Solidbyte Documentation

Release 0.8.2b1

Mike Shultz

Mar 26, 2019

Contents

Solidbyte is a toolkit for writing Ethereum smart contracts.

CHAPTER 1

Quickstart

Here's the quickest way to get started.

```
#!/bin/sh
pip install solidbyte
sb init
```

You may want to setup a Python virtual environment and your system may require some installed dependencies. For full installation instructions, see *Install*.

1.1 Install

1.1.1 System Requirements

Some system level depenencies are required first for Solidbyte to work. Python >= 3.6 is required.

Linux

Ubuntu

```
apt install python3.6 libssl-dev libffi-dev
```

Arch Linux

```
pacman -S openssl libffi
```

NOTE: python3 should already be installed on your system.

REHL/CentOS

```
yum install openssl-devel libffi-devel
```

Windows

TBD. Please submit a pull request if you figure it out.

OSX

TBD. Please submit a pull request if you figure it out.

1.1.2 Installing Solidbyte

Install it with system python:

```
pip install --user solidbyte
```

Or, with a virtual environment:

```
python -m venv ~/virtualenvs/solidbyte
source ~/virtualenvs/solidbyte/bin/activate
pip install solidbyte
```

1.2 Commands

1.2.1 `init`

Create a project using a template or bare. For instance, creating an ERC20 project from the template:

```
sb init -t erc20
```

1.2.2 `compile`

Compile the contracts.

```
sb compile
```

1.2.3 `test`

Test the contracts using pytest(?)

```
sb test
```

1.2.4 `console`

Start a pythonic console for testing contracts. Provides web3 and contracts as local variables.

```
$ sb console dev
2018-10-28 17:42:38,022 [INFO] solidbyte.cli.console - Starting interactive console...
Solidbyte Console (0.0.1b1)
------------------------------
Network Chain ID: 1540751678531
Available deployed contracts: MyToken
Available locals: web3
>>>
```

1.2.5 `deploy`

Deploy contracts using the user-written deploy scripts. For more details, see *Deployment Scripts*.

1.2.6 `help`

Show usage

1.2.7 `show`

Show details about the deployed contracts

1.2.8 `version`

Show versions of solidbyte, the compiler, and associated tools

1.2.9 `script`

Execute a python script within the context of soidbyte

1.2.10 `install` [Prototype]

Ethereum package manager support. Coming soon…

1.2.11 `metafile`

Commands to backup and cleanup the metafile.

`metafile cleanup`

Cleanup and compact `metafile.json` by removing deployed contract instances for test networks.

`metafile backup`

Make a copy of `metafile.json` to the given location and verify.

1.2.12 `sigs`

Show all event and function signatures for the compiled contracts.

1.3 Deployment Scripts

1.3.1 Overview

Solidbyte aims to make deployment easy. For the most part, it will keep track of contract deployments and will know when the source changed and a new version needs to go up.

However, most deployments are not as simple as just compiling the bytecode and sending the TX. That have constructor arguments, or little transactions that need to be made after deployment is done. For this, you need to create a deployment script.

All scripts are in the `deploy/` directory in your project root, and should be named starting with `deploy_`. And Solidbyte will only call `main()` within your deploy scripts. Any other functions you have will be ignored.

For instance, if you initialized your project with an ERC20 template, you would get the following deployment script by default. It's got a little logic for funding your accounts on test network, setting the `initialSupply`, and verifying it after deployment.

```python
def main(contracts, deployer_account, web3, network):
    assert contracts is not None
    assert deployer_account is not None
    assert web3 is not None
    assert network is not None

    deployer_balance = web3.eth.getBalance(deployer_account)

    if network in ('dev', 'test'):
        # If this is the test network, make sure our deployment account is funded
        if deployer_balance == 0:
            tx = web3.eth.sendTransaction({
                'from': web3.eth.accounts[0],  # The pre-funded account in ganace-cli
                'to': deployer_account,
                'value': int(1e18),
                'gasPrice': int(3e9),
                })
            receipt = web3.eth.waitForTransactionReceipt(tx)
            assert receipt.status == 1
    else:
        # Make sure deployer account has at least 0.5 ether
        assert deployer_balance < int(5e17), "deployer account needs to be funded"

    # Get the sb Contract instance
    token = contracts.get('MyERC20')

    # Deploy (if necessary) and return the web3.eth.Contract instance
    initial_supply = int(1e21)
```

