Monero Python module Documentation

Release 1.1.1

Michal Salaban

Sep 28, 2022

Contents:

Warning: **URGENT SECURITY UPDATE** The version 1.0.2 contains an urgent security update in the output recognition code. If you're using the module for scanning transactions and identifying outputs using the secret view key, UPDATE THE SOFTWARE IMMEDIATELY. Otherwise you're safe. Standard wallet operations like receiving payments, spending, address generation etc. are NOT AFFECTED.

Welcome to the documentation for the `monero` Python module.

The aim of this project is to offer a set of tools for interacting with Monero cryptocurrency in Python. It provides higher level classes representing objects from the Monero environment, like wallets, accounts, addresses, transactions.

Currently it operates over JSON RPC protocol, however other backends are planned as well.

Project homepage: https://github.com/monero-ecosystem/monero-python

CHAPTER 1

Quick start

This quick start tutorial will guide you through the first steps of connecting to the Monero wallet. We assume you:

- have basic knowledge of Monero concepts of the wallet and daemon,
- know how to use CLI (*command line interface*),
- have experience with Python.

1.1 Connect to testnet for your own safety

The testnet is another Monero network where worthless coins circulate and where, as the name suggests, all tests are supposed to be run. It's also a place for early deployment of future features of the currency itself. You may read a brief explanation at stackexchange.

Warning: Please run all tests on testnet. The code presented in these docs will perform the requested operations right away, without asking for confirmation. This is live code, not a wallet application that makes sure the user has not made a mistake. **Running on the live net, if you make a mistake, you may lose money.**

1.2 Start the daemon and create a wallet

In order to connect to the testnet network you need to start the daemon:

```
$ monerod --testnet
```

If you haven't used testnet before, it will begin downloading the blockchain, exactly like it does on the live network. In January 2018 the testnet blockchain was slightly over 2 GiB. It may take some time to get it.

You may however create a wallet in the meantime:

```
$ monero-wallet-cli --testnet --generate-new-wallet testwallet
```

For now you may leave the password empty (testnet coins are worthless).

1.3 Start the RPC server

The RPC server is a small utility that will operate on the wallet, exposing a JSON RPC interface. Start it by typing:

```
$ monero-wallet-rpc --testnet --wallet-file testwallet --password "" --rpc-bind-port␣
↪28088 --disable-rpc-login
```

Now you're almost ready to start using Python.

1.4 Install Dependencies

Before you can use the library, you first must download the Python library dependencies with `pip`. It is recommended to use a virtual environment to isolate library versions. Assuming you have `virtualenv` installed to your system, set up a new env, activate it, and install the dependencies.

```
$ virtualenv .venv
$ source .venv/bin/activate
$ pip install -r requirements.txt
$ python
```

Now you can proceed.

1.5 Connect to the wallet

```
In [1]: from monero.wallet import Wallet

In [2]: from monero.backends.jsonrpc import JSONRPCWallet

In [3]: w = Wallet(JSONRPCWallet(port=28088))

In [4]: w.address()
Out[4]:␣
↪A2GmyHHJ9jtUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC4QxAiAX

In [5]: w.balance()
Out[5]: Decimal('0E-12')
```

Congratulations! You have connected to the wallet. You may now proceed to the next part, which will tell you about *interaction with wallet and accounts*.

CHAPTER 2

Using wallet and accounts

The Wallet class provides an abstraction layer to retrieve wallet information, manage accounts and subaddresses, and of course send transfers.

2.1 The wallet

The following example shows how to create and retrieve wallet's accounts and addresses via the default JSON RPC backend:

```
In [1]: from monero.wallet import Wallet

In [2]: w = Wallet(port=28088)

In [3]: w.address()
Out[3]:␣
↪A2GmyHHJ9jtUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC4QxAiAX
```

2.2 Accounts and subaddresses

The accounts are stored in wallet's `accounts` attribute, which is a list.

Regardless of the version, **the wallet by default operates on its account of index 0**, which makes it consistent with the behavior of the CLI wallet client.

```
In [4]: len(w.accounts)
Out[4]: 1

In [5]: w.accounts[0]
Out[5]: <monero.account.Account at 0x7f78992d6898>

In [6]: w.accounts[0].address()
```

(continues on next page)

(continued from previous page)

```
Out[6]:␣
↪A2GmyHHJ9jtUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC4QxAiAX

In [7]: w.addresses()
Out[7]:␣
↪[A2GmyHHJ9jtUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC4QxAiAX]
```

2.2.1 Creating accounts and addresses

Every wallet can have separate accounts and each account can have numerous addresses. The `Wallet.new_account()` and `Account.new_address()` will create new instances, then return a tuple consisting of the subaddress itself, and the subaddress index within the account.

```
In [8]: w.new_address()
Out[8]:␣
↪(BenuGf8eyVhjZwdcxEJY1MHrUfqHjPvE3d7Pi4XY5vQz53VnVpB38bCBsf8AS5rJuZhuYrqdG9URc2eFoCNPwLXtLENT4R7,
↪ 1)

In [9]: w.addresses()
Out[9]:
[A2GmyHHJ9jtUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC4QxAiAX,
↪
␣
↪BenuGf8eyVhjZwdcxEJY1MHrUfqHjPvE3d7Pi4XY5vQz53VnVpB38bCBsf8AS5rJuZhuYrqdG9URc2eFoCNPwLXtLENT4R7]

In [10]: w.new_account()
Out[10]: <monero.account.Account at 0x7f7894dffbe0>

In [11]: len(w.accounts)
Out[11]: 2

In [12]: w.accounts[1].address()
Out[12]:␣
↪Bhd3PRVCnq5T5jjNey2hDSM8DxUgFpNjLUrKAa2iYVhYX71RuCGTekDKZKXoJPAGL763kEXaDSAsvDYb8bV77YT7Jo19GKY

In [13]: w.accounts[1].new_address()
Out[13]:␣
↪(Bbz5uCtnn3Gaj1YAizaHw1FPeJ6T7kk7uQoeY48SWjezEAyrWScozLxYbqGxsV5L6VJkvw5VwECAuLVJKQtHpA3GFXJNPYu,
↪ 1)

In [14]: w.accounts[1].addresses()
Out[14]:
[Bhd3PRVCnq5T5jjNey2hDSM8DxUgFpNjLUrKAa2iYVhYX71RuCGTekDKZKXoJPAGL763kEXaDSAsvDYb8bV77YT7Jo19GKY,
↪
␣
↪Bbz5uCtnn3Gaj1YAizaHw1FPeJ6T7kk7uQoeY48SWjezEAyrWScozLxYbqGxsV5L6VJkvw5VwECAuLVJKQtHpA3GFXJNPYu]
```

As mentioned above, the wallet by default operates on the first account, so `w.new_address()` is equivalent to `w.accounts[0].new_address()`.

