aEDC03c7720: 0 wei + 1000000 ×␣
↪20000000000 gas"]
        },
        '0xB2916C870Cf66967B6510B76c07E9d13a5D23514': {
          2: ["0x576f25199D60982A8f31A8DfF4da8aCB982e6ABa: 26000000000000000000 wei␣
↪+ 90000 × 20000000000 gas"]
        },
        '0xBc0CA4f217E052753614d6B019948824d0d8688B': {
          0: ["0x2910543Af39abA0Cd09dBb2D50200b3E800A63D2: 10000000000000000000 wei␣
↪+ 50000 × 1171602790622 gas"]
        },
        '0xea674fdde714fd979de3edf0f56aa9716b898ec8': {
          70148: ["0xe39c55ead9f997f7fa20ebe40fb4649943d7db66: 1000767667434026200␣
↪wei + 90000 × 20000000000 gas"]
        }
      },
      'queued': {
        '0x0F6000De1578619320aBA5e392706b131FB1dE6f': {
          6: ["0x8383534d0bcd0186d326C993031311c0Ac0D9B2d: 9000000000000000000 wei␣
↪+ 21000 × 20000000000 gas"]
        },
        '0x5b30608c678e1ac464A8994C3B33E5CdF3497112': {
          6: ["0x9773547e27f8303C87089dc42D9288aa2B9d8F06: 50000000000000000000 wei␣
↪+ 90000 × 50000000000 gas"]
        },
        '0x976A3Fc5d6f7d259EBfb4cc2Ae75115475E9867C': {
          3: ["0x346FB27dE7E7370008f5da379f74dd49F5f2F80F: 140000000000000000 wei +␣
↪90000 × 20000000000 gas"]
        },
        '0x9B11bF0459b0c4b2f87f8CEBca4cfc26f294B63A': {
          2: ["0x24a461f25eE6a318BDef7F33De634A67bb67Ac9D: 17000000000000000000 wei␣
↪+ 90000 × 50000000000 gas"],
          6: ["0x6368f3f8c2B42435D6C136757382E4A59436a681: 17990000000000000000 wei␣
↪+ 90000 × 20000000000 gas", "0x8db7b4e0ecb095fbd01dffa62010801296a9ac78:␣
↪16998950000000000000 wei + 90000 × 20000000000 gas"],
          7: ["0x6368f3f8c2B42435D6C136757382E4A59436a681: 17900000000000000000 wei␣
↪+ 90000 × 20000000000 gas"]
        }
      }
}
```

`TxPool.status()`

- Delegates to `txpool_status` RPC Method

Returns a textual summary of all transactions currently pending for inclusion in the next block(s) as well as ones that are scheduled for future execution.

```
{
    pending: 10,
    queued: 7,
}
```

TxPool.**content**()

- Delegates to `txpool_content` RPC Method

Returns the exact details of all transactions that are pending or queued.

```
>>> web3.geth.txpool.content()
{
  'pending': {
    '0x0216D5032f356960Cd3749C31Ab34eEFF21B3395': {
      806: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x0216D5032f356960Cd3749C31Ab34eEFF21B3395",
        'gas': "0x5208",
        'gasPrice': None,
        'hash': "0xaf953a2d01f55cfe080c0c94150a60105e8ac3d51153058a1f03dd239dd08586
↪",
        'input': "0x",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
        'nonce': "0x326",
        'to': "0x7f69a91A3CF4bE60020fB58B893b7cbb65376db8",
        'transactionIndex': None,
        'value': "0x19a99f0cf456000"
      }]
    },
    '0x24d407e5A0B506E1Cb2fae163100B5DE01F5193C': {
      34: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x24d407e5A0B506E1Cb2fae163100B5DE01F5193C",
        'gas': "0x44c72",
        'gasPrice': None,
        'hash': "0xb5b8b853af32226755a65ba0602f7ed0e8be2211516153b75e9ed640a7d359fe
↪",
        'input':
↪"0xb61d27f6000000000000000000000000024d407e5a0b506e1cb2fae163100b5de01f5193c0000000000000000000000000
↪",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
        'nonce': "0x22",
        'to': "0x7320785200f74861B69C49e4ab32399a71b34f1a",
        'transactionIndex': None,
        'value': "0x0"
      }]
    }
```

(continues on next page)

(continued from previous page)

