Populus Documentation

Release 5.20.1

Piper Merriam

Jul 08, 2021

INTRO

1 Getting Started 3

2 Table of Contents 5

3 Indices and tables 227

Python Module Index 229

Index 231

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

Your next steps depend on where you're standing:

- Unfamiliar with Ethereum? → ethereum.org
- Looking for Ethereum Python tutorials? → ethereum.org/python
- Ready to code? → *Quickstart*
- Interested in a quick tour? → *Overview*
- Need help debugging? → StackExchange
- Like to give back? → *Contribute*
- Want to chat? → Discord

CHAPTER

TWO

TABLE OF CONTENTS

2.1 Quickstart

- *Installation*
- *Using Web3*
  - *Provider: Local Geth Node*
  - *Provider: Infura*
- *Getting Blockchain Info*

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.

Provider: Local Geth Node

For locally run nodes, an IPC connection is the most secure option, but HTTP and websocket configurations are also available. By default, Geth 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

# 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.isConnected()
True
```

If you stick to the default ports or IPC file locations, you can utilize a *convenience method* to automatically detect the provider and save a few keystrokes:

```
>>> from web3.auto import w3
>>> w3.isConnected()
True
```

Provider: Infura

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

```
>>> from web3 import Web3
>>> w3 = Web3(Web3.HTTPProvider('https://mainnet.infura.io/v3/<infura-project-id>'))
```

This endpoint is provided by Infura after you create a (free) account.

Again, a convenience method exists to save a few keystrokes:

```
>>> from web3.auto.infura import w3
>>> w3.eth.block_number
4000000
```

Note that this requires your Infura Project ID to be set as the environment variable `WEB3_INFURA_PROJECT_ID` before running your script or application:

```
$ export WEB3_INFURA_PROJECT_ID=YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
```

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(
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪'),
 'miner': '0x0000000000000000000000000000000000000000',
 'mixHash': HexBytes(
↪'0x0000000000000000000000000000000000000000000000000000000000000000'),
 'nonce': HexBytes('0x0000000000000000'),
 'number': 0,
 'parentHash': HexBytes(
↪'0x0000000000000000000000000000000000000000000000000000000000000000'),
 'proofOfAuthorityData': HexBytes(
↪'0x0000000000000000000000000000000000000000000000000000000000000000dddc391ab2bf6701c74d0c8698c2e13
↪'),
 '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.

Many of the typical things you'll want to do will be in the *`w3.eth`* API, so that is a good place to start.

If you want to dive straight into contracts, check out the section on *Contracts*, including a *Contract Deployment Example*, and how to create a contract instance using *`w3.eth.contract()`*.

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 specify the provider and any middleware you want to use beyond the defaults.

Providers

Providers are how Web3.py connects to the blockchain. The library comes with the following built-in providers:

- `Web3.IPCProvider` for connecting to ipc socket based JSON-RPC servers.
- `Web3.HTTPProvider` for connecting to http and https based JSON-RPC servers.
- `Web3.WebsocketProvider` for connecting to ws and wss websocket based JSON-RPC servers.

