neo3-boa Documentation

Release 1.1.1

CoZ

Feb 01, 2024

CONTENTS:

CHAPTER

ONE

GETTING STARTED

Check out our GitHub README page to see how you can install Neo3-Boa.

1.1 Writing a smart contract

It's pretty easy to write a Python3 script with Neo3-Boa, since it is compatible with a lot of Python features. However, there are some key differences that you should be aware of, here's the 4 most prominent ones:

- there is no floating point arithmetic, only the `int` type is implemented;
- you need to specify a function's return type and parameter types;
- if you want to call other smart contracts, you can only call public functions;
- to interact with the Neo blockchain, you need to use a function, variable, or class inside the `boa3.builtin` package.

1.1.1 Overview of Neo3-Boa features

Packages	Contains:	Important features
boa3.builtin	all packages below, it also contains an env variable that lets you change the value of a variable when compiling the smart contract.	*env*
boa3.builtin.compile_time	methods and classes that are needed when you are compiling your smart contract, as opposed to when it's being executed on the blockchain.	*public*, *metadata*, *contract*, *CreateNewEvent*, *NeoMetadata*
boa3.builtin.contract	events and methods that might help when writing something specific about Neo blockchain	*abort*, *Nep17TransferEvent*, *Nep11TransferEvent*
boa3.builtin.interop	other packages that have a lot of helpful interoperable services. Has some overlap with the native contracts.	*storage*, *runtime*, *contract*, *blockchain*
boa3.builtin.interop.blockchain	features related to get information on the Neo blockchain.	*current_hash*, *get_contract*, *Transaction*
boa3.builtin.interop.contract	features related to smart contracts.	*call_contract*, *update_contract*, *Contract*
boa3.builtin.interop.crypto	features related to cryptography.	*sha256*, *hash160*, *hash256*, *check_sig*
boa3.builtin.interop.iterator	the iterator class.	*Iterator*
boa3.builtin.interop.json	methods to serialize and deserialize JSON.	*json_serialize*, *json_deserialize*
boa3.builtin.interop.oracle	features related with Neo Oracle, it is used to get information from outside the blockchain.	*Oracle*
boa3.builtin.interop.policy	features related to policies that affect the entire Neo blockchain.	*get_exec_fee_factor*, *get_storage_price*
boa3.builtin.interop.role	methods to get information about the nodes on the blockchain.	*get_designated_by_role*
boa3.builtin.interop.runtime	features to get information that can only be acquired when running the smart contract.	*check_witness*, *calling_script_hash*, *executing_script_hash*, *script_container*, *Notification*
boa3.builtin.interop.stdlib	methods that convert one data to another or methods that can check and compare memory.	*serialize*, *deserialize*
boa3.builtin.interop.storage	features to store, get, or change values inside the blockchain.	*get*, *put*, *find*, *FindOptions*
boa3.builtin.nativecontract	classes that interface Neo's native contracts.	ContractManagement, *GAS*, *NEO*, *StdLib*
boa3.builtin.type	Neo types.	*UInt160*, *UInt256*, *Event*, *ECPoint*
boa3.builtin.vm	Opcodes used internally by the Neo VM, used to create scripts dynamically.	*Opcode*
boa3.builtin.math	a small sample of functions similar to Python's math.	*sqrt*, *floor*, *ceil*

1.1.2 Hello World

Let's write a quick Hello World script that has a method that will save `"Hello World"` on the blockchain and another method that will return the string.

Those 2 functions will need to be callable and will also need to change the values inside the storage, so let's import both the `public` decorator and the `storage` package.

```
# hello_world.py
from boa3.builtin.compile_time import public
```

(continues on next page)

(continued from previous page)

```
from boa3.builtin.interop import storage


@public     # the public decorator will make this method callable
def save_hello_world():             # an empty return type indicates that the return is␣
↪None
    storage.put(b"first script", "Hello World")      # the put method will store the␣
↪"Hello World" value with the "first script" key


@public     # the public decorator will make this method callable too
def get_hello_world() -> str:       # this method will return a string, so it needs to␣
↪specify it
    return str(storage.get(b"first script"))              # the get method will return␣
↪the value associated with "first script" key
```

1.1.3 Neo Methods

Neo currently has 2 special methods: `_deploy` and `verify`:

```
from typing import Any
from boa3.builtin.compile_time import public

@public
def _deploy(data: Any, update: bool):
    """
    This method will automatically be called when the smart contract is deployed or␣
↪updated.
    """
    pass

@public
def verify() -> bool:
    """
    When this contract address is included in the transaction signature,
    this method will be triggered as a VerificationTrigger to verify that the signature␣
↪is correct.
    For example, this method needs to be called when withdrawing token from the contract.

    :return: whether the transaction signature is correct
    """
    pass
```

So, using the example above, if you want to set the `"Hello World"` message when you deploy your smart contract and have another method to save a given string you could do the following:

```
# hello_world_with_deploy.py
from typing import Any

from boa3.builtin.compile_time import public
from boa3.builtin.interop import storage
```

(continues on next page)

(continued from previous page)

```
@public
def _deploy(data: Any, update: bool):                # the _deploy function needs to have␣
↪this signature
    storage.put(b"second script", "Hello World")      # "Hello World" will be stored␣
↪when this smart contract is deployed


@public
def get_message() -> str:                           # this method will still try to get the␣
↪value saved on the blockchain
    return str(storage.get(b"second script"))


@public
def set_message(new_message: str):                  # now this method will overwrite a new␣
↪string on the blockchain
    storage.put(b"second script", new_message)
```

1.2 Compiling your Smart Contract

1.2.1 Using CLI

```
$ neo3-boa compile path/to/your/file.py
```

> Note: When resolving compilation errors it is recommended to resolve the first reported error and try to compile again. An error can have a cascading effect and throw more errors all caused by the first.

1.2.2 Using Python Script

```
from boa3.boa3 import Boa3

Boa3.compile_and_save('path/to/your/file.py')
```

1.3 Reference Examples and Tutorials

Check out Neo3-boa tutorials on Neo Developer.

For an extensive collection of examples:

- Smart contract examples
- Features tests

1.4 What's next

- *Testing and debugging your smart contract*
- *How to call other smart contracts on the blockchain*
- *Invoking smart contracts with NeoNova*

CHAPTER

TWO

TESTING AND DEBUGGING

2.1 Configuring the Debugger

Neo3-Boa is compatible with the Neo Debugger. Debugger launch configuration example:

```
{
    //Launch configuration example for Neo3-Boa.
    //Make sure you compile your smart-contract before you try to debug it.
    "version": "0.2.0",
    "configurations": [
        {
            "name": "example.nef",
            "type": "neo-contract",
            "request": "launch",
            "program": "${workspaceFolder}\\example.nef",
            "operation": "main",
            "args": [],
            "storage": [],
            "runtime": {
                "witnesses": {
                    "check-result": true
                }
            }
        }
    ]
}
```

It's necessary to generate the nef debugger info file to use Neo Debugger.

2.1.1 Using CLI

```
$ neo3-boa compile path/to/your/file.py -d|--debug
```

2.1.2 Using Python Script

```
from boa3.boa3 import Boa3

Boa3.compile_and_save('path/to/your/file.py', debug=True)
```

2.2 Neo Test Runner

2.2.1 Downloading

Install Neo-Express and Neo Test Runner.

2.2.2 Testing

Before writing your tests, make sure you have a Neo-Express network for local tests. If you do not yet have a local network, open a terminal and run `neoxp create`. Please refer to Neo-Express documentation for more details of how to configure your local network.

Create a Python Script, import the NeoTestRunner class, and define a function to test your smart contract. In this function you'll need a NeoTestRunner object, which takes the path of your Neo-Express network configuration file as an argument to set up the test environment.

You'll have to call the method `call_contract()` to interact with your smart contract. Its parameters are the path of the compiled smart contract, the smart contract's method, and the arguments if necessary. This call doesn't return the result directly, but includes it in a queue of invocations. To execute all the invocations set up, call the method `execute()`. Then assert the result of your invoke to see if it's correct.

Note that `invoke.result` won't be set if the execution fails, so you should also assert if `runner.vm_state` is valid for your test case.

Your Python Script should look something like this:

```
from boa3.builtin.interop.blockchain.vmstate import VMState
from boa3_test.test_drive.testrunner.neo_test_runner import NeoTestRunner


def test_hello_world_main():
    neoxp_config_file = '{path-to-neo-express-config-file}'
    project_root_folder = '{path-to-project-root-folder}'
    path = f'{project_root_folder}/boa3_test/examples/hello_world.nef'
    runner = NeoTestRunner(neoxp_config_file)

    invoke = runner.call_contract(path, 'main')
    runner.execute()
    assert runner.vm_state is VMState.HALT
    assert invoke.result is None
```

Alternatively you can change the value of `env.NEO_EXPRESS_INSTANCE_DIRECTORY` to the path of your .neo-express data file:

```
from boa3.builtin.interop.blockchain.vmstate import VMState
from boa3_test.test_drive.testrunner.neo_test_runner import NeoTestRunner
from boa3.internal import env

env.NEO_EXPRESS_INSTANCE_DIRECTORY = '{path-to-neo-express-config-file}'


def test_hello_world_main():
    root_folder = '{path-to-project-root-folder}'
    path = f'{root_folder}/boa3_test/examples/hello_world.nef'
    runner = NeoTestRunner()  # the default path to the Neo-Express is the one on env.
↪NEO_EXPRESS_INSTANCE_DIRECTORY

    invoke = runner.call_contract(path, 'main')
    runner.execute()
    assert runner.vm_state is VMState.HALT
    assert invoke.result is None
```

CHAPTER

THREE

CALLING SMART CONTRACTS

Currently, there are 2 different ways to call another smart contract on the blockchain: using a `call_contract` method or using an interface, but internally they work both the same way.

Let us suppose the following contract was deployed on Neo:

```
from boa3.builtin.compile_time import public


@public
def hello_stranger(name: str) -> str:
    return "Hello " + name
```

To call the `hello_stranger` method, first you'll need to know the smart contract's script hash, Let us also assume it is `0x0001020304050607080090A0B0C0D0E0F10111213`.

3.1 with call_contract

Use the `call_contract` method from `boa3.builtin.interop.contract` on your smart contract. You'll need to use the script hash, followed by the name of the function you want to call, followed by the arguments of said function. You'll also need to type cast the return so the compiler type checker works as expected, otherwise the return type will be considered as `Any`.

```
# calling_with_call_contract.py
from boa3.builtin.compile_time import public
from boa3.builtin.interop.contract import call_contract
from boa3.builtin.type import UInt160

@public
def calling_other_contract() -> str:
    greetings: str = call_contract(UInt160(b'\x13\x12\x11\x10\x0F\x0E\x0D\x0C\x0B\x0A\
↪x09\x08\x07\x06\x05\x04\x03\x02\x01\x00'),     # usually, script hashes that starts␣
↪with "0x" means that they are using big endian, so when using `bytes` you'll need to␣
↪revert the order
                                'hello_stranger',     # it's the function's name
                                'John Doe'            # the parameter of 'hello_
↪stranger'
                               )
    return greetings
```

Note: If you are going to call a contract only once, then it's ok to use `call_contract`, however, it might be hard to keep track of a lot of `call_contracts` on the same file. It's pretty much always better to use an interface when dealing with other smart contracts.

3.2 with Interface

Use the `contract` decorator from `boa3.builtin.compile_time` using the script hash and create a class that have the same methods you want call.

```
# calling_with_interface.py
from boa3.builtin.compile_time import contract, public
from boa3.builtin.type import UInt160

@public
def calling_other_contract() -> str:
    greetings = HelloStrangerContract.hello_stranger('John Doe')
    return greetings


@contract('0x0001020304050607080090A0B0C0D0E0F10111213')
class HelloStrangerContract:
    hash: UInt160   # this class variable will reflect the value you passed to the␣
↪`contract` decorator

    @staticmethod
    def hello_stranger(name: str) -> str:
        pass
```

3.2.1 Calling native contracts

Neo3-Boa already has interfaces for all the native contracts that you can import from `boa3.builtin.nativecontract`

```
# calling_native_contract.py
from boa3.builtin.compile_time import public
from boa3.builtin.nativecontract.neo import NEO

@public
def calling_other_contract() -> str:
    neo_symbol = NEO.symbol()
    return neo_symbol
```

3.2.2 Automate with CPM

Instead of manually writing the smart contract interface, you can use CPM to generate it automatically. After installing Neo3-Boa, you can install CPM by typing `install_cpm` on CLI (without the `neo3-boa` prefix). Then, you'll need to create a cpm.yaml config file, put the smart contract information there, and run cpm.

For example, if you use CPM to create a dice smart contract interface, the following file will be generated:

```
# cpm_out/python/dice/contract.py
from boa3.builtin.type import UInt160, UInt256, ECPoint
from boa3.builtin.compile_time import contract, display_name
from typing import cast, Any


@contract('0x4380f2c1de98bb267d3ea821897ec571a04fe3e0')
class Dice:
    hash: UInt160

    @staticmethod
    def rand_between(start: int, end: int) -> int:
        pass

    @staticmethod
    def map_bytes_onto_range(start: int, end: int, entropy: bytes) -> int:
        pass

    @staticmethod
    def roll_die(die: str) -> int:
        pass

    @staticmethod
    def roll_dice_with_entropy(die: str, precision: int, entropy: bytes) -> list:
        pass

    @staticmethod
    def update(script: bytes, manifest: bytes, data: Any) -> None:
        pass
```

Then, all you need to do is import this class onto your smart contract.

```
# calling_with_cpm.py
from boa3.builtin.compile_time import public
from cpm_out.python.dice.contract import Dice


@public
def calling_other_contract() -> str:
    d6_roll = Dice.rand_between(1, 6)
    return "Dice result is " + str(d6_roll)
```

CHAPTER

FOUR

INVOKING SMART CONTRACTS WITH NEONOVA

After writing, compiling and deploying your smart contract locally, you can invoke it using Neonova.

First, connect to your local network by clicking on the top right button next to "Select Wallet", then "Custom" and input your local address with the rpc-port and magic number (they should be inside the .neo-express file), and click on "Save".

If you want to change or add data to the blockchain, you'll also need to connect a wallet to sign your transactions, click on the "Select Wallet" button, choose one of the platforms and then click on "Connect" (you need to have the app you chose on your device).

Then, select the "Read" type on the menu if you want to check a value or test an invocation on the blockchain or select "Write" to persist your invocation on the blockchain, input the script hash of the smart contract you want to invoke, the name of the method you want to call and the arguments if the method has any, and click on "Send" to send the transaction.

If you did a "Read" invocation, then the result should appear on the right side.

If you did a "Write" invocation, you'll also have the sign the transaction and have enough GAS to pay the fees. On the right side the transaction id should appear.

CHAPTER

FIVE

PACKAGE REFERENCE

5.1 boa3 package

5.1.1 boa3.builtin package

env: *str*

Gets the compiled environment. This allows specific environment validations to be easily included in the smart contract logic without the need to rewrite anything before compiling (i.e. changes in smart contracts hashes between testnet and mainnet).

```
>>> # compiling with 'neo3-boa compile -e test_net ./path/to/contract.py'
... from boa3.builtin.interop.contract import call_contract
... from boa3.builtin.type import UInt160
... call_contract(UInt160(b'12345678901234567890') if env == 'test_net' else b
↪'abcdeabcdeabcdeabcde',
...               'balanceOf',
...               UInt160(b'zyxwvzyxwvzyxwvzyxwv'))
110000
```

```
>>> # compiling with 'neo3-boa compile -e main_net ./path/to/contract.py'
... from boa3.builtin.interop.contract import call_contract
... from boa3.builtin.type import UInt160
... call_contract(UInt160(b'12345678901234567890') if env == 'test_net' else b
↪'abcdeabcdeabcdeabcde',
...               'balanceOf',
...               UInt160(b'zyxwvzyxwvzyxwvzyxwv'))
250
```

compile_time

CreateNewEvent(*arguments: List[Tuple[str, type]] = []*, *event_name: str = ''*) → *Event*

Creates a new Event.

