PyOTA Documentation

Phoenix Zerin

Oct 21, 2019

Contents

PyOTA is compatible with Python 3.7, 3.6, 3.5 and 2.7.

Install PyOTA using *pip*:

```
pip install pyota[ccurl,pow]
```

Note: The `[ccurl]` extra installs the optional PyOTA-CCurl extension.

This extension boosts the performance of certain crypto operations significantly (speedups of 60x are common).

Note: The `[pow]` extra installs the optional PyOTA-PoW extension.

This extension makes it possible to perform proof-of-work (api call `attach_to_tangle`) locally, without relying on an iota node. Use the `local_pow` parameter at api instantiation:

```
api = Iota('https://nodes.thetangle.org:443', local_pow=True)
```

Or the `set_local_pow` method of the api class to dynamically enable/disable the local proof-of-work feature.

CHAPTER 1

Getting Started

In order to interact with the IOTA network, you will need access to a node.

You can:

- Run your own node.
- Use a light wallet node.
- Use the sandbox node.

Note that light wallet nodes often disable certain features like PoW for security reasons.

Once you've gotten access to an IOTA node, initialize an `iota.Iota` object with the URI of the node, and optional seed:

```
from iota import Iota

# Generate a random seed.
api = Iota('http://localhost:14265')

# Specify seed.
api = Iota('http://localhost:14265', 'SEED9GOES9HERE')
```

Test your connection to the server by sending a `getNodeInfo` command:

```
print(api.get_node_info())
```

You are now ready to send commands to your IOTA node!

1.1 Using the Sandbox Node

To connect to the sandbox node, you will need to inject a `SandboxAdapter` into your `Iota` object. This will modify your API requests so that they contain the necessary authentication metadata.

```
from iota.adapter.sandbox import SandboxAdapter

api = Iota(
  # To use sandbox mode, inject a ``SandboxAdapter``.
  adapter = SandboxAdapter(
    # URI of the sandbox node.
    uri = 'https://sandbox.iotatoken.com/api/v1/',

    # Access token used to authenticate requests.
    # Contact the node maintainer to get an access token.
    auth_token = 'auth token goes here',
  ),

  # Seed used for cryptographic functions.
  # If null, a random seed will be generated.
  seed = b'SEED9GOES9HERE',
)
```

CHAPTER 2

Basic Concepts

Before diving into the API, it's important to understand the fundamental data types of IOTA.

todo Link to IOTA docs

CHAPTER 3

PyOTA Types

PyOTA defines a few types that will make it easy for you to model objects like Transactions and Bundles in your own code.

3.1 TryteString

```
from iota import TryteString

trytes_1 = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')
trytes_2 = TryteString(b'LH9GYEMHCF9GWHZFEELHVFOEOHNEEEWHZFUD')

if trytes_1 != trytes_2:
  trytes_combined = trytes_1 + trytes_2

index = {
  trytes_1: 42,
  trytes_2: 86,
}
```

A `TryteString` is an ASCII representation of a sequence of trytes. In many respects, it is similar to a Python `bytes` object (which is an ASCII representation of a sequence of bytes).

In fact, the two objects behave very similarly; they support concatenation, comparison, can be used as dict keys, etc.

However, unlike `bytes`, a `TryteString` can only contain uppercase letters and the number 9 (as a regular expression: `^[A-Z9]*$`).

As you go through the API documentation, you will see many references to `TryteString` and its subclasses:

- `Fragment`: A signature or message fragment inside a transaction. Fragments are always 2187 trytes long.
- `Hash`: An object identifier. Hashes are always 81 trytes long. There are many different types of hashes:
- `Address`: Identifies an address on the Tangle.
- `BundleHash`: Identifies a bundle on the Tangle.

- `TransactionHash`: Identifies a transaction on the Tangle.
- `Seed`: A TryteString that is used for crypto functions such as generating addresses, signing inputs, etc. Seeds can be any length, but 81 trytes offers the best security.
- `Tag`: A tag used to classify a transaction. Tags are always 27 trytes long.
- `TransactionTrytes`: A TryteString representation of a transaction on the Tangle. `TransactionTrytes` are always 2673 trytes long.

3.1.1 Encoding

```
from iota import TryteString

message_trytes = TryteString.from_unicode('Hello, IOTA!')
```

To encode character data into trytes, use the `TryteString.from_unicode` method.

You can also convert a tryte sequence into characters using `TryteString.decode`. Note that not every tryte sequence can be converted; garbage in, garbage out!

```
from iota import TryteString

trytes = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')
message = trytes.decode()
```

Note: PyOTA also supports encoding non-ASCII characters, but this functionality is **experimental** and has not yet been evaluated by the IOTA community!

Until this feature has been standardized, it is recommended that you only use ASCII characters when generating `TryteString` objects from character strings.

3.2 Transaction Types

PyOTA defines two different types used to represent transactions:

3.2.1 Transaction

```
from iota import Address, ProposedTransaction, Tag, Transaction

txn_1 =\
  Transaction.from_tryte_string(
    b'GYPRVHBEZOOFXSHQBLCYW9ICTCISLHDBNMMVYD9JJHQMPQCTIQAQTJNNNJ9IDXLRCC'
    b'OYOXYPCLR9PBEY9ORZIEPPDNTI9CQWYZUOTAVBXPSBOFEQAPFLWXSWUIUSJMSJIIIZ'
    b'WIKIRH9GCOEVZFKNXEVCUCIIWZQCQEUVRZOCMEL9AMGXJNMLJCIA9UWGRPPHCEOPTS'
    b'VPKPPPCMQXYBHMSODTWUOABPKWFFFQJHCBVYXLHEWPD9YUDFTGNCYAKQKVEZYRBQRB'
    b'XIAUX9SVEDUKGMTWQIYXRGSWYRK9SRONVGTW9YGHSZRIXWGPCCUCDRMAXBPDFVHSRY'
    b'WHGB9DQSQFQKSNICGPIPTRZINYRXQAFSWSEWIFRMSBMGTNYPRWFSOIIWWT9IDSELM9'
    b'JUOOWFNCCSHUSMGNROBFJX9JQ9XT9PKEGQYQAWAFPRVRRVQPUQBHLSNTEFCDKBWRCD'
    b'X9EYOBB9KPMTLNNQLADBDLZPRVBCKVCYQEOLARJYAGTBFR9QLPKZBOYWZQOVKCVYRG'
    b'YI9ZEFIQRKYXLJBZJDBJDJVQZCGYQMROVHNDBLGNLQODPUXFNTADDVYNZJUVPGB9LV'
    b'PJIYLAPBOEHPMRWUIAJXVQOEM9ROEYUOTNLXVVQEYRQWDTQGDLEYFIYNDPRAIXOZEB'
```