```
>>> from web3 import Web3

# 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.isConnected()
True
```

For more information, (e.g., connecting to remote nodes, provider auto-detection, using a test provider) 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 Your 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*.

2.2.3 Base API

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

Encoding and Decoding Helpers

- *Web3.is_encodable()*
- *Web3.toBytes()*
- *Web3.toHex()*
- *Web3.toInt()*
- *Web3.toJSON()*
- *Web3.toText()*

Address Helpers

- *Web3.isAddress()*
- *Web3.isChecksumAddress()*
- *Web3.toChecksumAddress()*

Currency Conversions

- *Web3.fromWei()*
- *Web3.toWei()*

Cryptographic Hashing

- *Web3.keccak()*
- *Web3.solidityKeccak()*

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()`

Making Transactions

The most common use cases will be satisfied with `send_transaction` or the combination of `sign_transaction` and `send_raw_transaction`.

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

The two most common use cases involving smart contracts are deploying and executing functions on a deployed contract.

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

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 loaded into a Contract object, the functions of a deployed 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.encodeABI()*
- *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.createFilter(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 *Filtering* 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.createFilter()*
- *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. As an example, instead of `0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359`, you can send funds to `ethereumfoundation.eth`. Web3.py has support for ENS, documented *here*.

2.3 Release Notes

2.3.1 v5.20.1 (2021-07-01)

2.3.2 Web3 5.20.1 (2021-07-01)

Bugfixes

- Have the geth dev IPC auto connection check for the `WEB3_PROVIDER_URI` environment variable. (#2023)

Improved Documentation

- Remove reference to allowing multiple providers in docs (#2018)
- Update "Contract Deployment Example" docs to use `py-solc-x` as `solc` is no longer maintained. (#2020)
- Detail using unreleased Geth builds in CI (#2037)
- Clarify that a missing trie node error could occur when using `block_identifier` with `.call()` on a node that isn't running in archive mode (#2048)

Misc

- #1938, #2015, #2021, #2025, #2028, #2029, #2035

2.3.3 v5.20.0 (2021-06-09)

2.3.4 Web3 5.20.0 (2021-06-09)

Features

- Add new AsyncHTTPProvider. No middleware or session caching support yet.

  Also adds async `w3.eth.gas_price`, and async `w3.isConnected()` methods. (#1978)
- Add ability for AsyncHTTPProvider to accept middleware

  Also adds async gas_price_strategy middleware, and moves gas estimate to middleware.

  AsyncEthereumTesterProvider now inherits from AsyncBase (#1999)
- Support state_override in contract function call. (#2005)

Bugfixes

- Test ethpm caching + bump Sphinx version. (#1977)

Improved Documentation

- Clarify solidityKeccak documentation. (#1971)
- Improve contributor documentation context and ordering. (#2008)
- Add docs for unstable AsyncHTTPProvider (#2017)

Misc

- #1979, #1980, #1993, #2002

2.3.5 v5.19.0 (2021-04-28)

2.3.6 Web3 5.19.0 (2021-04-28)

Features

- Handle optional `eth_call` state override param. (#1921)
- Add list_storage_keys deprecate listStorageKeys (#1944)
- Add net_peers deprecate netPeers (#1946)
- Add trace_replay_transaction deprecate traceReplayTransaction (#1949)
- Add add_reserved_peer deprecate addReservedPeer (#1951)
- Add `parity.set_mode`, deprecate `parity.setMode` (#1954)

- Add `parity.trace_raw_transaction`, deprecate `parity.traceRawTransaction` (#1955)
- Add `parity.trace_call`, deprecate `parity.traceCall` (#1957)
- Add trace_filter deprecate traceFilter (#1960)
- Add trace_block, deprecate traceBlock (#1961)
- Add trace_replay_block_transactions, deprecate traceReplayBlockTransactions (#1962)
- Add `parity.trace_transaction`, deprecate `parity.traceTransaction` (#1963)

Improved Documentation

- Document `eth_call` state overrides. (#1965)

Misc

- #1774, #1805, #1945, #1964

2.3.7 v5.18.0 (2021-04-08)

2.3.8 Web3 5.18.0 (2021-04-08)

Features

- Add `w3.eth.modify_transaction` deprecate `w3.eth.modifyTransaction` (#1886)
- Add `w3.eth.get_transaction_receipt`, deprecate `w3.eth.getTransactionReceipt` (#1893)
- Add `w3.eth.wait_for_transaction_receipt` deprecate `w3.eth.waitForTransactionReceipt` (#1896)
- Add `w3.eth.set_contract_factory` deprecate `w3.eth.setContractFactory` (#1900)
- Add `w3.eth.generate_gas_price` deprecate `w3.eth.generateGasPrice` (#1905)
- Add `w3.eth.set_gas_price_strategy` deprecate `w3.eth.setGasPriceStrategy` (#1906)
- Add `w3.eth.estimate_gas` deprecate `w3.eth.estimateGas` (#1913)
- Add `w3.eth.sign_typed_data` deprecate `w3.eth.signTypedData` (#1915)
- Add `w3.eth.get_filter_changes` deprecate `w3.eth.getFilterChanges` (#1916)
- Add `eth.get_filter_logs`, deprecate `eth.getFilterLogs` (#1919)
- Add `eth.uninstall_filter`, deprecate `eth.uninstallFilter` (#1920)
- Add `w3.eth.get_logs` deprecate `w3.eth.getLogs` (#1925)
- Add `w3.eth.submit_hashrate` deprecate `w3.eth.submitHashrate` (#1926)
- Add `w3.eth.submit_work` deprecate `w3.eth.submitWork` (#1927)
- Add `w3.eth.get_work`, deprecate `w3.eth.getWork` (#1934)
- Adds public get_block_number method. (#1937)

Improved Documentation

- Add ABI type examples to docs (#1890)
- Promote the new Ethereum Python Discord server on the README. (#1898)
- Escape reserved characters in install script of Contributing docs. (#1909)
- Add detailed event filtering examples. (#1910)
- Add docs example for tuning log levels. (#1928)
- Add some performance tips in troubleshooting docs. (#1929)
- Add existing contract interaction to docs examples. (#1933)
- Replace Gitter links with the Python Discord server. (#1936)

Misc

- #1887, #1907, #1917, #1930, #1935

2.3.9 v5.17.0 (2021-02-24)

Features

- Added `get_transaction_count`, and deprecated `getTransactionCount` (#1844)
- Add `w3.eth.send_transaction`, deprecate `w3.eth.sendTransaction` (#1878)
- Add `web3.eth.sign_transaction`, deprecate `web3.eth.signTransaction` (#1879)
- Add `w3.eth.send_raw_transaction`, deprecate `w3.eth.sendRawTransaction` (#1880)
- Add `w3.eth.replace_transaction` deprecate `w3.eth.replaceTransaction` (#1882)

Improved Documentation

- Fix return type of `send_transaction` in docs. (#686)

2.3.10 v5.16.0 (2021-02-04)

Features

- Added `get_block_transaction_count`, and deprecated `getBlockTransactionCount` (#1841)
- Move `defaultAccount` to `default_account`. Deprecate `defaultAccount`. (#1848)
- Add `eth.default_block`, deprecate `eth.defaultBlock`. Also adds `parity.default_block`, and deprecates `parity.defaultBlock`. (#1849)
- Add `eth.gas_price`, deprecate `eth.gasPrice` (#1850)
- Added `eth.block_number` property. Deprecated `eth.blockNumber` (#1851)
- Add `eth.chain_id`, deprecate `eth.chainId` (#1852)
- Add `eth.protocol_version`, deprecate `eth.protocolVersion` (#1853)

- Add `eth.get_code`, deprecate `eth.getCode` (#1856)
- Deprecate `eth.getProof`, add `eth.get_proof` (#1857)
- Add `eth.get_transaction`, deprecate `eth.getTransaction` (#1858)
- Add `eth.get_transaction_by_block`, deprecate `eth.getTransactionByBlock` (#1859)
- Add get_uncle_by_block, deprecate getUncleByBlock (#1862)
- Add get_uncle_count, deprecate getUncleCount (#1863)

Bugfixes

- Fix event filter creation if the event ABI contains a `values` key. (#1807)

Improved Documentation

- Remove v5 breaking changes link from the top of the release notes. (#1837)
- Add account creation troubleshooting docs. (#1855)
- Document passing a struct into a contract function. (#1860)
- Add instance configuration troubleshooting docs. (#1865)
- Clarify nonce lookup in sendRawTransaction docs. (#1866)
- Updated docs for web3.eth methods: eth.getTransactionReceipt and eth.waitForTransactionReceipt (#1868)

2.3.11 v5.15.0 (2021-01-15)

Features

- Add `get_storage_at` method and deprecate `getStorageAt`. (#1828)
- Add `eth.get_block` method and deprecate `eth.getBlock`. (#1829)

Bugfixes

- PR #1585 changed the error that was coming back from eth-tester when the Revert opcode was called, which broke some tests in downstream libraries. This PR reverts back to raising the original error. (#1813)
- Added a new `ContractLogicError` for when a contract reverts a transaction. `ContractLogicError` will replace `SolidityError`, in v6. (#1814)

Improved Documentation

- Introduce Beacon API documentation (#1836)

Misc

- #1602, #1827, #1831, #1833, #1834

2.3.12 v5.14.0 (2021-01-05)

Bugfixes

- Remove docs/web3.* from the gitignore to allow for the beacon docs to be added to git, and add `beacon` to the default web3 modules that get loaded. (#1824)
- Remove auto-documenting from the Beacon API (#1825)

Features

- Introduce experimental Ethereum 2.0 beacon node API (#1758)
- Add new get_balance method on Eth class. Deprecated getBalance. (#1806)

Misc

- #1815, #1816

2.3.13 v5.13.1 (2020-12-03)

Bugfixes

- Handle revert reason parsing for Ganache (#1794)

Improved Documentation

- Document Geth and Parity/OpenEthereum fixture generation (#1787)

Misc

- #1778, #1780, #1790, #1791, #1796

2.3.14 v5.13.0 (2020-10-29)

Features

- Raise *SolidityError* exceptions that contain the revert reason when a *call* fails. (#941)

Bugfixes

- Update eth-tester dependency to fix tester environment install version conflict. (#1782)

Misc

- #1757, #1767

2.3.15 v5.12.3 (2020-10-21)

Misc

- #1752, #1759, #1773, #1775

2.3.16 v5.12.2 (2020-10-12)

Bugfixes

- Address the use of multiple providers in the docs (#1701)
- Remove stale connection errors from docs (#1737)
- Allow ENS name resolution for methods that use the `Method` class (#1749)

Misc

- #1727, #1728, #1733, #1735, #1741, #1746, #1748, #1753, #1768

2.3.17 v5.12.1 (2020-09-02)

Misc

- #1708, #1709, #1715, #1722, #1724

2.3.18 v5.12.0 (2020-07-16)

Features

- Update *web3.pm* and *ethpm* module to EthPM v3 specification. (#1652)
- Allow consumer to initialize *HttpProvider* with their own *requests.Session*. This allows the *HttpAdapter* connection pool to be tuned as desired. (#1469)

Improved Documentation

- Use ethpm v3 packages in examples documentation. (#1683)
- Modernize the deploy contract example. (#1679)
- Add contribution guidelines and a code of conduct. (#1691)

Misc

- #1687
- #1690

2.3.19 v5.12.0-beta.3 (2020-07-15)

Bugfixes

- Include ethpm-spec solidity examples in distribution. (#1686)

2.3.20 v5.12.0-beta.2 (2020-07-14)

Bugfixes

- Support ethpm-spec submodule in distributions. (#1682)

Improved Documentation

- Modernize the deploy contract example. (#1679)
- Use ethpm v3 packages in examples documentation. (#1683)

2.3.21 v5.12.0-beta.1 (2020-07-09)

Features

- Allow consumer to initialize *HttpProvider* with their own *requests.Session*. This allows the *HttpAdapter* connection pool to be tuned as desired. (#1469)
- Update *web3.pm* and *ethpm* module to EthPM v3 specification. (#1652)

Bugfixes

- Update outdated reference url in ethpm docs and tests. (#1680)

Improved Documentation

- Add a `getBalance()` example and provide more context for using the *fromWei* and *toWei* utility methods. (#1676)
- Overhaul the Overview documentation to provide a tour of major features. (#1681)

2.3.22 v5.11.1 (2020-06-17)

Bugfixes

- Added formatter rules for eth_tester middleware to allow `getBalance()` by using integer block numbers (#1660)
- Fix type annotations within the `eth.py` module. Several arguments that defaulted to `None` were not declared `Optional`. (#1668)
- Fix type annotation warning when using string URI to instantiate an HTTP or WebsocketProvider. (#1669)
- Fix type annotations within the `web3` modules. Several arguments that defaulted to `None` were not declared `Optional`. (#1670)

Improved Documentation

- Breaks up links into three categories (Intro, Guides, and API) and adds content to the index page: a lib introduction and some "Getting Started" links. (#1671)
- Fills in some gaps in the Quickstart guide and adds provider connection details for local nodes. (#1673)

2.3.23 v5.11.0 (2020-06-03)

Features

- Accept a block identifier in the `Contract.estimateGas` method. Includes a related upgrade of eth-tester to v0.5.0-beta.1. (#1639)
- Introduce a more specific validation error, `ExtraDataLengthError`. This enables tools to detect when someone may be connected to a POA network, for example, and provide a smoother developer experience. (#1666)

Bugfixes

- Correct the type annotations of *FilterParams.address* (#1664)

Improved Documentation

- Corrects the return value of `getTransactionReceipt`, description of caching middleware, and deprecated method names. (#1663)
- Corrects documentation of websocket timeout configuration. (#1665)

2.3.24 v5.10.0 (2020-05-18)

Features

- An update of `eth-tester` includes a change of the default fork from Constantinople to Muir Glacier. #1636

Bugfixes

- `my_contract.events.MyEvent` was incorrectly annotated so that `MyEvent` was marked as a `ContractEvent` instance. Fixed to be a class type, i.e., `Type[ContractEvent]`. (#1646)
- IPCProvider correctly handled `pathlib.Path` input, but warned against its type. Fixed to permit Path objects in addition to strings. (#1647)

Misc

- #1636

2.3.25 v5.9.0 (2020-04-30)

Features

- Upgrade eth-account to use v0.5.2+. eth-account 0.5.2 adds support for hd accounts

  Also had to pin eth-keys to get dependencies to resolve. (#1622)

Bugfixes

- Fix local_filter_middleware new entries bug (#1514)
- ENS `name` and ENS `address` can return `None`. Fixes return types. (#1633)

2.3.26 v5.8.0 (2020-04-23)

Features

- Introduced `list_wallets` method to the `GethPersonal` class. (#1516)
- Added block_identifier parameter to *ContractConstructor.estimateGas* method. (#1588)
- Add snake_case methods to Geth and Parity Personal Modules.

  Deprecate camelCase methods. (#1589)

- Added new weighted keyword argument to the time based gas price strategy.

  If `True`, it will more give more weight to more recent block times. (#1614)

- Adds support for Solidity's new(ish) receive function.

  Adds a new contract API that mirrors the existing fallback API: `contract.receive` (#1623)

Bugfixes

- Fixed hasattr overloader method in the web3.ContractEvent, web3.ContractFunction, and web3.ContractCaller classes by implementing a try/except handler that returns False if an exception is raised in the __getattr__ overloader method (since __getattr__ HAS to be called in every __hasattr__ call).

  Created two new Exception classes, 'ABIEventFunctionNotFound' and 'ABIFunctionNotFound', which inherit from both AttributeError and MismatchedABI, and replaced the MismatchedABI raises in ContractEvent, ContractFunction, and ContractCaller with a raise to the created class in the __getattr__ overloader method of the object. (#1594)

- Change return type of rpc_gas_price_strategy from int to Wei (#1612)

Improved Documentation

- Fix typo in "Internals" docs. Changed asyncronous –> asynchronous (#1607)
- Improve documentation that introduces and troubleshoots Providers. (#1609)
- Add documentation for when to use each transaction method. (#1610)
- Remove incorrect web3 for w3 in doc example (#1615)
- Add examples for using web3.contract via the ethpm module. (#1617)
- Add dark mode to documentation. Also fixes a bunch of formatting issues in docs. (#1626)

Misc

- #1545

2.3.27 v5.7.0 (2020-03-16)

Features

- Add snake_case methods for the net module

  Also moved net module to use ModuleV2 instead of Module (#1592)

Bugfixes

- Fix return type of eth_getCode. Changed from Hexstr to HexBytes. (#1601)

Misc

- #1590

2.3.28 v5.6.0 (2020-02-26)

Features

- Add snake_case methods to Geth Miner class, deprecate camelCase methods (#1579)
- Add snake_case methods for the net module, deprecate camelCase methods (#1581)
- Add PEP561 type marker (#1583)

Bugfixes

- Increase replacement tx minimum gas price bump

  Parity/OpenEthereum requires a replacement transaction's gas to be a minimum of 12.5% higher than the original (vs. Geth's 10%). (#1570)

2.3.29 v5.5.1 (2020-02-10)

Improved Documentation

- Documents the *getUncleCount* method. (#1534)

Misc

- #1576

2.3.30 v5.5.0 (2020-02-03)

Features

- ENS had to release a new registry to push a bugfix. See this article for background information. Web3.py uses the new registry for all default ENS interactions, now. (#1573)

Bugfixes

- Minor bugfix in how ContractCaller looks up abi functions. (#1552)
- Update modules to use compatible typing-extensions import. (#1554)
- Make 'from' and 'to' fields checksum addresses in returned transaction receipts (#1562)
- Use local Trinity's IPC socket if it is available, for newer versions of Trinity. (#1563)

Improved Documentation

- Add Matomo Tracking to Docs site.

  Matomo is an Open Source web analytics platform that allows us to get better insights and optimize for our audience without the negative consequences of other compareable platforms.

  Read more: https://matomo.org/why-matomo/ (#1541)
- Fix web3 typo in docs (#1559)

Misc

- #1521, #1546, #1571

2.3.31 v5.4.0 (2019-12-06)

Features

- Add __str__ to IPCProvider (#1536)

Bugfixes

- Add required typing-extensions library to setup.py (#1544)

2.3.32 v5.3.1 (2019-12-05)

Bugfixes

- Only apply hexbytes formatting to r and s values in transaction if present (#1531)
- Update eth-utils dependency which contains mypy bugfix. (#1537)

Improved Documentation

- Update Contract Event documentation to show correct example (#1515)
- Add documentation to methods that raise an error in v5 instead of returning `None` (#1527)

Misc

- #1518, #1532

2.3.33 v5.3.0 (2019-11-14)

Features

- Support handling ENS domains in ERC1319 URIs. (#1489)

Bugfixes

- Make local block filter return empty list when when no blocks mined (#1255)
- Google protobuf dependency was updated to *3.10.0* (#1493)
- Infura websocket provider works when no secret key is present (#1501)

Improved Documentation

- Update Quickstart instructions to use the auto Infura module instead of the more complicated web3 auto module (#1482)
- Remove outdated py.test command from readme (#1483)

Misc

- #1461, #1471, #1475, #1476, #1479, #1488, #1492, #1498

2.3.34 v5.2.2 (2019-10-21)

Features

- Add poll_latency to waitForTransactionReceipt (#1453)

Bugfixes

- Fix flaky Parity whisper module test (#1473)

Misc

- #1472, #1474

2.3.35 v5.2.1 (2019-10-17)

Improved Documentation

- Update documentation for unlock account duration (#1464)
- Clarify module installation command for OSX>=10.15 (#1467)

Misc

- #1468

2.3.36 v5.2.0 (2019-09-26)

Features

- Add `enable_strict_bytes_type_checking` flag to web3 instance (#1419)
- Move Geth Whisper methods to snake case and deprecate camel case methods (#1433)

Bugfixes

- Add null check to logsbloom formatter (#1445)

Improved Documentation

- Reformat autogenerated towncrier release notes (#1460)

2.3.37 Web3 5.1.0 (2019-09-18)

Features

- Add `contract_types` property to `Package` class. (#1440)

Bugfixes

- Fix flaky parity integration test in the whisper module (#1147)

Improved Documentation

- Remove whitespace, move `topics` key -> `topic` in Geth docs (#1425)
- Enforce stricter doc checking, turning warnings into errors to fail CI builds to catch issues quickly.

  Add missing `web3.tools.rst` to the table of contents and fix incorrectly formatted JSON example. (#1437)
- Add example using Geth POA Middleware with Infura Rinkeby Node (#1444)

Misc

- #1446, #1451

2.3.38 v5.0.2

Released August 22, 2019

- Bugfixes
  - [ethPM] Fix bug in package id and release id fetching strategy - #1427

2.3.39 v5.0.1

Released August 15, 2019

- Bugfixes
  - [ethPM] Add begin/close chars to package name regex - #1418
  - [ethPM] Update deployments to work when only abi available - #1417
  - Fix tuples handled incorrectly in `decode_function_input` - #1410
- Misc
  - Eliminate `signTransaction` warning - #1404

2.3.40 v5.0.0

Released August 1, 2019

- Features
  - `web3.eth.chainId` now returns an integer instead of hex - #1394
- Bugfixes
  - Deprecation Warnings now show for methods that have a `@combomethod` decorator - #1401
- Misc
  - [ethPM] Add ethPM to the docker file - #1405
- Docs

- Docs are updated to use checksummed addresses - #1390
- Minor doc formatting fixes - #1338 & #1345

2.3.41 v5.0.0-beta.5

Released July 31, 2019

This is intended to be the final release before the stable v5 release.

- Features
  - Parity operating mode can be read and set - #1355
  - Process a single event log, instead of a whole transaction receipt - #1354
- Docs
  - Remove doctest dependency on ethtoken - #1395
- Bugfixes
  - [ethPM] Bypass IPFS validation for large files - #1393
- Misc
  - [ethPM] Update default Registry solidity contract - #1400
  - [ethPM] Update web3.pm to use new simple Registry implementation - #1398
  - Update dependency requirement formatting for releasing - #1403

2.3.42 v5.0.0-beta.4

Released July 18,2019

- Features
  - [ethPM] Update registry uri to support basic uris w/o package id - #1389
- Docs
  - Clarify in docs the return of `Eth.sendRawTransaction()` as a HexBytes object, not a string. - #1384
- Misc
  - [ethPM] Migrate tests over from pytest-ethereum - #1385

2.3.43 v5.0.0-beta.3

Released July 15, 2019

- Features
  - Add eth_getProof support - #1185
  - Implement web3.pm.get_local_package() - #1372
  - Update registry URIs to support chain IDs - #1382
  - Add error flags to `event.processReceipt` - #1366
- Bugfixes

- Remove full IDNA processing in favor of UTS46 - #1364
- Misc
  - Migrate py-ethpm library to web3/ethpm - #1379
  - Relax canonical address requirement in ethPM - #1380
  - Replace ethPM's infura strategy with web3's native infura support - #1383
  - Change `combine_argument_formatters` to `apply_formatters_to_sequence` - #1360
  - Move `pytest.xfail` instances to `@pytest.mark.xfail` - #1376
  - Change `net.version` to `eth.chainId` in default transaction params - #1378

2.3.44 v5.0.0-beta.2

Released May 13, 2019

- Features
  - Mark deprecated sha3 method as static - #1350
  - Upgrade to eth-account v0.4.0 - #1348
- Docs
  - Add note about web3[tester] in documentation - #1325
- Misc
  - Replace `web3._utils.toolz` imports with `eth_utils.toolz` - #1317

2.3.45 v5.0.0-beta.1

Released May 6, 2019

- Features
  - Add support for tilda in provider IPC Path - #1049
  - EIP 712 Signing Supported - #1319
- Docs
  - Update contract example to use `compile_standard` - #1263
  - Fix typo in middleware docs - #1339

2.3.46 v5.0.0-alpha.11

Released April 24, 2019

- Docs
  - Add documentation for web3.py unit tests - #1324
- Misc
  - Update deprecated collections.abc imports - #1334
  - Fix documentation typo - #1335

- Upgrade eth-tester version - #1332

2.3.47 v5.0.0-alpha.10

Released April 15, 2019

- Features
  - Add getLogs by blockHash - #1269
  - Implement chainId endpoint - #1295
  - Moved non-standard JSON-RPC endpoints to applicable Parity/Geth docs. Deprecated `web3.version` for `web3.api` - #1290
  - Moved Whisper endpoints to applicable Geth or Parity namespace - #1308
  - Added support for Goerli provider - #1286
  - Added addReservedPeer to Parity module - #1311
- Bugfixes
  - Cast gas price values to integers in gas strategies - #1297
  - Missing constructor function no longer ignores constructor args - #1316
- Misc
  - Require eth-utils >= 1.4, downgrade Go version for integration tests - #1310
  - Fix doc build warnings - #1331
  - Zip Fixture data - #1307
  - Update Geth version for integration tests - #1301
  - Remove unneeded testrpc - #1322
  - Add ContractCaller docs to v5 migration guide - #1323

2.3.48 v5.0.0-alpha.9

Released March 26, 2019

- Breaking Changes
  - Raise error if there is no Infura API Key - #1294 & - #1299
- Misc
  - Upgraded Parity version for integration testing - #1292

2.3.49 v5.0.0-alpha.8

Released March 20, 2019

- Breaking Changes
  - Removed `web3/utils` directory in favor of `web3/_utils` - #1282
  - Relocated personal RPC endpoints to Parity and Geth class - #1211
  - Deprecated `web3.net.chainId()`, `web3.eth.getCompilers()`, and `web3.eth.getTransactionFromBlock()`. Removed `web3.eth.enableUnauditedFeatures()` - #1270
  - Relocated eth_protocolVersion and web3_clientVersion - #1274
  - Relocated `web3.txpool` to `web3.geth.txpool` - #1275
  - Relocated admin module to Geth namespace - #1288
  - Relocated miner module to Geth namespace - #1287
- Features
  - Implement `eth_submitHashrate` and `eth_submitWork` JSONRPC endpoints. - #1280
  - Implement `web3.eth.signTransaction` - #1277
- Docs
  - Added v5 migration docs - #1284

2.3.50 v5.0.0-alpha.7

Released March 11, 2019

- Breaking Changes
  - Updated JSON-RPC calls that lookup txs or blocks to raise an error if lookup fails - #1218 and #1268
- Features
  - Tuple ABI support - #1235
- Bugfixes
  - One last `middleware_stack` was still hanging on. Changed to `middleware_onion` - #1262

2.3.51 v5.0.0-alpha.6

Released February 25th, 2019

- Features
  - New `NoABIFound` error for cases where there is no ABI - #1247
- Misc
  - Interact with Infura using an API Key. Key will be required after March 27th. - #1232
  - Remove `process_type` utility function in favor of eth-abi functionality - #1249

2.3.52 v5.0.0-alpha.5

Released February 13th, 2019

- Breaking Changes
  - Remove deprecated `buildTransaction`, `call`, `deploy`, `estimateGas`, and `transact` methods - #1232
- Features
  - Adds `Web3.toJSON` method - #1173
  - Contract Caller API Implemented - #1227
  - Add Geth POA middleware to use Rinkeby with Infura Auto - #1234
  - Add manifest and input argument validation to `pm.release_package()` - #1237
- Misc
  - Clean up intro and block/tx sections in Filter docs - #1223
  - Remove unnecessary `EncodingError` exception catching - #1224
  - Improvements to `merge_args_and_kwargs` utility function - #1228
  - Update vyper registry assets - #1242

2.3.53 v5.0.0-alpha.4

Released January 23rd, 2019

- Breaking Changes
  - Rename `middleware_stack` to `middleware_onion` - #1210
  - Drop already deprecated `web3.soliditySha3` - #1217
  - ENS: Stop inferring `.eth` TLD on domain names - #1205
- Bugfixes
  - Validate `ethereum_tester` class in `EthereumTesterProvider` - #1217
  - Support `getLogs()` method without creating filters - #1192
- Features
  - Stablize the `PM` module - #1125
  - Implement async `Version` module - #1166
- Misc
  - Update .gitignore to ignore `.DS_Store` and `.mypy_cache/` - #1215
  - Change CircleCI badge link to CircleCI project - #1214

2.3.54 v5.0.0-alpha.3

Released January 15th, 2019

- Breaking Changes
  - Remove `web3.miner.hashrate` and `web3.version.network` - #1198
  - Remove `web3.providers.tester.EthereumTesterProvider` and `web3.providers.tester.TestRPCProvider` - #1199
  - Change `manager.providers` from list to single `manager.provider` - #1200
  - Replace deprecated `web3.sha3` method with `web3.keccak` method - #1207
  - Drop auto detect testnets for IPCProvider - #1206
- Bugfixes
  - Add check to make sure blockHash exists - #1158
- Misc
  - Remove some unreachable code in *providers/base.py* - #1160
  - Migrate tester provider results from middleware to defaults - #1188
  - Fix doc formatting for build_filter method - #1187
  - Add ERC20 example in docs - #1178
  - Code style improvements - #1194 & #1191
  - Convert Web3 instance variables to w3 - #1186
  - Update eth-utils dependencies and clean up other dependencies - #1195

2.3.55 v5.0.0-alpha.2

Released December 20th, 2018

- Breaking Changes
  - Remove support for python3.5, drop support for eth-abi v1 - #1163
- Features
  - Support for custom ReleaseManager was fixed - #1165
- Misc
  - Fix doctest nonsense with unicorn token - 3b2047
  - Docs for installing web3 in FreeBSD - #1156
  - Use latest python in readthedocs - #1162
  - Use twine in release script - #1164
  - Upgrade eth-tester, for eth-abi v2 support - #1168

2.3.56 v5.0.0-alpha.1

Released December 13th, 2018

- Features
  - Add Rinkeby and Kovan Infura networks; made mainnet the default - #1150
  - Add parity-specific `listStorageKeys` RPC - #1145
  - Deprecated `Web3.soliditySha3`; use `Web3.solidityKeccak` instead. - #1139
  - Add default trinity locations to IPC path guesser - #1121
  - Add wss to `AutoProvider` - #1110
  - Add timeout for `WebsocketProvider` - #1109
  - Receipt timeout raises `TimeExhausted` - #1070
  - Allow specification of block number for `eth_estimateGas` - #1046
- Misc
  - Removed `web3._utils.six` support - #1116
  - Upgrade eth-utils to 1.2.0 - #1104
  - Require Python version 3.5.3 or greater - #1095
  - Bump websockets version to 7.0.0 - #1146
  - Bump parity test binary to 1.11.11 - #1064

2.3.57 v4.8.2

Released November 15, 2018

- Misc
  - Reduce unneeded memory usage - #1138

2.3.58 v4.8.1

Released October 28, 2018

- Features
  - Add timeout for WebsocketProvider - #1119
  - Reject transactions that send ether to non-payable contract functions - #1115
  - Add Auto Infura Ropsten support: `from web3.auto.infura.ropsten import w3` - #1124
  - Auto-detect trinity IPC file location - #1129
- Misc
  - Require Python >=3.5.3 - #1107
  - Upgrade eth-tester and eth-utils - #1085
  - Configure readthedocs dependencies - #1082
  - soliditySha3 docs fixup - #1100

- Update ropsten faucet links in troubleshooting docs

2.3.59 v4.7.2

Released September 25th, 2018

- Bugfixes
    - IPC paths starting with ~ are appropriately resolved to the home directory - #1072
    - You can use the local signing middleware with `bytes`-type addresses - #1069

2.3.60 v4.7.1

Released September 11th, 2018

- Bugfixes
    - old pip bug used during release made it impossible for non-windows users to install 4.7.0.

2.3.61 v4.7.0

Released September 10th, 2018

- Features
    - Add traceFilter method to the parity module. - #1051
    - Move `datastructures` to public namespace `datastructures` to improve support for type checking. - #1038
    - Optimization to contract calls - #944
- Bugfixes
    - ENS name resolution only attempted on mainnet by default. - #1037
    - Fix attribute access error when attributedict middleware is not used. - #1040
- Misc - Upgrade eth-tester to 0.1.0-beta.32, and remove integration tests for py-ethereum. - Upgrade eth-hash to 0.2.0 with pycryptodome 3.6.6 which resolves a vulnerability.

2.3.62 v4.6.0

Released Aug 24, 2018

- Features
    - Support for Python 3.7, most notably in `WebsocketProvider` - #996
    - You can now decode a transaction's data to its original function call and arguments with: `contract.decode_function_input()` - #991
    - Support for `IPCProvider` in FreeBSD (and more readme docs) - #1008
- Bugfixes
    - Fix crash in time-based gas strategies with small number of transactions - #983
    - Fx crash when passing multiple addresses to `w3.eth.getLogs()` - #1005

- Misc
  - Disallow configuring filters with both manual and generated topic lists - #976
  - Add support for the upcoming eth-abi v2, which does ABI string decoding differently - #974
  - Add a lot more filter tests - #997
  - Add more tests for filtering with `None`. Note that geth & parity differ here. - #985
  - Follow-up on Parity bug that we reported upstream (parity#7816): they resolved in 1.10. We removed xfail on that test. - #992
  - Docs: add an example of interacting with an ERC20 contract - #995
  - A couple doc typo fixes
    * #1006
    * #1010

2.3.63 v4.5.0

Released July 30, 2018

- Features
  - Accept addresses supplied in `bytes` format (which does not provide checksum validation)
  - Improve estimation of gas prices
- Bugfixes
  - Can now use a block number with *getCode()* when connected to *EthereumTesterProvider* (without crashing)
- Misc
  - Test Parity 1.11.7
  - Parity integration tests upgrade to use sha256 instead of md5
  - Fix some filter docs
  - eth-account upgrade to v0.3.0
  - eth-tester upgrade to v0.1.0-beta.29

2.3.64 v4.4.1

Released June 29, 2018

- Bugfixes
  - eth-pm package was renamed (old one deleted) which broke the web3 release. eth-pm was removed from the web3.py install until it's stable.
- Misc
  - *IPCProvider* now accepts a `pathlib.Path` argument for the IPC path
  - Docs explaining the *new custom autoproviders in web3*

2.3.65 v4.4.0

Released June 21, 2018

- Features
  - Add support for https in WEB3_PROVIDER_URI environment variable
  - Can send websocket connection parameters in `WebsocketProvider`
  - Two new auto-initialization options:
    * `from web3.auto.gethdev import w3`
    * `from web3.auto.infura import w3` (After setting the `INFURA_API_KEY` environment variable)
  - Alpha support for a new package management tool based on ethpm-spec, see *Package Manager API*
- Bugfixes
  - Can now receive large responses in `WebsocketProvider` by specifying a large `max_size` in the websocket connection parameters.
- Misc
  - Websockets dependency upgraded to v5
  - Raise deprecation warning on `getTransactionFromBlock()`
  - Fix docs for `waitForTransactionReceipt()`
  - Developer Dockerfile now installs testing dependencies

2.3.66 v4.3.0

Released June 6, 2018

- Features
  - Support for the ABI types like: fixedMxN which is used by Vyper.
  - In-flight transaction-signing middleware: Use local keys as if they were hosted keys using the new `sign_and_send_raw_middleware`
  - New `getUncleByBlock()` API
  - New name `getTransactionByBlock()`, which replaces the deprecated `getTransactionFromBlock()`
  - Add several new Parity trace functions
  - New API to resolve ambiguous function calls, for example:
    * Two functions with the same name that accept similar argument types, like `myfunc(uint8)` and `myfunc(int8)`, and you want to call `contract.functions.myfunc(1).call()`
    * See how to use it at: *Invoke Ambiguous Contract Functions Example*
- Bugfixes
  - Gas estimation doesn't crash, when 0 blocks are available. (ie~ on the genesis block)
  - Close out all HTTPProvider sessions, to squash warnings on exit
  - Stop adding Contract address twice to the filter. It was making some nodes unhappy

- Misc
  - Friendlier json encoding/decoding failure error messages
  - Performance improvements, when the responses from the node are large (by reducing the number of times we evaluate if the response is valid json)
  - Parity CI test fixes (ugh, environment setup hell, thanks to the community for cleaning this up!)
  - Don't crash when requesting a transaction that was created with the parity bug (which allowed an unsigned transaction to be included, so `publicKey` is `None`)
  - Doc fixes: addresses must be checksummed (or ENS names on mainnet)
  - Enable local integration testing of parity on non-Debian OS
  - README:
    * Testing setup for devs
    * Change the build badge from Travis to Circle CI
  - Cache the parity binary in Circle CI, to reduce the impact of their binary API going down
  - Dropped the dot: `py.test` -> `pytest`

2.3.67 v4.2.1

Released May 9, 2018

- Bugfixes
  - When *`getting a transaction`* with data attached and trying to *`modify it`* (say, to increase the gas price), the data was not being reattached in the new transaction.
  - `web3.personal.sendTransaction()` was crashing when using a transaction generated with `buildTransaction()`
- Misc
  - Improved error message when connecting to a geth-style PoA network
  - Improved error message when address is not checksummed
  - Started in on support for `fixedMxN` ABI arguments
  - Lots of documentation upgrades, including:
    * Guide for understanding nodes/networks/connections
    * Simplified Quickstart with notes for common issues
    * A new Troubleshooting section
  - Potential pypy performance improvements (use toolz instead of cytoolz)
  - eth-tester upgraded to beta 24

2.3.68 v4.2.0

Released Apr 25, 2018

- Removed audit warning and opt-in requirement for `w3.eth.account`. See more in: *Working with Local Private Keys*
- Added an API to look up contract functions: `fn = contract.functions['function_name_here']`
- Upgrade Whisper (shh) module to use v6 API
- Bugfix: set 'to' field of transaction to empty when using `transaction = contract.constructor().buildTransaction()`
- You can now specify *nonce* in `buildTransaction()`
- Distinguish between chain id and network id – currently always return *None* for `chainId`
- Better error message when trying to use a contract function that has 0 or >1 matches
- Better error message when trying to install on a python version <3.5
- Installs pypiwin32 during pip install, for a better Windows experience
- Cleaned up a lot of test warnings by upgrading from deprecated APIs, especially from the deprecated `contract.deploy(txn_dict, args=contract_args)` to the new `contract.constructor(*contract_args).transact(txn_dict)`
- Documentation typo fixes
- Better template for Pull Requests

2.3.69 v4.1.0

Released Apr 9, 2018

- New `WebsocketProvider`. If you're looking for better performance than HTTP, check out websockets.
- New `w3.eth.waitForTransactionReceipt()`
- Added name collision detection to ConciseContract and ImplicitContract
- Bugfix to allow fromBlock set to 0 in createFilter, like `contract.events.MyEvent.createFilter(fromBlock=0, ...)`
- Bugfix of ENS automatic connection
- eth-tester support for Byzantium
- New migration guide for v3 -> v4 upgrade
- Various documentation updates
- Pinned eth-account to older version

2.3.70 v4.0.0

Released Apr 2, 2018

- Marked beta.13 as stable
- Documentation tweaks

2.3.71 v4.0.0-beta.13

Released Mar 27, 2018

This is intended to be the final release before the stable v4 release.

- Add support for geth 1.8 (fixed error on `getTransactionReceipt()`)
- You can now call a contract method at a specific block with the `block_identifier` keyword argument, see: `call()`
- In preparation for stable release, disable `w3.eth.account` by default, until a third-party audit is complete & resolved.
- New API for contract deployment, which enables gas estimation, local signing, etc. See `constructor()`.
- Find contract events with *contract.events.$my_event.createFilter()*
- Support auto-complete for contract methods.
- Upgrade most dependencies to stable
  - eth-abi
  - eth-utils
  - hexbytes
  - *not included: eth-tester and eth-account*
- Switch the default EthereumTesterProvider backend from eth-testrpc to eth-tester: `web3.providers.eth_tester.EthereumTesterProvider`
- A lot of documentation improvements
- Test node integrations over a variety of providers
- geth 1.8 test suite

2.3.72 v4.0.0-beta.12

A little hiccup on release. Skipped.

2.3.73 v4.0.0-beta.11

Released Feb 28, 2018

- New methods to modify or replace pending transactions
- A compatibility option for connecting to `geth --dev` – see *Geth-style Proof of Authority*
- A new `web3.net.chainId`
- Create a filter object from an existing filter ID.
- eth-utils v1.0.1 (stable) compatibility

2.3.74 v4.0.0-beta.10

Released Feb 21, 2018

- bugfix: Compatibility with eth-utils v1-beta2 (the incompatibility was causing fresh web3.py installs to fail)
- bugfix: crash when sending the output of `contract.functions.myFunction().buildTransaction()` to `sendTransaction()`. Now, having a chainID key does not crash sendTransaction.
- bugfix: a TypeError when estimating gas like: `contract.functions.myFunction().estimateGas()` is fixed
- Added parity integration tests to the continuous integration suite!
- Some py3 and docs cleanup

2.3.75 v4.0.0-beta.9

Released Feb 8, 2018

- Access event log parameters as attributes
- Support for specifying nonce in eth-tester
- Bugfix dependency conflicts between eth-utils, eth-abi, and eth-tester
- Clearer error message when invalid keywords provided to contract constructor function
- New docs for working with private keys + set up doctests
- First parity integration tests
- replace internal implementation of w3.eth.account with `eth_account.account.Account`

2.3.76 v4.0.0-beta.8

Released Feb 7, 2018, then recalled. It added 32MB of test data to git history, so the tag was deleted, as well as the corresponding release. (Although the release would not have contained that test data)

2.3.77 v4.0.0-beta.7

Released Jan 29, 2018

- Support for `web3.eth.Eth.getLogs()` in eth-tester with py-evm
- Process transaction receipts with Event ABI, using *Contract.events.myEvent(*args, **kwargs).processReceipt(transaction_receipt)* see *Event Log Object* for the new type.
- Add timeout parameter to `web3.providers.ipc.IPCProvider`
- bugfix: make sure *idna* package is always installed
- Replace ethtestrpc with py-evm, in all tests
- Dockerfile fixup
- Test refactoring & cleanup
- Reduced warnings during tests

2.3.78 v4.0.0-beta.6

Released Jan 18, 2018

- New contract function call API: *my_contract.functions.my_func().call()* is preferred over the now deprecated *my_contract.call().my_func()* API.
- A new, sophisticated gas estimation algorithm, based on the https://ethgasstation.info approach. You must opt-in to the new approach, because it's quite slow. We recommend using the new caching middleware. See `web3.gas_strategies.time_based.construct_time_based_gas_price_strategy()`
- New caching middleware that can cache based on time, block, or indefinitely.
- Automatically retry JSON-RPC requests over HTTP, a few times.
- ConciseContract now has the address directly
- Many eth-tester fixes. `web3.providers.eth_tester.main.EthereumTesterProvider` is now a legitimate alternative to `web3.providers.tester.EthereumTesterProvider`.
- ethtest-rpc removed from testing. Tests use eth-tester only, on pyethereum. Soon it will be eth-tester with py-evm.
- Bumped several dependencies, like eth-tester
- Documentation updates

2.3.79 v4.0.0-beta.5

Released Dec 28, 2017

- Improvements to working with eth-tester, using `EthereumTesterProvider`:
  - Bugfix the key names in event logging
  - Add support for `sendRawTransaction()`
- `IPCProvider` now automatically retries on a broken connection, like when you restart your node
- New gas price engine API, laying groundwork for more advanced gas pricing strategies

2.3.80 v4.0.0-beta.4

Released Dec 7, 2017

- New `buildTransaction()` method to prepare contract transactions, offline
- New automatic provider detection, for `w3 = Web3()` initialization
- Set environment variable *WEB3_PROVIDER_URI* to suggest a provider for automatic detection
- New API to set providers like: `w3.providers = [IPCProvider()]`
- Crashfix: *web3.eth.Eth.filter()* when retrieving logs with the argument 'latest'
- Bump eth-tester to v0.1.0-beta.5, with bugfix for filtering by topic
- Removed GPL lib `pylru`, now believed to be in full MIT license compliance.

2.3.81 v4.0.0-beta.3

Released Dec 1, 2017

- Fix encoding of ABI types: `bytes[]` and `string[]`
- Windows connection error bugfix
- Bugfix message signatures that were broken ~1% of the time (zero-pad `r` and `s`)
- Autoinit web3 now produces None instead of raising an exception on `from web3.auto import w3`
- Clearer errors on formatting failure (includes field name that failed)
- Python modernization, removing Py2 compatibility cruft
- Update dependencies with changed names, now:
  - `eth-abi`
  - `eth-keyfile`
  - `eth-keys`
  - `eth-tester`
  - `eth-utils`
- Faster Travis CI builds, with cached geth binary

2.3.82 v4.0.0-beta.2

Released Nov 22, 2017

Bug Fixes:

- *sendRawTransaction()* accepts raw bytes
- *contract()* accepts an ENS name as contract address
- `signTransaction()` returns the expected hash (*after* signing the transaction)
- `Account` methods can all be called statically, like: `Account.sign(...)`
- *getTransactionReceipt()* returns the `status` field as an `int`
- `Web3.soliditySha3()` looks up ENS names if they are supplied with an "address" ABI

- If running multiple threads with the same w3 instance, `ValueError: Recursively called ...` is no longer raised

Plus, various python modernization code cleanups, and testing against geth 1.7.2.

2.3.83 v4.0.0-beta.1

- Python 3 is now required
- ENS names can be used anywhere that a hex address can
- Sign transactions and messages with local private keys
- New filter mechanism: *get_all_entries()* and *get_new_entries()*
- Quick automatic initialization with `from web3.auto import w3`
- All addresses must be supplied with an EIP-55 checksum
- All addresses are returned with a checksum
- Renamed `Web3.toDecimal()` to `toInt()`, see: *Encoding and Decoding Helpers*
- All filter calls are synchronous, gevent integration dropped
- Contract `eventFilter()` has replaced both `Contract.on()` and `Contract.pastEvents()`
- Contract arguments of `bytes` ABI type now accept hex strings.
- Contract arguments of `string` ABI type now accept python `str`.
- Contract return values of `string` ABI type now return python `str`.
- Many methods now return a `bytes`-like object where they used to return a hex string, like in `Web3.sha3()`
- IPC connection left open and reused, rather than opened and closed on each call
- A number of deprecated methods from v3 were removed

2.3.84 3.16.1

- Addition of `ethereum-tester` as a dependency

2.3.85 3.16.0

- Addition of *named* middlewares for easier manipulation of middleware stack.
- Provider middlewares can no longer be modified during runtime.
- Experimental custom ABI normalization API for Contract objects.

2.3.86 3.15.0

- Change docs to use RTD theme
- Experimental new `EthereumTesterProvider` for the `ethereum-tester` library.
- Bugfix for `function` type abi encoding via `ethereum-abi-utils` upgrade to `v0.4.1`
- Bugfix for `Web3.toHex` to conform to RPC spec.

2.3.87 3.14.2

- Fix PyPi readme text.

2.3.88 3.14.1

- Fix PyPi readme text.

2.3.89 3.14.0

- New `stalecheck_middleware`
- Improvements to `Web3.toHex` and `Web3.toText`.
- Improvements to `Web3.sha3` signature.
- Bugfixes for `Web3.eth.sign` api

2.3.90 3.13.5

- Add experimental `fixture_middleware`
- Various bugfixes introduced in middleware API introduction and migration to formatter middleware.

2.3.91 3.13.4

- Bugfix for formatter handling of contract creation transaction.

2.3.92 3.13.3

- Improved testing infrastructure.

2.3.93 3.13.2

- Bugfix for retrieving filter changes for both new block filters and pending transaction filters.

2.3.94 3.13.1

- Fix mispelled `attrdict_middleware` (was spelled `attrdict_middlware`).

2.3.95 3.13.0

- New Middleware API
- Support for multiple providers
- New `web3.soliditySha3`
- Remove multiple functions that were never implemented from the original web3.
- Deprecated `web3.currentProvider` accessor. Use `web3.provider` now instead.
- Deprecated password prompt within `web3.personal.newAccount`.

2.3.96 3.12.0

- Bugfix for abi filtering to correctly handle `constructor` and `fallback` type abi entries.

2.3.97 3.11.0

- All web3 apis which accept `address` parameters now enforce checksums if the address *looks* like it is checksummed.
- Improvements to error messaging with when calling a contract on a node that may not be fully synced
- Bugfix for `web3.eth.syncing` to correctly handle `False`

2.3.98 3.10.0

- Web3 now returns `web3.utils.datastructures.AttributeDict` in places where it previously returned a normal `dict`.
- `web3.eth.contract` now performs validation on the `address` parameter.
- Added `web3.eth.getWork` API

2.3.99 3.9.0

- Add validation for the `abi` parameter of `eth`
- Contract return values of `bytes`, `bytesXX` and `string` are no longer converted to text types and will be returned in their raw byte-string format.

2.3.100 3.8.1

- Bugfix for `eth_sign` double hashing input.
- Removed deprecated `DelegatedSigningManager`
- Removed deprecate `PrivateKeySigningManager`

2.3.101 3.8.0

- Update pyrlp dependency to `>=0.4.7`
- Update eth-testrpc dependency to `>=1.2.0`
- Deprecate `DelegatedSigningManager`
- Deprecate `PrivateKeySigningManager`

2.3.102 3.7.1

- upstream version bump for bugfix in eth-abi-utils

2.3.103 3.7.0

- deprecate `eth.defaultAccount` defaulting to the coinbase account.

2.3.104 3.6.2

- Fix error message from contract factory creation.
- Use `ethereum-utils` for utility functions.

2.3.105 3.6.1

- Upgrade `ethereum-abi-utils` dependency for upstream bugfix.

2.3.106 3.6.0

- Deprecate `Contract.code`: replaced by `Contract.bytecode`
- Deprecate `Contract.code_runtime`: replaced by `Contract.bytecode_runtime`
- Deprecate `abi`, `code`, `code_runtime` and `source` as arguments for the `Contract` object.
- Deprecate `source` as a property of the `Contract` object
- Add `Contract.factory()` API.
- Deprecate the `construct_contract_factory` helper function.

2.3.107 3.5.3

- Bugfix for how `requests` library is used. Now reuses session.

2.3.108 3.5.2

- Bugfix for construction of `request_kwargs` within HTTPProvider

2.3.109 3.5.1

- Allow `HTTPProvider` to be imported from `web3` module.
- make `HTTPProvider` accessible as a property of `web3` instances.

2.3.110 3.5.0

- Deprecate `web3.providers.rpc.RPCProvider`
- Deprecate `web3.providers.rpc.KeepAliveRPCProvider`
- Add new `web3.providers.rpc.HTTPProvider`
- Remove hard dependency on gevent.

2.3.111 3.4.4

- Bugfix for `web3.eth.getTransaction` when the hash is unknown.

2.3.112 3.4.3

- Bugfix for event log data decoding to properly handle dynamic sized values.
- New `web3.tester` module to access extra RPC functionality from `eth-testrpc`

2.3.113 3.4.2

- Fix package so that `eth-testrpc` is not required.

2.3.114 3.4.1

- Force gevent<1.2.0 until this issue is fixed: https://github.com/gevent/gevent/issues/916

2.3.115 3.4.0

- Bugfix for contract instances to respect `web3.eth.defaultAccount`
- Better error reporting when ABI decoding fails for contract method response.

2.3.116 3.3.0

- New `EthereumTesterProvider` now available. Faster test runs than `TestRPCProvider`
- Updated underlying eth-testrpc requirement.

2.3.117 3.2.0

- `web3.shh` is now implemented.
- Introduced `KeepAliveRPCProvider` to correctly recycle HTTP connections and use HTTP keep alive

2.3.118 3.1.1

- Bugfix for contract transaction sending not respecting the `web3.eth.defaultAccount` configuration.

2.3.119 3.1.0

- New DelegatedSigningManager and PrivateKeySigningManager classes.

2.3.120 3.0.2

- Bugfix or IPCProvider not handling large JSON responses well.

2.3.121 3.0.1

- Better RPC compliance to be compatable with the Parity JSON-RPC server.

2.3.122 3.0.0

- `Filter` objects now support controlling the interval through which they poll using the `poll_interval` property

2.3.123 2.9.0

- Bugfix generation of event topics.
- Web3.Iban now allows access to Iban address tools.

2.3.124 2.8.1

- Bugfix for `geth.ipc` path on linux systems.

2.3.125 2.8.0

- **Changes to the `Contract` API:**
  - `Contract.deploy()` parameter arguments renamed to args
  - `Contract.deploy()` now takes args and kwargs parameters to allow constructing with keyword arguments or positional arguments.
  - `Contract.pastEvents` now allows you to specify a `fromBlock or ``toBlock.` Previously these were forced to be `'earliest'` and `web3.eth.blockNumber` respectively.
  - `Contract.call`, `Contract.transact` and `Contract.estimateGas` are now callable as class methods as well as instance methods. When called this way, an address must be provided with the transaction parameter.
  - `Contract.call`, `Contract.transact` and `Contract.estimateGas` now allow specifying an alternate address for the transaction.
- **`RPCProvider` now supports the following constructor arguments.**
  - `ssl` for enabling SSL
  - `connection_timeout` and `network_timeout` for controlling the timeouts for requests.

2.3.126 2.7.1

- Bugfix: Fix KeyError in merge_args_and_kwargs helper fn.

2.3.127 2.7.0

- Bugfix for usage of block identifiers 'latest', 'earliest', 'pending'
- Sphinx documentation
- Non-data transactions now default to 90000 gas.
- Web3 object now has helpers set as static methods rather than being set at initialization.
- RPCProvider now takes a `path` parameter to allow configuration for requests to go to paths other than `/`.

2.3.128 2.6.0

- TestRPCProvider no longer dumps logging output to stdout and stderr.
- Bugfix for return types of `address[]`
- Bugfix for event data types of `address`

2.3.129 2.5.0

- All transactions which contain a `data` element will now have their gas automatically estimated with 100k additional buffer. This was previously only true with transactions initiated from a `Contract` object.

2.3.130 2.4.0

- Contract functions can now be called using keyword arguments.

2.3.131 2.3.0

- Upstream fixes for filters
- Filter APIs `on` and `pastEvents` now callable as both instance and class methods.

2.3.132 2.2.0

- The filters that come back from the contract `on` and `pastEvents` methods now call their callbacks with the same data format as `web3.js`.

2.3.133 2.1.1

- Cast RPCProvider port to an integer.

2.3.134 2.1.0

- Remove all monkeypatching

2.3.135 2.0.0

- Pull in downstream updates to proper gevent usage.
- Fix `eth_sign`
- Bugfix with contract operations mutating the transaction object that is passed in.
- More explicit linting ignore statements.

2.3.136 1.9.0

- BugFix: fix for python3 only `json.JSONDecodeError` handling.

2.3.137 1.8.0

- BugFix: `RPCProvider` not sending a content-type header
- Bugfix: `web3.toWei` now returns an integer instead of a decimal.Decimal

2.3.138 1.7.1

- `TestRPCProvider` can now be imported directly from `web3`

2.3.139 1.7.0

- Add `eth.admin` interface.
- Bugfix: Format the return value of `web3.eth.syncing`
- Bugfix: IPCProvider socket interactions are now more robust.

2.3.140 1.6.0

- Downstream package upgrades for `eth-testrpc` and `ethereum-tester-client` to handle configuration of the Homestead and DAO fork block numbers.

2.3.141 1.5.0

- Rename `web3.contract._Contract` to `web3.contract.Contract` to expose it for static analysis and auto completion tools
- Allow passing string parameters to functions
- Automatically compute gas requirements for contract deployment and
- transactions.
- Contract Filters
- Block, Transaction, and Log filters
- `web3.eth.txpool` interface
- `web3.eth.mining` interface
- Fixes for encoding.

2.3.142 1.4.0

- Bugfix to allow address types in constructor arguments.

2.3.143 1.3.0

- Partial implementation of the `web3.eth.contract` interface.

2.3.144 1.2.0

- Restructure project modules to be more *flat*
- Add ability to run test suite without the *slow* tests.
- Breakup `encoding` utils into smaller modules.
- Basic pep8 formatting.
- Apply python naming conventions to internal APIs
- Lots of minor bugfixes.
- Removal of dead code left behing from `1.0.0` refactor.
- Removal of `web3/solidity` module.

2.3.145 1.1.0

- Add missing `isConnected()` method.
- Add test coverage for `setProvider()`

2.3.146 1.0.1

- Specify missing `pyrlp` and `gevent` dependencies

2.3.147 1.0.0

- Massive refactor to the majority of the app.

2.3.148 0.1.0

- Initial release

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. We won't advocate for any particular node, but list some popular options and some basic details on each.

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.

The most popular self-run node options are:

- geth (go-ethereum)
- parity

You can find a fuller list of node software at ethdocs.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.

The most popular hosted node option is Infura. You can connect to it 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. One test network, Ropsten, is the most similar to the production network. However, spam and mining attacks have happened, which is disruptive when you want to test out a contract.

There are some alternative networks that limit the damage of spam attacks, but they are not standardized across node software. Geth runs their own (Rinkeby), and Parity runs their own (Kovan). See a full comparison in this Stackexchange Q&A.

So roughly, choose this way:

- If using Parity, connect to Kovan
- If using Geth, connect to Rinkeby
- If using a different node, or testing mining, connect to Ropsten

Each of their networks has their 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`
- `HTTPProvider`

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*.

2.5.2 Automatic vs Manual Providers

The `Web3` object will look for the Ethereum node in a few standard locations if no providers are specified. Auto-detection happens when you initialize like so:

```
from web3.auto import w3

# which is equivalent to:

from web3 import Web3
w3 = Web3()
```

Sometimes, web3 cannot automatically detect where your node is.

- If you are not sure which kind of connection method to use, see *Choosing How to Connect to Your Node*.

- If you know the connection method, but not the other information needed to connect (like the path to the IPC file), you will need to look up that information in your node's configuration.
- If you're not sure which node you are using, see *How do I choose which node to use?*

For a deeper dive into how automated detection works, see:

How Automated Detection Works

Web3 attempts to connect to nodes in the following order, using the first successful connection it can make:

1. The connection specified by an environment variable, see *Provider via Environment Variable*
2. *IPCProvider*, which looks for several IPC file locations. `IPCProvider` will not automatically detect a testnet connection, it is suggested that the user instead uses a `w3` instance from `web3.auto.infura` (e.g. `from web3.auto.infura.ropsten import w3`) if they want to auto-detect a testnet.
3. *HTTPProvider*, which attempts to connect to "http://localhost:8545"
4. `None` - if no providers are successful, you can still use Web3 APIs that do not require a connection, like:
   - *Encoding and Decoding Helpers*
   - *Currency Conversions*
   - *Addresses*
   - *Working with Local Private Keys*
   - etc.

Examples Using Automated Detection

Some nodes provide APIs beyond the standards. Sometimes the same information is provided in different ways across nodes. If you want to write code that works across multiple nodes, you may want to look up the node type you are connected to.

For example, the following retrieves the client enode endpoint for both geth and parity:

```
from web3.auto import w3

connected = w3.isConnected()

if connected and w3.clientVersion.startswith('Parity'):
    enode = w3.parity.enode

elif connected and w3.clientVersion.startswith('Geth'):
    enode = w3.geth.admin.nodeInfo['enode']

else:
    enode = None
```

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`

2.5.3 Auto-initialization Provider Shortcuts

There are a couple auto-initialization shortcuts for common providers.

Infura Mainnet

To easily connect to the Infura Mainnet remote node, first register for a free project ID if you don't have one at https://infura.io/register .

Then set the environment variable `WEB3_INFURA_PROJECT_ID` with your Project ID:

```
$ export WEB3_INFURA_PROJECT_ID=YourProjectID
```

If you have checked the box in the Infura UI indicating that requests need an optional secret key, set the environment variable `WEB3_INFURA_API_SECRET`:

```
$ export WEB3_INFURA_API_SECRET=YourProjectSecret
```

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

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

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.isConnected()
True
```

2.5.4 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 per python process, as the HTTPProvider recycles underlying TCP/IP network connections, for better performance.

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 the first IPC file it can find from this list:

- On Linux and FreeBSD:
  - `~/.ethereum/geth.ipc`
  - `~/.local/share/io.parity.ethereum/jsonrpc.ipc`

    - ~/.local/share/trinity/mainnet/ipcs-eth1/jsonrpc.ipc

- On Mac OS:
    - ~/Library/Ethereum/geth.ipc
    - ~/Library/Application Support/io.parity.ethereum/jsonrpc.ipc
    - ~/.local/share/trinity/mainnet/ipcs-eth1/jsonrpc.ipc
- On Windows:
    - \\\.\pipe\geth.ipc
    - \\\.\pipe\jsonrpc.ipc

WebsocketProvider

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))
```

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]`

AutoProvider

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

AsyncHTTPProvider

Warning: This provider is unstable and there are still gaps in functionality. However, it is being actively developed.

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.

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

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

Supported Methods

- *web3.eth.block_number*
- *web3.eth.coinbase*
- *web3.eth.gas_price*
- *web3.eth.estimate_gas()*
- *web3.eth.generate_gas_price()*
- *web3.eth.get_block()*
- *web3.eth.get_transaction()*
- *web3.eth.send_transaction()*

Supported Middleware

- *Gas Price Strategy*

2.6 Examples

- *Looking up blocks*
- *Getting the latest block*
- *Checking the balance of an account*
- *Converting currency denominations*
- *Making transactions*
- *Looking up transactions*
- *Looking up receipts*
- *Working with Contracts*
  - *Interacting with existing contracts*
  - *Deploying new contracts*
- *Working with Contracts via ethPM*
- *Working with an ERC20 Token Contract*
  - *Creating the contract factory*
  - *Querying token metadata*
  - *Query account balances*
  - *Sending tokens*
  - *Creating an approval for external transfers*
  - *Performing an external transfer*
- *Contract Unit Tests in Python*
- *Using Infura Rinkeby Node*
- *Adjusting log levels*
- *Advanced example: Fetching all token transfer events*
  - *eth_getLogs limitations*
  - *Example code*

Here are some common things you might want to do with web3.

2.6.1 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':
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
    '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')
{...}
```

2.6.2 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
```

2.6.3 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.

2.6.4 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 `fromWei()` method to convert that balance to ether (or another denomination).

```
>>> web3.fromWei(3841357360894980500000001, 'ether')
Decimal('3841357.360894980500000001')
```

To convert back to wei, you can use the inverse function, `toWei()`. 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.toWei(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.toWei(Decimal('0.000000005'), 'ether')
5000000000
>>> web3.fromWei(5000000000, 'gwei')
Decimal('5')
```

2.6.5 Making transactions

There are a few options for making transactions:

- *send_transaction()*

  Use this method if:

  – you want to send ether from one account to another.

- *send_raw_transaction()*

  Use this method if:

  – you want to sign the transaction elsewhere, e.g., a hardware wallet.

  – you want to broadcast a transaction through another provider, e.g., Infura.

  – you have some other advanced use case that requires more flexibility.

- *Contract Functions*

  Use these methods if:

  – you want to interact with a contract. Web3.py parses the contract ABI and makes those functions available via the `functions` property.

- *construct_sign_and_send_raw_middleware()*

  Use this middleware if:

  – you want to automate signing when using `w3.eth.send_transaction` or `ContractFunctions`.

Note: The location of your keys (e.g., local or hosted) will have implications on these methods. Read about the differences *here*.

