PyOTA Documentation

Phoenix Zerin

Jun 04, 2020

CONTENTS

PyOTA is compatible with Python 3.7 and 3.6.

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

ONE

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.

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!

CHAPTER

TWO

BASIC CONCEPTS

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

The official IOTA documentation site gives a good and in-depth explanation of the concepts used in IOTA. PyOTA documentation will try to give references to the official site wherever possible.

2.1 Ternary

IOTA uses the ternary numerical system to represent data. The smallest unit of information is a `trit`, that can have a value of -1, 0 or 1 in a balanced ternary system. A combination of 3 `trits` equals one `tryte`, therefore a `tryte` can have 3 * 3 * 3 = 27 different values.

To represent a `tryte`, IOTA encodes these 27 values into characters based on the tryte alphabet.

In PyOTA, `trits` are represented as a sequence of numerical values (`List[int]`) while trytes have their own class called *`TryteString`*.

2.2 IOTA token

The IOTA token is a unit of value that can be transferred over an IOTA network through transfer bundles.

The IOTA token was launched on the Mainnet in June 2017. At this point, the nodes in the network were hard-coded with a total supply of 2,779,530,283 277,761. This large supply allows each of the billions of devices, which are expected to be a part of the Internet of Things, to have its own wallet and transact micropayments with other devices.

2.3 Seed

Seed in IOTA is your unique password. It is the digital key that unlocks your safe that holds your tokens, or proves the ownership of messages.

Seeds in PyOTA are always 81 trytes long and may only contain characters from the tryte alphabet.

Warning: Treat your seed(s) the same as you would the password for any other financial service. Never share your seed with anyone, and never use online generators to create a seed. The library can help you to create your own locally and it does not require internet connection: `iota.crypto.Seed.random()`.

For PyOTA-specific implementation details on seeds, see `crypto.Seed`.

2.4 Address

To send or receive any transaction (let them be zero-value or value transacitons) in IOTA, you will need to specify an address. An address is like a physical mailbox on your entrance door: anyone can drop things in your mailbox (send you messages or tokens), but only you can empty it (withdraw tokens).

Warning: Addresses should not be re-used once they are spent from. **You can receive as many transactions to an address as you wish, but only spend from that address once.**

Addresses are generated from your seed through cryptographic functions. There are 9^{57} different addresses that one might generate from a seed, which is quite a lot. Given your `seed`, the `index` and `security level` parameters specify which address will be generated from it. The process is deterministic, meaning that same input paramteres always generate the same address.

Addresses are 81 trytes long and may contain extra 9 trytes for checksum. The checksum may be used to verify that an address is in fact a valid IOTA address.

For-PyOTA specific implementation details on addresses, see *`Address`*.

2.5 Transaction

A transaction is a single transfer instruction that can either withdraw IOTA tokens from an address, deposit them into an address, or have zero-value (contain data, a message, or a signature). If you want to send anything to an IOTA network, you must send it to a node as a transaction.

—from the official IOTA documentation site

Transactions are always 2673 trytes long and their structure is defined by the protocol. They can be classified into three categories:

- Input transaction: A transaction that withdraws tokens from an address.
- Output transaction: A transaction that deposits tokens to an address.
- Zero-value transaction: A transaction that has 0 value and might carry messages or signatures.

Depending on the type of the transaction, different fields are required to be filled.

A transaction's unique identifier in IOTA is the *`TransactionHash`*, that is generated from the trytes of the transaction. If any trytes change in the transaction, the returning transaction hash would alter. This way, transaction hashes ensure the immutability of the Tangle.

To become accepted by the network, a transaction has to be attached to the Tangle. The attachment process means that the transaction should reference two unconfirmed transactions (tips) in the Tangle and do a small proof-of-work. This process might be performed by a node, or by using the local proof-of-work feature of the client libraries.

For PyOTA-specific implementation details on transactions, see *`Transaction`* and *`ProposedTransaction`*.

2.6 Bundle

> *A bundle is a group of transactions that rely on each other's validity. For example, a transaction that deposits IOTA tokens into an address relies on another transaction to withdraw those IOTA tokens from another address. Therefore, those transactions must be in the same bundle.*
>
> —from the official IOTA documentation site

In other words, a bundle is collection of transactions, treated as an atomic unit when attached to the Tangle.

Note: Unlike a block in a blockchain, bundles are not first-class citizens in IOTA; only transactions get stored in the Tangle.

Instead, bundles must be inferred by following linked transactions with the same bundle hash.

Transactions in the bundle are linked together through their `trunkTransaction` fields, furthermore they are indexed within the bundle and contain a `bundleHash` field that is a unique identifier for the bundle.

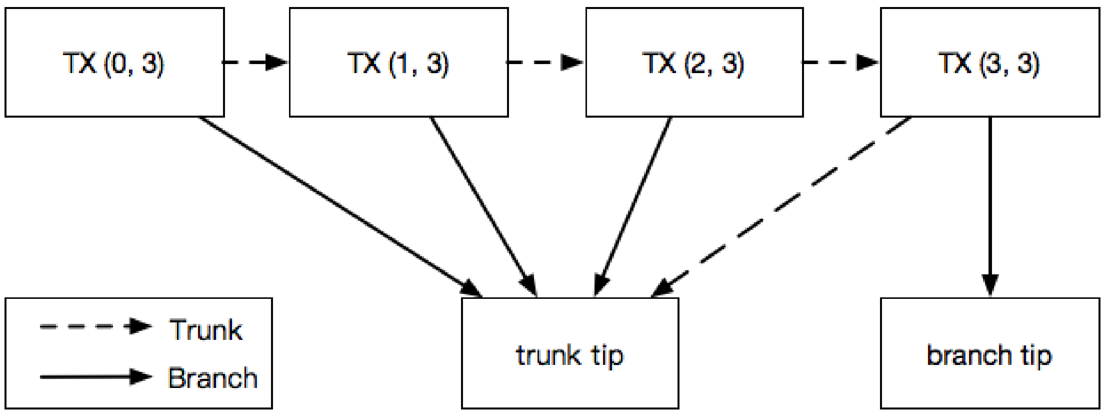

Fig. 1: Structure of a bundle with four transactions. Numbers in brackets denote (`currentIndex`, `lastIndex`) fields. Head of the bundle has index 3, while tail has index 0.

Read more about how bundles are structured.

Bundles can be classified into two categories:

- Transfer bundles: Bundles that contain input and output transactions. A bundle always has to be balanced, meaning that input transaction values should equal to output transaction values.
- Zero-value bundles: Bundles that contain only zero-value transactions.

For PyOTA-specific implementation details on bundles, see *`Bundle`* and *`ProposedBundle`*.

Now that you are familiar with some basic IOTA concepts, it is time to explore how PyOTA implements these and how you can work with them.

CHAPTER

THREE

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.

Since everything in IOTA is represented as a sequence of `trits` and `trytes`, let us take a look on how you can work with them in PyOTA.

3.1 TryteString

class iota.**TryteString**(*trytes: Union[AnyStr, bytearray, TryteString], pad: Optional[int] = None*)

A string representation of a sequence of trytes.

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]*$`).

Important: A TryteString does not represent a numeric value!

Parameters

- **trytes** (*TrytesCompatible*) – Byte string or bytearray.
- **pad** (*Optional[int]*) – Ensure at least this many trytes.

  If there are too few, null trytes will be appended to the TryteString.

  Note: If the TryteString is too long, it will *not* be truncated!

Example usage:

```
from iota import TryteString

# Create a TryteString object from bytes.
trytes_1 = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')

# Ensure the created object is 81 trytes long by padding it with zeros.
```

(continues on next page)

(continued from previous page)

```
# The value zero is represented with character '9' in trytes.
trytes_1 = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA', pad=81)

# Create a TryteString object from text type.
# Note that this will throw error if text contains unsupported characters.
trytes_2 = TryteString('LH9GYEMHCF9GWHZFEELHVFOEOHNEEEWHZFUD')

# Comparison and concatenation:
if trytes_1 != trytes_2:
  trytes_combined = trytes_1 + trytes_2

# As dictionary keys:
index = {
  trytes_1: 42,
  trytes_2: 86,
}
```

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.

Let's explore the capabilities of the `TryteString` base class.

3.1.1 Encoding

You may use classmethods to create a `TryteString` from `bytes`, `unicode string` or from a list of `trits`.

from_bytes

classmethod `TryteString.`**`from_bytes`**(*bytes_: Union[bytes, bytearray], codec: str = 'trytes_ascii', *args: Any, **kwargs: Any*) → T

Creates a TryteString from a sequence of bytes.

Parameters

- **bytes_** (*Union[bytes,bytearray]*) – Source bytes. ASCII representation of a sequence of bytes. Note that only tryte alphabet supported!
- **codec** (*str*) – Reserved for future use. Currently supports only the 'trytes_ascii' codec. See https://github.com/iotaledger/iota.py/issues/62 for more information.
- **args** – Additional positional arguments to pass to the initializer.
- **kwargs** – Additional keyword arguments to pass to the initializer.

Returns *TryteString* object.

Example usage:

```
from iota import TryteString
message_trytes = TryteString.from_bytes(b'HELLO999IOTA')
```

from_unicode

classmethod TryteString.**from_unicode**(*string: str*, **args: Any*, ***kwargs: Any*) → T

Creates a TryteString from a Unicode string.

Parameters

- **string** (*str*) – Source Unicode string.
- **args** – Additional positional arguments to pass to the initializer.
- **kwargs** – Additional keyword arguments to pass to the initializer.

Returns *TryteString* object.

Example usage:

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

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.

from_trits

classmethod TryteString.**from_trits**(*trits: Iterable[int], *args: Any, **kwargs: Any*) → T

Creates a TryteString from a sequence of trits.

Parameters

- **trits** (*Iterable[int]*) – Iterable of trit values (-1, 0, 1).
- **args** – Additional positional arguments to pass to the initializer.
- **kwargs** – Additional keyword arguments to pass to the initializer.

Returns *TryteString* object.

Example usage:

```
from iota import TryteString
message_trytes = TryteString.from_trits(
    [1, 0, -1, -1, 1, 0, 1, -1, 0, -1, 1, 0, 0, 1, 0, 0, 1, 0, -1, 1, 1, -1, 1, 0]
)
```

References:

- `int_from_trits()`

- *as_trits()*

from_trytes

classmethod TryteString.**from_trytes**(*trytes: Iterable[Iterable[int]], *args: Any, **kwargs: Any*) → T

Creates a TryteString from a sequence of trytes.

Parameters

- **trytes** (*Iterable[Iterable[int]]*) – Iterable of tryte values.

  In this context, a tryte is defined as a list containing 3 trits.
- **args** – Additional positional arguments to pass to the initializer.
- **kwargs** – Additional keyword arguments to pass to the initializer.

Returns *TryteString* object.

Example usage:

```
from iota import TryteString
message_trytes = TryteString.from_trytes(
    [
        [1, 0, -1],
        [-1, 1, 0],
        [1, -1, 0],
        [-1, 1, 0],
        [0, 1, 0],
        [0, 1, 0],
        [-1, 1, 1],
        [-1, 1, 0],
    ]
)
```

References:

- *as_trytes()*

Additionally, you can encode a *TryteString* into a lower-level primitive (usually bytes). This might be useful when the *TryteString* contains ASCII encoded characters but you need it as `bytes`. See the example below:

encode

TryteString.**encode**(*errors: str = 'strict', codec: str = 'trytes_ascii'*) → bytes

Encodes the TryteString into a lower-level primitive (usually bytes).

Parameters

- **errors** (*str*) – How to handle trytes that can't be converted:

  'strict' raise an exception (recommended).

  'replace' replace with '?'.

  'ignore' omit the tryte from the result.
- **codec** (*str*) – Reserved for future use.

  See https://github.com/iotaledger/iota.py/issues/62 for more information.

Raises

- `iota.codecs.TrytesDecodeError` if the trytes cannot be decoded into bytes.

Returns Python `bytes` object.

Example usage:

```
from iota import TryteString

# Message payload as unicode string
message = 'Hello, iota!'

# Create TryteString
message_trytes = TryteString.from_unicode(message)

# Encode TryteString into bytes
encoded_message_bytes = message_trytes.encode()

# This will be b'Hello, iota'
print(encoded_message_bytes)

# Get the original message
decoded = encoded_message_bytes.decode()

print(decoded == message)
```

3.1.2 Decoding

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!

decode

`TryteString.`**`decode`**(*errors: str = 'strict'*, *strip_padding: bool = True*) → str

Decodes the TryteString into a higher-level abstraction (usually Unicode characters).

Parameters

- **`errors`** (*`str`*) – How to handle trytes that can't be converted, or bytes that can't be decoded using UTF-8:

  'strict' raise an exception (recommended).

  'replace' replace with a placeholder character.

  'ignore' omit the invalid tryte/byte sequence.

- **`strip_padding`** (*`bool`*) – Whether to strip trailing null trytes before converting.

Raises

- `iota.codecs.TrytesDecodeError` if the trytes cannot be decoded into bytes.
- `UnicodeDecodeError` if the resulting bytes cannot be decoded using UTF-8.

Returns `Unicode string` object.

Example usage:

```
from iota import TryteString

trytes = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')

message = trytes.decode()
```

as_json_compatible

`TryteString.`**`as_json_compatible`**`()` → str

Returns a JSON-compatible representation of the object.

References:

- `iota.json.JsonEncoder`.

Returns JSON-compatible representation of the object (string).

Example usage:

```
from iota import TryteString

trytes = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')

json_payload = trytes.as_json_compatible()
```

as_integers

`TryteString.`**`as_integers`**`()` → List[int]

Converts the TryteString into a sequence of integers.

Each integer is a value between -13 and 13.

See the tryte alphabet for more info.

Returns `List[int]`

Example usage:

```
from iota import TryteString

trytes = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')

tryte_ints = trytes.as_integers()
```

as_trytes

`TryteString.`**`as_trytes`**`()` → List[List[int]]

Converts the TryteString into a sequence of trytes.

Each tryte is represented as a list with 3 trit values.

See *`as_trits()`* for more info.

Important: *TryteString* is not a numeric type, so the result of this method should not be interpreted as an integer!

Returns `List[List[int]]`

Example usage:

```
from iota import TryteString

trytes = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')

tryte_list = trytes.as_trytes()
```

as_trits

`TryteString.`**`as_trits`**`()` → List[int]

Converts the TryteString into a sequence of trit values.

A trit may have value 1, 0, or -1.

References:

- https://en.wikipedia.org/wiki/Balanced_ternary

Important: *TryteString* is not a numeric type, so the result of this method should not be interpreted as an integer!

Returns `List[int]`

Example usage:

```
from iota import TryteString

trytes = TryteString(b'RBTC9D9DCDQAEASBYBCCKBFA')

trits = trytes.as_trits()
```

3.1.3 Generation

random

`classmethod` `TryteString.`**`random`**(*length: Optional[int] = None*) → T

Generates a random sequence of trytes.

Parameters **`length`** (*`Optional[int]`*) – Number of trytes to generate.

Returns *TryteString* object.

Raises **`TypeError`** –

- if `length` is negative,
- if `length` is not defined, and the class doesn't have `LEN` attribute.

3.2 Seed

class iota.**Seed**(*trytes: Union[AnyStr, bytearray, TryteString, None] = None*)

An *iota.TryteString* that acts as a seed for crypto functions.

Note: This class is identical to *iota.TryteString*, but it has a distinct type so that seeds can be identified in Python code.

IMPORTANT: For maximum security, a seed must be EXACTLY 81 trytes!

Parameters trytes (*TrytesCompatible*) – Byte string or bytearray.

Raises Warning – if trytes are longer than 81 trytes in length.

References:

- https://iota.stackexchange.com/q/249

3.2.1 random

classmethod Seed.**random**(*length: int = 81*) → iota.crypto.types.Seed

Generates a random seed using a CSPRNG.

Parameters length (*int*) – Length of seed, in trytes.

For maximum security, this should always be set to 81, but you can change it if you're 110% sure you know what you're doing.

See https://iota.stackexchange.com/q/249 for more info.

Returns *iota.Seed* object.

Example usage:

```
from iota import Seed

my_seed = Seed.random()

print(my_seed)
```

3.3 Address

class iota.**Address**(*trytes: Union[AnyStr, bytearray, TryteString], balance: Optional[int] = None, key_index: Optional[int] = None, security_level: Optional[int] = None*)

A *TryteString* that acts as an address, with support for generating and validating checksums.

Parameters

- **trytes** (*TrytesCompatible*) – Object to construct the address from.
- **balance** (*Optional[int]*) – Known balance of the address.
- **key_index** (*Optional[int]*) – Index of the address that was used during address generation. Must be greater than zero.
- **security_level** (*Optional[int]*) – Security level that was used during address generation. Might be 1, 2 or 3.

:raises ValueError: if trytes is longer than 81 trytes, unless it is exactly 90 trytes long (address + checksum).

address: TryteString = None
Address trytes without the checksum.

balance = None
Balance owned by this address. Defaults to `None`; usually set via the `getInputs` command.

References:

- *Iota.get_inputs()*
- *ProposedBundle.add_inputs()*

key_index = None
Index of the key used to generate this address. Defaults to `None`; usually set via `AddressGenerator`.

References:

- *iota.crypto.addresses.AddressGenerator*

security_level = None
Number of hashes in the digest that was used to generate this address.

3.3.1 as_json_compatible

`Address.`**`as_json_compatible`**`()` → Dict[str, Union[str, int]]
Returns a JSON-compatible representation of the Address.

Returns

`dict` with the following structure:

```
{
    'trytes': str,
    'balance': int,
    'key_index': int,
    'security_level': int,
}
```

Example usage:

```
from iota import Address

# Example address only, do not use in your code!
addy = Address(
    b'LVHHIXQNYKWQMGXGLFOKOCDFHPKXAUKWMSZVDRAT'
    b'TICUZXFACM9DNJELJGMLMK99KDVVOOWLINVBZIGWZ'
)

print(addy.as_json_compatible())
```

3.3.2 is_checksum_valid

`Address.is_checksum_valid()` → bool

Returns whether this address has a valid checksum.

Returns `bool`

Example usage:

```
from iota import Address

# Example address only, do not use in your code!
addy = Address(
    b'LVHHIXQNYKWQMGXGLFOKOCDFHPKXAUKWMSZVDRAT'
    b'TICUZXFACM9DNJELJGMLMK99KDVVOOWLINVBZIGWZ'
)

# Should be ``False``
print(addy.is_checksum_valid())

addy.add_checksum()

# Should be ``True``
print(addy.is_checksum_valid())
```

3.3.3 with_valid_checksum

`Address.with_valid_checksum()` → iota.types.Address

Returns the address with a valid checksum attached.

Returns *`Address`* object.

Example usage:

```
from iota import Address

# Example address only, do not use in your code!
addy = Address(
    b'LVHHIXQNYKWQMGXGLFOKOCDFHPKXAUKWMSZVDRAT'
    b'TICUZXFACM9DNJELJGMLMK99KDVVOOWLINVBZIGWZ'
)

addy_with_checksum = addy.with_valid_checksum()

print(addy_with_checksum)

# Should be ``True``
print(addy_with_checksum.is_checksum_valid())
```

3.3.4 add_checksum

`Address.add_checksum()` → None

Adds checksum to *Address* object.

Returns `None`

Example usage:

```
from iota import Address

# Example address only, do not use in your code!
addy = Address(
    b'LVHHIXQNYKWQMGXGLFOKOCDFHPKXAUKWMSZVDRAT'
    b'TICUZXFACM9DNJELJGMLMK99KDVVOOWLINVBZIGWZ'
)

# Should be ``False``
print(addy.is_checksum_valid())

print(addy.checksum)

addy.add_checksum()

# Should be ``True``
print(addy.is_checksum_valid())

print(addy.checksum)
```

3.3.5 remove_checksum

`Address.remove_checksum()` → None

Removes checksum from *Address* object.

Returns `None`

Example usage:

```
from iota import Address

# Example address only, do not use in your code!
addy = Address(
    b'LVHHIXQNYKWQMGXGLFOKOCDFHPKXAUKWMSZVDRAT'
    b'TICUZXFACM9DNJELJGMLMK99KDVVOOWLINVBZIGWZ'
    b'AACAMCWUW'   # 9 checksum trytes
)

# Should be ``True``
print(addy.is_checksum_valid())

print(addy.checksum)

addy.remove_checksum()

# Should be ``False``
print(addy.is_checksum_valid())

print(addy.checksum)
```

3.4 AddressChecksum

class `iota.`**`AddressChecksum`**(*trytes: Union[AnyStr, bytearray, TryteString]*)

A *`TryteString`* that acts as an address checksum.

Parameters **`trytes`** (*`TrytesCompatible`*) – Checksum trytes.

Raises **`ValueError`** – if `trytes` is not exactly 9 trytes in length.

`LEN = 9`

Length of an address checksum.

3.5 Hash

class `iota.`**`Hash`**(*trytes: Union[AnyStr, bytearray, TryteString]*)

A *`TryteString`* that is exactly one hash long.

Parameters **`trytes`** (*`TrytesCompatible`*) – Object to construct the hash from.

Raises **`ValueError`** – if `trytes` is longer than 81 trytes.

`LEN = 81`

Length is always 81 trytes long.

3.6 TransactionHash

class `iota.`**`TransactionHash`**(*trytes: Union[AnyStr, bytearray, TryteString]*)

An *`TryteString`* (*`Hash`*) that acts as a transaction hash.

3.7 BundleHash

class `iota.`**`BundleHash`**(*trytes: Union[AnyStr, bytearray, TryteString]*)

An *`TryteString`* (*`Hash`*) that acts as a bundle hash.

3.8 TransactionTrytes

class `iota.`**`TransactionTrytes`**(*trytes: Union[AnyStr, bytearray, TryteString]*)

An *`TryteString`* representation of a Transaction.

Raises **`ValueError`** – if `trytes` is longer than 2673 trytes in length.

`LEN = 2673`

Length of a transaction in trytes.

3.9 Fragment

class iota.**Fragment**(*trytes: Union[AnyStr, bytearray, TryteString]*)

An *TryteString* representation of a signature/message fragment in a transaction.

Raises ValueError – if `trytes` is longer than 2187 trytes in length.

LEN = 2187

Length of a fragment in trytes.

3.10 Nonce

class iota.**Nonce**(*trytes: Union[AnyStr, bytearray, TryteString]*)

An *TryteString* that acts as a transaction nonce.

Raises ValueError – if `trytes` is longer than 27 trytes in length.

LEN = 27

Length of a nonce in trytes.

3.11 Tag

class iota.**Tag**(*trytes: Union[AnyStr, bytearray, TryteString]*)

A TryteString that acts as a transaction tag.

Parameters trytes (*TrytesCompatible*) – Tag trytes.

Raises ValueError – if `trytes` is longer than 27 trytes in length.

LEN = 27

Length of a tag.

3.12 Transaction Types

PyOTA defines two different types used to represent transactions:

- *Transaction* for transactions that have already been attached to 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.
- *ProposedTransaction* for transactions that have been created locally and have not been broadcast yet.

3.12.1 Transaction

Each *Transaction* object has several instance attributes that you may manipulate and properties you can use to extract their values as trytes. See the class documentation below:

class iota.**Transaction**(*hash_: Optional[iota.transaction.types.TransactionHash], signature_message_fragment: Optional[iota.transaction.types.Fragment], address: iota.types.Address, value: int, timestamp: int, current_index: Optional[int], last_index: Optional[int], bundle_hash: Optional[iota.transaction.types.BundleHash], trunk_transaction_hash: Optional[iota.transaction.types.TransactionHash], branch_transaction_hash: Optional[iota.transaction.types.TransactionHash], tag: Optional[iota.types.Tag], attachment_timestamp: Optional[int], attachment_timestamp_lower_bound: Optional[int], attachment_timestamp_upper_bound: Optional[int], nonce: Optional[iota.transaction.types.Nonce], legacy_tag: Optional[iota.types.Tag] = None*)

A transaction that has been attached to the Tangle.