```
  },
  'queued': {
    '0x976A3Fc5d6f7d259EBfb4cc2Ae75115475E9867C': {
      3: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x976A3Fc5d6f7d259EBfb4cc2Ae75115475E9867C",
        'gas': "0x15f90",
        'gasPrice': None,
        'hash': "0x57b30c59fc39a50e1cba90e3099286dfa5aaf60294a629240b5bbec6e2e66576
↪",
        'input': "0x",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
        'nonce': "0x3",
        'to': "0x346FB27dE7E7370008f5da379f74dd49F5f2F80F",
        'transactionIndex': None,
        'value': "0x1f161421c8e0000"
      }]
    },
    '0x9B11bF0459b0c4b2f87f8CEBca4cfc26f294B63A': {
      2: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x9B11bF0459b0c4b2f87f8CEBca4cfc26f294B63A",
        'gas': "0x15f90",
        'gasPrice': None,
        'hash': "0x3a3c0698552eec2455ed3190eac3996feccc806970a4a056106deaf6ceb1e5e3
↪",
        'input': "0x",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
        'nonce': "0x2",
        'to': "0x24a461f25eE6a318BDef7F33De634A67bb67Ac9D",
        'transactionIndex': None,
        'value': "0xebec21ee1da40000"
      }],
      6: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x9B11bF0459b0c4b2f87f8CEBca4cfc26f294B63A",
        'gas': "0x15f90",
        'gasPrice': None,
        'hash': "0xbbcd1e45eae3b859203a04be7d6e1d7b03b222ec1d66dfcc8011dd39794b147e
↪",
        'input': "0x",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
        'nonce': "0x6",
```

(continues on next page)

(continued from previous page)

```
        'to': "0x6368f3f8c2B42435D6C136757382E4A59436a681",
        'transactionIndex': None,
        'value': "0xf9a951af55470000"
      }, {
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x9B11bF0459b0c4b2f87f8CEBca4cfc26f294B63A",
        'gas': "0x15f90",
        'gasPrice': None,
        'hash': "0x60803251d43f072904dc3a2d6a084701cd35b4985790baaf8a8f76696041b272
↪",
        'input': "0x",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
        'nonce': "0x6",
        'to': "0x8DB7b4e0ECB095FBD01Dffa62010801296a9ac78",
        'transactionIndex': None,
        'value': "0xebe866f5f0a06000"
      }],
    }
  }
}
```

2.27 Tracing API

The `web3.tracing` object exposes modules that enable you to interact with the JSON-RPC `trace_` endpoints supported by Erigon and Nethermind.

The following methods are available on the `web3.tracing` namespace:

`web3.tracing.`**`trace_replay_transaction`**`()`

`web3.tracing.`**`trace_replay_block_transactions`**`()`

`web3.tracing.`**`trace_filter`**`()`

`web3.tracing.`**`trace_block`**`()`

`web3.tracing.`**`trace_transaction`**`()`

`web3.tracing.`**`trace_call`**`()`

`web3.tracing.`**`trace_raw_transaction`**`()`

2.28 Utils

The `utils` module houses public utility functions and classes.

2.28.1 ABI

`utils.get_abi_input_names(abi)`

Return the `input` names for an ABI function or event.

`utils.get_abi_output_names(abi)`

Return the `output` names an ABI function or event.

2.28.2 Address

`utils.get_create_address(sender, nonce)`

Return the checksummed contract address generated by using the `CREATE` opcode by a sender address with a given nonce.

`utils.get_create2_address(sender, salt, init_code)`

Return the checksummed contract address generated by using the `CREATE2` opcode by a sender address with a given salt and contract bytecode. See EIP-1014.

2.28.3 Caching

`class utils.SimpleCache`

The main cache class being used internally by web3.py. In some cases, it may prove useful to set your own cache size and pass in your own instance of this class where supported.

2.28.4 Exception Handling

`utils.handle_offchain_lookup(offchain_lookup_payload, transaction)`

Handle `OffchainLookup` reverts on contract function calls manually. For an example, see *CCIP Read support for offchain lookup* within the examples section.

`utils.async_handle_offchain_lookup(offchain_lookup_payload, transaction)`

The async version of the `handle_offchain_lookup()` utility method described above.

2.29 Gas Price API

> **Warning:** Gas price strategy is only supported for legacy transactions. The London fork introduced `maxFeePerGas` and `maxPriorityFeePerGas` transaction parameters which should be used over `gasPrice` whenever possible.

For Ethereum (legacy) transactions, gas price is a delicate property. For this reason, Web3 includes an API for configuring it.

The Gas Price API allows you to define Web3's behaviour for populating the gas price. This is done using a "Gas Price Strategy" - a method which takes the Web3 object and a transaction dictionary and returns a gas price (denominated in wei).

2.29.1 Retrieving gas price

To retrieve the gas price using the selected strategy simply call *generate_gas_price()*

```
>>> web3.eth.generate_gas_price()
20000000000
```

2.29.2 Creating a gas price strategy

A gas price strategy is implemented as a python method with the following signature:

```
def gas_price_strategy(web3, transaction_params=None):
...
```

The method must return a positive integer representing the gas price in wei.

To demonstrate, here is a rudimentary example of a gas price strategy that returns a higher gas price when the value of the transaction is higher than 1 Ether.

```
from web3 import Web3

def value_based_gas_price_strategy(web3, transaction_params):
    if transaction_params['value'] > Web3.to_wei(1, 'ether'):
        return Web3.to_wei(20, 'gwei')
    else:
        return Web3.to_wei(5, 'gwei')
```

2.29.3 Selecting the gas price strategy

The gas price strategy can be set by calling *set_gas_price_strategy()*.