2.6.6 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,
```

(continues on next page)

(continued from previous page)

```
    'gasPrice': None,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'maxFeePerGas': 2000000000,
    'maxPriorityFeePerGas': 1000000000,
    'networkId': None,
    'nonce': 0,
    'publicKey':
↪'0x376fc429acc35e610f75b14bc96242b13623833569a5bb3d72c17be7e51da0bb58e48e2462a59897cead8ab88e78709
↪',
    'r': '0x88ff6cf0fefd94db46111149ae4bfc179e9b94721fffd821d38d16464b3f71d0',
    'raw':
↪'0xf86780862d79883d2000825208945df9b87991262f6ba471f09758cde1c0fc1de734827a69801ca088ff6cf0fefd94d
↪',
    's': '0x45e0aff800961cfce805daef7016b9b675c137a6a41a548f7b60a3484c06a33a',
    'standardV': '0x1',
    'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
    'transactionIndex': 0,
    'v': '0x1c',
    'value': 31337,
}
```

If no transaction for the given hash can be found, then this function will instead return `None`.

2.6.7 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':
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
    'root': '0x96a8e009d2b88b1483e6941e6812e32263b05683fac202abc622a3e31aed1957',
    'transactionHash':
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'transactionIndex': 0,
}
```

If the transaction has not yet been mined then this method will return `None`.

2.6.8 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
```

(continues on next page)

(continued from previous page)

```
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)


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()))

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).estimateGas()
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,
```

(continues on next page)

(continued from previous page)

```
'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}

Was transaction successful?
1
```

2.6.9 Working with Contracts via ethPM

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.

```python
from web3.auto.infura import w3

# 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.

# 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

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.
from web3.auto.infura.goerli import w3 as goerli_w3
goerli_registrar = ens_package.update_w3(goerli_w3)
```

2.6.10 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()
'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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

2.6.11 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:

```
import pytest

from web3 import (
    EthereumTesterProvider,
    Web3,
)


@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):
```

(continues on next page)

(continued from previous page)

```
    # 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);
    #
    #     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 = ""
↪"608060405234801561001057600080fd5b5060408051908101604052806000b81526020017f68656c6c6f20776f726c6400
↪"""  # 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(
```

(continues on next page)

(continued from previous page)

```
        "testing contracts is easy",
    ).transact({
        'from': w3.eth.accounts[1],
    })
    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.getLogs()
    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"
```

2.6.12 Using Infura Rinkeby Node

Import your required libraries

```
from web3 import Web3, HTTPProvider
```

Initialize a web3 instance with an Infura node

```
w3 = Web3(Web3.HTTPProvider("https://rinkeby.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).buildTransaction()
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

2.6.13 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 dislay 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
```

2.6.14 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 heurestics 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
import logging
from abc import ABC, abstractmethod
from typing import Tuple, Optional, Callable, List, Iterable

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
```

(continues on next page)

(continued from previous page)

```
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.

        :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 tranformation.
        """

    @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:
```

(continues on next page)

(continued from previous page)

```
    """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, web3: Web3, contract: Contract, state: EventScannerState,␣
↪events: List, filters: {},
                 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 getLogs
        :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.web3 = web3
        self.state = state
        self.events = events
        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 was 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:
```

(continues on next page)

(continued from previous page)

```
            block_info = self.web3.eth.getBlock(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 a hour in␣
↪Ethereum).
        These heurestics 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
        return self.web3.eth.blockNumber - 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:
```

(continues on next page)

(continued from previous page)

```
            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.web3,
                                                    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

            # 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("Processing event %s, block:%d count:%d", evt["event"],␣
↪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
```

(continues on next page)

(continued from previous page)

```
        * 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␣
↪occured in a blockchain
        and our heuristics try to accelerate block fetching (chunk size) until we see␣
↪the first event.

        These heurestics 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

        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
```

(continues on next page)

(continued from previous page)

```
        # 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(
                "Scanning token transfers for blocks: %d - %d, chunk size %d, last␣
↪chunk scan took %f, last logs found %d",
                current_block, estimated_end_block, chunk_size, last_scan_duration,␣
↪last_logs_found)

            start = time.time()
            actual_end_block, end_block_timestamp, new_entries = self.scan_
↪chunk(current_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
```

(continues on next page)

(continued from previous page)

```
    :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(
                    "Retrying events for block range %d - %d (%d) failed with %s,␣
↪retrying in %s seconds",
                    start_block,
                    end_block,
                    end_block-start_block,
                    e,
                    delay)
                # 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(
        web3,
        event,
        argument_filters: dict,
        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 createFilter.
    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 getLogs: fromBlock")

    # 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.
```

(continues on next page)

(continued from previous page)

```
    # 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 = web3.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,
        address=argument_filters.get("address"),
        argument_filters=argument_filters,
        fromBlock=from_block,
        toBlock=to_block
    )

    logger.debug("Querying eth_getLogs with the following parameters: %s", event_
↪filter_params)

    # Call JSON-RPC API on your Ethereum node.
    # get_logs() returns raw AttributedDict entries
    logs = web3.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 processLog 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 persistant 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
```

(continues on next page)

(continued from previous page)

```
    # 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": [
                {
                    "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")
```

(continues on next page)

(continued from previous page)

```
        except (IOError, json.decoder.JSONDecodeError):
            print("State starting from scratch")
            self.reset()

    def save(self):
        """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()

    web3 = Web3(provider)

    # Prepare stub ERC-20 contract object
    abi = json.loads(ABI)
    ERC20 = web3.eth.contract(abi=abi)

    # Restore/create our persistent state
    state = JSONifiedState()
    state.restore()

    # chain_id: int, web3: Web3, abi: dict, state: EventScannerState, events:␣
↪List, filters: {}, max_chunk_scan_size: int=10000
    scanner = EventScanner(
        web3=web3,
        contract=ERC20,
        state=state,
        events=[ERC20.events.Transfer],
```

(continues on next page)

(continued from previous page)

```
            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 Etherueum 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()
```

2.7 Troubleshooting

2.7.1 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`

2.7.2 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.isConnected()
True
>>> w3.eth.send_transaction(...)
```

Refer to the *Providers* documentation for further help with configuration.

2.7.3 Why isn't my web3 instance connecting to the network?

You can check that your instance is connected via the `isConnected` method:

```
>>> w3.isConnected()
False
```

There's a variety of explanations for why you may see `False` here. 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*.

2.7.4 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*.

2.7.5 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):

- Kovan
- Rinkeby
- Ropsten

2.7.6 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!

2.7.7 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.

    This greatly improves JSON-RPC API access speeds, when fetching
    multiple and large responses.
    """
    provider.decode_rpc_response = _fast_decode_rpc_response
```

2.8 Working with Local Private Keys

2.8.1 Local vs Hosted Nodes

Local Node A local node is started and controlled by you. It is as safe as you keep it. When you run `geth` or `parity` on your machine, you are running a local node.

Hosted Node A hosted node is controlled by someone else. When you connect to Infura, you are connected to a hosted node.

2.8.2 Local vs Hosted Keys

Local Private Key A key is 32 bytes of data that you can use to sign transactions and messages, before sending them to your node. You must use *send_raw_transaction()* when working with local keys, instead of *send_transaction()* .

Hosted Private Key This is a common way to use accounts with local nodes. Each account returned by *w3.eth.accounts* has a hosted private key stored in your node. This allows you to use *send_transaction()*.

> **Warning:** It is unacceptable for a hosted node to offer hosted private keys. It gives other people complete control over your account. "Not your keys, not your Ether" in the wise words of Andreas Antonopoulos.

2.8.3 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`.

2.8.4 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
```

2.8.5 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.auto import w3
>>> from eth_account.messages import encode_defunct

>>> 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(
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7680c19e
↪'))
```

2.8.6 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'
```

2.8.7 Verify a Message from message hash

Sometimes, for historical reasons, you don't have the original message, all you have is the prefixed & hashed message. To verify it, use:

> **Caution:** This method is deprecated, only having a hash typically indicates that you're using some old kind of mechanism. Expect this method to go away in the next major version upgrade.

```
>>> message_hash = '0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750
↪'
>>> signature =
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7680c19e
↪'
>>> w3.eth.account.recoverHash(message_hash, signature=signature)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
```

2.8.8 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.toHex(Web3.toBytes(val).rjust(32, b'\0'))

>>> ec_recover_args = (msghash, v, r, s) = (
...   Web3.toHex(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 =
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7680c19e
↪'

# 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.toHex(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.toBytes(hexstr=hex_signature)
>>> v, hex_r, hex_s = Web3.toInt(sig[-1]), Web3.toHex(sig[:32]), Web3.
↪toHex(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)
```

(continues on next page)

(continued from previous page)

```
>>> ec_recover_args
('0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750',
 28,
 '0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3',
 '0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce')
```

2.8.9 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`.

2.8.10 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,
...     'gasPrice': 234567897654321,
...     'nonce': 0,
...     'chainId': 1
... }
>>> key = '0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318'
>>> signed = w3.eth.account.sign_transaction(transaction, key)
>>> signed.rawTransaction
HexBytes(
↪'0xf86a8086d55698372431831e848094f0109fc8df283027b6285cc889f5aa624eac1f55843b9aca008025a009ebb6ca05
↪')
>>> signed.hash
HexBytes('0xd8f64a42b57be0d565f385378db2f6bf324ce14a594afc05de90436e9ce01f60')
>>> signed.r
4487286261793418179817841024889747115779324305375823110249149479905075174044
>>> signed.s
30785525769477805655994251009256770582792548537338581640010273753578382951464
>>> signed.v
```

(continues on next page)

(continued from previous page)

```
37

# When you run send_raw_transaction, you get back the hash of the transaction:
>>> w3.eth.send_raw_transaction(signed.rawTransaction)
'0xd8f64a42b57be0d565f385378db2f6bf324ce14a594afc05de90436e9ce01f60'
```

2.8.11 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.auto import w3

>>> 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,
... ).buildTransaction({
...     'chainId': 1,
...     'gas': 70000,
...     'gasPrice': w3.toWei('1', 'gwei'),
...     'nonce': nonce,
... })

>>> unicorn_txn
{'value': 0,
 'chainId': 1,
 'gas': 70000,
 'gasPrice': 1000000000,
 'nonce': 0,
 'to': '0xfB6916095ca1df60bB79Ce92cE3Ea74c37c5d359',
 'data':
↪'0xa9059cbb000000000000000000000000fb6916095ca1df60bb79ce92ce3ea74c37c5d359000000000000000
↪'}

>>> 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('0x4795adc6a719fa64fa21822630c0218c04996e2689ded114b6553cef1ae36618')
```

(continues on next page)

(continued from previous page)

```
>>> signed_txn.rawTransaction
HexBytes(
↪'0xf8a980843b9aca008301117094fb6916095ca1df60bb79ce92ce3ea74c37c5d35980b844a9059cbb000000000000000
↪')
>>> signed_txn.r
7104843568152743554992057394334744036860247658813231830421570918634460546288
>>> signed_txn.s
20971591154030974221209741174186570949918731455961098911091818811306894497524
>>> signed_txn.v
37

>>> 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.toHex(w3.keccak(signed_txn.rawTransaction))
'0x4795adc6a719fa64fa21822630c0218c04996e2689ded114b6553cef1ae36618'
```

2.9 Filtering

Note: Most one-liners below assume `w3` to be a *`web3.Web3`* instance; obtainable, for example, with:

```
from web3.auto import w3
```

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

- Pending Transactions: `web3.eth.filter('pending')`
- New Blocks `web3.eth.filter('latest')`
- Event Logs

  Through the contract instance api:

  ```
  event_filter = mycontract.events.myEvent.createFilter(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. It does not work with Infura nodes. To get event logs on Infura or other stateless nodes please see *`web3.contract.ContractEvents`*.

2.9.1 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 it's callback functions.

By default this returns the entry parameter umodified.

Filter.**is_valid_entry**(*entry*)
Hook for subclasses to add additional programatic filtering. The default implementation always returns True.

2.9.2 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()
```

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 *BlockFilter* object.

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

2.9.3 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.createFilter()*, which provides some conveniences for creating event log filters. Refer to the following example:

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

See *web3.contract.Contract.events.your_event_name.createFilter()* 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.

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.

2.9.4 Getting events without setting up a filter

You can query an Ethereum node for direct fetch of events, without creating a filter first. This works on all node types, including Infura.

For examples see `web3.contract.ContractEvents.getLogs()`.

2.9.5 Examples: Listening For Events

Synchronous

```
from web3.auto import w3
import time

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.auto import w3
import asyncio


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.auto import w3
from threading import Thread
import time


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

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

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

- gevent
- twisted

- celery

2.10 Contracts

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

2.10.1 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 run the contract deployment example below:

```
>>> 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';
...         }
...
...         function setGreeting(string memory _greeting) public {
...             greeting = _greeting;
...         }
...
.. .        function greet() view public returns (string memory) {
...             return greeting;
...         }
...     }
...     '''
... )

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

(continues on next page)

(continued from previous page)

```
# 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'
```

2.10.2 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`. See the example in `ConciseContract` for specifying an alternate factory.

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`.

class `web3.contract.`**`ConciseContract`**(*Contract()*)

> **Warning:** Deprecated: This method is deprecated in favor of the `ContractCaller` API or the verbose syntax

This variation of `Contract` is designed for more succinct read access, without making write access more wordy. This comes at a cost of losing access to features like `deploy()` and properties like `address`. It is recommended to use the classic `Contract` for those use cases. Just to be be clear, *ConciseContract* only exposes

contract functions and all other *Contract* class methods and properties are not available with the *ConciseContract* API. This includes but is not limited to `contract.address`, `contract.abi`, and `contract.deploy()`.

Create this type of contract by passing a *Contract* instance to *ConciseContract*:

```
>>> concise = ConciseContract(myContract)
```

This variation invokes all methods as a call, so if the classic contract had a method like `contract.functions.owner().call()`, you could call it with `concise.owner()` instead.

For access to send a transaction or estimate gas, you can add a keyword argument like so:

```
>>> concise.withdraw(amount, transact={'from': eth.accounts[1], 'gas': 100000, ...
↪})

>>>  # which is equivalent to this transaction in the classic contract:

>>> contract.functions.withdraw(amount).transact({'from': eth.accounts[1], 'gas':␣
↪100000, ...})
```

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

> **Warning:** Deprecated: This method is deprecated in favor of the verbose syntax

This variation mirrors *ConciseContract*, but it invokes all methods as a transaction rather than a call, so if the classic contract had a method like `contract.functions.owner.transact()`, you could call it with `implicit.owner()` instead.

Create this type of contract by passing a *Contract* instance to *ImplicitContract*:

```
>>> concise = ImplicitContract(myContract)
```

2.10.3 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 array.

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.**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`.

2.10.4 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).estimateGas(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).estimateGas()
12563
```

classmethod `Contract.constructor`(**args, **kwargs).buildTransaction(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
}
```

(continues on next page)

(continued from previous page)

```
>>> contract_data = token_contract.constructor(web3.eth.coinbase, 12345).
↪buildTransaction(transaction)
>>> web3.eth.send_transaction(contract_data)
```

classmethod Contract.events.your_event_name.**createFilter**(*fromBlock=block, toBlock=block, argument_filters={'arg1': 'value'}, 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 with 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.**deploy**(*transaction=None, args=None*)

> **Warning:** Deprecated: this method is deprecated in favor of `constructor()`, which provides more flexibility.

Construct and send a transaction to deploy the contract.

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 arguments they should be provided as a list via the `args` parameter.

If any of the `args` 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.

classmethod `Contract.`**`encodeABI`**(*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.encodeABI(fn_name="register", args=["rainbows", 10])

↪"0xea87152b000000000000000000000000000000000000000000000000000000000000004000000000000000000000
↪"
```

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.

2.10.5 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 ValidationError

>>> identity_func = ambiguous_contract.get_function_by_signature('identity(uint256,
↪bool)')
>>> identity_func(1, True)
```

(continues on next page)

(continued from previous page)

```
<Function identity(uint256,bool) bound to (1, True)>
>>> identity_func(1, True).call()
1
```

2.10.6 Enabling Strict Checks for Bytes Types

By default, web3 is not very strict when it comes to hex and bytes values. A bytes type will take a hex string, a bytestring, or a regular python string that can be decoded as a hex. Additionally, if an abi specifies a byte size, but the value that gets passed in is less than the specified size, web3 will automatically pad the value. For example, if an abi specifies a type of `bytes4`, web3 will handle all of the following values:

Table 1: Valid byte and hex strings for a 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'`

The following values will raise an error by default:

Table 2: Invalid byte and hex strings for a bytes4 type

Input	Reason
`b'abcde'`	Bytestring with more than 4 bytes
`'0x6162636423'`	Hex string with more than 4 bytes
`2`	Wrong type
`'ah'`	String is not valid hex

However, you may want to be stricter with acceptable values for bytes types. For this you can use the `w3.enable_strict_bytes_type_checking()` method, which is available on the web3 instance. A web3 instance which has had this method invoked will enforce a stricter set of rules on which values are accepted.

- A Python string that is not prefixed with `0x` will throw an error.
- A bytestring whose length not exactly the specified byte size will raise an error.

Table 3: Valid byte and hex strings for a strict bytes4 type

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

Table 4: Invalid byte and hex strings with strict bytes4 type checking

Input	Reason
`''`	Needs to be prefixed with a "0x" to be interpreted as an empty hex string
`'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

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) {
... #          return bytes2Value;
... #      }
... #  }

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

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

>>> tx_hash = ArraysContract.constructor([b'b']).transact()
>>> tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)

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

>>> array_contract.functions.getBytes2Value().call()
[b'b\x00']
>>> array_contract.functions.setBytes2Value([b'a']).transact({'gas': 420000, 'gasPrice
↪': 21000})
HexBytes('0x89f9b3a00651e406c568e85c1d2336c66b4ec40ba82c5e72726fbd072230a41c')
>>> array_contract.functions.getBytes2Value().call()
[b'a\x00']
>>> w3.enable_strict_bytes_type_checking()
>>> array_contract.functions.setBytes2Value([b'a']).transact()
Traceback (most recent call last):
   ...
ValidationError:
Could not identify the intended function with name `setBytes2Value`
```

2.10.7 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.

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)

# 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.`**`estimateGas`**(*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).estimateGas(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).estimateGas()
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.**buildTransaction**(*transaction*)

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

Refer to the following invocation:

```
myContract.functions.myMethod(*args, **kwargs).buildTransaction(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).buildTransaction({'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).buildTransaction({'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).buildTransaction({'gasPrice':␣
↪21000000000})
{
    'to': '0x6Bc272FCFcf89C14cebFC57B8f1543F5137F97dE',
    'data':
↪'0x7cf5dab00000000000000000000000000000000000000000000000000000000000000005',
    'value': 0,
    'gas': 43242,
    'gasPrice': 21000000000,
    '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.**estimateGas**(*transaction*)

Call fallback function and return the gas estimation.

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

Execute fallback function by sending a new public transaction.

Contract.fallback.**buildTransaction**(*transaction*)

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

2.10.8 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().processReceipt(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).processReceipt(transaction_receipt, errors=WARN*)

Extracts the pertinent logs from a transaction receipt.

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

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

(continues on next page)

(continued from previous page)

```
        'blockNumber': 3
    })
)


# Or, if there were errors encountered during processing:
>>> from web3.logs import STRICT, IGNORE, DISCARD, WARN
>>> processed_logs = contract.events.myEvent().processReceipt(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().processReceipt(tx_receipt,␣
↪errors=DISCARD)
>>> assert processed_logs == ()
True
```

`ContractEvents.myEvent(*args, **kwargs).processLog(log)`

Similar to *processReceipt*, 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().processLog(log_to_process)
>>> processed_log
AttributeDict({
    'args': AttributeDict({}),
    'event': 'myEvent',
    'logIndex': 0,
    'transactionIndex': 0,
    'transactionHash': HexBytes(
↪'0xfb95ccb6ab39e19821fb339dee33e7afe2545527725b61c64490a5613f8d11fa'),
    'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
    'blockHash': HexBytes(
↪'0xd74c3e8bdb19337987b987aee0fa48ed43f8f2318edfc84e3a8643e009592a68'),
```

(continues on next page)

(continued from previous page)

```
    '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.createFilter(fromBlock="0x0",␣
↪argument_filters={'from': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf'})
>>> transfer_filter.get_new_entries()
[AttributeDict({'args': AttributeDict({'from':
↪'0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'to': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'value': 10}),
 'event': 'Transfer',
 'logIndex': 0,
 'transactionIndex': 0,
 'transactionHash': HexBytes(
↪'0x0005643c2425552308b4a28814a4dedafb5d340a811b3d2b1c019b290ffd7410'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 2})]
>>> transfer_filter.get_new_entries()
[]
>>> tx_hash = contract.functions.transfer(alice, 10).transact({'gas': 899000,
↪'gasPrice': 200000})
>>> 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(
↪'0xea111a49b82b0a0729d49f9ad924d8f87405d01e3fa87463cf2903848aacf7d9'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 3})]
>>> transfer_filter.get_all_entries()
[AttributeDict({'args': AttributeDict({'from':
↪'0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
```

(continues on next page)

(continued from previous page)

```
 'to': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'value': 10}),
 'event': 'Transfer',
 'logIndex': 0,
 'transactionIndex': 0,
 'transactionHash': HexBytes(
↪'0x0005643c2425552308b4a28814a4dedafb5d340a811b3d2b1c019b290ffd7410'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 2}),
 AttributeDict({'args': AttributeDict({'from':
↪'0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'to': '0x7E5F4552091A69125d5DfCb7b8C2659029395Bdf',
 'value': 10}),
 'event': 'Transfer',
 'logIndex': 0,
 'transactionIndex': 0,
 'transactionHash': HexBytes(
↪'0xea111a49b82b0a0729d49f9ad924d8f87405d01e3fa87463cf2903848aacf7d9'),
 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b',
 'blockHash': HexBytes('...'),
 'blockNumber': 3})]
```

2.10.9 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

↪'0x612e45a3000000000000000000000000b656b2a9c3b2416437a811e07466ca712f5a5b5a00000000000000000000
↪'
>>> 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})
```

2.10.10 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
>>> 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.

2.10.11 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;
```

(continues on next page)

(continued from previous page)

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

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

2.11 ABI Types

The Web3 library follows the following conventions.

2.11.1 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.

2.11.2 Hexadecimal Representations

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

2.11.3 Ethereum Addresses

All addresses must be supplied in one of three ways:

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

2.11.4 Strict Bytes Type Checking

Note: In version 6, this will be the default behavior

There is a method on web3 that will enable stricter bytes type checking. The default is to allow Python strings, and to allow bytestrings less than the specified byte size. To enable stricter checks, use `w3.enable_strict_bytes_type_checking()`. This method will cause the web3 instance to raise an error if a Python string is passed in without a "0x" prefix. It will also raise an error if the byte string or hex string is not the exact number of bytes specified by the ABI type. See the *Enabling Strict Checks for Bytes Types* section for an example and more details.

2.11.5 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;
```

(continues on next page)

(continued from previous page)

```
    // 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()
```

2.12 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).

