# dposlib

*Release 0.3.4*

Dec 19, 2020

# Contents

Release v0.3.4 - (*Installation*)

`dposlib` is a simple package providing efficient API to interact with Ark blockchain and its forks. It is designed to run with both python 2 and 3.

Simplicity of `REST` API:

```
>>> from dposlib import rest
>>> # ~ https://explorer.ark.io:8443/api/delegates/arky
>>> rest.GET.api.delegates.arky(peer="https://explorer.ark.io:8443")
{u'data': {u'username': u'arky', u'votes': u'172572088664599', u'blocks': {u'produced
↪': 199859, u'last': {u'timestamp': {u'epoch': 84182056, u'unix': 1574283256, u'human
↪': u'2019-11-20T20:54:16.000Z'}, u'id': u
↪'5f5f9897f8fca2a5600ace0d75d67811c67df8111a7deea13d7d6b2c532fae43', u'height':␣
↪10380869}}, u'rank': 11, u'publicKey': u
↪'030da05984d579395ce276c0dd6ca0a60140a3c3d964423a04e7abe110d60a15e9', u'production
↪': {u'approval': 1.35}, u'forged': {u'total': u'40118247659340', u'rewards': u
↪'39687400000000', u'fees': u'430847659340'}, u'address': u
↪'ARfDVWZ7Zwkox3ZXtMQQY1HYSANMB88vWE'}}
>>> # using returnKey arktoshi values are converted to ark
>>> rest.GET.api.transactions(peer="https://explorer.ark.io:8443", returnKey="data
↪")[0]
{u'fee': 0.00816, u'type': 0, u'sender': u'AKATy581uXWrbm8B4DTQh4R9RbqaWRiKRY', u
↪'timestamp': {u'epoch': 84182307, u'unix': 1574283507, u'human': u'2019-11-
↪20T20:58:27.000Z'}, u'blockId': u
↪'a1b305a87217c2f622a922a97a778c677f7dbd23031dae42e3b494883b855a70', u'vendorField':␣
↪u'Payout from arkmoon', u'senderPublicKey': u
↪'0232b96d57ac27f9a99242bc886e433baa89f596d435153c9dae47222c0d1cecc3', u'amount': 20.
↪52064264, u'version': 1, u'signSignature': u
↪'304402200ac41802f33a5f377975efc9ebf39a666a9d76c2facb8773783289df7f6a9cd302206c5d2aed3359d3858fb3f4
↪', u'confirmations': 21, u'signature': u
↪'3045022100dc6dbaa4b056f10268b587da290900725246e3239df1fa3e3c53445da36f03ee02206d57bbdff6d7f9ebca71
↪', u'recipient': u'AXPLW2TzBsXcPiaeVGBSELEAXj4RPaWNjB', u'id': u
↪'efeab09925c3347b4a18854a9192d7d722ee32850a7bf91d57628cb77714192e'}
>>> # peer keyword is not mandatory when a blockchain is linked using rest.use␣
↪directive
>>> rest.use("ark")
>>> # ~ GET /api/blocks endpoint
>>> rest.GET.api.blocks(returnKey="data")[0]
{u'payload': {u'length': 0, u'hash': u
↪'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855'}, u'generator':
↪{u'username': u'arkmoon', u'publicKey': u
↪'0232b96d57ac27f9a99242bc886e433baa89f596d435153c9dae47222c0d1cecc3', u'address': u
↪'AKATy581uXWrbm8B4DTQh4R9RbqaWRiKRY'}, u'transactions': 0, u'timestamp': {u'epoch':␣
↪84183376, u'unix': 1574284576, u'human': u'2019-11-20T21:16:16.000Z'}, u'height':␣
↪10381034, u'version': 0, u'forged': {u'fee': 0.0, u'amount': 0.0, u'total': 2.0, u
↪'reward': 2.0}, u'confirmations': 1, u'signature': u
↪'3045022100a8b6b48c0094f9c84b7da5ae457ca33d5ba0d9a3df963c1e17c42cb52fb563a9022020ea96cf76529943b03b
↪', u'id': u'd2e042495ab64e7cf5bb0fc8d4ce6972a98f29a56d960b707f3c6abd2791a5e2', u
↪'previous': u'ea1b7082424592545860a671a77ef7f59c3730665208080d2481e363be6c1ed0'}
```

`ECDSA` and `SCHNORR` signatures can be performed using *`dposlib.ark.sig`* and *`dposlib.ark.crypto`* modules:

```
>>> import dposlib.ark.sig as sig
>>> import dposlib.ark.crypto as crypto
>>> keys = crypto.getKeys("secret")
>>> keys
{'publicKey': '03a02b9d5fdd1307c2ee4652ba54d492d1fd11a7d1bb3f3a44c4a05e79f19de933',
 ↪'privateKey': '2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b',
 ↪'wif': 'SB3BGPGRh1SRuQd52h7f5jsHUg1G9ATEvSeA7L5Bz4qySQww4k7N'}
>>> s = sig.Signature.ecdsa_sign("simple message", keys["privateKey"])
>>> s
<secp256k1 signature:
  r:d811a0321a2e31b0492c1b1b1c4dc3b58055b53cdc9308492b3de71c765f5914
  s:4747219a0d74d49a42305c040a91e6a8acd39e6d06b21ec1805bd31c6d871b4f
>
>>> s.der
b"0D\x02 N\x13\x108J\xd0\xd6\xff\x80'\xf2\xf8`\xd6(\xb2\xa6@\x03\x0bF
 ↪#\xa3\x93\xe1\xdf&\xf7\xdd\xce\\u\x02 g\x8b\xa9\x90V\xaa\xdf\xa7\xf2-;z\xa5.
 ↪D\x8bq8ehG\xb7\x11\x07-`\xd2\xd9\xd3.\xc4v"
>>> crypto.hexlify(s.der)

 ↪'3044022041e5aa3da79523a2b342180cb7c04056f8f02e005ea6ec1f14094c66d692f04402200261177cdd88525249a061
 ↪'
>>> crypto.hexlify(s.raw)

 ↪'4e1310384ad0d6ff8027f2f860d628b2a640030b4623a393e1df26f7ddce5c75678ba99056aadfa7f22d3b7aa52e448b71
 ↪'
>>> crypto.hexlify(sig.Signature.schnorr_sign("simple message", keys["privateKey"]).
 ↪raw)

 ↪'5fbb0bb00b043400e1fc435c867c738ac80d2c268cd2d61616785315ad330c884a3cfb50bf0da8de9021d42ce2139b6b65
 ↪'
```