(continues on next page)

(continued from previous page)

```
    web3_contract_instance = token.deployed(initialSupply=initial_supply)

    # If we have an address, deployment was successful
    assert web3_contract_instance.address is not None, "Deploy failed.  No address␣
↪found"
    assert web3_contract_instance.functions.totalSupply().call() == initial_supply, \
        "totalSupply does not equal initialSupply"

    return True
```

The important bit is this:

```
    web3_contract_instance = token.deployed(initialSupply=initial_supply)
```

The `.deployed()` method on the *`solidbyte.deploy.objects.Contract`* instance is where the magic happens. This will trigger Solidbyte to deploy the contract if necessary. The arguments to this function are the same arguments you would provide to your contract's construtor. It will return a `web3.contract.Contract` instance.

NOTE: Using `Contract.deployed()` is not required. It's there to help. Feel free not to use it.

Solidbyte expects all deploy functions to return True upon success.

Linking Libraries

Linking libraries can be done simply, like so:

```
w3Instance = myContract.deployed(links={
        'MyLibrary': '0x48292eafdc...',
    })
```

The Solidbyte linker will automatically splice these addresss into your solc compiled bytecode. A more real-world example would be deploying both at the same time:

```
myLibrary = contracts.get('MyLibrary')
myContract = contracts.get('MyContract')

library = myLibrary.deployed()
inst = myContract.deployed(links={
        'MyLibrary': library.address
    })
```

Arguments

Solidbyte offers your deploy script's *main()* functions a few optional kwargs.

- `contracts` - an AttrDict instance of your contract instances stored by name
- `web3` - An initialized instance of Web3
- `deployer_account` - The address of the deployer account given on the CLI
- `network` - The name of the network given on the CLI

Just add any of these kwargs that you want to use to your deploy script's `main()` function. For instance:

```
def main(contracts):
    assert isinstance(contracts.ERC20, solidbyte.deploy.objects.Contract)
```

1.3.2 Contract Instances

For details on what methods and properties are available for your `Contract`, see: *`solidbyte.deploy.objects.Contract`*.

More TBD.

1.4 Testing Your Contracts

Testing your contracts with SolidByte is pretty straight forward. SolidByte uses pytest as a test runner and provides some useful fixtures to help ease testing.

1.4.1 Fixtures

`contracts`

The `contracts` fixture is an `attrdict.AttrDict` instance with all of your deployed contracts as `web3.contract.Contract` instances.

`web3`

This is the initialized instance of `web3.Web3` that should already be connected to whatever network you gave on the CLI.

`local_accounts`

`list` of addresses of the known local accounts.

`std_tx`

Function to update a transaction dict with standard values.

`solidbyte.testing.fixtures.`**`std_tx`**(*tx: Union[dict, attrdict.dictionary.AttrDict, web3.datastructures.AttributeDict]*)

Create a test transaction with default gas and gasPrice

Parameters **`tx`** – (`dict`) representation of an Ethereum transaction

Returns (`dict`) representation of an Ethereum transaction with defaults

`has_event`

Function to check if a receipt contains an event.

`solidbyte.testing.fixtures.`**`has_event`**(*web3contract: web3.contract.Contract, event_name: str, rcpt: Union[dict, attrdict.dictionary.AttrDict, web3.datastructures.AttributeDict]*) → bool

Check if a receipt contains an event by name

Parameters

- **web3contract** – (web3.contract.Contract) The contract that has the event ABI we are looking for.
- **event_name** – (str) the name of the event
- **rcpt** – (dict) object of the transaction receipt

Returns (bool)

get_event

Function to pull the event data from a receipt.

