Populus Documentation

Release 4.8.2

Piper Merriam

Nov 15, 2018

Contents

Web3.py is a python library for interacting with Ethereum. Its API is derived from the Web3.js Javascript API and should be familiar to anyone who has used `web3.js`.

CHAPTER 1

Contents

1.1 Quickstart

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

1.1.1 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 *Set up a clean environment*.

Installation from source can be done from the root of the project with the following command.

```
$ pip install .
```

1.1.2 Using Web3

To use the web3 library you will need to initialize the *Web3* class.

Use the `auto` module to *guess at common node connection options*.

```
>>> from web3.auto import w3
>>> w3.eth.blockNumber
4000000
```

This `w3` instance will now allow you to interact with the Ethereum blockchain.

Note: If you get the result `UnhandledRequest:  No providers responded to the RPC request` then you are not connected to a node. See *Why do I need to connect to a node?* and *Choosing How to Connect to Your Node*

1.1.3 Getting Blockchain Info

It's time to start using Web3.py! Try getting all the information about the latest block.

```
>>> w3.eth.getBlock('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': []}
```

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()`.

1.2 Overview

The common entrypoint for interacting with the Web3 library is the `Web3` object. The web3 object provides APIs for interacting with the ethereum blockchain, typically by connecting to a JSON-RPC server.

1.2.1 Providers

Providers are how web3 connects to the blockchain. The Web3 library comes with a the following built-in providers that should be suitable for most normal use cases.

- `web3.HTTPProvider` for connecting to http and https based JSON-RPC servers.
- `web3.IPCProvider` for connecting to ipc socket based JSON-RPC servers.
- `web3.WebsocketProvider` for connecting to ws and wss websocket based JSON-RPC servers.

The `HTTPProvider` takes the full URI where the server can be found. For local development this would be something like `http://localhost:8545`.

The `IPCProvider` takes the filesystem path where the IPC socket can be found. If no argument is provided it will use the *default* path for your operating system.

The `WebsocketProvider` takes the full URI where the server can be found. For local development this would be something like `ws://127.0.0.1:8546`.

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

# Note that you should create only one RPCProvider per
# process, as it recycles underlying TCP/IP network connections between
# your process and Ethereum node
>>> web3 = Web3(HTTPProvider('http://localhost:8545'))

# or for an IPC based connection
>>> web3 = Web3(IPCProvider())

# or for Websocket based connection
>>> web3 = Web3(WebsocketProvider('ws://127.0.0.1:8546'))
```

1.2.2 Base API

The `Web3` class exposes the following convenience APIs.

Type Conversions

`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'
>>> 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'
```

(continues on next page)

(continued from previous page)

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

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')
```

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'
```

Cryptographic Hashing

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

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*)

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")
```

1.2.3 Modules

The JSON-RPC functionality is split across multiple modules which *loosely* correspond to the namespaces of the underlying JSON-RPC methods.

1.3 Your Ethereum Node

1.3.1 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.

1.3.2 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.sendTransaction()` 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 *How do I use my MetaMask accounts from Web3.py?*

1.3.3 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*.

1.4 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:

1.4.1 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.

1.4.2 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.

1.4.3 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]
```

1.4.4 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 *Type Conversions*.

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()`*

1.4.5 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

1.4.6 Personal API

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

1.5 Filtering

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 = web3.eth.filter({"address": contract_address})
  ```

- Attaching to an existing filter

  ```
  from web3.auto import w3
  existing_filter = web3.eth.filter(filter_id="0x0")
  ```

1.5.1 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.getFilterChanges()* 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.getFilterLogs()* 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`.

1.5.2 Block and Transaction Filter Classes

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

BlockFilter is a subclass of :class:`Filter`.

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

```
>>> new_block_filter = web.eth.filter('latest')
>>> new_block_filter.get_new_entries()
```

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

TransactionFilter is a subclass of :class:`Filter`.

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

```
>>> new_transaction_filter = web.eth.filter('pending')
>>> new_transaction_filter.get_new_entries()
```

1.5.3 Event Log Filters

You can set up a filter for event logs using the web3.py contract api: `web3.contract.Contract.events.<event_name>.createFilter()`, which provides some conveniances 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.<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.sha3(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.

1.5.4 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:
        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
import asyncio


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


async 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():
    loop = asyncio.new_event_loop()
```

(continues on next page)

(continued from previous page)

```
    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

1.6 Contracts

It is worth taking your time to understand all about contracts. To get started, check out this example:

1.6.1 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]`.
- The `solc` solidity compiler. See Installing the Solidity Compiler

```
import json
import web3

from web3 import Web3
from solc import compile_source
from web3.contract import ConciseContract

# Solidity source code
contract_source_code = '''
pragma solidity ^0.4.21;

contract Greeter {
    string public greeting;

    function Greeter() public {
        greeting = 'Hello';
    }

    function setGreeting(string _greeting) public {
        greeting = _greeting;
    }

    function greet() view public returns (string) {
        return greeting;
    }
}
'''
```

(continues on next page)

(continued from previous page)

```
compiled_sol = compile_source(contract_source_code) # Compiled source code
contract_interface = compiled_sol['<stdin>:Greeter']

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

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

# Instantiate and deploy contract
Greeter = w3.eth.contract(abi=contract_interface['abi'], bytecode=contract_interface[
↪'bin'])

# 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.waitForTransactionReceipt(tx_hash)

# Create the contract instance with the newly-deployed address
greeter = w3.eth.contract(
    address=tx_receipt.contractAddress,
    abi=contract_interface['abi'],
)

# Display the default greeting from the contract
print('Default contract greeting: {}'.format(
    greeter.functions.greet().call()
))

print('Setting the greeting to Nihao...')
tx_hash = greeter.functions.setGreeting('Nihao').transact()

# Wait for transaction to be mined...
w3.eth.waitForTransactionReceipt(tx_hash)

# Display the new greeting value
print('Updated contract greeting: {}'.format(
    greeter.functions.greet().call()
))

# When issuing a lot of reads, try this more concise reader:
reader = ConciseContract(greeter)
assert reader.greet() == "Nihao"
```

1.6.2 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()*)

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()*)

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)
```

1.6.3 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`.

1.6.4 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.sendTransaction(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.estimateGas()` method.

Returns the transaction hash for the deploy transaction.

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

classmethod `Contract.constructor`(**args*, ***kwargs*).*estimateGas(transaction=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.

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 sendTransaction method.

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

Contract.events.<event name>.createFilter(fromBlock=block, toBlock=block, argument_filters=
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 the 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.`**`eventFilter`**(*event_name*, *filter_params=None*)

> **Warning:** Contract.eventFilter() has been deprecated for `Contract.events.<event name>.createFilter()`

Creates a new *`web3.utils.filters.LogFilter`* instance.

The `event_name` parameter should be the name of the contract event you want to filter on.

If provided, `filter_params` should be a dictionary specifying additional filters for log entries. The following keys are supported.

- `filter`: `dictionary` - (optional) Dictionary keys should be argument names for the Event arguments. Dictionary values should be the value you want to filter on, or a list of values to be filtered on. Lists of values will match log entries whose argument matches any value in the list. Indexed and unindexed event arguments are accepted. The processing of indexed argument values into hex encoded topics is handled internally when using the `filter` parameter.
- `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, with the keccak hash of the event signature as the first item, and the remaining items as hex encoded argument values. 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. This argument is useful when relying on the internally generated topic lists via the `filter` argument is not desired. If `topics` is included with the `filter` argument, the `topics` list will be prepended to any topic lists inferred from the `filter` arguments.

The event topic for the event specified by `event_name` will be added to the `filter_params['topics']` list.

If the *`Contract.address`* attribute for this contract is non-null, the contract address will be added to the `filter_params`.

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.sendTransaction(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.estimateGas()` method.

Returns the transaction hash for the deploy transaction.

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.

1.6.5 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)
<Function identity(uint256,bool) bound to (1, True)>
>>> identity_func(1, True).call()
1
```

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 its pending.
- `blockNumber`: Number - the block number where this log was in. null when its pending.

```
>>> transfer_filter = my_token_contract.eventFilter('Transfer', {'filter': {
↪'_from': '0xdc3a9db694bcdd55ebae4a89b22ac6d12b3f0c24'}})
>>> transfer_filter.get_new_entries()
[...]  # array of Event Log Objects that match the filter.

# wait a while...

>>> transfer_filter.get_new_entries()
[...]  # new events since the last call

>>> transfer_filter.get_all_entries()
[...]  # all events that match the filter.
```

1.6.6 Contract Functions

class web3.contract.**ContractFunction**

The named functions exposed through the *Contract.functions* property are of the ContractFunction type. This class it 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.sendTransaction(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.estimateGas()` 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')
```

`ContractFunction.`**`estimateGas`**(*transaction*)

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
```

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.
↪getTransactionCount('0xF5...')})
```

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

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

```
>>> math_contract.functions.increment(5).buildTransaction({'gasPrice':␣
↪21000000000})
{
    'to': '0x6Bc272FCFcf89C14cebFC57B8f1543F5137F97dE',
    'data':
↪'0x7cf5dab0000000000000000000000000000000000000000000000000000000000000005',
    '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.

1.6.7 Events

`class` `web3.contract.`**`ContractEvents`**

The named events exposed through the *`Contract.events`* property are of the ContractEvents type. This class it 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.getTransactionReceipt(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*)
Extracts the pertinent logs from a transaction receipt.

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.getTransactionReceipt(tx_hash)
>>> rich_logs = contract.events.myEvent().processReceipt(tx_receipt)
>>> rich_logs[0]['args']
{'myArg': 12345}
```

1.6.8 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.getTransaction(
↪'0x5798fbc45e3b63832abc4984b0f3574a13545f415dd672cd8540cd71f735db56')
>>> transaction.input

↪'0x612e45a3000000000000000000000000b656b2a9c3b2416437a811e07466ca712f5a5b5a0000000000000000000
↪'
>>> 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'',
```

(continues on next page)

(continued from previous page)

```
  '_debatingPeriod': 604800,
  '_newCurator': True})
```

1.7 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.

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

1.7.1 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*.

1.7.2 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 succesful 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
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:
    - *Type Conversions*
    - *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.version.node.startswith('Parity'):
    enode = w3.parity.enode
```

(continues on next page)

(continued from previous page)

```
elif connected and w3.version.node.startswith('Geth'):
    enode = w3.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 the 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`

1.7.3 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 API key if you don't have one at https://infura.io/signup .

Then set the environment variable `INFURA_API_KEY` with your API key:

```
$ export INFURA_API_KEY=YourApiKey
```

```
>>> 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
```

1.7.4 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*])

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` this should be a dictionary of keyword arguments which will be passed onto the http/https request.

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

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.:56

```
>>> from web3 import Web3
>>> web3 = 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`
- On Mac OS:
  - `~/Library/Ethereum/geth.ipc`
  - `~/Library/Application Support/io.parity.ethereum/jsonrpc.ipc`
- On Windows:
  - `\\\.\pipe\geth.ipc`
  - `\\\.\pipe\jsonrpc.ipc`

WebsocketProvider

class web3.providers.websocket.**WebsocketProvider**(*endpoint_uri*[, *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_kwargs` this should be a dictionary of keyword arguments which will be passed onto the ws/wss websocket connection.

```
>>> from web3 import Web3
>>> web3 = 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. A common use case for this is increasing the timeout for each request.