```
from web3 import Web3

def value_based_gas_price_strategy(web3, transaction_params):
    ...

w3 = Web3(...)
w3.eth.set_gas_price_strategy(value_based_gas_price_strategy)
```

2.29.4 Available gas price strategies

`web3.gas_strategies.rpc.`**`rpc_gas_price_strategy`**(*web3*, *transaction_params=None*)

Makes a call to the JSON-RPC eth_gasPrice method which returns the gas price configured by the connected Ethereum node.

`web3.gas_strategies.time_based.`**`construct_time_based_gas_price_strategy`**(*max_wait_seconds*, *sample_size=120*, *probability=98*, *weighted=False*)

Constructs a strategy which will compute a gas price such that the transaction will be mined within a number of seconds defined by `max_wait_seconds` with a probability defined by `probability`. The gas price is computed by sampling `sample_size` of the most recently mined blocks. If `weighted=True`, the block time will be weighted towards more recently mined blocks.

- `max_wait_seconds` The desired maximum number of seconds the transaction should take to mine.
- `sample_size` The number of recent blocks to sample
- `probability` An integer representation of the desired probability that the transaction will be mined within `max_wait_seconds`. 0 means 0% and 100 means 100%.

The following ready to use versions of this strategy are available.

- `web3.gas_strategies.time_based.fast_gas_price_strategy`: Transaction mined within 60 seconds.
- `web3.gas_strategies.time_based.medium_gas_price_strategy`: Transaction mined within 5 minutes.
- `web3.gas_strategies.time_based.slow_gas_price_strategy`: Transaction mined within 1 hour.
- `web3.gas_strategies.time_based.glacial_gas_price_strategy`: Transaction mined within 24 hours.

> **Warning:** Due to the overhead of sampling the recent blocks it is recommended that a caching solution be used to reduce the amount of chain data that needs to be re-fetched for each request.

```python
from web3 import Web3, middleware
from web3.gas_strategies.time_based import medium_gas_price_strategy

w3 = Web3()
w3.eth.set_gas_price_strategy(medium_gas_price_strategy)