(continues on next page)

(continued from previous page)

```
    b'CS9P99AZTQQLKEILEVXMSHBIDHLXKUOMMNFKPYHONKEYDCHMUNTTNRYVMMEYHPGASP'
    b'ZXASKRUPWQSHDMU9VPS99ZZ9SJJYFUJFFMFORBYDILBXCAVJDPDFHTTTIYOVGLRDYR'
    b'TKHXJORJVYRPTDH9ZCPZ9ZADXZFRSFPIQKWLBRNTWJHXTOAUOL9FVGTUMMPYGYICJD'
    b'XMOESEVDJWLMCVTJLPIEKBE9JTHDQWV9MRMEWFLPWGJFLUXI9BXPSVWCMUWLZSEWHB'
    b'DZKXOLYNOZAPOYLQVZAQMOHGTTQEUAOVKVRRGAHNGPUEKHFVPVCOYSJAWHZU9DRROH'
    b'BETBAFTATVAUGOEGCAYUXACLSSHHVYDHMDGJP9AUCLWLNTFEVGQGHQXSKEMVOVSKQE'
    b'EWHWZUDTYOBGCURRZSJZLFVQQAAYQO9TRLFFN9HTDQXBSPPJYXMNGLLBHOMNVXNOWE'
    b'IDMJVCLLDFHBDONQJCJVLBLCSMDOUQCKKCQJMGTSTHBXPXAMLMSXRIPUBMBAWBFNLH'
    b'LUJTRJLDERLZFUBUSMF999XNHLEEXEENQJNOFFPNPQ9PQICHSATPLZVMVIWLRTKYPI'
    b'XNFGYWOJSQDAXGFHKZPFLPXQEHCYEAGTIWIJEZTAVLNUMAFWGGLXMBNUQTOFCNLJTC'
    b'DMWVVZGVBSEBCPFSM99FLOIDTCLUGPSEDLOKZUAEVBLWNMODGZBWOVQT9DPFOTSKRA'
    b'BQAVOQ9RXWBMAKFYNDCZOJGTCIDMQSQQSODKDXTPFLNOKSIZEOY9HFUTLQRXQMEPGO'
    b'XQGLLPNSXAUCYPGZMNWMQWSWCKAQYKXJTWINSGPPZG9HLDLEAWUWEVCTVRCBDFOXKU'
    b'ROXH9HXXAXVPEJFRSLOGRVGYZASTEBAQNXJJROCYRTDPYFUIQJVDHAKEG9YACV9HCP'
    b'JUEUKOYFNWDXCCJBIFQKYOXGRDHVTHEQUMHO99999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999999999999999999999999999999999999'
    b'999999999999RKWEEVD99A99999999A99999999NFDPEEZCWVYLKZGSLCQNOFUSENI'
    b'XRHWWTZFBXMPSQHEDFWZULBZFEOMNLRNIDQKDNNIELAOXOVMYEI9PGTKORV9IKTJZQ'
    b'UBQAWTKBKZ9NEZHBFIMCLV9TTNJNQZUIJDFPTTCTKBJRHAITVSKUCUEMD9M9SQJ999'
    b'999TKORV9IKTJZQUBQAWTKBKZ9NEZHBFIMCLV9TTNJNQZUIJDFPTTCTKBJRHAITVSK'
    b'UCUEMD9M9SQJ99999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999'
)
```

`Transaction` is a transaction that has been loaded from the Tangle.

Generally, you will never need to create `Transaction` objects; the API will build them for you, as the result of various API methods.

Each `Transaction` has the following attributes:

- `address`: `Address`: The address associated with this transaction. Depending on the transaction's `value`, this address may be a sender or a recipient.
- `attachment_timestamp`: `int`: Estimated epoch time of the attachment to the tangle.
- `attachment_time_lower_bound`: `int`: The lowest possible epoch time of the attachment to the tangle.
- `attachment_time_upper_bound`: `int`: The highest possible epoch time of the attachment to the tangle.
- `branch_transaction_hash`: `TransactionHash`: An unrelated transaction that this transaction "approves". Refer to the Basic Concepts section for more information.
- `bundle_hash`: `BundleHash`: The bundle hash, used to identify transactions that are part of the same bundle. This value is generated by taking a hash of the metadata from all transactions in the bundle.
- `current_index`: `int`: The transaction's position in the bundle.
- If the `current_index` value is 0, then this is the "tail transaction".

- If it is equal to `last_index`, then this is the "head transaction".
- `hash: TransactionHash`: The transaction hash, used to uniquely identify the transaction on the Tangle. This value is generated by taking a hash of the raw transaction trits.
- `is_confirmed: Optional[bool]`: Whether this transaction has been "confirmed". Refer to the Basic Concepts section for more information.
- `last_index: int`: The index of the final transaction in the bundle. This value is attached to every transaction to make it easier to traverse and verify bundles.
- `legacy_tag: Tag`: A short message attached to the transaction. Deprecated, use `tag` instead.
- `nonce: Hash`: This is the product of the PoW process.
- `signature_message_fragment: Fragment`: Additional data attached to the transaction:
- If `value < 0`, this value contains a fragment of the cryptographic signature authorizing the spending of the IOTAs.
- If `value > 0`, this value is an (optional) string message attached to the transaction.
- If `value = 0`, this value could be either a signature or message fragment, depending on the previous transaction.
- `tag: Tag`: Used to classify the transaction. Many transactions have empty tags (`Tag(b'999999999999999999999999999')`).
- `timestamp: int`: Unix timestamp when the transaction was created. Note that devices can specify any timestamp when creating transactions, so this value is not safe to use for security measures (such as resolving double-spends).
- `trunk_transaction_hash: TransactionHash`: The transaction hash of the next transaction in the bundle. If this transaction is the head transaction, its `trunk_transaction_hash` will be pseudo-randomly selected, similarly to `branch_transaction_hash`.
- `value: int`: The number of IOTAs being transferred in this transaction:
- If this value is negative, then the `address` is spending IOTAs.
- If it is positive, then the `address` is receiving IOTAs.
- If it is zero, then this transaction is being used to carry metadata (such as a signature fragment or a message) instead of transferring IOTAs.