`dposlib.ark.v2` package provides *dposlib.blockchain.Transaction* class and its associated builders:

```
>>> from dposlib import rest
>>> rest.use("dark")
True
>>> from dposlib.ark.v2 import *
>>> tx = transfer(1, "D7seWn8JLVwX4nHd9hh2Lf7gvZNiRJ7qLk", u"simple message with␣
 ↪sparkle \u2728", version=2)
>>> tx.finalize("first secret", "second secret")
>>> tx
{
    "amount": 100000000,
    "asset": {},
    "expiration": 0,
    "fee": 4013642,
    "id": "041ad1e3dd06d29ef59b2c7e19fea4ced0e7fcf9fdc22edcf26e5cc016e10f38",
    "network": 30,
    "nonce": 377,
    "recipientId": "D7seWn8JLVwX4nHd9hh2Lf7gvZNiRJ7qLk",
    "senderId": "D7seWn8JLVwX4nHd9hh2Lf7gvZNiRJ7qLk",
    "senderPublicKey":
 ↪"03a02b9d5fdd1307c2ee4652ba54d492d1fd11a7d1bb3f3a44c4a05e79f19de933",
    "signSignature":
 ↪"3d29356c77b63c2d6ce679dad95961b40ea606823bf729a158df5c8378c79c5588ad675ee147a7f77b18518c5bdf9b1a7
 ↪",
    "signature":
 ↪"871ac31e7bad08b684b27f1b8a4b9f9f760bb32d1d36cc03e03872edc6070f8d9fec2621ea87e2ea0ae7750e0e7a5db52
 ↪",
```

(continues on next page)

(continued from previous page)

```
    "timestamp": 84186531,
    "type": 0,
    "typeGroup": 1,
    "vendorField": "simple message with sparkle \u2728",
    "version": 2
}
>>> broadcastTransactions(tx)
{u'data': {u'broadcast': [u
↪'041ad1e3dd06d29ef59b2c7e19fea4ced0e7fcf9fdc22edcf26e5cc016e10f38'], u'invalid': [],
↪ u'accept': [u'041ad1e3dd06d29ef59b2c7e19fea4ced0e7fcf9fdc22edcf26e5cc016e10f38'], u
↪'excess': []}}
```

See the transaction in devnet explorer

CHAPTER 1

# Install

## 1.1 Get the Source Code

`dposlib` is developed on GitHub, where the code is always available. You can either clone the public repository:

```
$ git clone git://github.com/Moustikitos/dpos.git
```

You can also download the zip. `dposlib` will be available if zip file is added as is in python pathes.

## 1.2 Install `dposlib` using `pip`

To install last version of `dposlib`:

```
$ pip install dposlib
```

To install development vesion:

```
$ pip install git+https://github.com/Moustikitos/dpos#egg=dposlib
```

You may whant to install a specific branch of dposlib:

```
$ pip install git+https://github.com/Moustikitos/dpos#egg=dposlib@<branch>
```

**Where `<branch>` can be:**

- a commit number
- a repo branch name
- a release number

## 1.3 Deploy a multisignature server

Install developpement version:

```
$ bash <(curl -s https://raw.githubusercontent.com/Moustikitos/dpos/master/bash/mssrv-
↪install.sh)
```

Once `dpos` repository cloned, there is no need to install dposlib because python pathes are set accordingly.

Deploy using `flask` server:

```
$ . ~/.local/share/ms-server/venv/bin/activate
$ export PYTHONPATH=${PYTHONPATH}:${HOME}/dpos
$ python ~/dpos/mssrv/srv.py
```

Deploy using `gunicorn` server:

```
$ . ~/.local/share/ms-server/venv/bin/activate
$ export PYTHONPATH=${PYTHONPATH}:${HOME}/dpos
$ gunicorn --bind=0.0.0.0:5050 --workers=5 mssrv:app
```

Deploy using `ms` command:

```
$ # activate virtual environment
$ bash ~/dpos/bash/activate
$ ./ms --help
$ Usage:
$    ms start-api [-p <api-port>]
$    ms start-app [-p <port> -s <server>]
$    ms (restart-api | restart-app | stop-api | stop-app)
$    ms (log-api | log-app)
$
$ Options:
$ -p --port=<port>       : the port to use [default: 5050]
$ -s --server=<server>   : the ms-api server to link to [default: http://127.0.0.
↪1:5050]
$
$ Subcommands:
$    start-api     : start multi signature server
$    start-app     : start multi signature app
$    restart-app/api : restart multi signature api/app
$    stop-api/app  : stop multi signature server/app
$    log-api/app   : show multi signature server/app logs
```

# CHAPTER 2

## Secp256k1 curve package

Pure python implementation for `scp256k1` curve algebra and associated `ECDSA - SCHNORR` signatures.

```
>>> from dposlib.ark import secp256k1
>>> G = secp256k1.
↪Point(0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798)
>>> G.y
32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> G
<secp256k1 point:
  x:79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
  y:483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
>
>>> G+G == 2*G
True
```

```
>>> secp256k1.PublicKey.from_int(secp256k1.int_from_bytes(secp256k1.hash_sha256(
↪"secret")))
<secp256k1 public key:
  x:a02b9d5fdd1307c2ee4652ba54d492d1fd11a7d1bb3f3a44c4a05e79f19de933
  y:924aa2580069952b0140d88de21c367ee4af7c4a906e1498f20ab8f62e4c2921
>
>>> secp256k1.PublicKey.from_seed(secp256k1.hash_sha256("secret"))
<secp256k1 public key:
  x:a02b9d5fdd1307c2ee4652ba54d492d1fd11a7d1bb3f3a44c4a05e79f19de933
  y:924aa2580069952b0140d88de21c367ee4af7c4a906e1498f20ab8f62e4c2921
>
>>> secp256k1.PublicKey.from_secret("secret")
<secp256k1 public key:
  x:a02b9d5fdd1307c2ee4652ba54d492d1fd11a7d1bb3f3a44c4a05e79f19de933
  y:924aa2580069952b0140d88de21c367ee4af7c4a906e1498f20ab8f62e4c2921
>
```