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

2.12.1 Default Middleware

Some middlewares are added by default if you do not supply any. The defaults are likely to change regularly, so this list may not include the latest version's defaults. You can find the latest defaults in the constructor in `web3/manager.py`

AttributeDict

`web3.middleware.`**`attrdict_middleware`**`()`

This middleware converts the output of a function from a dictionary to an `AttributeDict` which enables dot-syntax access, like `eth.get_block('latest').number` in addition to `eth.get_block('latest')['number']`.

.eth Name Resolution

`web3.middleware.`**`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 if invoked with the mainnet (where the ENS contract is deployed), for all other cases will result in an `InvalidAddress` error

Pythonic

`web3.middleware.pythonic_middleware()`

This converts arguments and returned values to python primitives, where appropriate. For example, it converts the raw hex string returned by the RPC call `eth_blockNumber` into an `int`.

Gas Price Strategy

> **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.

`web3.middleware.gas_price_strategy_middleware()`

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.

HTTPRequestRetry

`web3.middleware.http_retry_request_middleware()`

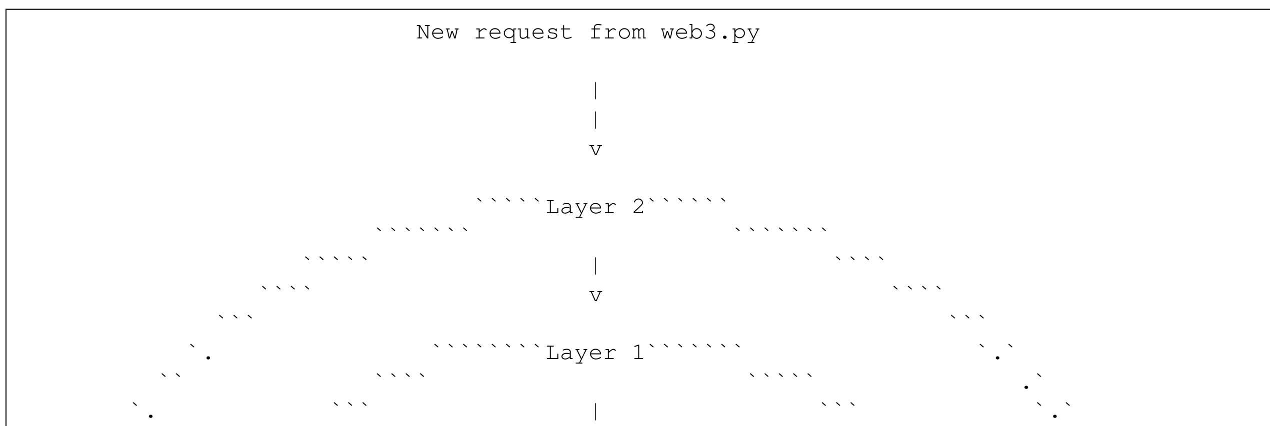
This middleware is a default specifically for HTTPProvider that retries failed requests that return the following errors: `ConnectionError`, `HTTPError`, `Timeout`, `TooManyRedirects`. Additionally there is a whitelist that only allows certain methods to be retried in order to not resend transactions,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

2.12.2 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 or parity) receives and responds to the request inside the innermost layer of the onion. Here is a (simplified) diagram:



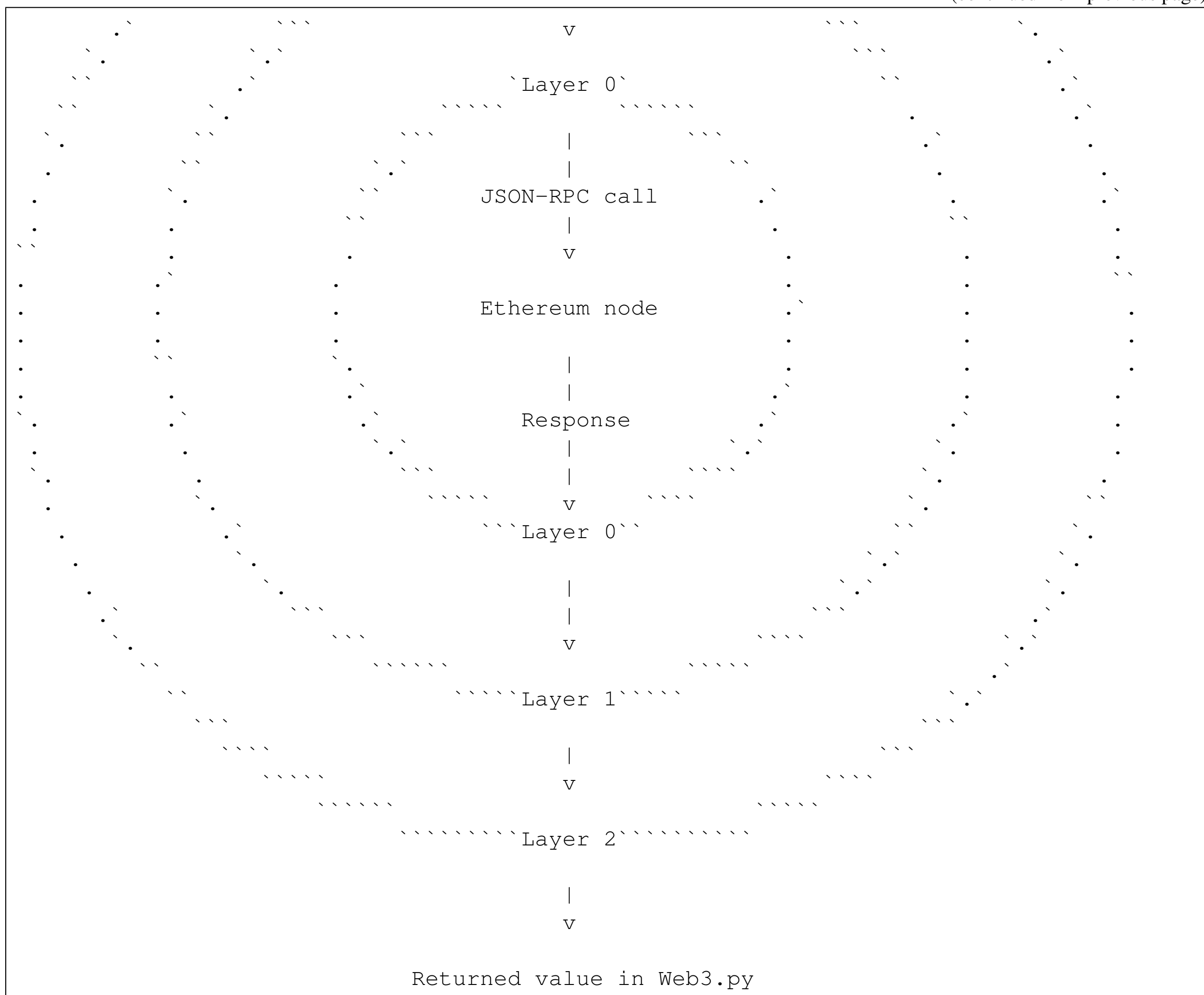
~~~
                         New request from web3.py

                                     |
                                     |
                                     v

                             `````Layer 2``````
                      ```````                  ```````
                 `````                                ````
              ````                   |                    ````
           ```                       v                        ```
         `.                                                      `.`
       ``                 ````````Layer 1```````                  .`
     `.              ````                      `````               `.`
               ```                   |                   ```
~~~

(continues on next page)

(continued from previous page)



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 `list(w3.middleware_onion)` 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.pythonic_middleware)
# or
>>> w3.middleware_onion.add(web3.middleware.pythonic_middleware, 'pythonic')
```

`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 pythonic middleware at the innermost layer
>>> w3 = Web3(...)
>>> w3.middleware_onion.inject(web3.middleware.pythonic_middleware, layer=0)
# or
>>> w3.middleware_onion.inject(web3.middleware.pythonic_middleware, 'pythonic',␣
↪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.pythonic_middleware)
# or
>>> w3.middleware_onion.remove('pythonic')
```

`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 pythonic_middleware, attrdict_middleware
>>> w3 = Web3(...)

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

`Web3.middleware_onion.clear()`

Empty all the middlewares, including the default ones.

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

2.12.3 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 built-in middleware, which is not enabled by default.

Stalecheck

web3.middleware.**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

All of the caching middlewares accept these common arguments.

- `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.
- `should_cache_fn` must be a callable with the signature `fn(method, params, response)` which returns whether the response should be cached.

web3.middleware.**construct_simple_cache_middleware**(*cache_class*, *rpc_whitelist*, *should_cache_fn*)

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

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

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

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.middlewares.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*)

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.middlewares.latest_block_based_cache_middleware`.

Geth-style Proof of Authority

This middleware is required to connect to `geth --dev` or the Rinkeby public network.

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.clientVersion
'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
>>> w3.middleware_onion.inject(geth_poa_middleware, layer=0)

# confirm that the connection succeeded
>>> w3.clientVersion
'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 it's development mode and the Rinkeby 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

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*`)`

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

2.13 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.

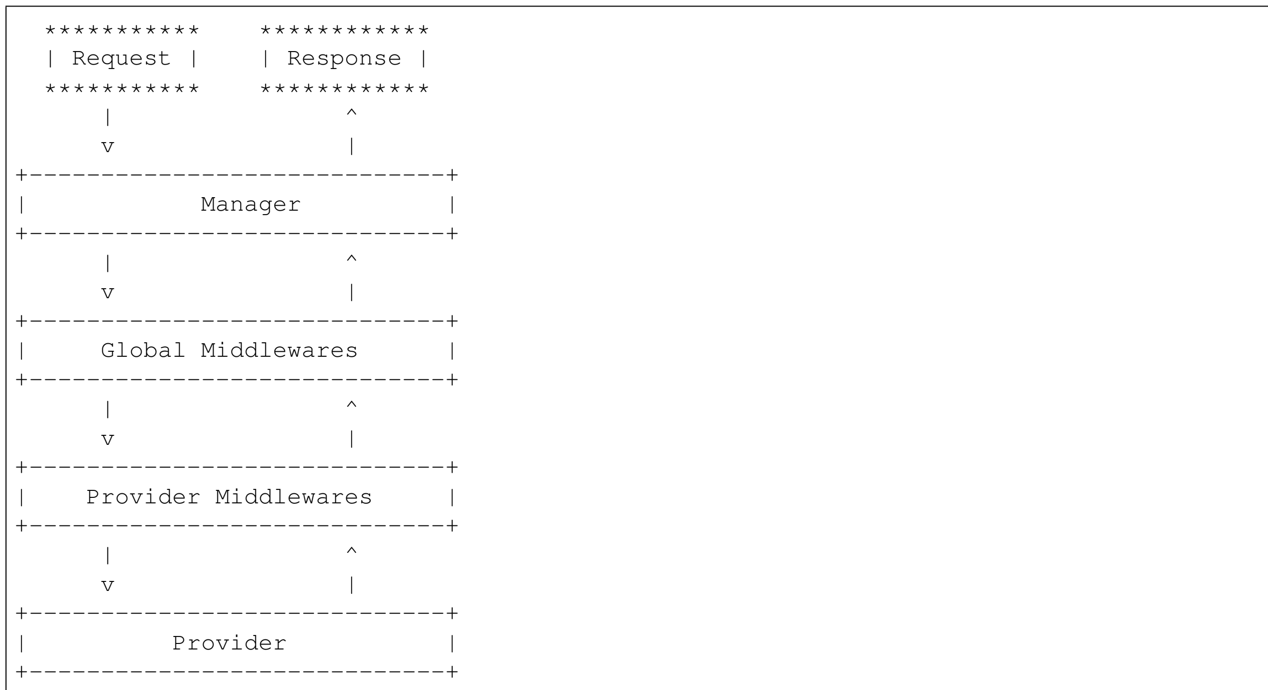
- **Middlewares** provide hooks for monitoring and modifying requests and responses to and from the provider. These can be *global* operating on all providers or specific to one provider.
- **Managers** provide thread safety and primatives to allow for asynchronous usage of web3.

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

2.13.1 Request Lifecycle

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



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

In the situation where web3 is operating with multiple providers the same lifecycle applies. The manager will iterate over each provider, returning the response from the first provider that returns a response.

2.13.2 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.`**`isConnected`**`()`

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.

`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.

2.13.3 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.

By default, Web3 will use the `web3.middleware.pythonic_middleware`. This middleware performs the following translations for requests and responses.

- Numeric request parameters will be converted to their hexadecimal representation
- Numeric responses will be converted from their hexadecimal representations to their integer representations.

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*.

2.13.4 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.

2.14 ethPM

2.14.1 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.

2.14.2 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: Optional[str] = None*)

`__repr__`() → str
String readable representation of the Package.

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

property name
The name of this `Package`.

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

property version
The package version of a `Package`.

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

property manifest_version
The manifest version of a `Package`.

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

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

property contract_types
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: Optional[str] = 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: pathlib.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*) → ethpm.contract.LinkableContract

Return the contract factory for a given contract type, generated from the data vailable 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*) → *web3.contract.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.

2.14.3 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)

>>> # Link Escrow factory to deployed SafeSendLib instance
```

(continues on next page)

(continued from previous page)

```
>>> 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.

2.14.4 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 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 it's 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
- 3: Ropsten
- 4: Rinkeby
- 5: Goerli
- 42: Kovan
- `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://0x808B53bF4D70A24bA5cb720D37A4835621A9df00:1/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`

2.14.5 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(
```

(continues on next page)

(continued from previous page)

```
    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(
...     {},
```

(continues on next page)

(continued from previous page)

```
...     package_name("owned"),
...     manifest_version("ethpm/3"),
...     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 tw

- 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 manfiest 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",
```

(continues on next page)

(continued from previous page)

```
...   "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"]
            }
        }
    }
}
```

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']
>>> 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"
```

(continues on next page)

(continued from previous page)

```
...       }
...     }
... }
>>> # 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"]
...     }
...   }
... }
>>> # 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': {
...       'abi': [{'inputs': [], 'stateMutability': 'nonpayable', 'type': 'constructor
↪'}],
...       'deploymentBytecode': {
...         'bytecode':
↪'0x6080604052348015600f57600080fd5b50600080546001600160a01b031916331790556003f80602f6000396000f3fe6
↪'
...       },
...       '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

- `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 stil default to including all availabe 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': {
...     'blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef': {
...       'Owned': {
...         'contractType': 'Owned',
...         'address': '0x4F5B11C860B37B68De6d14FB7e7b5f18A9a1BD00',
...       }
...     }
...   }
... }
>>> built_manifest = build(
...     BASE_MANIFEST,
...     deployment(
...         block_uri='blockchain://
↪1234567890123456789012345678901234567890123456789012345678901234/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
...         contract_instance='Owned',
...         contract_type='Owned',
...         address='0x4F5B11C860B37B68De6d14FB7e7b5f18A9a1BD00',
...     ),
... )
>>> 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":
↪"0x6080604052348015610010576000080fd5b5060405160208061045383398101604081815291516002819055336000818
↪"
    },
    runtime_bytecode = {
        "bytecode":
↪"0x6080604052600436106100775763ffffffff7c010000000000000000000000000000000000000000000000000000000
↪"
    },
    compiler = {
        "name": "solc",
        "version": "0.4.24+commit.e67f0147.Emscripten.clang",
        "settings": {
            "optimize": True
        }
    }
)
manifest = build(
    package_name("escrow"),
    version("1.0.0"),
    manifest_version("ethpm/3"),
    owned_type(
        block_uri='blockchain://
↪abcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcd/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://
↪abcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcdefabcd/block/
↪1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        address=escrow_testnet_address,
    ),
    escrow_type(
```

(continues on next page)

(continued from previous page)

```
        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',
...   '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
```

2.14.6 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.'}
```

(continues on next page)

(continued from previous page)

2.15 Ethereum Name Service

The Ethereum Name Service 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 an address from a name, set up your own address, and more.

2.15.1 Setup

Create an `ENS` object (named `ns` below) in one of three ways:

1. Automatic detection
2. Specify an instance or list of *Providers*
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
from ens import ENS

w3 = Web3(...)
ns = ENS.fromWeb3(w3)
```

2.15.2 Usage

Name info

Look up the address for an ENS name

```
from ens.auto import ns


# look up the hex representation of the address for a name

eth_address = ns.address('jasoncarver.eth')

assert eth_address == '0x5B2063246F2191f18F2675ceDB8b28102e957458'
```

The ENS module has no opinion as to which TLD you can use, but will not infer a TLD if it is not provided with the name.

Get name from address

```
domain = ns.name('0x5B2063246F2191f18F2675ceDB8b28102e957458')

# name() also accepts the bytes version of the address
assert ns.name(b'[ c$o!\x91\xf1\x8f&u\xce\xdb\x8b(\x10.\x95tX') == domain

# confirm that the name resolves back to the address that you looked up:
assert ns.address(domain) == '0x5B2063246F2191f18F2675ceDB8b28102e957458'
```

Get owner of name

```
eth_address = ns.owner('exchange.eth')
```

Set up your name

Point your name to your address

Do you want to set up your name so that *address()* will show the address it points to?

```
ns.setup_address('jasoncarver.eth', '0x5B2063246F2191f18F2675ceDB8b28102e957458')
```

You must already be the owner of the domain (or its parent).

In the common case where you want to point the name to the owning address, you can skip the address

```
ns.setup_address('jasoncarver.eth')
```

You can claim arbitrarily deep subdomains. *Gas costs scale up with the number of subdomains!*

```
ns.setup_address('supreme.executive.power.derives.from.a.mandate.from.the.masses.
↪jasoncarver.eth')
```

Wait for the transaction to be mined, then:

```
assert ns.address('supreme.executive.power.derives.from.a.mandate.from.the.masses.
↪jasoncarver.eth') == \
    '0x5B2063246F2191f18F2675ceDB8b28102e957458'
```

Allow people to find your name using your address

Do you want to 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.

```
ns.setup_name('jasoncarver.eth', '0x5B2063246F2191f18F2675ceDB8b28102e957458')
```

Note: Do not rely on reverse resolution for security.

Anyone can claim any "caller ID". Only forward resolution implies that the owner of the name gave their stamp of approval.

If you don't supply the address, *setup_name()* will assume you want the address returned by *address()*.

```
ns.setup_name('jasoncarver.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('0x5B2063246F2191f18F2675ceDB8b28102e957458') == 'jasoncarver.eth'
```

2.16 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:

2.16.1 Python 3.5 no longer supported

You will need to upgrade to either Python 3.6 or 3.7

2.16.2 `eth-abi` v1 no longer supported

You will need to upgrade the `eth-abi` dependency to v2

2.16.3 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. Something like `from web3.auto.infura.ropsten import w3` should be used instead.

2.16.4 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.

2.16.5 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.

2.17 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:

2.17.1 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.

2.17.2 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 "New Block: {0}".format(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("New Block: {}".format(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.

2.17.3 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]
```

2.17.4 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()`

2.17.5 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

2.17.6 Personal API

`w3.personal.signAndSendTransaction` is no longer available. Use `w3.personal.sendTransaction()` instead.

2.18 Web3 API

- *Providers*
- *Attributes*
- *Encoding and Decoding Helpers*
- *Currency Conversions*
- *Addresses*

- *Cryptographic Hashing*
- *Check Encodability*
- *RPC APIS*

class web3.**Web3**(*provider*)

Each `web3` instance exposes the following APIs.

2.18.1 Providers

Web3.**HTTPProvider**
Convenience API to access *web3.providers.rpc.HTTPProvider*

Web3.**IPCProvider**
Convenience API to access *web3.providers.ipc.IPCProvider*

2.18.2 Attributes

Web3.**api**
Returns the current Web3 version.

```
>>> web3.api
"4.7.0"
```

Web3.**clientVersion**

- Delegates to `web3_clientVersion` RPC Method

Returns the current client version.