In the next chapter we will *learn about addresses*.

2.3 API reference

class monero.account.**Account**(*backend*, *index*, *label=None*)
Monero account.

Provides interface to operate on a wallet's account.

Accounts belong to a `Wallet` and act like separate sub-wallets. No funds can be moved between accounts off-chain (without a transaction).

Parameters

- **backend** – a wallet backend
- **index** – the account's index within the wallet
- **label** – optional account label as *str*

address()
Return account's main address.

Return type *SubAddress*

address_balance(*addresses=None*)
Returns balances of given addresses, or all addresses if none given.

Parameters **addresses** – a sequence of address as `Address` or their indexes within the account as 'int's

Return type list of index, subaddress, balance, num_UTXOs: (*int*, *Address*, *Decimal*, *int*)

addresses()
Returns all addresses of the account.

Return type list

balance(*unlocked=False*)
Returns specified balance.

Parameters **unlocked** – if *True*, return the unlocked balance, otherwise return total balance

Return type Decimal

balances()
Returns a tuple of balance and unlocked balance.

Return type (Decimal, Decimal)

new_address(*label=None*)
Creates a new address.

Parameters **label** – address label as *str*

Return type tuple of subaddress, subaddress index (minor): (*SubAddress*, *int*)

sweep_all(*address*, *priority=2*, *payment_id=None*, *subaddr_indices=None*, *unlock_time=0*, *relay=True*)
Sends all unlocked balance to an address. Returns a list of resulting transactions.

Parameters

- **address** – destination *Address* or subtype
- **priority** – transaction priority, implies fee. The priority can be a number from 1 to 4 (unimportant, normal, elevated, priority) or a constant from *monero.const.PRIO_**.

- **payment_id** – ID for the payment (must be None if *IntegratedAddress* is used as the destination)
- **subaddr_indices** – a sequence of subaddress indices to sweep from. Empty sequence or *None* means sweep all positive balances.
- **unlock_time** – the extra unlock delay
- **relay** – if *True*, the wallet will relay the transaction(s) to the network immediately; when *False*, it will only return the transaction(s) so they might be broadcast later

Return type list of *Transaction*

transfer(*address*, *amount*, *priority=2*, *payment_id=None*, *unlock_time=0*, *relay=True*)
Sends a transfer. Returns a list of resulting transactions.

Parameters

- **address** – destination *Address* or subtype
- **amount** – amount to send
- **priority** – transaction priority, implies fee. The priority can be a number from 1 to 4 (unimportant, normal, elevated, priority) or a constant from *monero.const.PRIO_**.
- **payment_id** – ID for the payment (must be None if *IntegratedAddress* is used as the destination)
- **unlock_time** – the extra unlock delay
- **relay** – if *True*, the wallet will relay the transaction(s) to the network immediately; when *False*, it will only return the transaction(s) so they might be broadcasted later

Return type list of *Transaction*

transfer_multiple(*destinations*, *priority=2*, *payment_id=None*, *unlock_time=0*, *relay=True*)
Sends a batch of transfers. Returns a list of resulting transactions.

Parameters

- **destinations** – a list of destination and amount pairs: [(*Address*, *Decimal*), . . .]
- **priority** – transaction priority, implies fee. The priority can be a number from 1 to 4 (unimportant, normal, elevated, priority) or a constant from *monero.const.PRIO_**.
- **payment_id** – ID for the payment (must be None if *IntegratedAddress* is used as the destination)
- **unlock_time** – the extra unlock delay
- **relay** – if *True*, the wallet will relay the transaction(s) to the network immediately; when *False*, it will only return the transaction(s) so they might be broadcast later

Return type list of transaction and amount pairs: [(*Transaction*, *Decimal*), . . .]

CHAPTER 3

Addresses and payment IDs

The first, original address of the wallet is usually known as the *master address*. All others are just *subaddresses*, even if they represent a separate account within the wallet.

Monero addresses are base58-encoded strings. You may disassemble each of them using the excellent address analysis tool from *luigi1111*.

While the ordinary string representation is perfectly valid to use, you may want to use validation and other features provided by the `monero.address` package.

3.1 Address validation and instatination

The function `monero.address.address()` will recognize and validate Monero address, returning an instance that provides additional functionality.

The following example uses addresses from the wallet *we have generated in the previous chapter*.

Let's start with the master address:

```
In [1]: from monero.address import address

In [2]: a = address(
↪'A2GmyHHJ9jtUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC4QxAiAX
↪')

In [3]: a.net
Out[3]: 'test'

In [4]: a.spend_key()
Out[4]: 'f0481b63cb937fa5960529247ebf6db627ff1b0bb88de9feccc3c504c16aa4b0'

In [5]: a.view_key()
Out[5]: '2c5ba76d22e48a7ea4ddabea3cce66808ba0cc91265371910f893962e977af1e'
```

(continues on next page)

(continued from previous page)

```
In [6]: type(a)
Out[6]: monero.address.Address
```

We may use a subaddress too:

```
In [7]: b = address(
↪'BenuGf8eyVhjZwdcxEJY1MHrUfqHjPvE3d7Pi4XY5vQz53VnVpB38bCBsf8AS5rJuZhuYrqdG9URc2eFoCNPwLXtLENT4R7
↪')

In [8]: b.net
Out[8]: 'test'

In [9]: b.spend_key()
Out[9]: 'ae7e136f46f618fe7f4a6b323ed60864c20070bf110978d7e3868686d5677318'

In [10]: b.view_key()
Out[10]: '2bf801cdaf3a8b41020098a6d5e194f48fa62129fe9d8f09d19fee9260665baa'

In [11]: type(b)
Out[11]: monero.address.SubAddress
```

These two classes, `Address` and `SubAddress` have similar functionality but one significant difference. Only the former may form *integrated address*.

3.2 Generating subaddresses

It is possible to get subaddresses in two ways:

1. Creating them in the wallet file by calling `.new_address()` on `Account` or `Wallet` instance. In properly synced wallet this will return an address that is guaranteed to be fresh and unused. It is the right way if you plan to use one-time addresses to identify payments or to improve your privacy by avoiding address reuse.
2. Requesting arbitrary subaddress by calling `Wallet.get_address(major, minor)` where `major` is the account index and `minor` is the index of the address within an account. Addresses obtained this way are not guaranteed to be fresh and **will not be saved as already generated within the wallet file**. (Watch out for unintentional address reuse!)

3.3 Payment IDs and integrated addresses

Each Monero transaction may carry a **payment ID**. It is a 64 or 256-bit long number that carries additional information between parties. For example, a merchant can generate a payment ID for each order, or an exchange can assign one to each user. The customer/user would then attach the ID to the transaction, so the site operator would know what is the purpose of incoming payment.