3.2.2 ProposedTransaction

`ProposedTransaction` is a transaction that was created locally and hasn't been broadcast yet.

```
txn_2 =\
  ProposedTransaction(
    address =
      Address(
        b'TESTVALUE9DONTUSEINPRODUCTION99999XE9IVG'
        b'EFNDOCQCMERGUATCIEGGOHPHGFIAQEZGNHQ9W99CH'
      ),

    message = TryteString.from_unicode('thx fur cheezburgers'),
    tag     = Tag(b'KITTEHS'),
    value   = 42,
  )
```

This type is useful when creating new transactions to broadcast to the Tangle. Note that creating a `ProposedTransaction` requires only a small subset of the attributes needed to create a `Transaction` object.

To create a `ProposedTransaction`, specify the following values:

- `address: Address`: The address associated with the transaction. Note that each transaction references exactly one address; in order to transfer IOTAs from one address to another, you must create at least two transactions: One to deduct the IOTAs from the sender's balance, and one to add the IOTAs to the recipient's balance.
- `message: Optional[TryteString]`: Optional trytes to attach to the transaction. This could be any value (character strings, binary data, or raw trytes), as long as it's converted to a `TryteString` first.
- `tag: Optional[Tag]`: Optional tag to classify this transaction. Each transaction may have exactly one tag, and the tag is limited to 27 trytes.
- `value: int`: The number of IOTAs being transferred in this transaction. This value can be 0; for example, to send a message without spending any IOTAs.

3.3 Bundle Types

As with transactions, PyOTA defines two bundle types.

3.3.1 Bundle

```
from iota import Bundle

bundle = Bundle.from_tryte_strings([
  b'GYPRVHBEZOOFXSHQBLCYW9ICTCISLHDBNMMVYD9JJHQMPQCTIQAQTJNNNJ9IDXLRCC...',
  b'OYOXYPCLR9PBEY9ORZIEPPDNTI9CQWYZUOTAVBXPSBOFEQAPFLWXSWUIUSJMSJIIIZ...',
  # etc.
])
```

`Bundle` represents a bundle of transactions published on the Tangle. It is intended to be a read-only object, allowing you to inspect the transactions and bundle metadata.

Each bundle has the following attributes:

- `hash: BundleHash`: The hash of this bundle. This value is generated by taking a hash of the metadata from all transactions in the bundle.
- `is_confirmed: Optional[bool]`: Whether the transactions in this bundle have been confirmed. Refer to the Basic Concepts section for more information.
- `tail_transaction: Optional[Transaction]`: The bundle's tail transaction.
- `transactions: List[Transaction]`: The transactions associated with this bundle.

3.3.2 ProposedBundle

```
from iota import Address, ProposedBundle, ProposedTransaction
from iota.crypto.signing import KeyGenerator

bundle = ProposedBundle()
```

(continues on next page)

(continued from previous page)

```
bundle.add_transaction(ProposedTransaction(...))
bundle.add_transaction(ProposedTransaction(...))
bundle.add_transaction(ProposedTransaction(...))

bundle.add_inputs([
  Address(
    address =
      b'TESTVALUE9DONTUSEINPRODUCTION99999HAA9UA'
      b'MHCGKEUGYFUBIARAXBFASGLCHCBEVGTBDCSAEBTBM',

    balance   = 86,
    key_index = 0,
  ),
])

bundle.send_unspent_inputs_to(
  Address(
    b'TESTVALUE9DONTUSEINPRODUCTION99999D99HEA'
    b'M9XADCPFJDFANCIHR9OBDHTAGGE9TGCI9EO9ZCRBN'
  ),
)

bundle.finalize()
bundle.sign_inputs(KeyGenerator(b'SEED9GOES9HERE'))
```

Note: This section contains information about how PyOTA works "under the hood".

The `prepare_transfer` API method encapsulates this functionality for you; it is not necessary to understand how `ProposedBundle` works in order to use PyOTA.

`ProposedBundle` provides a convenient interface for creating new bundles, listed in the order that they should be invoked:

- `add_transaction: (ProposedTransaction) -> None`: Adds a transaction to the bundle. If necessary, it may split the transaction into multiple (for example, if the transaction's message is too long to fit into 2187 trytes).
- `add_inputs: (List[Address]) -> None`: Specifies inputs that can be used to fund transactions that spend IOTAs. The `ProposedBundle` will use these to create the necessary input transactions.
- You can use the `get_inputs` API command to find suitable inputs.
- `send_unspent_inputs_to: (Address) -> None`: Specifies the address that will receive unspent IOTAs. The `ProposedBundle` will use this to create the necessary change transaction, if necessary.
- `finalize: () -> None`: Prepares the bundle for PoW. Once this method is invoked, no new transactions may be added to the bundle.
- `sign_inputs: (KeyGenerator) -> None`: Generates the necessary cryptographic signatures to authorize spending the inputs. You do not need to invoke this method if the bundle does not contain any transactions that spend IOTAs.

Once the `ProposedBundle` has been finalized (and inputs signed, if necessary), invoke its `as_tryte_strings` method to generate the raw trytes that should be included in an `attach_to_tangle` API request.

CHAPTER 4

Adapters and Wrappers

The `Iota` class defines the API methods that are available for interacting with the node, but it delegates the actual interaction to another set of classes: Adapters and Wrappers.

4.1 AdapterSpec

In a few places in the PyOTA codebase, you may see references to a meta-type called `AdapterSpec`.