```
>>> from web3 import Web3
>>> web3 = Web3(Web3.WebsocketProvider("http://127.0.0.1:8546", websocket_kwargs={
↪'timeout': 60}))
```

EthereumTesterProvider

Warning: Experimental: This provider is experimental. There are still significant gaps in functionality. However, it is the default replacement for *web3.providers.tester.EthereumTesterProvider* and 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` 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())
```

EthereumTesterProvider (legacy)

Warning: Deprecated: This provider is deprecated in favor of *EthereumTesterProvider* and the newly created eth-tester.

class web3.providers.tester.**EthereumTesterProvider**

This provider can be used for testing. It uses an ephemeral blockchain backed by the `ethereum.tester` module.

```
>>> from web3 import Web3
>>> from web3.providers.tester import EthereumTesterProvider
>>> w3 = Web3(EthereumTesterProvider())
```

TestRPCProvider

Warning: Deprecated: This provider is deprecated in favor of *EthereumTesterProvider* and the newly created eth-tester.

class web3.providers.eth_tester.**TestRPCProvider**

This provider can be used for testing. It uses an ephemeral blockchain backed by the `ethereum.tester` module. This provider will be slower than the `EthereumTesterProvider` since it uses an HTTP server for RPC interactions with.

AutoProvider

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

1.7.5 Using Multiple Providers

Web3 supports the use of multiple providers. This is useful for cases where you wish to delegate requests across different providers. To use this feature, simply instantiate your web3 instance with an iterable of provider instances.

```
>>> from web3 import Web3, HTTPProvider
>>> from . import MySpecialProvider
>>> special_provider = MySpecialProvider()
>>> infura_provider = HTTPProvider('https://ropsten.infura.io')
>>> web3 = Web3([special_provider, infura_provider])
```

When web3 has multiple providers it will iterate over them in order, trying the RPC request and returning the first response it receives. Any provider which *cannot* respond to a request **must** throw a `web3.exceptions.CannotHandleRequest` exception.

If none of the configured providers are able to handle the request, then a `web3.exceptions.UnhandledRequest` exception will be thrown.

1.8 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.

1.8.1 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)
```

1.8.2 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'


# ens.py will assume you want a .eth name if you don't specify a full name

assert ns.address('jasoncarver') == eth_address
```

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'
```

1.9 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.

1.9.1 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.getBlock('latest').number` in addition to `eth.getBlock('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 `sendTransaction()` 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

`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

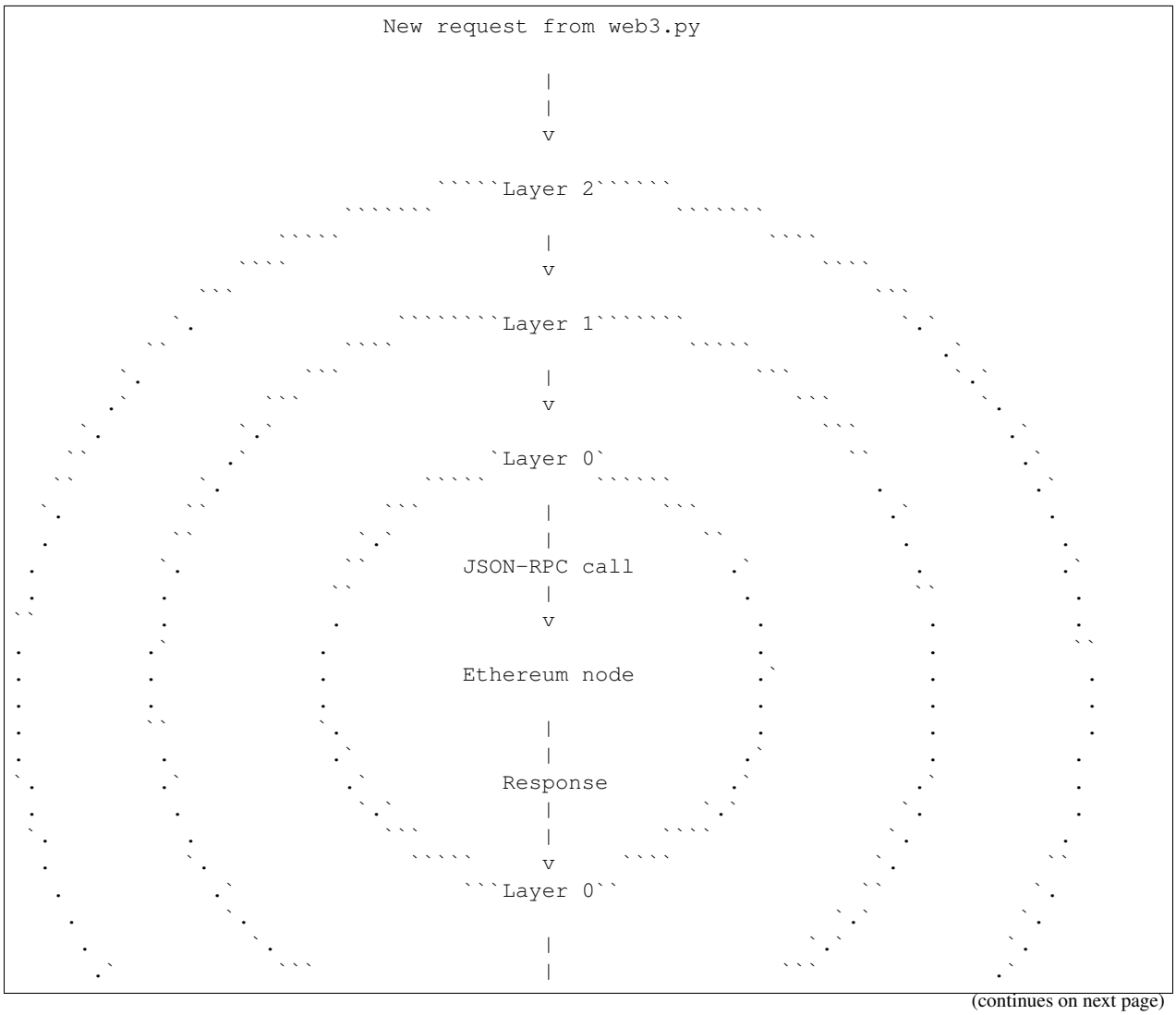
web3.middleware.**http_retry_request_middleware**()
This middleware is a default specifically for HTTPProvider that retries failed requests that return the following errors: *ConnectionError*, *HTTPError*, *Timeout*, *TooManyRedirects*. Additionally there is a whitelist that only allows certain methods to be retried in order to not resend transactions, excluded methods are: *eth_sendTransaction*, *personal_signAndSendTransaction*, *personal_sendTransaction*.

1.9.2 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.

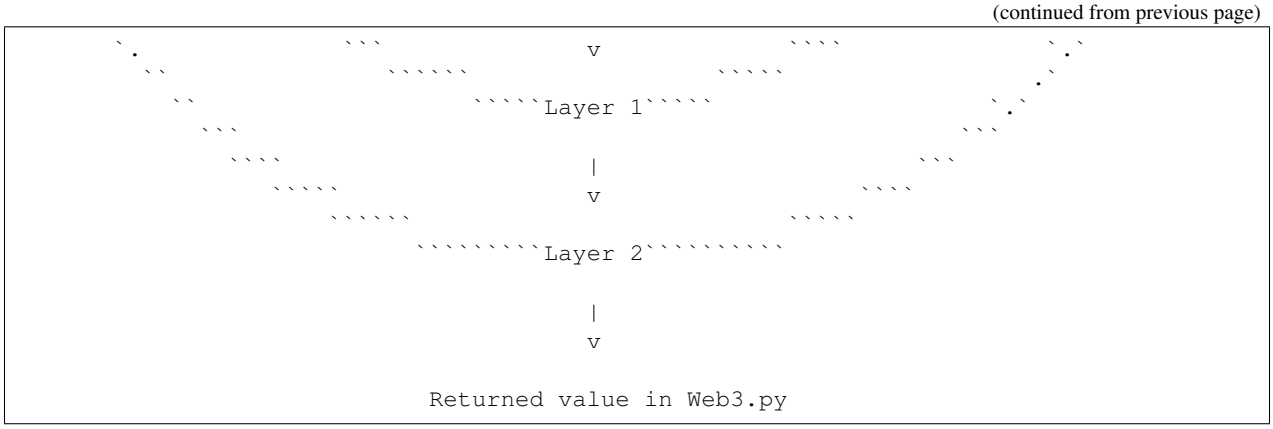
Middleware Order

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:



(continues on next page)

(continued from previous page)



The middlewares are maintained in `Web3.middleware_stack`. 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_stack)` 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_stack.`**`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_stack.add(web3.middleware.pythonic_middleware)
# or
>>> w3.middleware_stack.add(web3.middleware.pythonic_middleware, 'pythonic')
```

`Web3.middleware_stack.`**`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_stack.add()`* .

```
# Either of these will put the pythonic middleware at the innermost layer
>>> w3 = Web3(...)
>>> w3.middleware_stack.inject(web3.middleware.pythonic_middleware, layer=0)
# or
>>> w3.middleware_stack.inject(web3.middleware.pythonic_middleware, 'pythonic',␣
↪layer=0)
```

`Web3.middleware_stack.`**`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_stack.remove(web3.middleware.pythonic_middleware)
# or
>>> w3.middleware_stack.remove('pythonic')
```

Web3.middleware_stack.**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_stack.replace(pythonic_middleware, attrdict_middleware)
# this is now referenced by the new middleware object, so to remove it:
>>> w3.middleware_stack.remove(attrdict_middleware)

# or, if it was named

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

Web3.middleware_stack.**clear**()

Empty all the middlewares, including the default ones.

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

1.9.3 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_stack.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_stack.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.getBlock()`.

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 an amount of time defined by `cache_expire_seconds`.

- `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
```

(continues on next page)

(continued from previous page)

```
>>> w3.version.node
'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_stack.inject(geth_poa_middleware, layer=0)

# confirm that the connection succeeded
>>> w3.version.node
'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.

1.10 Examples

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

1.10.1 Looking up blocks

Blocks can be looked up by either their number or hash using the `web3.eth.getBlock` API. Block hashes should be in their hexadecimal representation. Block numbers

```
# get a block by number
>>> web3.eth.getBlock(12345)
{
    'author': '0xad5c1768e5974c231b2148169da064e61910f31a',
    'difficulty': 735512610763,
    'extraData': '0x476574682f76312e302e302f6c696e75782f676f312e342e32',
    'gasLimit': 5000,
    'gasUsed': 0,
    'hash': '0x767c2bfb3bdee3f78676c1285cd757bcd5d8c272cef2eb30d9733800a78c0b6d',
    'logsBloom':
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
    '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.getBlock(
↪'0x767c2bfb3bdee3f78676c1285cd757bcd5d8c272cef2eb30d9733800a78c0b6d')
{...}
```

1.10.2 Getting the latest block

You can also retrieve the latest block using the string `'latest'` in the `web3.eth.getBlock` API.