A short, 64-bit payment ID can be integrated into an address, creating, well... an **integrated address**.

```
In [12]: ia = a.with_payment_id(0xfeedbadbeef)

In [13]: ia
Out[13]:␣
↪ABySz66nm1QUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC623BT1ptXv
```

(continues on next page)

(continued from previous page)

```
In [14]: ia.base_address()
Out[14]:␣
↪A2GmyHHJ9jtUhPiwoAbR2tXU9LJu2U6fJjcsv3rxgkVRWU6tEYcn6C1NBc7wqCv5V7NW3zeYuzKf6RGGgZTFTpVC4QxAiAX

In [15]: ia.base_address() == a
Out[15]: True

In [16]: ia.payment_id()
Out[16]: 00000feedbadbeef
```

Since subaddresses have been introduced, merchants may generate a separate address for each order, user or any other object they expect the payments coming to. Therefore, it has been decided that subaddresses cannot generate integrated addresses.

```
In [17]: b.with_payment_id(0xfeedbadbeef)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-5a5811a6962a> in <module>()
----> 1 b.with_payment_id(0xfeedbadbeef)

~/devel/monero-python/monero/address.py in with_payment_id(self, _)
     99
    100     def with_payment_id(self, _):
--> 101         raise TypeError("SubAddress cannot be integrated with payment ID")
    102
    103

TypeError: SubAddress cannot be integrated with payment ID
```

The `monero.numbers.PaymentID` class validates payment IDs. It accepts both integer and hexadecimal string representations.

```
In [18]: from monero.numbers import PaymentID

In [19]: p1 = PaymentID(0xfeedbadbeef)

In [20]: p2 = PaymentID('feedbadbeef')

In [21]: p1 == p2
Out[21]: True

In [22]: p1.is_short()
Out[22]: True

In [23]: p3 = PaymentID('1234567890abcdef0')

In [24]: p3
Out[24]: 0000000000000000000000000000000000000000000000001234567890abcdef0

In [25]: p3.is_short()
Out[25]: False
```

Long payment IDs cannot be integrated:

```
In [26]: a.with_payment_id(p3)
---------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
TypeError                                 Traceback (most recent call last)
<ipython-input-8-7098746f0b69> in <module>
----> 1 a.with_payment_id(p3)

~/devel/monero-python/monero/address.py in with_payment_id(self, payment_id)
    138         payment_id = numbers.PaymentID(payment_id)
    139         if not payment_id.is_short():
--> 140             raise TypeError("Payment ID {0} has more than 64 bits and cannot␣
↪be integrated".format(payment_id))
    141         prefix = const.INTADDRR_NETBYTES[const.NETS.index(self.net)]
    142         data = bytearray([prefix]) + self._decoded[1:65] + struct.pack('>Q',␣
↪int(payment_id))

TypeError: Payment ID␣
↪0000000000000000000000000000000000000000000000001234567890abcdef0 has more than 64␣
↪bits and cannot be integrated
```

3.4 API reference

class monero.address.**Address**(*addr*, *label=None*)

Monero address.

Address of this class is the master address for a `Wallet`.

Parameters

- **address** – a Monero address as string-like object
- **label** – a label for the address (defaults to *None*)

check_private_spend_key(*key*)

Checks if private spend key matches this address.

Return type bool

check_private_view_key(*key*)

Checks if private view key matches this address.

Return type bool

with_payment_id(*payment_id=0*)

Integrates payment id into the address.

Parameters **payment_id** – int, hexadecimal string or *PaymentID* (max 64-bit long)

Return type *IntegratedAddress*

Raises *TypeError* if the payment id is too long

class monero.address.**IntegratedAddress**(*address*)

Monero integrated address.

A master address integrated with payment id (short one, max 64 bit).

base_address()

Returns the base address without payment id. :rtype: *Address*

payment_id()

Returns the integrated payment id.

Return type *PaymentID*

class monero.address.**SubAddress**(*addr*, *label=None*)
Monero subaddress.

Any type of address which is not the master one for a wallet.

monero.address.**address**(*addr*, *label=None*)
Discover the proper class and return instance for a given Monero address.

Parameters

- **addr** – the address as a string-like object
- **label** – a label for the address (defaults to *None*)

Return type *Address*, *SubAddress* or *IntegratedAddress*

CHAPTER 4

Sending and receiving payments

Payments in Monero desire a bit of explanation even for people experienced with cryptocurrency.

The main difference from coins which use transparent blockchain is that Monero transactions do not disclose sender or recipient's address, nor they tell what the amount is. This is a great feature that makes Monero stand out, however at the same time it causes difficulties. In the outgoing payments you won't see the recipient address and, in the incoming ones you won't see the sender.

For this reason, there are two classes representing those, `IncomingPayment` and `OutgoingPayment`. They share most attributes from the parent `Payment` class but carry only one address, depending on which end of the payment your wallet is. Your end address is present in `local_address` attribute.

4.1 Retrieving payments

Each `Wallet` and `Account` object has two methods which will return the list of incoming or outgoing payments:

```
In [4]: wallet.incoming()
Out[4]:
[in: e9a71c01875bec20812f71d155bfabf42024fde3ec82475562b817dcc8cbf8dc @ 1087530 2.
↪120000000000 id=cb248105ea6a9189,
 in: a0b876ebcf7c1d499712d84cedec836f9d50b608bb22d6cb49fd2feae3ffed14 @ 1087606 1.
↪000000000000 id=0166d8da6c0045c51273dd65d6f63734beb8a84e0545a185b2cfd053fced9f5d,
 in: d29264ad317e8fdb55ea04484c00420430c35be7b3fe6dd663f99aebf41a786c @ 1087858 3.
↪140000000000 id=03f6649304ea4cb2,
 in: f349c6badfa7f6e46666db3996b569a05c6ac4e85417551ec208d96f8a37294a @ 1088400 1.
↪000000000000 id=0000000000000000,
 in: bc8b7ef53552c2d4bce713f513418894d0e2c8dcaf72e681e1d4d5a202f1eb62 @ 1088394 8.
↪000000000000 id=0000000000000000,
 in: 5ef7ead6a041101ed326568fbb59c128403cba46076c3f353cd110d969dac808 @ 1087601 7.
↪000000000000 id=0000000000000000,
 in: cc44568337a186c2e1ccc080b43b4ae9db26a07b7afd7edeed60ce2fc4a6477f @ 1087530 10.
↪000000000000 id=0000000000000000,
 in: 41304bbb514d1abdfdb0704bf70f8d2ec4e753c57aa34b6d0525631d79113b87 @ 1088400 1.
↪000000000000 id=1f2510a597bd634bbd130cf21e63b4ad01f4565faf0d3eb21589f496bf28f7f2,
```