`AdapterSpec` is a placeholder that means "URI or adapter instance".

For example, the first argument of `Iota.__init__` is an `AdapterSpec`. This means that you can initialize an `Iota` object using either a node URI, or an adapter instance:

- Node URI: `Iota('http://localhost:14265')`
- Adapter instance: `Iota(HttpAdapter('http://localhost:14265'))`

4.2 Adapters

Adapters are responsible for sending requests to the node and returning the response.

PyOTA ships with a few adapters:

4.2.1 HttpAdapter

```
from iota import Iota
from iota.adapter import HttpAdapter

# Use HTTP:
api = Iota('http://localhost:14265')
api = Iota(HttpAdapter('http://localhost:14265'))
```

(continues on next page)

(continued from previous page)

```
# Use HTTPS:
api = Iota('https://service.iotasupport.com:14265')
api = Iota(HttpAdapter('https://service.iotasupport.com:14265'))

# Use HTTPS with basic authentication and 60 seconds timeout:
api = Iota(
    HttpAdapter(
        'https://service.iotasupport.com:14265',
        authentication=('myusername', 'mypassword'),
        timeout=60))
```

`HttpAdapter` uses the HTTP protocol to send requests to the node.

To configure an `Iota` instance to use `HttpAdapter`, specify an `http://` or `https://` URI, or provide an `HttpAdapter` instance.

The `HttpAdapter` raises a `BadApiResponse` exception if the server sends back an error response (due to invalid request parameters, for example).

Debugging HTTP Requests

```
from logging import getLogger

from iota import Iota

api = Iota('http://localhost:14265')
api.adapter.set_logger(getLogger(__name__))
```

To see all HTTP requests and responses as they happen, attach a `logging.Logger` instance to the adapter via its `set_logger` method.

Any time the `HttpAdapter` sends a request or receives a response, it will first generate a log message. Note: if the response is an error response (e.g., due to invalid request parameters), the `HttpAdapter` will log the request before raising `BadApiResponse`.

Note: `HttpAdapter` generates log messages with `DEBUG` level, so make sure that your logger's `level` attribute is set low enough that it doesn't filter these messages!

4.2.2 SandboxAdapter

```
from iota import Iota
from iota.adapter.sandbox import SandboxAdapter

api =\
  Iota(
    SandboxAdapter(
      uri = 'https://sandbox.iotatoken.com/api/v1/',
      auth_token = 'demo7982-be4a-4afa-830e-7859929d892c',
    ),
  )
```

The `SandboxAdapter` is a specialized `HttpAdapter` that sends authenticated requests to sandbox nodes.

Note: See Sandbox Documentation for more information about sandbox nodes.

Sandbox nodes process certain commands asynchronously. When `SandboxAdapter` determines that a request is processed asynchronously, it will block, then poll the node periodically until it receives a response.

The result is that `SandboxAdapter` abstracts away the sandbox node's asynchronous functionality so that your API client behaves exactly the same as if it were connecting to a non-sandbox node.

To create a `SandboxAdapter`, you must provide the URI of the sandbox node and the auth token that you received from the node maintainer. Note that `SandboxAdapter` only works with `http://` and `https://` URIs.

You may also specify the polling interval (defaults to 15 seconds) and the number of polls before giving up on an asynchronous job (defaults to 8 times).

Note: For parity with the other adapters, `SandboxAdapter` blocks until it receives a response from the node.

If you do not want `SandboxAdapter` to block the main thread, it is recommended that you execute it in a separate thread or process.

4.2.3 MockAdapter

```
from iota import Iota
from iota.adapter import MockAdapter

# Inject a mock adapter.
api = Iota('mock://')
api = Iota(MockAdapter())

# Seed responses from the node.
api.adapter.seed_response('getNodeInfo', {'message': 'Hello, world!'})
api.adapter.seed_response('getNodeInfo', {'message': 'Hello, IOTA!'})

# Invoke API commands, using the adapter.
print(api.get_node_info()) # {'message': 'Hello, world!'}
print(api.get_node_info()) # {'message': 'Hello, IOTA!'}
print(api.get_node_info()) # raises BadApiResponse exception
```

`MockAdapter` is used to simulate the behavior of an adapter without actually sending any requests to the node.

This is particularly useful in unit and functional tests where you want to verify that your code works correctly in specific scenarios, without having to engineer your own subtangle.

To configure an `Iota` instance to use `MockAdapter`, specify `mock://` as the node URI, or provide a `MockAdapter` instance.

To use `MockAdapter`, you must first seed the responses that you want it to return by calling its `seed_response` method.

`seed_response` takes two parameters:

- `command`: `Text`: The name of the command. Note that this is the camelCase version of the command name (e.g., `getNodeInfo`, not `get_node_info`).
- `response`: `dict`: The response that the adapter will return.

You can seed multiple responses for the same command; the `MockAdapter` maintains a queue for each command internally, and it will pop a response off of the corresponding queue each time it processes a request.

Note that you have to call `seed_response` once for each request you expect it to process. If `MockAdapter` does not have a seeded response for a particular command, it will raise a `BadApiResponse` exception (simulates a 404 response).

4.3 Wrappers

Wrappers act like decorators for adapters; they are used to enhance or otherwise modify the behavior of adapters.

4.3.1 RoutingWrapper

```
from iota import Iota
from iota.adapter.wrappers import RoutingWrapper

api =\
  Iota(
    # Send PoW requests to local node.
    # All other requests go to light wallet node.
    RoutingWrapper('https://service.iotasupport.com:14265')
      .add_route('attachToTangle', 'http://localhost:14265')
      .add_route('interruptAttachingToTangle', 'http://localhost:14265')
  )
```

`RoutingWrapper` allows you to route API requests to different nodes depending on the command name.

For example, you could use this wrapper to direct all PoW requests to a local node, while sending the other requests to a light wallet node.

Note: A common use case for `RoutingWrapper` is to perform proof-of-work on a specific (local) node, but let all other requests go to another node. Take care when you use `RoutingWrapper` adapter and `local_pow` parameter together in an API instance, because the behavior might not be obvious.