```
>>> web3.clientVersion
'Geth/v1.4.11-stable-fed692f6/darwin/go1.7'
```

2.18.3 Encoding and Decoding Helpers

Web3.**toHex**(*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.toHex(0)
'0x0'
>>> Web3.toHex(1)
'0x1'
>>> Web3.toHex(0x0)
'0x0'
>>> Web3.toHex(0x000F)
'0xf'
>>> Web3.toHex(b'')
'0x'
>>> Web3.toHex(b'\x00\x0F')
'0x000f'
>>> Web3.toHex(False)
'0x0'
```

(continues on next page)

(continued from previous page)

```
>>> Web3.toHex(True)
'0x1'
>>> Web3.toHex(hexstr='0x000F')
'0x000f'
>>> Web3.toHex(hexstr='000F')
'0x000f'
>>> Web3.toHex(text='')
'0x'
>>> Web3.toHex(text='cowmö')
'0x636f776dc3b6'
```

Web3.**toText**(*primitive=None*, *hexstr=None*, *text=None*)

Takes a variety of inputs and returns its string equivalent. Text gets decoded as UTF-8.

```
>>> Web3.toText(0x636f776dc3b6)
'cowmö'
>>> Web3.toText(b'cowm\xc3\xb6')
'cowmö'
>>> Web3.toText(hexstr='0x636f776dc3b6')
'cowmö'
>>> Web3.toText(hexstr='636f776dc3b6')
'cowmö'
>>> Web3.toText(text='cowmö')
'cowmö'
```

Web3.**toBytes**(*primitive=None*, *hexstr=None*, *text=None*)

Takes a variety of inputs and returns its bytes equivalent. Text gets encoded as UTF-8.

```
>>> Web3.toBytes(0)
b'\x00'
>>> Web3.toBytes(0x000F)
b'\x0f'
>>> Web3.toBytes(b'')
b''
>>> Web3.toBytes(b'\x00\x0F')
b'\x00\x0f'
>>> Web3.toBytes(False)
b'\x00'
>>> Web3.toBytes(True)
b'\x01'
>>> Web3.toBytes(hexstr='0x000F')
b'\x00\x0f'
>>> Web3.toBytes(hexstr='000F')
b'\x00\x0f'
>>> Web3.toBytes(text='')
b''
>>> Web3.toBytes(text='cowmö')
b'cowm\xc3\xb6'
```

Web3.**toInt**(*primitive=None*, *hexstr=None*, *text=None*)

Takes a variety of inputs and returns its integer equivalent.

```
>>> Web3.toInt(0)
0
>>> Web3.toInt(0x000F)
15
```

(continues on next page)

(continued from previous page)

```
>>> Web3.toInt(b'\x00\x0F')
15
>>> Web3.toInt(False)
0
>>> Web3.toInt(True)
1
>>> Web3.toInt(hexstr='0x000F')
15
>>> Web3.toInt(hexstr='000F')
15
```

`Web3.toJSON(obj)`
Takes a variety of inputs and returns its JSON equivalent.

```
>>> Web3.toJSON(3)
'3'
>>> Web3.toJSON({'one': 1})
'{"one": 1}'
```

2.18.4 Currency Conversions

`Web3.toWei(value, currency)`
Returns the value in the denomination specified by the `currency` argument converted to wei.

```
>>> Web3.toWei(1, 'ether')
1000000000000000000
```

`Web3.fromWei(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.fromWei(1000000000000000000, 'ether')
Decimal('1')
```

2.18.5 Addresses

`Web3.isAddress(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.isAddress('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
```

`Web3.isChecksumAddress(value)`
Returns `True` if the value is a valid EIP55 checksummed address

```
>>> Web3.isChecksumAddress('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
True
>>> Web3.isChecksumAddress('0xd3cda913deb6f67967b99d67acdfa1712c293601')
False
```

Web3.**toChecksumAddress**(*value*)

Returns the given address with an EIP55 checksum.

```
>>> Web3.toChecksumAddress('0xd3cda913deb6f67967b99d67acdfa1712c293601')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

2.18.6 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')
>>> Web3.keccak(hexstr='747874')
>>> Web3.keccak(text='txt')
HexBytes('0xd7278090a36507640ea6b7a0034b69b0d240766fa3f98e3722be93c613b29d2e')
```

classmethod Web3.**solidityKeccak**(*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.solidityKeccak(['bool'], [True])
HexBytes("0x5fe7f977e71dba2ea1a68e21057beebb9be2ac30c6410aa38d4f3fbe41dcffd2")

>>> Web3.solidityKeccak(['uint8', 'uint8', 'uint8'], [97, 98, 99])
HexBytes("0x4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45")

>>> Web3.solidityKeccak(['uint8[]'], [[97, 98, 99]])
HexBytes("0x233002c671295529bcc50b76a2ef2b0de2dac2d93945fca745255de1a9e4017e")

>>> Web3.solidityKeccak(['address'], ["0x49EdDD3769c0712032808D86597B84ac5c2F5614
↪"])
HexBytes("0x2ff37b5607484cd4eecf6d13292e22bd6e5401eaffcc07e279583bc742c68882")

>>> Web3.solidityKeccak(['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");
```

classmethod Web3.**sha3**(*primitive=None*, *hexstr=None*, *text=None*)

> **Warning:** This method has been deprecated for *keccak()*

Returns the Keccak SHA256 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.sha3(0x747874)
>>> Web3.sha3(b'\x74\x78\x74')
>>> Web3.sha3(hexstr='0x747874')
>>> Web3.sha3(hexstr='747874')
>>> Web3.sha3(text='txt')
HexBytes('0xd7278090a36507640ea6b7a0034b69b0d240766fa3f98e3722be93c613b29d2e')
```

classmethod Web3.**soliditySha3**(*abi_types*, *value*)

Warning: This method has been deprecated for *solidityKeccak()*

Returns the sha3 as it would be computed by the solidity `sha3` function on the provided `value` and `abi_types`. The `abi_types` value should be a list of solidity type strings which correspond to each of the provided values.

```
>>> Web3.soliditySha3(['bool'], [True])
HexBytes("0x5fe7f977e71dba2ea1a68e21057beebb9be2ac30c6410aa38d4f3fbe41dcffd2")

>>> Web3.soliditySha3(['uint8', 'uint8', 'uint8'], [97, 98, 99])
HexBytes("0x4e03657aea45a94fc7d47ba826c8d667c0d1e6e33a64a036ec44f58fa12d6c45")

>>> Web3.soliditySha3(['uint8[]'], [[97, 98, 99]])
HexBytes("0x233002c671295529bcc50b76a2ef2b0de2dac2d93945fca745255de1a9e4017e")

>>> Web3.soliditySha3(['address'], ["0x49EdDD3769c0712032808D86597B84ac5c2F5614"])
HexBytes("0x2ff37b5607484cd4eecf6d13292e22bd6e5401eaffcc07e279583bc742c68882")

>>> Web3.soliditySha3(['address'], ["ethereumfoundation.eth"])
HexBytes("0x913c99ea930c78868f1535d34cd705ab85929b2eaaf70fcd09677ecd6e5d75e9")
```

2.18.7 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', b'1')
True
>>> w3.is_encodable('bytes2', '0x1234')
True
>>> w3.is_encodable('bytes2', b'123')
False
```

w3.**enable_strict_bytes_type_checking**()

Enables stricter bytes type checking. For more examples see *Enabling Strict Checks for Bytes Types*

```
>>> from web3.auto.gethdev import w3
>>> w3.enable_strict_bytes_type_checking()
```

(continues on next page)

(continued from previous page)

```
>>> w3.is_encodable('bytes2', b'12')
True
>>> w3.is_encodable('bytes2', b'1')
False
```

2.18.8 RPC APIS

Each `web3` instance also exposes these namespaced APIs.

`Web3.eth`
: See *web3.eth API*

`Web3.miner`
: See *Miner API*

`Web3.pm`
: See *Package Manager API*

`Web3.geth`
: See *Geth API*

`Web3.parity`
: See *Parity API*

2.19 web3.eth API

class `web3.eth.`**`Eth`**

The `web3.eth` object exposes the following properties and methods to interact with the RPC APIs under the `eth_` namespace.

Often, 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
```

2.19.1 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.

Eth.**defaultAccount**

Warning: Deprecated: This property is deprecated in favor of *default_account*

Eth.**default_block**
The default block number that will be used for any RPC methods that accept a block identifier. Defaults to `'latest'`.

Eth.**defaultBlock**

Warning: Deprecated: This property is deprecated in favor of *default_block*

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
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

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.**gas_price**

- Delegates to `eth_gasPrice` RPC Method

Returns the current gas price in Wei.

```
>>> web3.eth.gas_price
20000000000
```

Eth.**gasPrice**

> **Warning:** Deprecated: This property is deprecated in favor of *gas_price*

Eth.**accounts**

- Delegates to `eth_accounts` RPC Method

Returns the list of known accounts.

```
>>> web3.eth.accounts
['0xd3CdA913deB6f67967B99D67aCDFa1712C293601']
```

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.**blockNumber**

> **Warning:** Deprecated: This property is deprecated in favor of *block_number*

Eth.**protocol_version**

- Delegates to `eth_protocolVersion` RPC Method

Returns the id of the current Ethereum protocol version.

```
>>> web3.eth.protocol_version
'63'
```

Eth.**protocolVersion**

> **Warning:** Deprecated: This property is deprecated in favor of *protocol_version*

`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
```

`Eth.`**`chainId`**

> **Warning:** Deprecated: This property is deprecated in favor of `chain_id`

2.19.2 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.`**`getBalance`**(*account*, *block_identifier=eth.default_block*)

> **Warning:** Deprecated: This method is deprecated in favor of `get_balance()`

`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.getStorageAt`(*account*, *position*, *block_identifier=eth.default_block*)

> **Warning:** Deprecated: This method is deprecated in favor of `get_storage_at()`

`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': [
↪'0xf90211a03841a7ddd65c70c94b8efa79190d00f0ab134b26f18dcad508f60a7e74559d0ba0464b07429a05039e2
↪',

↪'0xf90151a097b17a89fd2c03ee98cb6459c08f51b269da5cee46650e84470f62bf83b43efe80a03b269d284a4c3cf
↪',

↪'0xf869a020d13b52a61d3c1325ce3626a51418adebd6323d4840f1bdd93906359d11c933b846f8440180a01ab7c0b
↪'
                    ],
    'balance': 0,
    'codeHash':
↪'0x551332d96d085185ab4019ad8bcf89c45321e136c261eb6271e574a2edf1461f',
    'nonce': 1,
    'storageHash':
↪'0x1ab7c0b0a2a4bbb5a1495da8c142150891fc64e0c321e1feb70bd5f881951f7e',
    'storageProof': [
        AttributeDict({
            'key': '0x00',
            'value':
↪'0x48656c6c6f00000000000000000000000000000000000000000000000000000a',
            'proof': [
↪'0xf9019180a01ace80e7bed79fbadbe390876bd1a7d9770edf9462049ef8f4b555d05715d53ea049347a3c2eac652
↪',

↪'0xf891808080a0c7d094301e0c54da37b696d85f72de5520b224ab2cf4f045d8db1a3374caf048880a0fc5581783
↪',

↪'0xf843a0200decd9548b62a8d60345a988386fc84ba6bc95484008f6362f93160ef3e563a1a048656c6c6f0000000
↪'
                    ]
        })
    ]
})
```

- 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)
    ), "Failed to verify account proof {}".format(proof.address)

    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)
        ), "Failed to verify storage proof {}".format(storage_proof.key)

    return True
```

(continues on next page)

(continued from previous page)

```
block = w3.eth.get_block(3391)
proof = w3.eth.get_proof('0x6C8f2A135f6ed072DE4503Bd7C4999a1a17F824B', [0, 1],␣
↪3391)
assert verify_eth_get_proof(proof, block.stateRoot)
```

`Eth.getProof(account, positions, block_identifier=eth.default_block)`

> **Warning:** Deprecated: This method is deprecated in favor of `get_proof()`

`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')
'0x6060604052361561027c5760e060020a60003504630199.....'
# For a private key address.
>>> web3.eth.get_code('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
'0x'
```

`Eth.getCode(account, block_identifier=eth.default_block)`

> **Warning:** Deprecated: This method is deprecated in favor of `get_code()`

`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'`, 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':
↪'0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
    'miner': '0x61c808d82a3ac53231750dadc13c777b59310bd9',
```

(continues on next page)

(continued from previous page)

```
    'nonce': '0x3b05c6d5524209f1',
    'number': 2000000,
    'parentHash':
↪'0x57ebf07eb9ed1137d41447020a25e51d30a0c272b5896571499c82c33ecb7288',
    'receiptRoot':
↪'0x84aea4a7aad5c5899bd5cfc7f309cc379009d30179316a2a7baa4a2ea4a438ac',
    'sha3Uncles':
↪'0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
    'size': 650,
    'stateRoot':
↪'0x96dbad955b166f5119793815c36f11ffa909859bbfeb64b735cca37cbf10bef1',
    'timestamp': 1470173578,
    'totalDifficulty': 44010101827705409388,
    'transactions': [
↪'0xc55e2b90168af6972193c1f86fa4d7d7b31a29c156665d15b9cd48618b5177ef'],
    'transactionsRoot':
↪'0xb31f174d27b99cdae8e746bd138a01ce60d8dd7b224f7c60845914def05ecc58',
    'uncles': [],
})
```

Eth.**getBlock**(*block_identifier=eth.default_block, full_transactions=False*)

> **Warning:** Deprecated: This method is deprecated in favor of *get_block()*

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', 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.**getBlockTransactionCount**(*block_identifier*)

> **Warning:** Deprecated: This method is deprecated in favor of *get_block_transaction_count()*

Eth.**getUncle**(*block_identifier*)

> **Note:** Method to get an Uncle from its hash is not available through RPC, a possible substitute is the method

`Eth.get_uncle_by_block`

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',
  'gasUsed': '0x0',
  'hash': '0xc78c35720d930f9ef34b4e6fb9d02ffec936f9b02a8f0fa858456e4afd4d5614',
  'logsBloom':
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
  '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.**getUncleByBlock**(*block_identifier*, *uncle_index*)

> **Warning:** Deprecated: This method is deprecated in favor of *get_uncle_by_block()*

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.**getUncleCount**(*block_identifier*)

> **Warning:** Deprecated: This method is deprecated in favor of *get_uncle_count()*

Eth.**get_transaction**(*transaction_hash*)

- Delegates to `eth_getTransactionByHash` RPC Method

Returns the transaction specified by `transaction_hash`. If the transaction has not yet been mined throws `web3.exceptions.TransactionNotFound`.

```
>>> web3.eth.get_transaction(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
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.**getTransaction**(*transaction_hash*)

Warning: Deprecated: This method is deprecated in favor of *get_transaction*

`Eth.getTransactionFromBlock(block_identifier, transaction_index)`

Note: This method is deprecated in EIP 1474.

`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'`, otherwise delegates to `eth_getTransactionByBlockHashAndIndex`. If the transaction has not yet been mined throws `web3.exceptions.TransactionNotFound`.

```
>>> web3.eth.get_transaction_by_block(46147, 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,
})
>>> 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.getTransactionByBlock(block_identifier, transaction_index)`

> **Warning:** Deprecated: This method is deprecated in favor of *get_transaction_by_block*

`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': '0x0000000000000000000000000000000000000000000000...0000',
    'status': 1,
    'to': '0x5DF9B87991262F6BA471F09758CDE1c0FC1De734',
    'transactionHash':
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'transactionIndex': 0,
})
```

`Eth.`**`waitForTransactionReceipt`**(*transaction_hash*, *timeout=120*, *poll_latency=0.1*)

> **Warning:** Deprecated: This method is deprecated in favor of *wait_for_transaction_receipt()*

`Eth.`**`get_transaction_receipt`**(*transaction_hash*)

- Delegates to `eth_getTransactionReceipt` RPC Method

Returns the transaction receipt specified by `transaction_hash`. If the transaction has not yet been mined 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(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
```

(continues on next page)

(continued from previous page)

```
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.`**`getTransactionReceipt`**(*transaction_hash*)

> **Warning:** Deprecated: This method is deprecated in favor of *get_transaction_receipt()*

`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.`**`getTransactionCount`**(*account*, *block_identifier=web3.eth.default_block*)

> **Warning:** Deprecated: This method is deprecated in favor of *get_transaction_count()*

`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 good reason to, 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 determines gas and fee)
>>> web3.eth.send_transaction({
  'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
  'from': web3.eth.coinbase,
  'value': 12345
})

# EIP 1559-style transaction
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
>>> web3.eth.send_transaction({
  'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
  'from': web3.eth.coinbase,
  'value': 12345,
  'gas': 21000,
  'maxFeePerGas': web3.toWei(250, 'gwei'),
  'maxPriorityFeePerGas': web3.toWei(2, 'gwei'),
})
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')

# Legacy transaction (less efficient)
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
>>> web3.eth.send_transaction({
  'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
  'from': web3.eth.coinbase,
  'value': 12345,
  'gas': 21000,
  'gasPrice': web3.toWei(50, 'gwei'),
})
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
```

`Eth.`**`sendTransaction`**`(`*transaction*`)`

> **Warning:** Deprecated: This method is deprecated in favor of `send_transaction()`

`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),
    gasPrice=w3.eth.gas_price,
    gas=100000,
    to='0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
    value=1,
    data=b'',
    )
)
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\x971F|\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.`**`signTransaction`**(*transaction*)

> **Warning:** Deprecated: This method is deprecated in favor of *`sign_transaction()`*

`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='0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
    value=12345,
    data=b'',
  ),
  private_key_for_senders_account,
)
>>> w3.eth.send_raw_transaction(signed_txn.rawTransaction)
HexBytes('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331')
```

`Eth.`**`sendRawTransaction`**(*raw_transaction*)

> **Warning:** Deprecated: This method is deprecated in favor of *`send_raw_transaction()`*

`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.

```
>>> tx = web3.eth.send_transaction({
        'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
        'from': web3.eth.coinbase,
        'value': 1000
    })
'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331'
>>> web3.eth.replace_transaction(
↪'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331', {
        'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
        'from': web3.eth.coinbase,
        'value': 2000
    })
```

`Eth.`**`replaceTransaction`**(*transaction_hash*, *new_transaction*)

> **Warning:** Deprecated: This method is deprecated in favor of *replace_transaction()*

`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.

```
>>> tx = web3.eth.send_transaction({
        'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
        'from': web3.eth.coinbase,
        'value': 1000
    })
'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331'
>>> web3.eth.modify_transaction
('0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331', value=2000)
```

`Eth.modifyTransaction`(*transaction_hash*, ***transaction_params*)

> **Warning:** Deprecated: This method is deprecated in favor of `modify_transaction()`

`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(
      '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
      text='some-text-tö-sign')

↪'0x1a8bbe6eab8c72a219385681efefe565afd3accee35f516f8edf5ae82208fbd45a58f9f9116d8d88ba40fcd2907
↪'

>>> web3.eth.sign(
      '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
      data=b'some-text-t\xc3\xb6-sign')

↪'0x1a8bbe6eab8c72a219385681efefe565afd3accee35f516f8edf5ae82208fbd45a58f9f9116d8d88ba40fcd2907
↪'

>>> web3.eth.sign(
      '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
      hexstr='0x736f6d652d746578742d74c3b62d7369676e')

↪'0x1a8bbe6eab8c72a219385681efefe565afd3accee35f516f8edf5ae82208fbd45a58f9f9116d8d88ba40fcd2907
↪'
```

`Eth.sign_typed_data`(*account*, *jsonMessage*)

- Delegates to `eth_signTypedData` RPC Method

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.signTypedData`(*account*, *jsonMessage*)

> **Warning:** Deprecated: This property is deprecated in favor of *sign_typed_data()*

`Eth.call(transaction, block_identifier=web3.eth.default_block, state_override=None)`

- 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:
>>> we3.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.

`Eth.estimate_gas(transaction, block_identifier=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.

```
>>> web3.eth.estimate_gas({'to': '0xd3CdA913deB6f67967B99D67aCDFa1712C293601',
↪'from':web3.eth.coinbase, 'value': 12345})
21000
```

`Eth.estimateGas(transaction, block_identifier=None)`

> **Warning:** Deprecated: This method is deprecated in favor of *estimate_gas()*

`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.`**`generateGasPrice`**(*transaction_params=None*)

> **Warning:** Deprecated: This method is deprecated in favor of *`generate_gas_price()`*

`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.

`Eth.`**`setGasPriceStrategy`**(*gas_price_strategy*)

> **Warning:** Deprecated: This method is deprecated in favor of *`set_gas_price_strategy()`*

2.19.3 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 "latest" for the last mined block or "pending", "earliest" for not yet mined transactions.
- `toBlock`: `integer/tag` - (optional, default: "latest") Integer block number, or "latest" for the last mined block or "pending", "earliest" for not yet mined transactions.
- `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.

See *Filtering* 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`

```
>>> filt = web3.eth.filter()
>>> web3.eth.get_filter_changes(filt.filter_id)
[
    {
        'address': '0xDc3A9Db694BCdd55EBaE4A89B22aC6D12b3F0c24',
        'blockHash':
↪'0xb72256286ca528e09022ffd408856a73ef90e7216ac560187c6e43b4c4efd2f0',
        'blockNumber': 2217196,
        'data':
↪'0x0000000000000000000000000000000000000000000000000000000000000001',
        'logIndex': 0,
        'topics': [
↪'0xe65b00b698ba37c614af350761c735c5f4a82b4ab365a1f1022d49d9dfc8e930',
        '0x000000000000000000000000754c50465885f1ed1fa1a55b95ee8ecf3f1f4324',
        '0x296c7fb6ccafa3e689950b947c2895b07357c95b066d5cdccd58c301f41359a3'],
        'transactionHash':
↪'0xfe1289fd3915794b99702202f65eea2e424b2f083a12749d29b4dd51f6dce40d',
        'transactionIndex': 1,
    },
    ...
]
```

Eth.**getFilterChanges**(*self*, *filter_id*)

> **Warning:** Deprecated: This property is deprecated in favor of *get_filter_changes()*

Eth.**get_filter_logs**(*self*, *filter_id*)

- Delegates to `eth_getFilterLogs` RPC Method.

Returns all entries for the given `filter_id`

```
>>> filt = web3.eth.filter()
>>> web3.eth.get_filter_logs(filt.filter_id)
[
    {
        'address': '0xDc3A9Db694BCdd55EBaE4A89B22aC6D12b3F0c24',
        'blockHash':
↪'0xb72256286ca528e09022ffd408856a73ef90e7216ac560187c6e43b4c4efd2f0',
        'blockNumber': 2217196,
        'data':
↪'0x0000000000000000000000000000000000000000000000000000000000000001',
```

(continues on next page)

(continued from previous page)

```
        'logIndex': 0,
        'topics': [
↪'0xe65b00b698ba37c614af350761c735c5f4a82b4ab365a1f1022d49d9dfc8e930',
        '0x000000000000000000000000754c50465885f1ed1fa1a55b95ee8ecf3f1f4324',
        '0x296c7fb6ccafa3e689950b947c2895b07357c95b066d5cdccd58c301f41359a3'],
        'transactionHash':
↪'0xfe1289fd3915794b99702202f65eea2e424b2f083a12749d29b4dd51f6dce40d',
        'transactionIndex': 1,
    },
    ...
]
```

`Eth.getFilterLogs(self, filter_id)`

> **Warning:** Deprecated: This method is deprecated in favor of `get_filter_logs()`

`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.

```
>>> filt = web3.eth.filter()
>>> web3.eth.uninstall_filter(filt.filter_id)
True
>>> web3.eth.uninstall_filter(filt.filter_id)
False  # already uninstalled.
```

`Eth.uninstallFilter(self, filter_id)`

> **Warning:** Deprecated: This method is deprecated in favor of `uninstall_filter()`

`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.getLogs(filter_params)`

> **Warning:** Deprecated: This property is deprecated in favor of `get_logs()`

`Eth.submit_hashrate(hashrate, nodeid)`

- Delegates to `eth_submitHashrate` RPC Method

```
>>> node_id = '59daa26581d0acd1fce254fb7e85952f4c09d0915afd33d3886cd914bc7d283c'
>>> web3.eth.submit_hashrate(5000, node_id)
True
```

`Eth.`**`submitHashrate`**`(`*hashrate*, *nodeid*`)`

> **Warning:** Deprecated: This property is deprecated in favor of *`submit_hashrate()`*

`Eth.`**`submit_work`**`(`*nonce*, *pow_hash*, *mix_digest*`)`

- Delegates to `eth_submitWork` RPC Method.