Parameters

- **hash** (*Optional[TransactionHash]*) – Transaction ID
- **signature_message_fragment** (*Optional[Fragment]*) – Signature or message fragment.
- **address** (Address) – The address associated with this transaction.
- **value** (*int*) – Value of the transaction in iotas. Can be negative as well (spending from address).
- **timestamp** (*int*) – Unix timestamp in seconds.
- **current_index** (*Optional[int]*) – Index of the transaction within the bundle.
- **last_index** (*Optional[int]*) – Index of head transaction in the bundle.
- **bundle_hash** (*Optional[BundleHash]*) – Bundle hash of the bundle containing the transaction.
- **trunk_transaction_hash** (*Optional[TransactionHash]*) – Hash of trunk transaction.
- **branch_transaction_hash** (*Optional[TransactionHash]*) – Hash of branch transaction.
- **tag** (*Optional[Tag]*) – Optional classification tag applied to this transaction.
- **attachment_timestamp** (*Optional[int]*) – Unix timestamp in milliseconds, decribes when the proof-of-work for this transaction was done.
- **attachment_timestamp_lower_bound** (*Optional[int]*) – Unix timestamp in milliseconds, lower bound of attachment.
- **attachment_timestamp_upper_bound** (*Optional[int]*) – Unix timestamp in milliseconds, upper bound of attachment.
- **nonce** (*Optional[Nonce]*) – Unique value used to increase security of the transaction hash. Result of the proof-of-work aglorithm.
- **legacy_tag** (*Optional[Tag]*) – Optional classification legacy_tag applied to this transaction.

Returns *Transaction* object.

address: Address = None

The address associated with this transaction.

Depending on the transaction's `value`, this address may be a sender or a recipient. If `value` is != 0, the associated address' balance is adjusted as a result of this transaction.

Type *`Address`*

`attachment_timestamp: Optional[int] = None`

Estimated epoch time of the attachment to the tangle.

Decribes when the proof-of-work for this transaction was done.

Type `int`, unix timestamp in milliseconds,

`property attachment_timestamp_as_trytes`

Returns a TryteString representation of the transaction's *`attachment_timestamp`*.

`attachment_timestamp_lower_bound: Optional[int] = None`

The lowest possible epoch time of the attachment to the tangle.

Type `int`, unix timestamp in milliseconds.

`property attachment_timestamp_lower_bound_as_trytes`

Returns a TryteString representation of the transaction's *`attachment_timestamp_lower_bound`*.

`attachment_timestamp_upper_bound: Optional[int] = None`

The highest possible epoch time of the attachment to the tangle.

Type `int`, unix timestamp in milliseconds.

`property attachment_timestamp_upper_bound_as_trytes`

Returns a TryteString representation of the transaction's *`attachment_timestamp_upper_bound`*.

`branch_transaction_hash: Optional[TransactionHash] = None`

An unrelated transaction that this transaction "approves".

In order to add a transaction to the Tangle, the client must perform PoW to "approve" two existing transactions, called the "trunk" and "branch" transactions.

The branch transaction may be selected strategically to maximize the bundle's chances of getting confirmed; otherwise it usually has no significance.

Type *`TransactionHash`*

`bundle_hash: Optional[BundleHash] = None`

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.

Type *`BundleHash`*

`current_index: Optional[int] = None`

The position of the transaction inside the bundle.

- If the `current_index` value is 0, then this is the "head transaction".
- If it is equal to `last_index`, then this is the "tail transaction".

For value transfers, the "spend" transaction is generally in the 0th position, followed by inputs, and the "change" transaction is last.

Type `int`

`property current_index_as_trytes`

Returns a TryteString representation of the transaction's *`current_index`*.

hash: TransactionHash = None

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.

Type *TransactionHash*

is_confirmed: bool = None

Whether this transaction has been confirmed by neighbor nodes. Must be set manually via the `getInclusionStates` API command.

Type `Optional[bool]`

References:

- *Iota.get_inclusion_states()*
- *Iota.get_transfers()*

property is_tail

Returns whether this transaction is a tail (first one in the bundle).

Because of the way the Tangle is organized, the tail transaction is generally the last one in the bundle that gets attached, even though it occupies the first logical position inside the bundle.

last_index: Optional[int] = None

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.

Type `int`

property last_index_as_trytes

Returns a TryteString representation of the transaction's *last_index*.

property legacy_tag

Return the legacy tag of the transaction. If no legacy tag was set, returns the tag instead.

nonce: Optional[Nonce] = None

Unique value used to increase security of the transaction hash.

This is the product of the PoW process.

Type *Nonce*

signature_message_fragment: Optional[Fragment] = None

"Signature/Message Fragment" (note the slash):

- For inputs, this contains a fragment of the cryptographic signature, used to verify the transaction (depending on the security level of the corresponding address, the entire signature is usually too large to fit into a single transaction, so it is split across multiple transactions instead).
- For other transactions, this contains a fragment of the message attached to the transaction (if any). This can be pretty much any value. Like signatures, the message may be split across multiple transactions if it is too large to fit inside a single transaction.

Type *Fragment*

tag: Optional[Tag] = None

Optional classification tag applied to this transaction.

Many transactions have empty tags (`Tag(b'999999999999999999999999999')`).

Type *Tag*

timestamp: int = None

Timestamp used to increase the security of the transaction hash.

Describes when the transaction was created.

Important: This value is easy to forge! Do not rely on it when resolving conflicts!

Type `int`, unix timestamp in seconds.

property timestamp_as_trytes

Returns a TryteString representation of the transaction's *timestamp*.

trunk_transaction_hash: Optional[TransactionHash] = None

The transaction hash of the next transaction in the bundle.

In order to add a transaction to the Tangle, the client must perform PoW to "approve" two existing transactions, called the "trunk" and "branch" transactions.

The trunk transaction is generally used to link transactions within a bundle.

Type *TransactionHash*

value: int = None

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.

Type `int`

property value_as_trytes

Returns a TryteString representation of the transaction's *value*.

as_json_compatible

`Transaction.`**`as_json_compatible`**`()` → dict

Returns a JSON-compatible representation of the object.

Returns

`dict` with the following structure:

```
{
    'hash_': TransactionHash,
    'signature_message_fragment': Fragment,
    'address': Address,
    'value': int,
    'legacy_tag': Tag,
    'timestamp': int,
    'current_index': int,
    'last_index': int,
    'bundle_hash': BundleHash,
    'trunk_transaction_hash': TransactionHash,
    'branch_transaction_hash': TransactionHash,
```

(continues on next page)

(continued from previous page)

```
    'tag': Tag,
    'attachment_timestamp': int,
    'attachment_timestamp_lower_bound': int,
    'attachment_timestamp_upper_bound': int,
    'nonce': Nonce,
}
```

References:

- `iota.json.JsonEncoder`.

as_tryte_string

`Transaction.as_tryte_string()` → iota.transaction.types.TransactionTrytes

Returns a TryteString representation of the transaction.

Returns `TryteString` object.

from_tryte_string

classmethod `Transaction.from_tryte_string`(*trytes: Union[AnyStr, bytearray, TryteString], hash_: Optional[iota.transaction.types.TransactionHash] = None*) → T

Creates a Transaction object from a sequence of trytes.

Parameters

- **trytes** (*TrytesCompatible*) – Raw trytes. Should be exactly 2673 trytes long.
- **hash** (*Optional[TransactionHash]*) – The transaction hash, if available.

  If not provided, it will be computed from the transaction trytes.

Returns `Transaction` object.

Example usage:

```
from iota import Transaction

txn =\
  Transaction.from_tryte_string(
    b'GYPRVHBEZOOFXSHQBLCYW9ICTCISLHDBNMMVYD9JJHQMPQCTIQAQTJNNNJ9IDXLRCC'
    b'OYOXYPCLR9PBEY9ORZIEPPDNTI9CQWYZUOTAVBXPSBOFEQAPFLWXSWUIUSJMSJIIIZ'
    b'WIKIRH9GCOEVZFKNXEVCUCIIWZQCQEUVRZOCMEL9AMGXJNMLJCIA9UWGRPPHCEOPTS'
    b'VPKPPPCMQXYBHMSODTWUOABPKWFFFQJHCBVYXLHEWPD9YUDFTGNCYAKQKVEZYRBQRB'
    b'XIAUX9SVEDUKGMTWQIYXRGSWYRK9SRONVGTW9YGHSZRIXWGPCCUCDRMAXBPDFVHSRY'
    b'WHGB9DQSQFQKSNICGPIPTRZINYRXQAFSWSEWIFRMSBMGTNYPRWFSOIIWWT9IDSELM9'
    b'JUOOWFNCCSHUSMGNROBFJX9JQ9XT9PKEGQYQAWAFPRVRRVQPUQBHLSNTEFCDKBWRCD'
    b'X9EYOBB9KPMTLNNQLADBDLZPRVBCKVCYQEOLARJYAGTBFR9QLPKZBOYWZQOVKCVYRG'
    b'YI9ZEFIQRKYXLJBZJDBJDJVQZCGYQMROVHNDBLGNLQODPUXFNTADDVYNZJUVPGB9LV'
    b'PJIYLAPBOEHPMRWUIAJXVQOEM9ROEYUOTNLXVVQEYRQWDTQGDLEYFIYNDPRAIXOZEB'
    b'CS9P99AZTQQLKEILEVXMSHBIDHLXKUOMMNFKPYHONKEYDCHMUNTTNRYVMMEYHPGASP'
    b'ZXASKRUPWQSHDMU9VPS99ZZ9SJJYFUJFFMFORBYDILBXCAVJDPDFHTTTIYOVGLRDYR'
    b'TKHXJORJVYRPTDH9ZCPZ9ZADXZFRSFPIQKWLBRNTWJHXTOAUOL9FVGTUMMPYGYICJD'
    b'XMOESEVDJWLMCVTJLPIEKBE9JTHDQWV9MRMEWFLPWGJFLUXI9BXPSVWCMUWLZSEWHB'
    b'DZKXOLYNOZAPOYLQVZAQMOHGTTQEUAOVKVRRGAHNGPUEKHFVPVCOYSJAWHZU9DRROH'
```

(continues on next page)

(continued from previous page)

```
    b'BETBAFTATVAUGOEGCAYUXACLSSHHVYDHMDGJP9AUCLWLNTFEVGQGHQXSKEMVOVSKQE'
    b'EWHWZUDTYOBGCURRZSJZLFVQQAAYQO9TRLFFN9HTDQXBSPPJYXMNGLLBHOMNVXNOWE'
    b'IDMJVCLLDFHBDONQJCJVLBLCSMDOUQCKKCQJMGTSTHBXPXAMLMSXRIPUBMBAWBFNLH'
    b'LUJTRJLDERLZFUBUSMF999XNHLEEXEENQJNOFFPNPQ9PQICHSATPLZVMVIWLRTKYPI'
    b'XNFGYWOJSQDAXGFHKZPFLPXQEHCYEAGTIWIJEZTAVLNUMAFWGGLXMBNUQTOFCNLJTC'
    b'DMWVVZGVBSEBCPFSM99FLOIDTCLUGPSEDLOKZUAEVBLWNMODGZBWOVQT9DPFOTSKRA'
    b'BQAVOQ9RXWBMAKFYNDCZOJGTCIDMQSQQSODKDXTPFLNOKSIZEOY9HFUTLQRXQMEPGO'
    b'XQGLLPNSXAUCYPGZMNWMQWSWCKAQYKXJTWINSGPPZG9HLDLEAWUWEVCTVRCBDFOXKU'
    b'ROXH9HXXAXVPEJFRSLOGRVGYZASTEBAQNXJJROCYRTDPYFUIQJVDHAKEG9YACV9HCP'
    b'JUEUKOYFNWDXCCJBIFQKYOXGRDHVTHEQUMHO9999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'9999999999999999999999999999999999999999999999999999999999999999'
    b'99999999999RKWEEVD99A99999999A9999999NFDPEEZCWVYLKZGSLCQNOFUSENI'
    b'XRHWWTZFBXMPSQHEDFWZULBZFEOMNLRNIDQKDNNIELAOXOVMYEI9PGTKORV9IKTJZQ'
    b'UBQAWTKBKZ9NEZHBFIMCLV9TTNJNQZUIJDFPTTCTKBJRHAITVSKUCUEMD9M9SQJ999'
    b'999TKORV9IKTJZQUBQAWTKBKZ9NEZHBFIMCLV9TTNJNQZUIJDFPTTCTKBJRHAITVSK'
    b'UCUEMD9M9SQJ9999999999999999999999999999999999999999999999999999'
    b'99999999999999999999999999999999'
  )
```

get_bundle_essence_trytes

Transaction.**get_bundle_essence_trytes**() → iota.types.TryteString

Returns the values needed for calculating bundle hash. The bundle hash is the hash of the bundle essence, which is itself the hash of the following fields of transactions in the bundle:

- `address`,
- `value`,
- `legacy_tag`,
- `current_index`,
- `last_index`,
- and `timestamp`.

The transaction's `signature_message_fragment` field contains the signature generated by signing the bundle hash with the address's private key.

Returns *TryteString* object.

3.12.2 ProposedTransaction

class iota.**ProposedTransaction**(*address: iota.types.Address*, *value: int*, *tag: Optional[iota.types.Tag] = None*, *message: Optional[iota.types.TryteString] = None*, *timestamp: Optional[int] = None*)

A transaction that has not yet been attached to the Tangle.

Proposed transactions are created locally. Note that for creation, only a small subset of the `Transaction` attributes is needed.

Provide to `Iota.send_transfer()` to attach to tangle and publish/store.

Note: In order to follow naming convention of other libs, you may use the name `Transfer` interchangeably with `ProposedTransaction`. See https://github.com/iotaledger/iota.py/issues/72 for more info.

Parameters

- **address** (`Address`) – Address associated with the transaction.
- **value** (*int*) – Transaction value.
- **tag** (*Optional[*`Tag`*]*) – Optional classification tag applied to this transaction.
- **message** (*Optional[*`TryteString`*]*) – Message to be included in `transaction.Transaction.signature_or_message_fragment` field of the transaction. Should not be longer than `transaction.Fragment.LEN`.
- **timestamp** (*Optional[int]*) – Timestamp of transaction creation. If not supplied, the library will generate it.

Returns `iota.ProposedTransaction` object.

Example usage:

```
txn=\
    ProposedTransaction(
        address =
            Address(
                b'TESTVALUE9DONTUSEINPRODUCTION99999XE9IVG'
                b'EFNDOCQCMERGUATCIEGGOHPHGFIAQEZGNHQ9W99CH'
            ),
        message = TryteString.from_unicode('thx fur cheezburgers'),
        tag     = Tag(b'KITTENS'),
        value   = 42,
    )
```

as_tryte_string

`ProposedTransaction.`**`as_tryte_string`**`()` → iota.types.TryteString

Returns a TryteString representation of the transaction.

Returns *`TryteString`* object.

Raises **`RuntimeError`** – if the transaction doesn't have a bundle hash field, meaning that the bundle containing the transaction hasn't been finalized yet.

increment_legacy_tag

`ProposedTransaction.`**`increment_legacy_tag`**`()` → None

Increments the transaction's legacy tag, used to fix insecure bundle hashes when finalizing a bundle.

References:

- https://github.com/iotaledger/iota.py/issues/84

3.13 Bundle Types

As with transactions, PyOTA defines two different types to represent bundles:

- *`Bundle`* for bundles that have already been broadcast to the Tangle. Generally, you will never need to create *`Bundle`* objects; the API will build them for you, as the result of various API methods.
- *`ProposedBundle`* for bundles that have been created locally and have not been broadcast yet.

3.13.1 Bundle

`class` `iota.`**`Bundle`**`(`*transactions: Optional[Iterable[iota.transaction.base.Transaction]] = None*`)`

A collection of transactions, treated as an atomic unit when attached to the Tangle.

Note: unlike a block in a blockchain, bundles are not first-class citizens in IOTA; only transactions get stored in the Tangle.

Instead, Bundles must be inferred by following linked transactions with the same bundle hash.

Parameters **`transactions`** (*`Optional[Iterable[Transaction]]`*) – Transactions in the bundle. Note that transactions will be sorted into ascending order based on their `current_index`.

Returns *`Bundle`* object.

References:

- *`Iota.get_transfers`*

`property hash`

Returns the hash of the bundle.

This value is determined by inspecting the bundle's tail transaction, so in a few edge cases, it may be incorrect.

Returns

- *`BundleHash`* object, or
- If the bundle has no transactions, this method returns `None`.

property is_confirmed
Returns whether this bundle has been confirmed by neighbor nodes.

This attribute must be set manually.

Returns `bool`

References:

- *Iota.get_transfers*

property tail_transaction
Returns the tail transaction of the bundle.

Returns *Transaction*

transactions: List[Transaction] = None
List of *Transaction* objects that are in the bundle.

as_json_compatible

`Bundle.`**as_json_compatible**`()` → List[dict]
Returns a JSON-compatible representation of the object.

Returns `List[dict]`. The `dict` list elements contain individual transactions as in *Transaction.as_json_compatible()*.

References:

- `iota.json.JsonEncoder`.

as_tryte_strings

`Bundle.`**as_tryte_strings**`(`*head_to_tail: bool = False*`)` → List[iota.transaction.types.TransactionTrytes]
Returns TryteString representations of the transactions in this bundle.

Parameters **head_to_tail** (*bool*) – Determines the order of the transactions:

- `True`: head txn first, tail txn last.
- `False` (default): tail txn first, head txn last.

Note that the order is reversed by default, as this is the way bundles are typically broadcast to the Tangle.

Returns `List[TransactionTrytes]`

from_tryte_strings

classmethod `Bundle.`**from_tryte_strings**`(`*trytes: Iterable[iota.types.TryteString]*`)` → B
Creates a Bundle object from a list of tryte values.

Note, that this is effectively calling *Transaction.from_tryte_string()* on the iterbale elements and constructing the bundle from the created transactions.

Parameters **trytes** (*Iterable[TryteString]*) – List of raw transaction trytes.

Returns *Bundle* object.

Example usage:

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

get_messages

`Bundle.`**`get_messages`**(*errors: str = 'drop'*) → List[str]

Attempts to decipher encoded messages from the transactions in the bundle.

Parameters **`errors`** (*`str`*) – How to handle trytes that can't be converted, or bytes that can't be decoded using UTF-8:

'drop' Drop the trytes from the result.

'strict' Raise an exception.

'replace' Replace with a placeholder character.

'ignore' Omit the invalid tryte/byte sequence.

Returns `List[str]`

group_transactions

`Bundle.`**`group_transactions`**() → List[List[iota.transaction.base.Transaction]]

Groups transactions in the bundle by address.

Returns `List[List[Transaction]]`

3.13.2 ProposedBundle

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

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

`class` `iota.`**`ProposedBundle`**(*transactions: Optional[Iterable[iota.transaction.creation.ProposedTransaction]] = None, inputs: Optional[Iterable[iota.types.Address]] = None, change_address: Optional[iota.types.Address] = None*)

A collection of proposed transactions, to be treated as an atomic unit when attached to the Tangle.

Parameters

- **`transactions`** (*`Optional[Iterable[ProposedTransaction]]`*) – Proposed transactions that should be put into the proposed bundle.
- **`inputs`** (*`Optional[Iterable[Address]]`*) – Addresses that hold iotas to fund outgoing transactions in the bundle. If provided, the library will create and sign withdrawing transactions from these addresses.

  See *`Iota.get_inputs()`* for more info.

- **change_address** (*Optional*[Address]) – Due to the signatures scheme of IOTA, you can only spend once from an address. Therefore the library will always deduct the full available amount from an input address. The unused tokens will be sent to `change_address` if provided, or to a newly-generated and unused address if not.

Returns *ProposedBundle*

property balance
Returns the bundle balance. In order for a bundle to be valid, its balance must be 0:

- A positive balance means that there aren't enough inputs to cover the spent amount; add more inputs using *add_inputs()*.
- A negative balance means that there are unspent inputs; use *send_unspent_inputs_to()* to send the unspent inputs to a "change" address.

Returns `bool`

property tag
Determines the most relevant tag for the bundle.

Returns `transaction.Tag`

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

add_transaction

`ProposedBundle.`**add_transaction**(*transaction: iota.transaction.creation.ProposedTransaction*) → None
Adds a transaction to the bundle.

If the transaction message is too long, it will be split automatically into multiple transactions.

Parameters **transaction** (ProposedTransaction) – The transaction to be added.

Raises

- **RuntimeError** – if bundle is already finalized
- **ValueError** – if trying to add a spending transaction. Use *add_inputs()* instead.

add_inputs

`ProposedBundle.`**add_inputs**(*inputs: Iterable[iota.types.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.

Note that each input may require multiple transactions, in order to hold the entire signature.

Parameters **inputs** (*Iterable*[Address]) – Addresses to use as the inputs for this bundle.

Important: Must have `balance` and `key_index` attributes! Use *Iota.get_inputs()* to prepare inputs.

Raises

- **RuntimeError** – if bundle is already finalized.

- **`ValueError`** –
  - if input address has no `balance`.
  - if input address has no `key_index`.

send_unspent_inputs_to

`ProposedBundle.`**`send_unspent_inputs_to`**(*address: iota.types.Address*) → None

Specifies the address that will receive unspent iotas.

The *`ProposedBundle`* will use this to create the necessary change transaction, if necessary.

If the bundle has no unspent inputs, this method does nothing.

Parameters **`address`** (`Address`) – Address to send unspent inputs to.

Raises **`RuntimeError`** – if bundle is already finalized.

add_signature_or_message

`ProposedBundle.`**`add_signature_or_message`**(*fragments: Iterable[iota.transaction.types.Fragment], start_index: Optional[int] = 0*) → None

Adds signature/message fragments to transactions in the bundle starting at start_index. If a transaction already has a fragment, it will be overwritten.

Parameters

- **`fragments`** (*`Iterable[Fragment]`*) – List of fragments to add. Use [Fragment(. . .),Fragment(. . .),. . .] to create this argument. Fragment() accepts any TryteString compatible type, or types that can be converted to TryteStrings (bytearray, unicode string, etc.). If the payload is less than `FRAGMENT_LENGTH`, it will pad it with 9s.
- **`start_index`** (*`int`*) – Index of transaction in bundle from where addition shoudl start.

Raises

- **`RuntimeError`** – if bundle is already finalized.
- **`ValueError`** –
  - if empty list is provided for `fragments`
  - if wrong `start_index` is provided.
  - if `fragments` is too long and does't fit into the bundle.
- **`TypeError`** –
  - if `fragments` is not an `Iterable`
  - if `fragments` contains other types than *`Fragment`*.

finalize

`ProposedBundle.`**`finalize`**`()` → None

Finalizes the bundle, preparing it to be attached to the Tangle.

This operation includes checking if the bundle has zero balance, generating the bundle hash and updating the transactions with it, furthermore to initialize signature/message fragment fields.

Once this method is invoked, no new transactions may be added to the bundle.

Raises

- **`RuntimeError`** – if bundle is already finalized.
- **`ValueError`** –
  - if bundle has no transactions.
  - if bundle has unspent inputs (there is no `change_address` attribute specified.)
  - if inputs are insufficient to cover bundle spend.

sign_inputs

`ProposedBundle.`**`sign_inputs`**`(`*key_generator: iota.crypto.signing.KeyGenerator*`)` → None

Sign inputs in a finalized bundle.

Generates the necessary cryptographic signatures to authorize spending the inputs.

Note: You do not need to invoke this method if the bundle does not contain any transactions that spend iotas.

Parameters **`key_generator`** (*`KeyGenerator`*) – Generator to create private keys for signing.

Raises

- **`RuntimeError`** – if bundle is not yet finalized.
- **`ValueError`** –
  - if the input transaction specifies an address that doesn't have `key_index` attribute defined.
  - if the input transaction specifies an address that doesn't have `security_level` attribute defined.

sign_input_at

`ProposedBundle.`**`sign_input_at`**`(`*start_index: int*, *private_key: iota.crypto.types.PrivateKey*`)` → None

Signs the input at the specified index.

Parameters

- **`start_index`** (*`int`*) – The index of the first input transaction.

  If necessary, the resulting signature will be split across multiple transactions automatically (i.e., if an input has `security_level=2`, you still only need to call *`sign_input_at()`* once).

- **private_key** (*PrivateKey*) – The private key that will be used to generate the signature.

  Important: Be sure that the private key was generated using the correct seed, or the resulting signature will be invalid!

Raises **RuntimeError** – if bundle is not yet finalized.

as_json_compatible

ProposedBundle.**as_json_compatible**() → List[dict]
Returns a JSON-compatible representation of the object.

Returns List[dict]. The dict list elements contain individual transactions as in ProposedTransaction.as_json_compatible().

References:

- iota.json.JsonEncoder.