Check out Neo's Documentation to learn more about Events.

```
>>> new_event: Event = CreateNewEvent(
...     [
...         ('name', str),
...         ('amount', int)
...     ],
...     'New Event'
... )
```

Parameters

- **arguments** (*List[Tuple[str, type]]*) – the list of the events args' names and types
- **event_name** (*str*) – custom name of the event. It's filled with the variable name if not specified

Returns

the new event

Return type

Event

public(*name: Optional[str] = None*, *safe: bool = True*, **args*, ***kwargs*)

This decorator identifies which methods should be included in the abi file. Adding this decorator to a function means it could be called externally.

```
>>> @public     # this method will be added to the abi
... def callable_function() -> bool:
...     return True
{
    "name": "callable_function",
    "offset": 0,
    "parameters": [],
    "safe": false,
    "returntype": "Boolean"
}
```

```
>>> @public(name='callableFunction')     # the method will be added with the␣
↪different name to the abi
... def callable_function() -> bool:
...     return True
{
    "name": "callableFunction",
    "offset": 0,
    "parameters": [],
    "safe": false,
    "returntype": "Boolean"
}
```

```
>>> @public(safe=True)      # the method will be added with the safe flag to the abi
... def callable_function() -> bool:
...     return True
{
    "name": "callable_function",
    "offset": 0,
    "parameters": [],
    "safe": true,
    "returntype": "Boolean"
}
```

Parameters

- **name** (*str*) – Identifier for this method that'll be used on the abi. If not specified, it'll be the same identifier from Python method definition
- **safe** (*bool*) – Whether this method is an abi safe method. False by default

metadata(**args*)

This decorator identifies the function that returns the metadata object of the smart contract. This can be used to only one function. Using this decorator in multiple functions will raise a compiler error.

Deprecated in 1.1.1. Will be removed in 1.2.0

```
>>> @metadata   # this indicates that this function will have information about the␣
↪smart contract
... def neo_metadata() -> NeoMetadata:      # needs to return a NeoMetadata
...     meta = NeoMetadata()
...     meta.name = 'NewContractName'
...     return meta
```

contract(*script_hash: Union[str, bytes]*)

This decorator identifies a class that should be interpreted as an interface to an existing contract.

If you want to use the script hash in your code, you can use the *hash* class attribute that automatically maps the script hash parameter onto it. You don't need to declare it in your class, but your IDE might send a warning about the attribute if you don't.

Check out Our Documentation to learn more about this decorator.

```
>>> @contract('0xd2a4cff31913016155e38e474a2c06d08be276cf')
... class GASInterface:
...     hash: UInt160      # you don't need to declare this class variable, we are␣
↪only doing it to avoid IDE warnings
...                        # but if you do declare, you need to import the type␣
↪UInt160 from boa3.builtin.type
...     @staticmethod
...     def symbol() -> str:
...         pass
... @public
... def main() -> str:
...     return "Symbol is " + GASInterface.symbol()
... @public
... def return_hash() -> UInt160:
...     return GASInterface.hash    # neo3-boa will understand that this attribute␣
↪exists even if you don't declare it
```

(continues on next page)

(continued from previous page)

Parameters
script_hash (*str or bytes*) – Script hash of the interfaced contract

display_name(*name: str*)

This decorator identifies which methods from a contract interface should have a different identifier from the one interfacing it. It only works in contract interface classes.

```
>>> @contract('0xd2a4cff31913016155e38e474a2c06d08be276cf')
... class GASInterface:
...     @staticmethod
...     @display_name('totalSupply')
...     def total_supply() -> int:      # the smart contract will call "totalSupply
↪", but when writing the script you can call this method whatever you want to
...         pass
... @public
... def main() -> int:
...     return GASInterface.total_supply()
```

Parameters
name (*str*) – Method identifier from the contract manifest.

class NeoMetadata

Bases: object

This class stores the smart contract manifest information.

Check out Neo's Documentation to learn more about the Manifest.

```
>>> def neo_metadata() -> NeoMetadata:
...     meta = NeoMetadata()
...     meta.name = 'NewContractName'
...     meta.add_permission(methods=['onNEP17Payment'])
...     meta.add_trusted_source("0x1234567890123456789012345678901234567890")
...     meta.date = "2023/05/30"    # this property will end up inside the extra␣
↪property
...     return meta
```

Variables

- **name** – the smart contract name. Will be the name of the file by default;
- **supported_standards** (*List[str]*) – Neo standards supported by this smart contract. Empty by default;
- **permissions** (*List[str]*) – a list of contracts and methods that this smart contract permits to invoke and call. All contracts and methods permitted by default;
- **trusts** (*List[str]*) – a list of contracts that this smart contract trust. Empty by default;
- **author** (*str or None*) – the smart contract author. None by default;
- **email** (*str or None*) – the smart contract author email. None by default;
- **description** (*str or None*) – the smart contract description. None by default;

property extras: Dict[str, Any]

Gets the metadata extra information.

Returns

a dictionary that maps each extra value with its name. Empty by default

add_trusted_source(*hash_or_address: str*)

Adds a valid contract hash, valid group public key, or the '*' wildcard to trusts.

```
>>> self.add_trusted_source("0x1234567890123456789012345678901234abcdef")
```

```
>>> self.add_trusted_source(
↪"035a928f201639204e06b4368b1a93365462a8ebbff0b8818151b74faab3a2b61a")
```

```
>>> self.add_trusted_source("*")
```

Parameters

hash_or_address (*str*) – a contract hash, group public key or '*'

add_group(*pub_key: str*, *signature: str*)

Adds a pair of public key and signature to the groups in the manifest.

```
>>> self.add_group(
↪"031f64da8a38e6c1e5423a72ddd6d4fc4a777abe537e5cb5aa0425685cda8e063b",
...
↪"fhsOJNF3N5Pm3oV1b7wYTx0QVelYNu7whwXMi8GsNGFKUnu3ZG8z7oWLfzzEz9pbnzwQe8WFCALEiZhLD1jG/
↪w==")
```

Parameters

- **pub_key** (*str*) – public key of the group
- **signature** (*str*) – signature of the contract hash encoded in Base64

add_permission(*, *contract: str = '*'*, *methods: Union[List[str], str] = '*'*)

Adds a valid contract and a valid methods to the permissions in the manifest.

```
>>> self.add_permission(methods=['onNEP17Payment'])
```

```
>>> self.add_permission(contract='0x3846a4aa420d9831044396dd3a56011514cd10e3',␣
↪methods=['get_object'])
```

```
>>> self.add_permission(contract=
↪'0333b24ee50a488caa5deec7e021ff515f57b7993b93b45d7df901e23ee3004916')
```

Parameters

- **contract** (*str*) – a contract hash, group public key or '*'
- **methods** (*Union[List[str], str]*) – a list of methods or '*'

contract

Nep11TransferEvent: *Event*

The NEP-11 Transfer event that should be triggered whenever a non-fungible token is transferred, minted or burned. It needs the addresses of the sender, receiver, amount transferred and the id of the token.

Check out the proposal or Neo's Documentation about this NEP.

```
>>> Nep11TransferEvent(b'\xd1\x17\x92\x82\x12\xc6\xbe\xfa\x05\xa0\x23\x07\xa1\x12\
↪x55\x41\x06\x55\x10\xe6',  # when calling, it will return None, but the event␣
↪will be triggered
...                     b'\x18\xb7\x30\x14\xdf\xcb\xee\x01\x30\x00\x13\x9b\x8d\xa0\
↪x13\xfb\x96\xac\xd1\xc0', 1, '01')
{
    'name': 'Transfer',
    'script hash': b'\x13\xb4\x51\xa2\x1c\x10\x12\xd6\x13\x12\x19\x0c\x15\x61\x9b\
↪x1b\xd1\xa2\xf4\xb2',
    'state': {
        'from': b'\xd1\x17\x92\x82\x12\xc6\xbe\xfa\x05\xa0\x23\x07\xa1\x12\x55\x41\
↪x06\x55\x10\xe6',
        'to': b'\x18\xb7\x30\x14\xdf\xcb\xee\x01\x30\x00\x13\x9b\x8d\xa0\x13\xfb\
↪x96\xac\xd1\xc0',
        'amount': 1,
        'tokenId': '01'
    }
}
```

Nep17TransferEvent: *Event*

The NEP-17 Transfer event that should be triggered whenever a fungible token is transferred, minted or burned. It needs the addresses of the sender, receiver and the amount transferred.

Check out the proposal or Neo's Documentation about this NEP.

```
>>> Nep17TransferEvent(b'\xd1\x17\x92\x82\x12\xc6\xbe\xfa\x05\xa0\x23\x07\xa1\x12\
↪x55\x41\x06\x55\x10\xe6',  # when calling, it will return None, but the event␣
↪will be triggered
...                     b'\x18\xb7\x30\x14\xdf\xcb\xee\x01\x30\x00\x13\x9b\x8d\xa0\
↪x13\xfb\x96\xac\xd1\xc0', 100)
{
    'name': 'Transfer',
    'script hash': b'\x17\xe3\xca\x91\xca\xb7\xaf\xdd\xe6\xba\x07\xaa\xba\xa1\x66\
↪xab\xcf\x00\x04\x50',
    'state': {
        'from': b'\xd1\x17\x92\x82\x12\xc6\xbe\xfa\x05\xa0\x23\x07\xa1\x12\x55\x41\
↪x06\x55\x10\xe6',
        'to': b'\x18\xb7\x30\x14\xdf\xcb\xee\x01\x30\x00\x13\x9b\x8d\xa0\x13\xfb\
↪x96\xac\xd1\xc0',
        'amount': 100
    }
}
```

class Nep17Contract

Bases: *Contract*

symbol() → str

decimals() → int

total_supply() → int

balance_of(*account:* UInt160) → int

transfer(*from_address:* UInt160, *to_address:* UInt160, *amount:* *int*, *data:* *Optional[Any]* = *None*) → bool

class NeoAccountState

Bases: object

Represents the account state of NEO token in the NEO system.

Variables

- **balance** (*int*) – the current account balance, which equals to the votes cast
- **height** (*int*) – the height of the block where the balance changed last time
- **vote_to** (ECPoint) – the voting target of the account

abort(*msg:* *Optional[str]* = *None*)

Aborts the execution of a smart contract. Using this will cancel the changes made on the blockchain by the transaction.

```
>>> abort()     # abort doesn't return anything by itself, but the execution will␣
↪stop and the VMState will be FAULT
VMState.FAULT
```

```
>>> abort('abort message')
VMState.FAULT
```

to_hex_str(*data:* *bytes*) → str

Converts bytes into its string hex representation.

```
>>> to_hex_str(ECPoint(bytes(range(33))))
'201f1e1d1c1b1a19181716151413121110 0f0e0d0c0b0a09080706050403020100'
```

```
>>> to_hex_str(b'1234567891')
'31393837363534333231'
```

Parameters

data (*bytearray or bytes*) – data to represent as hex.

Returns

the hex representation of the data

Return type

str

to_script_hash(*data_bytes: Any*) → bytes

Converts a data to a script hash.

```
>>> to_script_hash(ECPoint(bytes(range(33))))
b'\x12\xc8z\xfb3k\x1e4>\xb3\x83\tK\xc7\xdch\xe5\xee\xc7\x98'
```

```
>>> to_script_hash(b'1234567891')
b'\x4b\x56\x34\x17\xed\x99\x7f\x13\x22\x67\x40\x79\x36\x8b\xa2\xcd\x72\x41\x25\x6d'
```

Parameters
data_bytes (*Any*) – data to hash

Returns
the script hash of the data

Return type
bytes

interop

Subpackages

blockchain

get_contract(*hash:* UInt160) → Optional[*Contract*]

Gets a contract with a given hash. If the script hash is not associated with a smart contract, then it will return None.

```
>>> get_contract(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\x01\x13\x19\xf3\xcf\
↪xa4\xd2'))    # GAS script hash
{
    'id': -6,
    'update_counter': 0,
    'hash': b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\x01\x13\x19\xf3\xcf\xa4\xd2',
    'nef': b'NEF3neo-core-v3.0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00#\x10A\x1a\xf7{g@\x10A\x1a\xf7{g@\x10A\x1a\xf7{g@\x10A\x1a\xf7{g@\x10A\
↪x1a\xf7{g@QA\xc7\x9e',
    'manifest': {
        'name': 'GasToken',
        'group': [],
        'supported_standards': ['NEP-17'],
        'abi': [[['balanceOf', [['account', 20]], 17, 0, True], ['decimals', [], 17,
↪ 7, True], ['symbol', [], 19, 14, True], ['totalSupply', [], 17, 21, True], [
↪'transfer', [['from', 20], ['to', 20], ['amount', 17], ['data', 0]], 16, 28,␣
↪False]], [['Transfer', [['from', 20], ['to', 20], ['amount', 17]]]]],
        'permissions': [[None, None]],
        'trusts': [],
        'extras': 'null'
    },
}
```

```
>>> get_contract(UInt160(bytes(20)))    # there is no smart contract associated␣
↪with this script hash
None
```

Parameters
hash (UInt160) – a smart contract hash

Returns
a contract

Return type
Contract

Raises
Exception – raised if hash length isn't 20 bytes.

get_block(*index_or_hash: Union[int,* UInt256*]*) → Optional[*Block*]

Gets the block with the given index or hash. Will return None if the index or hash is not associated with a Block.

```
>>> get_block(0)        # first block
{
    'hash': b"S{\xed'\x85&\xf5\x93U=\xc1\xbf'\x95\xc4/\x80X\xdb\xd5\xa1-\x97q\x85\
↪xe3I\xe5\x99cd\x04",
    'version': 0,
    'previous_hash': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    'merkle_root': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    'timestamp': 1468595301000,
    'nonce': 2083236893,
    'index': 0,
    'primary_index': 0,
    'next_consensus': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\
↪x00\x12\x04',
    'transaction_count': 0,
}
```

```
>>> get_block(UInt256(b"S{\xed'\x85&\xf5\x93U=\xc1\xbf'\x95\xc4/\x80X\xdb\xd5\xa1-\
↪x97q\x85\xe3I\xe5\x99cd\x04"))        # first block
{
    'hash': b"S{\xed'\x85&\xf5\x93U=\xc1\xbf'\x95\xc4/\x80X\xdb\xd5\xa1-\x97q\x85\
↪xe3I\xe5\x99cd\x04",
    'version': 0,
    'previous_hash': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    'merkle_root': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    'timestamp': 1468595301000,
    'nonce': 2083236893,
    'index': 0,
    'primary_index': 0,
    'next_consensus': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\
↪x00\x12\x04',
```

(continues on next page)

(continued from previous page)

```
    'transaction_count': 0,
}
```

```
>>> get_block(9999999)      # block doesn't exist
None
```

```
>>> get_block(UInt256(bytes(32)))   # block doesn't exist
None
```

Parameters
index_or_hash (*int or* UInt256) – index or hash identifier of the block

Returns
the desired block, if exists. None otherwise

Return type
Block or None

get_transaction(*hash_:* UInt256) → Optional[*Transaction*]

Gets a transaction with the given hash. Will return None if the hash is not associated with a Transaction.

```
>>> get_transaction(UInt256(b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\xdf\xc8\xb0\
↪x9f>\xd0\xd2G\xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr'))
{
    'hash': b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\xdf\xc8\xb0\x9f>\xd0\xd2G\
↪xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr',
    'version': 0,
    'nonce': 2025056010,
    'sender': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\x00\x12\
↪x04',
    'system_fee': 2028330,
    'network_fee': 1206580,
    'valid_until_block': 5761,
    'script': b'\x0c\x14\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\
↪x00\x12\x04\x11\xc0\x1f\x0c\tbalanceOf\x0c\x14\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\
↪x01\x13\x19\xf3\xcf\xa4\xd2Ab}[R',
}
```

```
>>> get_transaction(UInt256(bytes(32)))   # transaction doesn't exist
None
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
the Transaction, if exists. None otherwise

get_transaction_from_block(*block_hash_or_height: Union[*UInt256*, int]*, *tx_index: int*) → Optional[*Transaction*]

Gets a transaction from a block. Will return None if the block hash or height is not associated with a Transaction.

```
>>> get_transaction_from_block(1, 0)
{
    'hash': b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\xdf\xc8\xb0\x9f>\xd0\xd2G\
↪xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr',
    'version': 0,
    'nonce': 2025056010,
    'sender': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\x00\x12\
↪x04',
    'system_fee': 2028330,
    'network_fee': 1206580,
    'valid_until_block': 5761,
    'script': b'\x0c\x14\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\
↪x00\x12\x04\x11\xc0\x1f\x0c\tbalanceOf\x0c\x14\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\
↪x01\x13\x19\xf3\xcf\xa4\xd2Ab}[R',
}
```

```
>>> get_transaction_from_block(UInt256(b'\x29\x41\x06\xdb\x4c\xf3\x84\xa7\x20\x4d\
↪xba\x0a\x04\x03\x72\xb3\x27\x76\xf2\x6e\xd3\x87\x49\x88\xd0\x3e\xff\x5d\xa9\x93\
↪x8c\xa3'), 0)
{
    'hash': b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\xdf\xc8\xb0\x9f>\xd0\xd2G\
↪xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr',
    'version': 0,
    'nonce': 2025056010,
    'sender': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\x00\x12\
↪x04',
    'system_fee': 2028330,
    'network_fee': 1206580,
    'valid_until_block': 5761,
    'script': b'\x0c\x14\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\
↪x00\x12\x04\x11\xc0\x1f\x0c\tbalanceOf\x0c\x14\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\
↪x01\x13\x19\xf3\xcf\xa4\xd2Ab}[R',
}
```

```
>>> get_transaction_from_block(123456789, 0)     # height does not exist yet
None
```

```
>>> get_transaction_from_block(UInt256(bytes(32)), 0)     # block hash does not␣
↪exist
None
```

Parameters

- **block_hash_or_height** (UInt256 *or* *int*) – a block identifier
- **tx_index** (*int*) – the transaction identifier in the block

Returns

the Transaction, if exists. None otherwise

get_transaction_height(*hash_:* UInt256) → int

Gets the height of a transaction. Will return -1 if the hash is not associated with a Transaction.