solidbyte.testing.fixtures.**get_event**(*web3contract: web3.contract.Contract, event_name: str, rcpt: Union[dict, attrdict.dictionary.AttrDict, web3.datastructures.AttributeDict]*) → Optional[web3.datastructures.AttributeDict]

Return the event data from a transaction receipt

Parameters

- **web3contract** – (web3.contract.Contract) The contract that has the event ABI we are looking for.
- **event_name** – (str) the name of the event
- **rcpt** – (dict) object of the transaction receipt

Returns (dict) the event data

1.4.2 Example Test

Here's an example test provided with the erc20 template:

```
def test_erc20(web3, contracts):
    print("contracts: ", contracts)

    """ We're just going to test to make sure the contracts fixture is being
        populated with deployed contract instances
    """
    assert 'MyERC20' in contracts, "Contract not deployed"
    assert hasattr(contracts.MyERC20, 'address')
    assert type(contracts.MyERC20.address) == str
    assert len(contracts.MyERC20.address) == 42
    assert contracts.MyERC20.address[:2] == '0x'

    assert len(web3.eth.accounts) > 0
    admin = web3.eth.accounts[0]

    # Deployed version should have no tokens to start
    assert contracts.MyERC20.functions.balanceOf(admin).call() == 0
    assert contracts.MyERC20.functions.totalSupply().call() > 0
```

1.5 Scripts

1.5.1 Overview

You can create scrpits that can be run by solidbyte. Solidbyte will provide these scripts with some useful things, like an instantiated `web3.Web3` object and `web3.contract.Contract` representations of your smart contracts.

There's no reason it's necessary to create scripts this way, but it's intended to make things easier.

Example Implementations

For example scripts, see the scripts directory of the solidbyte-test-project repository.

1.5.2 Requirements

The following **must** be implemented in your script for Solidbyte to be able to run it.

`main()`

A `main()` function is expected by Solidbyte when running the `sb script` command. The following kwargs will be provided if you include them in your function definition:

- `network` - The name of the network used in the CLI command
- `contracts` - An *AttrDict* of your deployed contracts.
- `web3` - An instantiated `web3.Web3` object.

A return value is not required, but if `main()` returns `False`, Solidbyte will consider that an error state.

1.6 Project Templates

Project templates are example project structures that may include things like contracts, deploy scripts, and tests all ready to go. They can help you get common project structures setup with a simple `sb init -t [template]` command.

For instnace, you can get an ERC20 project structure setup pretty quick like so:

1.6.1 Available Project Templates

The `bare` template is used by default by the `sb init` command. For now, there are only options but there may be more to come in the future.

bare

This is the most rudimentary structure. It provides you with the expected directories and some basically empty files.

This template is the default.

erc20

This is an example ERC20 token contract. It provides a *MyERC20.sol* contract source file that you can use as a reference to create your own. This template includes example tests and a deployment contract ready to go.

1.7 Project Structure

The project directory structure pretty straight forward. Most of this will be created by `sb init` with a simple template. This example is what is created by the `erc20` template:

```
project_directory/
    |- build/  # Files created by the compilers, including contract ABIs and their␣
↪compiled bytecode.
    |- contracts/  # Solidity and/or Vyper contract source files
        |- ERC20.sol
        |- IERC20.sol
        |- SafeMath.sol
    |- deploy/  # Your deployment scripts.
        |- __init__.py
        |- deploy_main.py
    |- tests/  # Contains your pytest tests to test your contracts
        |- __init__.py
        |- test_erc20.py
    |- networks.yml  # Network/node connection configuration
    |- metafile.json  # Project state
```

For further detailed information, see below.

1.7.1 build/

This directory should be pretty much hands-off and completely managed by Solidbyte. Referencing these files may be useful, but arbitrarily changing anything may cause unexpected behavior. There's no real reason to keep this directory in version control.

1.7.2 contracts/

This directory contains all of your contract source files. They can be Vyper or Solidity or a mix of both if you prefer. The directory structure under this can be whatever you want.

1.7.3 deploy/

`deploy/` contains your deployment scripts. See: *Deployment Scripts*.

1.7.4 tests/

This contains your pytest scripts. See *Testing Your Contracts*.