**Sources:**

- BIP schnorr

- Python reference
- Bcrypto 4.10 schnorr scheme

**Variables:**

- `secret` (`str`): passphrase
- `secret0` (`bytes`): private key
- `P` (`list`): public key as `secp256k1` curve point
- `pubkey` (`bytes`): compressed - encoded public key
- `pubkeyB` (`bytes`): compressed - encoded public key according to bip schnorr spec
- `msg` (`bytes`): sha256 hash of message to sign
- Uppercase variables refer to points on the curve with equation $y^2=x^3+7$ over the integers modulo p

**class** dposlib.ark.secp256k1.**Point**(**xy*)

`secp256k1` point . Initialization can be done with sole `x` value. *Point* overrides `*` and `+` operators which accepts `list` as argument and returns *Point*.

**static decode**(*pubkey*)

See *point_from_encoded()*.

**encode**()

See *encoded_from_point()*.

**x**

Return list item #0

**y**

Return list item #1

**class** dposlib.ark.secp256k1.**PublicKey**(**xy*)

*Point* extension providing specific initialization methods.

**static from_int**(*value*)

Compute a public key from `int` value.

**Parameters** **value** (`int`) – scalar to use

**Returns** the public key

**Return type** *PublicKey*

**static from_secret**(*secret*)

Compute a public key from secret passphrase.

**Parameters** **value** (`str`) – secret passphrase to use

**Returns** the public key

**Return type** *PublicKey*

**static from_seed**(*seed*)

Compute a public key from `bytes` value.

**Parameters** **value** (`bytes`) – bytes sequence to use

**Returns** the public key

**Return type** *PublicKey*

dposlib.ark.secp256k1.**der_from_sig**(*r*, *s*)

Encode a signature according `DER` spec.

**Parameters**

- **r** (int) – signature part #1
- **s** (int) – signature part #2

**Returns** encoded signature

**Return type** bytes

dposlib.ark.secp256k1.**encoded_from_point**(*P*)

**Encode and compress a secp256k1 point:**

- bytes(2) || bytes(x) if y is even
- bytes(3) || bytes(x) if y is odd

**Parameters** **P** (list) – secp256k1 point

**Returns** compressed and encoded point

**Return type** bytes

dposlib.ark.secp256k1.**hash_sha256**(*b*)

**Parameters** **b** (bytes or str) – sequence to be hashed

**Returns** sha256 hash

**Return type** bytes

dposlib.ark.secp256k1.**point_add**(*P1*, *P2*)

Add secp256k1 points.

**Parameters**

- **P1** (list) – first secp256k1 point
- **P2** (list) – second secp256k1 point

**Returns** secp256k1 point

**Return type** list

dposlib.ark.secp256k1.**point_from_encoded**(*pubkey*)

Decode and decompress a secp256k1 point.

**Parameters** **pubkey** (bytes) – compressed and encoded point

**Returns** secp256k1 point

**Return type** list

dposlib.ark.secp256k1.**point_mul**(*P*, *n*)

Multiply secp256k1 point with scalar.

**Parameters**

- **P** (list) – secp256k1 point
- **n** (int) – scalar

**Returns** secp256k1 point

**Return type** list

dposlib.ark.secp256k1.**rand_k**()

Generate a random nonce.

dposlib.ark.secp256k1.**rfc6979_k**(*msg*, *secret0*, *V=None*)
Generate a deterministic nonce according to rfc6979 spec.

**Parameters**

- **msg** (bytes) – 32-bytes sequence
- **secret0** (bytes) – private key
- **V** (bytes) –

**Returns** deterministic nonce

**Return type** int

dposlib.ark.secp256k1.**sig_from_der**(*der*)
Decode a DER signature.

**Parameters** **der** (bytes) – encoded signature

**Returns** signature (r, s)

**Return type** (int, int)

dposlib.ark.secp256k1.**tagged_hash**(*tag*, *msg*)
Return sha256(sha256(tag) || sha256(tag) || msg). Tagged hash are registered to speed up code execution.

**Parameters**

- **tag** (str) – tag to use
- **msg** (bytes) – sha256 hash of message to sign

**Returns** tagged hash

**Return type** bytes

dposlib.ark.secp256k1.**x**(*P*)
Return P.x or P[0].

**Parameters** **P** (list) – secp256k1 point

**Returns** x

**Return type** int

dposlib.ark.secp256k1.**y**(*P*)
Return P.y or P[1].

**Parameters** **P** (list) – secp256k1 point

**Returns** y

**Return type** int

dposlib.ark.secp256k1.**y_from_x**(*x*)
Compute P.y from P.x according to $y^2=x^3+7$.

## 2.1 ECDSA signatures

dposlib.ark.secp256k1.ecdsa.**rfc6979_sign**(*msg*, *secret0*, *canonical=True*)
Generate signature according to ECDSA scheme using a RFC-6979 nonce

**Parameters**

- **msg** (bytes) – sha256 message-hash
- **secret0** (bytes) – private key
- **canonical** (bool) – canonalize signature

**Returns** DER signature

**Return type** bytes

dposlib.ark.secp256k1.ecdsa.**sign**(*msg*, *secret0*, *k=None*, *canonical=True*)
Generate signature according to ECDSA scheme.