Example usage

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

bundle = ProposedBundle()

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'))
```

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

CHAPTER

FOUR

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*.

The API instance's methods contain the logic and handle PyOTA-specific types, construct and translate objects, while the API instance's adapter deals with the networking, communicating with a node.

You can choose and configure the available adapters to be used with the API:

- HttpAdapter,
- MockAdapter.

4.1 AdapterSpec

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

`iota.adapter.`**`AdapterSpec = typing.Union[str, ForwardRef('BaseAdapter')]`**

Placeholder that means "URI or adapter instance".

Will be resolved to a correctly-configured adapter instance upon API instance creation.

For example, when creating an `Iota` object, 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:

```
api = Iota('http://localhost:14265')
```

- Adapter instance:

```
api = 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, HttpAdapter

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

# Use HTTPS:
api = Iota('https://nodes.thetangle.org:443')
api = Iota(HttpAdapter('https://nodes.thetangle.org:443'))

# Use HTTPS with basic authentication and 60 seconds timeout:
api = Iota(
    HttpAdapter(
        'https://nodes.thetangle.org:443',
        authentication=('myusername', 'mypassword'),
        timeout=60))
```

class `iota.`**`HttpAdapter`**(*uri: Union[str, urllib.parse.SplitResult], timeout: Optional[int] = None, authentication: Optional[Tuple[str, str]] = None*)

Sends standard HTTP(S) requests to the node.

Parameters

- **`uri`** (*`AdapterSpec`*) – URI or adapter instance.

  If `uri` is a `str`, it is parsed to extract `scheme`, `hostname` and `port`.
- **`timeout`** (*`Optional[int]`*) – Connection timeout in seconds.
- **`authentication`** (*`Optional[Tuple(str,str)]`*) – Credetentials for basic authentication with the node.

Returns *`HttpAdapter`* object.

Raises `InvalidUri` –

- if protocol is unsupported.
- if hostname is empty.
- if non-numeric port is supplied.

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

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!

Logging to console with default format

```
from logging import getLogger, basicConfig, DEBUG
from iota import Iota

api = Iota("https://nodes.thetangle.org:443")

# Sets the logging level for the root logger (and for its handlers)
basicConfig(level=DEBUG)

# Get a new logger derived from the root logger
logger = getLogger(__name__)

# Attach the logger to the adapter
api.adapter.set_logger(logger)

# Execute a command that sends request to the node
api.get_node_info()

# Log messages should be printed to console
```

Logging to a file with custom format

```
from logging import getLogger, DEBUG, FileHandler, Formatter
from iota import Iota

# Create a custom logger
logger = getLogger(__name__)

# Set logging level to DEBUG
logger.setLevel(DEBUG)

# Create handler to write to a log file
f_handler = FileHandler(filename='pyota.log',mode='a')
f_handler.setLevel(DEBUG)

# Create formatter and add it to handler
f_format = Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
f_handler.setFormatter(f_format)

# Add handler to the logger
logger.addHandler(f_handler)

# Create API instance
api = Iota("https://nodes.thetangle.org:443")

# Add logger to the adapter of the API instance
api.adapter.set_logger(logger)

# Sends a request to the node
api.get_node_info()

# Open 'pyota.log' file and observe the logs
```

Logging to console with custom format

```
from logging import getLogger, DEBUG, StreamHandler, Formatter
from iota import Iota

# Create a custom logger
logger = getLogger(__name__)

# Set logging level to DEBUG
logger.setLevel(DEBUG)

# Create handler to write to sys.stderr
s_handler = StreamHandler()
s_handler.setLevel(DEBUG)

# Create formatter and add it to handler
s_format = Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
s_handler.setFormatter(s_format)

# Add handler to the logger
logger.addHandler(s_handler)

# Create API instance
api = Iota("https://nodes.thetangle.org:443")

# Add logger to the adapter of the API instance
api.adapter.set_logger(logger)

# Sends a request to the node
api.get_node_info()

# Observe log messages in console
```

4.2.2 MockAdapter

```
from iota import Iota, 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
```

class iota.**MockAdapter**

A mock adapter used for simulating API responses 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 use this adapter, you must first “seed” the responses that the adapter should return for each request. The adapter will then return the appropriate seeded response each time it "sends" a request.

Parameters **`None`** – To construct a `MockAdapter`, you don't need to supply any arguments.

Returns *`MockAdapter`* object.

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

Example usage:

```
from iota import Iota, MockAdapter

# Create API with a mock adapter.
api = Iota('mock://')
api = Iota(MockAdapter())
```

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

seed_response

`MockAdapter.`**`seed_response`**(*command: str*, *response: dict*) → iota.adapter.MockAdapter

Sets the response that the adapter will return for the specified command.

You can seed multiple responses per command; the adapter will put them into a FIFO queue. When a request comes in, the adapter will pop the corresponding response off of the queue.

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).

Parameters

- **`command`** (*`str`*) – 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.

Example usage:

```
adapter.seed_response('sayHello', {'message': 'Hi!'})
adapter.seed_response('sayHello', {'message': 'Hello!'})

adapter.send_request({'command': 'sayHello'})
# {'message': 'Hi!'}

adapter.send_request({'command': 'sayHello'})
# {'message': 'Hello!'}
```

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

class iota.adapter.wrappers.**RoutingWrapper**(*default_adapter: Union[str, BaseAdapter]*)

Routes commands (API requests) to different nodes depending on the command name.

This allows you to, for example, send POW requests to a local node, while routing all other requests to a remote one.

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

Parameters **default_adapter** (*AdapterSpec*) – 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.

Returns *RoutingWrapper* object.

Example usage:

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

# Route POW to localhost, everything else to 'https://nodes.thetangle.org:443'.
api = Iota(
  RoutingWrapper('https://nodes.thetangle.org:443.'')
    .add_route('attachToTangle', 'http://localhost:14265')
    .add_route('interruptAttachingToTangle', 'http://localhost:14265')
)
```

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 (see *iota.Iota*), because the behavior might not be obvious.

`local_pow` tells the API to perform proof-of-work (*iota.Iota.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*.

add_route

RoutingWrapper.**add_route**(*command: str, adapter: Union[str, BaseAdapter]*) → iota.adapter.wrappers.RoutingWrapper

Adds a route to the wrapper.

Parameters

- **command** (*str*) – The name of the command. Note that this is the camelCase version of the command name (e.g., `attachToTangle`, not `attach_to_tangle`).
- **adapter** (*AdapterSpec*) – The adapter object or URI to route requests to.

Returns The `RoutingWrapper` object it was called on. Useful for chaining the operation of adding routes in code.

See `RoutingWrapper` for example usage.

CHAPTER

FIVE

PYOTA API CLASSES

PyOTA offers you the Python API to interact with the IOTA network. The available methods can be grouped into two categories:

Core API	Extended API
API commands for direct interaction with a node.	Builds on top of the Core API to perform more complex operations, and abstract away low-level IOTA specific procedures.

PyOTA supports both synchronous and asynchronous communication with the network, therefore the Core and Extended API classes are available in synchronous and asynchronous versions.

To use the API in your Python application or script, declare an API instance of any of the API classes. **Since the Extended API incorporates the Core API, usually you end up only using the Extended API,** but if for some reason you need only the core functionality, the library is there to help you.

```
# Synchronous API classes
from iota import Iota, StrictIota

# This is how you declare a sync Extended API, use the methods of this object.
api = Iota('adapter-specification')

# This is how you declare a sync Core API, use the methods of this object.
api = StrictIota('adapter-specification')
```

The PyOTA speific `StrictIota` class implements the Core API, while `Iota` implements the Extended API. From a Python implementation point of view, `Iota` is a subclass of `StrictIota`, therefore it inherits every method and attribute the latter has.

To use the functionally same, but asynchronous API classes, you can do the following:

```
# Asynchronous API classes
from iota import AsyncIota, AsyncStrictIota

# This is how you declare an async Extended API, use the methods of this object.
api = AsyncIota('adapter-specification')

# This is how you declare an async  Core API, use the methods of this object.
api = AsyncStrictIota('adapter-specification')
```

Take a look on the class definitions and notice that `Iota` and `AsyncIota` have a `Seed` attribute. This is because the Extended API is able to generate private keys, addresses and signatures from your seed. **Your seed never leaves the library and your machine!**

5.1 Core API Classes

5.1.1 Synchronous

class iota.**StrictIota**(*adapter: Union[str, BaseAdapter], devnet: bool = False, local_pow: bool = False*)

Synchronous API to send HTTP requests for communicating with an IOTA node.

This implementation only exposes the "core" API methods. For a more feature-complete implementation, use `Iota` instead.

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference

Parameters

- **adapter** (*AdapterSpec*) – URI string or BaseAdapter instance.
- **devnet** (*Optional[bool]*) – Whether to use devnet settings for this instance. On the devnet, minimum weight magnitude is set to 9, on mainnet it is 1 by default.
- **local_pow** (*Optional[bool]*) – Whether to perform proof-of-work locally by redirecting all calls to `attach_to_tangle()` to ccurl pow interface.

  See README:Optional Local Pow for more info and *find out* how to use it.

set_local_pow(*local_pow: bool*) → None

Sets the `local_pow` attribute of the adapter of the api instance. If it is `True`, `attach_to_tangle()` command calls external interface to perform proof of work, instead of sending the request to a node.

By default, `local_pow` is set to `False`. This particular method is needed if one wants to change local_pow behavior dynamically.

Parameters **local_pow** (*bool*) – Whether to perform pow locally.

Returns None

5.1.2 Asynchronous

class iota.**AsyncStrictIota**(*adapter: Union[str, BaseAdapter], devnet: bool = False, local_pow: bool = False*)

Asynchronous API to send HTTP requests for communicating with an IOTA node.

This implementation only exposes the "core" API methods. For a more feature-complete implementation, use `AsyncIota` instead.

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference

Parameters

- **adapter** (*AdapterSpec*) – URI string or BaseAdapter instance.

- **devnet** (*Optional[bool]*) – Whether to use devnet settings for this instance. On the devnet, minimum weight magnitude is set to 9, on mainnet it is 1 by default.
- **local_pow** (*Optional[bool]*) – Whether to perform proof-of-work locally by redirecting all calls to `attach_to_tangle()` to ccurl pow interface.

  See README:Optional Local Pow for more info and *find out* how to use it.

set_local_pow(*local_pow: bool*) → None

Sets the `local_pow` attribute of the adapter of the api instance. If it is `True`, `attach_to_tangle()` command calls external interface to perform proof of work, instead of sending the request to a node.

By default, `local_pow` is set to `False`. This particular method is needed if one wants to change local_pow behavior dynamically.

Parameters **local_pow** (*bool*) – Whether to perform pow locally.

Returns None

5.2 Extended API Classes

5.2.1 Synchronous

class iota.**Iota**(*adapter: Union[str, BaseAdapter], seed: Union[AnyStr, bytearray, TryteString, None] = None, devnet: bool = False, local_pow: bool = False*)

Implements the synchronous core API, plus additional synchronous wrapper methods for common operations.

Parameters

- **adapter** (*AdapterSpec*) – URI string or BaseAdapter instance.
- **seed** (*Optional[Seed]*) – Seed used to generate new addresses. If not provided, a random one will be generated.

  Note: This value is never transferred to the node/network.

- **devnet** (*Optional[bool]*) – Whether to use devnet settings for this instance. On the devnet, minimum weight magnitude is decreased, on mainnet it is 14 by default.

  For more info on the Mainnet and the Devnet, visit *the official docs site<https://docs.iota.org/docs/getting-started/0.1/network/iota-networks/>*.
- **local_pow** (*Optional[bool]*) – Whether to perform proof-of-work locally by redirecting all calls to `attach_to_tangle()` to ccurl pow interface.

  See README:Optional Local Pow for more info and *find out* how to use it.

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference
- https://github.com/iotaledger/wiki/blob/master/api-proposal.md

set_local_pow(*local_pow: bool*) → None

Sets the `local_pow` attribute of the adapter of the api instance. If it is `True`, `attach_to_tangle()` command calls external interface to perform proof of work, instead of sending the request to a node.

By default, `local_pow` is set to `False`. This particular method is needed if one wants to change local_pow behavior dynamically.

Parameters **`local_pow`** (*`bool`*) – Whether to perform pow locally.

Returns None

5.2.2 Asynchronous

class `iota.`**`AsyncIota`**(*adapter: Union[str, BaseAdapter], seed: Union[AnyStr, bytearray, TryteString, None] = None, devnet: bool = False, local_pow: bool = False*)

Implements the async core API, plus additional async wrapper methods for common operations.

Parameters

- **`adapter`** (*`AdapterSpec`*) – URI string or BaseAdapter instance.
- **`seed`** (*`Optional[Seed]`*) – Seed used to generate new addresses. If not provided, a random one will be generated.

  Note: This value is never transferred to the node/network.

- **`devnet`** (*`Optional[bool]`*) – Whether to use devnet settings for this instance. On the devnet, minimum weight magnitude is decreased, on mainnet it is 14 by default.

  For more info on the Mainnet and the Devnet, visit *the official docs site<https://docs.iota.org/docs/getting-started/0.1/network/iota-networks/>*.

- **`local_pow`** (*`Optional[bool]`*) – Whether to perform proof-of-work locally by redirecting all calls to *`attach_to_tangle()`* to ccurl pow interface.

  See README:Optional Local Pow for more info and *find out* how to use it.

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference
- https://github.com/iotaledger/wiki/blob/master/api-proposal.md

`set_local_pow`(*local_pow: bool*) → None

Sets the `local_pow` attribute of the adapter of the api instance. If it is `True`, *`attach_to_tangle()`* command calls external interface to perform proof of work, instead of sending the request to a node.

By default, `local_pow` is set to `False`. This particular method is needed if one wants to change local_pow behavior dynamically.

Parameters **`local_pow`** (*`bool`*) – Whether to perform pow locally.

Returns None

CHAPTER

SIX

CORE API METHODS

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.

Note: Below you will find the documentation for both the synchronous and asynchronous versions of the Core API methods.

It should be made clear, that they do exactly the same IOTA related operations, accept the same arguments and return the same structures. Asynchronous API calls are non-blocking, so your application can do other stuff while waiting for the result from the network.

While synchronous API calls are regular Python methods, their respective asynchronous versions are Python coroutines. You can `await` their results, schedule them for execution inside and event loop and much more. PyOTA uses the built-in asyncio Python module for asynchronous operation. For an overview of what you can do with it, head over to this article.

6.1 add_neighbors

`Iota.`**`add_neighbors`**(*uris: Iterable[str]*) → dict

Add one or more neighbors to the node. Lasts until the node is restarted.

Parameters **`uris`** (*`Iterable[str]`*) – Use format `<protocol>://<ip address>:<port>`. Example: `add_neighbors(['udp://example.com:14265'])`

Note: These URIs are for node-to-node communication (e.g., weird things will happen if you specify a node's HTTP API URI here).

Returns

`dict` with the following structure:

```
{
    'addedNeighbors': int,
        Total number of added neighbors.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#addneighbors

async `AsyncIota.`**`add_neighbors`**(*uris: Iterable[str]*) → dict

Add one or more neighbors to the node. Lasts until the node is restarted.

Parameters **`uris`** (*`Iterable[str]`*) – Use format `<protocol>://<ip address>:<port>`. Example: `add_neighbors(['udp://example.com:14265'])`

Note: These URIs are for node-to-node communication (e.g., weird things will happen if you specify a node's HTTP API URI here).

Returns

`dict` with the following structure:

```
{
    'addedNeighbors': int,
        Total number of added neighbors.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#addneighbors

6.2 attach_to_tangle

`Iota.`**`attach_to_tangle`**(*trunk_transaction: iota.transaction.types.TransactionHash, branch_transaction: iota.transaction.types.TransactionHash, trytes: Iterable[iota.types.TryteString], min_weight_magnitude: Optional[int] = None*) → dict

Attaches the specified transactions (trytes) to the Tangle by doing Proof of Work. You need to supply branchTransaction as well as trunkTransaction (basically the tips which you're going to validate and reference with this transaction) - both of which you'll get through the *`get_transactions_to_approve()`* API call.

The returned value is a different set of tryte values which you can input into *`broadcast_transactions()`* and *`store_transactions()`*.

Parameters

- **`trunk_transaction`** (`TransactionHash`) – Trunk transaction hash.
- **`branch_transaction`** (`TransactionHash`) – Branch transaction hash.
- **`trytes`** (*`Iterable[TransactionTrytes]`*) – List of transaction trytes in the bundle to be attached.
- **`min_weight_magnitude`** (*`Optional[int]`*) – Minimum weight magnitude to be used for attaching trytes. 14 by default on mainnet, 9 on devnet/devnet.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Transaction trytes that include a valid nonce field.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#attachtotangle

async AsyncIota.**attach_to_tangle**(*trunk_transaction: iota.transaction.types.TransactionHash, branch_transaction: iota.transaction.types.TransactionHash, trytes: Iterable[iota.types.TryteString], min_weight_magnitude: Optional[int] = None*) → dict

Attaches the specified transactions (trytes) to the Tangle by doing Proof of Work. You need to supply branchTransaction as well as trunkTransaction (basically the tips which you're going to validate and reference with this transaction) - both of which you'll get through the *get_transactions_to_approve()* API call.

The returned value is a different set of tryte values which you can input into *broadcast_transactions()* and *store_transactions()*.

Parameters

- **trunk_transaction** (TransactionHash) – Trunk transaction hash.
- **branch_transaction** (TransactionHash) – Branch transaction hash.
- **trytes** (*Iterable[TransactionTrytes]*) – List of transaction trytes in the bundle to be attached.
- **min_weight_magnitude** (*Optional[int]*) – Minimum weight magnitude to be used for attaching trytes. 14 by default on mainnet, 9 on devnet/devnet.

Returns

dict with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Transaction trytes that include a valid nonce field.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#attachtotangle

6.3 broadcast_transactions

Iota.**broadcast_transactions**(*trytes: Iterable[iota.types.TryteString]*) → dict

Broadcast a list of transactions to all neighbors.

The input trytes for this call are provided by *attach_to_tangle()*.

Parameters trytes (*Iterable[TransactionTrytes]*) – List of transaction trytes to be broadcast.

Returns

dict with the following structure:

```
{
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#broadcasttransactions

async AsyncIota.**broadcast_transactions**(*trytes: Iterable[iota.types.TryteString]*) → dict

Broadcast a list of transactions to all neighbors.

The input trytes for this call are provided by *attach_to_tangle()*.

Parameters trytes (*Iterable[TransactionTrytes]*) – List of transaction trytes to be broadcast.

Returns

`dict` with the following structure:

```
{
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#broadcasttransactions

6.4 check_consistency

Iota.**check_consistency**(*tails: Iterable[iota.transaction.types.TransactionHash]*) → dict

Used to ensure tail resolves to a consistent ledger which is necessary to validate before attempting promotion. Checks transaction hashes for promotability.

This is called with a pending transaction (or more of them) and it will tell you if it is still possible for this transaction (or all the transactions simultaneously if you give more than one) to be confirmed, or not (because it conflicts with another already confirmed transaction).

Parameters tails (*Iterable[TransactionHash]*) – Transaction hashes. Must be tail transactions.

Returns

`dict` with the following structure:

```
{
    'state': bool,
        Whether tails resolve to consistent ledger.
    'info': str,
        This field will only exist if 'state' is ``False``.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#checkconsistency

async AsyncIota.**check_consistency**(*tails: Iterable[iota.transaction.types.TransactionHash]*) → dict

Used to ensure tail resolves to a consistent ledger which is necessary to validate before attempting promotion. Checks transaction hashes for promotability.

This is called with a pending transaction (or more of them) and it will tell you if it is still possible for this transaction (or all the transactions simultaneously if you give more than one) to be confirmed, or not (because it conflicts with another already confirmed transaction).

Parameters **tails** (*Iterable[TransactionHash]*) – Transaction hashes. Must be tail transactions.

Returns

`dict` with the following structure:

```
{
    'state': bool,
        Whether tails resolve to consistent ledger.
    'info': str,
        This field will only exist if 'state' is ``False``.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#checkconsistency

6.5 find_transactions

Iota.**find_transactions**(*bundles: Optional[Iterable[iota.transaction.types.BundleHash]] = None, addresses: Optional[Iterable[iota.types.Address]] = None, tags: Optional[Iterable[iota.types.Tag]] = None, approvees: Optional[Iterable[iota.transaction.types.TransactionHash]] = None*) → dict

Find the transactions which match the specified input and return.

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.

Parameters

- **bundles** (*Optional[Iterable[BundleHash]]*) – List of bundle IDs.
- **addresses** (*Optional[Iterable[Address]]*) – List of addresses.
- **tags** (*Optional[Iterable[Tag]]*) – List of tags.
- **approvees** (*Optional[Iterable[TransactionHash]]*) – List of approvee transaction IDs.

Returns

`dict` with the following structure:

```
{
    'hashes': List[TransationHash],
        Found transactions.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#findtransactions

async AsyncIota.**find_transactions**(*bundles: Optional[Iterable[iota.transaction.types.BundleHash]] = None*, *addresses: Optional[Iterable[iota.types.Address]] = None*, *tags: Optional[Iterable[iota.types.Tag]] = None*, *approvees: Optional[Iterable[iota.transaction.types.TransactionHash]] = None*) → dict

Find the transactions which match the specified input and return.

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.

Parameters

- **bundles** (*Optional[Iterable[BundleHash]]*) – List of bundle IDs.
- **addresses** (*Optional[Iterable[Address]]*) – List of addresses.
- **tags** (*Optional[Iterable[Tag]]*) – List of tags.
- **approvees** (*Optional[Iterable[TransactionHash]]*) – List of approvee transaction IDs.

Returns

`dict` with the following structure:

```
{
    'hashes': List[TransationHash],
        Found transactions.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#findtransactions

6.6 get_balances

Iota.**get_balances**(*addresses: Iterable[iota.types.Address]*, *tips: Optional[Iterable[iota.transaction.types.TransactionHash]] = None*) → dict

Returns the confirmed balance which a list of addresses have at the latest confirmed milestone.

In addition to the balances, it also returns the milestone as well as the index with which the confirmed balance was determined. The balances are returned as a list in the same order as the addresses were provided as input.

Parameters

- **addresses** (*Iterable[Address]*) – List of addresses to get the confirmed balance for.
- **tips** (*Optional[Iterable[TransactionHash]]*) – Tips whose history of transactions to traverse to find the balance.

Returns

`dict` with the following structure:

```
{
    'balances': List[int],
        List of balances in the same order as the addresses
        parameters that were passed to the endpoint.
    'references': List[TransactionHash],
        The referencing tips. If no tips parameter was passed
        to the endpoint, this field contains the hash of the
        latest milestone that confirmed the balance.
    'milestoneIndex': int,
        The index of the milestone that confirmed the most
        recent balance.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getbalances

async AsyncIota.**get_balances**(*addresses: Iterable[iota.types.Address], tips: Optional[Iterable[iota.transaction.types.TransactionHash]] = None*) → dict

Returns the confirmed balance which a list of addresses have at the latest confirmed milestone.

In addition to the balances, it also returns the milestone as well as the index with which the confirmed balance was determined. The balances are returned as a list in the same order as the addresses were provided as input.

Parameters

- **addresses** (*Iterable[Address]*) – List of addresses to get the confirmed balance for.
- **tips** (*Optional[Iterable[TransactionHash]]*) – Tips whose history of transactions to traverse to find the balance.

Returns

`dict` with the following structure:

```
{
    'balances': List[int],
        List of balances in the same order as the addresses
        parameters that were passed to the endpoint.
    'references': List[TransactionHash],
        The referencing tips. If no tips parameter was passed
        to the endpoint, this field contains the hash of the
        latest milestone that confirmed the balance.
    'milestoneIndex': int,
        The index of the milestone that confirmed the most
        recent balance.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getbalances

6.7 `get_inclusion_states`

`Iota.`**`get_inclusion_states`**(*transactions: Iterable[iota.transaction.types.TransactionHash]*) → dict

Get the inclusion states of a set of transactions. This is for determining if a transaction was accepted and confirmed by the network or not.

Parameters **`transactions`** (*`Iterable[TransactionHash]`*) – List of transactions you want to get the inclusion state for.