`local_pow` tells the API to perform proof-of-work (`attach_to_tangle`) without relying on an actual node. It does this by calling an extension package PyOTA-PoW that does the job. In PyOTA, this means the request doesn't reach the adapter, it is redirected before. As a consequence, `local_pow` has precedence over the route that is defined in `RoutingWrapper`.

`RoutingWrapper` must be initialized with a default URI/adapter. This is the adapter that will be used for any command that doesn't have a route associated with it.

Once you've initialized the `RoutingWrapper`, invoke its `add_route` method to specify a different adapter to use for a particular command.

`add_route` requires two arguments:

- `command: Text`: The name of the command. Note that this is the camelCase version of the command name (e.g., `getNodeInfo`, not `get_node_info`).
- `adapter: AdapterSpec`: The adapter or URI to send this request to.

CHAPTER 5

Generating Addresses

In IOTA, addresses are generated deterministically from seeds. This ensures that your account can be accessed from any location, as long as you have the seed.

Note that this also means that anyone with access to your seed can spend your IOTAs! Treat your seed(s) the same as you would the password for any other financial service.

Note: PyOTA's crytpo functionality is currently very slow; on average it takes 8-10 seconds to generate each address.

These performance issues will be fixed in a future version of the library; please bear with us!

In the meantime, if you are using Python 3, you can install a C extension that boosts PyOTA's performance significantly (speedups of 60x are common!).

To install the extension, run `pip install pyota[ccurl]`.

Important: The extension is not yet compatible with Python 2.

If you are familiar with Python 2's C API, we'd love to hear from you! Check the GitHub issue for more information.

PyOTA provides two methods for generating addresses:

5.1 Using the API

```
from iota import Iota

api = Iota('http://localhost:14265', b'SEED9GOES9HERE')

# Generate 5 addresses, starting with index 0.
gna_result = api.get_new_addresses(count=5)
addresses = gna_result['addresses']
```

(continues on next page)

(continued from previous page)

```
# Generate 1 address, starting with index 42:
gna_result = api.get_new_addresses(index=42)
addresses = gna_result['addresses']

# Find the first unused address, starting with index 86:
gna_result = api.get_new_addresses(index=86, count=None)
addresses = gna_result['addresses']
```

To generate addresses using the API, invoke its `get_new_addresses` method, using the following parameters:

- `index: int`: The starting index (defaults to 0). This can be used to skip over addresses that have already been generated.
- `count: Optional[int]`: The number of addresses to generate (defaults to 1).
- If `None`, the API will generate addresses until it finds one that has not been used (has no transactions associated with it on the Tangle). It will then return the unused address and discard the rest.
- `security_level: int`: Determines the security level of the generated addresses. See *Security Levels* below.

`get_new_addresses` returns a dict with the following items:

- `addresses: List[Address]`: The generated address(es). Note that this value is always a list, even if only one address was generated.

5.2 Using AddressGenerator

```
from iota.crypto.addresses import AddressGenerator

generator = AddressGenerator(b'SEED9GOES9HERE')

# Generate a list of addresses:
addresses = generator.get_addresses(start=0, count=5)

# Generate a list of addresses in reverse order:
addresses = generator.get_addresses(start=42, count=10, step=-1)

# Create an iterator, advancing 5 indices each iteration.
iterator = generator.create_iterator(start=86, step=5)
for address in iterator:
  ...
```

If you want more control over how addresses are generated, you can use the `AddressGenerator` class.

`AddressGenerator` can create iterators, allowing your application to generate addresses as needed, instead of having to generate lots of addresses up front.

You can also specify an optional `step` parameter, which allows you to skip over multiple addresses between iterations... or even iterate over addresses in reverse order!

`AddressGenerator` provides two methods:

- `get_addresses: (int, int, int) -> List[Address]`: Returns a list of addresses. This is the same method that the `get_new_addresses` API command uses internally.
- `create_iterator: (int, int) -> Generator[Address]`: Returns an iterator that will create addresses endlessly. Use this if you have a feature that needs to generate addresses "on demand".

CHAPTER 6

Security Levels

```
gna_result = api.get_new_addresses(security_level=3)

generator =\
  AddressGenerator(
    seed = b'SEED9GOES9HERE',
    security_level = 3,
  )
```

If desired, you may change the number of iterations that `AddressGenerator` uses internally when generating new addresses, by specifying a different `security_level` when creating a new instance.

`security_level` should be between 1 and 3, inclusive. Values outside this range are not supported by the IOTA protocol.

Use the following guide when deciding which security level to use:

- `security_level=1`: Least secure, but generates addresses the fastest.
- `security_level=2`: Default; good compromise between speed and security.
- `security_level=3`: Most secure; results in longer signatures in transactions.

CHAPTER 7

Core API

The Core API includes all of the core API calls that are made available by the current IOTA Reference Implementation.

These methods are "low level" and generally do not need to be called directly.

For the full documentation of all the Core API calls, please refer to the official documentation.

CHAPTER 8

Extended API

The Extended API includes a number of "high level" commands to perform tasks such as sending and receiving transfers.

8.1 broadcast_and_store

Broadcasts and stores a set of transaction trytes.

8.1.1 Parameters

- `trytes: Iterable[TransactionTrytes]`: Transaction trytes.

8.1.2 Return

This method returns a `dict` with the following items:

- `trytes: List[TransactionTrytes]`: Transaction trytes that were broadcast/stored. Should be the same as the value of the `trytes` parameter.

8.2 find_transaction_objects

A more extensive version of the core API `find_transactions` that returns transaction objects instead of hashes.

Effectively, this is `find_transactions` + `get_trytes` + converting the trytes into transaction objects. It accepts the same parameters as `find_transactions`

Find the transactions which match the specified input. All input values are lists, for which a list of return values (transaction hashes), in the same order, is returned for all individual elements. Using multiple of these input fields returns the intersection of the values.

8.2.1 Parameters

- `bundles: Optional[Iterable[BundleHash]]`: List of bundle IDs.
- `addresses: Optional[Iterable[Address]]`: List of addresses.
- `tags: Optional[Iterable[Tag]]`: List of tags.
- `param: Optional[Iterable[TransactionHash]]`: List of approvee transaction IDs.