**Parameters**

- **msg** (bytes) – sha256 message-hash
- **secret0** (bytes) – private key
- **k** (int) – nonce (random nonce used if k=None)
- **canonical** (bool) – canonalize signature

**Returns** DER signature

**Return type** bytes

dposlib.ark.secp256k1.ecdsa.**verify**(*msg*, *pubkey*, *sig*)
Check signature according to ECDSA scheme.

**Parameters**

- **msg** (bytes) – sha256 message-hash
- **pubkey** (bytes) – encoded public key
- **sig** (bytes) – signature

**Returns** True if match

**Return type** bool

## 2.2 Schnorr signatures

dposlib.ark.secp256k1.schnorr.**bcrypto410_sign**(*msg*, *seckey0*)
Generate message signature according to Bcrypto 4.10 schnorr spec.

**Parameters**

- **msg** (bytes) – sha256 message-hash
- **secret0** (bytes) – private key

**Returns** RAW signature

**Return type** bytes

dposlib.ark.secp256k1.schnorr.**bcrypto410_verify**(*msg*, *pubkey*, *sig*)
Check if public key match message signature according to Bcrypto 4.10 schnorr spec.

**Parameters**

- **msg** (bytes) – sha256 message-hash
- **pubkey** (bytes) – encoded public key
- **sig** (bytes) – signature

**Returns** True if match

**Return type** bool

dposlib.ark.secp256k1.schnorr.**bytes_from_point**(*P*)
Encode a public key as defined in BIP schnorr spec.

**Parameters** **P** (Point) – secp256k1 curve point

**Returns** encoded public key

**Return type** bytes

dposlib.ark.secp256k1.schnorr.**point_from_bytes**(*pubkeyB*)
Decode a public key as defined in BIP schnorr spec.

**Parameters** **pubkeyB** (bytes) – encoded public key

**Returns** secp256k1 curve point

**Return type** Point

dposlib.ark.secp256k1.schnorr.**sign**(*msg*, *seckey0*)
Generate message signature according to BIP schnorr spec.

**Parameters**

- **msg** (bytes) – sha256 message-hash
- **seckey0** (bytes) – private key

**Returns** RAW signature

**Return type** bytes

dposlib.ark.secp256k1.schnorr.**verify**(*msg*, *pubkey*, *sig*)
Check if public key match message signature according to BIP schnorr spec.

**Parameters**

- **msg** (bytes) – sha256 message-hash
- **pubkey** (bytes) – encoded public key
- **sig** (bytes) – signature

**Returns** True if match

**Return type** bool

# CHAPTER 3

## Very easy start

`rest` module provides network loaders and root `EndPoint` `GET`, `POST`, `PUT` and `DELETE`. See Ark API documentation to see how to use http calls.

`rest` also creates a core module containing `Transaction` builders, `crypto` and `api` modules.

```
>>> from dposlib import rest
>>> rest.use("ark")
True
>>> import dposlib
>>> dlgt = dposlib.core.api.Delegate("arky")
>>> dlgt.forged
{u'rewards': 397594.0, u'total': 401908.71166083, u'fees': 4314.71166083}
>>> dposlib.core.crypto.getKeys("secret")
{'publicKey': '03a02b9d5fdd1307c2ee4652ba54d492d1fd11a7d1bb3f3a44c4a05e79f19de933',
↪'privateKey': '2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b',
↪'wif': 'SB3BGPGRh1SRuQd52h7f5jsHUg1G9ATEvSeA7L5Bz4qySQww4k7N'}
>>> dposlib.core.transfer(1, "ARfDVWZ7Zwkox3ZXtMQQY1HYSANMB88vWE", u"\u2728 simple␣
↪transfer vendorField")
{
  "amount": 100000000,
  "asset": {},
  "recipientId": "ARfDVWZ7Zwkox3ZXtMQQY1HYSANMB88vWE",
  "type": 0,
  "vendorField": "\u2728 simple transfer vendorField",
  "version": 1
}
>>> dposlib.core.htlcLock(1, "ARfDVWZ7Zwkox3ZXtMQQY1HYSANMB88vWE", "my secret lock",␣
↪expiration=12, vendorField=u"\u2728 simple htlcLock vendorField")
{
  "amount": 100000000,
  "asset": {
    "lock": {
      "secretHash": "dbaed2f2747c7aa5a834b082ccb2b648648758a98d1a415b2ed9a22fd29d47cb
↪",
      "expiration": {
```

(continues on next page)

(continued from previous page)

```
        "type": 1,
        "value": 82567745
      }
    }
  },
  "network": 23,
  "recipientId": "ARfDVWZ7Zwkox3ZXtMQQY1HYSANMB88vWE",
  "type": 8,
  "typeGroup": 1,
  "vendorField": "\u2728 simple htlcLock vendorField",
  "version": 2
}
```

dposlib.rest.**load**(*name*)

Load a given blockchain package as `dposlib.core` module. A valid blockchain package must provide `init(peer=None)()` and `stop()` definitions. Available blockchains are referenced in `dposli.net` module.

**Parameters** **name** (`str`) – package name to load

dposlib.rest.**use**(*network*, ***kwargs*)

Sets the blockchain parameters in the `cfg` module and initialize blockchain package. Network options can be created or overriden using `**kwargs` argument.

**Parameters** **network** (`str`) – network to initialize

**Returns** True if network connection established

**Return type** `bool`

CHAPTER 4

# Blockchain core

## 4.1 Transaction class

**class** dposlib.blockchain.**Transaction**(**args*, ***kwargs*)

A python `dict` that implements all the necessities to manually generate valid transactions.

**dump**()

Dumps transaction in registry.

**feeExcluded**()

Arrange `amount` and `fee` values so the total `satoshi` flow is the desired spent plus the fee.

**feeIncluded**()

Arrange `amount` and `fee` values so the total `satoshi` flow is the desired spent.

**finalize**(*secret=None*, *secondSecret=None*, *fee=None*, *fee_included=False*)

Finalize a transaction by setting `fee`, signatures and `id`.