```
>>> web3.eth.submit_work(
        1,
        '0x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef',
        '0xD1FE5700000000000000000000000000D1FE5700000000000000000000000000',
    )
True
```

`Eth.`**`submitWork`**`(`*nonce*, *pow_hash*, *mix_digest*`)`

> **Warning:** Deprecated: This property is deprecated in favor of *`submit_work()`*

2.19.4 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 *Look up 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=...)
```

(continues on next page)

(continued from previous page)

```
# 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.

- `abi`
- `asm`
- `ast`
- `bytecode`
- `bytecode_runtime`
- `clone_bin`
- `dev_doc`
- `interface`
- `metadata`
- `opcodes`
- `src_map`
- `src_map_runtime`
- `user_doc`

See *Contracts* for more information about how to use 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`.

An example of an alternative Contract Factory is `ConciseContract`.

`Eth.`**`setContractFactory`**`(`*contractFactoryClass*`)`

> **Warning:** Deprecated: This method is deprecated in favor of `set_contract_factory()`

2.20 Eth 2.0 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")
```

2.20.1 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': {
```

(continues on next page)

(continued from previous page)

```
      'epoch': '5024',
      'root': '0x499ba555e8e8be639dd84be1be6d54409738facefc662f37d97065aa91a1a8d4'
    }
  }
}
```

Beacon.**get_validators**(*state_id='head'*)

```
>>> beacon.get_validators()
 {
   'data': [
     {
       'index': '110280',
       'balance': '32000000000',
       'status': 'pending_queued',
       'validator': {
         'pubkey':
↪'0x99d37d1f7dd15859995330f75c158346f86d298e2ffeedfbf1b38dcf3df89a7dbd1b34815f3bcd1b2a5588592a3
↪',
         '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':
↪'0x99d37d1f7dd15859995330f75c158346f86d298e2ffeedfbf1b38dcf3df89a7dbd1b34815f3bcd1b2a5588592a3
↪',
      'withdrawal_credentials':
↪'0x00f338cfdb0c22bb85beed9042bd19fff58ad6421c8a833f8bc902b7cca06f5f',
      'effective_balance': '32000000000',
      'slashed': False,
      'activation_eligibility_epoch': '5029',
      'activation_epoch': '18446744073709551615',
      'exit_epoch': '18446744073709551615',
      'withdrawable_epoch': '18446744073709551615'
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

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':
↪'0x87b549448d36e5e8b1783944b5511a05f34bb78ad3fcbf71a1adb346eed363d46e50d51ac53cd23bd03d0107d06
↪'
```

(continues on next page)

(continued from previous page)

```
      }
    }
  ]
}
```

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':
↪'0xa30d70b3e62ff776fe97f7f8b3472194af66849238a958880510e698ec3b8a470916680b1a82f9d4753c023153f
↪'
    }
  }
}
```

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':
↪'0x8e245a52a0a680fcfe789013e123880c321f237de10cad108dc55dd47290d7cfe50cdaa003c6f783405efdac48c
↪',
        'eth1_data': {
          'deposit_root':
↪'0x4e910ac762815c13e316e72506141f5b6b441d58af8e0a049cd3341c25728752',
          'deposit_count': '100596',
          'block_hash':
↪'0x89cb78044843805fb4dab8abd743fc96c2b8e955c58f9b7224d468d85ef57130'
        },
        'graffiti':
↪'0x74656b752f76302e31322e31342b34342d673863656562663600000000000000',
```

(continues on next page)

(continued from previous page)

```
        'proposer_slashings': [],
        'attester_slashings': [],
        'attestations': [
          {
            'aggregation_bits': '0x0080020004000000008208000102000905',
            'data': {
              'slot': '0',
              'index': '7',
              'beacon_block_root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87',
              'source': {
                'epoch': '0',
                'root':
↪'0x0000000000000000000000000000000000000000000000000000000000000000'
              },
              'target': {
                'epoch': '0',
                'root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87'
              }
            },
            'signature':
↪'0x967dd2946358db7e426ed19d4576bc75123520ef6a489ca50002222070ee4611f9cef394e5e3071236a93b825f1
↪'
          }
        ],
        'deposits': [],
        'voluntary_exits': []
      }
    },
    'signature':
↪'0xa30d70b3e62ff776fe97f7f8b3472194af66849238a958880510e698ec3b8a470916680b1a82f9d4753c023153f
↪'
  }
}
```

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': '0x0080020004000000008208000102000905',
      'data': {
        'slot': '0',
```

(continues on next page)

(continued from previous page)

```
        'index': '7',
        'beacon_block_root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87',
        'source': {
          'epoch': '0',
          'root':
↪'0x0000000000000000000000000000000000000000000000000000000000000000'
        },
        'target': {
          'epoch': '0',
          'root':
↪'0x6a89af5df908893eedbed10ba4c13fc13d5653ce57db637e3bfded73a987bb87'
        }
      },
      'signature':
↪'0x967dd2946358db7e426ed19d4576bc75123520ef6a489ca50002222070ee4611f9cef394e5e3071236a93b825f1
↪'
    },
    ...
  ]
}
```

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'
```

(continues on next page)

(continued from previous page)

```
    }
  ]
}
```

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'
```

(continues on next page)

(continued from previous page)

```
    },
    'block_roots': [
↪'0xcf8e0d4e9587369b2301d0790347320302cc0943d5a1884560367e8208d920f2'],
    '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()
{
```

(continues on next page)

(continued from previous page)

```
  'data': {
    'peer_id': '16Uiu2HAmLZ1CYVFKpa3wwn4cknZqosum8HX3GHDhUpEULQc9ixE',
    'enr': 'enr:-KG4QCIp6eCZ6hG_
↪fd93qsw12qmbfsl2rUTfQvwVP4FOTlWeNXYo0Gg9y3WVYIdF6FQC6R0E8CbK0Ywq_
↪6TKMx1BpGlAhGV0aDKQOwiHlQAAIAn___________4JpZIJ2NIJpcIR_
↪AAABiXNlY3AyNTZrMaEDdVT4g1gw86BfbrtLCq2fRBlG0AnMxsXtAQgA327S5FeDdGNwgiMog3VkcIIjKA
↪',
    'p2p_addresses': ['/ip4/127.0.0.1/tcp/9000/p2p/
↪16Uiu2HAmLZ1CYVFKpa3wwn4cknZqosum8HX3GHDhUpEULQc9ixE'],
    'discovery_addresses': ['/ip4/127.0.0.1/udp/9000/p2p/
↪16Uiu2HAmLZ1CYVFKpa3wwn4cknZqosum8HX3GHDhUpEULQc9ixE'],
    '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'
  }
}
```

2.21 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.auto import w3
>>> w3.pm
...
AttributeError: The Package Management feature is disabled by default ...
```

In order to access these features, you can turn it on with...

```
>>> web3.enable_unstable_package_management_api()
>>> w3.pm
<web3.pm.PM at 0x....>
```

2.21.1 Methods

The following methods are available on the `web3.pm` namespace.

class `web3.pm.`**`PM`**`(`*web3: 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*`)` → ethpm.package.Package

Returns a Package instance built with the given manifest.

- **Parameters:**

- `manifest`: A dict representing a valid manifest

get_package_from_uri (*manifest_uri: URI*) → ethpm.package.Package

Returns a Package 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: Optional[pathlib.Path] = None*) → ethpm.package.Package

Returns a Package 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: Union[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*) → web3.pm.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*) → web3.pm.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*) → ethpm.package.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.main.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 definied 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.main.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 containg 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*) → web3.pm.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.

2.21.2 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.

2.22 Net API

The `web3.net` object exposes methods to interact with the RPC APIs under the `net_` namespace.

2.22.1 Properties

The following properties are available on the `web3.net` namespace.

property `web3.net.`**`chainId`**

> **Warning:** Deprecated: This property is deprecated as of EIP 1474.

property `web3.net.`**`listening`**

- Delegates to `net_listening` RPC method

Returns true if client is actively listening for network connections.

```
>>> web3.net.listening
True
```

property `web3.net.`**`peer_count`**

- Delegates to `net_peerCount` RPC method

Returns number of peers currently connected to the client.

```
>>> web3.net.peer_count
1
```

property `web3.net.`**`peerCount`**

> **Warning:** Deprecated: This property is deprecated in favor of `peer_count`

property `web3.net.`**`version`**

- Delegates to `net_version` RPC Method

Returns the current network id.

```
>>> web3.net.version
'8996'
```

2.23 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.

2.23.1 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.makeDAG(number)`

> **Warning:** Deprecated: This method is deprecated in favor of `make_dag()`

`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.setExtra(extra)`

> **Warning:** Deprecated: This method is deprecated in favor of `set_extra()`

`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.setGasPrice(gas_price)`

> **Warning:** Deprecated: This method is deprecated in favor of `set_gas_price()`

`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.startAutoDag()`

> **Warning:** Deprecated: This method is deprecated in favor of `start_auto_dag()`

`GethMiner.stop_auto_dag()`

- Delegates to `miner_stopAutoDag` RPC Method

Disable automatic DAG generation.

```
>>> web3.geth.miner.stop_auto_dag()
```

`GethMiner.stopAutoDag()`

> **Warning:** Deprecated: This method is deprecated in favor of `stop_auto_dag()`

2.24 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.

2.24.1 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/piper/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://
↪e54eebad24dce1f6d246bea455ffa756d97801582420b9ed681a2ea84bf376d0bd87ae8dd6dc06cdb862a2ca89ecab
↪',
    'id':
↪'e54eebad24dce1f6d246bea455ffa756d97801582420b9ed681a2ea84bf376d0bd87ae8dd6dc06cdb862a2ca89eca
↪',
    '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.nodeInfo()`

> **Warning:** Deprecated: This method is deprecated in favor of *node_info()*

`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':
↪'146e8e3e2460f1e18939a5da37c4a79f149c8b9837240d49c7d94c122f30064e07e4a42ae2c2992d0f8e7e6f68a30
↪',
        '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,
            },
        }
    },
    {
        'caps': ['eth/62', 'eth/63'],
        'id':
↪'76cb6cd3354be081923a90dfd4cda40aa78b307cc3cf4d5733dc32cc171d00f7c08356e9eb2ea47eab5aad7a15a34
↪',
        '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://
↪e54eebad24dce1f6d246bea455ffa756d97801582420b9ed681a2ea84bf376d0bd87ae8dd6dc06cdb862a2ca89ecab
↪71.255.237:30303')
True
```

web3.geth.admin.**addPeer**(*node_url*)

Warning: Deprecated: This method is deprecated in favor of *add_peer()*

web3.geth.admin.**setSolc**(*solc_path*)

Warning: This method has been removed from Geth

web3.geth.admin.**start_rpc**(*host='localhost'*, *port=8545*, *cors="*, *apis='eth,net,web3'*)

- Delegates to `admin_startRPC` 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_rpc()
True
```

web3.geth.admin.**startRPC**(*host='localhost'*, *port=8545*, *cors="*, *apis='eth,net,web3'*)

Warning: Deprecated: This method is deprecated in favor of *start_rpc()*

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.**startWS**(*host='localhost'*, *port=8546*, *cors="*, *apis='eth,net,web3'*)

Warning: Deprecated: This method is deprecated in favor of *start_ws()*

web3.geth.admin.**stop_rpc**()

- Delegates to `admin_stopRPC` RPC Method

Stops the HTTP based JSON RPC server.

```
>>> web3.geth.admin.stop_rpc()
True
```

web3.geth.admin.**stopRPC**()

> **Warning:** Deprecated: This method is deprecated in favor of `stop_rpc()`

`web3.geth.admin.stop_ws()`

- Delegates to `admin_stopWS` RPC Method

Stops the Websocket based JSON RPC server.

```
>>> web3.geth.admin.stop_ws()
True
```

`web3.geth.admin.stopWS()`

> **Warning:** Deprecated: This method is deprecated in favor of `stop_ws()`

2.24.2 GethPersonal API

The following methods are available on the `web3.geth.personal` namespace.

`web3.geth.personal.list_accounts()`

- Delegates to `personal_listAccounts` RPC Method

Returns the list of known accounts.

```
>>> web3.geth.personal.list_accounts()
['0xd3CdA913deB6f67967B99D67aCDFa1712C293601']
```

`web3.geth.personal.listAccounts()`

> **Warning:** Deprecated: This method is deprecated in favor of `list_accounts()`

`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.import_raw_key`(*self*, *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.importRawKey()`

Warning: Deprecated: This method is deprecated in favor of *import_raw_key()*

`web3.geth.personal.new_account(self, 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')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

`web3.geth.personal.newAccount()`

Warning: Deprecated: This method is deprecated in favor of *new_account()*

`web3.geth.personal.lock_account(self, account)`

- Delegates to `personal_lockAccount` RPC Method

Locks the given `account`.

```
>>> web3.geth.personal.lock_account('0xd3CdA913deB6f67967B99D67aCDFa1712C293601')
```

`web3.geth.personal.lockAccount()`

Warning: Deprecated: This method is deprecated in favor of *lock_account()*

`web3.geth.personal.unlock_account(self, 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 300 seconds (which is current default by Geth v1.9.5); if `duration` is set to `0`, the account will remain unlocked indefinitely. Returns boolean as to whether the account was successfully unlocked.

```
>>> web3.geth.personal.unlock_account('0xd3CdA913deB6f67967B99D67aCDFa1712C293601
↪', 'wrong-passphrase')
False
>>> web3.geth.personal.unlock_account('0xd3CdA913deB6f67967B99D67aCDFa1712C293601
↪', 'the-passphrase')
True
```

`web3.geth.personal.unlockAccount()`

> **Warning:** Deprecated: This method is deprecated in favor of *unlock_account()*

`web3.geth.personal.send_transaction(self, transaction, passphrase)`

- Delegates to `personal_sendTransaction` RPC Method

Sends the transaction.

`web3.geth.personal.sendTransaction()`

> **Warning:** Deprecated: This method is deprecated in favor of *send_transaction()*

2.24.3 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 nor supported in Parity.

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 inclusing in the next block(s) as will 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: 1000000000000000000␣
↪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 inclusing in the next block(s) as will 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':
↪"0xb61d27f600000000000000000000000024d407e5a0b506e1cb2fae163100b5de01f5193c00000000000000000000
↪",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
        'nonce': "0x22",
        'to': "0x7320785200f74861B69C49e4ab32399a71b34f1a",
        'transactionIndex': None,
        'value': "0x0"
      }]
    }
  },
  'queued': {
    '0x976A3Fc5d6f7d259EBfb4cc2Ae75115475E9867C': {
      3: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x976A3Fc5d6f7d259EBfb4cc2Ae75115475E9867C",
        'gas': "0x15f90",
        'gasPrice': None,
```

(continues on next page)

(continued from previous page)

```
        '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",
        'to': "0x6368f3f8c2B42435D6C136757382E4A59436a681",
        'transactionIndex': None,
        'value': "0xf9a951af55470000"
      }, {
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x9B11bF0459b0c4b2f87f8CEBca4cfc26f294B63A",
        'gas': "0x15f90",
        'gasPrice': None,
        'hash':
↪"0x60803251d43f072904dc3a2d6a084701cd35b4985790baaf8a8f76696041b272",
        'input': "0x",
        'maxFeePerGas': '0x77359400',
        'maxPriorityFeePerGas': '0x3b9aca00',
```

(continues on next page)

(continued from previous page)

```
            'nonce': "0x6",
            'to': "0x8DB7b4e0ECB095FBD01Dffa62010801296a9ac78",
            'transactionIndex': None,
            'value': "0xebe866f5f0a06000"
          }],
        }
    }
}
```

2.25 Parity API

The `web3.parity` object exposes modules that enable you to interact with the JSON-RPC endpoints supported by Parity that are not defined in the standard set of Ethereum JSONRPC endpoints according to EIP 1474.

2.25.1 ParityPersonal

The following methods are available on the `web3.parity.personal` namespace.

`web3.parity.personal.`**`list_accounts`**`()`

- Delegates to `personal_listAccounts` RPC Method

Returns the list of known accounts.

```
>>> web3.parity.personal.list_accounts()
['0xd3CdA913deB6f67967B99D67aCDFa1712C293601']
```

`web3.parity.personal.`**`listAccounts`**`()`

> **Warning:** Deprecated: This method is deprecated in favor of `list_accounts()`

`web3.parity.personal.`**`import_raw_key`**`(`*self*, *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.parity.personal.import_raw_key(some_private_key, 'the-passphrase')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

`web3.parity.personal.`**`importRawKey`**`(`*self*, *private_key*, *passphrase*`)`

> **Warning:** Deprecated: This method is deprecated in favor of `import_raw_key()`

`web3.parity.personal.`**`new_account`**`(`*self*, *password*`)`

- 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.parity.personal.new_account('the-passphrase')
'0xd3CdA913deB6f67967B99D67aCDFa1712C293601'
```

`web3.parity.personal.`**`newAccount`**(*self*, *password*)

> **Warning:** Deprecated: This method is deprecated in favor of *`new_account()`*

`web3.parity.personal.`**`unlock_account`**(*self*, *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 indefinitely. Returns boolean as to whether the account was successfully unlocked.

```
# Invalid call to personal_unlockAccount on Parity currently returns True, due to␣
↪Parity bug
>>> web3.parity.personal.unlock_account(
↪'0xd3CdA913deB6f67967B99D67aCDFa1712C293601', 'wrong-passphrase')
True
>>> web3.parity.personal.unlock_account(
↪'0xd3CdA913deB6f67967B99D67aCDFa1712C293601', 'the-passphrase')
True
```

`web3.parity.personal.`**`unlockAccount`**(*self*, *account*, *passphrase*, *duration=None*)

> **Warning:** Deprecated: This method is deprecated in favor of *`unlock_account()`*

`web3.parity.personal.`**`send_transaction`**(*self*, *transaction*, *passphrase*)

- Delegates to `personal_sendTransaction` RPC Method

Sends the transaction.

`web3.parity.personal.`**`sendTransaction`**(*self*, *account*, *passphrase*, *duration=None*)

> **Warning:** Deprecated: This method is deprecated in favor of *`send_transaction()`*

`web3.parity.personal.`**`sign_typed_data`**(*self*, *jsonMessage*, *account*, *passphrase*)

- Delegates to `personal_signTypedData` RPC Method

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 passphrase of the given `account`

`web3.parity.personal.`**`signTypedData`**(*self*, *jsonMessage*, *account*, *passphrase*)

Warning: Deprecated: This method is deprecated in favor of *sign_typed_data()*

2.26 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).

2.26.1 Retrieving gas price

To retrieve the gas price using the selected strategy simply call *generate_gas_price()*

```
>>> web3.eth.generate_gas_price()
20000000000
```

2.26.2 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.toWei(1, 'ether'):
        return Web3.toWei(20, 'gwei')
    else:
        return Web3.toWei(5, 'gwei')
```

2.26.3 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)
```

2.26.4 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 maxiumum 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.

```
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)
```

2.27 ENS API

Ethereum Name Service has a friendly overview.

Continue below for the detailed specs on each method and class in the ens module.

2.27.1 ens.main module

class ens.main.**ENS**(*provider: BaseProvider = <object object>*, *addr: ChecksumAddress = 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"`

static namehash(*name: str*) → hexbytes.main.HexBytes

Generate the namehash. This is also known as the `node` in ENS contracts.

In normal operation, generating the namehash is handled behind the scenes. For advanced usage, it is a helpful utility.

This normalizes the name with nameprep before hashing.

Parameters **name** (*str*) – ENS name to hash

Returns the namehash

Return type bytes

Raises ***InvalidName*** – if `name` has invalid syntax

static nameprep(*name: str*) → str

Clean the fully qualified name, as defined in ENS EIP-137

This does *not* enforce whether `name` is a label or fully qualified domain.

Parameters **name** (*str*) – the dot-separated ENS name

Raises ***InvalidName*** – if `name` has invalid syntax

static is_valid_name(*name: str*) → bool

Validate whether the fully qualified name is valid, as defined in ENS EIP-137

Parameters **name** (*str*) – the dot-separated ENS name

Returns True if `name` is set, and *nameprep()* will not raise InvalidName

classmethod fromWeb3(*web3: Web3*, *addr: ChecksumAddress = None*) → *ENS*

Generate an ENS instance with web3

Parameters

- **web3** (`web3.Web3`) – to infer connection information
- **addr** (*hex-string*) – the address of the ENS registry on-chain. If not provided, ENS.py will default to the mainnet ENS registry address.

address (*name: str*) → Optional[ChecksumAddress]
Look up the Ethereum address that *name* currently points to.

Parameters **name** (*str*) – an ENS name to look up

Raises ***InvalidName*** – if *name* has invalid syntax

name (*address: ChecksumAddress*) → Optional[str]
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_address (*name: str, address: Union[Address, ChecksumAddress, HexAddress] = <object object>, transact: TxParams = {}*) → hexbytes.main.HexBytes
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.
- **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*

setup_name (*name: str, address: ChecksumAddress = None, transact: TxParams = {}*) → hexbytes.main.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*) – 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 = {}*) → ChecksumAddress

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

2.27.2 ens.exceptions module

exception ens.exceptions.**AddressMismatch**

Bases: `ValueError`

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: `idna.core.IDNAError`

This exception is raised if the provided name does not meet the syntax standards specified in EIP 137 name syntax.

For example: names may not start with a dot, or include a space.

exception ens.exceptions.**UnauthorizedError**

Bases: `Exception`

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: `Exception`

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.**BidTooLow**
Bases: ValueError

Raised if you bid less than the minimum amount

exception ens.exceptions.**InvalidBidHash**
Bases: ValueError

Raised if you supply incorrect data to generate the bid hash.

exception ens.exceptions.**InvalidLabel**
Bases: ValueError

Raised if you supply an invalid label

exception ens.exceptions.**OversizeTransaction**
Bases: ValueError

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: ValueError

Raised if you send less wei with your bid than you declared as your intent to bid.

2.28 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.

2.28.1 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.

2.28.2 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
```

2.28.3 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.