8.2.2 Return

This method returns a `dict` with the following items:

- `transactions: List[Transaction]`: List of Transaction objects that match the input

8.3 get_account_data

More comprehensive version of `get_transfers` that returns addresses and account balance in addition to bundles.

This function is useful in getting all the relevant information of your account.

8.3.1 Parameters

- `start: int`: Starting key index.
- `stop: Optional[int]`: Stop before this index. Note that this parameter behaves like the `stop` attribute in a `slice` object; the stop index is *not* included in the result.
- If `None` (default), then this method will check every address until it finds one without any transfers.
- `inclusion_states: bool` Whether to also fetch the inclusion states of the transfers. This requires an additional API call to the node, so it is disabled by default.

8.3.2 Return

This method returns a dict with the following items:

- `addresses: List[Address]`: List of generated addresses. Note that this list may include unused addresses.
- `balance: int`: Total account balance. Might be 0.
- `bundles: List[Bundles]`: List of bundles with transactions to/from this account.

8.4 get_bundles

Given a `TransactionHash`, returns the bundle(s) associated with it.

8.4.1 Parameters

- `transaction: TransactionHash`: Hash of a tail transaction.

8.4.2 Return

This method returns a `dict` with the following items:

- `bundles: List[Bundle]`: List of matching bundles. Note that this value is always a list, even if only one bundle was found.

8.5 get_inputs

Gets all possible inputs of a seed and returns them with the total balance.

This is either done deterministically (by generating all addresses until `find_transactions` returns an empty result), or by providing a key range to search.

8.5.1 Parameters

- `start: int`: Starting key index. Defaults to 0.
- `stop: Optional[int]`: Stop before this index.
- Note that this parameter behaves like the `stop` attribute in a `slice` object; the stop index is *not* included in the result.
- If `None` (default), then this method will not stop until it finds an unused address.
- `threshold: Optional[int]`: If set, determines the minimum threshold for a successful result:
- As soon as this threshold is reached, iteration will stop.
- If the command runs out of addresses before the threshold is reached, an exception is raised.
- If `threshold` is 0, the first address in the key range with a non-zero balance will be returned (if it exists).
- If `threshold` is `None` (default), this method will return **all** inputs in the specified key range.

Note that this method does not attempt to "optimize" the result (e.g., smallest number of inputs, get as close to `threshold` as possible, etc.); it simply accumulates inputs in order until the threshold is met.

8.5.2 Return

This method returns a `dict` with the following items:

- `inputs: List[Address]`: Addresses with nonzero balances that can be used as inputs.
- `totalBalance: int`: Aggregate balance of all inputs found.

8.6 get_latest_inclusion

Fetches the inclusion state for the specified transaction hashes, as of the latest milestone that the node has processed.

8.6.1 Parameters

- `hashes: Iterable[TransactionHash]`: Iterable of transaction hashes.

8.6.2 Return

This method returns a `dict` with the following items:

- `<TransactionHash>: bool`: Inclusion state for a single transaction.

There will be one item per transaction hash in the `hashes` parameter.

8.7 get_new_addresses

Generates one or more new addresses from the seed.

8.7.1 Parameters

- `index: int`: Specify the index of the new address (must be >= 1).
- `count: Optional[int]`: Number of addresses to generate (must be >= 1).
- If `None`, this method will scan the Tangle to find the next available unused address and return that.
- `security_level: int`: Number of iterations to use when generating new addresses. Lower values generate addresses faster, higher values result in more secure signatures in transactions.

8.7.2 Return

This method returns a `dict` with the following items:

- `addresses: List[Address]`: The generated address(es). Note that this value is always a list, even if only one address was generated.

8.8 get_transaction_objects

Returns a list of transaction objects given a list of transaction hashes. This is effectively calling `get_trytes` and converting the trytes to transaction objects. Similar to `find_transaction_objects`, but input is list of hashes.

8.8.1 Parameters

- `hashes`: List of transaction hashes that should be fetched.

8.8.2 Return

Returns a `dict` with the following items:

- `transactions: List[Transaction]`: List of transaction objects.

8.9 get_transfers

Returns all transfers associated with the seed.

8.9.1 Parameters

- `start: int`: Starting key index.
- `stop: Optional[int]`: Stop before this index.
- Note that this parameter behaves like the `stop` attribute in a `slice` object; the stop index is *not* included in the result.
- If `None` (default), then this method will check every address until it finds one without any transfers.

8.9.2 Return

This method returns a `dict` with the following items:

- `bundles: List[Bundle]`: Matching bundles, sorted by tail transaction timestamp.

8.10 is_reattachable

This API function helps you to determine whether you should replay a transaction or make a new one (either with the same input, or a different one).

This method takes one or more input addresses (i.e. from spent transactions) as input and then checks whether any transactions with a value transferred are confirmed.

If yes, it means that this input address has already been successfully used in a different transaction, and as such you should no longer replay the transaction.

8.10.1 Parameters

- `address: Iterable[Address]`: List of addresses.

8.10.2 Return

This method returns a `dict` with the following items:

- `reattachable: List[Bool]`: Always a list, even if only one address was queried.

8.11 prepare_transfer

Prepares transactions to be broadcast to the Tangle, by generating the correct bundle, as well as choosing and signing the inputs (for value transfers).

8.11.1 Parameters

- `transfers: Iterable[ProposedTransaction]`: Transaction objects to prepare.
- `inputs: Optional[Iterable[Address]]`: List of addresses used to fund the transfer. Ignored for zero-value transfers.
- If not provided, addresses will be selected automatically by scanning the Tangle for unspent inputs.

- `change_address: Optional[Address]`: If inputs are provided, any unspent amount will be sent to this address.
- If not specified, a change address will be generated automatically.

8.11.2 Return

This method returns a `dict` with the following items:

- `trytes: List[TransactionTrytes]`: Raw trytes for the transactions in the bundle, ready to be provided to `send_trytes`.

8.12 promote_transaction

Promotes a transaction by adding spam on top of it.