1.7.5 networks.yml

This file contains your connection configuration. See: *networks.yml*.

1.7.6 metafile.json

This is the file Solidbyte uses to keep track of your project state. Things like the default account, and known deployments of your contracts. Generally, you probably shouldn't fiddle with this file and it's a great idea to keep this file in version control if working in a team. For more information, see *metafile.json*.

1.8 metafile.json

1.8.1 Overview

`metafile.json` is a file that holds your project state. SolidByte may store things like your default account, or the addresses for your contract deployments.

If you're working in a team, it may be a good idea to check this in to your VCS.

WARNING: If you lose this file, SolidByte will have no idea if your contracts are already deployed or not. This could cause duplicate or broken deployments of your contracts. It's also a great idea to at least back it up if you aren't commiting it to a VCS.

WARNING: Editing this file manually, while an option, may cause Solidbyte to behave unexpectedly. Edit it at your own risk and make sure to back it up. See the command: *metafile*

1.8.2 Example `metafile.json`

Here's an example structure of the *metafile.json* file:

```
{
  "contracts": [
    {
      "name": "ExampleContract",
      "networks": {
        "1": {
          "deployedHash": "0xdeadbeef...",
          "deployedInstances": [
            {
              "hash": "0xdeadbeef...",
              "date": "2018-10-21 00:00:00T-7",
              "address": "0xdeadbeef...",
              "abi": [],
            }
          ]
        }
      }
    }
  ],
  "seenAccounts": [
    "0x208B6deadbeef..."
  ],
  "defaultAccount": "0x208B6deadbeef..."
}
```

1.9 networks.yml

`networks.yml` is the YAML file you use to configure connections to Ethereum JSON-RPC providers and nodes. Some templates may provide some pre-configured connections.

1.9.1 Default File

This is the default `networks.yml` provided by the `bare` template:

```yaml
# networks.yml
---
dev:
  type: auto

test:
  type: eth_tester
  autodeploy_allowed: true
  use_default_account: true

infura-mainnet:
  type: websocket
  url: wss://mainnet.infura.io/ws

geth:
  type: ipc
  file: ~/.ethereum/geth.ipc
```

Each root-level node is the network name you will use to reference the configuration. For instance using the above file, if you want to connect to your local go-ethereum IPC endpoint: *sb console geth*

1.9.2 Connection Parameters

`type`

The available connection types are:

- `auto` - Setting the connection to *auto* will allow web3.py to automagically try common configurations for a connection.
- `websocket` - Connect to a Web socket JSON-RPC provider
- `http` - Connect to a plain HTTP(or HTTPS) JSON-RPC provider
- `ipc` - Use the local IPC socket to connect to a local node
- `eth_tester` - A virtual ephemeral chain to test against. Very useful for running unit tests. **NOTE**: eth_tester is in alpha and has been known to show bugs.

`url`

The URL endpoint to connect to. Only available for `http` and `websocket`.

`file`

The IPC socket to connect to. Only available for type `ipc`.

`autodeploy_allowed`

This is a per-network setting that allows Solidbyte to automatically deploy your contracts if it needs to use this network. This is great for test backends, but use at your own risk on public networks. This defaults to `false`.

`use_default_account`

This allows the network to use the account set as default for deployment and testing. This defaults to `false` for safety.

1.9.3 Infura

To use Solidbyte with Infura, make sure you register for an API key and set the `WEB3_INFURA_API_KEY` environmental variable. For more information, see the Web3.py Infura Documentation

CHAPTER 2

Development

2.1 Solidbyte Development

You can find general information here about Solidbyte development and internal classes and objects. This part of the documentation is pretty raw and in its early stages.

2.1.1 Hacker's Guide

If you're looking to hack on SolidByte, you're in the right place.

Pull Requests

. . . are welcome! Best practices TBD

Testing

```
pytest
```

Release

Bump the version with tbump. This will update the version in the source, create a commit, tag the release as *v[version]* and push it up in the current branch. All versions will deploy to test.pypi, but alpha will **NOT** be deployed to prod pypi.