w3.middleware_onion.add(middleware.time_based_cache_middleware)
w3.middleware_onion.add(middleware.latest_block_based_cache_middleware)
w3.middleware_onion.add(middleware.simple_cache_middleware)
```

2.30 ENS API

Ethereum Name Service (ENS) has a friendly overview.

Continue below for the detailed specs on each method and class in the ens module.

2.30.1 ens.ens module

class `ens.ens.ENS`(*provider: BaseProvider = <object object>*, *addr: ChecksumAddress = None*, *middlewares: ~typing.Sequence[~typing.Tuple[Middleware, str]] | None = None*)

Quick access to common Ethereum Name Service functions, like getting the address for a name.

Unless otherwise specified, all addresses are assumed to be a *str* in checksum format, like: `"0x314159265dD8dbb310642f98f50C066173C1259b"`

classmethod `from_web3`(*w3: Web3*, *addr: ChecksumAddress = None*) → *ENS*

Generate an ENS instance from a Web3 instance

Parameters

- **w3** (`web3.Web3`) – to infer connection, middleware, and codec information
- **addr** (*hex-string*) – the address of the ENS registry on-chain. If not provided, defaults to the mainnet ENS registry address.

`address`(*name: str*, *coin_type: int | None = None*) → ChecksumAddress | None

Look up the Ethereum address that *name* currently points to.

Parameters

- **name** (*str*) – an ENS name to look up
- **coin_type** (*int*) – if provided, look up the address for this coin type

Raises

- *InvalidName* – if *name* has invalid syntax
- *ResolverNotFound* – if no resolver found for *name*
- *UnsupportedFunction* – if the resolver does not support the `addr()` function

`setup_address`(*name: str*, *address: Address | ChecksumAddress | HexAddress = <object object>*, *coin_type: int | None = None*, *transact: TxParams | None = None*) → HexBytes | None

Set up the name to point to the supplied address. The sender of the transaction must own the name, or its parent name.

Example: If the caller owns `parentname.eth` with no subdomains and calls this method with `sub.parentname.eth`, then `sub` will be created as part of this call.

Parameters

- **name** (*str*) – ENS name to set up
- **address** (*str*) – name will point to this address, in checksum format. If `None`, erase the record. If not specified, name will point to the owner's address.
- **coin_type** (*int*) – if provided, set up the address for this coin type
- **transact** (*dict*) – the transaction configuration, like in *send_transaction()*

Raises

- *InvalidName* – if name has invalid syntax
- *UnauthorizedError* – if 'from' in *transact* does not own *name*

name(*address: ChecksumAddress*) → str | None

Look up the name that the address points to, using a reverse lookup. Reverse lookup is opt-in for name owners.

Parameters
address (*hex-string*) –

setup_name(*name: str*, *address: ChecksumAddress | None = None*, *transact: TxParams | None = None*) → HexBytes

Set up the address for reverse lookup, aka "caller ID". After successful setup, the method name() will return *name* when supplied with *address*.

Parameters

- **name** (*str*) – ENS name that address will point to
- **address** (*str*) – address to set up, in checksum format
- **transact** (*dict*) – the transaction configuration, like in send_transaction()

Raises

- *AddressMismatch* – if the name does not already point to the address
- *InvalidName* – if *name* has invalid syntax
- *UnauthorizedError* – if 'from' in *transact* does not own *name*
- *UnownedName* – if no one owns *name*

owner(*name: str*) → ChecksumAddress

Get the owner of a name. Note that this may be different from the deed holder in the '.eth' registrar. Learn more about the difference between deed and name ownership in the ENS Managing Ownership docs

Parameters
name (*str*) – ENS name to look up

Returns
owner address

Return type
str

setup_owner(*name: str*, *new_owner: ChecksumAddress = <object object>*, *transact: TxParams | None = None*) → ChecksumAddress | None

Set the owner of the supplied name to *new_owner*.

For typical scenarios, you'll never need to call this method directly, simply call *setup_name()* or *setup_address()*. This method does *not* set up the name to point to an address.

If *new_owner* is not supplied, then this will assume you want the same owner as the parent domain.

If the caller owns parentname.eth with no subdomains and calls this method with sub.parentname.eth, then sub will be created as part of this call.

Parameters

- **name** (*str*) – ENS name to set up
- **new_owner** – account that will own *name*. If None, set owner to empty addr. If not specified, name will point to the parent domain owner's address.

- **transact** (*dict*) – the transaction configuration, like in *send_transaction()*

Raises

- *InvalidName* – if *name* has invalid syntax
- *UnauthorizedError* – if 'from' in *transact* does not own *name*

Returns
the new owner's address

resolver(*name: str*) → *Contract* | None
Get the resolver for an ENS name.

Parameters
name (*str*) – The ENS name

get_text(*name: str, key: str*) → str
Get the value of a text record by key from an ENS name.

Parameters

- **name** (*str*) – ENS name to look up
- **key** (*str*) – ENS name's text record key

Returns
ENS name's text record value

Return type
str

Raises

- *UnsupportedFunction* – If the resolver does not support the "0x59d1d43c" interface id
- *ResolverNotFound* – If no resolver is found for the provided name

set_text(*name: str, key: str, value: str, transact: TxParams = None*) → HexBytes
Set the value of a text record of an ENS name.

Parameters

- **name** (*str*) – ENS name
- **key** (*str*) – Name of the attribute to set
- **value** (*str*) – Value to set the attribute to
- **transact** (*dict*) – The transaction configuration, like in *send_transaction()*

Returns
Transaction hash

Return type
HexBytes

Raises

- *UnsupportedFunction* – If the resolver does not support the "0x59d1d43c" interface id
- *ResolverNotFound* – If no resolver is found for the provided name

2.30.2 ens.async_ens module

class ens.async_ens.**AsyncENS**(*provider: AsyncBaseProvider = <object object>, addr: ChecksumAddress = None, middlewares: ~typing.Sequence[~typing.Tuple[AsyncMiddleware, str]] | None = None*)

Quick access to common Ethereum Name Service functions, like getting the address for a name.

Unless otherwise specified, all addresses are assumed to be a *str* in checksum format, like: `"0x314159265dD8dbb310642f98f50C066173C1259b"`

classmethod from_web3(*w3: AsyncWeb3, addr: ChecksumAddress = None*) → *AsyncENS*

Generate an AsyncENS instance with web3

Parameters

- **w3** (`web3.Web3`) – to infer connection information
- **addr** (*hex-string*) – the address of the ENS registry on-chain. If not provided, defaults to the mainnet ENS registry address.

async address(*name: str, coin_type: int | None = None*) → ChecksumAddress | None

Look up the Ethereum address that *name* currently points to.