```
>>> web3.eth.getBlock('latest')
{...}
```

If you want to know the latest block number you can use the `web3.eth.blockNumber` property.

```
>>> web3.eth.blockNumber
4194803
```

1.10.3 Currency conversions

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

```
>>> web3.toWei('1', 'ether')
1000000000000000000
>>> web3.fromWei('1000000000000000000', 'ether')
Decimal('1')
>>> from_wei(123456789, 'ether')
Decimal('1.23456789E-10')
```

1.10.4 Looking up transactions

You can look up transactions using the `web3.eth.getTransaction` function.

```
>>> web3.eth.getTransaction(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
{
    'blockHash': '0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd',
    'blockNumber': 46147,
    'condition': None,
    'creates': None,
    'from': '0xa1e4380a3b1f749673e270229993ee55f35663b4',
    'gas': 21000,
    'gasPrice': 50000000000000,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'networkId': None,
    'nonce': 0,
    'publicKey':
↪'0x376fc429acc35e610f75b14bc96242b13623833569a5bb3d72c17be7e51da0bb58e48e2462a59897cead8ab88e78709
↪',
```

(continues on next page)

(continued from previous page)

```
    'r': '0x88ff6cf0fefd94db46111149ae4bfc179e9b94721fffd821d38d16464b3f71d0',
    'raw':
↪'0xf86780862d79883d2000825208945df9b87991262f6ba471f09758cde1c0fc1de734827a69801ca088ff6cf0fefd94db
↪',
    '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`.

1.10.5 Looking up receipts

Transaction receipts can be retrieved using the `web3.eth.getTransactionReceipt` API.

```
>>> web3.eth.getTransactionReceipt(
↪'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`.

1.10.6 Working with 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;
        MyEvent(_var);
    }

    function getVar() public view returns (uint8) {
        return _myVar;
```

(continues on next page)

(continued from previous page)

```
    }

}
```

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 solc 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']).deploy()

    address = w3.eth.getTransactionReceipt(tx_hash)['contractAddress']
    return address


def wait_for_receipt(w3, tx_hash, poll_interval):
   while True:
       tx_receipt = w3.eth.getTransactionReceipt(tx_hash)
       if tx_receipt:
         return tx_receipt
       time.sleep(poll_interval)


w3 = Web3(EthereumTesterProvider())

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("Deployed {0} to: {1}\n".format(contract_id, address))

store_var_contract = w3.eth.contract(
```

(continues on next page)

(continued from previous page)

```
    address=address,
    abi=contract_interface['abi'])

gas_estimate = store_var_contract.functions.setVar(255).estimateGas()
print("Gas estimate to transact with setVar: {0}\n".format(gas_estimate))

if gas_estimate < 100000:
  print("Sending transaction to setVar(255)\n")
  tx_hash = store_var_contract.functions.setVar(255).transact()
  receipt = wait_for_receipt(w3, tx_hash, 1)
  print("Transaction receipt mined: \n")
  pprint.pprint(dict(receipt))
else:
  print("Gas cost exceeds 100000")
```

Output:

```
Deployed <stdin>:StoreVar to: 0xF2E246BB76DF876Cef8b38ae84130F4F55De395b

Gas estimate to transact with setVar: 32463

Sending transaction to setVar(255)

Transaction receipt mined:

{'blockHash': HexBytes(
↪'0x94e07b0b88667da284e914fa44b87d4e7fec39761be51245ef94632a3b5ab9f0'),
 'blockNumber': 2,
 'contractAddress': None,
 'cumulativeGasUsed': 43106,
 'gasUsed': 43106,
 'logs': [AttributeDict({'type': 'mined', 'logIndex': 0, 'transactionIndex': 0,
↪'transactionHash': HexBytes(
↪'0x3ac3518cc59d1698aa03a0bab7fb8191a4ef017aeda7429b11e8c6462b20a62a'), 'blockHash':␣
↪HexBytes('0x94e07b0b88667da284e914fa44b87d4e7fec39761be51245ef94632a3b5ab9f0'),
↪'blockNumber': 2, 'address': '0xF2E246BB76DF876Cef8b38ae84130F4F55De395b', 'data':
↪'0x', 'topics': [HexBytes(
↪'0x6c2b4666ba8da5a95717621d879a77de725f3d816709b9cbe9f059b8f875e284'), HexBytes(
↪'0x00000000000000000000000000000000000000000000000000000000000000ff')]})],
 'transactionHash': HexBytes(
↪'0x3ac3518cc59d1698aa03a0bab7fb8191a4ef017aeda7429b11e8c6462b20a62a'),
 'transactionIndex': 0}
```

1.10.7 Interacting with an ERC20 Contract

The ERC20 (formally EIP20) token standard is the most widely used standard in Ethereum. Here's how to check the current state of an ERC20 token contract.

First, create your Python object representing the ERC20 contract. For this example, we'll be inspecting the Unicorns token contract.

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

>>> ERC20_ABI = json.loads('[{"constant":true,"inputs":[],"name":"name","outputs":[{
↪"name":"","type":"string"}],"payable":false,"stateMutability":"view","type":
↪"function"},{"constant":false,"inputs":[{"name":"_spender","type":"address"},{"name
↪":"_value","type":"uint256"}],"name":"approve","outputs":[{"name":"","type":"bool"}
↪],"payable":false,"stateMutability":"nonpayable","type":"function"},{"constant
↪":true,"inputs":[],"name":"totalSupply","outputs":[{"name":"","type":"uint256"}],
↪"payable":false,"stateMutability":"view","type":"function"},{"constant":false,
↪"inputs":[{"name":"_from","type":"address"},{"name":"_to","type":"address"},{"name":
↪"_value","type":"uint256"}],"name":"transferFrom","outputs":[{"name":"","type":"bool
```

(continues on next page)

(continued from previous page)

```
# Address field should be the checksum address at which the ERC20 contract was␣
↪deployed
>>> erc20 = w3.eth.contract(address='0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7',␣
↪abi=ERC20_ABI)
```

Fetch various data about the current state of the ERC20 contract.

```
>>> erc20.functions.name().call()
'Unicorns'
>>> erc20.functions.symbol().call()
''
>>> erc20.functions.decimals().call()
0
>>> erc20.functions.totalSupply().call()
2039
```

Get the balance of an account address.

```
>>> erc20.functions.balanceOf('0xD1220A0cf47c7B9Be7A2E6BA89F429762e7b9aDb').call()
1
```

Calculate the token count from the token balance of an account address.

```
>>> from decimal import Decimal

# Most applications expect *exact* results, so using the Decimal library,
# with default precision of 28, is usually sufficient to avoid any rounding error.
>>> decimals = erc20.functions.decimals().call()
>>> balance = erc20.functions.balanceOf('0xD1220A0cf47c7B9Be7A2E6BA89F429762e7b9aDb').
↪call()
>>> balance / Decimal(10 ** decimals)
Decimal('1')
```

Balance is *always* an integer in the currency's smallest natural unit (equivalent to 'wei' for ether). Token balance is typically used only for backend calculations.

Token count *may* be a integer or fraction in the currency's primary unit (equivalent to 'eth' for ether). Token count is typically displayed to the user on the front-end since it is more readable.

1.11 Troubleshooting

1.11.1 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
```

(continues on next page)

(continued from previous page)

```
# *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`

1.11.2 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*.

1.11.3 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:
  - ropsten.be
  - Kyber

1.12 Web3 API

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

Each web3 instance exposes the following APIs.

1.12.1 Providers

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

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

Web3.**setProviders**(*provider*)
Updates the current web3 instance with the new list of providers. It also accepts a single provider.

1.12.2 Encoding and Decoding Helpers

See *Type Conversions*

1.12.3 Currency Conversions

See *Currency Conversions*

1.12.4 Addresses

See *Addresses*

RPC APIS

Each web3 instance also exposes these namespaced APIs.

Web3.**eth**
See *web3.eth API*

Web3.**shh**
See *SHH API*

Web3.**personal**
See *Personal API*

Web3.**version**
See *Version API*

Web3.**txpool**
See *TX Pool API*

`Web3.`**`miner`**
See *Miner API*

`Web3.`**`admin`**
See *Admin API*

1.13 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.getBlock('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
```

1.13.1 Properties

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

`Eth.`**`defaultAccount`**
The ethereum address that will be used as the default `from` address for all transactions.

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

`Eth.`**`syncing`**

- Delegates to `eth_syncing` RPC Method

Returns either `False` if the node is not syncing or a dictionary showing sync status.

```
>>> web3.eth.syncing
AttributeDict({
    'currentBlock': 2177557,
    'highestBlock': 2211611,
    'knownStates': 0,
```

(continues on next page)

(continued from previous page)

```
    '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.**gasPrice**

- Delegates to `eth_gasPrice` RPC Method

Returns the current gas price in Wei.

```
>>> web3.eth.gasPrice
20000000000
```

Eth.**accounts**

- Delegates to `eth_accounts` RPC Method

Returns the list of known accounts.

```
>>> web3.eth.accounts
['0xd3cda913deb6f67967b99d67acdfa1712c293601']
```

Eth.**blockNumber**

- Delegates to `eth_blockNumber` RPC Method

Returns the number of the most recent block

```
>>> web3.eth.blockNumber
2206939
```

1.13.2 Methods

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

`Eth.`**`getBalance`**`(`*account*, *block_identifier=eth.defaultBlock*`)`

- Delegates to `eth_getBalance` RPC Method

Returns the balance of the given `account` at the block specified by `block_identifier`.