For example, a beta release:

```
tbump v0.3.1b1
```

And a prod release:

```
tbump v0.3.1
```

These will be automagically deployed to PyPi by TravisCI.

Linting

flake8 is used for linting to PEP8 conventions. Best to configure it with your preferred IDE, but you can also run it with the command `python setup.py lint`.

Type Hinting

Type hinting is not required but encouraged. It isn't checked during test builds but if you use it, verify it with mypy or another type checker.

Docstrings

Modules, classes, objects, should all be documented according to the Sphinx docstring syntax

2.1.2 Roadmap

For more information, see the project's milestones on GitHub.

Items marked with a check have work completed and will be released when their version is released.

1.0

- Gas usage reports
- Improved documentation hosted on Read The Docs
- Vyper and Solidity co-mingling (Vyper can not use any libraries, however)
- More commonly used helper functions for contract unit tests
- Reasonable unit test completion
- All around bug fixes

1.1

- Vyper Linting
- EthPM 2.0 Support (if web3 v5 releases)
- Developer experience review

1.2

- Coverage integration
- Solidity Linting

1.3

- Hardware Wallet Support

2.1.3 Solidbyte Modules

Solidbyte Modules

`accounts` Module

The `accounts` module of Solidbyte.

Objects and utility functions for account operations

class `solidbyte.accounts.`**`Accounts`**(*network_name: str = None*, *keystore_dir: str = None*, *web3: web3.main.Web3 = None*)

Deal with local Ethereum secret store account operations

`__init__`(*network_name: str = None*, *keystore_dir: str = None*, *web3: web3.main.Web3 = None*) → None

Init Accounts

Parameters

- **`network_name`** – (`str`) - The name of the network as defined in networks.yml.
- **`keystore_dir`** – (`pathlib.Path`) - Path to the keystore. (default: `~/.ethereum/keystore`)
- **`web3`** – (`web3.Web3`) - The Web3 instance to use

`account_known`(*address: str*) → bool

Check if an account is known

Parameters **`address`** – (`str`) Address of an account to check for

`accounts`

Return all the known account addresses

Returns (`list`) of account addresses

`create_account`(*password: str*) → str

Create a new account and encrypt it with password

Parameters **`password`** – (`str`) Password to use to encrypt the new account

Returns (`str`) address of the new account

`get_account`(*address: str*) → attrdict.dictionary.AttrDict

Return all the known account addresses

Parameters **`address`** – (`str`) Address of account to get

Returns (`attrdict.AttrDict`) of the account

`get_accounts`() → List[attrdict.dictionary.AttrDict]

Return all the known account addresses

Returns (`list`) of account addresses

`refresh`() → None

Load accounts, ignoring cache

set_account_attribute(*address: str*, *key: str*, *val: T*) → None
Set an attribute of an account

Parameters

- **address** – (`str`) address of account
- **key** – (`str`) name of the attribute to set
- **val** – (`T`) new value of the attribute

sign_tx(*account_address: str*, *tx: dict*, *password: str = None*) → str
Sign a transaction using the provided account

Parameters

- **account_address** – (`str`) address of the account to unlock
- **tx** – (`dict`) transaction object to sign
- **password** – (`str`) password to use to decrypt the account

Returns (`str`) transaction hash if successful

unlock(*account_address: str*, *password: str = None*) → bytes
Unlock an account keystore file and return the private key

Parameters

- **account_address** – (`str`) address of the account to unlock
- **password** – (`str`) password to use to decrypt the account

Returns (`bytes`) The account's private key if decryption is successful

`solidbyte.accounts.`**autoload**(*f: Callable*) → Callable
Accounts decorator to utomatically load the accounts before method execution

`common` Module

The `common` module

Solidbyte Exceptions

exception `solidbyte.common.exceptions.`**AccountError**

exception `solidbyte.common.exceptions.`**CompileError**

exception `solidbyte.common.exceptions.`**ConfigurationError**

exception `solidbyte.common.exceptions.`**DeploymentError**

exception `solidbyte.common.exceptions.`**DeploymentValidationError**

exception `solidbyte.common.exceptions.`**InvalidScriptError**

exception `solidbyte.common.exceptions.`**LinkError**

exception `solidbyte.common.exceptions.`**ScriptError**

exception `solidbyte.common.exceptions.`**SolidbyteException**

exception `solidbyte.common.exceptions.`**ValidationError**

exception `solidbyte.common.exceptions.`**WrongPassword**

Solidbyte Session Store

Very simple module we can use to store session-level data. This saves certain things from having to be passed through dozens of functions or objects.