Parameters

- **name** (*str*) – an ENS name to look up
- **coin_type** (*int*) – if provided, look up the address for this coin type

Raises

InvalidName – if *name* has invalid syntax

async setup_address(*name: str, address: Address | ChecksumAddress | HexAddress = <object object>, coin_type: int | None = None, transact: TxParams | None = None*) → HexBytes | None

Set up the name to point to the supplied address. The sender of the transaction must own the name, or its parent name.

Example: If the caller owns `parentname.eth` with no subdomains and calls this method with `sub.parentname.eth`, then `sub` will be created as part of this call.

Parameters

- **name** (*str*) – ENS name to set up
- **address** (*str*) – name will point to this address, in checksum format. If `None`, erase the record. If not specified, name will point to the owner's address.
- **coin_type** (*int*) – if provided, set up the address for this coin type
- **transact** (*dict*) – the transaction configuration, like in *send_transaction()*

Raises

- *InvalidName* – if `name` has invalid syntax
- *UnauthorizedError* – if `'from'` in *transact* does not own *name*

async name(*address: ChecksumAddress*) → str | None

Look up the name that the address points to, using a reverse lookup. Reverse lookup is opt-in for name owners.

Parameters

address (*hex-string*) –

async setup_name(*name: str*, *address: ChecksumAddress | None = None*, *transact: TxParams | None = None*) → HexBytes

Set up the address for reverse lookup, aka "caller ID". After successful setup, the method `name()` will return *name* when supplied with *address*.

Parameters

- **name** (*str*) – ENS name that address will point to
- **address** (*str*) – address to set up, in checksum format
- **transact** (*dict*) – the transaction configuration, like in `send_transaction()`

Raises

- ***AddressMismatch*** – if the name does not already point to the address
- ***InvalidName*** – if *name* has invalid syntax
- ***UnauthorizedError*** – if `'from'` in *transact* does not own *name*
- ***UnownedName*** – if no one owns *name*

async owner(*name: str*) → ChecksumAddress

Get the owner of a name. Note that this may be different from the deed holder in the '.eth' registrar. Learn more about the difference between deed and name ownership in the ENS Managing Ownership docs

Parameters
name (*str*) – ENS name to look up

Returns
owner address

Return type
str

async setup_owner(*name: str*, *new_owner: ChecksumAddress = <object object>*, *transact: TxParams | None = None*) → ChecksumAddress | None

Set the owner of the supplied name to *new_owner*.

For typical scenarios, you'll never need to call this method directly, simply call `setup_name()` or `setup_address()`. This method does *not* set up the name to point to an address.

If *new_owner* is not supplied, then this will assume you want the same owner as the parent domain.

If the caller owns `parentname.eth` with no subdomains and calls this method with `sub.parentname.eth`, then `sub` will be created as part of this call.

Parameters

- **name** (*str*) – ENS name to set up
- **new_owner** – account that will own *name*. If `None`, set owner to empty addr. If not specified, name will point to the parent domain owner's address.
- **transact** (*dict*) – the transaction configuration, like in `send_transaction()`

Raises

- ***InvalidName*** – if *name* has invalid syntax
- ***UnauthorizedError*** – if `'from'` in *transact* does not own *name*

Returns
the new owner's address

async resolver(*name: str*) → AsyncContract | None

Get the resolver for an ENS name.

Parameters

name (*str*) – The ENS name

async get_text(*name: str, key: str*) → str

Get the value of a text record by key from an ENS name.

Parameters

- **name** (*str*) – ENS name to look up
- **key** (*str*) – ENS name's text record key

Returns

ENS name's text record value

Return type

str

Raises

- *UnsupportedFunction* – If the resolver does not support the "0x59d1d43c" interface id
- *ResolverNotFound* – If no resolver is found for the provided name

async set_text(*name: str, key: str, value: str, transact: TxParams = None*) → HexBytes

Set the value of a text record of an ENS name.

Parameters

- **name** (*str*) – ENS name
- **key** (*str*) – The name of the attribute to set
- **value** (*str*) – Value to set the attribute to
- **transact** (*dict*) – The transaction configuration, like in *send_transaction()*

Returns

Transaction hash

Return type

HexBytes

Raises

- *UnsupportedFunction* – If the resolver does not support the "0x59d1d43c" interface id
- *ResolverNotFound* – If no resolver is found for the provided name

2.30.3 ens.exceptions module

exception ens.exceptions.**ENSException**

Bases: Exception

Base class for all ENS Errors

exception ens.exceptions.**AddressMismatch**

Bases: *ENSException*

In order to set up reverse resolution correctly, the ENS name should first point to the address. This exception is raised if the name does not currently point to the address.

exception ens.exceptions.**InvalidName**

Bases: `IDNAError`, *ENSException*

This exception is raised if the provided name does not meet the normalization standards specified in ENSIP-15.

exception ens.exceptions.**UnauthorizedError**

Bases: *ENSException*

Raised if the sending account is not the owner of the name you are trying to modify. Make sure to set `from` in the `transact` keyword argument to the owner of the name.

exception ens.exceptions.**UnownedName**

Bases: *ENSException*

Raised if you are trying to modify a name that no one owns.

If working on a subdomain, make sure the subdomain gets created first with `setup_address()`.

exception ens.exceptions.**ResolverNotFound**

Bases: *ENSException*

Raised if no resolver was found for the name you are trying to resolve.

exception ens.exceptions.**UnsupportedFunction**

Bases: *ENSException*

Raised if a resolver does not support a particular method.

exception ens.exceptions.**BidTooLow**

Bases: *ENSException*

Raised if you bid less than the minimum amount

exception ens.exceptions.**InvalidBidHash**

Bases: *ENSException*

Raised if you supply incorrect data to generate the bid hash.

exception ens.exceptions.**InvalidLabel**

Bases: *ENSException*

Raised if you supply an invalid label

exception ens.exceptions.**OversizeTransaction**

Bases: *ENSException*

Raised if a transaction you are trying to create would cost so much gas that it could not fit in a block.

For example: when you try to start too many auctions at once.

exception ens.exceptions.**UnderfundedBid**

Bases: *ENSException*

Raised if you send less wei with your bid than you declared as your intent to bid.

exception ens.exceptions.**ENSValidationError**

Bases: *ENSException*, `ValidationError`

Raised if there is a validation error

2.31 Constants

The web3.contants module contains commonly used values.

2.31.1 Strings