```
>>> get_transaction_height(UInt256(b'\x28\x89\x4f\xb6\x10\x62\x9d\xea\x4c\xcd\x00\
↪x2e\x9e\x11\xa6\xd0\x3d\x28\x90\xc0\xe5\xd4\xfc\x8f\xc6\x4f\xcc\x32\x53\xb5\x48\
↪x01'))
2108703
```

```
>>> get_transaction_height(UInt256(b'\x29\x41\x06\xdb\x4c\xf3\x84\xa7\x20\x4d\xba\
↪x0a\x04\x03\x72\xb3\x27\x76\xf2\x6e\xd3\x87\x49\x88\xd0\x3e\xff\x5d\xa9\x93\x8c\
↪xa3'))
10
```

```
>>> get_transaction_height(UInt256(bytes(32)))   # transaction doesn't exist
-1
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
height of the transaction

get_transaction_signers(*hash_:* UInt256) → List[*Signer*]

Gets a list with the signers of a transaction.

```
>>> get_transaction_signers(UInt256(b'\x29\x41\x06\xdb\x4c\xf3\x84\xa7\x20\x4d\xba\
↪x0a\x04\x03\x72\xb3\x27\x76\xf2\x6e\xd3\x87\x49\x88\xd0\x3e\xff\x5d\xa9\x93\x8c\
↪xa3'))
[
    {
        "account": b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\
↪x00\x12\x04',
        "scopes": 1,
        "allowed_contracts": [],
        "allowed_groups": [],
        "rules": [],
    },
]
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
VM state of the transaction

get_transaction_vm_state(*hash_:* UInt256) → *VMState*

Gets the VM state of a transaction.

```
>>> get_transaction_vm_state(UInt256(b'\x29\x41\x06\xdb\x4c\xf3\x84\xa7\x20\x4d\xba\
↪x0a\x04\x03\x72\xb3\x27\x76\xf2\x6e\xd3\x87\x49\x88\xd0\x3e\xff\x5d\xa9\x93\x8c\
↪xa3'))
VMState.HALT
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
VM state of the transaction

current_hash: *UInt256*

Gets the hash of the current block.

```
>>> current_hash
b'\x3e\x65\xe5\x4d\x75\x5a\x94\x90\xd6\x98\x3a\x77\xe4\x82\xaf\x7a\x38\xc9\x8c\x1a\
↪xc6\xd9\xda\x48\xbd\x7c\x22\xb3\x2a\x9e\x34\xea'
```

current_index: int

Gets the index of the current block.

```
>>> current_index
10908937
```

```
>>> current_index
2108690
```

```
>>> current_index
3529755
```

Subpackages

class Block

Bases: object

Represents a block.

Check out Neo's Documentation to learn more about Blocks.

Variables

- **hash** (UInt256) – a unique identifier based on the unsigned data portion of the object
- **version** (*int*) – the data structure version of the block
- **previous_hash** (UInt256) – the hash of the previous block
- **merkle_root** (UInt256) – the merkle root of the transactions
- **timestamp** (*int*) – UTC timestamp of the block in milliseconds
- **nonce** (*int*) – a random number used once in the cryptography
- **index** (*int*) – the index of the block
- **primary_index** (*int*) – the primary index of the consensus node that generated this block
- **next_consensus** (UInt160) – the script hash of the consensus nodes that generates the next block
- **transaction_count** (*int*) – the number of transactions on this block

class Signer

Bases: object

Represents a signer.

Check out Neo's Documentation to learn more about Signers.

Variables

- **account** (UInt160) –
- **scopes** (WitnessScope) –
- **allowed_contracts** (*List[*UInt160*]*) –
- **allowed_groups** (*List[*UInt160*]*) –
- **rules** (*List[*WitnessRule*]*) –

class WitnessRule

Bases: object

Represents a witness rule.

Check out Neo's Documentation to learn more about WitnessRules.

Variables

- **action** (WitnessRuleAction) –
- **condition** (WitnessCondition) –

class WitnessCondition

Bases: object

Represents a witness condition.

Check out Neo's Documentation to learn more about WitnessConditions.

Variables

type (WitnessConditionType) –

class WitnessConditionType(*value*)

Bases: IntEnum

An enumeration.

BOOLEAN = 0

NOT = 1

AND = 2

OR = 3

SCRIPT_HASH = 24

GROUP = 25

CALLED_BY_ENTRY = 32

CALLED_BY_CONTRACT = 40

CALLED_BY_GROUP = 41

class WitnessRuleAction(*value*)

Bases: IntEnum

An enumeration.

DENY = 0

ALLOW = 1

class WitnessScope(*value*)

Bases: IntFlag

Determine the rules for a smart contract CheckWitness() sys call.

NONE = 0

No Contract was witnessed. Only sign the transaction.

CALLED_BY_ENTRY = 1

Allow the witness if the current calling script hash equals the entry script hash into the virtual machine. Using this prevents passing CheckWitness() in a smart contract called via another smart contract.

CUSTOM_CONTRACTS = 16

Allow the witness if called from a smart contract that is whitelisted in the signer allowed_contracts attribute.

CUSTOM_GROUPS = 32

Allow the witness if any public key is in the signer allowed_groups attribute is whitelisted in the contracts manifest.groups array.

WITNESS_RULES = 64

Allow the witness if the specified rules are satisfied

GLOBAL = 128

Allow the witness in all context. Equal to NEO 2.x's default behaviour.

class Transaction

Bases: object

Represents a transaction.

Check out Neo's Documentation to learn more about Transactions.

Variables

- **hash** (UInt256) – a unique identifier based on the unsigned data portion of the object
- **version** (*int*) – the data structure version of the transaction
- **nonce** (*int*) – a random number used once in the cryptography
- **sender** (UInt160) – the sender is the first signer of the transaction, they will pay the fees of the transaction
- **system_fee** (*int*) – the fee paid for executing the *script*
- **network_fee** (*int*) – the fee paid for the validation and inclusion of the transaction in a block by the consensus node
- **valid_until_block** (*int*) – indicates that the transaction is only valid before this block height
- **script** (*bytes*) – the array of instructions to be executed on the transaction chain by the virtual machine

class VMState(*value*)

Bases: IntEnum

Represents the VM execution state.

NONE

Indicates that the execution is in progress or has not yet begun.

HALT

Indicates that the execution has been completed successfully.

FAULT

Indicates that the execution has ended, and an exception that cannot be caught is thrown.

BREAK

Indicates that a breakpoint is currently being hit.

contract

call_contract(*script_hash:* UInt160, *method: str*, *args: Sequence = ()*, *call_flags:* CallFlags = *CallFlags.ALL*) → Any

Calls a smart contract given the method and the arguments. Since the return is type Any, you'll probably need to type cast the return.

```
>>> call_contract(NEO, 'balanceOf', [UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\
↪x01\x13\x19\xf3\xcf\xa4\xd2')])
100
```

Parameters

- **script_hash** (UInt160) – the target smart contract's script hash
- **method** (*str*) – the name of the method to be executed
- **args** (*Sequence[Any]*) – the specified method's arguments
- **call_flags** (CallFlags) – the CallFlags to be used to call the contract

Returns

the result of the specified method

Return type

Any

Raises

Exception – raised if there isn't a valid CallFlags, the script hash is not a valid smart contract or the method was not found or the arguments aren't valid to the specified method.

create_contract(*nef_file: bytes*, *manifest: bytes*, *data: Optional[Any] = None*) → *Contract*

Creates a smart contract given the script and the manifest.

```
>>> nef_file_ = get_script(); manifest_ = get_manifest()    # get the script and␣
↪manifest somehow
... create_contract(nef_file_, manifest_, None)             # smart contract will␣
↪be deployed
{
    'id': 2,
    'update_counter': 0,
    'hash': b'\x92\x8f+1q\x86z_@\x94\xf5pE\xcb\xb8 \x0f\\`Z',
    'nef': b'NEF3neo3-boa by COZ-1.0.0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x07W\x00\
↪x02xy\x9e@\xf9\7b\xbb\xcc',
    'manifest': {
        'name': 'TestContract',
        'group': [],
        'supported_standards': [],
        'abi': [[['test', [['a', 17], ['b', 17]], 17, 0, False]], []],
        'permissions': [],
        'trusts': [],
        'extras': 'null'
    },
}
```

Parameters

- **nef_file** (*bytes*) – the target smart contract's compiled nef
- **manifest** (*bytes*) – the manifest.json that describes how the script should behave
- **data** (*Any*) – the parameters for the _deploy function

Returns

the contract that was created

Return type

Contract

Raises

Exception – raised if the nef or the manifest are not a valid smart contract.

update_contract(*nef_file: bytes*, *manifest: bytes*, *data: Optional[Any] = None*)

Updates the executing smart contract given the script and the manifest.

```
>>> nef_file_ = get_script(); manifest_ = get_manifest()    # get the script and␣
↪manifest somehow
... update_contract(nef_file_, manifest_, None)             # smart contract will␣
↪be updated
None
```

Parameters

- **nef_file** (*bytes*) – the new smart contract's compiled nef
- **manifest** (*bytes*) – the new smart contract's manifest

- **data** (*Any*) – the parameters for the _deploy function

Raises

Exception – raised if the nef and the manifest are not a valid smart contract or the new contract is the same as the old one.

destroy_contract()

Destroy the executing smart contract.

```
>>> destroy_contract()
None
```

get_minimum_deployment_fee() → int

Gets the minimum fee of contract deployment.

```
>>> get_minimum_deployment_fee()
1000000000
```

Returns

the minimum fee of contract deployment

get_call_flags() → *CallFlags*

Gets the CallFlags in the current context.

```
>>> get_call_flags()
CallFlags.READ_ONLY
```

create_standard_account(*pub_key:* ECPoint) → *UInt160*

Calculates the script hash from a public key.

```
>>> create_standard_account(ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\x06\xb4\
↪x36\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\xb3\xa2\
↪xb6\x1a'))
b'\r\xa9g\xa4\x00C+\xf2\x7f\x8e\x8e\xb4o\xe8\xace\x9e\xcc\xde\x04'
```

Parameters

pub_key (ECPoint) – the given public key

Returns

the corresponding script hash of the public key

Return type

UInt160

create_multisig_account(*m: int*, *pub_keys: List[*ECPoint*]*) → *UInt160*

Calculates corresponding multisig account script hash for the given public keys.

```
>>> create_multisig_account(1, [ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\x06\
↪xb4\x36\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\xb3\
↪xa2\xb6\x1a')])
b'"5,\xd2\x9e\xe7\xb4\x02\x08b\xdbd\x1e\xedx\x82\x8fU(m'
```

Parameters

- **m** (*int*) – the minimum number of correct signatures need to be provided in order for the verification to pass.
- **pub_keys** (*List[ECPoint]*) – the public keys of the account

Returns
the hash of the corresponding account

Return type
UInt160

NEO: *UInt160*

NEO's token script hash.

```
>>> NEO
b'\xf5c\xea@\xbc(=M\x0e\x05\xc4\x8e\xa3\x05\xb3\xf2\xa0s@\xef'
```

GAS: *UInt160*

GAS' token script hash.

```
>>> GAS
b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\x01\x13\x19\xf3\xcf\xa4\xd2'
```

Subpackages

class CallFlags(*value*)

Bases: IntFlag

Defines special behaviors allowed when invoking smart contracts, e.g., chain calls, sending notifications and modifying states.

Check out Neo's Documentation to learn more about CallFlags.

NONE

Special behaviors of the invoked contract are not allowed, such as chain calls, sending notifications, modifying state, etc.

READ_STATES

Indicates that the called contract is allowed to read states.

WRITE_STATES

Indicates that the called contract is allowed to write states.

ALLOW_CALL

Indicates that the called contract is allowed to call another contract.

ALLOW_NOTIFY

Indicates that the called contract is allowed to send notifications.

STATES

Indicates that the called contract is allowed to read or write states.

READ_ONLY

Indicates that the called contract is allowed to read states or call another contract.

ALL

All behaviors of the invoked contract are allowed.

class Contract

Bases: object

Represents a contract that can be invoked.

Check out Neo's Documentation to learn about Smart Contracts.

Variables

- **id** (*int*) – the serial number of the contract
- **update_counter** (*int*) – the number of times the contract was updated
- **hash** (UInt160) – the hash of the contract
- **nef** (*bytes*) – the serialized Neo Executable Format (NEF) object holding of the smart contract code and compiler information
- **manifest** (ContractManifest) – the manifest of the contract

class ContractManifest

Bases: object

Represents the manifest of a smart contract.

When a smart contract is deployed, it must explicitly declare the features and permissions it will use.

When it is running, it will be limited by its declared list of features and permissions, and cannot make any behavior beyond the scope of the list.

For more details, check out NEP-15 or Neo's Documentation.

Variables

- **name** (*str*) – The name of the contract.
- **groups** (*List[*ContractGroup*]*) – The groups of the contract.
- **supported_standards** (*List[str]*) – Indicates which standards the contract supports. It can be a list of NEPs.
- **abi** (ContractAbi) – The ABI of the contract.
- **permissions** (*List[*ContractPermission*]*) – The permissions of the contract.

- **trusts** (*List[*ContractPermissionDescriptor*] or None*) – The trusted contracts and groups of the contract.

  If a contract is trusted, the user interface will not give any warnings when called by the contract.

- **extras** (*str*) – Custom user data as a json string.

class ContractPermission

Bases: object

Represents a permission of a contract. It describes which contracts may be invoked and which methods are called.

If a contract invokes a contract or method that is not declared in the manifest at runtime, the invocation will fail.

Variables

- **contract** (ContractPermissionDescriptor *or None*) – Indicates which contract to be invoked.

  It can be a hash of a contract, a public key of a group, or a wildcard *.

  If it specifies a hash of a contract, then the contract will be invoked; If it specifies a public key of a group, then any contract in this group may be invoked; If it specifies a wildcard *, then any contract may be invoked.

- **methods** (*List[str] or None*) – Indicates which methods to be called.

  It can also be assigned with a wildcard *. If it is a wildcard *, then it means that any method can be called.

class ContractPermissionDescriptor

Bases: object

Indicates which contracts are authorized to be called.

Variables

- **hash** (UInt160 *or None*) – The hash of the contract.
- **group** (ECPoint *or None*) – The group of the contracts.

class ContractGroup

Bases: object

Represents a set of mutually trusted contracts.

A contract will trust and allow any contract in the same group to invoke it, and the user interface will not give any warnings.

A group is identified by a public key and must be accompanied by a signature for the contract hash to prove that the contract is indeed included in the group.

Variables

- **pubkey** (ECPoint) – The public key of the group.
- **signature** (*bytes*) – The signature of the contract hash which can be verified by *pubkey*.

class ContractAbi

Bases: object

Represents the ABI of a smart contract.

For more details, check out NEP-14 or Neo's Documentation.

Variables

- **methods** (*List[*ContractMethodDescriptor*]*) – Gets the methods in the ABI.
- **events** (*List[*ContractEventDescriptor*]*) – Gets the events in the ABI.

class ContractMethodDescriptor

Bases: object

Represents a method in a smart contract ABI.

Variables

- **name** (*str*) – The name of the method.
- **parameters** (*List[*ContractParameterDefinition*]*) – The parameters of the method.
- **return_type** (ContractParameterType) – Indicates the return type of the method.
- **offset** (*int*) – The position of the method in the contract script.
- **safe** (*bool*) – Indicates whether the method is a safe method.

  If a method is marked as safe, the user interface will not give any warnings when it is called by other contracts.

class ContractEventDescriptor

Bases: object

Represents an event in a smart contract ABI.

Variables

- **name** (*str*) – The name of the event.
- **parameters** (*List[*ContractParameterDefinition*]*) – The parameters of the event.

class ContractParameterDefinition

Bases: object

Represents a parameter of an event or method in ABI.

Variables

- **name** (*str*) – The name of the parameter.
- **type** (ContractParameterType) – The type of the parameter.

class ContractParameterType(*value*)

Bases: IntEnum

An enumeration.