Returns

`dict` with the following structure:

```
{
    'states': List[bool],
        List of boolean values in the same order as the
        transactions parameters. A ``True`` value means the
        transaction was confirmed.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getinclusionstates

`async` `AsyncIota.`**`get_inclusion_states`**(*transactions: Iterable[iota.transaction.types.TransactionHash]*) → dict

Get the inclusion states of a set of transactions. This is for determining if a transaction was accepted and confirmed by the network or not.

Parameters **`transactions`** (*`Iterable[TransactionHash]`*) – List of transactions you want to get the inclusion state for.

Returns

`dict` with the following structure:

```
{
    'states': List[bool],
        List of boolean values in the same order as the
        transactions parameters. A ``True`` value means the
        transaction was confirmed.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getinclusionstates

6.8 `get_missing_transactions`

`Iota.get_missing_transactions()` → dict

Returns all transaction hashes that a node is currently requesting from its neighbors.

Returns

`dict` with the following structure:

```
{
    'hashes': List[TransactionHash],
        Array of missing transaction hashes.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getmissingtransactions

async `AsyncIota.get_missing_transactions()` → dict

Returns all transaction hashes that a node is currently requesting from its neighbors.

Returns

`dict` with the following structure:

```
{
    'hashes': List[TransactionHash],
        Array of missing transaction hashes.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getmissingtransactions

6.9 `get_neighbors`

`Iota.get_neighbors()` → dict

Returns the set of neighbors the node is connected with, as well as their activity count.

The activity counter is reset after restarting IRI.

Returns

`dict` with the following structure:

```
{
    'neighbors': List[dict],
        Array of objects, including the following fields with
        example values:
            "address": "/8.8.8.8:14265",
            "numberOfAllTransactions": 158,
            "numberOfRandomTransactionRequests": 271,
            "numberOfNewTransactions": 956,
            "numberOfInvalidTransactions": 539,
```

(continues on next page)

(continued from previous page)

```
                "numberOfStaleTransactions": 663,
                "numberOfSentTransactions": 672,
                "connectiontype": "TCP"
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getneighbors

async AsyncIota.**get_neighbors**() → dict

Returns the set of neighbors the node is connected with, as well as their activity count.

The activity counter is reset after restarting IRI.

Returns

`dict` with the following structure:

```
{
    'neighbors': List[dict],
        Array of objects, including the following fields with
        example values:
            "address": "/8.8.8.8:14265",
            "numberOfAllTransactions": 158,
            "numberOfRandomTransactionRequests": 271,
            "numberOfNewTransactions": 956,
            "numberOfInvalidTransactions": 539,
            "numberOfStaleTransactions": 663,
            "numberOfSentTransactions": 672,
            "connectiontype": "TCP"
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getneighbors

6.10 `get_node_api_configuration`

Iota.**get_node_api_configuration**() → dict

Returns a node's API configuration settings.

Returns

`dict` with the following structure:

```
{
    '<API-config-settings>': type,
        Configuration parameters for a node.
    ...
    ...
    ...

}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/iri-configuration-options
- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getnodeapiconfiguration

async `AsyncIota.`**`get_node_api_configuration`**`()` → dict

Returns a node's API configuration settings.

Returns

`dict` with the following structure:

```
{
    '<API-config-settings>': type,
        Configuration parameters for a node.
    ...
    ...
    ...

}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/iri-configuration-options
- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getnodeapiconfiguration

6.11 get_node_info

`Iota.`**`get_node_info`**`()` → dict

Returns information about the node.

Returns

`dict` with the following structure:

```
{
    'appName': str,
        Name of the IRI network.
    'appVersion': str,
        Version of the IRI.
    'jreAvailableProcessors': int,
        Available CPU cores on the node.
    'jreFreeMemory': int,
        Amount of free memory in the Java virtual machine.
    'jreMaxMemory': int,
        Maximum amount of memory that the Java virtual machine
        can use,
    'jreTotalMemory': int,
        Total amount of memory in the Java virtual machine.
    'jreVersion': str,
        The version of the Java runtime environment.
    'latestMilestone': TransactionHash
        Transaction hash of the latest milestone.
    'latestMilestoneIndex': int,
        Index of the latest milestone.
    'latestSolidSubtangleMilestone': TransactionHash,
        Transaction hash of the latest solid milestone.
```

(continues on next page)

(continued from previous page)

```
    'latestSolidSubtangleMilestoneIndex': int,
        Index of the latest solid milestone.
    'milestoneStartIndex': int,
        Start milestone for the current version of the IRI.
    'neighbors': int,
        Total number of connected neighbor nodes.
    'packetsQueueSize': int,
        Size of the packet queue.
    'time': int,
        Current UNIX timestamp.
    'tips': int,
        Number of tips in the network.
    'transactionsToRequest': int,
        Total number of transactions that the node is missing in
        its ledger.
    'features': List[str],
        Enabled configuration options.
    'coordinatorAddress': Address,
        Address (Merkle root) of the Coordinator.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getnodeinfo

async AsyncIota.**get_node_info**() → dict

Returns information about the node.

Returns

`dict` with the following structure:

```
{
    'appName': str,
        Name of the IRI network.
    'appVersion': str,
        Version of the IRI.
    'jreAvailableProcessors': int,
        Available CPU cores on the node.
    'jreFreeMemory': int,
        Amount of free memory in the Java virtual machine.
    'jreMaxMemory': int,
        Maximum amount of memory that the Java virtual machine
        can use,
    'jreTotalMemory': int,
        Total amount of memory in the Java virtual machine.
    'jreVersion': str,
        The version of the Java runtime environment.
    'latestMilestone': TransactionHash
        Transaction hash of the latest milestone.
    'latestMilestoneIndex': int,
        Index of the latest milestone.
    'latestSolidSubtangleMilestone': TransactionHash,
        Transaction hash of the latest solid milestone.
    'latestSolidSubtangleMilestoneIndex': int,
        Index of the latest solid milestone.
```

(continues on next page)

(continued from previous page)

```
    'milestoneStartIndex': int,
        Start milestone for the current version of the IRI.
    'neighbors': int,
        Total number of connected neighbor nodes.
    'packetsQueueSize': int,
        Size of the packet queue.
    'time': int,
        Current UNIX timestamp.
    'tips': int,
        Number of tips in the network.
    'transactionsToRequest': int,
        Total number of transactions that the node is missing in
        its ledger.
    'features': List[str],
        Enabled configuration options.
    'coordinatorAddress': Address,
        Address (Merkle root) of the Coordinator.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getnodeinfo

6.12 `get_transactions_to_approve`

`Iota.`**`get_transactions_to_approve`**(*depth: int*, *reference: Optional[iota.transaction.types.TransactionHash] = None*) → dict

Tip selection which returns `trunkTransaction` and `branchTransaction`.

Parameters

- **`depth`** (*`int`*) – Number of milestones to go back to start the tip selection algorithm.

  The higher the depth value, the more "babysitting" the node will perform for the network (as it will confirm more transactions that way).

- **`reference`** (`TransactionHash`) – Transaction hash from which to start the weighted random walk. Use this parameter to make sure the returned tip transaction hashes approve a given reference transaction.

Returns

`dict` with the following structure:

```
{
    'trunkTransaction': TransactionHash,
        Valid trunk transaction hash.
    'branchTransaction': TransactionHash,
        Valid branch transaction hash.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#gettransactionstoapprove

async AsyncIota.**get_transactions_to_approve**(*depth: int*, *reference: Optional[iota.transaction.types.TransactionHash] = None*) → dict

Tip selection which returns `trunkTransaction` and `branchTransaction`.

Parameters

- **depth** (*int*) – Number of milestones to go back to start the tip selection algorithm.

  The higher the depth value, the more "babysitting" the node will perform for the network (as it will confirm more transactions that way).

- **reference** (`TransactionHash`) – Transaction hash from which to start the weighted random walk. Use this parameter to make sure the returned tip transaction hashes approve a given reference transaction.

Returns

`dict` with the following structure:

```
{
    'trunkTransaction': TransactionHash,
        Valid trunk transaction hash.
    'branchTransaction': TransactionHash,
        Valid branch transaction hash.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#gettransactionstoapprove

6.13 get_trytes

Iota.**get_trytes**(*hashes: Iterable[iota.transaction.types.TransactionHash]*) → dict

Returns the raw transaction data (trytes) of one or more transactions.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        List of transaction trytes for the given transaction
        hashes (in the same order as the parameters).
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

Note: If a node doesn't have the trytes for a given transaction hash in its ledger, the value at the index of that transaction hash is either `null` or a string of 9s.

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#gettrytes

async AsyncIota.**get_trytes**(*hashes: Iterable[iota.transaction.types.TransactionHash]*) → dict

Returns the raw transaction data (trytes) of one or more transactions.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        List of transaction trytes for the given transaction
        hashes (in the same order as the parameters).
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

Note: If a node doesn't have the trytes for a given transaction hash in its ledger, the value at the index of that transaction hash is either `null` or a string of 9s.

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#gettrytes

6.14 interrupt_attaching_to_tangle

Iota.**interrupt_attaching_to_tangle**() → dict

Interrupts and completely aborts the *attach_to_tangle()* process.

Returns

`dict` with the following structure:

```
{
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#interruptattachingtotangle

async AsyncIota.**interrupt_attaching_to_tangle**() → dict

Interrupts and completely aborts the *attach_to_tangle()* process.

Returns

`dict` with the following structure:

```
{
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#interruptattachingtotangle

6.15 `remove_neighbors`

`Iota.`**`remove_neighbors`**(*uris: Iterable[str]*) → dict

Removes one or more neighbors from the node. Lasts until the node is restarted.

Parameters **`uris`** (*`str`*) – Use format `<protocol>://<ip address>:<port>`. Example: *remove_neighbors(['udp://example.com:14265'])*

Returns

`dict` with the following structure:

```
{
    'removedNeighbors': int,
        Total number of removed neighbors.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#removeneighbors

`async` `AsyncIota.`**`remove_neighbors`**(*uris: Iterable[str]*) → dict

Removes one or more neighbors from the node. Lasts until the node is restarted.

Parameters **`uris`** (*`str`*) – Use format `<protocol>://<ip address>:<port>`. Example: *remove_neighbors(['udp://example.com:14265'])*

Returns

`dict` with the following structure:

```
{
    'removedNeighbors': int,
        Total number of removed neighbors.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#removeneighbors

6.16 `store_transactions`

`Iota.`**`store_transactions`**(*trytes: Iterable[iota.types.TryteString]*) → dict

Store transactions into local storage of the node.

The input trytes for this call are provided by *`attach_to_tangle()`*.

Parameters **`trytes`** (`TransactionTrytes`) – Valid transaction trytes returned by *`attach_to_tangle()`*.

Returns

`dict` with the following structure:

```
{
    'trytes': TransactionTrytes,
        Stored trytes.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#storetransactions

async AsyncIota.**store_transactions**(*trytes: Iterable[iota.types.TryteString]*) → dict
Store transactions into local storage of the node.

The input trytes for this call are provided by *attach_to_tangle()*.

Parameters trytes (TransactionTrytes) – Valid transaction trytes returned by *attach_to_tangle()*.

Returns

`dict` with the following structure:

```
{
    'trytes': TransactionTrytes,
        Stored trytes.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#storetransactions

6.17 were_addresses_spent_from

Iota.**were_addresses_spent_from**(*addresses: Iterable[iota.types.Address]*) → dict
Check if a list of addresses was ever spent from, in the current epoch, or in previous epochs.

If an address has a pending transaction, it's also considered 'spent'.

Parameters addresses (*Iterable[Address]*) – List of addresses to check.

Returns

`dict` with the following structure:

```
{
    'states': List[bool],
        States of the specified addresses in the same order as
        the values in the addresses parameter. A ``True`` value
        means that the address has been spent from.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#wereaddressesspentfrom

async AsyncIota.**were_addresses_spent_from**(*addresses: Iterable[iota.types.Address]*) → dict

Check if a list of addresses was ever spent from, in the current epoch, or in previous epochs.

If an address has a pending transaction, it's also considered 'spent'.

Parameters **addresses** (*Iterable[Address]*) – List of addresses to check.

Returns

`dict` with the following structure:

```
{
    'states': List[bool],
        States of the specified addresses in the same order as
        the values in the addresses parameter. A ``True`` value
        means that the address has been spent from.
    'duration': int,
        Number of milliseconds it took to complete the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#wereaddressesspentfrom

CHAPTER

SEVEN

EXTENDED API METHODS

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

Note: Below you will find the documentation for both the synchronous and asynchronous versions of the Extebded API methods.

It should be made clear, that they do exactly the same IOTA related operations, accept the same arguments and return the same structures. Asynchronous API calls are non-blocking, so your application can do other stuff while waiting for the result from the network.

While synchronous API calls are regular Python methods, their respective asynchronous versions are Python coroutines. You can `await` their results, schedule them for execution inside and event loop and much more. PyOTA uses the built-in asyncio Python module for asynchronous operation. For an overview of what you can do with it, head over to this article.

7.1 broadcast_and_store

`Iota.`**`broadcast_and_store`**(*trytes: Iterable[iota.transaction.types.TransactionTrytes]*) → dict

Broadcasts and stores a set of transaction trytes.

Parameters trytes (*Iterable[TransactionTrytes]*) – Transaction trytes to broadcast and store.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        List of TransactionTrytes that were broadcast.
        Same as the input ``trytes``.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#broadcastandstore

async `AsyncIota.`**`broadcast_and_store`**(*trytes: Iterable[iota.transaction.types.TransactionTrytes]*) → dict

Broadcasts and stores a set of transaction trytes.

Parameters trytes (*Iterable[TransactionTrytes]*) – Transaction trytes to broadcast and store.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        List of TransactionTrytes that were broadcast.
        Same as the input ``trytes``.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#broadcastandstore

7.2 broadcast_bundle

`Iota.`**`broadcast_bundle`**(*tail_transaction_hash: iota.transaction.types.TransactionHash*) → dict

Re-broadcasts all transactions in a bundle given the tail transaction hash. It might be useful when transactions did not properly propagate, particularly in the case of large bundles.

Parameters **`tail_transaction_hash`** (`TransactionHash`) – Tail transaction hash of the bundle.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        List of TransactionTrytes that were broadcast.
}
```

References:

- https://github.com/iotaledger/iota.js/blob/next/api_reference.md#module_core.broadcastBundle

async `AsyncIota.`**`broadcast_bundle`**(*tail_transaction_hash: iota.transaction.types.TransactionHash*) → dict

Re-broadcasts all transactions in a bundle given the tail transaction hash. It might be useful when transactions did not properly propagate, particularly in the case of large bundles.

Parameters **`tail_transaction_hash`** (`TransactionHash`) – Tail transaction hash of the bundle.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        List of TransactionTrytes that were broadcast.
}
```

References:

- https://github.com/iotaledger/iota.js/blob/next/api_reference.md#module_core.broadcastBundle

7.3 `find_transaction_objects`

`Iota.`**`find_transaction_objects`**(*bundles: Optional[Iterable[iota.transaction.types.BundleHash]] = None, addresses: Optional[Iterable[iota.types.Address]] = None, tags: Optional[Iterable[iota.types.Tag]] = None, approvees: Optional[Iterable[iota.transaction.types.TransactionHash]] = None*) → dict

A more extensive version of *`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.

Parameters

- **`bundles`** (*`Optional[Iterable[BundleHash]]`*) – List of bundle IDs.
- **`addresses`** (*`Optional[Iterable[Address]]`*) – List of addresses.
- **`tags`** (*`Optional[Iterable[Tag]]`*) – List of tags.
- **`approvees`** (*`Optional[Iterable[TransactionHash]]`*) – List of approvee transaction IDs.

Returns

`dict` with the following structure:

```
{
    'transactions': List[Transaction],
        List of Transaction objects that match the input.
}
```

`async` `AsyncIota.`**`find_transaction_objects`**(*bundles: Optional[Iterable[iota.transaction.types.BundleHash]] = None, addresses: Optional[Iterable[iota.types.Address]] = None, tags: Optional[Iterable[iota.types.Tag]] = None, approvees: Optional[Iterable[iota.transaction.types.TransactionHash]] = None*) → dict

A more extensive version of *`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.

Parameters

- **`bundles`** (*`Optional[Iterable[BundleHash]]`*) – List of bundle IDs.
- **`addresses`** (*`Optional[Iterable[Address]]`*) – List of addresses.
- **`tags`** (*`Optional[Iterable[Tag]]`*) – List of tags.

- **approvees** (*Optional[Iterable[TransactionHash]]*) – List of approvee transaction IDs.

Returns

`dict` with the following structure:

```
{
    'transactions': List[Transaction],
        List of Transaction objects that match the input.
}
```

7.4 get_account_data

`Iota.`**`get_account_data`**(*start: int = 0*, *stop: Optional[int] = None*, *inclusion_states: bool = False*, *security_level: Optional[int] = None*) → dict

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.

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 that is unused.

  Note: An unused address is an address that **has not been spent from** and **has no transactions** referencing it on the Tangle.

  A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that this API does not return the correct account data with `stop` being `None`.

  As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `start` and `stop` parameters to tell the API from where to start checking and where to stop.

- **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.

- **security_level** (*Optional[int]*) – Number of iterations to use when generating new addresses (see *get_new_addresses()*).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    '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[Bundle],
        List of bundles with transactions to/from this
        account.
}
```

async `AsyncIota.`**`get_account_data`**(*start: int = 0*, *stop: Optional[int] = None*, *inclusion_states: bool = False*, *security_level: Optional[int] = None*) → dict

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.

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 that is unused.

  Note: An unused address is an address that **has not been spent from** and **has no transactions** referencing it on the Tangle.

  A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that this API does not return the correct account data with `stop` being `None`.

  As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `start` and `stop` parameters to tell the API from where to start checking and where to stop.

- **`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.

- **`security_level`** (*`Optional[int]`*) – Number of iterations to use when generating new addresses (see *`get_new_addresses()`*).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    'addresses': List[Address],
```

(continues on next page)

(continued from previous page)

```
        List of generated addresses.

        Note that this list may include unused
        addresses.

    'balance': int,
        Total account balance.  Might be 0.

    'bundles': List[Bundle],
        List of bundles with transactions to/from this
        account.
}
```

7.5 `get_bundles`

`Iota.get_bundles`(*transactions: Iterable[iota.transaction.types.TransactionHash]*) → dict

Returns the bundle(s) associated with the specified transaction hashes.

Parameters **`transactions`** (*`Iterable[TransactionHash]`*) – Transaction hashes. Must be a tail transaction.

Returns

`dict` with the following structure:

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

:raise `iota.adapter.BadApiResponse`:

- if any of the bundles fails validation.
- if any of the bundles is not visible on the Tangle.

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#getbundle

async `AsyncIota.get_bundles`(*transactions: Iterable[iota.transaction.types.TransactionHash]*) → dict

Returns the bundle(s) associated with the specified transaction hashes.

Parameters **`transactions`** (*`Iterable[TransactionHash]`*) – Transaction hashes. Must be a tail transaction.

Returns

`dict` with the following structure:

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

:raise `iota.adapter.BadApiResponse`:

- if any of the bundles fails validation.
- if any of the bundles is not visible on the Tangle.

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#getbundle

7.6 `get_inputs`

`Iota.get_inputs`(*start: int = 0*, *stop: Optional[int] = None*, *threshold: Optional[int] = None*, *security_level: Optional[int] = None*) → dict

Gets all possible inputs of a seed and returns them, along 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.

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.

  Note: An unused address is an address that **has not been spent from** and **has no transactions** referencing it on the Tangle.

  A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that this API does not return the correct inputs with `stop` being `None`.

  As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `start` and `stop` parameters to tell the API from where to start checking for inputs and where to stop.

- **`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.

  Note: 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.

  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.

- **security_level** (*Optional[int]*) – Number of iterations to use when generating new addresses (see *get_new_addresses()*).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    'inputs': List[Address],
        Addresses with nonzero balances that can be used
        as inputs.

    'totalBalance': int,
        Aggregate balance from all matching addresses.
}
```

Note that each *Address* in the result has its *Address.balance* attribute set.

Example:

```
response = iota.get_inputs(...)

input0 = response['inputs'][0] # type: Address
input0.balance # 42
```

Raise

- `iota.adapter.BadApiResponse` if `threshold` is not met. Not applicable if `threshold` is `None`.

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#getinputs

async AsyncIota.**get_inputs**(*start: int = 0*, *stop: Optional[int] = None*, *threshold: Optional[int] = None*, *security_level: Optional[int] = None*) → dict

Gets all possible inputs of a seed and returns them, along 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.

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.

  Note: An unused address is an address that **has not been spent from** and **has no transactions** referencing it on the Tangle.

  A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that this API does not return the correct inputs with `stop` being `None`.

As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `start` and `stop` parameters to tell the API from where to start checking for inputs and where to stop.

- **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.

  Note: 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.

  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.

- **security_level** (*Optional[int]*) – Number of iterations to use when generating new addresses (see *get_new_addresses()*).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    'inputs': List[Address],
        Addresses with nonzero balances that can be used
        as inputs.

    'totalBalance': int,
        Aggregate balance from all matching addresses.
}
```

Note that each *Address* in the result has its *Address.balance* attribute set.

Example:

```
response = iota.get_inputs(...)

input0 = response['inputs'][0] # type: Address
input0.balance # 42
```

Raise

- `iota.adapter.BadApiResponse` if `threshold` is not met. Not applicable if `threshold` is `None`.

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#getinputs

7.7 `get_new_addresses`

`Iota.`**`get_new_addresses`**(*index: int = 0, count: int = 1, security_level: int = 2, checksum: bool = False*)

Generates one or more new addresses from the seed.

Parameters

- **`index`** (*`int`*) – The key index of the first new address to generate (must be >= 0).
- **`count`** (*`int`*) – Number of addresses to generate (must be >= 1).

  Tip: This is more efficient than calling *`get_new_addresses()`* inside a loop.

  If `None`, this method will progressively generate addresses and scan the Tangle until it finds one that has no transactions referencing it and was never spent from.

  Note: A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that when `count` is `None`, this API call returns a "new" address that used to have transactions before the snapshot. As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `index` parameter to tell the API from where to start generating and checking new addresses.

- **`security_level`** (*`int`*) – Number of iterations to use when generating new addresses.

  Larger values take longer, but the resulting signatures are more secure.

  This value must be between 1 and 3, inclusive.

- **`checksum`** (*`bool`*) – Specify whether to return the address with the checksum. Defaults to `False`.

Returns

`dict` with the following structure:

```
{
    'addresses': List[Address],
        Always a list, even if only one address was
        generated.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#getnewaddress

`async` `AsyncIota.`**`get_new_addresses`**(*index: int = 0, count: int = 1, security_level: int = 2, checksum: bool = False*)

Generates one or more new addresses from the seed.

Parameters

- **`index`** (*`int`*) – The key index of the first new address to generate (must be >= 0).
- **`count`** (*`int`*) – Number of addresses to generate (must be >= 1).

  Tip: This is more efficient than calling *`get_new_addresses()`* inside a loop.

If `None`, this method will progressively generate addresses and scan the Tangle until it finds one that has no transactions referencing it and was never spent from.

Note: A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that when `count` is `None`, this API call returns a "new" address that used to have transactions before the snapshot. As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `index` parameter to tell the API from where to start generating and checking new addresses.

- **`security_level`** (*`int`*) – Number of iterations to use when generating new addresses.

  Larger values take longer, but the resulting signatures are more secure.

  This value must be between 1 and 3, inclusive.

- **`checksum`** (*`bool`*) – Specify whether to return the address with the checksum. Defaults to `False`.

Returns

`dict` with the following structure:

```
{
    'addresses': List[Address],
        Always a list, even if only one address was
        generated.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#getnewaddress

7.8 `get_transaction_objects`

`Iota.`**`get_transaction_objects`**(*hashes: [typing.Iterable[iota.transaction.types.TransactionHash]]*) → dict

Fetches transaction objects from the Tangle given their transaction IDs (hashes).

Effectively, this is *`get_trytes()`* + converting the trytes into transaction objects.

Similar to *`find_transaction_objects()`*, but accepts list of transaction hashes as input.

Parameters **`hashes`** (*`Iterable[TransactionHash]`*) – List of transaction IDs (transaction hashes).

Returns

`dict` with the following structure:

```
{
    'transactions': List[Transaction],
        List of Transaction objects that match the input.
}
```

`async` `AsyncIota.`**`get_transaction_objects`**(*hashes: [typing.Iterable[iota.transaction.types.TransactionHash]]*) → dict

Fetches transaction objects from the Tangle given their transaction IDs (hashes).

Effectively, this is *`get_trytes()`* + converting the trytes into transaction objects.

Similar to *find_transaction_objects()*, but accepts list of transaction hashes as input.

Parameters **hashes** (*Iterable[TransactionHash]*) – List of transaction IDs (transaction hashes).