- `transaction: TransactionHash`: Transaction hash. Must be a tail.
- `depth: int`: Depth at which to attach the bundle.
- `min_weight_magnitude: Optional[int]`: Min weight magnitude, used by the node to calibrate Proof of Work.
- If not provided, a default value will be used.

8.12.1 Return

This method returns a `dict` with the following items:

- `bundle: Bundle`: The newly-published bundle.

8.13 replay_bundle

Takes a tail transaction hash as input, gets the bundle associated with the transaction and then replays the bundle by attaching it to the Tangle.

8.13.1 Parameters

- `transaction: TransactionHash`: Transaction hash. Must be a tail.
- `depth: int`: Depth at which to attach the bundle.
- `min_weight_magnitude: Optional[int]`: Min weight magnitude, used by the node to calibrate Proof of Work.
- If not provided, a default value will be used.

8.13.2 Return

This method returns a `dict` with the following items:

- `trytes: List[TransactionTrytes]`: Raw trytes that were published to the Tangle.

8.14 send_transfer

Prepares a set of transfers and creates the bundle, then attaches the bundle to the Tangle, and broadcasts and stores the transactions.

8.14.1 Parameters

- `depth: int`: Depth at which to attach the bundle.
- `transfers: Iterable[ProposedTransaction]`: Transaction objects to prepare.
- `inputs: Optional[Iterable[Address]]`: List of addresses used to fund the transfer. Ignored for zero-value transfers.
- If not provided, addresses will be selected automatically by scanning the Tangle for unspent inputs.
- `change_address: Optional[Address]`: If inputs are provided, any unspent amount will be sent to this address.
- If not specified, a change address will be generated automatically.
- `min_weight_magnitude: Optional[int]`: Min weight magnitude, used by the node to calibrate Proof of Work.
- If not provided, a default value will be used.

8.14.2 Return

This method returns a `dict` with the following items:

- `bundle: Bundle`: The newly-published bundle.

8.15 send_trytes

Attaches transaction trytes to the Tangle, then broadcasts and stores them.

8.15.1 Parameters

- `trytes: Iterable[TransactionTrytes]`: Transaction trytes to publish.
- `depth: int`: Depth at which to attach the bundle.
- `min_weight_magnitude: Optional[int]`: Min weight magnitude, used by the node to calibrate Proof of Work.
- If not provided, a default value will be used.

8.15.2 Return

This method returns a `dict` with the following items:

- `trytes: List[TransactionTrytes]`: Raw trytes that were published to the Tangle.

CHAPTER 9

Multisignature

Multisignature transactions are transactions which require multiple signatures before execution. In simplest example it means that, if there is token wallet which require 5 signatures from different parties, all 5 parties must sign spent transaction, before it will be processed.

It is standard functionality in blockchain systems and it is also implemented in IOTA

Note: You can read more about IOTA multisignature on the wiki.

9.1 Generating multisignature address

In order to use multisignature functionality, a special multisignature address must be created. It is done by adding each key digest in agreed order into digests list. At the end, last participant is converting digests list (Curl state trits) into multisignature address.

Note: Each multisignature addresses participant has to create its own digest locally. Then, when it is created it can be safely shared with other participants, in order to build list of digests which then will be converted into multisignature address.

Created digests should be shared with each multisignature participant, so each one of them could regenerate address and ensure it is OK.

Here is the example where digest is created:

```
# Create digest 3 of 3.
api_3 =\
  MultisigIota(
    adapter = 'http://localhost:14265',

    seed =
```

(continues on next page)

(continued from previous page)

```
      Seed(
        b'TESTVALUE9DONTUSEINPRODUCTION99999JYFRTI'
        b'WMKVVBAIEIYZDWLUVOYTZBKPKLLUMPDF9PPFLO9KT',
      ),
  )

gd_result = api_3.get_digests(index=8, count=1, security_level=2)

digest_3 = gd_result['digests'][0] # type: Digest
```

And here is example where digests are converted into multisignature address:

```
cma_result =\
  api_1.create_multisig_address(digests=[digest_1,
                                         digest_2,
                                         digest_3])

# For consistency, every API command returns a dict, even if it only
# has a single value.
multisig_address = cma_result['address'] # type: MultisigAddress
```

Note: As you can see in above example, multisignature addresses is created from list of digests, and in this case **order** is important. The same order need to be used in **signing transfer**.

9.2 Prepare transfer

Note: Since spending tokens from the same address more than once is insecure, remainder should be transferred to other address. So, this address should be created before as next to be used multisignature address.

First signer for multisignature wallet is defining address where tokens should be transferred and next wallet address for reminder:

```
pmt_result =\
  api_1.prepare_multisig_transfer(
    # These are the transactions that will spend the IOTAs.
    # You can divide up the IOTAs to send to multiple addresses if you
    # want, but to keep this example focused, we will only include a
    # single spend transaction.
    transfers = [
      ProposedTransaction(
        address =
          Address(
            b'TESTVALUE9DONTUSEINPRODUCTION99999NDGYBC'
            b'QZJFGGWZ9GBQFKDOLWMVILARZRHJMSYFZETZTHTZR',
          ),

        value = 42,

        # If you'd like, you may include an optional tag and/or
        # message.
```

(continues on next page)

(continued from previous page)

```
        tag = Tag(b'KITTEHS'),
        message = TryteString.from_unicode('thanx fur cheezburgers'),
      ),
    ],

    # Specify our multisig address as the input for the spend
    # transaction(s).
    # Note that PyOTA currently only allows one multisig input per
    # bundle (although the protocol does not impose a limit).
    multisig_input = multisig_address,

    # If there will be change from this transaction, you MUST specify
    # the change address!  Unlike regular transfers, multisig transfers
    # will NOT automatically generate a change address; that wouldn't
    # be fair to the other participants!
    change_address = None,
  )

prepared_trytes = pmt_result['trytes'] # type: List[TransactionTrytes]
```

9.3 Sign the inputs

When trytes are prepared, round of signing must be performed. Order of signing must be the same as in generate multisignature addresses procedure (as described above).