(continues on next page)

(continued from previous page)

```
 in: f34b495cec77822a70f829ec8a5a7f1e727128d62e6b1438e9cb7799654d610e @ 1087601 3.
↪000000000000 id=f75ad90e25d71a12,
 in: 5c3ab739346e9d98d38dc7b8d36a4b7b1e4b6a16276946485a69797dbf887cd8 @ 1087530 10.
↪000000000000 id=f75ad90e25d71a12,
 in: 4ea70add5d0c7db33557551b15cd174972fcfc73bf0f6a6b47b7837564b708d3 @ 1087530 4.
↪000000000000 id=f75ad90e25d71a12]

In [5]: wallet.outgoing()
Out[5]:
[out: a8829744952facbfdaab21ca193298edb1fa16f688cd5dbcdff3ed3968155f28 @ 1088411 2.
↪220000000000 id=0000000000000000,
 out: e291fe40c6102a6193c82ac33227c08e5b30a863dba1bc63e13043a25abbb97a @ 1088523 0.
↪123000000000 id=0000000000000000,
 out: 40de45db57eb87eb8395baf5c1dc705602938317d043f463e68ed85b7108f9f3 @ 1088184 1.
↪000000000000 id=0000000000000000,
 out: 2b41226d45edb875634694fccd95f3c174daa5062763eee619ed4475a7b9207a @ 1088184 2.
↪450000000000 id=6cc9350927868849,
 out: 5e8f392a42899294e6b679ddac13cfe1dfe7f034b1e347424218af06c3dfdc85 @ 1088394 1.
↪000000000000 id=6cc9350927868849,
 out: 5d15fef66fe8de715bcbf2c181f97b9932f9f843aef4724f3026fa3cd1082c68 @ 1088521 3.
↪333333333330 id=0000000000000000,
 out: edc7c28e7b07486be48dac0a178f25a3505114267ddaf3e62ab00d9a6e996044 @ 1088394 21.
↪000000000000 id=0000000000000000,
 out: e32cccd7522e760b1c8a571fd08c75e7a1d822874380edc9656f58284eeb2fe5 @ 1088441 0.
↪070000000000 id=0000000000000000,
 out: d09666238129a1e2a71a9b7c6b30564a95baef926556bb658785cb9c38d9b25a @ 1088479 0.
↪210000000000 id=0000000000000000,
 out: 551721b5358b02565d6a9862867e3806b9a2e0d5aa5158d4d30940251d27bbdd @ 1088516 1.
↪111111111000 id=0000000000000000,
 out: 21e7eb651e8a2bc7661975e965ac6b30a6f4033c6a02e96320e41139ad3e307c @ 1088438 0.
↪070000000000 id=0000000000000000,
 out: 34833fef78c7b7c15383a78912344ecfb3ace479d27c4bd6f3e3f136ddc1d6a9 @ 1088538 3.
↪141592653589 id=0000000000000000000000000000000079323846264338327950288419716939]
```

4.1.1 Filtering payments

Retrieving all payments and processing them each time sounds uncomfortable, especially in old wallets which have seen a lot of transfers. To make it easier, you may use filtering on payment queries.

For example, you may ask for payments from a recent period, limiting the blockchain height:

```
In [1]: wallet.incoming(min_height=1088000)
Out[1]:
[in: f349c6badfa7f6e46666db3996b569a05c6ac4e85417551ec208d96f8a37294a @ 1088400 1.
↪000000000000 id=0000000000000000,
 in: bc8b7ef53552c2d4bce713f513418894d0e2c8dcaf72e681e1d4d5a202f1eb62 @ 1088394 8.
↪000000000000 id=0000000000000000,
 in: 41304bbb514d1abdfdb0704bf70f8d2ec4e753c57aa34b6d0525631d79113b87 @ 1088400 1.
↪000000000000 id=1f2510a597bd634bbd130cf21e63b4ad01f4565faf0d3eb21589f496bf28f7f2]
```

Or ask for specific payment ID:

```
In [2]: wallet.incoming(payment_id='f75ad90e25d71a12')
Out[2]:
[in: f34b495cec77822a70f829ec8a5a7f1e727128d62e6b1438e9cb7799654d610e @ 1087601 3.
↪000000000000 id=f75ad90e25d71a12,
```

(continues on next page)

(continued from previous page)

```
 in: 5c3ab739346e9d98d38dc7b8d36a4b7b1e4b6a16276946485a69797dbf887cd8 @ 1087530 10.
↪000000000000 id=f75ad90e25d71a12,
 in: 4ea70add5d0c7db33557551b15cd174972fcfc73bf0f6a6b47b7837564b708d3 @ 1087530 4.
↪000000000000 id=f75ad90e25d71a12]
```

Or limit by both criteria at the same time:

```
In [3]: wallet.incoming(payment_id='f75ad90e25d71a12', min_height=1087601)
Out[3]: [in: f34b495cec77822a70f829ec8a5a7f1e727128d62e6b1438e9cb7799654d610e @␣
↪1087601 3.000000000000 id=f75ad90e25d71a12]
```

You may also filter payments by the address:

```
In [4]: wallet.incoming(local_address=
↪'BhE3cQvB7VF2uuXcpXp28Wbadez6GgjypdRS1F1Mzqn8Advd6q8VfaX8ZoEDobjejrMfpHeNXoX8MjY8q8prW1PEALgr1En
↪')
Out[4]:
[in: 5ef7ead6a041101ed326568fbb59c128403cba46076c3f353cd110d969dac808 @ 1087601 7.
↪000000000000 id=0000000000000000,
 in: 41304bbb514d1abdfdb0704bf70f8d2ec4e753c57aa34b6d0525631d79113b87 @ 1088400 1.
↪000000000000 id=1f2510a597bd634bbd130cf21e63b4ad01f4565faf0d3eb21589f496bf28f7f2,
 in: f34b495cec77822a70f829ec8a5a7f1e727128d62e6b1438e9cb7799654d610e @ 1087601 3.
↪000000000000 id=f75ad90e25d71a12]
```

The same criteria may be used for filtering outgoing payments.

Note: In outgoing payments the *local_address* is always set to the account's main address, making such filtering useless.