Returns

`dict` with the following structure:

```
{
    'transactions': List[Transaction],
        List of Transaction objects that match the input.
}
```

7.9 get_transfers

Iota.**get_transfers**(*start: int = 0*, *stop: Optional[int] = None*, *inclusion_states: bool = False*) → dict
Returns all transfers associated with the seed.

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 that is unused.

  Note: An unused address is an address that **has not been spent from** and **has no transactions** referencing it on the Tangle.

  A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that this API does not return the expected transfers with `stop` being `None`.

  As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `start` and `stop` parameters to tell the API from where to start checking for transfers and where to stop.

- **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.

Returns

`dict` with the following structure:

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

        This value is always a list, even if only one
        bundle was found.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#gettransfers

async AsyncIota.**get_transfers**(*start: int = 0*, *stop: Optional[int] = None*, *inclusion_states: bool = False*) → dict

Returns all transfers associated with the seed.

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 that is unused.

  Note: An unused address is an address that **has not been spent from** and **has no transactions** referencing it on the Tangle.

  A snapshot removes transactions from the Tangle. As a consequence, after a snapshot, it may happen that this API does not return the expected transfers with `stop` being `None`.

  As a workaround, you can save your used addresses and their `key_index` attribute in a local database. Use the `start` and `stop` parameters to tell the API from where to start checking for transfers and where to stop.

- **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.

Returns

`dict` with the following structure:

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

        This value is always a list, even if only one
        bundle was found.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#gettransfers

7.10 is_confirmed

Iota.**is_confirmed**(*transactions: Iterable[iota.transaction.types.TransactionHash]*) → dict

Get the inclusion states of a set of transactions. This is for determining if a transaction was accepted and confirmed by the network or not.

Parameters **transactions** (*Iterable[TransactionHash]*) – List of transactions you want to get the inclusion state for.

Returns

`dict` with the following structure:

```
{
    'states': List[bool],
        List of boolean values in the same order as the
        transactions parameters. A ``True`` value means the
        transaction was confirmed.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getinclusionstates

async AsyncIota.**is_confirmed**(*transactions: Iterable[iota.transaction.types.TransactionHash]*) → dict

Get the inclusion states of a set of transactions. This is for determining if a transaction was accepted and confirmed by the network or not.

Parameters **transactions** (*Iterable[TransactionHash]*) – List of transactions you want to get the inclusion state for.

Returns

`dict` with the following structure:

```
{
    'states': List[bool],
        List of boolean values in the same order as the
        transactions parameters. A ``True`` value means the
        transaction was confirmed.
    'duration': int,
        Number of milliseconds it took to process the request.
}
```

References:

- https://docs.iota.org/docs/node-software/0.1/iri/references/api-reference#getinclusionstates

7.11 is_promotable

Iota.**is_promotable**(*tails: Iterable[iota.transaction.types.TransactionHash]*) → dict

Checks if tail transaction(s) is promotable by calling *check_consistency()* and verifying that `attachmentTimestamp` is above a lower bound. Lower bound is calculated based on number of milestones issued since transaction attachment.

Parameters **tails** (*Iterable(TransactionHash)*) – List of tail transaction hashes.

Returns

The return type mimics that of *check_consistency()*. `dict` with the following structure:

```
{
    'promotable': bool,
        If ``True``, all tails are promotable. If ``False``, see
        `info` field.

    'info': Optional(List[str])
        If `promotable` is ``False``, this contains info about what
```

(continues on next page)

(continued from previous page)

```
        went wrong.
        Note that when 'promotable' is ``True``, 'info' does not
        exist.

}
```

References: - https://github.com/iotaledger/iota.js/blob/next/api_reference.md#module_core.isPromotable

async AsyncIota.**is_promotable**(*tails: Iterable[iota.transaction.types.TransactionHash]*) → dict

Checks if tail transaction(s) is promotable by calling *check_consistency()* and verifying that attachmentTimestamp is above a lower bound. Lower bound is calculated based on number of milestones issued since transaction attachment.

Parameters **tails** (*Iterable(TransactionHash)*) – List of tail transaction hashes.

Returns

The return type mimics that of *check_consistency()*. dict with the following structure:

```
{
    'promotable': bool,
        If ``True``, all tails are promotable. If ``False``, see
        `info` field.

    'info': Optional(List[str])
        If `promotable` is ``False``, this contains info about what
        went wrong.
        Note that when 'promotable' is ``True``, 'info' does not
        exist.

}
```

References: - https://github.com/iotaledger/iota.js/blob/next/api_reference.md#module_core.isPromotable

7.12 is_reattachable

Iota.**is_reattachable**(*addresses: Iterable[iota.types.Address]*) → dict

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.

Parameters **addresses** (*Iterable[Address]*) – List of addresses.

Returns

dict with the following structure:

```
{
  'reattachable': List[bool],
    Always a list, even if only one address was queried.
}
```

async `AsyncIota.`**`is_reattachable`**(*addresses: Iterable[iota.types.Address]*) → dict

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.

Parameters **`addresses`** (*`Iterable[Address]`*) – List of addresses.

Returns

`dict` with the following structure:

```
{
  'reattachable': List[bool],
    Always a list, even if only one address was queried.
}
```

7.13 `prepare_transfer`

`Iota.`**`prepare_transfer`**(*transfers: Iterable[iota.transaction.creation.ProposedTransaction], inputs: Optional[Iterable[iota.types.Address]] = None, change_address: Optional[iota.types.Address] = None, security_level: Optional[int] = None*) → dict

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

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. Depending on how many transfers you've already sent with your seed, this process could take awhile.
- **`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.
- **`security_level`** (*`Optional[int]`*) – Number of iterations to use when generating new addresses (see *`get_new_addresses()`*).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Raw trytes for the transactions in the bundle,
        ready to be provided to :py:meth:`send_trytes`.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#preparetransfers

async AsyncIota.**prepare_transfer**(*transfers: Iterable[iota.transaction.creation.ProposedTransaction], inputs: Optional[Iterable[iota.types.Address]] = None, change_address: Optional[iota.types.Address] = None, security_level: Optional[int] = None*) → dict

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

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. Depending on how many transfers you've already sent with your seed, this process could take awhile.
- **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.
- **security_level** (*Optional[int]*) – Number of iterations to use when generating new addresses (see *get_new_addresses()*).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Raw trytes for the transactions in the bundle,
        ready to be provided to :py:meth:`send_trytes`.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#preparetransfers

7.14 `promote_transaction`

`Iota.`**`promote_transaction`**(*transaction: iota.transaction.types.TransactionHash*, *depth: int = 3*, *min_weight_magnitude: Optional[int] = None*) → dict

Promotes a transaction by adding spam on top of it.

Parameters

- **`transaction`** (`TransactionHash`) – Transaction hash. Must be a tail transaction.
- **`depth`** (*`int`*) – Depth at which to attach the bundle. Defaults to 3.
- **`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.

Returns

`dict` with the following structure:

```
{
    'bundle': Bundle,
        The newly-published bundle.
}
```

`async` `AsyncIota.`**`promote_transaction`**(*transaction: iota.transaction.types.TransactionHash*, *depth: int = 3*, *min_weight_magnitude: Optional[int] = None*) → dict

Promotes a transaction by adding spam on top of it.

Parameters

- **`transaction`** (`TransactionHash`) – Transaction hash. Must be a tail transaction.
- **`depth`** (*`int`*) – Depth at which to attach the bundle. Defaults to 3.
- **`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.

Returns

`dict` with the following structure:

```
{
    'bundle': Bundle,
        The newly-published bundle.
}
```

7.15 `replay_bundle`

`Iota.`**`replay_bundle`**(*transaction: iota.transaction.types.TransactionHash, depth: int = 3, min_weight_magnitude: Optional[int] = None*) → dict

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.

Parameters

- **`transaction`** (`TransactionHash`) – Transaction hash. Must be a tail.
- **`depth`** (*`int`*) – Depth at which to attach the bundle. Defaults to 3.
- **`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.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Raw trytes that were published to the Tangle.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#replaytransfer

`async` `AsyncIota.`**`replay_bundle`**(*transaction: iota.transaction.types.TransactionHash, depth: int = 3, min_weight_magnitude: Optional[int] = None*) → dict

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.

Parameters

- **`transaction`** (`TransactionHash`) – Transaction hash. Must be a tail.
- **`depth`** (*`int`*) – Depth at which to attach the bundle. Defaults to 3.
- **`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.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Raw trytes that were published to the Tangle.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#replaytransfer

7.16 send_transfer

Iota.**send_transfer**(*transfers: Iterable[iota.transaction.creation.ProposedTransaction], depth: int = 3, inputs: Optional[Iterable[iota.types.Address]] = None, change_address: Optional[iota.types.Address] = None, min_weight_magnitude: Optional[int] = None, security_level: Optional[int] = None*) → dict

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

Parameters

- **transfers** (*Iterable[ProposedTransaction]*) – Transfers to include in the bundle.
- **depth** (*int*) – Depth at which to attach the bundle. Defaults to 3.
- **inputs** (*Optional[Iterable[Address]]*) – List of inputs used to fund the transfer. Not needed for zero-value transfers.
- **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.
- **security_level** (*Optional[int]*) – Number of iterations to use when generating new addresses (see *get_new_addresses()*).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    'bundle': Bundle,
        The newly-published bundle.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#sendtransfer

async AsyncIota.**send_transfer**(*transfers: Iterable[iota.transaction.creation.ProposedTransaction], depth: int = 3, inputs: Optional[Iterable[iota.types.Address]] = None, change_address: Optional[iota.types.Address] = None, min_weight_magnitude: Optional[int] = None, security_level: Optional[int] = None*) → dict

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

Parameters

- **transfers** (*Iterable[ProposedTransaction]*) – Transfers to include in the bundle.
- **depth** (*int*) – Depth at which to attach the bundle. Defaults to 3.

- **inputs** (*Optional[Iterable[Address]]*) – List of inputs used to fund the transfer. Not needed for zero-value transfers.
- **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.
- **security_level** (*Optional[int]*) – Number of iterations to use when generating new addresses (see `get_new_addresses()`).

  This value must be between 1 and 3, inclusive.

  If not set, defaults to `AddressGenerator.DEFAULT_SECURITY_LEVEL`.

Returns

`dict` with the following structure:

```
{
    'bundle': Bundle,
        The newly-published bundle.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#sendtransfer

7.17 send_trytes

`Iota.send_trytes`(*trytes: Iterable[iota.transaction.types.TransactionTrytes], depth: int = 3, min_weight_magnitude: Optional[int] = None*) → dict

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

Parameters

- **trytes** (*Iterable[TransactionTrytes]*) – Transaction encoded as a tryte sequence.
- **depth** (*int*) – Depth at which to attach the bundle. Defaults to 3.
- **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.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Raw trytes that were published to the Tangle.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#sendtrytes

async AsyncIota.**send_trytes**(*trytes: Iterable[iota.transaction.types.TransactionTrytes], depth: int = 3, min_weight_magnitude: Optional[int] = None*) → dict

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

Parameters

- **trytes** (*Iterable[TransactionTrytes]*) – Transaction encoded as a tryte sequence.
- **depth** (*int*) – Depth at which to attach the bundle. Defaults to 3.
- **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.

Returns

`dict` with the following structure:

```
{
    'trytes': List[TransactionTrytes],
        Raw trytes that were published to the Tangle.
}
```

References:

- https://github.com/iotaledger/wiki/blob/master/api-proposal.md#sendtrytes

7.18 traverse_bundle

Iota.**traverse_bundle**(*tail_hash: iota.transaction.types.TransactionHash*) → dict

Fetches and traverses a bundle from the Tangle given a tail transaction hash. Recursively traverse the Tangle, collecting transactions until we hit a new bundle.

This method is (usually) faster than *find_transactions()*, and it ensures we don't collect transactions from replayed bundles.

Parameters tail_hash (TransactionHash) – Tail transaction hash of the bundle.

Returns

`dict` with the following structure:

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

async AsyncIota.**traverse_bundle**(*tail_hash: iota.transaction.types.TransactionHash*) → dict

Fetches and traverses a bundle from the Tangle given a tail transaction hash. Recursively traverse the Tangle, collecting transactions until we hit a new bundle.

This method is (usually) faster than *find_transactions()*, and it ensures we don't collect transactions from replayed bundles.

Parameters tail_hash (TransactionHash) – Tail transaction hash of the bundle.

Returns

`dict` with the following structure:

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

CHAPTER

EIGHT

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.

Warning: 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, 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]`.

8.1 Algorithm

Fig. 1: Deriving addresses from a seed.

The following process takes place when you generate addresses in IOTA:

1. First, a sub-seed is derived from your seed by adding `index` to it, and hashing it once with the Kerl hash function.
2. Then the sub-seed is absorbed and squeezed in a sponge function 27 times for each security level. The result is a private key that varies in length depending on security level.

   Note: A private key with `security_level = 1` consists of 2187 trytes, which is exactly 27 x 81 trytes. As the security level increases, so does the length of the private key: 2 x 2187 trytes for `security_level = 2`, and 3 x 2187 trytes for `security_level = 3`.

3. A private key is split into 81-tryte segments, and these segments are hashed 26 times. A group of 27 hashed segments is called a key fragment. Observe, that a private key has one key fragment for each security level.
4. Each key fragment is hashed once more to generate key digests, that are combined and hashed once more to get the 81-tryte address.

Note: An address is the public key pair of the corresponding private key. When you spend iotas from an address, you need to sign the transaction with a key digest that was generated from the address's corresponing private key. This way you prove that you own the funds on that address.

PyOTA provides two methods for generating addresses:

8.2 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)
# Result is a dict that contains a list of addresses.
addresses = gna_result['addresses']

# 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 *`iota.Iota.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 and was never spent from). 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.

Depending on the `count` parameter, `Iota.get_new_addresses()` can be operated in two modes.

8.2.1 Offline mode

When `count` is greater than 0, the API generates `count` number of addresses starting from `index`. It does not check the Tangle if addresses were used or spent from before.

8.2.2 Online mode

When `count` is `None`, the API starts generating addresses starting from `index`. Then, for each generated address, it checks the Tangle if the address has any transactions associated with it, or if the address was ever spent from. If both of the former checks return "no", address generation stops and the address is returned (a new address is found).

Warning: Take care when using the online mode after a snapshot. Transactions referencing a generated address may have been pruned from a node's ledger, therefore the API could return an already-used address as "new" (note: The snapshot has no effect on the "was ever spent from" check).

To make your application more robust to handle snapshots, it is recommended that you keep a local database with at least the indices of your used addresses. After a snapshot, you could specify `index` parameter as the last index in your local used addresses database, and keep on generating truly new addresses.

PyOTA is planned to receive the account module in the future, that makes the library stateful and hence would solve the issue mentioned above.

8.3 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 *`iota.crypto.addresses.AddressGenerator`*.

`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!

8.3.1 AddressGenerator

class `iota.crypto.addresses.`**`AddressGenerator`**(*seed: Union[AnyStr, bytearray, TryteString], security_level: int = 2, checksum: bool = False*)

Generates new addresses using a standard algorithm.

Note: This class does not check if addresses have already been used; if you want to exclude used addresses, invoke *`iota.Iota.get_new_addresses()`* instead.

Note also that *`iota.Iota.get_new_addresses()`* uses `AddressGenerator` internally, so you get the best of both worlds when you use the API (:

Parameters

- **`seed`** (*`TrytesCompatible`*) – The seed to derive addresses from.
- **`security_level`** (*`int`*) – When generating a new address, you can specify a security level for it. The security level of an address affects how long the private key is, how secure a spent address is against brute-force attacks, and how many transactions are needed to contain the signature.

  Could be either 1, 2 or 3.

  Reference:

  – https://docs.iota.org/docs/getting-started/0.1/clients/security-levels
- **`checksum`** (*`bool`*) – Whether to generate address with or without checksum.

Returns *`iota.crypto.addresses.AddressGenerator`* object.

get_addresses

`AddressGenerator.`**`get_addresses`**(*start: int, count: int = 1, step: int = 1*) → List[iota.types.Address]

Generates and returns one or more addresses at the specified index(es).

This is a one-time operation; if you want to create lots of addresses across multiple contexts, consider invoking *`create_iterator()`* and sharing the resulting generator object instead.

Warning: This method may take awhile to run if the starting index and/or the number of requested addresses is a large number!

Parameters

- **`start`** (*`int`*) – Starting index. Must be >= 0.
- **`count`** (*`int`*) – Number of addresses to generate. Must be > 0.
- **`step`** (*`int`*) – Number of indexes to advance after each address. This may be any non-zero (positive or negative) integer.

Returns

`List[Address]`

Always returns a list, even if only one address is generated.

The returned list will contain `count` addresses, except when `step * count < start` (only applies when `step` is negative).

Raises **`ValueError`** –

- if `count` is lower than 1.
- if `step` is zero.

create_iterator

`AddressGenerator.`**`create_iterator`**(*start: int = 0*, *step: int = 1*) → Generator[iota.types.Address, None, None]

Creates an iterator that can be used to progressively generate new addresses.

Returns an iterator that will create addresses endlessly. Use this if you have a feature that needs to generate addresses "on demand".

Parameters

- **`start`** (*`int`*) – Starting index.

  > **Warning:** This method may take awhile to reset if `start` is a large number!

- **`step`** (*`int`*) – Number of indexes to advance after each address.

  > **Warning:** The generator may take awhile to advance between iterations if `step` is a large number!

Returns `Generator[Address, None, None]` object that you can iterate to generate addresses.

8.4 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 *`iota.crypto.addresses.AddressGenerator`* or *`iota.Iota.get_new_addresses`* 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

NINE

CREATING TRANSFERS

IOTA is a permissionless DLT solution, therefore anyone can send transactions to the network and initiate transfers. The IOTA client libraries help you to abstract away low-level operations required to construct and send a transfer to the Tangle.

In this section, we will explore in depth how to create transactions and bundles with IOTA, furthermore what tools you can use in PyOTA to ease your development process.

Note: Before proceeding, make sure you read and understood the *Basic Concepts* and *PyOTA Types* sections!

9.1 Anatomy of a Transfer

We already know that the Tangle consists of *transactions* referencing each other, each of them two others to be more precise. Transactions can be grouped together in *bundles*. Zero-value bundles contain only zero value transactions, while transfer bundles may also contain input and output transactions.

But how to construct these bundles and send them to the network?

The process can be boiled down to 5 steps:

1. Create individual transaction(s).
2. Construct a bundle from the transaction(s).
3. Obtain references to two unconfirmed transactions ("tips") from the Tangle.
4. Do proof-of-work for each transaction in the bundle.
5. Send the bundle to the network.

Fig. 1: Process of creating and sending a transfer to the Tangle.

9.1.1 1. Create Transactions

The first step is to create the individual transaction objects. You have to specify `address` and `value` for each transaction. Furthermore, you can define a `tag`, and for zero-value transactions, a `message`. A `timestamp` is also required, though this value is usually auto-generated by the IOTA libraries.

Note: Unlike on other decentralised ledgers, IOTA transactions can have positive *or* negative `value` amounts. In order to send iotas from one address to another, at least two transactions are required:

- one with *positive* `value` (to increment the balance of the receiver), and
- one with *negative* `value` (to decrement the balance of the sender).

In PyOTA, use `ProposedTransaction` to declare transactions.

9.1.2 2. Create Bundle from Transactions

A bundle is a collection of transactions, treated as an atomic unit when sent to the network. A bundle makes a value (iota token) transfer possible by grouping together input and output transactions.

A bundle always has to be balanced: the sum of `value` attributes of the transactions in the bundle should always be zero. Transactions in the bundle are also indexed individually and contain information on how many other transactions there are in the bundle.

Once complete, a bundle has to be finalized to generate the bundle hash based on the bundle essence. The bundle hash is the unique identifier of the bundle.

After finalization, input transactions in the bundle need to be signed to prove ownership of iotas being transferred.

Tip: `ProposedBundle` helps you in PyOTA to create bundles, add transactions, finalize the bundle and sign the inputs. We'll see how to use `ProposedBundle` in *Use the Library* below.

9.1.3 3. Select two tips

Tips are transactions that are yet to be confirmed by the network. We can obtain two tips by requesting them from a node. In PyOTA, `get_transactions_to_approve()` does the job: it returns a `trunk` and a `branch` `TransactionHash`.

Because our bundle references these two transactions, it will validate them once it is added to the Tangle.

9.1.4 4. Do Proof-of-Work

The bundle has been finalized, inputs have been signed, and we have two tips; now it's time to prepare the bundle to be attached to the Tangle. As noted in the previous section, every transaction references two other transactions in the Tangle; therefore we need to select these references for each transaction in our bundle.

We also know that transactions within the bundle are linked together through their trunk references. So how do we construct the correct bundle structure and also reference two tips from the network?

For all non-head transactions in the bundle, the trunk reference is the next transaction in the bundle, while the branch reference is the trunk transaction hash, one of the tips.

The head transaction is different: the trunk reference is the trunk tip, while the branch reference is the branch tip.

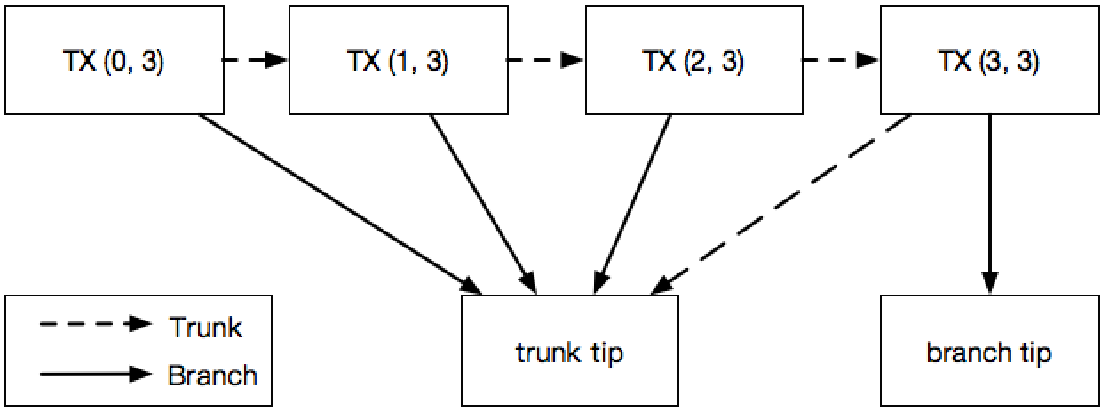

Fig. 2: Structure of a bundle with four transactions. Numbers in brackets denote (`currentIndex`, `lastIndex`) fields. Head of the bundle has index 3, while tail has index 0.

The proof-of-work calculation has to be done for each transaction individually, therefore the more transactions you have in the bundle, the more time it will take. The difficulty of the calculation also depends on the minimum weight magnitude set by the network.

The output of the proof-of-work algorithm is a `nonce` value that is appended to the the transaction, resulting in the attached transaction trytes. Nodes validate the proof-of-work of a transaction by calculating the transaction's hash from the attached transaction trytes. If the resulting hash has at least `minimum weight magnitude` number of trailing zero trits, the transaction is valid.

In PyOTA, use `attach_to_tangle()` to carry out this step.

9.1.5 5. Broadcast and Store

The final step is to send the bundle to the network. Nodes will broadcast the transactions in the network, and store them in their local database.

In PyOTA, use `broadcast_and_store()` to achieve this.

Observe the bird's-eye view of the Tangle depicted at the last step of the process. Our transactions are part of the Tangle, referencing each other and the two tips. Newer transactions may reference our transactions as branch or trunk.

Note: As more transactions are added to the Tangle that reference our transactions – and then more are added that reference those transactions, and so on – this increases the cumulative weight of our transactions. The higher the cumulative weight of our transactions, the higher the chance for them to get confirmed.

9.2 Use the Library

The IOTA libraries help you to abstract away the low-level operations needed to create transfers. The figure below illustrates the different ways you can build and send a transfer.

Fig. 3: API commands for sending transfers.

Let's look at some code snippets on how to perform the above with an imaginary bundle that has 3 fictional transactions.