Note: In example below, all signing is done on one local machine. In real case, each participant sign bundle locally and then passes it to next participant in previously defined order

index, **count** and **security_lavel** parameters for each private key should be the same as used in **get_digests** function in previous steps.

```
bundle = Bundle.from_tryte_strings(prepared_trytes)

gpk_result = api_1.get_private_keys(index=0, count=1, security_level=3)
private_key_1 = gpk_result['keys'][0] # type: PrivateKey
private_key_1.sign_input_transactions(bundle, 1)

gpk_result = api_2.get_private_keys(index=42, count=1, security_level=3)
private_key_2 = gpk_result['keys'][0] # type: PrivateKey
private_key_2.sign_input_transactions(bundle, 4)

gpk_result = api_3.get_private_keys(index=8, count=1, security_level=2)
private_key_3 = gpk_result['keys'][0] # type: PrivateKey
private_key_3.sign_input_transactions(bundle, 7)

signed_trytes = bundle.as_tryte_strings()
```

Note: After creation, bundle can be optionally validated:

```
validator = BundleValidator(bundle)
if not validator.is_valid():
```

(continues on next page)

(continued from previous page)

```
  raise ValueError(
    'Bundle failed validation:\n{errors}'.format(
      errors = '\n'.join(('  - ' + e) for e in validator.errors),
    ),
  )
```

9.4 Broadcast the bundle

When bundle is created it can be broadcasted in standard way:

```
api_1.send_trytes(trytes=signed_trytes, depth=3)
```

9.5 Remarks

Full code example.

Note: How M-of-N works

One of the key differences between IOTA multi-signatures is that M-of-N (e.g. 3 of 5) works differently. What this means is that in order to successfully spend inputs, all of the co-signers have to sign the transaction. As such, in order to enable M-of-N we have to make use of a simple trick: sharing of private keys.

This concept is best explained with a concrete example:

> Lets say that we have a multi-signature between 3 parties: Alice, Bob and Carol. Each has their own private key, and they generated a new multi-signature address in the aforementioned order. Currently, this is a 3 of 3 multisig. This means that all 3 participants (Alice, Bob and Carol) need to sign the inputs with their private keys in order to successfully spend them.
>
> In order to enable a 2 of 3 multisig, the cosigners need to share their private keys with the other parties in such a way that no single party can sign inputs alone, but that still enables an M-of-N multsig. In our example, the sharing of the private keys would look as follows:
>
> Alice -> Bob
>
> Bob -> Carol
>
> Carol -> Alice
>
> Now, each participant holds two private keys that he/she can use to collude with another party to successfully sign the inputs and make a transaction. But no single party holds enough keys (3 of 3) to be able to independently make the transaction.

9.6 Important

There are some general rules (repeated once again for convenience) which should be followed while working with multisignature addresses (and in general with IOTA):

9.6.1 Signing order is important

When creating a multi-signature address and when signing a transaction for that address, it is important to follow the exact order that was used during the initial creation. If we have a multi-signature address that was signed in the following order: Alice -> Bob -> Carol. You will not be able to spend these inputs if you provide the signatures in a different order (e.g. Bob -> Alice -> Carol). As such, keep the signing order in mind.

9.6.2 Never re-use keys

Probably the most important rule to keep in mind: absolutely never re-use private keys. IOTA uses one-time Winternitz signatures, which means that if you re-use private keys you significantly decrease the security of your private keys, up to the point where signing of another transaction can be done on a conventional computer within few days. Therefore, when generating a new multi-signature with your co-signers, always increase the private key **index counter** and only use a single private key once. Don't use it for any other multi-signatures and don't use it for any personal transactions.

9.6.3 Never share your private keys

Under no circumstances - other than wanting to reduce the requirements for a multi-signature (see section **How M-of-N works**) - should you share your private keys. Sharing your private keys with others means that they can sign your part of the multi-signature successfully.

CHAPTER 10

PyOTA

This is the official Python library for the IOTA Core.

It implements both the official API, as well as newly-proposed functionality (such as signing, bundles, utilities and conversion).

10.1 Join the Discussion

If you want to get involved in the community, need help with getting setup, have any issues related with the library or just want to discuss Blockchain, Distributed Ledgers and IoT with other people, feel free to join our Discord.

If you encounter any issues while using PyOTA, please report them using the PyOTA Bug Tracker.

CHAPTER 11

Dependencies

PyOTA is compatible with Python 3.7, 3.6, 3.5 and 2.7

CHAPTER 12

Installation

To install the latest version:

```
pip install pyota
```

12.1 Optional C Extension

PyOTA has an optional C extension that improves the performance of its cryptography features significantly (speedups of **60x** are common!).

To install this extension, use the following command:

```
pip install pyota[ccurl]
```

12.2 Optional Local Pow

To perform proof-of-work locally without relying on a node, you can install an extension module called PyOTA-PoW .

Specifiy the `local_pow=True` argument when creating an api instance, that will redirect all `attach_to_tangle` API calls to an interface function in the `pow` package.

To install this extension, use the following command:

```
pip install pyota[pow]
```

Alternativley you can take a look on the repository Ccurl.interface.py to install Pyota-PoW. Follow the steps depicted in the repo's README file.

12.3 Installing from Source

1. Create virtualenv (recommended, but not required).
2. `git clone https://github.com/iotaledger/iota.py.git`
3. `pip install -e .`

12.3.1 Running Unit Tests

To run unit tests after installing from source:

```
python setup.py test
```

PyOTA is also compatible with tox, which will run the unit tests in different virtual environments (one for each supported version of Python).

To run the unit tests, it is recommended that you use the `-p` argument. This speeds up the tests by running them in parallel.

Install PyOTA with the `test-runner` extra to set up the necessary dependencies, and then you can run the tests with the `tox` command:

```
pip install -e .[test-runner]
tox -v -p all
```

CHAPTER 13

Documentation

PyOTA's documentation is available on ReadTheDocs.

If you are installing from source (see above), you can also build the documentation locally:

1. Install extra dependencies (you only have to do this once):

```
pip install .[docs-builder]
```

   Tip: To install the CCurl extension and the documentation builder tools together, use the following command:

```
pip install .[ccurl,docs-builder]
```

2. Switch to the `docs` directory:

```
cd docs
```

3. Build the documentation:

```
make html
```