Any = 0

Boolean = 16

Integer = 17

ByteArray = 18

String = 19

Hash160 = 20

Hash256 = 21

PublicKey = 22

Signature = 23

Array = 32

Map = 34

InteropInterface = 48

Void = 255

crypto

sha256(*key: Any*) → bytes

Encrypts a key using SHA-256.

```
>>> sha256('unit test')
b'\xdau1>J\xc2W\xf8LN\xfb2\x0f\xbd\x01\x1cr@<\xf5\x93<\x90\xd2\xe3\xb8$\xd6H\x96\
↪xf8\x9a'
```

```
>>> sha256(10)
b'\x9c\x82r\x01\xb9@\x19\xb4/\x85pk\xc4\x9cY\xff\x84\xb5`M\x11\xca\xaf\xb9\n\xb9HV\
↪xc4\xe1\xddz'
```

Parameters
key (*Any*) – the key to be encrypted

Returns
a byte value that represents the encrypted key

Return type
bytes

ripemd160(*key: Any*) → bytes

Encrypts a key using RIPEMD-160.

```
>>> ripemd160('unit test')
b'H\x8e\xef\xf4Zh\x89:\xe6\xf1\xdc\x08\xdd\x8f\x01\rD\n\xbdH'
```

```
>>> ripemd160(10)
b'\xc0\xda\x02P8\xed\x83\xc6\x87\xdd\xc40\xda\x98F\xec\xb9\x7f9\x98'
```

Parameters
key (*Any*) – the key to be encrypted

Returns
a byte value that represents the encrypted key

Return type
bytes

hash160(*key: Any*) → bytes

Encrypts a key using HASH160.

```
>>> hash160('unit test')
b'#Q\xc9\xaf+c\x12\xb1\xb9\x9e\xa1\x89t\xa228g\xec\x0eF'
```

```
>>> hash160(10)
b'\x89\x86D\x19\xa8\xc3v%\x00\xfe\x9a\x98\xaf\x8f\xbbO3u\x08\xf0'
```

Parameters

key (*Any*) – the key to be encrypted

Returns

a byte value that represents the encrypted key

Return type

bytes

hash256(*key: Any*) → bytes

Encrypts a key using HASH256.

```
>>> hash256('unit test')
b'\xdau1>J\xc2W\xf8LN\xfb2\x0f\xbd\x01\x1cr@<\xf5\x93<\x90\xd2\xe3\xb8$\xd6H\x96\
↪xf8\x9a'
```

```
>>> hash256(10)
b'\x9c\x82r\x01\xb9@\x19\xb4/\x85pk\xc4\x9cY\xff\x84\xb5`M\x11\xca\xaf\xb9\n\xb9HV\
↪xc4\xe1\xddz'
```

Parameters

key (*Any*) – the key to be encrypted

Returns

a byte value that represents the encrypted key

Return type

bytes

check_sig(*pub_key:* ECPoint, *signature: bytes*) → bool

Checks the signature for the current script container.

```
>>> check_sig(ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\x06\xb4\x36\x8b\x1a\
↪x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\xb3\xa2\xb6\x1a'),
...           b'wrongsignature')
False
```

Parameters

- **pub_key** (ECPoint) – the public key of the account
- **signature** (*bytes*) – the signature of the current script container

Returns

whether the signature is valid or not

Return type
bool

check_multisig(*pubkeys: List[*ECPoint*]*, *signatures: List[bytes]*) → bool

Checks the signatures for the current script container.

```
>>> check_multisig([ECPoint(b"\x03\xcd\xb0\x67\xd9\x30\xfd\x5a\xda\xa6\xc6\x85\x45\
↪x01\x60\x44\xaa\xdd\xec\x64\xba\x39\xe5\x48\x25\x0e\xae\xa5\x51\x17\x2e\x53\x5c"),
...                  ECPoint(b"\x03l\x841\xccx\xb31w\xa6\x0bK\xcc\x02\xba\xf6\r\x05\
↪xfe\xe5\x03\x8es9\xd3\xa6\x88\xe3\x94\xc2\xcb\xd8C")],
...                 [b'wrongsignature1', b'wrongsignature2'])
False
```

Parameters

- **pubkeys** (*List[*ECPoint*]*) – a list of public keys
- **signatures** (*List[bytes]*) – a list of signatures

Returns
a boolean value that represents whether the signatures were validated

Return type
bool

verify_with_ecdsa(*message: bytes*, *pubkey:* ECPoint, *signature: bytes*, *curve:* NamedCurve) → bool

Using the elliptic curve, it checks if the signature of the message was originally produced by the public key.

```
>>> verify_with_ecdsa(b'unit test', ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\
↪x06\xb4\x36\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\
↪xb3\xa2\xb6\x1a'),
...                    b'wrong_signature', NamedCurve.SECP256R1)
False
```

Parameters

- **message** (*bytes*) – the encrypted message
- **pubkey** (ECPoint) – the public key that might have created the item
- **signature** (*bytes*) – the signature of the item
- **curve** (NamedCurve) – the curve that will be used by the ecdsa

Returns
a boolean value that represents whether the signature is valid

Return type
bool

murmur32(*data: bytes*, *seed: int*) → bytes

Computes the hash value for the specified byte array using the murmur32 algorithm.

```
>>> murmur32(b'unit test', 0)
b"\x90D'G"
```

Parameters

- **data** (*bytes*) – the input to compute the hash code for
- **seed** (*int*) – the seed of the murmur32 hash function

Returns
the hash value

Return type
bytes

bls12_381_add(*x: IBls12381*, *y: IBls12381*) → IBls12381

Add operation of two bls12381 points.

Parameters

- **x** (*IBls12381*) – The first point
- **y** (*IBls12381*) – The second point

Returns
the two points sum

Return type
IBls12381

bls12_381_deserialize(*data: bytes*) → IBls12381

Deserialize a bls12381 point.

Parameters
data (*bytes*) – The point as byte array

Returns
the deserialized point

Return type
IBls12381

bls12_381_equal(*x: IBls12381*, *y: IBls12381*) → bool

Determines whether the specified points are equal.

Parameters

- **x** (*bytes*) – The first point
- **y** (*bytes*) – The second point

Returns
whether the specified points are equal or not

Return type
bool

bls12_381_mul(*x: IBls12381*, *mul: bytes*, *neg: bool*) → IBls12381

Mul operation of gt point and multiplier.

Parameters

- **x** (*IBls12381*) – The point
- **mul** (*int*) – Multiplier, 32 bytes, little-endian
- **neg** (*bool*) – negative number

Returns
the two points product

Return type
IBls12381

bls12_381_pairing(*g1: IBls12381*, *g2: IBls12381*) → IBls12381

Pairing operation of g1 and g2.

Parameters

- **g1** (*IBls12381*) – The g1 point
- **g2** (*IBls12381*) – The g2 point

Returns
the two points pairing

Return type
IBls12381

bls12_381_serialize(*g: IBls12381*) → bytes

Serialize a bls12381 point.

Parameters
g (*IBls12381*) – The point to be serialized.

Returns
the serialized point

Return type
bytes

Subpackages

class NamedCurve(*value*)

Bases: IntFlag

Represents the named curve used in ECDSA.

Check out Neo's Documentation to learn more about ECDSA signing.

SECP256K1
The secp256k1 curve.

SECP256R1
The secp256r1 curve, which known as prime256v1 or nistP-256.

iterator

class Iterator

Bases: object

The iterator for smart contracts.

property value: *Any*

Gets the element in the collection at the current position of the iterator.

Returns
the element in the collection at the current position of the iterator

Return type
Any

next() → bool

Advances the iterator to the next element of the collection.

```
>>> from boa3.builtin.interop import storage
... iterator = storage.find(b'prefix')
... iterator.next()
True
```

Returns
true if it advanced, false if there isn't a next element

Return type
bool

json

json_serialize(*item: Any*) → str

Serializes an item into a json.

```
>>> json_serialize({'one': 1, 'two': 2, 'three': 3})
'{"one":1,"two":2,"three":3}'
```

Parameters
item (*Any*) – The item that will be serialized

Returns
The serialized item

Return type
str

Raises
Exception – raised if the item is an integer value out of the Neo's accepted range, is a dictionary with a bytearray key, or isn't serializable.

json_deserialize(*json: str*) → Any

Deserializes a json into some valid type.

```
>>> json_deserialize('{"one":1,"two":2,"three":3}')
{'one': 1, 'three': 3, 'two': 2}
```

Parameters
json (*str*) – A json that will be deserialized

Returns
The deserialized json

Return type
Any

Raises
Exception – raised if jsons deserialization is not valid.

oracle

class Oracle

Bases: object

Neo Oracle Service is an out-of-chain data access service built into Neo N3. It allows users to request the external data sources in smart contracts, and Oracle nodes designated by the committee will access the specified data source then pass the result in the callback function to continue executing the smart contract logic.

Check out Neo's Documentation to learn more about Oracles.

hash: *UInt160*

classmethod request(*url: str*, *request_filter: Optional[str]*, *callback: str*, *user_data: Any*, *gas_for_response: int*)

Requests an information from outside the blockchain.

This method just requests data from the oracle, it won't return the result.

```
>>> Oracle.request('https://dora.coz.io/api/v1/neo3/testnet/asset/
↪0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5',
...                 '', 'callback_name', None, 10 * 10 ** 8)
None
```

Parameters

- **url** (*str*) – External url to retrieve the data
- **request_filter** (*str or None*) – Filter to the request.

  See JSONPath format https://github.com/atifaziz/JSONPath
- **callback** (*str*) – Method name that will be as a callback.

  This method must be public and implement the following interface:

  `(url: str, user_data: Any, code: int, result: bytes) -> None`
- **user_data** (*Any*) – Optional data. It'll be returned as the same when the callback is called
- **gas_for_response** (*int*) – Amount of GAS needed to run the callback method.

  It MUST NOT be specified as the user representation.

  If it costs 1 gas, this value must be 1_00000000 (with the 8 decimals)

classmethod get_price() → int

Gets the price for an Oracle request.

```
>>> Oracle.get_price()
50000000
```

Returns
the price for an Oracle request

class OracleResponseCode(*value*)

Bases: IntFlag

Represents the response code for the oracle request.

SUCCESS

Indicates that the request has been successfully completed.

PROTOCOL_NOT_SUPPORTED

Indicates that the protocol of the request is not supported.

CONSENSUS_UNREACHABLE

Indicates that the oracle nodes cannot reach a consensus on the result of the request.

NOT_FOUND

Indicates that the requested Uri does not exist.

TIME_OUT

Indicates that the request was not completed within the specified time.

FORBIDDEN

Indicates that there is no permission to request the resource.

RESPONSE_TOO_LARGE

Indicates that the data for the response is too large.

INSUFFICIENT_FUNDS

Indicates that the request failed due to insufficient balance.

CONTENT_TYPE_NOT_SUPPORTED

Indicates that the content-type of the request is not supported.

ERROR

Indicates that the request failed due to other errors.

policy

get_exec_fee_factor() → int

Gets the execution fee factor. This is a multiplier that can be adjusted by the committee to adjust the system fees for transactions.

```
>>> get_exec_fee_factor()
30
```

Returns
the execution fee factor

Return type
int

get_fee_per_byte() → int

Gets the network fee per transaction byte.

```
>>> get_fee_per_byte()
1000
```

Returns
the network fee per transaction byte

Return type
int

get_storage_price() → int

Gets the storage price.

```
>>> get_storage_price()
100000
```

Returns
the snapshot used to read data

Return type
int

is_blocked(*account:* UInt160) → bool

Determines whether the specified account is blocked.

```
>>> is_blocked(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\x01\x13\x19\xf3\xcf\xa4\
↪xd2'))
False
```

Parameters
account (UInt160) – the account to be checked

Returns
whether the account is blocked or not

Return type
bool

role

get_designated_by_role(*role:* Role, *index: int*) → *ECPoint*

Gets the list of nodes for the specified role.

```
>>> get_designated_by_role(Role.ORACLE, 0)
[]
```

Parameters

- **role** (Role) – the type of the role
- **index** (*int*) – the index of the block to be queried

Returns
the public keys of the nodes

Return type
ECPoint

Subpackages

class Role(*value*)

Bases: IntFlag

Represents the roles in the NEO system. They are the permission types of the native contract *RoleManagement*.

STATE_VALIDATOR

The validators of state. Used to generate and sign the state root.

ORACLE

The nodes used to process Oracle requests.

NEO_FS_ALPHABET_NODE

NeoFS Alphabet nodes.

runtime

check_witness(*hash_or_pubkey: Union[*UInt160, ECPoint*]*) → bool

Verifies that the transactions or block of the calling contract has validated the required script hash.

```
>>> check_witness(calling_script_hash)
True
```

```
>>> check_witness(UInt160(bytes(20)))
False
```

Parameters
hash_or_pubkey (UInt160 *or* ECPoint) – script hash or public key to validate

Returns
a boolean value that represents whether the script hash was verified

Return type
bool

notify(*state: Any*, *notification_name: Optional[str] = None*)

Notifies the client from the executing smart contract.

```
>>> var = 10
... notify(var)     # An event will be triggered
None
```

```
>>> var = 10
... notify(var, 'custom event name')     # An event will be triggered
None
```

Parameters

- **state** (*Any*) – the notification message
- **notification_name** (*str*) – name that'll be linked to the notification

log(*message: str*)

Show log messages to the client from the executing smart contract.

```
>>> log('log sent')     # An event that can be shown on the CLI
log sent
```

Parameters
message (*str*) – the log message

get_trigger() → *TriggerType*

Return the smart contract trigger type.

```
>>> get_trigger()
TriggerType.APPLICATION
```

Returns
a value that represents the contract trigger type

Return type
TriggerType

get_notifications(*script_hash:* UInt160 = *b''*) → List[*Notification*]

This method gets current invocation notifications from specific 'script_hash'.

```
>>> notify(1); notify(2); notify(3)
... get_notifications(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\x01\x13\x19\xf3\
↪xcf\xa4\xd2'))
[
    [
        b'8\xfe\x11\n\xff7J\xb8}\xe9x6@\xea\x0b\x00\xf1|\x82v',
```

(continues on next page)

(continued from previous page)

```
        'notify',
        [1]
    ],
    [
        b'8\xfe\x11\n\xff7J\xb8}\xe9x6@\xea\x0b\x00\xf1|\x82v',
        'notify',
        [2]
    ],
    [
        b'8\xfe\x11\n\xff7J\xb8}\xe9x6@\xea\x0b\x00\xf1|\x82v',
        'notify',
        [3]
    ]
]
```

Parameters
script_hash (UInt160) – must have 20 bytes, but if it's all zero 0000…0000 it refers to all existing notifications (like a * wildcard)

Returns
It will return an array of all matched notifications

Return type
List[*Notification*]

get_network() → int

Gets the magic number of the current network.

```
>>> get_network()
860243278
```

Returns
the magic number of the current network

Return type
int

burn_gas(*gas: int*)

Burns GAS to benefit the NEO ecosystem.

```
>>> burn_gas(1000)
None
```

Parameters
gas (*int*) – the amount of GAS that will be burned

Raises
Exception – raised if gas value is negative.

get_random() → int

Gets the next random number.

```
>>> get_random()
191320526825634396960813166838892720709
```

```
>>> get_random()
990836694840016825626317290231915458O9
```

```
>>> get_random()
328056213623902365838800581788496514419
```

Returns
the next random number

Return type
int

load_script(*script: bytes*, *args: Sequence = ()*, *flags:* CallFlags = *CallFlags.NONE*) → Any

Loads a script at runtime.

```
>>> from typing import cast
... from boa3.builtin.vm import Opcode
... cast(int, load_script(Opcode.ADD, [10, 2]))
12
```

address_version: *int*

Gets the address version of the current network.

```
>>> address_version
53
```

executing_script_hash: *UInt160*

Gets the script hash of the current context.

```
>>> executing_script_hash
b'^b]\x90#\xbf\xcc\x1f\xd8\x9e\xe3\xa4zd\x14\xa4\xf0\x96\x9f`'
```

calling_script_hash: *UInt160*

Gets the script hash of the calling contract.

```
>>> calling_script_hash
b'\x05\x7f\xc2\x9d\xba\xb2\xc1x\xf5\x81\x83\xbf\xcb\x87/\xc3!\xca\xe1\xd0'
```

time: *int*

Gets the timestamp of the current block.

```
>>> time
1685395697108
```

gas_left: *int*

Gets the remaining GAS that can be spent in order to complete the execution.

```
>>> gas_left
1999015490
```

platform: *str*

Gets the name of the current platform.

```
>>> platform
'NEO'
```

invocation_counter: *int*

Gets the number of times the current contract has been called during the execution.

```
>>> invocation_counter
1
```

entry_script_hash: *UInt160*

Gets the script hash of the entry context.

```
>>> entry_script_hash
b'\tK\xb31\xa8\x13\x80`\xad\xf6\xda\xdf\xc6R\x9b\xfdB\xbf\x83\x8f'
```

script_container: *Any*

Gets the current script container.

```
>>> script_container
[
    b'\xf1y\xc2\xd6\x1c\xb6\x98\xa4\xdc\xf3\xd67s\xd7E\xf0<;\x98+\xa2T\x03P,T\xe8\
↪xc6{ \x101',
    0,
    442907905,
    '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00',
    0,
    0,
```

(continues on next page)

(continued from previous page)

```
    0,
    ''
]
```

```
>>> script_container
[
    b"S{\xed'\x85&\xf5\x93U=\xc1\xbf'\x95\xc4/\x80X\xdb\xd5\xa1-\x97q\x85\xe3I\xe5\
↪x99cd\x04",
    0,
    '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    1468595301000,
    2083236893,
    0,
    0,
    b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\x00\x12\x04',
    0,
]
```

Subpackages

class Notification

Bases: object

Represents a notification.