9.2.1 1. Level Padawan

The easiest and most convenient way is to use *send_transfer()* extended API method. You still need to create the transactions yourself with *ProposedTransaction*.

```
from iota import Iota, ProposedTransaction, Address

api = Iota('https://nodes.devnet.iota.org:443')

fictional_transactions = [
    ProposedTransaction(
            address=Address(b'FIRSTRANDOMADDRESS'),
            value=0,
            # You could add a tag or message here too!
    ),
    ProposedTransaction(
            address=Address(b'SECONDRANDOMADDRESS'),
            value=0,
    ),
    ProposedTransaction(
            address=Address(b'THIRDRANDOMADDRESS'),
            value=0,
    )
]

imaginary_bundle = api.send_transfer(
    transfers=fictional_transactions
)['bundle']
```

As all API methods in PyOTA, *send_transfer()* also returns a `dict`. The `bundle` key holds the value of *Bundle*.

It's important to note, that for value transfers, you will need your seed as well. *send_transfer()* will look for `input addresses` to fund outgoing transactions in the bundle, and auto-generate an unused `change address` if there is a remainder amount of tokens. It will also take care of finalizing the bundle and signing the necessary input transactions.

9.2.2 2. Level Obi-Wan

Instead of `send_transfer()`, you can use the combination of `prepare_transfer()` and `send_trytes()` to achieve the same result.

Tip: This can be useful if you want to prepare the transactions (including signing inputs) on one device, but you want to then transfer the data to another device for transmission to the Tangle. For example, you might `prepare_transfer()` on an air-gapped computer that has your seed stored on it, but then transfer the resulting trytes to a networked computer (that does not have your seed) to `send_trytes()`.

```
from iota import Iota, ProposedTransaction, Address

api = Iota('https://nodes.devnet.iota.org:443')

transactions = [
    ProposedTransaction(
            address=Address(b'FIRSTRANDOMADDRESS'),
            value=0,
    ),
    ProposedTransaction(
            address=Address(b'SECONDRANDOMADDRESS'),
            value=0,
    ),
    ProposedTransaction(
            address=Address(b'THIRDRANDOMADDRESS'),
            value=0,
    )
]

prepared_trytes = api.prepare_transfer(
    transfers=transactions
)['trytes']

imaginary_bundle_trytes = api.send_trytes(
    trytes=prepared_trytes
)['trytes']
```

A difference here is that the end result, `imaginary_bundle_trytes` is a list of `TransactionTrytes`, and not a `Bundle` object.

9.2.3 3. Level Yoda

Being the master Jedi of the PyOTA universe means that you know the most about the force of low-level API methods. Use it wisely!

Tip: You generally won't need to split out the process explicitly like this in your application code, but it is useful to understand what `send_transfer()` does under-the-hood, so that you are better-equipped to troubleshoot any issues that may occur during the process.

```
from iota import Iota, ProposedTransaction, Address, ProposedBundle

api = Iota('https://nodes.devnet.iota.org:443')
```

(continues on next page)

(continued from previous page)

```
transactions = [
    ProposedTransaction(
            address=Address(b'FIRSTRANDOMADDRESS'),
            value=0,
    ),
    ProposedTransaction(
            address=Address(b'SECONDRANDOMADDRESS'),
            value=0,
    ),
    ProposedTransaction(
            address=Address(b'THIRDRANDOMADDRESS'),
            value=0,
    )
]

bundle = ProposedBundle()

for tx in transactions:
    bundle.add_transaction(tx)

# If it was a value transfer, we would also need to:
# bundle.add_inputs()
# bundle.send_unspent_inputs_to()

bundle.finalize()

# Again, for value transfers, we would need to:
# bundle.sign_inputs(KeyGenerator(b'SEEDGOESHERE'))

gtta_response = api.get_transactions_to_approve(depth=3)

trunk = gtta_response['trunkTransaction']
branch = gtta_response['branchTransaction']

attached_trytes = api.attach_to_tangle(
    trunk_transaction=trunk,
    branch_transaction=branch,
    trytes=bundle.as_tryte_strings()
)['trytes']

api.broadcast_transactions(attached_trytes)

api.store_transactions(attached_trytes)

imaginary_bundle = Bundle.from_tryte_strings(attached_trytes)
```

CHAPTER

TEN

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.

First, we will take a look on what *Multisig API* (s) you can use, and what *PyOTA Multisignature Types* are there for you if the standard API is not enough for your application and you want to take more control.

Starting from *Generating multisignature address*, a tutorial follows to show you how to use the multisignature API to execute multisig transfers. The complete source code for the tutorial can be found here.

10.1 Multisig API

The multisignature API builds on top of the extended API to add multisignature features. Just like for the regular APIs, there is both a synchronous and an asynchronous version of the multisignature API, however, as there is no networking required during the multisignature API calls, the difference between them is only how you can call them.

10.1.1 Synchronous Multisignature API Class

class iota.multisig.**MultisigIota**(*adapter: Union[str, BaseAdapter], seed: Union[AnyStr, bytearray, TryteString, None] = None, devnet: bool = False, local_pow: bool = False*)

Extends the IOTA API so that it can send multi-signature transactions. Synchronous API.

Caution: Make sure you understand how multisig works before attempting to use it. If you are not careful, you could easily compromise the security of your private keys, send IOTAs to unspendable addresses, etc.

Example Usage:

```
# Import API class
from iota.multisig import MultisigIota

# Declare a multisig API instance
api = MultisigIota(
```

(continues on next page)

(continued from previous page)

```
        adapter = 'http://localhost:14265',

        seed =
            Seed(
                b'TESTVALUE9DONTUSEINPRODUCTION99999JYFRTI'
                b'WMKVVBAIEIYZDWLUVOYTZBKPKLLUMPDF9PPFLO9KT',
            ),
)

response = api.get_digests(...)
```

References:

- https://github.com/iotaledger/wiki/blob/master/multisigs.md

10.1.2 Asynchronous Multisignature API Class

class iota.multisig.**AsyncMultisigIota**(*adapter: Union[str, BaseAdapter], seed: Union[AnyStr, bytearray, TryteString, None] = None, devnet: bool = False, local_pow: bool = False*)

Extends the IOTA API so that it can send multi-signature transactions. Asynchronous API.

Caution: Make sure you understand how multisig works before attempting to use it. If you are not careful, you could easily compromise the security of your private keys, send IOTAs to unspendable addresses, etc.

Example Usage:

```
# Import API class
from iota.multisig import AsyncMultisigIota

# Declare a multisig API instance
api = AsyncMultisigIota(
        adapter = 'http://localhost:14265',

        seed =
            Seed(
                b'TESTVALUE9DONTUSEINPRODUCTION99999JYFRTI'
                b'WMKVVBAIEIYZDWLUVOYTZBKPKLLUMPDF9PPFLO9KT',
            ),
)

response = await api.get_digests(...)
```

References:

- https://github.com/iotaledger/wiki/blob/master/multisigs.md

10.1.3 `create_multisig_address`

`MultisigIota.`**`create_multisig_address`**(*digests: Iterable[iota.crypto.types.Digest]*) → dict

Generates a multisig address from a collection of digests.

Parameters **`digests`** (*`Iterable[Digest]`*) – Digests to use to create the multisig address.

Important: In order to spend IOTAs from a multisig address, the signature must be generated from the corresponding private keys in the exact same order.

Returns

`dict` with the following items:

```
{
    'address': MultisigAddress,
        The generated multisig address.
}
```

`async` `AsyncMultisigIota.`**`create_multisig_address`**(*digests: Iterable[iota.crypto.types.Digest]*) → dict

Generates a multisig address from a collection of digests.

Parameters **`digests`** (*`Iterable[Digest]`*) – Digests to use to create the multisig address.

Important: In order to spend IOTAs from a multisig address, the signature must be generated from the corresponding private keys in the exact same order.

Returns

`dict` with the following items:

```
{
    'address': MultisigAddress,
        The generated multisig address.
}
```

10.1.4 `get_digests`

`MultisigIota.`**`get_digests`**(*index: int = 0*, *count: int = 1*, *security_level: int = 2*) → dict

Generates one or more key digests from the seed.

Digests are safe to share; use them to generate multisig addresses.

Parameters

- **`index`** (*`int`*) – The starting key index.
- **`count`** (*`int`*) – Number of digests to generate.
- **`security_level`** (*`int`*) – Number of iterations to use when generating new addresses.

  Larger values take longer, but the resulting signatures are more secure.

  This value must be between 1 and 3, inclusive.

Returns

`dict` with the following items:

```
{
    'digests': List[Digest],
        Always contains a list, even if only one digest
        was generated.
}
```

async AsyncMultisigIota.**get_digests**(*index: int = 0*, *count: int = 1*, *security_level: int = 2*) → dict

Generates one or more key digests from the seed.

Digests are safe to share; use them to generate multisig addresses.

Parameters

- **index** (*int*) – The starting key index.
- **count** (*int*) – Number of digests to generate.
- **security_level** (*int*) – Number of iterations to use when generating new addresses.

  Larger values take longer, but the resulting signatures are more secure.

  This value must be between 1 and 3, inclusive.

Returns

`dict` with the following items:

```
{
    'digests': List[Digest],
        Always contains a list, even if only one digest
        was generated.
}
```

10.1.5 `get_private_keys`

MultisigIota.**get_private_keys**(*index: int = 0*, *count: int = 1*, *security_level: int = 2*) → dict

Generates one or more private keys from the seed.

As the name implies, private keys should not be shared. However, in a few cases it may be necessary (e.g., for M-of-N transactions).

Parameters

- **index** (*int*) – The starting key index.
- **count** (*int*) – Number of keys to generate.
- **security_level** (*int*) – Number of iterations to use when generating new keys.

  Larger values take longer, but the resulting signatures are more secure.

  This value must be between 1 and 3, inclusive.

Returns

`dict` with the following items:

```
{
    'keys': List[PrivateKey],
        Always contains a list, even if only one key was
        generated.
}
```

References:

- `iota.crypto.signing.KeyGenerator`
- https://github.com/iotaledger/wiki/blob/master/multisigs.md#how-m-of-n-works

async `AsyncMultisigIota.`**`get_private_keys`**(*index: int = 0*, *count: int = 1*, *security_level: int = 2*) → dict

Generates one or more private keys from the seed.

As the name implies, private keys should not be shared. However, in a few cases it may be necessary (e.g., for M-of-N transactions).

Parameters

- **`index`** (`int`) – The starting key index.
- **`count`** (`int`) – Number of keys to generate.
- **`security_level`** (`int`) – Number of iterations to use when generating new keys.

  Larger values take longer, but the resulting signatures are more secure.

  This value must be between 1 and 3, inclusive.

Returns

`dict` with the following items:

```
{
    'keys': List[PrivateKey],
        Always contains a list, even if only one key was
        generated.
}
```

References:

- `iota.crypto.signing.KeyGenerator`
- https://github.com/iotaledger/wiki/blob/master/multisigs.md#how-m-of-n-works

10.1.6 `prepare_multisig_transfer`

`MultisigIota.`**`prepare_multisig_transfer`**(*transfers: Iterable[iota.transaction.creation.ProposedTransaction]*, *multisig_input: iota.multisig.types.MultisigAddress*, *change_address: Optional[iota.types.Address] = None*) → dict

Prepares a bundle that authorizes the spending of IOTAs from a multisig address.

Note: This method is used exclusively to spend IOTAs from a multisig address.

If you want to spend IOTAs from non-multisig addresses, or if you want to create 0-value transfers (i.e., that don't require inputs), use *`prepare_transfer()`* instead.

Parameters

- **transfers** (*terable[ProposedTransaction]*) – Transaction objects to prepare.

  Important: Must include at least one transaction that spends IOTAs (i.e., has a nonzero `value`). If you want to prepare a bundle that does not spend any IOTAs, use *prepare_transfer()* instead.

- **multisig_input** (MultisigAddress) – The multisig address to use as the input for the transfers.

  Note: This method only supports creating a bundle with a single multisig input.

  If you would like to spend from multiple multisig addresses in the same bundle, create the *ProposedMultisigBundle* object manually.

- **change_address** (*Optional[Address]*) – If inputs are provided, any unspent amount will be sent to this address.

  If the bundle has no unspent inputs, ``change_address` is ignored.

  Important: Unlike *prepare_transfer()*, this method will NOT generate a change address automatically.

  If there are unspent inputs and `change_address` is empty, an exception will be raised.

  This is because multisig transactions typically involve multiple individuals, and it would be unfair to the participants if we generated a change address automatically using the seed of whoever happened to run the `prepare_multisig_transfer` method!

  Danger: Note that this protective measure is not a substitute for due diligence!

  Always verify the details of every transaction in a bundle (including the change transaction) before signing the input(s)!

Returns

`dict` containing the following values:

```
{
    'trytes': List[TransactionTrytes],
        Finalized bundle, as trytes.
        The input transactions are not signed.
}
```

In order to authorize the spending of IOTAs from the multisig input, you must generate the correct private keys and invoke the `iota.crypto.types.PrivateKey.sign_input_at()` method for each key, in the correct order.

Once the correct signatures are applied, you can then perform proof of work (`attachToTangle`) and broadcast the bundle using *send_trytes()*.

async AsyncMultisigIota.**prepare_multisig_transfer**(*transfers: Iterable[iota.transaction.creation.ProposedTransaction], multisig_input: iota.multisig.types.MultisigAddress, change_address: Optional[iota.types.Address] = None*) → dict

Prepares a bundle that authorizes the spending of IOTAs from a multisig address.

Note: This method is used exclusively to spend IOTAs from a multisig address.

If you want to spend IOTAs from non-multisig addresses, or if you want to create 0-value transfers (i.e., that don't require inputs), use *prepare_transfer()* instead.

Parameters

- **transfers** (*terable[ProposedTransaction]*) – Transaction objects to prepare.

  Important: Must include at least one transaction that spends IOTAs (i.e., has a nonzero value). If you want to prepare a bundle that does not spend any IOTAs, use *prepare_transfer()* instead.

- **multisig_input** (MultisigAddress) – The multisig address to use as the input for the transfers.

  Note: This method only supports creating a bundle with a single multisig input.

  If you would like to spend from multiple multisig addresses in the same bundle, create the *ProposedMultisigBundle* object manually.

- **change_address** (*Optional[Address]*) – If inputs are provided, any unspent amount will be sent to this address.

  If the bundle has no unspent inputs, ``change_address` is ignored.

  Important: Unlike *prepare_transfer()*, this method will NOT generate a change address automatically.

  If there are unspent inputs and change_address is empty, an exception will be raised.

  This is because multisig transactions typically involve multiple individuals, and it would be unfair to the participants if we generated a change address automatically using the seed of whoever happened to run the prepare_multisig_transfer method!

  Danger: Note that this protective measure is not a substitute for due diligence!

  Always verify the details of every transaction in a bundle (including the change transaction) before signing the input(s)!

Returns

dict wontaining the following values:

```
{
    'trytes': List[TransactionTrytes],
        Finalized bundle, as trytes.
        The input transactions are not signed.
}
```

In order to authorize the spending of IOTAs from the multisig input, you must generate the correct private keys and invoke the `iota.crypto.types.PrivateKey.sign_input_at()` method for each key, in the correct order.

Once the correct signatures are applied, you can then perform proof of work (`attachToTangle`) and broadcast the bundle using *`send_trytes()`*.

10.2 PyOTA Multisignature Types

There are some specific types defined in PyOTA to help you work with creating multisignature addresses and bundles.

10.2.1 Multisignature Address

class `iota.multisig.types.`**`MultisigAddress`**(*trytes: Union[AnyStr, bytearray, TryteString], digests: Iterable[iota.crypto.types.Digest], balance: Optional[int] = None*)

An address that was generated using digests from multiple private keys.

In order to spend inputs from a multisig address, the same private keys must be used, in the same order that the corresponding digests were used to generate the address.

Note: Usually, you don't have to create a MultisigAddress manually. Use *`create_multisig_address()`* to derive an address from a list of digests.

`MultisigAddress` is a subclass of *`iota.Address`*, so you can use all the regular *`iota.Address`* methods on a *`MultisigAddress`* object.

Parameters

- **`trytes`** (*`TrytesCompatible`*) – Address trytes (81 trytes long).
- **`digests`** (*`Iterable[Digest]`*) – List of digests that were used to create the address. Order is important!
- **`balance`** (*`Optional[int]`*) – Available balance of the address.

Returns *`MultisigAddress`* object.

`as_json_compatible`() → dict

Get a JSON represenation of the *`MultisigAddress`* object.

Returns

`dict` with the following structure:

```
{
    'trytes': str,
        String representation of the address trytes.
    'balance': int,
```

(continues on next page)

(continued from previous page)

```
        Balance of the address.
    'digests': Iterable[Digest]
        Digests that were used to create the address.
}
```

10.2.2 Multisignature ProposedBundle

class iota.multisig.transaction.**ProposedMultisigBundle**(*transactions: Optional[Iterable[iota.transaction.creation.ProposedTr* *= None*, *inputs: Optional[Iterable[iota.types.Address]] = None*, *change_address: Optional[iota.types.Address] = None*)

A collection of proposed transactions, with multisig inputs.

Note: at this time, only a single multisig input is supported per bundle.

Note: Usually you don't have to construct *ProposedMultisigBundle* bundle manually, *prepare_multisig_transfer()* does it for you.

Parameters

- **transactions** (*Optional[Iterable[ProposedTransaction]]*) – Proposed transactions that should be put into the proposed bundle.
- **inputs** (*Optional[Iterable[Address]]*) – Addresses that hold iotas to fund outgoing transactions in the bundle. Currently PyOTA supports only one mutlisig input address per bundle.
- **change_address** (*Optional[Address]*) – Due to the signatures scheme of IOTA, you can only spend once from an address. Therefore the library will always deduct the full available amount from an input address. The unused tokens will be sent to `change_address` if provided.

Returns *ProposedMultisigBundle* object.

add_inputs(*inputs: Iterable[iota.multisig.types.MultisigAddress]*) → None

Adds inputs to spend in the bundle.

Note that each input may require multiple transactions, in order to hold the entire signature.

Parameters inputs (*Iterable[MultisigAddress]*) – MultisigAddresses to use as the inputs for this bundle.

Note: at this time, only a single multisig input is supported.

10.3 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 (Kerl 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 =
      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**.

10.4 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.
        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]
```

10.5 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():
  raise ValueError(
    'Bundle failed validation:\n{errors}'.format(
      errors = '\n'.join(('  - ' + e) for e in validator.errors),
    ),
  )
```

10.6 Broadcast the bundle

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

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

10.7 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.

10.8 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):

10.8.1 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.

10.8.2 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.

10.8.3 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

ELEVEN

ADVANCED: PYOTA COMMANDS

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

It is absolutely not necessary to be familiar with the content described below if you just want to use the library.

However, if you are a curious mind or happen to do development on the library, the following information might be useful.

PyOTA provides the API interface (*Core API Methods* and *Extended API Methods*) for users of the library. These handle constructing and sending HTTP requests to the specified node through adapters, furthermore creating, transforming and translating between PyOTA-specific types and (JSON-encoded) raw data. They also filter outgoing requests and incoming responses to ensure that only appropriate data is communicated with the node.

PyOTA implements the Command Design Pattern. High level API interface methods (*Core API Methods* and *Extended API Methods*) internally call PyOTA commands to get the job done.

Most PyOTA commands are sub-classed from `FilterCommand` class, which is in turn sub-classed from `BaseCommand` class. The reason for the 2-level inheritance is simple: separating functionality. As the name implies, `FilterCommand` adds filtering capabilities to `BaseCommand`, that contains the logic of constructing the request and using its adapter to send it and receive a response.

11.1 Command Flow

As mentioned earlier, API methods rely on PyOTA commands to carry out specific operations. It is important to understand what happens during command execution so you are able to implement new methods that extend the current capabilities of PyOTA.

Let's investigate the process through an example of a core API method, for instance *`find_transactions()`*, that calls `FindTransactionCommand` PyOTA command internally.

Note: `FindTransactionCommand` is sub-classed from `FilterCommand`.

To illustrate what the happens inside the API method, take a look at the following figure

Fig. 1: Inner workings of a PyOTA Command.

- When you call *`find_transactions()`* core API method, it initializes a `FindTransactionCommand` object with the adapter of the API instance it belongs to.
- Then calls this command with the keyword arguments it was provided with.

- The command prepares the request by applying a `RequestFilter` on the payload. The command specific `RequestFilter` validates that the payload has correct types, in some cases it is even able to convert the payload to the required type and format.
- Command execution injects the name of the API command (see IRI API Reference for command names) in the request and sends it to the adapter.
- The adapter communicates with the node and returns its response.
- The response is prepared by going through a command-specific `ResponseFilter`.
- The response is returned to the high level API method as a `dict`, ready to be returned to the main application.

Note: A command object can only be called once without resetting it. When you use the high level API methods, you don't need to worry about resetting commands as each call to an API method will initialize a new command object.

11.2 Filters

If you take a look at the actual implementation of `FindTransactionsCommand`, you notice that you have to define your own request and response filter classes.

Filters in PyOTA are based on the Filters library. Read more about how they work at the filters documentation site.

In short, you can create filter chains through which the filtered value passes, and generates errors if something failed validation. Filter chains are specified in the custom filter class's `__init__()` function. If you also want to modify the filtered value before returning it, override the `_apply()` method of its base class. Read more about how to create custom filters.

PyOTA offers you some custom filters for PyOTA-specific types:

11.2.1 Trytes

class `iota.filters.`**`Trytes`**(*result_type: type = <class 'iota.types.TryteString'>*)

Validates a sequence as a sequence of trytes.

When a value doesn't pass the filter, a `ValueError` is raised with lots of contextual info attached to it.

Parameters **`result_type`** (`TryteString`) – Any subclass of *`TryteString`* that you want the filter to validate.

Raises

- **`TypeError`** – if value is not of `result_type`.
- **`ValueError`** – if `result_type` is not of *`TryteString`* type.

Returns *`Trytes`* object.

11.2.2 StringifiedTrytesArray

`filters.`**`StringifiedTrytesArray`**`(`***runtime_kwargs*`)` → filters.base.FilterChain

Validates that the incoming value is an array containing tryte strings corresponding to the specified type (e.g., `TransactionHash`).

When a value doesn't pass the filter, a `ValueError` is raised with lots of contextual info attached to it.

Parameters **`trytes_type`** (`TryteString`) – Any subclass of *`TryteString`* that you want the filter to validate.

Returns `filters.FilterChain` object.

Important: This filter will return string values, suitable for inclusion in an API request. If you are expecting objects (e.g., *`Address`*), then this is not the filter to use!

Note: This filter will allow empty arrays and *None*. If this is not desirable, chain this filter with `f.NotEmpty` or `f.Required`, respectively.

11.2.3 AddressNoChecksum

`class` `iota.filters.`**`AddressNoChecksum`**

Validates a sequence as an `Address`, then chops off the checksum if present.

When a value doesn't pass the filter, a `ValueError` is raised with lots of contextual info attached to it.

Returns *`AddressNoChecksum`* object.

11.2.4 GeneratedAddress

`class` `iota.filters.`**`GeneratedAddress`**

Validates an incoming value as a generated `Address` (must have `key_index` and `security_level` set).

When a value doesn't pass the filter, a `ValueError` is raised with lots of contextual info attached to it.

Returns *`GeneratedAddress`* object.

11.2.5 NodeUri

`class` `iota.filters.`**`NodeUri`**

Validates a string as a node URI.

When a value doesn't pass the filter, a `ValueError` is raised with lots of contextual info attached to it.

Returns *`NodeUri`* object.

`SCHEMES = {'tcp', 'udp'}`

Allowed schemes for node URIs.

11.2.6 SecurityLevel

`filters.`**`SecurityLevel`**`(`***runtime_kwargs*`)` → filters.base.FilterChain

Generates a filter chain for validating a security level.

Returns `filters.FilterChain` object.

Important: The general rule in PyOTA is that all requests going to a node are validated, but only responses that contain transaction/bundle trytes or hashes are checked.

Also note, that for extended commands, `ResponseFilter` is usually implemented with just a "pass" statement. The reason being that these commands do not directly receive their result a node, but rather from core commands that do have their `ResponseFilter` implemented. More about this topic in the next section.

11.3 Extended Commands

Core commands, like *`find_transactions()`* in the example above, are for direct communication with the node for simple tasks such as finding a transaction on the Tangle or getting info about the node. Extended commands (that serve *Extended API Methods*) on the other hand carry out more complex operations such as combining core commands, building objects, etc…

As a consequence, extended commands override the default execution phase of their base class.

Observe for example `FindTransactionObjectsCommand` extended command that is called in *`find_transaction_objects()`* extended API method. It overrides the `_execute()` method of its base class.