`account` may be a hex address or an ENS name

```
>>> web3.eth.getBalance('0xd3cda913deb6f67967b99d67acdfa1712c293601')
77320681768999138915
```

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

- 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 hex address or an ENS name

```
>>> web3.eth.getStorageAt('0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b', 0)
'0x00000000000000000000000000000000000000000000000000120a0b063499d4'
```

`Eth.`**`getCode`**`(`*account*, *block_identifier=eth.defaultBlock*`)`

- Delegates to `eth_getCode` RPC Method

Returns the bytecode for the given `account` at the block specified by `block_identifier`.

`account` may be a hex address or an ENS name

```
# For a contract address.
>>> web3.eth.getCode('0x6c8f2a135f6ed072de4503bd7c4999a1a17f824b')
'0x6060604052361561027c5760e060020a60003504630199.....'
# For a private key address.
>>> web3.eth.getCode('0xd3cda913deb6f67967b99d67acdfa1712c293601')
'0x'
```

`Eth.`**`getBlock`**`(`*block_identifier=eth.defaultBlock*, *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`.

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.getBlock(2000000)
AttributeDict({
    'difficulty': 49824742724615,
    'extraData': '0xe4b883e5bda9e7a59ee4bb99e9b1bc',
    'gasLimit': 4712388,
    'gasUsed': 21000,
    'hash': '0xc0f4906fea23cf6f3cce98cb44e8e1449e455b28d684dfa9ff65426495584de6',
    'logsBloom':
↪'0x0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
```

(continues on next page)

(continued from previous page)

```
    'miner': '0x61c808d82a3ac53231750dadc13c777b59310bd9',
    'nonce': '0x3b05c6d5524209f1',
    'number': 2000000,
    'parentHash':
↪'0x57ebf07eb9ed1137d41447020a25e51d30a0c272b5896571499c82c33ecb7288',
    'receiptRoot':
↪'0x84aea4a7aad5c5899bd5cfc7f309cc379009d30179316a2a7baa4a2ea4a438ac',
    'sha3Uncles':
↪'0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347',
    'size': 650,
    'stateRoot':
↪'0x96dbad955b166f5119793815c36f11ffa909859bbfeb64b735cca37cbf10bef1',
    'timestamp': 1470173578,
    'totalDifficulty': 4401010182770540938,
    'transactions': [
↪'0xc55e2b90168af6972193c1f86fa4d7d7b31a29c156665d15b9cd48618b5177ef'],
    'transactionsRoot':
↪'0xb31f174d27b99cdae8e746bd138a01ce60d8dd7b224f7c60845914def05ecc58',
    'uncles': [],
})
```

`Eth.`**`getBlockTransactionCount`**(*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`.

```
>>> web3.eth.getBlockTransactionCount(46147)
1
>>> web3.eth.getBlockTransactionCount(
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd')  # block␣
↪46147
1
```

`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.getUncleByBlock`

`Eth.`**`getUncleByBlock`**(*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`.

```
>>> web3.eth.getUncleByBlock(56160, 0)
AttributeDict({
```

(continues on next page)

(continued from previous page)

```
  '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.getUncleByBlock(
↪'0x685b2226cbf6e1f890211010aa192bf16f0a0cba9534264a033b023d7367b845', 0)
AttributeDict({
    ...
})
```

Eth.**getTransaction**(*transaction_hash*)

- Delegates to `eth_getTransactionByHAsh` RPC Method

Returns the transaction specified by `transaction_hash`

```
>>> web3.eth.getTransaction(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
AttributeDict({
    'blockHash':
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd',
    'blockNumber': 46147,
    'from': '0xa1e4380a3b1f749673e270229993ee55f35663b4',
    'gas': 21000,
    'gasPrice': 50000000000000,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'nonce': 0,
```

(continues on next page)

(continued from previous page)

```
    'to': '0x5df9b87991262f6ba471f09758cde1c0fc1de734',
    'transactionIndex': 0,
    'value': 31337,
})
```

Eth.**getTransactionFromBlock**(*block_identifier*, *transaction_index*)

Note: This method is deprecated and replaced by `Eth.getTransactionByBlock`

Eth.**getTransactionByBlock**(*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`.

```
>>> web3.eth.getTransactionFromBlock(46147, 0)
AttributeDict({
    'blockHash':
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd',
    'blockNumber': 46147,
    'from': '0xa1e4380a3b1f749673e270229993ee55f35663b4',
    'gas': 21000,
    'gasPrice': 50000000000000,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'nonce': 0,
    'to': '0x5df9b87991262f6ba471f09758cde1c0fc1de734',
    'transactionIndex': 0,
    'value': 31337,
})
>>> web3.eth.getTransactionFromBlock(
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd', 0)
AttributeDict({
    'blockHash':
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd',
    'blockNumber': 46147,
    'from': '0xa1e4380a3b1f749673e270229993ee55f35663b4',
    'gas': 21000,
    'gasPrice': 50000000000000,
    'hash': '0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'input': '0x',
    'nonce': 0,
    'to': '0x5df9b87991262f6ba471f09758cde1c0fc1de734',
    'transactionIndex': 0,
    'value': 31337,
})
```

Eth.**waitForTransactionReceipt**(*transaction_hash*, *timeout=120*)

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 *`waitForTransactionReceipt()`* raises a *Timeout* exception.

```
>>> web3.eth.waitForTransactionReceipt(
↪'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': [],
    'root': '96a8e009d2b88b1483e6941e6812e32263b05683fac202abc622a3e31aed1957',
    'to': '0x5df9b87991262f6ba471f09758cde1c0fc1de734',
    'transactionHash':
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'transactionIndex': 0,
})
```

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

- Delegates to `eth_getTransactionReceipt` RPC Method

Returns the transaction receipt specified by `transaction_hash`. If the transaction has not yet been mined returns `None`

```
>>> web3.eth.getTransactionReceipt(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')  # not␣
↪yet mined
None
# wait for it to be mined....
>>> web3.eth.getTransactionReceipt(
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060')
AttributeDict({
    'blockHash':
↪'0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd',
    'blockNumber': 46147,
    'contractAddress': None,
    'cumulativeGasUsed': 21000,
    'from': '0xa1e4380a3b1f749673e270229993ee55f35663b4',
    'gasUsed': 21000,
    'logs': [],
    'root': '96a8e009d2b88b1483e6941e6812e32263b05683fac202abc622a3e31aed1957',
    'to': '0x5df9b87991262f6ba471f09758cde1c0fc1de734',
    'transactionHash':
↪'0x5c504ed432cb51138bcf09aa5e8a410dd4a1e204ef84bfed1be16dfba1b22060',
    'transactionIndex': 0,
})
```

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

- 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 hex address or an ENS name

```
>>> web3.eth.getTransactionCount('0xd3cda913deb6f67967b99d67acdfa1712c293601')
340
```

`Eth.`**`sendTransaction`**(*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`, hex address or ENS name - (optional, default: `web3.eth.defaultAccount`) The address the transaction is send from.
- `to`: `bytes or text`, hex address or ENS name - (optional when creating new contract) The address the transaction is directed to.
- `gas`: `integer` - (optional, default: 90000) Integer of the gas provided for the transaction execution. It will return unused gas.
- `gasPrice`: `integer` - (optional, default: To-Be-Determined) Integer of the gasPrice used for each paid gas
- `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 *`estimateGas()`* function with an additional buffer of `100000` gas up to the `gasLimit` of the latest block. In the event that the value returned by *`estimateGas()`* method is greater than the `gasLimit` a `ValueError` will be raised.

```
>>> web3.eth.sendTransaction({'to': '0xd3cda913deb6f67967b99d67acdfa1712c293601',
↪'from': web3.eth.coinbase, 'value': 12345})
'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331'
```

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

- Delegates to `eth_sendRawTransaction` RPC Method

Sends a signed and serialized transaction. Returns the transaction hash.

```
>>> signed_txn = w3.eth.account.signTransaction(dict(
    nonce=w3.eth.getTransactionCount(w3.eth.coinbase),
    gasPrice=w3.eth.gasPrice,
    gas=100000,
    to='0xd3cda913deb6f67967b99d67acdfa1712c293601',
    value=12345,
    data=b'',
  ),
  private_key_for_senders_account,
)
>>> w3.eth.sendRawTransaction(signed_txn.rawTransaction)
'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331'
```

`Eth.`**`replaceTransaction`**(*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 *`sendTransaction()`*. 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 a `gasPrice` value, it must be greater than the pending transaction's `gasPrice`.

If the `new_transaction` does not specify a `gasPrice` value, the highest of the following 2 values will be used:

- The pending transaction's `gasPrice` * 1.1 - This is typically the minimum `gasPrice` increase a node requires before it accepts a replacement transaction.
- The `gasPrice` as calculated by the current gas price strategy(See *Gas Price API*).

This method returns the transaction hash of the replacement transaction.

```
>>> tx = web3.eth.sendTransaction({
        'to': '0xd3cda913deb6f67967b99d67acdfa1712c293601',
        'from': web3.eth.coinbase,
        'value': 1000
    })
'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331'
>>> web3.eth.replaceTransaction(
↪'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331', {
        'to': '0xd3cda913deb6f67967b99d67acdfa1712c293601',
        'from': web3.eth.coinbase,
        'value': 2000
    })
```

`Eth.`**`modifyTransaction`**(*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 *`sendTransaction()`*. The parameter values will override the pending transaction's values to create the replacement transaction to send.

The same validation and defaulting rules of *`replaceTransaction()`* apply.

This method returns the transaction hash of the newly modified transaction.

```
>>> tx = web3.eth.sendTransaction({
        'to': '0xd3cda913deb6f67967b99d67acdfa1712c293601',
        'from': web3.eth.coinbase,
        'value': 1000
    })
'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331'
>>> web3.eth.modifyTransaction(
↪'0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331', value=2000)
```

`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 hex 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.`**`call`**(*transaction*, *block_identifier=web3.eth.defaultBlock*)

- 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 *`sendTransaction()`* 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, 'gasPrice': 1, 'to':
↪'0xc305c901078781C232A2a521C2aF7980f8385ee9', 'data': '0x477a5c98'})
HexBytes('0x0000000000000000000000000000000000000000000000000000000000000001')
```

In most cases it is better to make contract function call through the *`web3.contract.Contract`* interface.