```
#The Address Zero, which is 20 bytes (40 nibbles) of zero.
web3.constants.ADDRESS_ZERO

#The hexadecimal version of Max uint256.
web3.constants.MAX_INT

#The Hash Zero, which is 32 bytes (64 nibbles) of zero.
web3.constants.HASH_ZERO
```

2.31.2 Int

```
#The amount of Wei in one Ether
web3.constants.WEI_PER_ETHER
```

2.32 Resources and Learning Material

web3.py and the Ethereum Python ecosystem have an active community of developers and educators. Here you'll find libraries, tutorials, examples, courses and other learning material.

Warning: Links on this page are community submissions and are not vetted by the team that maintains web3.py. As always, DYOR (Do Your Own Research).

2.32.1 First Steps

Resources for those brand new to Ethereum:

- A Developer's Guide to Ethereum, Pt. 1
- Ethereum Python Ecosystem Tour

2.32.2 Courses

- freeCodeCamp Solidity and Python Course (2022)
- Blockchain Python Programming Tutorial (2019)

2.32.3 Tutorials

- Intro to Ape development framework
- Intro to websockets and web3.py
- Intro to asynchronous web3.py
- Intro to threaded web3.py
- Sign typed data messages (EIP 712)
- Look up offchain data via CCIP Read
- Configure and customize web3.py
- Decode a signed transaction
- Find a historical contract revert reason
- Generate a vanity address
- Similate transactions with call state overrides
- Configure web3 for JSON-RPC fallback and MEV blocker providers

2.32.4 Conference Presentations and Videos

- Web3.Py - Now And Near Future by Marc Garreau (2022, 15 mins)
- Python and DeFi by Curve Finance (2022, 15 mins)
- Working with MetaMask in Python by Rishab Kattimani (2022, 15 mins)

2.32.5 Smart Contract Programming Languages

- Vyper - Contract-oriented, pythonic programming language that targets EVM

2.32.6 Frameworks and Tooling

- Ape - The Ethereum development framework for Python Developers, Data Scientists, and Security Professionals
- Titanoboa - A Vyper interpreter and testing framework
- Wake - A Python-based development and testing framework for Solidity
- Brownie - [No longer actively maintained] A Python-based development and testing framework for smart contracts targeting EVM

2.32.7 Libraries

- Web3 Ethereum DeFi - Library for DeFi trading and protocols (Uniswap, PancakeSwap, Sushi, Aave, Chainlink)
- lighter-v1-python - Lighter.xyz DEX client for Python
- uniswap-python - Library lets you easily retrieve prices and make trades on all Uniswap versions.
- pyWalletConnect - WalletConnect implementation for wallets in Python
- dydx-v3-python - Python client for dYdX v3
- Lido Python SDK - Library with which you can get all Lido validator's signatures and check their validity

2.32.8 Applications

- Curve Finance
- Yearn Finance
- StakeWise Oracle

2.32.9 Hackathon Helpers

- ape-hackathon-kit - Ape project template with a web front-end (Next.js, Tailwind, RainbowKit, wagmi)
- eth-flogger - Sample web app utilizing async web3.py, Flask, SQLite, Sourcify
- Temo - Sample terminal app utilizing async web3py, Textual, Anvil
- web3py-discord-bot - Sample Discord bot utilizing websockets, `eth_subscribe`, and discord.py
- py-signer - Demo of typed data message signing (EIP-712) with eth-account and Ape

2.33 Contributing

Thanks for your interest in contributing to web3.py! Read on to learn what would be helpful and how to go about it. If you get stuck along the way, reach for help in the Python Discord server.

2.33.1 How to Help

Without code:

- Answer user questions within GitHub issues, Stack Overflow, or the Python Discord server.
- Write or record tutorial content.
- Improve our documentation (including typo fixes).
- Open an issue on GitHub to document a bug. Include as much detail as possible, e.g., how to reproduce the issue and any exception messages.

With code:

- Fix a bug that has been reported in an issue.
- Add a feature that has been documented in an issue.
- Add a missing test case.

Warning: Before you start: always ask if a change would be desirable or let us know that you plan to work on something! We don't want to waste your time on changes we can't accept or duplicated effort.

2.33.2 Your Development Environment

Note: Use of a virtual environment is strongly advised for minimizing dependency issues. See this article for usage patterns.

All pull requests are made from a fork of the repository; use the GitHub UI to create a fork. web3.py depends on submodules, so when you clone your fork to your local machine, include the `--recursive` flag:

```
$ git clone --recursive https://github.com/<your-github-username>/web3.py.git
$ cd web3.py
```

Finally, install all development dependencies:

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

Using Docker

Developing within Docker is not required, but if you prefer that workflow, use the *sandbox* container provided in the **docker-compose.yml** file.

To start up the test environment, run:

```
$ docker compose up -d
```

This will build a Docker container set up with an environment to run the Python test code.

Note: This container does not have *go-ethereum* installed, so you cannot run the go-ethereum test suite.

To run the Python tests from your local machine:

```
$ docker compose exec sandbox bash -c 'pytest -n 4 -f -k "not goethereum"'
```

You can run arbitrary commands inside the Docker container by using the *bash -c* prefix.