Let's take a closer look at the implementation:

```
...
def _execute(self, request):
    bundles = request\
        .get('bundles')  # type: Optional[Iterable[BundleHash]]
    addresses = request\
        .get('addresses')  # type: Optional[Iterable[Address]]
    tags = request\
        .get('tags')  # type: Optional[Iterable[Tag]]
    approvees = request\
        .get('approvees')  # type: Optional[Iterable[TransactionHash]]

    ft_response = FindTransactionsCommand(adapter=self.adapter)(
        bundles=bundles,
        addresses=addresses,
        tags=tags,
        approvees=approvees,
    )

    hashes = ft_response['hashes']
    transactions = []
    if hashes:
        gt_response = GetTrytesCommand(adapter=self.adapter)(hashes=hashes)

        transactions = list(map(
            Transaction.from_tryte_string,
            gt_response.get('trytes') or [],
```

(continues on next page)

(continued from previous page)

```
        ))  # type: List[Transaction]

    return {
        'transactions': transactions,
    }
...
```

Instead of sending the request to the adapter, `FindTransactionObjectsCommand._execute()` calls `FindTransactionsCommand` core command, gathers the transaction hashes that it found, and collects the trytes of those transactions by calling `GetTrytesCommand` core command. Finally, using the obtained trytes, it constructs a list of transaction objects that are returned to *`find_transaction_objects()`*.

Important: If you come up with a new functionality for the PyOTA API, please raise an issue in the PyOTA Bug Tracker to facilitate discussion.

Once the community agrees on your proposal, you may start implementing a new extended API method and the corresponding extended PyOTA command.

Contributions are always welcome! :)

Visit the Contributing to PyOTA page to find out how you can make a difference!

CHAPTER

TWELVE

TUTORIALS

Are you new to IOTA in Python? Don't worry, we got you covered! With the walkthrough examples of this section, you will be a master of PyOTA.

In each section below, a code snippet will be shown and discussed in detail to help you understand how to carry out specific tasks with PyOTA.

The example scripts displayed here can also be found under `examples/tutorials/` directory in the repository. Run them in a Python environment that has PyOTA installed. See README:Install PyOTA for more info.

If you feel that something is missing or not clear, please post your questions and suggestions in the PyOTA Bug Tracker.

Let's get to it then!

12.1 1. Hello Node

In this example, you will learn how to:

- **Import the `iota` package into your application.**
- **Instantiate an API object for communication with the IOTA network.**
- **Request information about the IOTA node you are connected to.**

12.1.1 Code

```python
# Import neccessary modules
from iota import Iota
from pprint import pprint

# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443')

# Request information about the node
response = api.get_node_info()

# Using pprint instead of print for a nicer looking result in the console
pprint(response)
```

12.1.2 Discussion

```python
# Import neccessary modules
from iota import Iota
from pprint import pprint
```

First things first, we need to import in our application the modules we intend to use. PyOTA provide the `iota` package, therefore, whenever you need something from the library, you need to import it from there.

Notice, how we import the *`Iota`* object, that defines a so-called extended API object. We will use this to send and receive data from the network. Read more about API objects at *PyOTA API Classes*.

We also import the `pprint` method that prettifies the output before printing it to the console.

```python
# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443')
```

Next, we declare an API object. Since this object handles the communication, we need to specify an IOTA node to connect to in the form of an URI. Note, that the library will parse this string and will throw an exception if it is not a valid one.

```python
# Request information about the node
response = api.get_node_info()
```

Then we can call the *`Iota.get_node_info()`* method of the API object to get some basic info about the node.

```python
# Using pprint instead of print for a nicer looking result in the console
pprint(response)
```

Finally, we print out the response. It is important to note, that all API methods return a python dictionary. Refer to the method's documentation to determine what exactly is in the response `dict`. Here for example, we could list the `features` of the node:

```python
pprint(response['features'])
```

12.2 2. Send Data

In this example, you will learn how to:

- **Encode data to be stored on the Tangle.**
- **Generate a random IOTA address that doesn't belong to anyone.**
- **Create a zero-value transaction with custom payload.**
- **Send a transaction to the network.**

12.2.1 Code

```python
from iota import Iota, TryteString, Address, Tag, ProposedTransaction
from pprint import pprint

# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443', devnet=True)

# Prepare custom data
my_data = TryteString.from_unicode('Hello from the Tangle!')

# Generate a random address that doesn't have to belong to anyone
my_address = Address.random()

# Tag is optional here
my_tag = Tag(b'MY9FIRST9TAG')

# Prepare a transaction object
tx = ProposedTransaction(
    address=my_address,
    value=0,
    tag=my_tag,
    message=my_data
)

# Send the transaction to the network
response = api.send_transfer([tx])

pprint('Check your transaction on the Tangle!')
pprint('https://utils.iota.org/transaction/%s/devnet' % response['bundle'][0].hash)
```

12.2.2 Discussion

```python
from iota import Iota, TryteString, Address, Tag, ProposedTransaction
from pprint import pprint

# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443', devnet=True)
```

We have seen this part before. Note, that now we import more objects which we will use to construct our transaction.

Notice `devnet=True` in the argument list of the API instantiation. We tell the API directly that we will use IOTA's testnet, known as the devnet. By default, the API is configured for the mainnet.

```python
# Prepare custom data
my_data = TryteString.from_unicode('Hello from the Tangle!')
```

If you read Basic Concepts and PyOTA Types, it shouldn't be a surprise to you that most things in IOTA are represented as trytes, that are *`TryteString`* in PyOTA.

Here, we encode our message with *`TryteString.from_unicode()`* into trytes.

```python
# Generate a random address that doesn't have to belong to anyone
my_address = Address.random()
```

To put anything (transactions) on the Tangle, it needs to be associated with an address. **Since we will be posting a**

zero-value transaction, nobody has to own this address; therefore we can use the *TryteString.random()* (an *Address* is just a *TryteString* with some additional attributes and fixed length) method to generate one.

```
# Tag is optional here
my_tag = Tag(b'MY9FIRST9TAG')
```

To tag our transaction, we might define a custom *Tag* object. Notice, that the `b` means we are passing a bytestring value instead of a unicode string. This is so that PyOTA interprets our input as literal trytes, rather than a unicode string that needs to be encoded into trytes.

When passing a bytestring to a PyOTA class, each byte is interpreted as a tryte; therefore we are restricted to the tryte alphabet.

```
# Prepare a transaction object
tx = ProposedTransaction(
    address=my_address,
    value=0,
    tag=my_tag,
    message=my_data
)
```

It's time to construct the transaction. According to *Transaction Types*, PyOTA uses *ProposedTransaction* to build transactions that are not yet broadcast to the network. Oberve, that the `value=0` means this is a zero-value transaction.

```
# Send the transaction to the network
response = api.send_transfer([tx])
```

Next, we send the transfer to the node for tip selection, proof-of-work calculation, broadcasting and storing. The API takes care of all these tasks, and returns the resulting `Bundle` object.

Note: *send_transfer()* takes a list of *ProposedTransaction* objects as its `transfers` argument. An IOTA transfer (bundle) usually consists of multiple transactions linked together, however, in this simple example, there is only one transaction in the bundle. Regardless, you need to pass this sole transaction as a list of one transaction.

```
pprint('Check your transaction on the Tangle!')
pprint('https://utils.iota.org/transaction/%s/devnet' % response['bundle'][0].hash)
```

Finally, we print out the transaction's link on the Tangle Explorer. Observe how we extract the transaction hash from the response `dict`. We take the first element of the bundle, as it is just a sequence of transactions, and access its `hash` attribute.

12.3 3. Fetch Data

In this example, you will learn how to:

- **Fetch transaction objects from the Tangle based on a criteria.**
- **Decode messages from transactions.**

12.3.1 Code

```python
from iota import Iota, Address
from iota.codecs import TrytesDecodeError

# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443', devnet=True)

# Address to fetch data from
# Replace with your random generated address from Tutorial 2. to fetch the data
# that you uploaded.
addy = Address(b
↪'WWO9DRAUDDSDSTTUPKJRNPSYLWAVQBBXISLKLTNDPVKOPMUERDUELLUPHNT9L9YWBDKOLYVWRAFRKIBLP')

print('Fetching data from the Tangle...')
# Fetch the transaction objects of the address from the Tangle
response = api.find_transaction_objects(addresses=[addy])

if not response['transactions']:
    print('Couldn\'t find data for the given address.')
else:
    print('Found:')
    # Iterate over the fetched transaction objects
    for tx in response['transactions']:
        # data is in the signature_message_fragment attribute as trytes, we need
        # to decode it into a unicode string
        data = tx.signature_message_fragment.decode(errors='ignore')
        print(data)
```

12.3.2 Discussion

```python
from iota import Iota, Address
from iota.codecs import TrytesDecodeError

# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443', devnet=True)
```

The usual part again, but we also import `TrytesDecodeError` from `iota.codec`. We will use this to detect if the fetched trytes contain encoded text.

```python
# Address to fetch data from
# Replace with your random generated address from Tutorial 2. to fetch the data
# that you uploaded.
addy = Address(b
↪'WWO9DRAUDDSDSTTUPKJRNPSYLWAVQBBXISLKLTNDPVKOPMUERDUELLUPHNT9L9YWBDKOLYVWRAFRKIBLP')
```

We declare an IOTA address on the Tangle to fetch data from. You can replace this address with your own from the previous example *2. Send Data*, or just run it as it is.

```python
print('Fetching data from the Tangle...')
# Fetch the transaction objects of the address from the Tangle
response = api.find_transaction_objects(addresses=[addy])
```

We use *`find_transaction_objects()`* extended API method to gather the transactions that belong to our address. This method is also capable of returning *`Transaction`* objects for bundle hashes, tags or approving transactions. Note that you can supply multiple of these, the method always returns a `dict` with a list of transactions.

Note: Remember, that the parameters need to be supplied as lists, even if there is only one value.

```
if not response['transactions']:
    print('Couldn\'t find data for the given address.')
else:
    print('Found:')
    # Iterate over the fetched transaction objects
    for tx in response['transactions']:
        # data is in the signature_message_fragment attribute as trytes, we need
        # to decode it into a unicode string
        data = tx.signature_message_fragment.decode(errors='ignore')
        print(data)
```

Finally, we extract the data we are looking for from the transaction objects. A *`Transaction`* has several attributes, one of which is the `signature_message_fragment`. This contains the payload message for zero-value transactions, and the digital signature that authorizes spending for value transactions.

Since we are interested in data now, we decode its content (raw trytes) into text. Notice, that we pass the `errors='ignore'` argument to the `decode()` method to drop values we can't decode using `utf-8`, or if the raw trytes can't be decoded into legit bytes. A possible reason for the latter can be if the attribute contains a signature rather than a message.

12.4 4.a Generate Address

In this example, you will learn how to:

- **Generate a random seed.**
- **Generate an IOTA address that belongs to your seed.**
- **Acquire free devnet IOTA tokens that you can use to play around with.**

12.4.1 Code

```
from iota import Iota, Seed

# Generate a random seed, or use one you already have (for the devnet)
print('Generating a random seed...')
my_seed = Seed.random()
# my_seed = Seed(b'MYCUSTOMSEED')
print('Your seed is: ' + str(my_seed))

# Declare an API object
api = Iota(
    adapter='https://nodes.devnet.iota.org:443',
    seed=my_seed,
    devnet=True,
)

print('Generating the first unused address...')
# Generate the first unused address from the seed
response = api.get_new_addresses()

```

(continues on next page)

(continued from previous page)

```
20 addy = response['addresses'][0]
21
22 print('Your new address is: ' + str(addy))
23 print('Go to https://faucet.devnet.iota.org/ and enter you address to receive free␣
   ↪devnet tokens.')
```

12.4.2 Discussion

```
1 from iota import Iota, Seed
2
3 # Generate a random seed, or use one you already have (for the devnet)
4 print('Generating a random seed...')
5 my_seed = Seed.random()
6 # my_seed = Seed(b'MYCUSTOMSEED')
7 print('Your seed is: ' + str(my_seed))
```

We start off by generating a random seed with the help of the library. You are also free to use your own seed, just uncomment line 6 and put it there.

If you choose to generate one, your seed is written to the console so that you can save it for later. Be prepared to do so, because you will have to use it in the following tutorials.

```
9  # Declare an API object
10 api = Iota(
11     adapter='https://nodes.devnet.iota.org:443',
12     seed=my_seed,
13     devnet=True,
14 )
```

Notice, how we pass the `seed` argument to the API class's init method. Whenever the API needs to work with addresses or private keys, it will derive them from this seed.

Important: Your seed never leaves the library and your computer. Treat your (mainnet) seed like any other password for a financial service: safe. If your seed is compromised, attackers can steal your funds.

```
16 print('Generating the first unused address...')
17 # Generate the first unused address from the seed
18 response = api.get_new_addresses()
19
20 addy = response['addresses'][0]
```

To generate a new address, we call *`get_new_addresses()`* extended API method. Without arguments, this will return a `dict` with the first unused address starting from `index` 0. An unused address is address that has no transactions referencing it on the Tangle and was never spent from.

If we were to generate more addresses starting from a desired index, we could specify the `start` and `count` parameters. Read more about how to generate addresses in PyOTA at *Generating Addresses*.

On line 20 we access the first element of the list of addresses in the response dictionary.

```
22 print('Your new address is: ' + str(addy))
23 print('Go to https://faucet.devnet.iota.org/ and enter you address to receive free␣
   ↪devnet tokens.')
```

Lastly, the address is printed to the console, so that you can copy it. Visit https://faucet.devnet.iota.org/ and enter the address to receive free devnet tokens of 1000i.

You might need to wait 1-2 minutes until the sum arrives to you address. To check your balance, go to *4.b Check Balance* or *4.c Get Account Data*.

12.5 4.b Check Balance

In this example, you will learn how to:

- **Check the balance of a specific IOTA address.**

12.5.1 Code

```
from iota import Iota, Address
import time

# Put your address from Tutorial 4.a here
my_address = Address(b'YOURADDRESSFROMTHEPREVIOUSTUTORIAL')

# Declare an API object
api = Iota(adapter='https://nodes.devnet.iota.org:443', devnet=True)

# Script actually runs until you load up your address
success = False

while not success:
    print('Checking balance on the Tangle for a specific address...')
    # API method to check balance
    response = api.get_balances(addresses=[my_address])

    # response['balances'] is a list!
    if response['balances'][0]:
        print('Found the following information for address ' + str(my_address) + ':')
        print('Balance: ' + str(response['balances'][0]) + 'i')
        success = True
    else:
        print('Zero balance found, retrying in 30 seconds...')
        time.sleep(30)
```

12.5.2 Discussion

```
1  from iota import Iota, Address
2  import time
3
4  # Put your address from Tutorial 4.a here
5  my_address = Address(b'YOURADDRESSFROMTHEPREVIOUSTUTORIAL')
6
7  # Declare an API object
8  api = Iota(adapter='https://nodes.devnet.iota.org:443', devnet=True)
```

The first step to check the balance of an address is to actually have an address. Exchange the sample address on line 5 with your generated address from *4.a Generate Address*.

Since we don't need to generate an address, there is no need for a seed to be employed in the API object. Note the `time` import, we need it for later.

```python
# Script actually runs until you load up your address
success = False

while not success:
    print('Checking balance on the Tangle for a specific address...')
    # API method to check balance
    response = api.get_balances(addresses=[my_address])

    # response['balances'] is a list!
    if response['balances'][0]:
        print('Found the following information for address ' + str(my_address) + ':')
        print('Balance: ' + str(response['balances'][0]) + 'i')
        success = True
    else:
        print('Zero balance found, retrying in 30 seconds...')
        time.sleep(30)
```

Our script will poll the network for the address balance as long as the returned balance is zero. Therefore, the address you declared as `my_address` should have some balance. If you see the `Zero balance found...` message a couple of times, head over to https://faucet.devnet.iota.org/ and load up your address.

`get_balances()` returns the confirmed balance of the address. You could supply multiple addresses at the same time and get their respective balances in a single call. Don't forget, that the method returns a `dict`. More details about it can be found at *`get_balances()`*.

12.6 4.c Get Account Data

In this example, you will learn how to:

- **Gather addresses, balance and bundles associated with your seed on the Tangle.**

Warning: Account in the context of this example is not to be confused with the Account Module, that is a feature yet to be implemented in PyOTA.

Account here simply means the addresses and funds that belong to your seed.

12.6.1 Code

```python
from iota import Iota, Seed
from pprint import pprint
import time

# Put your seed from Tutorial 4.a here
my_seed = Seed(b
↪'YOURSEEDFROMTHEPREVIOUSTUTORIAL99999999999999999999999999999999999999999999999')

# Declare an API object
api = Iota(
    adapter='https://nodes.devnet.iota.org:443',
    seed=my_seed,
```

(continues on next page)

(continued from previous page)

```
12      devnet=True
13  )
14
15  # Script actually runs until it finds balance
16  success = False
17
18  while not success:
19      print('Checking account information on the Tangle...')
20      # Gather addresses, balance and bundles
21      response = api.get_account_data()
22
23      # response['balance'] is an integer!
24      if response['balance']:
25          print('Found the following information based on your seed:')
26          pprint(response)
27          success = True
28      else:
29          print('Zero balance found, retrying in 30 seconds...')
30          time.sleep(30)
```

12.6.2 Discussion

```
1  from iota import Iota, Seed
2  from pprint import pprint
3  import time
```

We will need `pprint` for a prettified output of the response `dict` and `time` for polling until we find non-zero balance.

```
5   # Put your seed from Tutorial 4.a here
6   my_seed = Seed(b
    ↪'YOURSEEDFROMTHEPREVIOUSTUTORIAL99999999999999999999999999999999999999999999999999')
7
8   # Declare an API object
9   api = Iota(
10      adapter='https://nodes.devnet.iota.org:443',
11      seed=my_seed,
12      devnet=True
13  )
```

Copy your seed from *4.a Generate Address* onto line 6. The API will use your seed to generate addresses and look for corresponding transactions on the Tangle.

```
15  # Script actually runs until it finds balance
16  success = False
17
18  while not success:
19      print('Checking account information on the Tangle...')
20      # Gather addresses, balance and bundles
21      response = api.get_account_data()
22
23      # response['balance'] is an integer!
24      if response['balance']:
25          print('Found the following information based on your seed:')
```

(continues on next page)

(continued from previous page)

```
        pprint(response)
        success = True
    else:
        print('Zero balance found, retrying in 30 seconds...')
        time.sleep(30)
```

Just like in the previous example, we will poll for information until we find a non-zero balance. *get_account_data()* without arguments generates addresses from `index` 0 until it finds the first unused. Then, it queries the node about bundles of those addresses and sums up their balance.

Note: If you read *get_account_data()* documentation carefully, you notice that you can gain control over which addresses are checked during the call by specifying the `start` and `stop` index parameters.

This can be useful when your addresses with funds do not follow each other in the address namespace, or a snapshot removed transactions from the Tangle. It is recommended that you keep a local database of your already used address indices.

Once implemented in PyOTA, Account Module will address the aforementioned problems.

The response `dict` contains the addresses, bundles and total balance of your seed.

12.7 5. Send Tokens

In this example, you will learn how to:

- **Construct a value transfer with PyOTA.**
- **Send a value transfer to an arbitrary IOTA address.**
- **Analyze a bundle of transactions on the Tangle.**

Note: As a prerequisite to this tutorial, you need to have completed *4.a Generate Address*, and have a seed that owns devnet tokens.

12.7.1 Code

```
from iota import Iota, Seed, Address, TryteString, ProposedTransaction, Tag

# Put your seed here from Tutorial 4.a, or a seed that owns tokens (devnet)
my_seed = Seed(b'YOURSEEDFROMTHEPREVIOUSTUTORIAL')

# Declare an API object
api = Iota(
    adapter='https://nodes.devnet.iota.org:443',
    seed=my_seed,
    devnet=True,
)

# Addres to receive 1i
# Feel free to replace it. For example, run the code from Tutorial 4.a
# and use that newly generated address with a 'fresh' seed.
```

(continues on next page)

(continued from previous page)

```
16  receiver = Address(b
    ↪'WWUTQBO99YDCBVBPAPVCANW9ATSNUPPLCPGDQXGQEVLUBSFHCEWOA9DIYYOXJONDIRHYPXQXOYXDPHREZ')
17
18  print('Constructing transfer of 1i...')
19  # Create the transfer object
20  tx = ProposedTransaction(
21      address=receiver,
22      value=1,
23      message=TryteString.from_unicode('I just sent you 1i, use it wisely!'),
24      tag=Tag('VALUETX'),
25  )
26
27  print('Preparing bundle and sending it to the network...')
28  # Prepare the transfer and send it to the network
29  response = api.send_transfer(transfers=[tx])
30
31  print('Check your transaction on the Tangle!')
32  print('https://utils.iota.org/bundle/%s/devnet' % response['bundle'].hash)
```

12.7.2 Discussion

```
1   from iota import Iota, Seed, Address, TryteString, ProposedTransaction, Tag
2
3   # Put your seed here from Tutorial 4.a, or a seed that owns tokens (devnet)
4   my_seed = Seed(b'YOURSEEDFROMTHEPREVIOUSTUTORIAL')
5
6   # Declare an API object
7   api = Iota(
8       adapter='https://nodes.devnet.iota.org:443',
9       seed=my_seed,
10      devnet=True,
11  )
```

We are going to send a value transaction, that requires us to prove that we own the address containg the funds to spend. Therefore, we need our seed from which the address was generated.

Put your seed from *4.a Generate Address* onto line 4. We pass this seed to the API object, that will utilize it for signing the transfer.

```
13  # Addres to receive 1i
14  # Feel free to replace it. For example, run the code from Tutorial 4.a
15  # and use that newly generated address with a 'fresh' seed.
16  receiver = Address(b
    ↪'WWUTQBO99YDCBVBPAPVCANW9ATSNUPPLCPGDQXGQEVLUBSFHCEWOA9DIYYOXJONDIRHYPXQXOYXDPHREZ')
```

In IOTA, funds move accross addresses, therefore we need to define a **receiver address**. For testing value transfers, you should send the funds only to addresses that you control; if you use a randomly-generated receiver address, you won't be able to recover the funds afterward! Re-run *4.a Generate Address* for a new seed and a new address, or just paste a valid IOTA address that you own onto line 16.

```
18  print('Constructing transfer of 1i...')
19  # Create the transfer object
20  tx = ProposedTransaction(
21      address=receiver,
```

(continues on next page)

(continued from previous page)

```python
    value=1,
    message=TryteString.from_unicode('I just sent you 1i, use it wisely!'),
    tag=Tag('VALUETX'),
)
```

We declare a *ProposedTransaction* object like we did before, but this time, with `value=1` parameter. The smallest value you can send is 1 iota (“1i”), there is no way to break it into smaller chunks. It is a really small value anyway. You can also attach a message to the transaction, for example a little note to the beneficiary of the payment.

```python
print('Preparing bundle and sending it to the network...')
# Prepare the transfer and send it to the network
response = api.send_transfer(transfers=[tx])
```

To actually send the transfer, all you need to do is call *send_transfer()* extended API method. This method will take care of:

- Gathering `inputs` (addresses you own and have funds) to fund the 1i transfer.
- Generating a new `change_address`, and automatically sending the remaining funds (`balance of chosen inputs` - 1i) from `inputs` to `change_address`.

> **Warning:** This step is extremely important, as it prevents you from spending twice from the same address.
>
> When an address is used as an input, all tokens will be withdrawn. Part of the tokens will be used to fund your transaction, the rest will be transferred to `change_address`.

- Constructing the transfer bundle with necessary input and output transactions.
- Finalizing the bundle and signing the spending transactions.
- Doing proof-of-work for each transaction in the bundle and sending it to the network.

```python
print('Check your transaction on the Tangle!')
print('https://utils.iota.org/bundle/%s/devnet' % response['bundle'].hash)
```

Open the link and observe the bundle you have just sent to the Tangle. Probably it will take a couple of seconds for the network to confirm it.

What you see is a bundle with 4 transactions in total, 1 input and 3 outputs. But why are there so many transactions?

- There is one transaction that withdraws iotas, this has negative value. To authorize this spending, a valid signature is included in the transaction’s `signature_message_fragment` field. The signature however is too long to fit into one transaction, therefore the library appends a new, zero-value transaction to the bundle that holds the second part of the signature. This you see on the output side of the bundle.
- A 1i transaction to the receiver address spends part of the withdrawn amount.
- The rest is transfered to `change_address` in a new output transaction.