`Eth.`**`estimateGas`**(*transaction*)

- 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` parameter is handled in the same manner as the *`sendTransaction()`* method.

```
>>> web3.eth.estimateGas({'to': '0xd3cda913deb6f67967b99d67acdfa1712c293601',
↪'from': web3.eth.coinbase, 'value': 12345})
21000
```

Eth.**generateGasPrice**(*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.generateGasPrice()
20000000000
```

Note: For information about how gas price can be customized in web3 see *Gas Price API*.

Eth.**setGasPriceStrategy**(*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.

1.13.3 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.**getFilterChanges**(*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.getFilterChanges(filt.filter_id)
[
    {
        'address': '0xdc3a9db694bcdd55ebae4a89b22ac6d12b3f0c24',
        'blockHash':
↪'0xb72256286ca528e09022ffd408856a73ef90e7216ac560187c6e43b4c4efd2f0',
        'blockNumber': 2217196,
        'data':
↪'0x0000000000000000000000000000000000000000000000000000000000000001',
        'logIndex': 0,
        'topics': [
↪'0xe65b00b698ba37c614af350761c735c5f4a82b4ab365a1f1022d49d9dfc8e930',
        '0x000000000000000000000000754c50465885f1ed1fa1a55b95ee8ecf3f1f4324',
        '0x296c7fb6ccafa3e689950b947c2895b07357c95b066d5cdccd58c301f41359a3'],
        'transactionHash':
↪'0xfe1289fd3915794b99702202f65eea2e424b2f083a12749d29b4dd51f6dce40d',
        'transactionIndex': 1,
    },
    ...
]
```

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

- Delegates to `eth_getFilterLogs` RPC Method.

Returns all entries for the given `filter_id`

```
>>> filt = web3.eth.filter()
>>> web3.eth.getFilterLogs(filt.filter_id)
[
    {
        'address': '0xdc3a9db694bcdd55ebae4a89b22ac6d12b3f0c24',
        'blockHash':
↪'0xb72256286ca528e09022ffd408856a73ef90e7216ac560187c6e43b4c4efd2f0',
        'blockNumber': 2217196,
        'data':
↪'0x0000000000000000000000000000000000000000000000000000000000000001',
        'logIndex': 0,
        'topics': [
↪'0xe65b00b698ba37c614af350761c735c5f4a82b4ab365a1f1022d49d9dfc8e930',
        '0x000000000000000000000000754c50465885f1ed1fa1a55b95ee8ecf3f1f4324',
        '0x296c7fb6ccafa3e689950b947c2895b07357c95b066d5cdccd58c301f41359a3'],
        'transactionHash':
↪'0xfe1289fd3915794b99702202f65eea2e424b2f083a12749d29b4dd51f6dce40d',
```

(continues on next page)

(continued from previous page)

```
        'transactionIndex': 1,
    },
    ...
]
```

Eth.**uninstallFilter**(*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.uninstallFilter(filt.filter_id)
True
>>> web3.eth.uninstallFilter(filt.filter_id)
False  # already uninstalled.
```

Eth.**getLogs**(*filter_params*)

This is the equivalent of: creating a new filter, running *getFilterLogs()*, and then uninstalling the filter. See *filter()* for details on allowed filter parameters.

1.13.4 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 hex 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 hex address.

If `address` is *not* provided, the newly created contract class will be returned. That class will then be initialized by supplying the address.

```
from web3 import Web3

w3 = Web3(...)

Contract = w3.eth.contract(abi=...)

# later, initialize contracts with the same metadata at different addresses:
```

(continues on next page)

(continued from previous page)

```
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.`**`setContractFactory`**(*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`.

1.14 Working with Local Private Keys

1.14.1 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.

1.14.2 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 *`sendRawTransaction()`* when working with local keys, instead of *`sendTransaction()`* .

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 *sendTransaction()*.

> **Warning:** It is unnacceptable 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.

1.14.3 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`.

1.14.4 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
```

1.14.5 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 defunct_hash_message

>>> 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_hash = defunct_hash_message(text=msg)
>>> signed_message = w3.eth.account.signHash(message_hash, private_key=private_key)
```

(continues on next page)

(continued from previous page)

```
>>> signed_message
AttrDict({'messageHash': HexBytes(
↪'0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750'),
 'r': 104389933075820307925104709181714897380569894203213074526835978196648170704563,
 's': 28205917190874851400050446352651915501321657673772411533993420917949420456142,
 'v': 28,
 'signature': HexBytes(
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7680c19e
↪')})
```

1.14.6 Verify a Message

With the original message text and a signature:

```
>>> message_hash = defunct_hash_message(text="ISF")
>>> w3.eth.account.recoverHash(message_hash, signature=signed_message.signature)
'0x5ce9454909639D2D17A3F753ce7d93fa0b9aB12E'
```

1.14.7 Verify a Message from message hash

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

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

1.14.8 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),
... )
```

(continues on next page)

(continued from previous page)

```
>>> 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 defunct_hash_message

>>> hex_message = '0x49e299a55346'
>>> hex_signature =
↪'0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb33e5bfbbf4d3e39b1a2fd816a7680c19e
↪'

# ecrecover in Solidity expects a prefixed & hashed version of the message
>>> message_hash = defunct_hash_message(hexstr=hex_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)
>>> ec_recover_args
('0x1476abb745d423bf09273f1afd887d951181d25adc66c4834a70491911b7f750',
 28,
 '0xe6ca9bba58c88611fad66a6ce8f996908195593807c4b38bd528d2cff09d4eb3',
 '0x3e5bfbbf4d3e39b1a2fd816a7680c19ebebaf3a141b239934ad43cb33fcec8ce')
```

1.14.9 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`.

1.14.10 Sign a Transaction

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

```
>>> transaction = {
...     'to': '0xF0109fC8DF283027b6285cc889F5aA624EaC1F55',
...     'value': 1000000000,
...     'gas': 2000000,
...     'gasPrice': 234567897654321,
...     'nonce': 0,
...     'chainId': 1
... }
>>> key = '0x4c0883a69102937d6231471b5dbb6204fe5129617082792ae468d01a3f362318'
>>> signed = w3.eth.account.signTransaction(transaction, key)
>>> signed.rawTransaction
HexBytes(
↪'0xf86a8086d55698372431831e848094f0109fc8df283027b6285cc889f5aa624eac1f55843b9aca008025a009ebb6ca0
↪')
>>> signed.hash
HexBytes('0xd8f64a42b57be0d565f385378db2f6bf324ce14a594afc05de90436e9ce01f60')
>>> signed.r
44872862617934181798178410248897471157793243053758231102491494799905075174044
>>> signed.s
30785525769477805655994251009256770582792548537338581640010273753578382951464
>>> signed.v
37

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

1.14.11 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.signTransaction()`
4. Broadcast the transaction with *sendRawTransaction()*

```
>>> from ethtoken.abi import EIP20_ABI
>>> from web3.auto import w3

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

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

# Build a transaction that invokes this contract's function, called transfer
>>> unicorn_txn = unicorns.functions.transfer(
```

(continues on next page)

(continued from previous page)

```
...     '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':
↪'0xa9059cbb000000000000000000000000fb6916095ca1df60bb79ce92ce3ea74c37c5d3590000000000000000000000000
↪'}

>>> 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.signTransaction(unicorn_txn, private_key=private_key)
>>> signed_txn.hash
HexBytes('0x4795adc6a719fa64fa21822630c0218c04996e2689ded114b6553cef1ae36618')
>>> signed_txn.rawTransaction
HexBytes(
↪'0xf8a980843b9aca008301117094fb6916095ca1df60bb79ce92ce3ea74c37c5d35980b844a9059cbb000000000000000
↪')
>>> signed_txn.r
71048435681527435549920573943347440368602476588132318304215709186344605462 88
>>> signed_txn.s
20971591154030974221209741174186570949918731455961098911091818811306894497524
>>> signed_txn.v
37

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

# When you run sendRawTransaction, you get the same result as the hash of the␣
↪transaction:
>>> w3.toHex(w3.sha3(signed_txn.rawTransaction))
'0x4795adc6a719fa64fa21822630c0218c04996e2689ded114b6553cef1ae36618'
```

1.15 Package Manager API

class web3.pm.**PM**

The `web3.pm` object exposes methods to interact with Packages as defined by ERC 1123.

1.15.1 Installation

Warning: The PM module is still under development, and not all use-cases are currently supported, so it is not included by default in the web3 instance.

You must install the eth-pm module separately, until it is stable. Install with:

```
pip install --upgrade ethpm
```

1.15.2 Attaching

To use `web3.pm`, attach it to your `web3` instance.

```
from web3.pm import PM
PM.attach(web3, 'pm')
```

1.15.3 Methods

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

`PM.`**`get_package_from_manifest`**(*self*, *manifest*)

- Manifest must currently be a dict representing a valid manifest
- Returns a Package instance representing the Manifest

1.16 SHH API

`class` `web3.shh.`**`Shh`**

The `web3.shh` object exposes methods to interact with the RPC APIs under the `shh_` namespace.

Warning: The Whisper protocol is in flux, with incompatible versions supported by different major clients. So it is not currently included by default in the web3 instance.

1.16.1 Properties

The following properties are available on the `web.shh` namespace.