**Parameters**

- **secret** (`str`) – passphrase
- **secondSecret** (`str`) – second passphrase
- **fee** (`int`) – manually set fee value in `satoshi`
- **fee_included** (`bool`) – see *feeIncluded() feeExcluded()*

**identify**()

Generate the `id` field. Transaction have to be signed.

**link**(*secret=None*, *secondSecret=None*)

Save public and private keys derived from secrets. This is equivalent to wallet login. it limits number of secret keyboard entries.

**Parameters**

- **secret** (`str`) – passphrase

- **secondSecret** (str) – second passphrase

**load**(*txid*)
Loads the transaction identified by txid from registry.

**multiSignWithKey**(*privateKey*)
Add a signature in signatures field according to given index and privateKey.

**Parameters** **privateKey** (str) – private key as hex string

**multiSignWithSecret**(*secret*)
Add a signature in signatures field.

**Parameters**

- **index** (int) – signature index
- **secret** (str) – passphrase

**path**()
Return current registry path.

**static setDynamicFee**(*value='minFee'*)
Activate and configure dynamic fees parameters. Value can be either an integer defining the fee multiplier constant or a string defining the fee level to use acccording to the 30-days-average. possible values are avgFee minFee (default) and maxFee.

**Parameters** **value** (str or int) – constant or fee multiplier

**setFee**(*value=None*)
Set fee field manually or according to inner parameters.

**Parameters** **value** (int) – fee value in statoshi to set manually

**static setStaticFee**()
Deactivate dynamic fees.

**sign**()
Generate the signature field. Private key have to be set first. See *link()*.

**signSign**()
Generate the signSignature field. Transaction have to be signed and second private key have to be set first. See *link()*.

**signSignWithKey**(*secondPrivateKey*)
Generate the signSignature field using second private key. It is stored till unlink() is called.

**Parameters** **secondPrivateKey** (str) – second private key as hex string

**signSignWithSecondSecret**(*secondSecret*)
Generate the signSignature field using second passphrase. The associated second public and private keys are stored till unlink() is called.

**Parameters** **secondSecret** (str) – second passphrase

**signWithKeys**(*publicKey*, *privateKey*)
Generate the signature field using public and private keys. They are till unlink() is called.

**Parameters**

- **publicKey** (str) – public key as hex string
- **privateKey** (str) – private key as hex string

**signWithSecret**(*secret*)
Generate the `signature` field using passphrase. The associated public and private keys are stored till `unlink()` is called.

**Parameters** **secret** (`str`) – passphrase

**static useDynamicFee**(*value='minFee'*)
Activate and configure dynamic fees parameters. Value can be either an integer defining the fee multiplier constant or a string defining the fee level to use acccording to the 30-days-average. possible values are `avgFee` `minFee` (default) and `maxFee`.

**Parameters** **value** (`str` or `int`) – constant or fee multiplier

**static useStaticFee**()
Deactivate dynamic fees.

## 4.2 Crypto utils

dposlib.ark.crypto.**checkTransaction**(*tx*, *secondPublicKey=None*, *multiPublicKeys=[]*)
Verify transaction validity.

**Parameters**

- **tx** (`dict` or `Transaction`) – transaction object
- **secondPublicKey** (`str`) – second public key to use if needed
- **multiPublicKeys** (`list`) – owners public keys (sorted according to associated type-4-tx asset)

**Returns** true if transaction is valid

**Return type** `bool`

dposlib.ark.crypto.**getAddress**(*publicKey*, *marker=None*)
Compute ARK address from publicKey.

**Parameters**

- **publicKey** (`str`) – public key
- **marker** (`int`) – network marker (optional)

**Returns** the address

**Return type** `str`

dposlib.ark.crypto.**getAddressFromSecret**(*secret*, *marker=None*)
Compute ARK address from secret.

**Parameters**

- **secret** (`str`) – secret string
- **marker** (`int`) – network marker (optional)

**Returns** the address

**Return type** `str`

dposlib.ark.crypto.**getBytes**(*tx*, ***options*)
Hash transaction.

**Parameters** **tx** (`dict` or `Transaction`) – transaction object

**Keyword Arguments**

- **exclude_sig** (bool) – exclude signature during tx serialization [defalut: True]
- **exclude_multi_sig** (bool) – exclude signatures during tx serialization [defalut: True]
- **exclude_second_sig** (bool) – exclude second signatures during tx serialization [defalut: True]

**Returns** bytes sequence

**Return type** bytes

dposlib.ark.crypto.**getId**(*tx*)
Generate transaction id.

**Parameters** **tx** (dict or Transaction) – transaction object

**Returns** id as hex string

**Return type** str

dposlib.ark.crypto.**getIdFromBytes**(*data*)
Generate data id.

**Parameters** **data** (bytes) – data as bytes sequence

**Returns** id as hex string

**Return type** str

dposlib.ark.crypto.**getKeys**(*secret*)
Generate keyring containing secp256k1 keys-apir and wallet import format (WIF).

**Parameters** **secret** (str, bytes or int) – anything that could issue a private key on secp256k1 curve

**Returns** public, private and WIF keys

**Return type** dict

dposlib.ark.crypto.**getMultiSignaturePublicKey**(*minimum*, **publicKeys*)
Compute ARK multi signature public key according to ARK AIP #18.

**Parameters**

- **minimum** (int) – minimum signature required
- **publicKeys** (list of str) – public key list

**Returns** the multisignature public key

**Return type** str

dposlib.ark.crypto.**getSignature**(*tx*, *privateKey*, ***options*)
Generate transaction signature using private key.

**Parameters**

- **tx** (dict or Transaction) – transaction description
- **privateKey** (str) – private key as hex string

**Keyword Arguments**

- **exclude_sig** (bool) – exclude signature during tx serialization [defalut: True]

- **exclude_multi_sig** (bool) – exclude signatures during tx serialization [defalut: True]
- **exclude_second_sig** (bool) – exclude second signatures during tx serialization [defalut: True]

**Returns** signature

**Return type** str

dposlib.ark.crypto.**getSignatureFromBytes**(*data*, *privateKey*)
Generate signature from data using private key.