Variables

- **script_hash** (UInt160) – the script hash of the notification sender
- **event_name** (*str*) – the notification's name
- **state** (*tuple*) – a tuple value storing all the notification contents.

class TriggerType(*value*)

Bases: IntFlag

Represents the triggers for running smart contracts. Triggers enable the contract to execute different logic under different usage scenarios.

Check out Neo's Documentation to learn more about TriggerTypes.

ON_PERSIST

Indicate that the contract is triggered by the system to execute the OnPersist method of the native contracts.

POST_PERSIST

Indicate that the contract is triggered by the system to execute the PostPersist method of the native contracts.

SYSTEM
The combination of all system triggers.

VERIFICATION
Indicates that the contract is triggered by the verification of a IVerifiable.

APPLICATION
Indicates that the contract is triggered by the execution of transactions.

ALL
The combination of all triggers.

stdlib

base58_encode(*key: bytes*) → str
Encodes a bytes value using base58.

```
>>> base58_encode(b'unit test')
b"2VhL46g69A1mu"
```

Parameters
key (*bytes*) – bytes value to be encoded

Returns
the encoded string

Return type
str

base58_decode(*key: str*) → bytes
Decodes a string value encoded with base58.

```
>>> base58_decode('2VhL46g69A1mu')
b"unit test"
```

Parameters
key (*str*) – string value to be decoded

Returns
the decoded bytes

Return type
bytes

base58_check_encode(*key: bytes*) → str
Converts a bytes value to its equivalent str representation that is encoded with base-58 digits. The encoded str contains the checksum of the binary data.

```
>>> base58_check_encode(b'unit test')
b"AnJcKqvgBwKxsjX75o"
```

Parameters
key (*bytes*) – bytes value to be encoded

Returns
the encoded string

Return type
str

base58_check_decode(*key: str*) → bytes

Converts the specified str, which encodes binary data as base-58 digits, to an equivalent bytes value. The encoded str contains the checksum of the binary data.

```
>>> base58_check_decode('AnJcKqvgBwKxsjX75o')
b"unit test"
```

Parameters
key (*str*) – string value to be decoded

Returns
the decoded bytes

Return type
bytes

base64_encode(*key: bytes*) → str

Encodes a bytes value using base64.

```
>>> base64_encode(b'unit test')
b"dW5pdCB0ZXN0"
```

Parameters
key (*bytes*) – bytes value to be encoded

Returns
the encoded string

Return type
str

base64_decode(*key: str*) → bytes

Decodes a string value encoded with base64.

```
>>> base64_decode("dW5pdCB0ZXN0")
b"unit test"
```

Parameters
key (*str*) – string value to be decoded

Returns
the decoded bytes

Return type
bytes

`serialize`(*item: Any*) → bytes

Serializes the given value into its bytes representation.

```
>>> serialize('42')
b'(}42'
```

```
>>> serialize(42)
b'!{*'
```

```
>>> serialize([2, 3, 5, 7])
b'@!{}!{!{!{'
```

```
>>> serialize({1: 1, 2: 1, 3: 2})
b'H!{{!{{!{}!{{!{!{}'
```

Parameters
item (*Any*) – value to be serialized

Returns
the serialized value

Return type
bytes

Raises
Exception – raised if the item's type is not serializable.

`deserialize`(*data: bytes*) → Any

Deserializes the given bytes value.

```
>>> deserialize(b'(}42')
'42'
```

```
>>> deserialize(b'!{*')
42
```

```
>>> deserialize(b'@!{}!{!{!{')
[2, 3, 5, 7]
```

```
>>> deserialize(b'H!{{!{{!{}!{{!{!{}')
{1: 1, 2: 1, 3: 2}
```

Parameters
data (*bytes*) – serialized value

Returns
the deserialized result

Return type
Any

Raises
Exception – raised when the date doesn't represent a serialized value.

atoi(*value: str*, *base: int = 10*) → int

Converts a character string to a specific base value, decimal or hexadecimal. The default is decimal.

```
>>> atoi('10')
10
```

```
>>> atoi('123')
123
```

```
>>> atoi('1f', 16)
31
```

```
>>> atoi('ff', 16)
-1
```

Parameters

- **value** (*str*) – the int value as a string
- **base** (*int*) – the value base

Returns
the equivalent value

Return type
int

Raises
Exception – raised when base isn't 10 or 16.

itoa(*value: int*, *base: int = 10*) → str

Converts the specific type of value to a decimal or hexadecimal string. The default is decimal.

```
>>> itoa(10)
'10'
```

```
>>> itoa(123)
'123'
```

```
>>> itoa(-1, 16)
'f'
```

```
>>> itoa(15, 16)
'0f'
```

Parameters

- **value** (*int*) – the int value
- **base** (*int*) – the value's base

Returns
the converted string

Return type
int

memory_search(*mem: Union[bytes, str]*, *value: Union[bytes, str]*, *start: int = 0*, *backward: bool = False*) → int

Searches for a given value in a given memory.

```
>>> memory_search('abcde', 'a', 0)
0
```

```
>>> memory_search('abcde', 'e', 0)
4
```

Parameters

- **mem** (*bytes or str*) – the memory
- **value** (*bytes or str*) – the value
- **start** (*int*) – the index the search should start from
- **backward** (*bool*) – whether it should invert the memory

Returns
the index of the value in the memory. Returns -1 if it's not found

Return type
int

memory_compare(*mem1: Union[bytes, str]*, *mem2: Union[bytes, str]*) → int

Compares a memory with another one.

```
>>> memory_compare('abc', 'abc')
0
```

```
>>> memory_compare('ABC', 'abc')
-1
```

```
>>> memory_compare('abc', 'ABC')
1
```

Parameters

- **mem1** (*bytes or str*) – a memory to be compared to another one
- **mem2** (*bytes or str*) – a memory that will be compared with another one

Returns
-1 if mem1 precedes mem2, 0 if mem1 and mem2 are equal, 1 if mem1 follows mem2

Return type
int

storage

get_context() → *StorageContext*

Gets current storage context.

```
>>> get_context()       # StorageContext cannot be read outside the blockchain
_InteropInterface
```

Returns
the current storage context

Return type
StorageContext

get(*key: bytes*, *context: Optional[*StorageContext*] = None*) → bytes

Gets a value from the persistent store based on the given key.

```
>>> put(b'unit', 'test')
... get(b'unit')
'test'
```

```
>>> get(b'fake_key')
''
```

Parameters

- **key** (*bytes*) – value identifier in the store
- **context** (StorageContext) – storage context to be used

Returns
the value corresponding to given key for current storage context

Return type
bytes

get_read_only_context() → *StorageContext*

Gets current read only storage context.

```
>>> get_context()       # StorageContext cannot be read outside the blockchain
_InteropInterface
```

Returns
the current read only storage context

Return type
StorageContext

put(*key: bytes*, *value: Union[int, bytes, str]*, *context: Optional[*StorageContext*] = None*)

Inserts a given value in the key-value format into the persistent storage.

```
>>> put(b'unit', 'test')
None
```

Parameters

- **key** (*bytes*) – the identifier in the store for the new value
- **value** (*int or str or bytes*) – value to be stored
- **context** (StorageContext) – storage context to be used

delete(*key: bytes*, *context: Optional[*StorageContext*] = None*)

Removes a given key from the persistent storage if exists.

```
>>> put(b'unit', 'test')
... delete()
... get(b'unit')
''
```

Parameters

- **key** (*bytes*) – the identifier in the store for the new value
- **context** (StorageContext) – storage context to be used

find(*prefix: bytes*, *context: Optional[*StorageContext*] = None*, *options:* FindOptions *= FindOptions.NONE*) → *Iterator*

Searches in the storage for keys that start with the given prefix.

```
>>> put(b'a1', 'one')
... put(b'a2', 'two')
... put(b'a3', 'three')
... put(b'b4', 'four')
... findIterator = find(b'a')
... findResults = []
... while findIterator.next():
...     findResults.append(findIterator.value)
... findResults
['one', 'two', 'three']
```

Parameters

- **prefix** (*bytes*) – prefix to find the storage keys
- **context** (StorageContext) – storage context to be used
- **options** (FindOptions) – the options of the search

Returns

an iterator with the search results

Return type

Iterator

Subpackages

class FindOptions(*value*)

Bases: IntEnum

Represents the options you can use when trying to find a set of values inside the storage.

Check out Neo's Documentation to learn more about the FindOption class.

NONE

No option is set. The results will be an iterator of (key, value).

KEYS_ONLY

Indicates that only keys need to be returned. The results will be an iterator of keys.

REMOVE_PREFIX

Indicates that the prefix byte of keys should be removed before return.

VALUES_ONLY

Indicates that only values need to be returned. The results will be an iterator of values.

DESERIALIZE_VALUES

Indicates that values should be deserialized before return.

PICK_FIELD_0

Indicates that only the field 0 of the deserialized values need to be returned. This flag must be set together with DESERIALIZE_VALUES, e.g., *DESERIALIZE_VALUES | PICK_FIELD_0*

PICK_FIELD_1

Indicates that only the field 1 of the deserialized values need to be returned. This flag must be set together with DESERIALIZE_VALUES, e.g., *DESERIALIZE_VALUES | PICK_FIELD_1*

BACKWARDS

Indicates that results should be returned in backwards (descending) order.

class StorageContext

Bases: object

The storage context used to read and write data in smart contracts.

Check out Neo's Documentation to learn more about the StorageContext class.

create_map(*prefix: bytes*) → *StorageMap*

Creates a storage map with the given prefix.

Parameters

prefix (*bytes*) – the identifier of the storage map

Returns
a map with the key-values in the storage that match with the given prefix

Return type
StorageMap

`as_read_only()` → *StorageContext*

Converts the specified storage context to a new readonly storage context.

Returns
current StorageContext as ReadOnly

Return type
StorageContext

`class StorageMap(context, prefix: bytes)`

Bases: `object`

The key-value storage for the specific prefix in the given storage context.

Check out Neo's Documentation to learn more about StorageMap.

`get(key: bytes)` → bytes

Gets a value from the map based on the given key.

Parameters
key (*bytes*) – value identifier in the store

Returns
the value corresponding to given key for current storage context

Return type
bytes

`put(key: bytes, value: Union[int, bytes, str])`

Inserts a given value in the key-value format into the map.

Parameters

- **key** (*bytes*) – the identifier in the store for the new value
- **value** (*int or str or bytes*) – value to be stored

`delete(key: bytes)`

Removes a given key from the map if exists.

Parameters
key (*bytes*) – the identifier in the store for the new value

nativecontract

Subpackages

contractmanagement

`class ContractManagement`

Bases: `object`

A class used to represent the ContractManagement native contract.

Check out Neo's Documentation to learn more about the ContractManagement class.

hash: *UInt160*

classmethod get_minimum_deployment_fee() → int

Gets the minimum fee of contract deployment.

```
>>> ContractManagement.get_minimum_deployment_fee()
1000000000
```

Returns
the minimum fee of contract deployment

classmethod get_contract(*script_hash:* UInt160) → Optional[*Contract*]

Gets a contract with a given hash. If the script hash is not associated with a smart contract, then it will return None.

```
>>> ContractManagement.get_contract(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\
↪xe3Ua\x01\x13\x19\xf3\xcf\xa4\xd2'))    # GAS script hash
{
    'id': -6,
    'update_counter': 0,
    'hash': b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\x01\x13\x19\xf3\xcf\xa4\xd2',
    'nef': b'NEF3neo-core-v3.0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00#\x10A\x1a\xf7{g@\x10A\x1a\xf7{g@\x10A\x1a\xf7{g@\x10A\
↪x1a\xf7{g@\x10A\x1a\xf7{g@QA\xc7\x9e',
    'manifest': {
        'name': 'GasToken',
        'group': [],
        'supported_standards': ['NEP-17'],
        'abi': [[['balanceOf', [['account', 20]], 17, 0, True], ['decimals', [],
↪ 17, 7, True], ['symbol', [], 19, 14, True], ['totalSupply', [], 17, 21,␣
↪True], ['transfer', [['from', 20], ['to', 20], ['amount', 17], ['data', 0]],␣
↪16, 28, False]], [['Transfer', [['from', 20], ['to', 20], ['amount', 17]]]]],
        'permissions': [[None, None]],
        'trusts': [],
        'extras': 'null'
    },
}
```

```
>>> ContractManagement.get_contract(UInt160(bytes(20)))    # there is no smart␣
↪contract associated with this script hash
None
```

Parameters
script_hash (UInt160) – a smart contract hash

Returns
a contract

Return type
Contract

Raises
Exception – raised if hash length isn't 20 bytes.

classmethod has_method(*hash:* UInt160, *method: str*, *parameter_count: int*) → bool

Check if a method exists in a contract.

```
>>> ContractManagement.has_method(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\
↪x01\x13\x19\xf3\xcf\xa4\xd2'),
...                               'balanceOf', 1)    # GAS script hash
True
```

```
>>> ContractManagement.has_method(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\
↪x01\x13\x19\xf3\xcf\xa4\xd2'),
...                               'balanceOf', 10)    # GAS script hash
False
```

```
>>> ContractManagement.has_method(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\
↪x01\x13\x19\xf3\xcf\xa4\xd2'),
...                               'invalid', 1)    # GAS script hash
False
```

Parameters

- **hash** (UInt160) – The hash of the deployed contract
- **method** (*str*) – The name of the method
- **parameter_count** (*int*) – The number of parameters

Returns

whether the method exists or not

Return type

bool

Raises

Exception – raised if hash length isn't 20 bytes or if the parameter_count is less than 0.

classmethod deploy(*nef_file: bytes*, *manifest: bytes*, *data: Optional[Any] = None*) → *Contract*

Creates a smart contract given the script and the manifest.

```
>>> nef_file_ = get_script(); manifest_ = get_manifest()    # get the script␣
↪and manifest somehow
... ContractManagement.deploy(nef_file_, manifest_, None)              # smart␣
↪contract will be deployed
{
    'id': 2,
    'update_counter': 0,
    'hash': b'\x92\x8f+1q\x86z_@\x94\xf5pE\xcb\xb8 \x0f\\`Z',
    'nef': b'NEF3neo3-boa by COZ-1.0.0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x07W\x00\x02xy\x9e@\xf9\7b\xbb\xcc',
    'manifest': {
        'name': 'TestContract',
        'group': [],
        'supported_standards': [],
        'abi': [[['test', [['a', 17], ['b', 17]], 17, 0, False]], []],
```

(continues on next page)

(continued from previous page)

```
        'permissions': [],
        'trusts': [],
        'extras': 'null'
    },
}
```

Parameters

- **nef_file** (*bytes*) – the target smart contract's compiled nef
- **manifest** (*bytes*) – the manifest.json that describes how the script should behave
- **data** (*Any*) – the parameters for the _deploy function

Returns
the contract that was created

Return type
Contract

Raises
Exception – raised if the nef or the manifest are not a valid smart contract.

classmethod update(*nef_file: bytes*, *manifest: bytes*, *data: Optional[Any] = None*)

Updates the executing smart contract given the script and the manifest.

```
>>> nef_file_ = get_script(); manifest_ = get_manifest()    # get the script␣
↪and manifest somehow
... ContractManagement.update(nef_file_, manifest_, None)              # smart␣
↪contract will be updated
None
```

Parameters

- **nef_file** (*bytes*) – the new smart contract's compiled nef
- **manifest** (*bytes*) – the new smart contract's manifest
- **data** (*Any*) – the parameters for the _deploy function

Raises
Exception – raised if the nef and the manifest are not a valid smart contract or the new contract is the same as the old one.

classmethod destroy()

Destroy the executing smart contract.

```
>>> ContractManagement.destroy()
None
```

class Contract

Bases: object

Represents a contract that can be invoked.

Check out Neo's Documentation to learn about Smart Contracts.

Variables

- **id** (*int*) – the serial number of the contract
- **update_counter** (*int*) – the number of times the contract was updated
- **hash** (UInt160) – the hash of the contract
- **nef** (*bytes*) – the serialized Neo Executable Format (NEF) object holding of the smart contract code and compiler information
- **manifest** (ContractManifest) – the manifest of the contract

criptolib

class CryptoLib

Bases: object

A class used to represent the CryptoLib native contract.