`Shh.`**`version`**
Returns the Whisper version this node offers.

```
>>>web3.shh.version
6.0
```

`Shh.`**`info`**
Returns the Whisper statistics for diagnostics.

```
>>>web3.shh.info
{'maxMessageSize': 1024, 'memory': 240, 'messages': 0, 'minPow': 0.2}
```

1.16.2 Methods

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

`Shh.`**`post`**(*self*, *message*)

- Creates a whisper message and injects it into the network for distribution.
- **Parameters:**
  - `symKeyID`: When using symmetric key encryption, holds the symmetric key ID.
  - `pubKey`: When using asymmetric key encryption, holds the public key.
  - `ttl`: Time-to-live in seconds.
  - `sig (optional)`: ID of the signing key.
  - `topic`: Message topic (four bytes of arbitrary data).
  - `payload`: Payload to be encrypted.
  - `padding (optional)`: Padding (byte array of arbitrary length).
  - `powTime`: Maximal time in seconds to be spent on prrof of work.
  - `powTarget`: Minimal PoW target required for this message.
  - `targetPeer (optional)`: Peer ID (for peer-to-peer message only).
- Returns `True` if the message was succesfully sent, otherwise `False`

```
>>>web3.shh.post({'payload': web3.toHex(text="test_payload"), 'pubKey': recipient_
↪public, 'topic': '0x12340000', 'powTarget': 2.5, 'powTime': 2})
True
```

`Shh.`**`newMessageFilter`**(*self*, *criteria*, *poll_interval=None*)

- Create a new filter within the node. This filter can be used to poll for new messages that match the set of criteria.
- If a `poll_interval` is specified, the client will asynchronously poll for new messages.
- **Parameters:**
  - `symKeyID`: When using symmetric key encryption, holds the symmetric key ID.
  - `privateKeyID`: When using asymmetric key encryption, holds the private key ID.
  - `sig`: Public key of the signature.
  - `minPoW`: Minimal PoW requirement for incoming messages.
  - `topics`: Array of possible topics (or partial topics).
  - `allowP2P`: Indicates if this filter allows processing of direct peer-to-peer messages.
- Returns `ShhFilter` which you can either `watch(callback)` or request `get_new_entries()`

```
>>>web3.shh.newMessageFilter({'topic': '0x12340000', 'privateKeyID': recipient_
↪private})
ShhFilter({'filter_id':
↪'b37c3106cfb683e8f01b5019342399e0d1d74e9160f69b27625faba7a6738554'})
```

Shh.**deleteMessageFilter**(*self*, *filter_id*)

- Deletes a message filter in the node.
- Returns `True` if the filter was sucesfully uninstalled, otherwise `False`

```
>>>web3.shh.deleteMessageFilter(
↪'b37c3106cfb683e8f01b5019342399e0d1d74e9160f69b27625faba7a6738554')
True
```

Shh.**getMessages**(*self*, *filter_id*)

- Retrieve messages that match the filter criteria and are received between the last time this function was called and now.
- Returns all new messages since the last invocation

```
>>>web3.shh.getMessages(
↪'b37c3106cfb683e8f01b5019342399e0d1d74e9160f69b27625faba7a6738554')
[{
    'ttl': 50,
    'timestamp': 1524497850,
    'topic': HexBytes('0x13370000'),
    'payload': HexBytes('0x74657374206d657373616765203a29'),
    'padding': HexBytes(
↪'0x50ab643f1b23bc6df1b1532bb6704ad947c2453366754aade3e3597553eeb96119f4f4299834d9989dc4ecc67e6
↪'),
    'pow': 6.73892030848329,
    'hash': HexBytes(
↪'0x7418f8f0989655ed2f4f9b496e6b1d9be51ef9f0f5ad89f6f750b0eee268b02f'),
    'recipientPublicKey': HexBytes(
↪'0x047d36c9e45fa82fcd27d35bc7d2fd41a2e41e512feec9e4b90ee4293ab12dc2cfc98250a6f5689b07650f8a5ca
↪')
}]
```

Shh.**setMaxMessageSize**(*self*, *size*)

- Sets the maximal message size allowed by this node. Incoming and outgoing messages with a larger size will be rejected. Whisper message size can never exceed the limit imposed by the underlying P2P protocol (10 Mb).
- Returns `True` if the filter was sucesfully uninstalled, otherwise `False`

```
>>>web3.shh.setMaxMessageSize(1024)
True
```

Shh.**setMinPoW**(*self*, *min_pow*)

- Sets the minimal PoW required by this node.
- Returns `True` if the filter was sucesfully uninstalled, otherwise `False`

```
>>>web3.shh.setMinPoW(0.4)
True
```

Shh.**markTrustedPeer**(*self*, *enode*)

- Marks specific peer trusted, which will allow it to send historic (expired) messages.
- Returns `True` if the filter was sucesfully uninstalled, otherwise `False`

```
>>>web3.shh.markTrustedPeer('enode://
↪d25474361659861e9e651bc728a17e807a3359ca0d344afd544ed0f11a31faecaf4d74b55db53c6670fd624f08d5c7
↪178.209.125:30379')
True
```

Asymmetric Keys

`Shh.`**`newKeyPair`**(*self*)

- Generates a new cryptographic identity for the client, and injects it into the known identities for message decryption
- Returns the new key pair's identity

```
>>>web3.shh.newKeyPair()
'86e658cbc6da63120b79b5eec0c67d5dcfb6865a8f983eff08932477282b77bb'
```

`Shh.`**`addPrivateKey`**(*self*, *key*)

- Stores a key pair derived from a private key, and returns its ID.
- Returns the added key pair's ID

```
>>>web3.shh.addPrivateKey(
↪'0x7b8190d96cd061a102e551ee36d08d4f3ca1f56fb0008ef5d70c56271d8c46d0')
'86e658cbc6da63120b79b5eec0c67d5dcfb6865a8f983eff08932477282b77bb'
```

`Shh.`**`deleteKeyPair`**(*self*, *id*)

- Deletes the specifies key if it exists.
- Returns `True` if the key pair was deleted, otherwise `False`

```
>>>web3.shh.deleteKeyPair(
↪'86e658cbc6da63120b79b5eec0c67d5dcfb6865a8f983eff08932477282b77bb')
True
```

`Shh.`**`hasKeyPair`**(*self*, *id*)

- Checks if the whisper node has a private key of a key pair matching the given ID.
- Returns `True` if the key pair exists, otherwise `False`

```
>>>web3.shh.hasKeyPair(
↪'86e658cbc6da63120b79b5eec0c67d5dcfb6865a8f983eff08932477282b77bb')
False
```

`Shh.`**`getPublicKey`**(*self*, *id*)

- Returns the public key associated with the key pair.

```
>>>web3.shh.getPublicKey(
↪'86e658cbc6da63120b79b5eec0c67d5dcfb6865a8f983eff08932477282b77bb')

↪'0x041b0777ceb8cf8748fe0bba5e55039d650a03eb0239a909f9ee345bbbad249f2aa236a4b8f41f51bd0a97d87c0
↪'
```

Shh.**getPrivateKey**(*self*, *id*)

- Returns the private key associated with the key pair.

```
>>>web3.shh.getPrivateKey(
↪'86e658cbc6da63120b79b5eec0c67d5dcfb6865a8f983eff08932477282b77bb')
'0x7b8190d96cd061a102e551ee36d08d4f3ca1f56fb0008ef5d70c56271d8c46d0'
```

Symmetric Keys

Shh.**newSymKey**(*self*)

- Generates a random symmetric key and stores it under id, which is then returned. Will be used in the future for session key exchange
- Returns the new key pair's identity

```
>>>web3.shh.newSymKey()
'6c388d63003deb378700c9dad87f67df0247e660647d6ba1d04321bbc2f6ce0c'
```

Shh.**addSymKey**(*self*, *key*)

- Stores the key, and returns its ID.
- Returns the new key pair's identity

```
>>>web3.shh.addSymKey(
↪'0x58f6556e56a0d41b464a083161377c8a9c2e95156921f954f99ef97d41cebaa2')
'6c388d63003deb378700c9dad87f67df0247e660647d6ba1d04321bbc2f6ce0c'
```

Shh.**generateSymKeyFromPassword**(*self*)

- Generates the key from password, stores it, and returns its ID.
- Returns the new key pair's identity

```
>>>web3.shh.generateSymKeyFromPassword('shh secret pwd')
'6c388d63003deb378700c9dad87f67df0247e660647d6ba1d04321bbc2f6ce0c'
```

Shh.**hasSymKey**(*self*, *id*)

- Checks if there is a symmetric key stored with the given ID.
- Returns `True` if the key exists, otherwise `False`

```
>>>web3.shh.hasSymKey(
↪'6c388d63003deb378700c9dad87f67df0247e660647d6ba1d04321bbc2f6ce0c')
False
```

Shh.**getSymKey**(*self*, *id*)

- Returns the symmetric key associated with the given ID.
- Returns the public key associated with the key pair

```
>>>web3.shh.getSymKey(
↪'6c388d63003deb378700c9dad87f67df0247e660647d6ba1d04321bbc2f6ce0c')
'0x58f6556e56a0d41b464a083161377c8a9c2e95156921f954f99ef97d41cebaa2'
```

Shh.**deleteSymKey**(*self*, *id*)

- Deletes the symmetric key associated with the given ID.
- Returns `True` if the key pair was deleted, otherwise `False`

```
>>>web3.shh.deleteSymKey(
↪'6c388d63003deb378700c9dad87f67df0247e660647d6ba1d04321bbc2f6ce0c')
True
```

1.17 Personal API

class web3.personal.**Personal**

The `web3.personal` object exposes methods to interact with the RPC APIs under the `personal_` namespace.

1.17.1 Properties

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

web3.personal.**listAccounts**

- Delegates to `personal_listAccounts` RPC Method

Returns the list of known accounts.

```
>>> web3.personal.listAccounts
['0xd3cda913deb6f67967b99d67acdfa1712c293601']
```

1.17.2 Methods

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

web3.personal.**importRawKey**(*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.personal.importRawKey(some_private_key, 'the-passphrase')
'0xd3cda913deb6f67967b99d67acdfa1712c293601'
```

web3.personal.**newAccount**(*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.personal.newAccount('the-passphrase')
'0xd3cda913deb6f67967b99d67acdfa1712c293601'
```

web3.personal.**lockAccount**(*self*, *account*)

- Delegates to `personal_lockAccount` RPC Method

Locks the given `account`.

```
>>> web3.personal.lockAccount('0xd3cda913deb6f67967b99d67acdfa1712c293601')
```

web3.personal.**unlockAccount**(*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.

```
>>> web3.personal.unlockAccount('0xd3cda913deb6f67967b99d67acdfa1712c293601',
↪'wrong-passphrase')
False
>>> web3.personal.unlockAccount('0xd3cda913deb6f67967b99d67acdfa1712c293601',
↪'the-passphrase')
True
```

web3.personal.**sendTransaction**(*self*, *transaction*, *passphrase*)

- Delegates to `personal_sendTransaction` RPC Method

Sends the transaction.

1.18 Net API

class web3.net.**Net**

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

1.18.1 Properties

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

Net.**chainId**(*self*)

This method is trivially implemented. It will always return *None*, which is a valid chainId to specify in the transaction.

If you want the real chainId of your node, you must manually determine it for now.