4.2 Payment and Transaction objects

Each of the payments returned by the wallet carries all essential data:

```
In [5]: incoming = wallet.incoming()

In [6]: incoming[0].amount
Out[6]: Decimal('2.120000000000')

In [7]: incoming[0].local_address
Out[7]:␣
↪9tQoHWyZ4yXUgbz9nvMcFZUfDy5hxcdZabQCxmNCUukKYicXegsDL7nQpcUa3A1pF6K3fhq3scsyY88tdB1MqucULcKzWZC

In [8]: incoming[0].payment_id
Out[8]: cb248105ea6a9189
```

It also has a related `Transaction` object which offers additional information:

```
In [9]: incoming[0].transaction.height
Out[9]: 1087530

In [10]: incoming[0].transaction.hash
Out[10]: 'e9a71c01875bec20812f71d155bfabf42024fde3ec82475562b817dcc8cbf8dc'
```

Having a running instance of the wallet you may always check the number of confirmations for each payment object:

```
In [11]: wallet.confirmations(incoming[0])
Out[11]: 5132
```

4.3 Mempool: Unconfirmed payments

New transactions, before they are mined in the blocks, land in place called mempool. Each network node updates the mempool contents with new transactions coming from their peers, while offering them the transactions they do not have.

Warning: The presence of a transaction in the mempool is an indication that someone has already attempted a payment, but **should never be used as a proof the payment has been done**. A transaction in mempool might be replaced by another one spending the same funds, it might expire before being included in a block due to competition of other transactions with higher fees. It might also be a result of a sophisticated attack.

With significant amounts you should also wait for a few confirmations to appear. The top of the blockchain sometimes gets replaced by a competing block. It is a popular practice to wait for at least 10 confirmations to appear, which is also the standard in Monero before funds get unlocked and can be used in subsequent transactions.

However, it is possible to query the wallet for transactions in the mempool. You may use them as proofs of payment for less significant amounts where time of acceptance is more important than limiting the risk of a fraud.

By default, the queries check only the blockchain. This behavior can be changed by `confirmed` and `unconfirmed` query parameters that accept boolean values:

```
In [12]: wallet.incoming(unconfirmed=True, confirmed=False)
Out[12]: [in: 21fd4c0b2671bfc32d7c968fdf3cab1001042128d9429d4a26d4f3dc76bcecb8 @ pool␣
↪3.141592653589 id=0000000000000000]

In [13]: incoming[0].transaction.height is None
Out[13]: True

In [14]: wallet.confirmations(incoming[0])
Out[14]: 0
```

You may as well query for both confirmed and unconfirmed transactions using `wallet.incoming(unconfirmed=True)` (the default value for `confirmed` is `True`).

Note: Mempool transactions don't belong to the blockchain (yet), therefore they have no height. Setting `min_height` or `max_height` arguments will **always exclude mempool transactions**. If `unconfirmed` is also set to `True`, a warning will be issued.

4.4 Sending payments

There are two methods for sending Monero. For a single payment use the `transfer` method of `Wallet` or `Account` object.

It returns a list of resulting transactions. In most cases it will contain only one element, but sometimes, for example when many small inputs are used, it might become necessary to split the payment into multiple transactions.

```
In [15]: from decimal import Decimal

In [16]: txs = wallet.transfer(

↪'BdYguH2fVo3G37o8bKp8RbTRuRsTpvBaUdxeo9fj6LFrE2XqNMYKytvBLXvNtnbmXtDUwrKLcpeH4NCuhFL2cXikDV4Rzq6
↪',
    Decimal('2.54'))

In [17]: txs
Out[17]: [f6e7532322f2cab837e668e7ee7be38f0ca4c0cb8c6cff7aa1cfaaf764735acb]

In [18]: txs[0].height is None
Out[18]: True

In [19]: wallet.confirmations(txs[0])
Out[19]: 0

In [20]: wallet.outgoing(unconfirmed=True, confirmed=False)
Out[20]: [out: f6e7532322f2cab837e668e7ee7be38f0ca4c0cb8c6cff7aa1cfaaf764735acb @␣
↪pool 2.540000000000 id=0000000000000000]
```

When sending multiple payments at once, it is more convenient and cheaper in terms of network fees to use `transfer_multiple`:

```
In [25]: txs = wallet.transfer_multiple([
    (
↪'Ba8xvGs5qw1JfiQVJDj8D28NuyL7MuKsB59jtnx2q1ydH4CazTWfJo9iKvTyeYEoYYQ6RT6A1DfoSj1UiwssKfdjUNumu2K
↪', Decimal('0.11')),
    (
↪'BcVT4P2r1Md1DftWBDKHdK38Md6NtFPu4Heof8atNpxx7zbKfhMtRmUUMooU4cJuH4EKXrpke5A77XVbPhekWuiCSTaDFjw
↪', Decimal('1.22')),
    (
↪'Bf2xXxMLdH9gyh35o6LEyKCz6ZsPRmcujBU9rFK81Brd8HmynFj16KFHAYCETU625hY2x7XBH7CvjCHAC6bxQfsjN77Jv7e
↪', Decimal('2.33'))])

In [26]: txs
Out[26]: [2785a1ad7f6d794802ea27a00e679f8c9706be0ec0b78b73d3182c551c6d69d2]

In [28]: wallet.outgoing(unconfirmed=True, confirmed=False)
Out[28]: [out: 2785a1ad7f6d794802ea27a00e679f8c9706be0ec0b78b73d3182c551c6d69d2 @␣
↪pool 3.660000000000 id=0000000000000000]

In [29]: txs[0].fee
Out[29]: Decimal('0.006282400000')
```

The fee is something you might like to verify before sending the transaction to the network. In such case you'd probably be interested in the chapter about *interaction with daemon*.

There are some additional options you may set when sending transfer, like payment ID, priority, ring size or unlock time. See API reference below for details.

Note: Be aware that transactions sent from another instance of the same wallet will not appear in mempool queries. They will, of course, become visible once mined.

4.5 API reference

class monero.transaction.**IncomingPayment**(***kwargs*)
An incoming payment (one that increases the balance of an *Account*)

class monero.transaction.**OutgoingPayment**(***kwargs*)
An outgoing payment (one that decreases the balance of an *Account*)

class monero.transaction.**Output**(***kwargs*)
A Monero one-time public output (A.K.A stealth address). Identified by *stealth_address*, or *index* and *amount* together, it can contain differing levels of information on an output.

This class is not intended to be turned into objects by the user, it is used by backends.

class monero.transaction.**Payment**(***kwargs*)
A payment base class, representing payment not associated with any *Account*.

This class is not intended to be turned into objects by the user, it is used by backends.

class monero.transaction.**PaymentFilter**(***filterparams*)
A helper class that filters payments retrieved by the backend.

This class is not intended to be turned into objects by the user, it is used by backends.

class monero.transaction.**PaymentManager**(*account_idx*, *backend*, *direction*)
A payment query manager, handling either incoming or outgoing payments of an *Account*.

This class is not intended to be turned into objects by the user, it is used by backends.

class monero.transaction.**Transaction**(***kwargs*)
A Monero transaction. Identified by *hash*, it can be a part of a block of some *height* or not yet mined (*height* is *None* then).