This is not fully implemented project wide yet. Currently experimental.

class `solidbyte.common.store.`**`Keys`**

Enum defining storage keys

`DECRYPT_PASSPHRASE = 'decrypt'`

The account decrypt passphrase that should be session-wide.

`KEYSTORE_DIR = 'keystore'`

The directory with the Ethereum secret store files

`NETWORK_NAME = 'network_name'`

The name of the network being used as defined in networks.yml

`PROJECT_DIR = 'project_dir'`

The project directory. Probably pwd.

`solidbyte.common.store.`**`defined`**(*key: solidbyte.common.store.Keys*) → bool

Check if the key is defined and in STORAGE

Parameters **`key`** – (`Keys`) The key to look for

Returns (`bool`) If the key is defined in storage

`solidbyte.common.store.`**`get`**(*key: solidbyte.common.store.Keys*) → Optional[Any]

Get the value stored for the key

Parameters **`key`** – (`Keys`) The key of the value to return

Returns (`Any`) The value of the key

`solidbyte.common.store.`**`set`**(*key: solidbyte.common.store.Keys*, *val: Any*) → Optional[Any]

Set the value of the key and return the new value

Parameters

- **`key`** – (`Keys`) The key of the value to return
- **`val`** – (`Any`) The value to set

Returns (`Any`) The value of the key

Common Utility Functions

class `solidbyte.common.utils.`**`Py36Datetime`**

Monkeypatch datetime for python<3.7

`fromisoformat`()

Load an `datetime.isoformat()` date string as a `datetime` object

`solidbyte.common.utils.`**`all_defs_in`**(*items: Iterable[T_co]*, *di: dict*) → bool

Check if all defs(tuple of name/placeholder) are in di

Parameters

- **`items`** – (`Iterable`) to check against di
- **`di`** – (`dict`) the dict to check against

Returns (`bool`) if all defs are in the dict

`solidbyte.common.utils.`**`builddir`**(*loc=None*)

Get (and create if necessary) the temporary build dir

Parameters **`loc`** – (`pathlib.Path`) to workdir (default: pwd)

Returns (`pathlib.Path`) to build dir

`solidbyte.common.utils.`**`collapse_oel`**(*lst*)

Collapse a one-element list to a single var

Parameters **`filename`** – (`list`) with one element to collapse

Returns (`Any`) the single element

`solidbyte.common.utils.`**`defs_not_in`**(*items: Iterable[T_co], di: dict*) → set

Find defs (tuple of name/placeholder) that aren't keys in a dict

Parameters

- **`items`** – (`Iterable`) to check against di
- **`di`** – (`dict`) the dict to check against

Returns (`set`) any defs not in di

`solidbyte.common.utils.`**`find_vyper`**()

Get the path to vyper. **DEPRECIATED**

`solidbyte.common.utils.`**`get_filename_and_ext`**(*filename*)

Return the filename and extension as a tuple

Parameters **`filename`** – (`pathlib.Path`) of file

Returns (`tuple`) of (name, extension)

`solidbyte.common.utils.`**`hash_file`**(*_file: pathlib.Path*) → str

Get an sha1 hash of a file

Parameters **`_file`** – (`pathlib.Path`) the file to hash

Returns (`str`) hex sha1 hash of the given file

`solidbyte.common.utils.`**`keys_with`**(*thedict*, *term*)

Return any keys from `thedict` that have `term` in their value

Parameters

- **`thedict`** – (`dict`) The dict to search
- **`term`** – (`Any`) The value to look for

Returns (`list`) List of keys that match

`solidbyte.common.utils.`**`pop_key_from_dict`**(*d*, *key*)

Remove and return an element from a dict and the modded dict without throwing an exception if a key does not exist.