Check out Neo's Documentation to learn more about the CryptoLib class.

hash: *UInt160*

classmethod murmur32(*data: bytes*, *seed: int*) → bytes

Computes the hash value for the specified byte array using the murmur32 algorithm.

```
>>> CryptoLib.murmur32(b'unit test', 0)
b"\x90D'G"
```

Parameters

- **data** (*bytes*) – the input to compute the hash code for
- **seed** (*int*) – the seed of the murmur32 hash function

Returns
the hash value

Return type
bytes

classmethod sha256(*key: Any*) → bytes

Encrypts a key using SHA-256.

```
>>> CryptoLib.sha256('unit test')
b'\xdau1>J\xc2W\xf8LN\xfb2\x0f\xbd\x01\x1cr@<\xf5\x93<\x90\xd2\xe3\xb8$\xd6H\
↪x96\xf8\x9a'
```

```
>>> CryptoLib.sha256(10)
b'\x9c\x82r\x01\xb9@\x19\xb4/\x85pk\xc4\x9cY\xff\x84\xb5`M\x11\xca\xaf\xb9\n\
↪xb9HV\xc4\xe1\xddz'
```

Parameters
key (*Any*) – the key to be encrypted

Returns
a byte value that represents the encrypted key

Return type
bytes

classmethod ripemd160(*key: Any*) → bytes

Encrypts a key using RIPEMD-160.

```
>>> CryptoLib.ripemd160('unit test')
b'H\x8e\xef\xf4Zh\x89:\xe6\xf1\xdc\x08\xdd\x8f\x01\rD\n\xbdH'
```

```
>>> CryptoLib.ripemd160(10)
b'\xc0\xda\x02P8\xed\x83\xc6\x87\xdd\xc40\xda\x98F\xec\xb9\x7f9\x98'
```

Parameters
key (*Any*) – the key to be encrypted

Returns
a byte value that represents the encrypted key

Return type
bytes

classmethod verify_with_ecdsa(*message: bytes*, *pubkey:* ECPoint, *signature: bytes*, *curve:* NamedCurve) → bool

Using the elliptic curve, it checks if the signature of the message was originally produced by the public key.

```
>>> CryptoLib.verify_with_ecdsa(b'unit test', ECPoint(b'\x03\x5a\x92\x8f\x20\
↪x16\x39\x20\x4e\x06\xb4\x36\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\
↪x81\x51\xb7\x4f\xaa\xb3\xa2\xb6\x1a'),
...                                b'wrong_signature', NamedCurve.SECP256R1)
False
```

Parameters

- **message** (*bytes*) – the encrypted message
- **pubkey** (ECPoint) – the public key that might have created the item
- **signature** (*bytes*) – the signature of the item
- **curve** (NamedCurve) – the curve that will be used by the ecdsa

Returns
a boolean value that represents whether the signature is valid

Return type
bool

classmethod bls12_381_add(*x: IBls12381*, *y: IBls12381*) → IBls12381

Add operation of two bls12381 points.

Parameters

- **x** (*IBls12381*) – The first point
- **y** (*IBls12381*) – The second point

Returns
the two points sum

Return type
IBls12381

classmethod bls12_381_deserialize(*data: bytes*) → IBls12381

Deserialize a bls12381 point.

Parameters
data (*bytes*) – The point as byte array

Returns
the deserialized point

Return type
IBls12381

classmethod bls12_381_equal(*x: IBls12381*, *y: IBls12381*) → bool

Determines whether the specified points are equal.

Parameters

- **x** (*IBls12381*) – The first point
- **y** (*IBls12381*) – The second point

Returns
whether the specified points are equal or not

Return type
bool

classmethod bls12_381_mul(*x: IBls12381*, *mul: bytes*, *neg: bool*) → IBls12381

Mul operation of gt point and multiplier.

Parameters

- **x** (*IBls12381*) – The point
- **mul** (*int*) – Multiplier, 32 bytes, little-endian
- **neg** (*bool*) – negative number

Returns
the two points product

Return type
IBls12381

classmethod bls12_381_pairing(*g1: IBls12381*, *g2: IBls12381*) → IBls12381

Pairing operation of g1 and g2.

Parameters

- **g1** (*IBls12381*) – The g1 point
- **g2** (*IBls12381*) – The g2 point

Returns
the two points pairing

Return type
IBls12381

classmethod bls12_381_serialize(*g: IBls12381*) → bytes

Serialize a bls12381 point.

Parameters
g (*IBls12381*) – The point to be serialized.

Returns
the serialized point

Return type
bytes

class NamedCurve(*value*)

Bases: IntFlag

Represents the named curve used in ECDSA.

Check out Neo's Documentation to learn more about ECDSA signing.

SECP256K1

The secp256k1 curve.

SECP256R1

The secp256r1 curve, which known as prime256v1 or nistP-256.

class IBls12381

Bases: object

gas

class GAS

Bases: object

A class used to represent the GAS native contract.

Check out Neo's Documentation to learn more about the GAS class.

hash: *UInt160*

classmethod symbol() → str

Gets the symbol of GAS.

```
>>> GAS.symbol()
'GAS'
```

Returns
the GAS string.

Return type
str

classmethod decimals() → int

Gets the amount of decimals used by GAS.

```
>>> GAS.decimals()
8
```

Returns
the number 8.

Return type
int

classmethod totalSupply() → int

Gets the total token supply deployed in the system.

```
>>> GAS.totalSupply()
5199999098939320
```

```
>>> GAS.totalSupply()
5522957322800300
```

Returns
the total token supply deployed in the system.

Return type
int

classmethod balanceOf(*account:* UInt160) → int

Get the current balance of an address.

```
>>> GAS.balanceOf(UInt160(bytes(20)))
0
```

```
>>> GAS.balanceOf(UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'))
1000000000
```

Parameters
account (UInt160) – the account's address to retrieve the balance for

Returns
the account's balance

Return type
int

classmethod transfer(*from_address:* UInt160, *to_address:* UInt160, *amount:* *int*, *data:* *Optional[Any]* = *None*) → bool

Transfers an amount of GAS from one account to another.

If the method succeeds, it will fire the *Transfer* event and must return true, even if the amount is 0, or from and to are the same address.

```
>>> GAS.transfer(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6   #\xe1\x9096\
↪x89U\xac\x13'),     # this script hash needs to have signed the transaction␣
↪or block
...               UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'),
...               10000, None)
True
```

```
>>> GAS.transfer(UInt160(bytes(20)),
...              UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'),
...              10000, None)
False
```

```
>>> GAS.transfer(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6   #\xe1\x9096\
↪x89U\xac\x13'),
...              UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'),
...              -1, None)
False
```

Parameters

- **from_address** (UInt160) – the address to transfer from
- **to_address** (UInt160) – the address to transfer to
- **amount** (*int*) – the amount of GAS to transfer
- **data** (*Any*) – whatever data is pertinent to the onNEP17Payment method

Returns
whether the transfer was successful

Return type
bool

Raises
Exception – raised if *from_address* or *to_address* length is not 20 or if *amount* is less than zero.

ledger

class Ledger

Bases: object

A class used to represent the Ledger native contract.

Check out Neo's Documentation to learn more about the Ledger class.

hash: *UInt160*

classmethod get_block(*index_or_hash: Union[int,* UInt256*]*) → Optional[*Block*]

Gets the block with the given index or hash.

```
>>> Ledger.get_block(0)        # first block
{
    'hash': b"S{\xed'\x85&\xf5\x93U=\xc1\xbf'\x95\xc4/\x80X\xdb\xd5\xa1-\x97q\
↪x85\xe3I\xe5\x99cd\x04",
    'version': 0,
    'previous_hash': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    'merkle_root': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
```

(continues on next page)

(continued from previous page)

```
    'timestamp': 1468595301000,
    'nonce': 2083236893,
    'index': 0,
    'primary_index': 0,
    'next_consensus': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\
↪xa5J\x00\x12\x04',
    'transaction_count': 0,
}
```

```
>>> Ledger.get_block(UInt256(b"S{\xed'\x85&\xf5\x93U=\xc1\xbf'\x95\xc4/\x80X\
↪\xdb\xd5\xa1-\x97q\x85\xe3I\xe5\x99cd\x04"))        # first block
{
    'hash': b"S{\xed'\x85&\xf5\x93U=\xc1\xbf'\x95\xc4/\x80X\xdb\xd5\xa1-\x97q\
↪x85\xe3I\xe5\x99cd\x04",
    'version': 0,
    'previous_hash': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    'merkle_root': '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
↪x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00',
    'timestamp': 1468595301000,
    'nonce': 2083236893,
    'index': 0,
    'primary_index': 0,
    'next_consensus': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\
↪xa5J\x00\x12\x04',
    'transaction_count': 0,
}
```

```
>>> Ledger.get_block(9999999)      # block doesn't exist
None
```

```
>>> Ledger.get_block(UInt256(bytes(32)))   # block doesn't exist
None
```

Parameters
index_or_hash (*int or* UInt256) – index or hash identifier of the block

Returns
the desired block, if exists. None otherwise

Return type
Block or None

classmethod get_current_index() → int

Gets the index of the current block.

```
>>> Ledger.get_current_index()
10908937
```

```
>>> Ledger.get_current_index()
2108690
```

```
>>> Ledger.get_current_index()
3529755
```

Returns
the index of the current block

Return type
int

classmethod get_transaction(*hash_:* UInt256) → Optional[*Transaction*]

Gets a transaction with the given hash.

```
>>> Ledger.get_transaction(UInt256(b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\
↪xdf\xc8\xb0\x9f>\xd0\xd2G\xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr'))
{
    'hash': b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\xdf\xc8\xb0\x9f>\xd0\
↪xd2G\xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr',
    'version': 0,
    'nonce': 2025056010,
    'sender': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\x00\
↪x12\x04',
    'system_fee': 2028330,
    'network_fee': 1206580,
    'valid_until_block': 5761,
    'script': b'\x0c\x14\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\
↪xa5J\x00\x12\x04\x11\xc0\x1f\x0c\tbalanceOf\x0c\x14\xcfv\xe2\x8b\xd0\x06,JG\
↪x8e\xe3Ua\x01\x13\x19\xf3\xcf\xa4\xd2Ab}[R',
}
```

```
>>> Ledger.get_transaction(UInt256(bytes(32)))   # transaction doesn't exist
None
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
the Transaction, if exists. None otherwise

classmethod get_transaction_from_block(*block_hash_or_height: Union[*UInt256*, int]*, *tx_index: int*) → Optional[*Transaction*]

Gets a transaction from a block.

```
>>> Ledger.get_transaction_from_block(1, 0)
{
    'hash': b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\xdf\xc8\xb0\x9f>\xd0\
↪xd2G\xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr',
    'version': 0,
    'nonce': 2025056010,
    'sender': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\x00\
↪x12\x04',
    'system_fee': 2028330,
    'network_fee': 1206580,
```

(continues on next page)

(continued from previous page)

```
    'valid_until_block': 5761,
    'script': b'\x0c\x14\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\
↪xa5J\x00\x12\x04\x11\xc0\x1f\x0c\tbalanceOf\x0c\x14\xcfv\xe2\x8b\xd0\x06,JG\
↪x8e\xe3Ua\x01\x13\x19\xf3\xcf\xa4\xd2Ab}[R',
}
```

```
>>> Ledger.get_transaction_from_block(UInt256(b'\x21|\xc2~U\t\x89^\x0c\xc0\
↪xd29wl\x0b\xad d\xe1\xf5U\xd7\xf5B\xa5/\x8b\x8f\x8b\x22\x24\x80'), 0)
{
    'hash': b'\xff\x7f\x18\x99\x8c\x1d\x10X{bA\xc2\xe3\xdf\xc8\xb0\x9f>\xd0\
↪xd2G\xe3\xba\xd8\x96\xb9\x0e\xc1iS\xcdr',
    'version': 0,
    'nonce': 2025056010,
    'sender': b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\xa5J\x00\
↪x12\x04',
    'system_fee': 2028330,
    'network_fee': 1206580,
    'valid_until_block': 5761,
    'script': b'\x0c\x14\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\
↪xa5J\x00\x12\x04\x11\xc0\x1f\x0c\tbalanceOf\x0c\x14\xcfv\xe2\x8b\xd0\x06,JG\
↪x8e\xe3Ua\x01\x13\x19\xf3\xcf\xa4\xd2Ab}[R',
}
```

```
>>> Ledger.get_transaction_from_block(123456789, 0)     # height does not exist␣
↪yet
None
```

```
>>> Ledger.get_transaction_from_block(UInt256(bytes(32)), 0)     # block hash␣
↪does not exist
None
```

Parameters

- **block_hash_or_height** (UInt256 *or* *int*) – a block identifier
- **tx_index** (*int*) – the transaction identifier in the block

Returns

the Transaction, if exists. None otherwise

classmethod get_transaction_height(*hash_:* UInt256) → int

Gets the height of a transaction.

```
>>> Ledger.get_transaction_height(UInt256(b'\x28\x89\x4f\xb6\x10\x62\x9d\xea\
↪x4c\xcd\x00\x2e\x9e\x11\xa6\xd0\x3d\x28\x90\xc0\xe5\xd4\xfc\x8f\xc6\x4f\xcc\
↪x32\x53\xb5\x48\x01'))
2108703
```

```
>>> Ledger.get_transaction_height(UInt256(b'\x29\x41\x06\xdb\x4c\xf3\x84\xa7\
↪x20\x4d\xba\x0a\x04\x03\x72\xb3\x27\x76\xf2\x6e\xd3\x87\x49\x88\xd0\x3e\xff\
↪x5d\xa9\x93\x8c\xa3'))
10
```

```
>>> Ledger.get_transaction_height(UInt256(bytes(32)))   # transaction doesn't␣
↪exist
-1
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
height of the transaction

classmethod get_transaction_signers(*hash_:* UInt256) → List[*Signer*]

Gets the VM state of a transaction.

```
>>> Ledger.get_transaction_signers(UInt256(b'\x29\x41\x06\xdb\x4c\xf3\x84\xa7\
↪\x20\x4d\xba\x0a\x04\x03\x72\xb3\x27\x76\xf2\x6e\xd3\x87\x49\x88\xd0\x3e\xff\
↪\x5d\xa9\x93\x8c\xa3'))
[
    {
        "account": b'\xa6\xea\xb0\xae\xaf\xb4\x96\xa1\x1b\xb0|\x88\x17\xcar\
↪xa5J\x00\x12\x04',
        "scopes": 1,
        "allowed_contracts": [],
        "allowed_groups": [],
        "rules": [],
    },
]
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
VM state of the transaction

classmethod get_transaction_vm_state(*hash_:* UInt256) → *VMState*

Gets the VM state of a transaction.

```
>>> Ledger.get_transaction_vm_state(UInt256(b'\x29\x41\x06\xdb\x4c\xf3\x84\xa7\
↪\x20\x4d\xba\x0a\x04\x03\x72\xb3\x27\x76\xf2\x6e\xd3\x87\x49\x88\xd0\x3e\xff\
↪\x5d\xa9\x93\x8c\xa3'))
VMState.HALT
```

Parameters
hash (UInt256) – hash identifier of the transaction

Returns
VM state of the transaction

neo

class NEO

Bases: object

A class used to represent the NEO native contract.

Check out Neo's Documentation to learn more about the NEO class.

hash: *UInt160*

classmethod symbol() → str

Gets the symbol of NEO.

```
>>> NEO.symbol()
'NEO'
```

Returns
the NEO string.

Return type
str

classmethod decimals() → int

Gets the amount of decimals used by NEO.

```
>>> NEO.decimals()
0
```

Returns
the number 0.

Return type
int

classmethod totalSupply() → int

Gets the total token supply deployed in the system.

```
>>> NEO.totalSupply()
100000000
```

Returns
the total token supply deployed in the system.

Return type
int

classmethod balanceOf(*account:* UInt160) → int

Get the current balance of an address.

```
>>> NEO.balanceOf(UInt160(bytes(20)))
0
```

```
>>> NEO.balanceOf(UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'))
100
```

Parameters
account (UInt160) – the account's address to retrieve the balance for

Returns
the account's balance

Return type
int

classmethod transfer(*from_address:* UInt160, *to_address:* UInt160, *amount:* *int*, *data:* *Optional[Any]* = *None*) → bool

Transfers an amount of GAS from one account to another.

If the method succeeds, it will fire the *Transfer* event and must return true, even if the amount is 0, or from and to are the same address.