2.28.4 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()
```

2.28.5 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.

2.28.6 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

Unit tests are meant to test the logic of smaller chunks (or units) of the codebase without having to be wired up to a client. Most of the time this means testing selected methods on their own. They are meant to test the logic of your code and make sure that you get expected outputs out of selected inputs.

Our unit tests live under appropriately named child directories within the `/tests` directory. The core of the unit tests live under `/tests/core`. Do your best to follow the existing structure when choosing where to add your unit test.

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 and run our tests.

Though the setup lives in `/tests/integration`, our integration module tests are written across different files within `/web3/_utils/module_testing`. The tests are written there but run configurations exist across the different files within `/tests/integration/`. The parent `/integration` directory houses some common configuration shared across all client tests, whereas the `/go_ethereum` and `/parity` directories house common code to be shared among each respective client tests.

- `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 as 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` (e.g. `test_goethereum_http.py`) files 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.

2.28.7 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
```

2.28.8 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>/`.

2.28.9 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.

2.28.10 Generating New Fixtures

Our integration tests make use of Geth and Parity/OpenEthereum 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 v1.10.4
   ```

2. Specify the Geth binary and run the fixture creation script (from within the web3.py directory):

   ```
   $ GETH_BINARY=~/.py-geth/geth-v1.10.4/bin/geth python ./tests/integration/
   ↪generate_fixtures/go_ethereum.py ./tests/integration/geth-1.10.4-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` file 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.

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 master 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.

Parity/OpenEthereum Fixtures

1. The most reliable way to get a specific Parity/OE binary is to download the source code via GitHub releases.
2. Build the binary from source. (This is will take a few minutes.)
3. Specify the path to this binary in the `get_parity_binary` function of the `/tests/integration/generate_fixtures/parity.py` file.
4. Run the fixture generation script:

```
$ python /tests/integration/generate_fixtures/parity.py /tests/integration/parity-X.Y.
↪Z-fixture
```

5. The output of this script is your fixture, a zip file. Store the fixture in the `/tests/integration/` directory and update the `/tests/integration/parity/conftest.py` file to point the new fixture.
6. By this point, you may have noticed that Parity fixture generation relies on a Geth network to sync from. In the output of the generation script are the hashes of the various contracts that it mined. Update the corresponding values in the `/parity/conftest.py` file with those hashes.
7. Run the tests.

2.28.11 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 master && git pull

$ make package

# in another shell, navigate to the virtualenv mentioned in output of ^
```

(continues on next page)

(continued from previous page)

```
# 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.auto import w3
>>> w3.isConnected()
>>> ...
```

Verify The Latest Documentation

To preview the documentation that will get published:

```
$ make docs
```

Preview The Release Notes

```
$ towncrier --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 master 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"`.

2.29 Code of Conduct

2.29.1 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.

2.29.2 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

2.29.3 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.

2.29.4 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.

2.29.5 Enforcement

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

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

2.29.6 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

e

ens, 215
ens.exceptions, 217
ens.main, 215

w

web3, 158
web3.contract, 103
web3.eth, 163
web3.gas_strategies.rpc, 214
web3.gas_strategies.time_based, 214
web3.geth, 201
web3.geth.admin, 202
web3.geth.miner, 200
web3.geth.personal, 205
web3.geth.txpool, 207
web3.net, 199
web3.parity, 211
web3.parity.personal, 211
web3.utils.filters, 98

INDEX

Symbols

__init__() (*web3.pm.ERC1319Registry method*), 197
__repr__() (*ethpm.Package method*), 133
_generate_release_id() (*web3.pm.ERC1319Registry method*), 198
_get_all_package_ids() (*web3.pm.ERC1319Registry method*), 198
_get_all_release_ids() (*web3.pm.ERC1319Registry method*), 198
_get_package_name() (*web3.pm.ERC1319Registry method*), 197
_get_release_data() (*web3.pm.ERC1319Registry method*), 198
_get_release_id() (*web3.pm.ERC1319Registry method*), 198
_num_package_ids() (*web3.pm.ERC1319Registry method*), 198
_num_release_ids() (*web3.pm.ERC1319Registry method*), 198
_release() (*web3.pm.ERC1319Registry method*), 197

A

abi (*web3.contract.Contract attribute*), 105
accounts (*web3.eth.Eth attribute*), 165
add() (*Web3.middleware_onion method*), 125
add_peer() (*in module web3.geth.admin*), 203
addPeer() (*in module web3.geth.admin*), 203
address (*web3.contract.Contract attribute*), 105
address() (*ens.main.ENS method*), 216
AddressMismatch, 217
all_functions() (*web3.contract.Contract class method*), 108
api (*web3.Web3 attribute*), 158
attrdict_middleware() (*web3.middleware method*), 123

B

BidTooLow, 217
block_number (*web3.eth.Eth attribute*), 165
BlockFilter (*class in web3.utils.filters*), 99
blockNumber (*web3.eth.Eth attribute*), 165
build_dependencies (*ethpm.Package attribute*), 134
build_dependency()
  built-in function, 149
build_filter() (*web3.contract.Contract.events.your_event_name class method*), 107
buildTransaction() (*web3.contract.Contract.fallback method*), 114
buildTransaction() (*web3.contract.ContractFunction method*), 113
built-in function
  build_dependency(), 149
  deployment(), 147
  deployment_type(), 147
bytecode (*web3.contract.Contract attribute*), 105
bytecode_runtime (*web3.contract.Contract attribute*), 105

C

call() (*web3.contract.Contract.fallback method*), 114
call() (*web3.contract.ContractFunction method*), 112
call() (*web3.eth.Eth method*), 180
can_resolve_uri() (*BaseURIBackend method*), 136
can_translate_uri() (*BaseURIBackend method*), 136
chain_id (*web3.eth.Eth attribute*), 166
chainId (*web3.eth.Eth attribute*), 166
chainId() (*in module web3.net*), 199
clear() (*Web3.middleware_onion method*), 126
clientVersion (*web3.Web3 attribute*), 158
coinbase (*web3.eth.Eth attribute*), 164
ConciseContract (*class in web3.contract*), 104
construct_latest_block_based_cache_middleware() (*web3.middleware method*), 127
construct_sign_and_send_raw_middleware() (*web3.middleware method*), 129
construct_simple_cache_middleware() (*web3.middleware method*), 127

construct_time_based_cache_middleware() (*web3.middleware method*), 127
construct_time_based_gas_price_strategy() (*in module web3.gas_strategies.time_based*), 214
constructor() (*web3.contract.Contract class method*), 106
content() (*web3.geth.txpool.TxPool method*), 208
Contract (*class in web3.contract*), 104
contract() (*web3.eth.Eth method*), 184
contract_types() (*ethpm.Package property*), 133
ContractCaller (*class in web3.contract*), 119
ContractEvents (*class in web3.contract*), 115
ContractFunction (*class in web3.contract*), 112
create_content_addressed_github_uri(), 137
createFilter() (*web3.contract.Contract.events.your_event_name class method*), 107

D

datadir() (*in module web3.geth.admin*), 202
decode_function_input() (*web3.contract.Contract class method*), 118
default_account (*web3.eth.Eth attribute*), 164
default_block (*web3.eth.Eth attribute*), 164
defaultAccount (*web3.eth.Eth attribute*), 164
defaultBlock (*web3.eth.Eth attribute*), 164
deploy() (*web3.contract.Contract class method*), 107
deploy_and_set_registry() (*web3.pm.PM method*), 196
deployment()
    built-in function, 147
deployment_type()
    built-in function, 147
deployments (*ethpm.Package attribute*), 134

E

enable_strict_bytes_type_checking() (*web3.w3 method*), 162
encodeABI() (*web3.contract.Contract class method*), 108
ens
    module, 215
ENS (*class in ens.main*), 215
ens.exceptions
    module, 217
ens.main
    module, 215
ERC1319Registry (*class in web3.pm*), 197
estimate_gas() (*web3.eth.Eth method*), 180
estimateGas() (*web3.contract.Contract.fallback method*), 114
estimateGas() (*web3.contract.ContractFunction method*), 113
estimateGas() (*web3.eth.Eth method*), 180
Eth (*class in web3.eth*), 163
eth (*web3.Web3 attribute*), 163
EthereumTesterProvider (*class in web3.providers.eth_tester*), 61
events (*web3.contract.Contract attribute*), 105

F

fetch_uri_contents() (*BaseURIBackend method*), 136
Filter (*class in web3.utils.filters*), 99
filter() (*web3.eth.Eth method*), 181
filter_id (*web3.utils.filters.Filter attribute*), 99
find_functions_by_args() (*web3.contract.Contract class method*), 109
find_functions_by_name() (*web3.contract.Contract class method*), 108
format_entry() (*web3.utils.filters.Filter method*), 99
fromWeb3() (*ens.main.ENS class method*), 215
fromWei() (*web3.Web3 method*), 160
functions (*web3.contract.Contract attribute*), 105

G

gas_price (*web3.eth.Eth attribute*), 165
gas_price_strategy_middleware() (*web3.middleware method*), 124
gasPrice (*web3.eth.Eth attribute*), 165
generate_gas_price() (*web3.eth.Eth method*), 180
generateGasPrice() (*web3.eth.Eth method*), 180
get_all_entries() (*web3.utils.filters.Filter method*), 99
get_all_package_names() (*web3.pm.PM method*), 196
get_all_package_releases() (*web3.pm.PM method*), 196
get_attestations() (*Beacon method*), 191
get_attester_slashings() (*Beacon method*), 191
get_balance() (*web3.eth.Eth method*), 166
get_beacon_heads() (*Beacon method*), 193
get_beacon_state() (*Beacon method*), 192
get_block() (*Beacon method*), 189
get_block() (*web3.eth.Eth method*), 169
get_block_attestations() (*Beacon method*), 190
get_block_header() (*Beacon method*), 189
get_block_headers() (*Beacon method*), 188
get_block_number() (*web3.eth.Eth method*), 166
get_block_root() (*Beacon method*), 190

get_block_transaction_count() (web3.eth.Eth method), 170
get_code() (web3.eth.Eth method), 169
get_deposit_contract() (Beacon method), 192
get_epoch_committees() (Beacon method), 188
get_filter_changes() (web3.eth.Eth method), 182
get_filter_logs() (web3.eth.Eth method), 182
get_finality_checkpoint() (Beacon method), 186
get_fork_data() (Beacon method), 186
get_fork_schedule() (Beacon method), 191
get_function_by_args() (web3.contract.Contract class method), 109
get_function_by_name() (web3.contract.Contract class method), 108
get_function_by_selector() (web3.contract.Contract class method), 108
get_function_by_signature() (web3.contract.Contract class method), 108
get_genesis() (Beacon method), 186
get_hash_root() (Beacon method), 186
get_health() (Beacon method), 194
get_local_package() (web3.pm.PM method), 196
get_logs() (web3.eth.Eth method), 183
get_new_entries() (web3.utils.filters.Filter method), 99
get_node_identity() (Beacon method), 193
get_package() (web3.pm.PM method), 197
get_package_count() (web3.pm.PM method), 196
get_package_from_manifest() (web3.pm.PM method), 195
get_package_from_uri() (web3.pm.PM method), 196
get_peer() (Beacon method), 194
get_peers() (Beacon method), 194
get_proof() (web3.eth.Eth method), 167
get_proposer_slashings() (Beacon method), 191
get_release_count() (web3.pm.PM method), 196
get_release_data() (web3.pm.PM method), 197
get_release_id() (web3.pm.PM method), 196
get_release_id_data() (web3.pm.PM method), 197
get_spec() (Beacon method), 192
get_storage_at() (web3.eth.Eth method), 166
get_syncing() (Beacon method), 195
get_transaction() (web3.eth.Eth method), 172
get_transaction_by_block() (web3.eth.Eth method), 173
get_transaction_count() (web3.eth.Eth method), 175
get_transaction_receipt() (web3.eth.Eth method), 174
get_uncle_by_block() (web3.eth.Eth method), 171
get_uncle_count() (web3.eth.Eth method), 172
get_validator() (Beacon method), 187
get_validator_balances() (Beacon method), 188
get_validators() (Beacon method), 187
get_version() (Beacon method), 194
get_voluntary_exits() (Beacon method), 191
getBalance() (web3.eth.Eth method), 166
getBlock() (web3.eth.Eth method), 170
getBlockTransactionCount() (web3.eth.Eth method), 170
getCode() (web3.eth.Eth method), 169
getFilterChanges() (web3.eth.Eth method), 182
getFilterLogs() (web3.eth.Eth method), 183
geth (web3.Web3 attribute), 163
getLogs() (web3.eth.Eth method), 183
getProof() (web3.eth.Eth method), 169
getStorageAt() (web3.eth.Eth method), 167
getTransaction() (web3.eth.Eth method), 172
getTransactionByBlock() (web3.eth.Eth method), 173
getTransactionCount() (web3.eth.Eth method), 175
getTransactionFromBlock() (web3.eth.Eth method), 173
getTransactionReceipt() (web3.eth.Eth method), 175
getUncle() (web3.eth.Eth method), 170
getUncleByBlock() (web3.eth.Eth method), 171
getUncleCount() (web3.eth.Eth method), 172

H

hashrate (web3.eth.Eth attribute), 164
http_retry_request_middleware() (web3.middleware method), 124
HTTPProvider (web3.Web3 attribute), 158

I

ImplicitContract (class in web3.contract), 105
import_raw_key() (in module web3.geth.personal), 205
import_raw_key() (in module web3.parity.personal), 211
importRawKey() (in module web3.geth.personal), 206
importRawKey() (in module web3.parity.personal), 211
inject() (Web3.middleware_onion method), 125

inspect() (web3.geth.txpool.TxPool method), 207
InvalidBidHash, 218
InvalidLabel, 218
InvalidName, 217
IPCProvider (web3.Web3 attribute), 158
is_encodable() (web3.w3 method), 162
is_valid_entry() (web3.utils.filters.Filter method), 99
is_valid_name() (ens.main.ENS static method), 215
isAddress() (web3.Web3 method), 160
isChecksumAddress() (web3.Web3 method), 160
isConnected() (BaseProvider method), 131

K

keccak() (web3.Web3 class method), 161

L

link_bytecode() (LinkableContract class method), 136
linked_references (LinkableContract attribute), 136
list_accounts() (in module web3.geth.personal), 205
list_accounts() (in module web3.parity.personal), 211
list_wallets() (in module web3.geth.personal), 205
listAccounts() (in module web3.geth.personal), 205
listAccounts() (in module web3.parity.personal), 211
listening() (in module web3.net), 199
lock_account() (in module web3.geth.personal), 206
lockAccount() (in module web3.geth.personal), 206
LogFilter (class in web3.utils.filters), 100

M

make_dag() (web3.geth.miner.GethMiner method), 200
make_request() (BaseProvider method), 131
make_stalecheck_middleware() (web3.middleware method), 127
makeDAG() (web3.geth.miner.GethMiner method), 200
manifest (Package attribute), 134
manifest_version() (ethpm.Package property), 133
middlewares (BaseProvider attribute), 131
miner (web3.Web3 attribute), 163
mining (web3.eth.Eth attribute), 164
modify_transaction() (web3.eth.Eth method), 178
modifyTransaction() (web3.eth.Eth method), 179
module
ens, 215
ens.exceptions, 217
ens.main, 215
web3, 158
web3.contract, 103
web3.eth, 163
web3.gas_strategies.rpc, 214
web3.gas_strategies.time_based, 214
web3.geth, 201
web3.geth.admin, 202
web3.geth.miner, 200
web3.geth.personal, 205
web3.geth.txpool, 207
web3.net, 199
web3.parity, 211
web3.parity.personal, 211
web3.utils.filters, 98
myEvent() (web3.contract.ContractEvents method), 116

N

name() (ens.main.ENS method), 216
name() (ethpm.Package property), 133
name_to_address_middleware() (web3.middleware method), 123
namehash() (ens.main.ENS static method), 215
nameprep() (ens.main.ENS static method), 215
needs_bytecode_linking (LinkableContract attribute), 136
new_account() (in module web3.geth.personal), 206
new_account() (in module web3.parity.personal), 211
newAccount() (in module web3.geth.personal), 206
newAccount() (in module web3.parity.personal), 212
node_info() (in module web3.geth.admin), 202
nodeInfo() (in module web3.geth.admin), 202

O

OversizeTransaction, 218
owner() (ens.main.ENS method), 216

P

Package (class in ethpm), 133
parity (web3.Web3 attribute), 163
peer_count() (in module web3.net), 199
peerCount() (in module web3.net), 199
peers() (in module web3.geth.admin), 202
pin_assets() (BaseIPFSBackend method), 137
PM (class in web3.pm), 195
pm (web3.Web3 attribute), 163
protocol_version (web3.eth.Eth attribute), 165
protocolVersion (web3.eth.Eth attribute), 165
pythonic_middleware() (web3.middleware method), 124

R

release_package() (*web3.pm.PM method*), 196
remove() (*Web3.middleware_onion method*), 126
replace() (*Web3.middleware_onion method*), 126
replace_transaction() (*web3.eth.Eth method*), 177
replaceTransaction() (*web3.eth.Eth method*), 178
rpc_gas_price_strategy() (*in module web3.gas_strategies.rpc*), 214

S

send_raw_transaction() (*web3.eth.Eth method*), 177
send_transaction() (*in module web3.geth.personal*), 207
send_transaction() (*in module web3.parity.personal*), 212
send_transaction() (*web3.eth.Eth method*), 175
sendRawTransaction() (*web3.eth.Eth method*), 177
sendTransaction() (*in module web3.geth.personal*), 207
sendTransaction() (*in module web3.parity.personal*), 212
sendTransaction() (*web3.eth.Eth method*), 176
set_contract_factory() (*web3.eth.Eth method*), 185
set_data_filters() (*web3.utils.filters.LogFilter method*), 100
set_extra() (*web3.geth.miner.GethMiner method*), 200
set_gas_price() (*web3.geth.miner.GethMiner method*), 200
set_gas_price_strategy() (*web3.eth.Eth method*), 181
set_registry() (*web3.pm.PM method*), 196
setContractFactory() (*web3.eth.Eth method*), 185
setExtra() (*web3.geth.miner.GethMiner method*), 200
setGasPrice() (*web3.geth.miner.GethMiner method*), 200
setGasPriceStrategy() (*web3.eth.Eth method*), 181
setSolc() (*in module web3.geth.admin*), 204
setup_address() (*ens.main.ENS method*), 216
setup_name() (*ens.main.ENS method*), 216
setup_owner() (*ens.main.ENS method*), 217
sha3() (*web3.Web3 class method*), 161
sign() (*web3.eth.Eth method*), 179
sign_transaction() (*web3.eth.Eth method*), 176
sign_typed_data() (*in module web3.parity.personal*), 212
sign_typed_data() (*web3.eth.Eth method*), 179
signTransaction() (*web3.eth.Eth method*), 177
signTypedData() (*in module web3.parity.personal*), 212
signTypedData() (*web3.eth.Eth method*), 179
solidityKeccak() (*web3.Web3 class method*), 161
soliditySha3() (*web3.Web3 class method*), 162
start() (*web3.geth.miner.GethMiner method*), 201
start_auto_dag() (*web3.geth.miner.GethMiner method*), 201
start_rpc() (*in module web3.geth.admin*), 204
start_ws() (*in module web3.geth.admin*), 204
startAutoDag() (*web3.geth.miner.GethMiner method*), 201
startRPC() (*in module web3.geth.admin*), 204
startWS() (*in module web3.geth.admin*), 204
status() (*web3.geth.txpool.TxPool method*), 208
stop() (*web3.geth.miner.GethMiner method*), 201
stop_auto_dag() (*web3.geth.miner.GethMiner method*), 201
stop_rpc() (*in module web3.geth.admin*), 204
stop_ws() (*in module web3.geth.admin*), 205
stopAutoDag() (*web3.geth.miner.GethMiner method*), 201
stopRPC() (*in module web3.geth.admin*), 204
stopWS() (*in module web3.geth.admin*), 205
submit_hashrate() (*web3.eth.Eth method*), 183
submit_work() (*web3.eth.Eth method*), 184
submitHashrate() (*web3.eth.Eth method*), 183
submitWork() (*web3.eth.Eth method*), 184
syncing (*web3.eth.Eth attribute*), 164

T

toBytes() (*web3.Web3 method*), 159
toChecksumAddress() (*web3.Web3 method*), 160
toHex() (*web3.Web3 method*), 158
toInt() (*web3.Web3 method*), 159
toJSON() (*web3.Web3 method*), 160
toText() (*web3.Web3 method*), 159
toWei() (*web3.Web3 method*), 160
transact() (*web3.contract.Contract.fallback method*), 114
transact() (*web3.contract.ContractFunction method*), 112
TransactionFilter (*class in web3.utils.filters*), 99

U

UnauthorizedError, 217
UnderfundedBid, 218
uninstall_filter() (*web3.eth.Eth method*), 183
uninstallFilter() (*web3.eth.Eth method*), 183
unlinked_references (*LinkableContract attribute*), 136

unlock_account() (*in module web3.geth.personal*), 206
unlock_account() (*in module web3.parity.personal*), 212
unlockAccount() (*in module web3.geth.personal*), 206
unlockAccount() (*in module web3.parity.personal*), 212
UnownedName, 217
uri() (*ethpm.Package property*), 133

V

version() (*ethpm.Package property*), 133
version() (*in module web3.net*), 199

W

w3 (*Package attribute*), 134
wait_for_transaction_receipt() (*web3.eth.Eth method*), 174
waitForTransactionReceipt() (*web3.eth.Eth method*), 174
web3
    module, 158
Web3 (*class in web3*), 158
web3.contract
    module, 103
web3.eth
    module, 163
web3.gas_strategies.rpc
    module, 214
web3.gas_strategies.time_based
    module, 214
web3.geth
    module, 201
web3.geth.admin
    module, 202
web3.geth.miner
    module, 200
web3.geth.personal
    module, 205
web3.geth.txpool
    module, 207
web3.net
    module, 199
web3.parity
    module, 211
web3.parity.personal
    module, 211
web3.providers.async_rpc.AsyncHTTPProvider (*class in web3.providers.eth_tester*), 62
web3.providers.ipc.IPCProvider (*built-in class*), 60
web3.providers.rpc.HTTPProvider (*built-in class*), 60
web3.providers.websocket.WebsocketProvider (*built-in class*), 61
web3.utils.filters
    module, 98