Parameters

- **`d`** – (`dict`) the original dict
- **`key`** – (`str`) they key to pop

Returns (`T`) The value of the key or None

`solidbyte.common.utils.`**`source_filename_to_name`**(*filename*)
Change a source filename to a plain name

Parameters **`filename`** – (`pathlib.Path`) of file

Returns (`str`) name of file without extension

`solidbyte.common.utils.`**`supported_extension`**(*filename*)
Check if the provided filename has a supported source code extension

Parameters **`filename`** – (`pathlib.Path`) of file

Returns (`bool`) if it's supported

`solidbyte.common.utils.`**`to_path`**(*v*) → pathlib.Path
Given a Path or str, return a Path

Parameters **`v`** – (`str` or `pathlib.Path`)

Returns (`pathlib.Path`)

`solidbyte.common.utils.`**`to_path_or_cwd`**(*v*) → pathlib.Path
Given a Path, str, or None, return a Path of the given path or the current working directory

Parameters **`v`** – (`str` or `pathlib.Path`)

Returns (`pathlib.Path`)

`solidbyte.common.utils.`**`unescape_newlines`**(*s*)
Unescape newlines in a text string

Parameters **`s`** – (`str`) String to search and replace against

Returns (`str`) String with unescaped newlines

`compile` Module

The `compile` module

`compile.artifacts` Module

The `compile.artifacts` module

`class` `solidbyte.compile.artifacts.`**`CompiledContract`**(*name: str, artifact_path: Union[pathlib.Path, str]*)
A representation of a compiled contract.

Attributes:

- `name` (`str`) - The name of the contract
- `artifact_path` (`pathlib.Path`) - The Path to the contract's artifact directory
- `paths` (`attrdict.AttrDict`) - Paths to eact artifact file
- `abi` (`dict`) - A Python dict of the contract's ABI
- `bytecode` (`str`) - The contract's compiled bytecode

`__init__`(*name: str, artifact_path: Union[pathlib.Path, str]*) → None
Initialize self. See help(type(self)) for accurate signature.

`solidbyte.compile.artifacts.`**`artifacts`**(*project_dir: Union[pathlib.Path, str]*) → Set[solidbyte.compile.artifacts.CompiledContract]

Get an *`solidbyte.compile.artifacts.CompiledContract`* object for all compiled contracts

`solidbyte.compile.artifacts.`**`available_contract_names`**(*project_dir: Union[pathlib.Path, str]*) → Set[str]

Return the names of all compiled contracts

`solidbyte.compile.artifacts.`**`contract_artifacts`**(*name: str, project_dir: Union[pathlib.Path, str] = None*) → solidbyte.compile.artifacts.CompiledContract

Return a *`solidbyte.compile.artifacts.CompiledContract`* object with the artifacts for a contract

`compile.compiler` Module

The `compile.compiler` module

Solidity compiling functionality

`class` `solidbyte.compile.compiler.`**`Compiler`**(*project_dir=None*)

Handle compiling of contracts

`__init__`(*project_dir=None*)

Initialize self. See help(type(self)) for accurate signature.

`compile`(*filename*)

Compile a single source contract at `filename`

Parameters **`filename`** – Source contract's filename

`compile_all`()

Compile all source contracts

`solc_version`

Get the version of the solidity copmiler

Returns A `str` representation of the version

`version`

A `list` of all compiler versions

`vyper_version`

Get the version of the vyper copmiler

Returns A `str` representation of the version

`solidbyte.compile.compiler.`**`get_all_source_files`**(*contracts_dir: pathlib.Path*) → Set[pathlib.Path]

Return a Path for every contract source file in the provided directory and any sub- directories.

Parameters **`contracts_dir`** – The Path of the directory to start at.

Returns List of Paths to every source file in the directory.