**Parameters**

- **data** (bytes) – bytes sequence
- **privateKey** (str) – private key as hex string

**Returns** signature as hex string

**Return type** str

dposlib.ark.crypto.**getWIF**(*seed*)
Compute WIF address from seed.

**Parameters** **seed** (bytes) – a sha256 sequence bytes

**Returns** WIF address

**Return type** str

dposlib.ark.crypto.**serialize**(*tx*, *version=None*, ***options*)
Serialize transaction.

**Parameters** **tx** (dict or Transaction) – transaction object

**Returns** bytes sequence

**Return type** bytes

dposlib.ark.crypto.**verifySignature**(*value*, *publicKey*, *signature*)
Verify signature.

**Parameters**

- **value** (str) – value as hex string
- **publicKey** (str) – public key as hex string
- **signature** (str) – signature as hex string

**Returns** true if signature matches the public key

**Return type** bool

dposlib.ark.crypto.**verifySignatureFromBytes**(*data*, *publicKey*, *signature*)
Verify signature.

**Parameters**

- **data** (bytes) – data
- **publicKey** (str) – public key as hex string
- **signature** (str) – signature as hex string

**Returns** true if signature matches the public key

**Return type** bool

dposlib.ark.crypto.**wifSignature**(*tx*, *wif*)
Generate transaction signature using private key.

**Parameters**

- **tx** (dict or Transaction) – transaction description
- **wif** (str) – wif key

**Returns** signature

**Return type** str

dposlib.ark.crypto.**wifSignatureFromBytes**(*data*, *wif*)
Generate signature from data using WIF key.

**Parameters**

- **data** (bytes) – bytes sequence
- **wif** (str) – wif key

**Returns** signature

**Return type** str

## 4.3 Signature utils

Advanced signature manipulation. It is the recomended module to manually issue signatures for ark blockchain and forks.

**Variables:**

- privateKey (str): hexlified private key
- publicKey (str): hexlified compressed - encoded public key
- message (str): message to sign as string

**class** dposlib.ark.sig.**Signature**(**rs*)

**static b410_schnorr_sign**(*message*, *privateKey*)
Generate message signature according to Bcrypto 4.10 schnorr scheme.

**Parameters**

- **message** (str) – message to verify
- **privateKey** (str) – private key

**Returns** signature

**Return type** *Signature*

**b410_schnorr_verify**(*message*, *publicKey*)
Check if public key match message signature according to Bcrypto 4.10 schnorr scheme.

**Parameters**

- **message** (str) – message to verify
- **publicKey** (str) – public key

**Returns** True if match

**Return type** bool

**der**
Return DER encoded signature as bytes sequence

**static ecdsa_rfc6979_sign**(*message*, *privateKey*, *canonical=True*)
Generate message signature according to ECDSA scheme using a deterministic nonce (RFC-6976).

**Parameters**

- **message** (str) – message to verify
- **privateKey** (str) – private key
- **canonical** (bool) – canonalize signature

**Returns** signature

**Return type** *Signature*

**static ecdsa_sign**(*message*, *privateKey*, *canonical=True*)
Generate message signature according to ECDSA scheme using a random nonce.

**Parameters**

- **message** (str) – message to verify
- **privateKey** (str) – private key
- **canonical** (bool) – canonalize signature

**Returns** signature

**Return type** *Signature*

**ecdsa_verify**(*message*, *publicKey*)
Check if public key match message signature according to ECDSA scheme.

**Parameters**

- **message** (str) – message to verify
- **publicKey** (str) – public key

**Returns** True if match

**Return type** bool

**static from_der**(*der*)
Decode signature from DER encoded bytes sequence.

**Parameters** **der** (bytes) – encoded signature

**Returns** signature

**Return type** *Signature*

**static from_raw**(*raw*)
Decode signature from RAW encoded bytes sequence.

**Parameters** **raw** (bytes) – encoded signature

**Returns** signature

**Return type** *Signature*

**r**
Signature part #1

**raw**
Return RAW Encode signature as bytes sequence

**s**
Signature part #2

**static schnorr_sign**(*message*, *privateKey*)
Generate message signature according to BIP schnorr scheme.

**Parameters**

- **message** (str) – message to verify
- **privateKey** (str) – private key

**Returns** signature

**Return type** *Signature*

**schnorr_verify**(*message*, *publicKey*)
Check if public key match message signature according to BIP schnorr scheme.

**Parameters**

- **message** (str) – message to verify
- **publicKey** (str) – public key

**Returns** True if match

**Return type** bool

## 4.4 Transaction builders

dposlib.ark.v2.**transfer**(*amount*, *address*, *vendorField=None*, *expiration=0*)
Build a transfer transaction. Emoji can be included in transaction vendorField using unicode formating.

```
>>> u"message with sparkles \u2728"
```

**Parameters**

- **amount** (float) – transaction amount in ark
- **address** (str) – valid recipient address
- **vendorField** (str) – vendor field message
- **expiration** (float) – time of persistance in hour

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**registerSecondSecret**(*secondSecret*)
Build a second secret registration transaction.

**Parameters** **secondSecret** (str) – passphrase

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**registerSecondPublicKey**(*secondPublicKey*)
Build a second secret registration transaction.

> **Note:** You must own the secret issuing secondPublicKey

**Parameters** **secondPublicKey** (str) – public key as hex string

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**registerAsDelegate**(*username*)
Build a delegate registration transaction.

**Parameters** **username** (str) – delegate username

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**upVote**(**usernames*)
Build an upvote transaction.

**Parameters** **usernames** (iterable) – delegate usernames as str iterable

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**downVote**(**usernames*)
Build a downvote transaction.

**Parameters** **usernames** (iterable) – delegate usernames as str iterable

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**registerMultiSignature**(*minSig*, **publicKeys*)
Build a multisignature registration transaction.