```
$ docker compose exec sandbox bash -c ''
```

Or, if you would like to open a session to the container, run:

```
$ docker compose exec sandbox bash
```

2.33.3 Code Style

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

We use Black for linting. To ignore the commits that introduced Black in git history, you can configure your git environment like so:

```
git config blame.ignoreRevsFile .git-blame-ignore-revs
```

2.33.4 Type Hints

This code base makes use of 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 include type hints, with the exception of tests.

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

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

2.33.5 Running The Tests

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

First, install the test dependencies:

```
$ pip install -e ".[tester]"
```

You can run all tests with:

```
$ pytest
```

However, running the entire test suite takes a very long time and is generally impractical. Typically, you'll just want to run a subset instead, like:

```
$ pytest tests/core/eth-module/test_accounts.py
```

You can use `tox` to run all the tests for a given version of Python:

```
$ tox -e py37-core
```

Linting is also performed by the CI. You can save yourself some time by checking for linting errors locally:

```
$ make lint
```

It is important to understand that each pull request must pass the full test suite as part of the CI check. This test suite will run in the CI anytime a pull request is opened or updated.

2.33.6 Writing Tests

We strongly encourage contributors to write good tests for their code as part of the code review process. This helps ensure that your code is doing what it should be doing.

We strongly encourage you to use our existing tests for both guidance and homogeneity / consistency across our tests. We use `pytest` for our tests. For more specific pytest guidance, please refer to the pytest documentation.

Within the `pytest` scope, `conftest.py` files are used for common code shared between modules that exist within the same directory as that particular `conftest.py` file.

Unit Testing and eth-tester Tests

Our unit tests are grouped together with tests against the `eth-tester` library, using the `py-evm` library as a backend, via the `EthereumTesterProvider`.

These tests live under appropriately named child directories within the `/tests` directory. The core of these tests live under `/tests/core`. Do your best to follow the existing structure when adding a test and make sure that its location makes sense.

Integration Testing

Our integration test suite setup lives under the `/tests/integration` directory. The integration test suite is dependent on what we call "fixtures" (not to be confused with pytest fixtures). These zip file fixtures, which also live in the `/tests/integration` directory, are configured to run the specific client we are testing against along with a genesis configuration that gives our tests some pre-determined useful objects (like unlocked, pre-loaded accounts) to be able to interact with the client when we run our tests.

The parent `/integration` directory houses some common configuration shared across all client tests, whereas the `/go_ethereum` directory houses common code to be shared across geth-specific provider tests. Though the setup and run configurations exist across the different files within `/tests/integration`, our integration module tests are written across different files within `/web3/_utils/module_testing`.

- `common.py` files within the client directories contain code that is shared across all provider tests (http, ipc, and ws). This is mostly used to override tests that span across all providers.
- `conftest.py` files within each of these directories contain mostly code that can be *used* by all test files that exist within the same directory or subdirectories of the `conftest.py` file. This is mostly used to house pytest fixtures to be shared among our tests. Refer to the pytest documentation on fixtures for more information.
- `test_{client}_{provider}.py` files (e.g. `test_goethereum_http.py`) are where client-and-provider-specific test configurations exist. This is mostly used to override tests specific to the provider type for the respective client.

Working With Test Contracts

Contracts used for testing exist under `web3/_utils/contract_sources`. These contracts get compiled via the `compile_contracts.py` script in the same directory. To use this script, simply pass the Solidity version to be used to compile the contracts as an argument at the command line.

Arguments for the script are:

-v or –version Solidity version to be used to compile the contracts. If
blank, the script uses the latest available version from solcx.

-f or –filename If left blank, all .sol files will be compiled and the
respective contract data will be generated. Pass in a specific `.sol` filename here to compile just one file.

To run the script, you will need the `py-solc-x` library for compiling the files as well as `black` for code formatting. You can install those independently or install the full `[dev]` package extra as shown below.

```
$ pip install "web3[dev]"
```

The following example compiles all the contracts and generates their respective contract data that is used across our test files for the test suites. This data gets generated within the `contract_data` subdirectory within the `contract_sources` folder.

```
$ cd ../web3.py/web3/_utils/contract_sources
$ python compile_contracts.py -v 0.8.17
Compiling OffchainLookup
...
...
reformatted ...
```

To compile and generate contract data for only one `.sol` file, specify using the filename with the `-f` (or `--filename`) argument flag.