```
>>> NEO.transfer(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6   #\xe1\x9096\
↪x89U\xac\x13'),     # this script hash needs to have signed the transaction␣
↪or block
...               UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'),
...               10, None)
True
```

```
>>> NEO.transfer(UInt160(bytes(20)),
...               UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'),
...               10, None)
False
```

```
>>> NEO.transfer(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6   #\xe1\x9096\
↪x89U\xac\x13'),
...               UInt160(b'\xabv\xe2\xcb\xb0\x16,vG\x2f\x44Va\x10\x14\x19\xf3\
↪xff\xa1\xe6'),
...               -1, None)
False
```

Parameters

- **from_address** (UInt160) – the address to transfer from
- **to_address** (UInt160) – the address to transfer to
- **amount** (*int*) – the amount of NEO to transfer
- **data** (*Any*) – whatever data is pertinent to the onNEP17Payment method

Returns
whether the transfer was successful

Return type
bool

Raises
Exception – raised if *from_address* or *to_address* length is not 20 or if *amount* is less than zero.

classmethod get_gas_per_block() → int

Gets the amount of GAS generated in each block.

```
>>> NEO.get_gas_per_block()
500000000
```

Returns
the amount of GAS generated

Return type
int

classmethod unclaimed_gas(*account:* UInt160, *end: int*) → int

Gets the amount of unclaimed GAS in the specified account.

```
>>> NEO.unclaimed_gas(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6      #\xe1\
↪x9096\x89U\xac\x13'), 0)
100000000
```

```
>>> NEO.unclaimed_gas(UInt160(bytes(20), 0)
100000000
```

Parameters
- **account** (UInt160) – the account to check
- **end** (*int*) – the block index used when calculating GAS

classmethod register_candidate(*pubkey:* ECPoint) → bool

Registers as a candidate.

```
>>> NEO.register_candidate(ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\x06\
↪xb4\x36\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\
↪xb3\xa2\xb6\x1a'))
False
```

Parameters
pubkey (ECPoint) – The public key of the account to be registered

Returns
whether the registration was a success or not

Return type
bool

classmethod unregister_candidate(*pubkey:* ECPoint) → bool

Unregisters as a candidate.

```
>>> NEO.unregister_candidate(ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\x06\
↪\xb4\x36\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\
↪\xb3\xa2\xb6\x1a'))
False
```

Parameters
pubkey (ECPoint) – The public key of the account to be unregistered

Returns
whether the unregistration was a success or not

Return type
bool

classmethod vote(*account:* UInt160, *vote_to:* ECPoint) → bool

Votes for a candidate.

```
>>> NEO.vote(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6       #\xe1\x9096\
↪\x89U\xac\x13'), ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\x06\xb4\x36\
↪\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\xb3\xa2\
↪\xb6\x1a'))
False
```

Parameters

- **account** (UInt160) – the account that is voting
- **vote_to** (ECPoint) – the public key of the one being voted

classmethod get_all_candidates() → *Iterator*

Gets the registered candidates iterator.

```
>>> NEO.get_all_candidates()
[]
```

Returns
all registered candidates

Return type
Iterator

classmethod un_vote(*account:* UInt160) → bool

Removes the vote of the candidate voted. It would be the same as calling vote(account, None).

```
>>> NEO.un_vote(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6     #\xe1\x9096\
↪\x89U\xac\x13'))
False
```

Parameters
account (UInt160) – the account that is removing the vote

classmethod get_candidates() → List[Tuple[*ECPoint*, int]]

Gets the list of all registered candidates.

```
>>> NEO.get_candidates()
[]
```

Returns
all registered candidates

Return type
List[Tuple[*ECPoint*, int]]

classmethod get_candidate_vote(*pubkey:* ECPoint) → int

Gets votes from specific candidate.

```
>>> NEO.get_candidate_vote(ECPoint(b'\x03\x5a\x92\x8f\x20\x16\x39\x20\x4e\x06\
↪xb4\x36\x8b\x1a\x93\x36\x54\x62\xa8\xeb\xbf\xf0\xb8\x81\x81\x51\xb7\x4f\xaa\
↪xb3\xa2\xb6\x1a'))
100
```

```
>>> NEO.get_candidate_vote(ECPoint(bytes(32)))
-1
```

Returns
Votes or -1 if it was not found.

Return type
int

classmethod get_committee() → List[*ECPoint*]

Gets all committee members list.

```
>>> NEO.get_committee()
[ b'\x02|\x84\xb0V\xc2j{$XG\x1em\xcfgR\xed\xd9k\x96\x88}x34\xe3Q\xdd\xfe\x13\
↪xc4\xbc\xa2' ]
```

Returns
all committee members

Return type
List[*ECPoint*]

classmethod get_next_block_validators() → List[*ECPoint*]

Gets validators list of the next block.

```
>>> NEO.get_next_block_validators()
[ b'\x02|\x84\xb0V\xc2j{$XG\x1em\xcfgR\xed\xd9k\x96\x88}x34\xe3Q\xdd\xfe\x13\
↪xc4\xbc\xa2' ]
```

Returns
the public keys of the validators

Return type
List[*ECPoint*]

classmethod get_account_state(*account:* UInt160) → *NeoAccountState*

Gets the latest votes of the specified account.

```
>>> NEO.get_account_state(UInt160(b'\xc9F\x17\xba!\x99\x07\xc1\xc5\xd6  #\xe1\
↪x9096\x89U\xac\x13'))
{
    'balance': 100,
    'height': 2,
    'vote_to': None,
}
```

Parameters

account (UInt160) – the specified account

Returns

the state of the account

Return type

NeoAccountState

oracle

class Oracle

Bases: object

Neo Oracle Service is an out-of-chain data access service built into Neo N3. It allows users to request the external data sources in smart contracts, and Oracle nodes designated by the committee will access the specified data source then pass the result in the callback function to continue executing the smart contract logic.

Check out Neo's Documentation to learn more about Oracles.

hash: *UInt160*

classmethod request(*url: str*, *request_filter: Optional[str]*, *callback: str*, *user_data: Any*, *gas_for_response: int*)

Requests an information from outside the blockchain.

This method just requests data from the oracle, it won't return the result.

```
>>> Oracle.request('https://dora.coz.io/api/v1/neo3/testnet/asset/
↪0xef4073a0f2b305a38ec4050e4d3d28bc40ea63f5',
...                 '', 'callback_name', None, 10 * 10 ** 8)
None
```

Parameters

- **url** (*str*) – External url to retrieve the data
- **request_filter** (*str or None*) – Filter to the request.

  See JSONPath format https://github.com/atifaziz/JSONPath
- **callback** (*str*) – Method name that will be as a callback.

  This method must be public and implement the following interface:

  `(url: str, user_data: Any, code: int, result: bytes) -> None`

- **user_data** (*Any*) – Optional data. It'll be returned as the same when the callback is called
- **gas_for_response** (*int*) – Amount of GAS needed to run the callback method.

  It MUST NOT be specified as the user representation.

  If it costs 1 gas, this value must be 1_00000000 (with the 8 decimals)

classmethod get_price() → int

Gets the price for an Oracle request.

```
>>> Oracle.get_price()
50000000
```

Returns
the price for an Oracle request

policy

class Policy

Bases: object

A class used to represent the Policy native contract.

Check out Neo's Documentation to learn more about the Policy class.

hash: *UInt160*

classmethod get_fee_per_byte() → int

Gets the network fee per transaction byte.

```
>>> Policy.get_fee_per_byte()
1000
```

Returns
the network fee per transaction byte

Return type
int

classmethod get_exec_fee_factor() → int

Gets the execution fee factor. This is a multiplier that can be adjusted by the committee to adjust the system fees for transactions.

```
>>> Policy.get_exec_fee_factor()
30
```

Returns
the execution fee factor

Return type
int

classmethod get_storage_price() → int

Gets the storage price.

```
>>> Policy.get_storage_price()
100000
```

Returns
the snapshot used to read data

Return type
int

classmethod is_blocked(*account:* UInt160) → bool

Determines whether the specified account is blocked.

```
>>> Policy.is_blocked(UInt160(b'\xcfv\xe2\x8b\xd0\x06,JG\x8e\xe3Ua\x01\x13\x19\
↪xf3\xcf\xa4\xd2'))
False
```

Parameters
account (UInt160) – the account to be checked

Returns
whether the account is blocked or not

Return type
bool

rolemanagement

class RoleManagement

Bases: object

A class used to represent the RoleManagement native contract.

Check out Neo's Documentation to learn more about the RoleManagement class.

hash: *UInt160*

classmethod get_designated_by_role(*role:* Role, *index: int*) → *ECPoint*

Gets the list of nodes for the specified role.

```
>>> RoleManagement.get_designated_by_role(Role.ORACLE, 0)
[]
```

Parameters

- **role** (Role) – the type of the role
- **index** (*int*) – the index of the block to be queried

Returns
the public keys of the nodes

Return type
ECPoint

class Role(*value*)

Bases: IntFlag

Represents the roles in the NEO system. They are the permission types of the native contract *RoleManagement*.

STATE_VALIDATOR

The validators of state. Used to generate and sign the state root.

ORACLE

The nodes used to process Oracle requests.

NEO_FS_ALPHABET_NODE

NeoFS Alphabet nodes.

stdlib

class StdLib

Bases: object

A class used to represent StdLib native contract.

Check out Neo's Documentation to learn more about the StdLib class.

hash: *UInt160*

classmethod **serialize**(*item: Any*) → bytes

Serializes the given value into its bytes representation.

```
>>> StdLib.serialize('42')
b'(\x0242'
```

```
>>> StdLib.serialize(42)
b'!\x01*'
```

```
>>> StdLib.serialize([2, 3, 5, 7])
b'@\x04!\x01\x02!\x01\x03!\x01\x05!\x01\x07'
```

```
>>> StdLib.serialize({1: 1, 2: 1, 3: 2})
b'H\x03!\x01\x01!\x01\x01!\x01\x02!\x01\x01!\x01\x03!\x01\x02'
```

Parameters
item (*Any*) – value to be serialized

Returns
the serialized value

Return type
bytes

Raises
Exception – raised if the item's type is not serializable.

classmethod deserialize(*data: bytes*) → Any

Deserializes the given bytes value.

```
>>> StdLib.deserialize(b'(\x0242')
'42'
```

```
>>> StdLib.deserialize(b'!\x01*')
42
```

```
>>> StdLib.deserialize(b'@\x04!\x01\x02!\x01\x03!\x01\x05!\x01\x07')
[2, 3, 5, 7]
```

```
>>> StdLib.deserialize(b'H\x03!\x01\x01!\x01\x01!\x01\x02!\x01\x01!\x01\x03!\
↪x01\x02')
{1: 1, 2: 1, 3: 2}
```

Parameters
data (*bytes*) – serialized value

Returns
the deserialized result

Return type
Any

Raises
Exception – raised when the date doesn't represent a serialized value.

classmethod json_serialize(*item: Any*) → str

Serializes an item into a json.

```
>>> StdLib.json_serialize({'one': 1, 'two': 2, 'three': 3})
'{"one":1,"two":2,"three":3}'
```

Parameters
item (*Any*) – The item that will be serialized

Returns
The serialized item

Return type
str

Raises
Exception – raised if the item is an integer value out of the Neo's accepted range, is a dictionary with a bytearray key, or isn't serializable.

classmethod json_deserialize(*json: str*) → Any

Deserializes a json into some valid type.

```
>>> StdLib.json_deserialize('{"one":1,"two":2,"three":3}')
{'one': 1, 'three': 3, 'two': 2}
```

Parameters
json (*str*) – A json that will be deserialized

Returns
The deserialized json

Return type
Any

Raises
Exception – raised if jsons deserialization is not valid.

classmethod base64_decode(*key: str*) → bytes

Decodes a string value encoded with base64.

```
>>> StdLib.base64_decode("dW5pdCB0ZXN0")
b"unit test"
```

Parameters
key (*str*) – string value to be decoded

Returns
the decoded string

Return type
bytes

classmethod base64_encode(*key: bytes*) → str

Encodes a bytes value using base64.

```
>>> StdLib.base64_encode(b'unit test')
b"dW5pdCB0ZXN0"
```

Parameters
key (*bytes*) – bytes value to be encoded

Returns
the encoded string

Return type
str

classmethod base58_decode(*key: str*) → bytes

Decodes a string value encoded with base58.

```
>>> StdLib.base58_decode('2VhL46g69A1mu')
b"unit test"
```

Parameters
key (*str*) – string value to be decoded

Returns
the decoded bytes

Return type
bytes

classmethod base58_encode(*key: bytes*) → str

Encodes a bytes value using base58.

```
>>> StdLib.base58_encode(b'unit test')
b"2VhL46g69A1mu"
```

Parameters
key (*bytes*) – bytes value to be encoded

Returns
the encoded string

Return type
str

classmethod base58_check_decode(*key: str*) → bytes

Converts the specified str, which encodes binary data as base-58 digits, to an equivalent bytes value. The encoded str contains the checksum of the binary data.

```
>>> StdLib.base58_check_decode('AnJcKqvgBwKxsjX75o')
b"unit test"
```

Parameters
key (*str*) – string value to be decoded

Returns
the decoded bytes

Return type
bytes

classmethod base58_check_encode(*key: bytes*) → str

Converts a bytes value to its equivalent str representation that is encoded with base-58 digits. The encoded str contains the checksum of the binary data.

```
>>> StdLib.base58_check_encode(b'unit test')
b"AnJcKqvgBwKxsjX75o"
```

Parameters
key (*bytes*) – bytes value to be encoded

Returns
the encoded string

Return type
str

classmethod itoa(*value: int*, *base: int = 10*) → str

Converts the specific type of value to a decimal or hexadecimal string. The default is decimal.

```
>>> StdLib.itoa(10)
'10'
```

```
>>> StdLib.itoa(123)
'123'
```

```
>>> StdLib.itoa(-1, 16)
'f'
```

```
>>> StdLib.itoa(15, 16)
'0f'
```

Parameters

- **value** (*int*) – the int value
- **base** (*int*) – the value's base

Returns
the converted string

Return type
int

classmethod atoi(*value: str*, *base: int = 10*) → int

Converts a character string to a specific base value, decimal or hexadecimal. The default is decimal.

```
>>> StdLib.atoi('10')
10
```

```
>>> StdLib.atoi('123')
123
```

```
>>> StdLib.atoi('1f', 16)
31
```

```
>>> StdLib.atoi('ff', 16)
-1
```

Parameters

- **value** (*str*) – the int value as a string
- **base** (*int*) – the value base

Returns
the equivalent value

Return type
int

Raises
Exception – raised when base isn't 10 or 16.

classmethod memory_compare(*mem1: Union[bytes, str]*, *mem2: Union[bytes, str]*) → int

Compares a memory with another one.

```
>>> StdLib.memory_compare('abc', 'abc')
0
```

```
>>> StdLib.memory_compare('ABC', 'abc')
-1
```

```
>>> StdLib.memory_compare('abc', 'ABC')
1
```

Parameters

- **mem1** (*bytes or str*) – a memory to be compared to another one
- **mem2** (*bytes or str*) – a memory that will be compared with another one

Returns
-1 if mem1 precedes mem2, 0 if mem1 and mem2 are equal, 1 if mem1 follows mem2

Return type
int

classmethod memory_search(*mem: Union[bytes, str]*, *value: Union[bytes, str]*, *start: int = 0*, *backward: bool = False*) → int

Searches for a given value in a given memory.

```
>>> StdLib.memory_search('abcde', 'a', 0)
0
```

```
>>> StdLib.memory_search('abcde', 'e', 0)
4
```

Parameters

- **mem** (*bytes or str*) – the memory
- **value** (*bytes or str*) – the value
- **start** (*int*) – the index the search should start from
- **backward** (*bool*) – whether it should invert the memory

Returns
the index of the value in the memory. Returns -1 if it's not found

Return type
int

type

class Event

Bases: object

Describes an action that happened in the blockchain. Neo3-Boa compiler won't recognize the *__init__* of this class. To create a new Event, use the method *CreateNewEvent*:

Check out Neo's Documentation to learn more about Events.

```
>>> from boa3.builtin.compile_time import CreateNewEvent
... new_event: Event = CreateNewEvent(  # create a new Event with the␣
↪CreateNewEvent method
...     [
...         ('name', str),
...         ('amount', int)
...     ],
...     'New Event'
... )
```

class UInt160(*arg: Union[bytes, int] = 0*)

Bases: bytes

Represents a 160-bit unsigned integer.

class UInt256(*arg: Union[bytes, int] = 0*)

Bases: bytes

Represents a 256-bit unsigned integer.

class ECPoint(*arg: bytes*)

Bases: bytes

Represents a coordinate pair for elliptic curve cryptography (ECC) structures.

to_script_hash() → bytes

Converts a data to a script hash.

Returns
the script hash of the data

Return type
bytes

Address

A type used only to indicate that a parameter or return on the manifest should be treated as an Address. Same as str.

BlockHash

A type used only to indicate that a parameter or return on the manifest should be treated as a BlockHash. Same as UInt256.

PublicKey

A type used only to indicate that a parameter or return on the manifest should be treated as a PublicKey. Same as ECPoint.

ScriptHash

A type used only to indicate that a parameter or return on the manifest should be treated as a ScriptHash. Same as UInt160.

ScriptHashLittleEndian

A type used only to indicate that a parameter or return on the manifest should be treated as a ScriptHashLittleEndian. Same as UInt160.

TransactionId

A type used only to indicate that a parameter or return on the manifest should be treated as a TransactionId. Same as UInt256.

Subpackages

helper

to_bool(*value: bytes*) → bool

Return a bytes value to the boolean it represents.