Note that your transactions (may be) replayable on forks of the network you intend, if *Working with Local Private Keys* and using a chainId of *None*.

```
>>> web3.net.chainId
None
```

Net.**version**(*self*)

- Delegates to `net_version` RPC Method

Returns the current network chainId/version.

```
>>> web3.net.version
1
```

1.19 Version API

class web3.version.**Version**

The `web3.version` object exposes methods to interact with the RPC APIs under the `version_` namespace.

1.19.1 Properties

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

`Version.`**`api`**(*self*)

Returns the current Web3 version.

```
>>> web3.version.api
"2.6.0"
```

`Version.`**`node`**(*self*)

- Delegates to `web3_clientVersion` RPC Method

Returns the current client version.

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

`Version.`**`network`**(*self*)

- Delegates to `net_version` RPC Method

Returns the current network protocol version.

```
>>> web3.version.network
1
```

`Version.`**`ethereum`**(*self*)

- Delegates to `eth_protocolVersion` RPC Method

Returns the current ethereum protocol version.

```
>>> web3.version.ethereum
63
```

1.20 TX Pool API

class web3.txpool.**TxPool**

The `web3.txpool` object exposes methods to interact with the RPC APIs under the `txpool_` namespace.

1.20.1 Properties

The following properties are available on the `web3.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.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"]
        },
        '0x9174e688d7de157c5c0583df424eaab2676ac162': {
          3: ["0xbb9bc244d798123fde783fcc1c72d3bb8c189413: 30000000000000000000␣
↪wei + 85000 × 21000000000 gas"]
        },
        '0xb18f9d01323e150096650ab989cfecd39d757aec': {
          777: ["0xcd79c72690750f079ae6ab6ccd7e7aedc03c7720: 0 wei + 1000000 ×␣
↪20000000000 gas"]
        },
        '0xb2916c870cf66967b6510b76c07e9d13a5d23514': {
          2: ["0x576f25199d60982a8f31a8dff4da8acb982e6aba: 26000000000000000000␣
↪wei + 90000 × 20000000000 gas"]
        },
        '0xbc0ca4f217e052753614d6b019948824d0d8688b': {
          0: ["0x2910543af39aba0cd09dbb2d50200b3e800a63d2: 10000000000000000000␣
↪wei + 50000 × 1171602790622 gas"]
        },
        '0xea674fdde714fd979de3edf0f56aa9716b898ec8': {
          70148: ["0xe39c55ead9f997f7fa20ebe40fb4649943d7db66:␣
↪1000767667434026200 wei + 90000 × 20000000000 gas"]
        }
      },
      'queued': {
        '0x0f6000de1578619320aba5e392706b131fb1de6f': {
          6: ["0x8383534d0bcd0186d326c993031311c0ac0d9b2d: 90000000000000000000␣
↪wei + 21000 × 20000000000 gas"]
        },
        '0x5b30608c678e1ac464a8994c3b33e5cdf3497112': {
          6: ["0x9773547e27f8303c87089dc42d9288aa2b9d8f06: 50000000000000000000␣
↪wei + 90000 × 50000000000 gas"]
```

(continues on next page)

(continued from previous page)

```
        },
        '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.txpool.content
{
  'pending': {
    '0x0216d5032f356960cd3749c31ab34eeff21b3395': {
      806: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x0216d5032f356960cd3749c31ab34eeff21b3395",
        'gas': "0x5208",
        'gasPrice': "0xba43b7400",
        'hash':
↪"0xaf953a2d01f55cfe080c0c94150a60105e8ac3d51153058a1f03dd239dd08586",
        'input': "0x",
        'nonce': "0x326",
        'to': "0x7f69a91a3cf4be60020fb58b893b7cbb65376db8",
        'transactionIndex': None,
        'value': "0x19a99f0cf456000"
      }]
    },
    '0x24d407e5a0b506e1cb2fae163100b5de01f5193c': {
      34: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
```

(continues on next page)

(continued from previous page)

```
        'blockNumber': None,
        'from': "0x24d407e5a0b506e1cb2fae163100b5de01f5193c",
        'gas': "0x44c72",
        'gasPrice': "0x4a817c800",
        'hash':
↪"0xb5b8b853af32226755a65ba0602f7ed0e8be2211516153b75e9ed640a7d359fe",
        'input':
↪"0xb61d27f60000000000000000000000024d407e5a0b506e1cb2fae163100b5de01f5193c0000000000000000000
↪",
        'nonce': "0x22",
        'to': "0x7320785200f74861b69c49e4ab32399a71b34f1a",
        'transactionIndex': None,
        'value': "0x0"
      }]
    }
  },
  'queued': {
    '0x976a3fc5d6f7d259ebfb4cc2ae75115475e9867c': {
      3: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x976a3fc5d6f7d259ebfb4cc2ae75115475e9867c",
        'gas': "0x15f90",
        'gasPrice': "0x4a817c800",
        'hash':
↪"0x57b30c59fc39a50e1cba90e3099286dfa5aaf60294a629240b5bbec6e2e66576",
        'input': "0x",
        'nonce': "0x3",
        'to': "0x346fb27de7e7370008f5da379f74dd49f5f2f80f",
        'transactionIndex': None,
        'value': "0x1f161421c8e0000"
      }]
    },
    '0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a': {
      2: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a",
        'gas': "0x15f90",
        'gasPrice': "0xba43b7400",
        'hash':
↪"0x3a3c0698552eec2455ed3190eac3996feccc806970a4a056106deaf6ceb1e5e3",
        'input': "0x",
        'nonce': "0x2",
        'to': "0x24a461f25ee6a318bdef7f33de634a67bb67ac9d",
        'transactionIndex': None,
        'value': "0xebec21ee1da40000"
      }],
      6: [{
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a",
        'gas': "0x15f90",
        'gasPrice': "0x4a817c800",
```

(continues on next page)

(continued from previous page)

```
        'hash':
↪"0xbbcd1e45eae3b859203a04be7d6e1d7b03b222ec1d66dfcc8011dd39794b147e",
        'input': "0x",
        'nonce': "0x6",
        'to': "0x6368f3f8c2b42435d6c136757382e4a59436a681",
        'transactionIndex': None,
        'value': "0xf9a951af55470000"
      }, {
        'blockHash':
↪"0x0000000000000000000000000000000000000000000000000000000000000000",
        'blockNumber': None,
        'from': "0x9b11bf0459b0c4b2f87f8cebca4cfc26f294b63a",
        'gas': "0x15f90",
        'gasPrice': "0x4a817c800",
        'hash':
↪"0x60803251d43f072904dc3a2d6a084701cd35b4985790baaf8a8f76696041b272",
        'input': "0x",
        'nonce': "0x6",
        'to': "0x8db7b4e0ecb095fbd01dffa62010801296a9ac78",
        'transactionIndex': None,
        'value': "0xebe866f5f0a06000"
      }],
    }
  }
}
```

1.21 Miner API

class web3.miner.**Miner**

The `web3.miner` object exposes methods to interact with the RPC APIs under the `miner_` namespace.

1.21.1 Properties

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

`Miner.`**`hashrate`**

- Delegates to `eth_hashrate` RPC Method

Returns the current number of hashes per second the node is mining with.

```
>>> web3.eth.hashrate
906
```

Note: This property is an alias to `web3.eth.hashrate`.

1.21.2 Methods

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

`Miner.`**`makeDAG`**(*number*)

- Delegates to `miner_makeDag` RPC Method

Generate the DAG for the given block number.

```
>>> web3.miner.makeDag(10000)
```

`Miner.`**`setExtra`**(*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.miner.setExtra('abcdefghijklmnopqrstuvwxyzABCDEF')
```

`Miner.`**`setGasPrice`**(*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.miner.setGasPrice(19999999999)
```

`Miner.`**`start`**(*num_threads*)

- Delegates to `miner_start` RPC Method

Start the CPU mining process using the given number of threads.

```
>>> web3.miner.start(2)
```

`Miner.`**`stop`**()

- Delegates to `miner_stop` RPC Method

Stop the CPU mining operation

```
>>> web3.miner.stop()
```

`Miner.`**`startAutoDAG`**()

- Delegates to `miner_startAutoDag` RPC Method

Enable automatic DAG generation.

```
>>> web3.miner.startAutoDAG()
```

`Miner.`**`stopAutoDAG`**()

- Delegates to `miner_stopAutoDag` RPC Method

Disable automatic DAG generation.

```
>>> web3.miner.stopAutoDAG()
```

1.22 Admin API

`class` `web3.admin.`**`Admin`**

The `web3.admin` object exposes methods to interact with the RPC APIs under the `admin_` namespace.

1.22.1 Properties

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

`web3.admin.`**`datadir`**

- Delegates to `admin_datadir` RPC Method

Returns the system path of the node's data directory.

```
>>> web3.admin.datadir
'/Users/piper/Library/Ethereum'
```

`web3.admin.`**`nodeInfo`**

- Delegates to `admin_nodeInfo` RPC Method

Returns information about the currently running node.

```
>>> web3.admin.nodeInfo
{
    '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.admin.`**`peers`**

- Delegates to `admin_peers` RPC Method

Returns the current peers the node is connected to.

```
>>> web3.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',
```

(continues on next page)

(continued from previous page)

```
        },
        '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,
            },
        },
    },
    ...
]
```

1.22.2 Methods

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

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

- Delegates to `admin_addPeer` RPC Method

Requests adding a new remote node to the list of tracked static nodes.

```
>>> web3.admin.addPeer('enode://
↪e54eebad24dce1f6d246bea455ffa756d97801582420b9ed681a2ea84bf376d0bd87ae8dd6dc06cdb862a2ca89ecab
↪71.255.237:30303')
True
```

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

- Delegates to `admin_setSolc` RPC Method

Sets the system path to the `solc` binary for use with the `eth_compileSolidity` RPC method. Returns the output reported by `solc --version`.

```
>>> web3.admin.setSolc('/usr/local/bin/solc')
"solc, the solidity compiler commandline interface\nVersion: 0.3.5-9da08ac3/
↪Release-Darwin/appleclang/JIT"
```

web3.admin.**startRPC**(*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.admin.startRPC()
True
```

web3.admin.**startWS**(*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.admin.startWS()
True
```

web3.admin.**stopRPC**()

- Delegates to admin_stopRPC RPC Method

Stops the HTTP based JSON RPC server.

```
>>> web3.admin.stopRPC()
True
```

web3.admin.**stopWS**()

- Delegates to admin_stopWS RPC Method

Stops the Websocket based JSON RPC server.

```
>>> web3.admin.stopWS()
True
```

1.23 Gas Price API

For Ethereum transactions, gas price is a delicate property. For this reason, Web3 includes an API for configuring it.