This class is not intended to be turned into objects by the user, it is used by backends.

outputs(*wallet=None*)
Returns a list of outputs. If wallet is given, decodes destinations and amounts for outputs directed to the wallet, provided that matching subaddresses have been already generated.

CHAPTER 5

Interacting with daemon

The module offers an interface to interact with Monero daemon. For the time being, the only available method to connnect to a daemon is by JSON RPC commands but the module allows for providing a *custom backend*. The initializer accepts keywords including, but not limited to, `host`, `port`, `user`, and `password`.

```
In [1]: from monero.daemon import Daemon

In [2]: daemon = Daemon(port=28081)

In [3]: daemon.height()
Out[3]: 1099108
```

Also, the `info()` method will return a dictionary with details about the current daemon status.

5.1 Connecting via proxy (or TOR)

`Daemon` also accepts optional `proxy_url` keyword. A prime example of use is to route your traffic via TOR:

```
In [4]: daemon = Daemon(host='xmrag4hf5xlabmob.onion', proxy_url='socks5h://127.0.0.
↪1:9050')

In [5]: daemon.height()
Out[5]: 1999790
```

Please refer to the docs of underlying requests library for more info on proxies.

5.2 Sending prepared transactions

The daemon connection may be used for two-step sending of transactions. For example, you may want to check the fee before broadcasting the transaction to the network.

To prepare a transaction, use `transfer()` or `transfer_multiple()` method of the wallet or account, as described in *the section about sending payments*. The only difference is that now you want to add the `relay=False` argument.

```
In [6]: from monero.wallet import Wallet

In [7]: from monero.backends.jsonrpc import JSONRPCWallet

In [8]: wallet = Wallet(JSONRPCWallet(port=28088))

In [9]: wallet.balance()
Out[9]: Decimal('17.642325205670')

In [10]: txs = wallet.transfer(
 ↪'Bg1nUjsEx6UUByxr68o6gXcQRF58BpQyKauoZSo2HwubGErEnz9x6AS9o5ybmk3QmgeUpX3Msgm74QkwZKx2CeVWFrrZZqt
 ↪', 10, relay=False)
```

Now the return value is a list of resulting transactions (usually just one) which may be inspected and validated.

```
In [11]: txs
Out[11]: [38964a0c8c3be041051464b413996ad8d696223dc34925d98156848ed76a3ae3]

In [12]: txs[0].fee
Out[12]: Decimal('0.003766080000')
```

If anything is not OK, just discard the transaction and create a new one. There's no need to clean up anything in the wallet.

Once you have the transaction accepted, it's time to post it to the daemon:

```
In [13]: result = daemon.send_transaction(txs[0])

In [14]: result
Out[14]:
{'double_spend': False,
 'fee_too_low': False,
 'invalid_input': False,
 'invalid_output': False,
 'low_mixin': False,
 'not_rct': False,
 'not_relayed': False,
 'overspend': False,
 'reason': '',
 'status': 'OK',
 'too_big': False}
```

5.3 No batching due to double spends

Warning: The workflow described above should not be used for preparing a batch of transactions to be sent later. The wallet doesn't remember which inputs have been spent and will very likely use the same in the next transaction, resulting in a double spend and broadcast failure.

The following example shows such behavior:

```
In [15]: txs1 = wallet.transfer(
↪'BYSXsmmK44xdjNVMGprUW5Yau9tsc9SAMJrQsANjGgpk2RB83cvVhWjZAgYNwLgmhdPawATh5q1CTEoLGKZSeZqtThefV7D
↪', 1, relay=False)

In [16]: txs2 = wallet.transfer(
↪'Bd5m5wTjWdYSaLBKe4i2avJjuFLYMEUKpiiE86F83NFiDXKE7QseSRvS7efTtJu5xHiHm5XmxgB2mfLu7NFrG7e3UTYRzEf
↪', 2, relay=False)

In [17]: txs1, txs2
Out[17]:
([315901f250a1018e89e1fc2b3953bd5acfdfa759f843cf5a38306a2255de6d54],
 [2bd978172226b486badc8a9dcbafb04acb4760c3f2a5794c694fee8575739c6e])

In [18]: daemon.send_transaction(txs1[0])
Out[18]:
{'double_spend': False,
 'fee_too_low': False,
 'invalid_input': False,
 'invalid_output': False,
 'low_mixin': False,
 'not_rct': False,
 'not_relayed': False,
 'overspend': False,
 'reason': '',
 'status': 'OK',
 'too_big': False}

In [19]: daemon.send_transaction(txs2[0])
---------------------------------------------------------------------------
TransactionBroadcastError                 Traceback (most recent call last)
<ipython-input-22-f24dc5d51c78> in <module>()
----> 1 daemon.send_transaction(txs2[0])

~/devel/monero-python/monero/daemon.py in send_transaction(self, tx, relay)
     10
     11     def send_transaction(self, tx, relay=True):
---> 12         return self._backend.send_transaction(tx.blob, relay=relay)
     13
     14     def mempool(self):

~/devel/monero-python/monero/backends/jsonrpc.py in send_transaction(self, blob,␣
↪relay)
     36         raise exceptions.TransactionBroadcastError(
     37                 "{status}: {reason}".format(**res),
---> 38                 details=res)
     39
     40     def mempool(self):

TransactionBroadcastError: Failed: double spend
```

The second transaction failed because it used the same inputs as the previous one. The daemon checks all incoming transactions for possible double-spends and rejects them if such conflict is discovered.

5.4 Other RPC Commands

Any RPC commands not available in the Daemon class, are likely available in the JSONRPCDaemon class. The official Monero Daemon RPC Documentation can be found *here <https://www.getmonero.org/resources/developer-guides/daemon-rpc.html>*. At the time of writing, all the RPC commands from the documentation have been implemented in JSONRPCDaemon, with the exception of any .bin commands, */get_txpool_backlog*, and */get_output_distribution*. These methods share the same name as their corresponding RPC names, and unlike the Daemon methods, the methods in JSONRPCDaemon are designed to be lower-level. As such, the return values of these methods reflect the raw JSON objects returned by the daemon. An example:

```
[In 20]: from monero.backends.jsonrpc import JSONRPCDaemon

[In 21]: daemon = JSONRPCDaemon(host='192.168.0.50')

[In 22]: sync_info = daemon.sync_info()

[In 23]: sync_info['height']
[Out 23]: 2304844

[In 24]: daemon.get_bans()
[Out 24]:
{
"bans": [
  {
    "host": "145.239.118.5",
    "ip": 91680657,
    "seconds": 72260
  },
  {
    "host": "146.59.156.116",
    "ip": 1956395922,
    "seconds": 69397
  }
],
"status": "OK",
"untrusted": False
}
```

5.5 API reference

CHAPTER 6

Output recognition

The module provides means to obtain output information from transactions as well as recognize and decrypt those destined to user's own wallet.