**Parameters**

- **minSig** (int) – minimum signature required
- **publicKeys** (list of str) – public key list

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**registerIpfs**(*ipfs*)
Build an IPFS registration transaction.

**Parameters** **ipfs** (str) – ipfs DAG

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**multiPayment**(**pairs*, ***kwargs*)
Build multi-payment transaction. Emoji can be included in transaction vendorField using unicode formating.

```
>>> u"message with sparkles \u2728"
```

**Parameters**

- **pairs** (iterable) – recipient-amount pair iterable
- **vendorField** (str) – vendor field message

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**delegateResignation**()
Build a delegate resignation transaction.

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**htlcSecret**(*secret*)
Compute an HTLC secret hex string from passphrase.

**Parameters** **secret** (str) – passphrase

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**htlcLock**(*amount*, *address*, *secret*, *expiration=24*, *vendorField=None*)
Build an HTLC lock transaction. Emoji can be included in transaction vendorField using unicode formating.

```
>>> u"message with sparkles \u2728"
```

**Parameters**

- **amount** (float) – transaction amount in ark
- **address** (str) – valid recipient address
- **secret** (str) – lock passphrase
- **expiration** (float) – transaction validity in hour
- **vendorField** (str) – vendor field message

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**htlcClaim**(*txid*, *secret*)
Build an HTLC claim transaction.

**Parameters**

- **txid** (str) – htlc lock transaction id
- **secret** (str) – passphrase used by htlc lock transaction

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

dposlib.ark.v2.**htlcRefund**(*txid*)
Build an HTLC refund transaction.

**Parameters** **txid** (str) – htlc lock transaction id

**Returns** transaction object

**Return type** *dposlib.blockchain.Transaction*

CHAPTER 5

# API

## 5.1 The `Wallet`

**class** dposlib.blockchain.**Wallet**(*endpoint*, **args*, ***kwargs*)

# CHAPTER 6

## Multisignature server

mssrv.**dump**(*network*, *tx*)
Add a transaction into registry. `senderPublicKey` field is used to create registry if it does not exist.

**Parameters**

- **network** (`str`) – blockchain name
- **tx** (`dict` or *dposlib.blockchain.Transaction*) – transaction to store

mssrv.**getAll**(*network*)
`GET /multisignature/{network}` endpoint. Return all public keys issuing multisignature transactions.

**Parameters** **network** (`str`) – blockchain network name

**Returns** all registries

**Return type** `dict`

mssrv.**getSerial**(*network*, *ms_publicKey*, *txid*)
`GET /multisignature/{network}/{ms_publicKey}/{txid}/serial` endpoint. Return specific pending transaction serial from a specific public key.

mssrv.**getTransaction**(*network*, *ms_publicKey*, *txid*)
`GET /multisignature/{network}/{ms_publicKey}/{txid}` endpoint. Return specific pending transaction from a specific public key.

mssrv.**getWallet**(*network*, *ms_publicKey*)
`GET /multisignature/{network}/{ms_publicKey}` endpoint. Return all pending transactions issued by a specific public key.

mssrv.**identify**(*tx*)
Identify a transaction.

**Parameters** **tx** (`dict` or *dposlib.blockchain.Transaction*) – transaction to identify

**Returns** transaction id used by registries

**Return type** `str`

mssrv.**load**(*network*, *ms_publicKey*, *txid*)
Load a transaction from a specific registry.

**Parameters**

- **network** (str) – blockchain name
- **ms_publicKey** (str) – encoded-compresed public key as hex string
- **txid** (str) – transaction id

**Returns** transaction data

**Return type** dict

mssrv.**pop**(*network*, *tx*)
Remove a transaction from registry. Wallet registry is removed if empty.

**Parameters**

- **network** (str) – blockchain name
- **publicKey** (str) – encoded-compresed public key as hex string

mssrv.**postNewTransactions**(*network*)
`POST /multisignature/{network}/post` endpoint. Post transaction from multisignature wallet to be remotly signed:

```
data = {"transactions": [tx1, tx2, ... txi ..., txn]}
```

See *putSignature()*.

mssrv.**putSignature**(*network*, *ms_publicKey*)
`PUT /multisignature/{network}/{ms_publicKey}/put` endpoint. Add signature to a pending transaction:

```
data = {
    "info": {
        "id": pending_transaction_id,
        "signature": signature,
        "publicKey": associated_public_key
    } [ + {
        "fee": optional_fee_value_to_use
    } ]
}
```

mssrv.**registerWallet**(*network*)
`POST /multisignature/{network}/create` endpoint. Register as multisignature wallet:

```
data = {
    "info": {
        "senderPublicKey": wallet_public_key_issuing_transaction,
        "min": minimum_signature_required,
        "publicKeys": public_key_list
    }
}
```

Once created on server, registration transaction have to be remotly signed. See *putSignature()*.

CHAPTER 7

# Code snippets

## 7.1 Advanced Crypto

Public key is a point on `secp256k1` curve defined by the multiplication of a scalar with the curve generator point. Scalar used in such a process is called the private key.

```
>>> from dposlib.ark import secp256k1 as curve
>>> curve.G  # curve generator point
[55066263022277343669578718895168534326250603453777594175500187360389116729240,␣
↪32670510020758816978083085130507043184471273380659243275938904335757337482424]
>>> 12 * curve.G
[94111259592240215275188773285036844871058226277992966241101117022315524122714,␣
↪76870767327212528811304566602812752860184934880685532702451763239157141742375]
```

In this example, `12` is the private key. In Ark blockchain, private key is an hexlified 32-bytes-length sequence. Public key is encoded as hex string.

```
>>> from dposlib.util.bin import hexlify
>>> puk = hexlify((12 * curve.G).encode())
>>> puk
'03d01115d548e7561b15c38f004d734633687cf4419620095bc5b0f47070afe85a'
>>> prk = hexlify(secp256k1.bytes_from_int(12))
>>> prk
'000000000000000000000000000000000000000000000000000000000000000c'
```