```
>>> to_bool(b'\x00')
False
```

```
>>> to_bool(b'\x01')
True
```

```
>>> to_bool(b'\x02')
True
```

to_bytes(*value: Union[str, int]*) → bytes

Converts a str or integer value to an array of bytes

```
>>> to_bytes(65)
b'A'
```

```
>>> to_bytes('A')
b'A'
```

to_int(*value: bytes*) → int

Converts a bytes value to the integer it represents.

```
>>> to_int(b'A')
65
```

to_str(*value: bytes*) → str

Converts a bytes value to a string.

```
>>> to_str(b'A')
'A'
```

vm

class Opcode(*value*)

Bases: bytes, Enum

Opcodes are similar to instructions in assembly language.

PUSHINT8

Pushes a 1-byte signed integer onto the stack.

PUSHINT16

Pushes a 2-byte signed integer onto the stack.

PUSHINT32

Pushes a 4-byte signed integer onto the stack.

PUSHINT64

Pushes a 8-byte signed integer onto the stack.

PUSHINT128

Pushes a 16-byte signed integer onto the stack.

PUSHINT256

Pushes a 32-byte signed integer onto the stack.

PUSHT

Pushes the boolean value True onto the stack.

PUSHF

Pushes the boolean value False onto the stack.

PUSHA

Converts the 4-bytes offset to a Pointer, and pushes it onto the stack.

PUSHNULL

The item null is pushed onto the stack.

PUSHDATA1

The next byte contains the number of bytes to be pushed onto the stack.

PUSHDATA2

The next two bytes contain the number of bytes to be pushed onto the stack.

PUSHDATA4

The next four bytes contain the number of bytes to be pushed onto the stack.

PUSHM1

The number -1 is pushed onto the stack.

PUSH0

The number 0 is pushed onto the stack.

PUSH1

The number 1 is pushed onto the stack.

PUSH2

The number 2 is pushed onto the stack.

PUSH3

The number 3 is pushed onto the stack.

PUSH4

The number 4 is pushed onto the stack.

PUSH5

The number 5 is pushed onto the stack.

PUSH6

The number 6 is pushed onto the stack.

PUSH7

The number 7 is pushed onto the stack.

PUSH8

The number 8 is pushed onto the stack.

PUSH9

The number 9 is pushed onto the stack.

PUSH10

The number 10 is pushed onto the stack.

PUSH11

The number 11 is pushed onto the stack.

PUSH12

The number 12 is pushed onto the stack.

PUSH13

The number 13 is pushed onto the stack.

PUSH14

The number 14 is pushed onto the stack.

PUSH15

The number 15 is pushed onto the stack.

PUSH16

The number 16 is pushed onto the stack.

NOP

The NOP operation does nothing. It is intended to fill in space if opcodes are patched.

JMP

Unconditionally transfers control to a target instruction. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMP_L

Unconditionally transfers control to a target instruction. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPIF

Transfers control to a target instruction if the value is True, not null, or non-zero. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPIF_L

Transfers control to a target instruction if the value is True, not null, or non-zero. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPIFNOT

Transfers control to a target instruction if the value is False, a null reference, or zero. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPIFNOT_L

Transfers control to a target instruction if the value is False, a null reference, or zero. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPEQ

Transfers control to a target instruction if two values are equal. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPEQ_L

Transfers control to a target instruction if two values are equal. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPNE

Transfers control to a target instruction when two values are not equal. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPNE_L

Transfers control to a target instruction when two values are not equal. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPGT

Transfers control to a target instruction if the first value is greater than the second value. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPGT_L

Transfers control to a target instruction if the first value is greater than the second value. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPGE

Transfers control to a target instruction if the first value is greater than or equal to the second value. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPGE_L

Transfers control to a target instruction if the first value is greater than or equal to the second value. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPLT

Transfers control to a target instruction if the first value is less than the second value. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPLT_L

Transfers control to a target instruction if the first value is less than the second value. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

JMPLE

Transfers control to a target instruction if the first value is less than or equal to the second value. The target instruction is represented as a 1-byte signed offset from the beginning of the current instruction.

JMPLE_L

Transfers control to a target instruction if the first value is less than or equal to the second value. The target instruction is represented as a 4-bytes signed offset from the beginning of the current instruction.

CALL

Calls the function at the target address which is represented as a 1-byte signed offset from the beginning of the current instruction.

CALL_L

Calls the function at the target address which is represented as a 4-bytes signed offset from the beginning of the current instruction.

CALLA

Pop the address of a function from the stack, and call the function.

CALLT

Calls the function which is described by the token.

ABORT

It turns the vm state to FAULT immediately, and cannot be caught.

ASSERT

Pop the top value of the stack, if it false, then exit vm execution and set vm state to FAULT.

THROW

Pop the top value of the stack, and throw it.

TRY

TRY CatchOffset(sbyte) FinallyOffset(sbyte). If there's no catch body, set CatchOffset 0. If there's no finally body, set FinallyOffset 0.

TRY_L

TRY_L CatchOffset(int) FinallyOffset(int). If there's no catch body, set CatchOffset 0. If there's no finally body, set FinallyOffset 0.

ENDTRY

Ensures that the appropriate surrounding finally blocks are executed. And then unconditionally transfers control to the specific target instruction, represented as a 1-byte signed offset from the beginning of the current instruction.

ENDTRY_L

Ensures that the appropriate surrounding finally blocks are executed. And then unconditionally transfers control to the specific target instruction, represented as a 4-byte signed offset from the beginning of the current instruction.

ENDFINALLY

End finally, If no exception happen or be catched, vm will jump to the target instruction of ENDTRY/ENDTRY_L. Otherwise vm will rethrow the exception to upper layer.

RET

Returns from the current method.

SYSCALL

Calls to an interop service.

DEPTH

Puts the number of stack items onto the stack.

DROP

Removes the top stack item.

NIP

Removes the second-to-top stack item.

XDROP

The item n back in the main stack is removed.

CLEAR

Clear the stack

DUP

Duplicates the top stack item.

OVER

Copies the second-to-top stack item to the top.

PICK

The item n back in the stack is copied to the top.

TUCK

The item at the top of the stack is copied and inserted before the second-to-top item.

SWAP

The top two items on the stack are swapped.

ROT

The top three items on the stack are rotated to the left.

ROLL

The item n back in the stack is moved to the top.

REVERSE3

Reverse the order of the top 3 items on the stack.

REVERSE4

Reverse the order of the top 4 items on the stack.

REVERSEN

Pop the number N on the stack, and reverse the order of the top N items on the stack.

INITSSLOT

Initialize the static field list for the current execution context.

INITSLOT

Initialize the argument slot and the local variable list for the current execution context.

LDSFLD0

Loads the static field at index 0 onto the evaluation stack.

LDSFLD1

Loads the static field at index 1 onto the evaluation stack.

LDSFLD2

Loads the static field at index 2 onto the evaluation stack.

LDSFLD3

Loads the static field at index 3 onto the evaluation stack.

LDSFLD4

Loads the static field at index 4 onto the evaluation stack.

LDSFLD5

Loads the static field at index 5 onto the evaluation stack.

LDSFLD6

Loads the static field at index 6 onto the evaluation stack.

LDSFLD

Loads the static field at a specified index onto the evaluation stack. The index is represented as a 1-byte unsigned integer.

STSFLD0

Stores the value on top of the evaluation stack in the static field list at index 0.

STSFLD1

Stores the value on top of the evaluation stack in the static field list at index 1.

STSFLD2

Stores the value on top of the evaluation stack in the static field list at index 2.

STSFLD3

Stores the value on top of the evaluation stack in the static field list at index 3.

STSFLD4

Stores the value on top of the evaluation stack in the static field list at index 4.

STSFLD5

Stores the value on top of the evaluation stack in the static field list at index 5.

STSFLD6

Stores the value on top of the evaluation stack in the static field list at index 6.

STSFLD

Stores the value on top of the evaluation stack in the static field list at a specified index. The index is represented as a 1-byte unsigned integer.

LDLOC0

Loads the local variable at index 0 onto the evaluation stack.

LDLOC1

Loads the local variable at index 1 onto the evaluation stack.

LDLOC2

Loads the local variable at index 2 onto the evaluation stack.

LDLOC3

Loads the local variable at index 3 onto the evaluation stack.

LDLOC4

Loads the local variable at index 4 onto the evaluation stack.

LDLOC5

Loads the local variable at index 5 onto the evaluation stack.

LDLOC6

Loads the local variable at index 6 onto the evaluation stack.

LDLOC

Loads the local variable at a specified index onto the evaluation stack. The index is represented as a 1-byte unsigned integer.

STLOC0

Stores the value on top of the evaluation stack in the local variable list at index 0.

STLOC1

Stores the value on top of the evaluation stack in the local variable list at index 1.

STLOC2

Stores the value on top of the evaluation stack in the local variable list at index 2.

STLOC3

Stores the value on top of the evaluation stack in the local variable list at index 3.

STLOC4

Stores the value on top of the evaluation stack in the local variable list at index 4.

STLOC5

Stores the value on top of the evaluation stack in the local variable list at index 5.

STLOC6

Stores the value on top of the evaluation stack in the local variable list at index 6.

STLOC

Stores the value on top of the evaluation stack in the local variable list at a specified index. The index is represented as a 1-byte unsigned integer.

LDARG0

Loads the argument at index 0 onto the evaluation stack.

LDARG1

Loads the argument at index 1 onto the evaluation stack.

LDARG2

Loads the argument at index 2 onto the evaluation stack.

LDARG3

Loads the argument at index 3 onto the evaluation stack.

LDARG4

Loads the argument at index 4 onto the evaluation stack.

LDARG5

Loads the argument at index 5 onto the evaluation stack.

LDARG6

Loads the argument at index 6 onto the evaluation stack.

LDARG

Loads the argument at a specified index onto the evaluation stack. The index is represented as a 1-byte unsigned integer.

STARG0

Stores the value on top of the evaluation stack in the argument slot at index 0.

STARG1

Stores the value on top of the evaluation stack in the argument slot at index 1.

STARG2

Stores the value on top of the evaluation stack in the argument slot at index 2.

STARG3

Stores the value on top of the evaluation stack in the argument slot at index 3.

STARG4

Stores the value on top of the evaluation stack in the argument slot at index 4.

STARG5

Stores the value on top of the evaluation stack in the argument slot at index 5.

STARG6

Stores the value on top of the evaluation stack in the argument slot at index 6.

STARG

Stores the value on top of the evaluation stack in the argument slot at a specified index. The index is represented as a 1-byte unsigned integer.

NEWBUFFER

Creates a new Buffer and pushes it onto the stack.

MEMCPY

Copies a range of bytes from one Buffer to another.

CAT

Concatenates two strings.

SUBSTR

Returns a section of a string.

LEFT

Keeps only characters left of the specified point in a string.

RIGHT

Keeps only characters right of the specified point in a string.

INVERT

Flips all of the bits in the input.

AND

Boolean and between each bit in the inputs.

OR

Boolean or between each bit in the inputs.

XOR

Boolean exclusive or between each bit in the inputs.

EQUAL

Returns 1 if the inputs are exactly equal, 0 otherwise.

NOTEQUAL

Returns 1 if the inputs are not equal, 0 otherwise.

SIGN

Puts the sign of top stack item on top of the main stack. If value is negative, put -1; if positive, put 1; if value is zero, put 0.

ABS

The input is made positive.

NEGATE

The sign of the input is flipped.

INC

1 is added to the input.

DEC

1 is subtracted from the input.

ADD

a is added to b.

SUB

b is subtracted from a.

MUL

a is multiplied by b.

DIV

a is divided by b.

MOD

Returns the remainder after dividing a by b.

POW

The result of raising value to the exponent power.

SQRT

Returns the square root of a specified number.

MODMUL

Performs modulus division on a number multiplied by another number.

MODPOW

Performs modulus division on a number raised to the power of another number. If the exponent is -1, it will have the calculation of the modular inverse.

SHL

Shifts a left b bits, preserving sign.

SHR

Shifts a right b bits, preserving sign.

NOT

If the input is 0 or 1, it is flipped. Otherwise the output will be 0.

BOOLAND

If both a and b are not 0, the output is 1. Otherwise 0.

BOOLOR

If a or b is not 0, the output is 1. Otherwise 0.

NZ

Returns 0 if the input is 0. 1 otherwise.

NUMEQUAL

Returns 1 if the numbers are equal, 0 otherwise.

NUMNOTEQUAL

Returns 1 if the numbers are not equal, 0 otherwise.

LT

Returns 1 if a is less than b, 0 otherwise.

LE

Returns 1 if a is less than or equal to b, 0 otherwise.

GT

Returns 1 if a is greater than b, 0 otherwise.

GE

Returns 1 if a is greater than or equal to b, 0 otherwise.

MIN

Returns the smaller of a and b.

MAX

Returns the larger of a and b.

WITHIN

Returns 1 if x is within the specified range (left-inclusive), 0 otherwise.

PACKMAP

A value n is taken from top of main stack. The next n*2 items on main stack are removed, put inside n-sized map and this map is put on top of the main stack.

PACKSTRUCT

A value n is taken from top of main stack. The next n items on main stack are removed, put inside n-sized struct and this struct is put on top of the main stack.

PACK

A value n is taken from top of main stack. The next n items on main stack are removed, put inside n-sized array and this array is put on top of the main stack.

UNPACK

A collection is removed from top of the main stack. Its elements are put on top of the main stack (in reverse order) and the collection size is also put on main stack.

NEWARRAY0

An empty array (with size 0) is put on top of the main stack.

NEWARRAY

A value n is taken from top of main stack. A null-filled array with size n is put on top of the main stack.

NEWARRAY_T

A value n is taken from top of main stack. An array of type T with size n is put on top of the main stack.

NEWSTRUCT0

An empty struct (with size 0) is put on top of the main stack.

NEWSTRUCT

A value n is taken from top of main stack. A zero-filled struct with size n is put on top of the main stack.

NEWMAP

A Map is created and put on top of the main stack.

SIZE

An array is removed from top of the main stack. Its size is put on top of the main stack.

HASKEY

An input index n (or key) and an array (or map) are removed from the top of the main stack. Puts True on top of main stack if array[n] (or map[n]) exist, and False otherwise.

KEYS

A map is taken from top of the main stack. The keys of this map are put on top of the main stack.

VALUES

A map is taken from top of the main stack. The values of this map are put on top of the main stack.

PICKITEM

An input index n (or key) and an array (or map) are taken from main stack. Element array[n] (or map[n]) is put on top of the main stack.

APPEND

The item on top of main stack is removed and appended to the second item on top of the main stack.

SETITEM

A value v, index n (or key) and an array (or map) are taken from main stack. Attribution array[n]=v (or map[n]=v) is performed.

REVERSEITEMS

An array is removed from the top of the main stack and its elements are reversed.

REMOVE

An input index n (or key) and an array (or map) are removed from the top of the main stack. Element array[n] (or map[n]) is removed.

CLEARITEMS

Remove all the items from the compound-type.

POPITEM

Remove the last element from an array, and push it onto the stack.

ISNULL

Returns true if the input is null;

ISTYPE

Returns true if the top item of the stack is of the specified type;

CONVERT

Returns true if the input is null;

ABORTMSG

Turns the vm state to FAULT immediately, and cannot be caught. Includes a reason.

ASSERTMSG

Pop the top value of the stack, if it false, then exit vm execution and set vm state to FAULT. Includes a reason.

math

ceil(*x: int*, *decimals: int*) → int

Return the ceiling of x given the amount of decimals. This is the smallest integer >= x.

```
>>> ceil(12345, 3)
13000
```

Parameters

- **x** (*int*) – any integer number
- **decimals** (*int*) – number of decimals

Returns
the ceiling of x

Return type
int

Raises
Exception – raised when decimals is negative.

floor(*x: int*, *decimals: int*) → int

Return the floor of x given the amount of decimals. This is the largest integer <= x.

```
>>> floor(12345, 3)
12000
```

Parameters

- **x** (*int*) – any integer number
- **decimals** (*int*) – number of decimals

Returns
the floor of x

Return type
int

Raises
Exception – raised when decimals is negative.

sqrt(*x: int*) → int

Gets the square root of a number.

```
>>> sqrt(1)
1
```

```
>>> sqrt(10)
3
```

```
>>> sqrt(25)
5
```

Parameters
x (*int*) – a non-negative number

Returns
the square root of a number

Return type
int

Raises
Exception – raised when number is negative.

PYTHON MODULE INDEX

b

INDEX

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X