```
$ cd ../web3.py/web3/_utils/contract_sources
$ python compile_contracts.py -v 0.8.17 -f OffchainLookup.sol
Compiling OffchainLookup.sol
reformatted ...
```

If there is any contract data that is not generated via the script but is important to pass on to the integration tests, the `_custom_contract_data.py` file within the `contract_data` subdirectory can be used to store that information when appropriate.

Be sure to re-generate the integration test fixture after running the script to update the contract bytecodes for the integration test suite - see the *Generating New Fixtures* section below.

2.33.7 Manual Testing

To import and test an unreleased version of web3.py in another context, you can install it from your development directory:

```
$ pip install -e ../path/to/web3py
```

2.33.8 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.

Pull requests generate their own preview of the latest documentation at `https://web3py--<pr-number>.org.readthedocs.build/en/<pr-number>/`.

2.33.9 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.

See GitHub's documentation for working on pull requests.

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.

If the pull request introduces changes that should be reflected in the release notes, please add a **newsfragment** file as explained here.

If possible, the change to the release notes file should be included in the commit that introduces the feature or bugfix.

2.33.10 Generating New Fixtures

Our integration tests make use of Geth private networks. When new versions of the client software are introduced, new fixtures should be generated.

Before generating new fixtures, make sure you have the test dependencies installed:

```
$ pip install -e ".[tester]"
```

Note: A "fixture" is a pre-synced network. It's the result of configuring and running a client, deploying the test contracts, and saving the resulting state for testing web3.py functionality against.

Geth Fixtures

1. Install the desired Geth version on your machine locally. We recommend py-geth for this purpose, because it enables you to easily manage multiple versions of Geth.

   Note that `py-geth` will need updating to support each new Geth version as well. Adding newer Geth versions to py-geth is straightforward; see past commits for a template.

   If py-geth has the Geth version you need, install that version locally. For example:

   ```
   $ python -m geth.install v1.13.9
   ```

2. Specify the Geth binary and run the fixture creation script (from within the web3.py directory):

   ```
   $ GETH_BINARY=~/.py-geth/geth-v1.13.9/bin/geth python ./tests/integration/generate_
   ↪fixtures/go_ethereum.py ./tests/integration/geth-1.13.9-fixture
   ```

3. The output of this script is your fixture, a zip file, which is now stored in `/tests/integration/`. Update the `/tests/integration/go_ethereum/conftest.py` and `/web3/tools/benchmark/node.py` files to point to this new fixture. Delete the old fixture.

4. Run the tests. To ensure that the tests run with the correct Geth version locally, you may again include the `GETH_BINARY` environment variable.

5. Update the `geth_version` and `pygeth_version` parameter defaults in `/.circleci/config.yml` to match the `go-ethereum` version used to generate the test fixture and the `py-geth` version that supports installing it.

CI Testing With a Nightly Geth Build

Occasionally you'll want to have CI run the test suite against an unreleased version of Geth, for example, to test upcoming hard fork changes. The workflow described below is for testing only, i.e., open a PR, let CI run the tests, but the changes should only be merged into main once the Geth release is published or you have some workaround that doesn't require test fixtures built from an unstable client.

1. Configure `tests/integration/generate_fixtures/go_ethereum/common.py` as needed.
2. Geth automagically compiles new builds for every commit that gets merged into the codebase. Download the desired build from the develop builds.
3. Build your test fixture, passing in the binary you just downloaded via `GETH_BINARY`. Don't forget to update the `/tests/integration/go_ethereum/conftest.py` file to point to your new fixture.
4. Our CI runs on Ubuntu, so download the corresponding 64-bit Linux develop build, then add it to the root of your web3.py directory. Rename the binary `custom_geth`.
5. In `.circleci/config.yml`, update jobs relying on `geth_steps`, to instead use `custom_geth_steps`.
6. Create a PR and let CI do its thing.

2.33.11 Releasing

Final Test Before Each Release

Before releasing a new version, build and test the package that will be released. There's a script to build and install the wheel locally, then generate a temporary virtualenv for smoke testing:

```
$ git checkout main && git pull

$ make package

# in another shell, navigate to the virtualenv mentioned in output of ^

# load the virtualenv with the packaged web3.py release
$ source package-smoke-test/bin/activate

# smoke test the release
$ pip install ipython
$ ipython
>>> from web3 import Web3, IPCProvider
>>> w3 = Web3(IPCProvider(provider_url))
>>> w3.is_connected()
>>> ...
```

Verify The Latest Documentation

To preview the documentation that will get published:

```
$ make docs
```

Preview The Release Notes

```
$ towncrier build --draft
```

Compile The Release Notes

After confirming that the release package looks okay, compile the release notes:

```
$ make notes bump=$$VERSION_PART_TO_BUMP$$
```

You may need to fix up any broken release note fragments before committing. Keep running `make build-docs` until it passes, then commit and carry on.

Push The Release to GitHub & PyPI

After committing the compiled release notes and pushing them to the main branch, 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 an alpha version, `make release bump=stage` will bump to beta. If you are in a beta version, `make release bump=stage` will bump to a stable version.

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"`.

2.34 Code of Conduct

2.34.1 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.

2.34.2 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

2.34.3 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.

2.34.4 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.

2.34.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at snakecharmers@ethereum.org. 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.

2.34.6 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

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

B

H

I

T

U

V

W