You can use *dposlib.ark.sig* module to issue and check signatures.

```
>>> from dposlib.ark.sig import Signature
>>> sig1 = Signature.ecdsa_rfc6979_sign("simple message", prk)  # ark-core <= 2.5
>>> hexlify(sig1.der)

↪'3045022100dcdf549f3904eaec24af8aff6fc790429d0ed98e2ec38919db85ffa23e80fb2902201018d303a10c589abfa
↪'
>>> sig2 = Signature.b410_schnorr_sign("simple message", prk)  # ark-core >= 2.6
```

(continues on next page)

(continued from previous page)

```
>>> hexlify(sig2.raw)

↪'5ed1dfd2923f8434bac014f4b0214f8e69730f9b9c7a859d05ec6897fc3e42d7171857d8a2c8bb18fb2358bd02baad8567
↪'
>>> sig1.ecdsa_verify("simple message", puk)
True
>>> sig2.b410_schnorr_verify("simple message", puk)
True
```

## 7.2 Peer targeting / JSON API access

*dposlib.rest* module provides easy way to target a specific peer when sending a http request in blockchain network. You can also access whatever JSON API endpoint.

> **Note:** Public ip of http request emitter have to be white listed on targetted peer.

```
>>> from dposlib import rest
>>> # no need to call rest.use directive...
>>> # https://min-api.cryptocompare.com/data/histoday?fsym=BTC&tsym=ARK&limit=365&
↪toTS=1577833140
>>> data = rest.GET.data.histoday(
...     peer="https://min-api.cryptocompare.com", fsym="BTC", tsym="ARK",
...     limit=365, toTs=1577833140
...)
>>> data["Data"][-1]
{u'volumeto': 242439.09, u'high': 42955.33, u'low': 40832.99, u'time': 1575072000, u
↪'volumefrom': 5.761, u'close': 42789.9, u'open': 40966.82}
>>> # get configuration of https://explorer.ark.io:8443 peer
>>> data = rest.GET.api.node.configuration(peer="https://explorer.ark.io:8443")
>>> data["data"]["transactionPool"]
{u'dynamicFees': {u'minFeePool': 3000, u'minFeeBroadcast': 3000, u'enabled': True, u
↪'addonBytes': {u'ipfs': 250, u'transfer': 100, u'timelockTransfer': 500, u
↪'multiSignature':
500, u'delegateRegistration': 400000, u'delegateResignation': 100, u'multiPayment':␣
↪500, u'vote': 100, u'secondSignature': 250}}}>>> rest.use("ark")
```

## 7.3 Emoji in `vendorField`

This transaction will show a nice sparkle in its `vendorField`:

```
>>> dposlib.core.transfer(1, "DChFFe4QMwZesdMYNEkJsqnqY4MnF4TYQu", vendorField=u
↪"message with sparkles \u2728")
{
  "amount": 100000000,
  "asset": {},
  "recipientId": "DChFFe4QMwZesdMYNEkJsqnqY4MnF4TYQu",
  "senderId": "D7seWn8JLVwX4nHd9hh2Lf7gvZNiRJ7qLk",
  "senderPublicKey":
↪"03a02b9d5fdd1307c2ee4652ba54d492d1fd11a7d1bb3f3a44c4a05e79f19de933",
  "timestamp": 85040681,
```

(continues on next page)

(continued from previous page)

```
  "type": 0,
  "vendorField": "message with sparkles \u2728",
  "version": 1
}
```

Emoji can be embeded in transaction `vendorField` using python unicode string. For example:

- : unicode hex value `2728`, use `\uXXXX` format:

```
>>> u"emoji defined by less than or equal 4 digits : \u2728 - "
```

- : unicode hex value `1f4b1`, use `\UXXXXXXXX` format:

```
>>> u"emoji defined by more than 4 digits : \U0001f4b1"
```

## 7.4 Multisignature server

`dpos` repository contains `mssrv` package that provides client - server modules to issue multisignature registration and transactions.

let's take an exemple of a two owner multisignature wallet. From owner terminal issuing the transaction:

```
>>> import dposlib
>>> from dposlib import rest
>>> from mssrv import client
>>> rest.use("dark")
True
>>> client.API_PEER = "http://mssrv.arky-delegate.info"
>>> t = dposlib.core.transfer(1, "D7seWn8JLVwX4nHd9hh2Lf7gvZNiRJ7qLk", u"ms-srv test
↪#4 \u2728", version=2)
>>> t.senderPublicKey =
↪"02cccf1a186bed2cf8d22f6c46d8497a4eceeb8e159bde4ee83b908145764da5e3"
>>> t.setFee()
>>> # one signature minimum is mandatory
>>> t.multiSignWithSecret("secret")
>>> client.postNewTransactions("dark", t)
{u'success': [u'transaction #1 successfully posted'], u'ids': [u
↪'7c01e5bd9d78a82f766db50c345cbcd227e47089b3fbeca7cde530a46bfcb77e']}
```

From second owner terminal:

```
>>> from mssrv import client
>>> client.API_PEER = "http://mssrv.arky-delegate.info"
>>> senderPublicKey =
↪"02cccf1a186bed2cf8d22f6c46d8497a4eceeb8e159bde4ee83b908145764da5e3"
>>> tx_id = "7c01e5bd9d78a82f766db50c345cbcd227e47089b3fbeca7cde530a46bfcb77e"
>>> # automated broadcast when minimum signature reached
>>> client.remoteSignWithSecret("dark", senderPublicKey, tx_id)
secret >
{u'broadcast': [u'47b7d0431a2996c04292ae9bddad36db52e3babcc666704d593da616ab6c207e'],␣
↪u'accept': [u'47b7d0431a2996c04292ae9bddad36db52e3babcc666704d593da616ab6c207e'], u
↪'invalid': [], u'excess': []}
```

# Python Module Index

## d



## m

# Index

## V



## W



## X



## Y