Once the bundle is confirmed, try rerunning the script from *4.c Get Account Data* with the same seed as in this tutorial. Your balance should be decremented by 1i, and you should see a new address, which was actually the `change_address`.

12.8 6. Store Encrypted Data

In this example, you will learn how to:

- **Convert Python data structures to JSON format.**
- **Encrypt data and include it in a zero-value transaction.**
- **Store the zero-value transaction with encrypted data on the Tangle.**

Warning: We will use the `simple-crypt` external library for encryption/decryption. Before proceeding to the tutorial, make sure you install it by running:

```
pip install simple-crypt
```

12.8.1 Code

```python
"""
Encrypt data and store it on the Tangle.

simplecrypt library is needed for this example (`pip install simple-crypt`)!
"""
from iota import Iota, TryteString, Tag, ProposedTransaction
from simplecrypt import encrypt
from base64 import b64encode
from getpass import getpass

import json

# Declare an API object
api = Iota(
    adapter='https://nodes.devnet.iota.org:443',
    seed=b'YOURSEEDFROMTHEPREVIOUSTUTORIAL',
    devnet=True,
)

# Some confidential information
data = {
    'name' : 'James Bond',
    'age' : '32',
    'job' : 'agent',
    'address' : 'London',
}

# Convert to JSON format
json_data = json.dumps(data)

# Ask user for a password to use for encryption
password = getpass('Please supply a password for encryption:')

print('Encrypting data...')
# Encrypt data
# Note, that in Python 3, encrypt returns 'bytes'
cipher = encrypt(password, json_data)

```

(continues on next page)

(continued from previous page)

```
# Encode to base64, output contains only ASCII chars
b64_cipher = b64encode(cipher)

print('Constructing transaction locally...')
# Convert to trytes
trytes_encrypted_data = TryteString.from_bytes(b64_cipher)

# Generate an address from your seed to post the transfer to
my_address = api.get_new_addresses(index=42)['addresses'][0]

# Tag is optional here
my_tag = Tag(b'CONFIDENTIALINFORMATION')

# Prepare a transaction object
tx = ProposedTransaction(
    address=my_address,
    value=0,
    tag=my_tag,
    message=trytes_encrypted_data,
)

print('Sending transfer...')
# Send the transaction to the network
response = api.send_transfer([tx])

print('Check your transaction on the Tangle!')
print('https://utils.iota.org/transaction/%s/devnet' % response['bundle'][0].hash)
print('Tail transaction hash of the bundle is: %s' % response['bundle'].tail_
↪transaction.hash)
```

12.8.2 Discussion

```
"""
Encrypt data and store it on the Tangle.

simplecrypt library is needed for this example (`pip install simple-crypt`)!
"""
from iota import Iota, TryteString, Tag, ProposedTransaction
from simplecrypt import encrypt
from base64 import b64encode
from getpass import getpass

import json

# Declare an API object
api = Iota(
    adapter='https://nodes.devnet.iota.org:443',
    seed=b'YOURSEEDFROMTHEPREVIOUSTUTORIAL',
    devnet=True,
)
```

We will use the `encrypt` method to encipher the data, and `b64encode` for representing it as ASCII characters. `getpass` will prompt the user for a password, and the `json` library is used for JSON formatting.

We will need an address to upload the data, therefore we need to supply the seed to the `Iota` API instance. The address will be generated from this seed.

```python
# Some confidential information
data = {
    'name' : 'James Bond',
    'age' : '32',
    'job' : 'agent',
    'address' : 'London',
}
```

The data to be stored is considered confidential information, therefore we can't just put it on the Tangle as plaintext so everyone can read it. Think of what would happen if the world's most famous secret agent's identity was leaked on the Tangle…

```python
# Convert to JSON format
json_data = json.dumps(data)
```

Notice, that `data` is a Python `dict` object. As a common way of exchanging data on the web, we would like to convert it to JSON format. The `json.dumps()` method does exactly that, and the result is a JSON formatted plaintext.

```python
# Ask user for a password to use for encryption
password = getpass('Please supply a password for encryption:')

print('Encrypting data...')
# Encrypt data
# Note, that in Python 3, encrypt returns 'bytes'
cipher = encrypt(password, json_data)

# Encode to base64, output contains only ASCII chars
b64_cipher = b64encode(cipher)
```

Next, we will encrypt this data with a secret password we obtain from the user.

Note: When you run this example, please remember the password at least until the next tutorial!

The output of the `encrypt` method is a `bytes` object in Python3 and contains many special characters. This is a problem, since we can only convert ASCII characters from `bytes` directly into *`TryteString`*.

Therefore, we first encode our binary data into ASCII characters with Base64 encoding.

```python
print('Constructing transaction locally...')
# Convert to trytes
trytes_encrypted_data = TryteString.from_bytes(b64_cipher)

# Generate an address from your seed to post the transfer to
my_address = api.get_new_addresses(index=42)['addresses'][0]

# Tag is optional here
my_tag = Tag(b'CONFIDENTIALINFORMATION')

# Prepare a transaction object
tx = ProposedTransaction(
    address=my_address,
    value=0,
    tag=my_tag,
    message=trytes_encrypted_data,
)
```

Now, we are ready to construct the transfer. We convert the encrypted Base64 encoded data to trytes and assign it to the *ProposedTransaction* object's `message` argument.

An address is also needed, so we generate one with the help of *get_new_addresses()* extended API method. Feel free to choose the index of the generated address, and don't forget, that the method returns a `dict` with a list of addresses, even if it contains only one. For more detailed explanation on how addresses are generated in PyOTA, refer to the adresses:Generating Addresses page.

We also attach a custom *Tag* to our *ProposedTransaction*. Note, that if our `trytes_encrypted_data` was longer than the maximum payload of a transaction, the library would split it accross more transactions that together form the transfer bundle.

```
print('Sending transfer...')
# Send the transaction to the network
response = api.send_transfer([tx])

print('Check your transaction on the Tangle!')
print('https://utils.iota.org/transaction/%s/devnet' % response['bundle'][0].hash)
print('Tail transaction hash of the bundle is: %s' % response['bundle'].tail_
↪transaction.hash)
```

Finally, we use *Iota.send_transfer()* to prepare the transfer and send it to the network.

Click on the link to check your transaction on the Tangle Explorer.

The tail transaction (a tail transaction is the one with index 0 in the bundle) hash is printed on the console, because you will need it in the next tutorial, and anyway, it is a good practice to keep a reference to your transfers.

In the next example, we will try to decode the confidential information from the Tangle.

12.9 7. Fetch Encrypted Data

In this example, you will learn how to:

- **Fetch bundles from the Tangle based on their tail transaction hashes.**
- **Extract messages from a bundle.**
- **Decrypt encrypted messages from a bundle.**

Warning: We will use the `simple-crypt` external library for encryption/decryption. Before proceeding to the tutorial, make sure you install it by running:

```
pip install simple-crypt
```

12.9.1 Code

```
"""
Decrypt data fetched from the Tangle.

simplecrypt library is needed for this example (`pip install simple-crypt`)!
"""
from iota import Iota
from simplecrypt import decrypt
```

(continues on next page)

(continued from previous page)

```python
from base64 import b64decode
from getpass import getpass

import json

# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443', devnet=True)

# Prompt user for tail tx hash of the bundle
tail_hash = input('Tail transaction hash of the bundle: ')

print('Looking for bundle on the Tangle...')
# Fetch bundle
bundle = api.get_bundles([tail_hash])['bundles'][0]

print('Extracting data from bundle...')
# Get all messages from the bundle and concatenate them
b64_encrypted_data = "".join(bundle.get_messages())

# Decode from base64
encrypted_data = b64decode(b64_encrypted_data)

# Prompt for passwword
password = getpass('Password to be used for decryption:')

print('Decrypting data...')
# Decrypt data
# decrypt returns 'bytes' in Python 3, decode it into string
json_data = decrypt(password, encrypted_data).decode('utf-8')

# Convert JSON string to python dict object
data = json.loads(json_data)

print('Succesfully decrypted the following data:')
print(data)
```

12.9.2 Discussion

```python
"""
Decrypt data fetched from the Tangle.

simplecrypt library is needed for this example (`pip install simple-crypt`)!
"""
from iota import Iota
from simplecrypt import decrypt
from base64 import b64decode
from getpass import getpass

import json

# Declare an API object
api = Iota('https://nodes.devnet.iota.org:443', devnet=True)
```

In contrast to *6. Store Encrypted Data* where we intended to encrypt data, in this tutorial we will do the reverse, and decrypt data from the Tangle. Therefore, we need the `decrypt` method from `simplecrypt` library and the b64decode method from base64 library.

Furthermore, getpass is needed to prompt the user for a decryption password, and json for deserializing JSON formatted string into Python object.

```
16 # Prompt user for tail tx hash of the bundle
17 tail_hash = input('Tail transaction hash of the bundle: ')
```

To fetch transactions or bundles from the Tangle, a reference is required to retreive them from the network. Transactions are identified by their transaction hash, while a group of transaction (a bundle) by bundle hash. Hashes ensure the integrity of the Tangle, since they contain verifiable information about the content of the transfer objects.

`input()` asks the user to give the tail transaction hash of the bundle that holds the encrypted messages. The tail transaction is the first in the bundle with index 0. Copy and paste the tail transaction hash from the console output of *6. Store Encrypted Data* when prompted.

```
19 print('Looking for bundle on the Tangle...')
20 # Fetch bundle
21 bundle = api.get_bundles([tail_hash])['bundles'][0]
```

Next, we fetch the bundle from the Tangle with the help of the `get_bundles()` extended API command. It takes a list of tail transaction hashes and returns the bundles for each of them. The response `dict` contains a `bundles` key with the value being a list of bundles in the same order as the input argument hashes. Also note, that the bundles in the response are actual PyOTA `Bundle` objects.

To simplify the code, several operations are happening on line 21:

- Calling `get_bundles()` that returns a `dict`,
- accessing the `'bundles'` key in the `dict`,
- and taking the first element of the the list of bundles in the value associated with the key.

```
23 print('Extracting data from bundle...')
24 # Get all messages from the bundle and concatenate them
25 b64_encrypted_data = "".join(bundle.get_messages())
26
27 # Decode from base64
28 encrypted_data = b64decode(b64_encrypted_data)
29
30 # Prompt for passwword
31 password = getpass('Password to be used for decryption:')
32
33 print('Decrypting data...')
34 # Decrypt data
35 # decrypt returns 'bytes' in Python 3, decode it into string
36 json_data = decrypt(password, encrypted_data).decode('utf-8')
37
38 # Convert JSON string to python dict object
39 data = json.loads(json_data)
```

The next step is to extract the content of the message fields of the transactions in the bundle. We call `Bundle.get_messages()` to carry out this operation. The method returns a list of unicode strings, essentially the `signature_message_fragment` fields of the transactions, decoded from trytes into unicode characters.

We then combine these message chunks into one stream of characters by using `string.join()`.

We know that at this stage that we can't make sense of our message, because it is encrypted and encoded into Base64. Let's peel that onion layer by layer:

- On line 28, we decode the message into bytes with `b64decode`.

- On line 31, we ask the user for thr decryption password (from the previous tutorial).
- On line 36, we decrypt the bytes cipher with the password and decode the result into a unicode string.
- Since we used JSON formatting in the previous tutorial, there is one additional step to arrive at our original data. On line 39, we deserialize the JSON string into a Python object, namely a `dict`.

```python
print('Succesfully decrypted the following data:')
print(data)
```

If everything went according to plan and the user supplied the right password, we should see our original data printed out to the console.

Now you know how to use the Tangle for data storage while keeping privacy. When you need more granular access control on how and when one could read data from the Tangle, consider using Masked Authenticated Messaging (MAM).

12.10 8. Send and Monitor Concurrently

In this example, you will learn how to:

- **Use the asynchronous PyOTA API.**
- **Send transactions concurrently.**
- **Monitor confirmation of transactions concurrently.**
- **Execute arbitrary code concurrently while doing the former two.**

Warning: If you are new to coroutines and asynchronous programming in Python, it is strongly recommended that you check out this article and the official asyncio documentation before proceeding.

12.10.1 Code

```python
from iota import AsyncIota, ProposedTransaction, Address, TryteString
from typing import List
import asyncio

# Asynchronous API instance.
api = AsyncIota(
        adapter='https://nodes.devnet.iota.org:443',
        devnet=True,
)

# An arbitrary address to send zero-value transactions to.
addy = Address(
↪'PZITJTHCIIANKQWEBWXUPHWPWVNBKW9GMNALMGGSIAUOYCKNWDLUUIGAVMJYCHZXHUBRIVPLFZHUVDLME')

# Timeout after which confirmation monitoring stops (seconds).
timeout = 120
# How often should we poll for confirmation? (seconds)
polling_interval = 5


```

(continues on next page)

(continued from previous page)

```
async def send_and_monitor(
    transactions: List[ProposedTransaction]
) -> bool:
    """
    Send a list of transactions as a bundle and monitor their confirmation
    by the network.
    """
    print('Sending bundle...')
    st_response = await api.send_transfer(transactions)

    sent_tx_hashes = [tx.hash for tx in st_response['bundle']]

    print('Sent bundle with transactions: ')
    print(*sent_tx_hashes, sep='\n')

    # Measure elapsed time
    elapsed = 0

    print('Checking confirmation...')
    while len(sent_tx_hashes) > 0:
        # Determine if transactions are confirmed
        gis_response = await api.get_inclusion_states(sent_tx_hashes, None)

        for i, (tx, is_confirmed) in enumerate(zip(sent_tx_hashes, gis_response[
↪'states'])):
            if is_confirmed:
                print('Transaction %s is confirmed.' % tx)
                # No need to check for this any more
                del sent_tx_hashes[i]
                del gis_response['states'][i]

        if len(sent_tx_hashes) > 0:
            if timeout <= elapsed:
                # timeout reached, terminate checking
                return False
            # Show some progress on the screen
            print('.')
            # Put on hold for polling_interval. Non-blocking, so you can
            # do other stuff in the meantime.
            await asyncio.sleep(polling_interval)
            elapsed = elapsed + polling_interval

    # All transactions in the bundle are confirmed
    return True


async def do_something() -> None:
    """
    While waiting for confirmation, you can execute arbitrary code here.
    """
    for _ in range(5):
        print('Doing something in the meantime...')
        await asyncio.sleep(2)


async def main() -> None:
    """
```

(continues on next page)

(continued from previous page)

```python
    A simple application that sends zero-value transactions to the Tangle and
    monitors the confirmation by the network. While waiting for the
    confirmation, we schedule a task (`do_something()`) to be executed concurrently.
    """
    # Transactions to be sent.
    transactions = [
        ProposedTransaction(
            address=addy,
            value=0,
            message=TryteString.from_unicode('First'),
        ),
        ProposedTransaction(
            address=addy,
            value=0,
            message=TryteString.from_unicode('Second'),
        ),
        ProposedTransaction(
            address=addy,
            value=0,
            message=TryteString.from_unicode('Third'),
        ),
    ]

    # Schedule coroutines as tasks, wait for them to finish and gather their
    # results.
    result = await asyncio.gather(
            send_and_monitor(transactions),
            # Send the same content. Bundle will be different!
            send_and_monitor(transactions),
            do_something(),
        )

    if not (result[0] and result[1]):
        print('Transactions did not confirm after %s seconds!' % timeout)
    else:
        print('All transactions are confirmed!')

if __name__ == '__main__':
    # Execute main() inside an event loop if the file is ran
    asyncio.run(main())
```

12.10.2 Discussion

This example is divided into 4 logical parts:

1. Imports and constant declarations
2. Coroutine to send and monitor a list of transactions as a bundle.
3. Coroutine to execute arbitrary code concurrently.
4. A main coroutine to schedule the execution of our application.

Let’s start with the most simple one: **Imports and Constants**.

```python
from iota import AsyncIota, ProposedTransaction, Address, TryteString
from typing import List
```

(continues on next page)

(continued from previous page)

```
 3  import asyncio
 4
 5  # Asynchronous API instance.
 6  api = AsyncIota(
 7          adapter='https://nodes.devnet.iota.org:443',
 8          devnet=True,
 9  )
10
11  # An arbitrary address to send zero-value transactions to.
12  addy = Address(
    ↪'PZITJTHCIIANKQWEBWXUPHWPWVNBKW9GMNALMGGSIAUOYCKNWDLUUIGAVMJYCHZXHUBRIVPLFZHUVDLME')
13
14  # Timeout after which confirmation monitoring stops (seconds).
15  timeout = 120
16  # How often should we poll for confirmation? (seconds)
17  polling_interval = 5
```

Notice, that we import the *AsyncIota* api class, because we would like to use the asynchronous and concurrent features of PyOTA. `List` from the `typing` library is needed for correct type annotations, and we also import the asyncio library. This will come in handy when we want to schedule and run the coroutines.

On line 6, we instantiate an asynchronous IOTA api. Functionally, it does the same operations as *Iota*, but the api calls are defined as coroutines. For this tutorial, we connect to a devnet node, and explicitly tell this as well to the api on line 8.

On line 12, we declare an IOTA address. We will send our zero value transactions to this address. Feel free to change it to your own address.

Once we have sent the transactions, we start monitoring their confirmation by the network. Confirmation time depends on current network activity, the referenced tips, etc., therefore we set a `timeout` of 120 seconds on line 15. You might have to modify this value later to see the confirmation of your transactions.

You can also fine-tune the example code by tinkering with `polling_interval`. This is the interval between two subsequent confirmation checks.

Let's move on to the next block, namely the **send and monitor coroutine**.

```
20  async def send_and_monitor(
21      transactions: List[ProposedTransaction]
22  ) -> bool:
23      """
24      Send a list of transactions as a bundle and monitor their confirmation
25      by the network.
26      """
27      print('Sending bundle...')
28      st_response = await api.send_transfer(transactions)
29
30      sent_tx_hashes = [tx.hash for tx in st_response['bundle']]
31
32      print('Sent bundle with transactions: ')
33      print(*sent_tx_hashes, sep='\n')
34
35      # Measure elapsed time
36      elapsed = 0
37
38      print('Checking confirmation...')
39      while len(sent_tx_hashes) > 0:
40          # Determine if transactions are confirmed
```

(continues on next page)

(continued from previous page)

```
41         gis_response = await api.get_inclusion_states(sent_tx_hashes, None)
42 
43         for i, (tx, is_confirmed) in enumerate(zip(sent_tx_hashes, gis_response[
↪'states'])):
44             if is_confirmed:
45                 print('Transaction %s is confirmed.' % tx)
46                 # No need to check for this any more
47                 del sent_tx_hashes[i]
48                 del gis_response['states'][i]
49 
50         if len(sent_tx_hashes) > 0:
51             if timeout <= elapsed:
52                 # timeout reached, terminate checking
53                 return False
54             # Show some progress on the screen
55             print('.')
56             # Put on hold for polling_interval. Non-blocking, so you can
57             # do other stuff in the meantime.
58             await asyncio.sleep(polling_interval)
59             elapsed = elapsed + polling_interval
60 
61     # All transactions in the bundle are confirmed
62     return True
```

Notice, that coroutines are defined in python by the `async def` keywords. This makes them awaitable.

From the type annotations, we see that `send_and_monitor()` accepts a list of *`ProposedTransaction`* objects and return a `bool`.

On line 28, we send the transfers with the help of *`AsyncIota.send_transfer()`*. Since this is not a regular method, but a coroutine, we have to `await` its result. *`AsyncIota.send_transfer()`* takes care of building the bundle, doing proof-of-work and sending the transactions within the bundle to the network.

Once we sent the transfer, we collect individual transaction hashes from the bundle, which we will use for confirmation checking.

On line 39, the so-called confirmation checking starts. With the help of *`AsyncIota.get_inclusion_states()`*, we determine if our transactions have been confirmed by the network.

Note: You might wonder how your transactions get accepted by the network, that is, how they become confirmed.

- Pre-Coordicide (current state), transactions are confirmed by directly or indirectly being referenced by a milestone. A milestone is a special transaction issued by the Coordinator.
- Post-Coordicide , confirmation is the result of nodes reaching consensus by a voting mechanism.

The `None` value for the `tips` parameter in the argument list basically means that we check against the latest milestone.

On line 43, we iterate over our original `sent_tx_hashes` list of sent transaction hashes and `gis_response['states']`, which is a list of `bool` values, at the same time using the built-in zip method. We also employ enumerate, because we need the index of the elements in each iteration.

If a transaction is confirmed, we delete the corresponding elements from the lists. When all transactions are confirmed, `sent_tx_hashes` becomes empty, and the loop condition becomes `False`.

If however, not all transactions have been confirmed, we should continue checking for confirmation. Observe line 58, where we suspend the coroutine with `asyncio.sleep()` for `polling_interval` seconds. Awaiting the result of `asyncio.sleep()` will cause our coroutine to continue execution in `polling_interval` time. While our coroutine is sleeping, other coroutines can run concurrently, hence it is a non-blocking call.

To do something in the meantime, we can **execute another coroutine concurrently**:

```python
async def do_something() -> None:
    """
    While waiting for confirmation, you can execute arbitrary code here.
    """
    for _ in range(5):
        print('Doing something in the meantime...')
        await asyncio.sleep(2)
```

This is really just a dummy coroutine that prints something to the terminal and then goes to sleep periodically, but in a real application, you could do meaningful tasks here.

Now let's look at how to **schedule the execution of our application with the main coroutine**:

```python
async def main() -> None:
    """
    A simple application that sends zero-value transactions to the Tangle and
    monitors the confirmation by the network. While waiting for the
    confirmation, we schedule a task (`do_something()`) to be executed concurrently.
    """
    # Transactions to be sent.
    transactions = [
        ProposedTransaction(
            address=addy,
            value=0,
            message=TryteString.from_unicode('First'),
        ),
        ProposedTransaction(
            address=addy,
            value=0,
            message=TryteString.from_unicode('Second'),
        ),
        ProposedTransaction(
            address=addy,
            value=0,
            message=TryteString.from_unicode('Third'),
        ),
    ]

    # Schedule coroutines as tasks, wait for them to finish and gather their
    # results.
    result = await asyncio.gather(
            send_and_monitor(transactions),
            # Send the same content. Bundle will be different!
            send_and_monitor(transactions),
            do_something(),
        )

    if not (result[0] and result[1]):
        print('Transactions did not confirm after %s seconds!' % timeout)
    else:
        print('All transactions are confirmed!')

if __name__ == '__main__':
    # Execute main() inside an event loop if the file is ran
```

(continues on next page)

(continued from previous page)

```
115     asyncio.run(main())
```

First, we declare a list of *ProposedTransaction()* objects, that will be the input for our `send_and_monitor()` coroutine.

The important stuff begins on line 101. We use `asyncio.gather()` to submit our coroutines for execution, wait for their results and then return them in a list. gather takes our coroutines, transforms them into runnable tasks, and runs them concurrently.

Notice, that we listed `send_and_monitor()` twice in `asyncio.gather()` with the same list of *ProposedTransaction()* objects. This is to showcase how you can send and monitor multiple transfers concurrently. In this example, two different bundles will be created from the same *ProposedTransaction()* objects. The two bundles post zero value transactions to the same address, contain the same messages respectively, but are not dependent on each other in any way. That is why we can send them concurrently.

As discussed previously, `result` will be a list of results of the coroutines submitted to `asyncio.gather()`, preserving their order. `result[0]` is the result from the first `send_and_monitor()`, and `result[1]` is the result from the second `send_and_monitor()` from the argument list. If any of these are `False`, confirmation did not happen before `timeout`.

When you see the message from line 109 in your terminal, try increasing `timeout`, or check the status of the network, maybe there is a temporary downtime on the devnet due to maintenance.

Lastly, observe lines 113-115. If the current file (python module) is run from the terminal, we use `ayncio.run()` to execute the main coroutine inside an event loop.

To run this example, navigate to `examples/tutorial` inside the cloned PyOTA repository, or download the source file of Tutorial 8 from GitHub and run the following in a terminal:

```
$ python 08_async_send_monitor.py
```

CHAPTER THIRTEEN

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).

13.1 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

FOURTEEN

DEPENDENCIES

PyOTA is compatible with Python 3.7 and 3.6.

CHAPTER

FIFTEEN

INSTALL PYOTA

To install the latest version:

```
pip install pyota
```

15.1 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]
```

15.2 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.

15.3 Installing from Source

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

15.3.1 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

SIXTEEN

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
```

PYTHON MODULE INDEX

INDEX