That functionality is a part of `Transaction.outputs(wallet=None)` method which may take a wallet as optional keyword, which will make it analyze outputs against all wallet's addresses. The wallet **must have the secret view key** while secret spend key is not required (which means a view-only wallet is enough).

Note: Be aware that ed25519 cryptography used there is written in pure Python. Don't expect high efficiency there. If you plan a massive analysis of transactions, please check if using Monero source code wouldn't be better for you.

Note: Please make sure the wallet you provide has all existing subaddresses generated. If you run another copy of the wallet and use subaddresses, the wallet you pass to `.outputs()` **must have the same or bigger set of subaddressses present**. For those missing from the wallet, no recognition will happen.

6.1 Output data

The method will return a set of `Output` objects. Each of them contains the following attributes:

- `stealth_address` — the stealth address of the output as hexadecimal string,
- `amount` — the amount of the output, `None` if unknown,
- `index` — the index of the output,
- `transaction` — the `Transaction` the output is a part of,
- **`payment` — a `Payment` object if the output is destined to provided wallet,** otherwise `None`,

An example usage:

```
In [1]: from monero.daemon import Daemon

In [2]: from monero.wallet import Wallet

In [3]: daemon = Daemon(port=28081)

In [4]: tx = daemon.transactions(
↪"f79a10256859058b3961254a35a97a3d4d5d40e080c6275a3f9779acde73ca8d")[0]

In [5]: wallet = Wallet(port=28088)

In [6]: outs = tx.outputs(wallet=wallet)

In [7]: outs[0].payment.local_address
Out [7]:␣
↪76Qt2xMZ3m7b2tagubEgkvG81pwf9P3JYdxR65H2BEv8c79A9pCBTacEFv87tfdcqXRemBsZLFVGHTWbqBpkoBJENBoJJS9

In [8]: outs[0].payment.amount
Out [8]: Decimal('4.000000000000')
```

CHAPTER 7

Backends

The module comes with possibility of replacing the underlying backend. Backends are the protocols and methods used to communicate with the Monero daemon or wallet. As of the time of this writing, the module offers the following options:

- `jsonrpc` for the HTTP based RPC server,
- `offline` for running the wallet without Internet connection and even without the wallet file.

7.1 JSON RPC

This backend requires a running `monero-wallet-rpc` process with a Monero wallet file opened. This can be on your local system or a remote node, depending on where the wallet file lives and where the daemon is running. Refer to the quickstart for general setup information.

The Python requests library is used in order to facilitate HTTP requests to the JSON RPC interface. It makes POST requests and passes proper headers, parameters, and payload data as per the official Wallet RPC documentation.

Also, `jsonrpc` backend is the default choice and both `Wallet` and `Daemon` classes can be invoked in a simple form with no `backend` argument given. They will assume connection to the default *mainnet* port on *localhost*, like below:

```
In [1]: wallet = Wallet()    # is equivalent to: wallet = Wallet(JSONRPCWallet(host=
↪'localhost', port=18081)
```

7.2 Offline

This backend allows creating a *Wallet* instance without network connection or even without the wallet itself. In version 0.5 the only practical use is to cold-generate *subaddresses* like in the example below:

```
In [8]: w = Wallet(OfflineWallet(
↪'47ewoP19TN7JEEnFKUJHAYhGxkeTRH82sf36giEp9AcNfDBfkAtRLX7A6rZz18bbNHPNV7ex6WYbMN3aKisFRJZ8Ebsmgef
↪', view_key='6d9056aa2c096bfcd2f272759555e5764ba204dd362604a983fa3e0aafd35901'))

In [9]: w.get_address(100,37847)
Out[9]:␣
↪883Gcsq65iqh4UL3fJTWLxY45skXyFVNQJZ4bdw4TJcqd8vafvtpX4p6HNmawqFMQ6TwJP7adzyLT1fbU6z1n9dqB9bJrfn
```

CHAPTER 8

Mnemonic seeds

You can utilize the `Seed` class in order to generate or supply a 25 word mnemonic seed. From this mnemonic seed you can derive public and private spend keys, public and private view keys, and public wallet address. Read more about mnemonic seeds here.

The class also reads 12 or 13 word seeds, also known as *MyMonero style*.

Warning: This class deals with highly sensitive strings in both inputs and outputs. The mnemonic seed and it's hexadecimal representation are essentially full access keys to your Monero funds and should be handled with the utmost care.

8.1 Generating a new seed

By default, constructing the `Seed` class without any parameters will generate a new 25 word mnemonic seed from a 32 byte hexadecimal string using `os.urandom(32)`. Class construction sets the attributes `phrase` and `hex` - the 25 word mnemonic seed and it's hexadecimal representation.

```
In  [1]: from monero.seed import Seed

In  [2]: s = Seed()

In  [3]: s.phrase
Out [3]: 'fewest lipstick auburn cocoa macro circle hurried impel macro hatchet␣
↪jeopardy swung aloof spiders gags jaws abducts buying alpine athlete junk patio␣
↪academy loudly academy'

In  [4]: s.hex
Out [4]: u'73192a945d7400a3a76a941be451a9623f37dd834006d02140a6a762b9142d80'
```

8.2 Supplying your own seed

If you have an existing mnemonic word or hexadecimal seed that you would like to derive keys for, simply pass the seed as a string to the `Seed` class. Class construction will automatically detect the seed type and encode or decode to set both `phrase` and `hex` attributes.

```
In  [1]: from monero.seed import Seed

In  [2]: s = Seed("73192a945d7400a3a76a941be451a9623f37dd834006d02140a6a762b9142d80")

In  [3]: s.phrase
Out [3]: 'fewest lipstick auburn cocoa macro circle hurried impel macro hatchet␣
↪jeopardy swung aloof spiders gags jaws abducts buying alpine athlete junk patio␣
↪academy loudly academy'

In  [4]: s.hex
Out [4]: u'73192a945d7400a3a76a941be451a9623f37dd834006d02140a6a762b9142d80'


In  [5]: p = Seed("fewest lipstick auburn cocoa macro circle hurried impel macro␣
↪hatchet jeopardy swung aloof spiders gags jaws abducts buying alpine athlete junk␣
↪patio academy loudly academy")

In  [6]: p.phrase
Out [6]: 'fewest lipstick auburn cocoa macro circle hurried impel macro hatchet␣
↪jeopardy swung aloof spiders gags jaws abducts buying alpine athlete junk patio␣
↪academy loudly academy'

In  [7]: p.hex
Out [7]: u'73192a945d7400a3a76a941be451a9623f37dd834006d02140a6a762b9142d80'
```