By default, Web3 will not include a gasPrice in the transaction as to relay this responsibility to the connected node. 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).

1.23.1 Retrieving gas price

To retreive the gas price using the selected strategy simply call *generateGasPrice()*

```
>>> Web3.eth.generateGasPrice()
20000000000
```

1.23.2 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')
```

1.23.3 Selecting the gas price strategy

The gas price strategy can be set by calling *setGasPriceStrategy()*.

```
from web3 import Web3

def value_based_gas_price_strategy(web3, transaction_params):
    ...

w3 = Web3(...)
w3.eth.setGasPriceStrategy(value_based_gas_price_strategy)
```

1.23.4 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*, *probability*)
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.

- `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.setGasPriceStrategy(medium_gas_price_strategy)

w3.middleware_stack.add(middleware.time_based_cache_middleware)
w3.middleware_stack.add(middleware.latest_block_based_cache_middleware)
w3.middleware_stack.add(middleware.simple_cache_middleware)
```

1.24 ENS API

Ethereum Name Service has a friendly overview.

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

1.24.1 ens.main module

class ens.main.**ENS**(*providers=<object object>*, *addr=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*)

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 will add '.eth' to name if no TLD given. Also, it 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*)
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*)
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*, *addr=None*)
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*, *guess_tld=True*)
Look up the Ethereum address that *name* currently points to.

Parameters

- **name** (*str*) – an ENS name to look up
- **guess_tld** (*bool*) – should *name* be appended with '.eth' if no common TLD found?

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

name(*address*)
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*) –

reverse(*address*)
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*, *address=<object object>*, *transact={}*)
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, in checksum format
- **address** (*str*) – name will point to this address. If `None`, erase the record. If not specified, name will point to the owner's address.
- **transact** (*dict*) – the transaction configuration, like in *sendTransaction()*

Raises

- ***InvalidName*** – if `name` has invalid syntax
- ***UnauthorizedError*** – if `'from'` in *transact* does not own *name*

setup_name(*name*, *address=None*, *transact={}*)

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 `sendTransaction()`

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*)

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*, *new_owner=<object object>*, *transact={}*)

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 *sendTransaction()*

Raises

- ***InvalidName*** – if *name* has invalid syntax
- ***UnauthorizedError*** – if `'from'` in *transact* does not own *name*

Returns the new owner's address

1.24.2 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.

1.25 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.

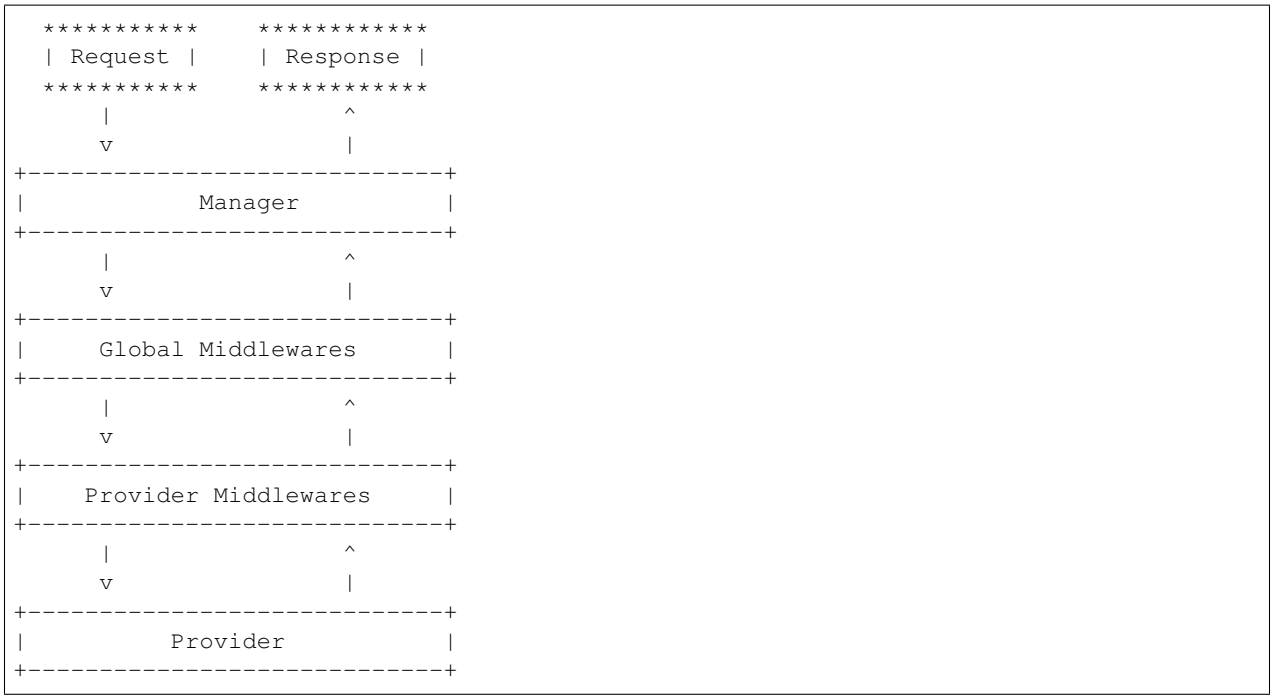
- **Providers** are responsible for the actual communication with the blockchain such as sending JSON-RPC requests over HTTP or an IPC socket.
- **Middlewares** provide hooks for monitoring and modifying requests and responses to and from the provider. These can be *global* operating on all providers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

1.25.1 Request Lifecycle

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



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.

1.25.2 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.

In most simple cases you will be using a single provider. However, if you would like to use Web3 with multiple providers, you can simply pass them in as a list when instantiating your `Web3` object.

```
>>> web3 = Web3([provider_a, provider_b])
```

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.

If a provider is unable to respond to certain RPC calls it should raise the `web3.exceptions.CannotHandleRequest` exception. When this happens, the request is issued to the next configured provider. If no providers are able to handle the request then a `web3.exceptions.UnhandledRequest` error will be raised.

`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.

1.25.3 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, web3):
    # 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)
```

(continues on next page)

(continued from previous page)

```
        # 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, web3):
        self.web3 = web3
        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_stack` object to manage middlewares. It is also exposed on the `Web3` object for convenience. That API is detailed in *Configuring Middleware*.

1.25.4 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.

1.26 Conventions

The Web3 library follows the following conventions.

1.26.1 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.

1.26.2 Hexadecimal Representations

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

1.26.3 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.

1.27 Release Notes

1.27.1 v4.8.2

Released November 15, 2018

- Misc
  - Reduce unneeded memory usage - #1138

1.27.2 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 and python version - #1082
  - solidity Sha3 docs fixup - #1100
  - Update ropsten faucet links in troubleshooting docs

1.27.3 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

1.27.4 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.

1.27.5 v4.7.0

Released September 10th, 2018

- Features
  - Add traceFilter method to the parity module. - #1051
  - Move `web3.utils.datastructures` to public namespace `web3.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.

1.27.6 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

1.27.7 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

1.27.8 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*

1.27.9 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

1.27.10 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`

1.27.11 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

1.27.12 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

1.27.13 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

1.27.14 v4.0.0

Released Apr 2, 2018

- Marked beta.13 as stable
- Documentation tweaks

1.27.15 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

1.27.16 v4.0.0-beta.12

A little hiccup on release. Skipped.

1.27.17 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

1.27.18 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

1.27.19 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`

1.27.20 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)

1.27.21 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

1.27.22 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

1.27.23 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

1.27.24 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.

1.27.25 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

1.27.26 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.

1.27.27 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: *Type Conversions*
- 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

1.27.28 3.16.1

- Addition of ethereum-tester as a dependency

1.27.29 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.

1.27.30 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.

1.27.31 3.14.2

- Fix PyPi readme text.

1.27.32 3.14.1

- Fix PyPi readme text.

1.27.33 3.14.0

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

1.27.34 3.13.5

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

1.27.35 3.13.4

- Bugfix for formatter handling of contract creation transaction.

1.27.36 3.13.3

- Improved testing infrastructure.

1.27.37 3.13.2

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

1.27.38 3.13.1

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

1.27.39 3.13.0

- New Middleware API
- Support for multiple providers
- New `web3.solidity_sha3`
- 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`.

1.27.40 3.12.0

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

1.27.41 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`

1.27.42 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

1.27.43 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.

1.27.44 3.8.1

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

1.27.45 3.8.0

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

1.27.46 3.7.1

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

1.27.47 3.7.0

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

1.27.48 3.6.2

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

1.27.49 3.6.1

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

1.27.50 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.

1.27.51 3.5.3

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

1.27.52 3.5.2

- Bugfix for construction of `request_kwargs` within HTTPProvider

1.27.53 3.5.1

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

1.27.54 3.5.0

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

1.27.55 3.4.4

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

1.27.56 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`

1.27.57 3.4.2

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

1.27.58 3.4.1

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

1.27.59 3.4.0

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

1.27.60 3.3.0

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

1.27.61 3.2.0

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

1.27.62 3.1.1

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

1.27.63 3.1.0

- New DelegatedSigningManager and PrivateKeySigningManager classes.

1.27.64 3.0.2

- Bugfix or IPCProvider not handling large JSON responses well.

1.27.65 3.0.1

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

1.27.66 3.0.0

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

1.27.67 2.9.0

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

1.27.68 2.8.1

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

1.27.69 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.

1.27.70 2.7.1

- Bugfix: Fix KeyError in merge_args_and_kwargs helper fn.

1.27.71 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 `/`.

1.27.72 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`

1.27.73 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.

1.27.74 2.4.0

- Contract functions can now be called using keyword arguments.

1.27.75 2.3.0

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

1.27.76 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`.

1.27.77 2.1.1

- Cast RPCProvider port to an integer.

1.27.78 2.1.0

- Remove all monkeypatching

1.27.79 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.

1.27.80 1.9.0

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

1.27.81 1.8.0

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

1.27.82 1.7.1

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

1.27.83 1.7.0

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

1.27.84 1.6.0

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

1.27.85 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.

1.27.86 1.4.0

- Bugfix to allow address types in constructor arguments.

1.27.87 1.3.0

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

1.27.88 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.

1.27.89 1.1.0

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

1.27.90 1.0.1

- Specify missing `pyrlp` and `gevent` dependencies

1.27.91 1.0.0

- Massive refactor to the majority of the app.

1.27.92 0.1.0

- Initial release

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

w

Index

F

G

H

I

L

M

N