8.3 Deriving account keys

Once the `Seed` class is constructed, you can derive all of the keys associated with the account.

```
In  [1]: from monero.seed import Seed

In  [2]: s = Seed("fewest lipstick auburn cocoa macro circle hurried impel macro␣
↪hatchet jeopardy swung aloof spiders gags jaws abducts buying alpine athlete junk␣
↪patio academy loudly academy")

In  [3]: s.secret_spend_key()
Out [3]: '0b7a7bac8a5b6de2f483d703ef82b1bb3e37dd834006d02140a6a762b9142d00'

In  [4]: s.secret_view_key()
Out [4]: '75ec665f4912cec813ff7f20bc75b1f375ee2f8d4bb7631ae8d1af302732a609'

In  [5]: s.public_spend_key()
Out [5]: 'd5db200426637399f0076090dea01394afc2b157f94d287516911dbbcf8b2275'

In  [6]: s.public_view_key()
Out [6]: 'cd235f236224b8a5f1e12568927e01a2879bfd49cec2517b0717adb97fe8ae39'

In  [7]: s.public_address()
```

(continues on next page)

(continued from previous page)

```
Out [7]:
↪'49j9ikUyGfkSkPV8TY66p2RsSs6xL7NR5LauJTt7y6LZLhpakUnjcddUksdDgccVPEUBk2obnM1YUMaXKsGsCnow7WYjktm
↪'
```

8.4 API reference

CHAPTER 9

Miscellaneous functions, types and constants

9.1 API reference

class monero.numbers.**PaymentID**(*payment_id*)
A class that validates Monero payment ID.

Payment IDs can be used as str or int across the module, however this class offers validation as well as simple conversion and comparison to those two primitive types.

Parameters **payment_id** – the payment ID as integer or hexadecimal string

is_short()
Returns True if payment ID is short enough to be included in *IntegratedAddress*.

monero.numbers.**as_monero**(*amount*)
Return the amount rounded to maximal Monero precision.

monero.numbers.**from_atomic**(*amount*)
Convert atomic integer of piconero to Monero decimal.

monero.numbers.**to_atomic**(*amount*)
Convert Monero decimal to atomic integer of piconero.

CHAPTER 10

Exceptions

exception monero.exceptions.**AccountException**

exception monero.exceptions.**AccountIndexOutOfBound**

exception monero.exceptions.**AddressIndexOutOfBound**

exception monero.exceptions.**AmountIsZero**

exception monero.exceptions.**BackendException**

exception monero.exceptions.**DaemonIsBusy**

exception monero.exceptions.**GenericTransferError**

exception monero.exceptions.**MoneroException**

exception monero.exceptions.**NoDaemonConnection**

exception monero.exceptions.**NotEnoughMoney**

exception monero.exceptions.**NotEnoughUnlockedMoney**

exception monero.exceptions.**SignatureCheckFailed**

exception monero.exceptions.**TransactionBroadcastError**(*message*, *details=None*)

exception monero.exceptions.**TransactionIncomplete**

exception monero.exceptions.**TransactionNotFound**

exception monero.exceptions.**TransactionNotPossible**

exception monero.exceptions.**TransactionWithoutBlob**

exception monero.exceptions.**TransactionWithoutJSON**

exception monero.exceptions.**WalletIsNotDeterministic**

exception monero.exceptions.**WalletIsWatchOnly**

exception monero.exceptions.**WrongAddress**

exception monero.exceptions.**WrongPaymentId**

CHAPTER 11

Release Notes

11.1 1.1.0

This version doesn't contain any major changes but drops support for Python 2 altogether.

11.2 1.0.2

A release with critical security fix. All since 0.99 (inclusively) are compromised and should be never used again.

11.3 1.0

A release with no significant changes from 0.99

11.4 0.99

This is a test release before 1.0. The reference library for Ed25519 cryptography has been dropped and replaced with pynacl which is a wrapper over libsodium, the industry standard lightning-fast C library.

There are no backward-incompatible changes in the API. The aim is to have the software tested thoroughly before the first stable release.

11.5 0.9

The hashing library `sha3` has been replaced by `pycryptodomex` which is a more actively maintained project. However, the code still may work with the old `sha3` module. Just ignore the new dependency and run as usual.

11.6 0.8

Backward-incompatible changes:

1. The `monero.prio` submodule has been removed. Switch to `monero.const`.
2. Methods `.is_mainnet()`, `.is_testnet()`, `.is_stagenet()` have been removed from `monero.address.Address` instances. Use `.net` attribute instead.

11.7 0.7

Backward-incompatible changes:

1. The `Transaction.blob` changes from hexadecimal to raw binary data (`bytes` in Python 3, `str` in Python 2).

Deprecations:

1. `monero.const` has been introduced. Transaction priority consts will move to `monero.const.PRIO_*`. The `monero.prio` submodule has been deprecated and will be gone in 0.8.
2. Methods `.is_mainnet()`, `.is_testnet()`, `.is_stagenet()` have been deprecated and new `.net` property has been added to all `monero.address.Address` instances. The values are from among `monero.const.NET_*` and have string representation of `"main"`, `"test"` and `"stage"` respectively. Likewise, `monero.seed.Seed.public_address()` accepts those new values. All deprecated uses will raise proper warnings in 0.7.x and will be gone with 0.8.

11.8 0.6

With version 0.6 the package name on PyPi has changed from *monero-python* to just *monero*.

Backward-incompatible changes:

1. The `.new_address()` method of both `Wallet` and `Account` returns a 2-element tuple of (*subaddress*, *index*) where the additional element is the index of the subaddress within current account.

11.9 0.5

Backward-incompatible changes:

1. The `ringsize` parameter is gone from `.transfer()` and `.transfer_multiple()` methods of both `Wallet` and `Account`. Since Monero 0.13 the ring size is of constant value 11.
2. The class hierarchy in `monero.address` has been reordered. `Address` now represents only master address of a wallet. `SubAddress` doesn't inherit after it anymore, but all classes share the common base of `BaseAddress`.

   In particular, make sure that your code doesn't check a presence of Monero address by checking `isinstance(x, monero.address.Address)`. That will not work for sub-addresses anymore. Replace it by `isinstance(x, monero.address.BaseAddress)`.

CHAPTER 12

License

CHAPTER 13

Authors

- Michał Sałaban <michal@salaban.info>
- MoneroPy Developers (`monero/base58.py` taken from MoneroPy)
- thomasv@gitorious (`monero/seed.py` based on Electrum)
- and other Contributors: lalanza808, cryptochangements34, atward, rooterkyberian, brucexiu, lialsoftlab, moneroexamples, massanchik, MrClottom, jeffro256, sometato, kayabaNerve, j-berman.

13.1 Acknowledgements

This project has been generously funded by the donors of Monero Forum Funding System. You may see the original project submission.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

W