`compile.linker` Module

The `compile.linker` module

Functions used for linking libraries for Solidity libraries

Example Solidity placeholder: `__$13811623e8434e588b8942cf9304d14b96$__`

`solidbyte.compile.linker.`**`address_placeholder`**(*name*)

Provide a false, but repeatable address for a link ref with name. Used in bytecode hashing.

Parameters **`name`** – (`str`) The name to use for the placeholder

Returns (`str`) A false address derrived from a placeholder

`solidbyte.compile.linker.`**`bytecode_link_defs`**(*bytecode*) → Set[Tuple[str, str]]

Return set of tuples with names and placeholders for link definitions from a bytecode file

Parameters **`bytecode`** – (`str`) Contents of a Solidity bytecode output file

`solidbyte.compile.linker.`**`clean_bytecode`**(*bytecode: str*) → str

Clean the bytecode string of any comments and whitespace

Parameters **`bytecode`** – (`str`) Bytecode output from the Solidity compiler

`solidbyte.compile.linker.`**`contract_from_def`**(*s: str*) → str

return a contract name form a solc file link definition

Parameters **`s`** – (`str`) A “definition” from a solidity bytecode output file

`solidbyte.compile.linker.`**`hash_linked_bytecode`**(*bytecode*) → str

Hash bytecode that has link references in a way that the addresses for delegate calls don’t matter. Useful for comparing bytecode hashes when you don’t know deployed addresses.

Parameters **`bytecode`** – (`str`) Bytecode output from the Solidity compiler

Returns (`str`) A link-agnostic hash of the bytecode

`solidbyte.compile.linker.`**`link_library`**(*bytecode: str*, *links: dict*) → str

Providing bytecode output from the Solidity copmiler and a dict of links, perform the placeholder replacement to create deployable bytecode.

Parameters

- **`bytecode`** – (`str`) Bytecode output from the Solidity compiler
- **`links`** – (`dict`) A dict of links. ContractName:Address

`solidbyte.compile.linker.`**`make_placeholder_regex`**(*placeholder: str*) → Pattern[str]

Return a regex pattern for a placeholder

Parameters **`placeholder`** – (`str`) Given a solidity placeholder, make a regex

`solidbyte.compile.linker.`**`placeholder_from_def`**(*s: str*) → str

return a placeholder form a solc file link definition

Parameters **`s`** – (`str`) A “definition” from a solidity bytecode output file

`solidbyte.compile.linker.`**`replace_placeholders`**(*bytecode: str*, *placeholder: str*, *addr: str*) → str

Replace the placeholders with the contract address

Parameters

- **`bytecode`** – (`str`) Solidity bytecode output
- **`placeholder`** – (`str`) The placeholder to replace
- **`addr`** – (`str`) Address to replace the placeholder with

`compile.solidity` Module

The `compile.solidity` module

Solidity compilation utilities

`solidbyte.compile.solidity.`**`is_solidity_interface_only`**(*filepath: Union[str, pathlib.Path]*) → bool

Given a path to a source file, check if the file only defines an `interface`, but no other `contract`.

Parameters **`filepath`** – (`str` or `pathlib.Path`) Path to the source file to check

Returns (`bool`) If it's recognize as a Solidity interface

`solidbyte.compile.solidity.`**`parse_file`**(*filepath: pathlib.Path*) → dict

Parse a file using solidity_parser

Parameters **`filepath`** – (`str` or `pathlib.Path`) Path to the source file to check

Returns (`dict`) A Python dict representation of the source file

`compile.vyper` Module

The `compile.vyper` module

Vyper utilities

`solidbyte.compile.vyper.`**`dirs_in_dir`**(*searchpath*)

Return a list of all child directories of a directory

Parameters **`searchpath`** – (`pathlib.Path`) The Path of a directory to search

Returns (`list`) A list of paths of each child directory

`solidbyte.compile.vyper.`**`is_bodyless_func`**(*func_text*)

Identify if a code block is a bodyless/interface function

Parameters **`func_text`** – (`str`) The source code for a function

Returns (`str`) If the function is "bodyless". (empty or only `pass`)

`solidbyte.compile.vyper.`**`is_vyper_interface`**(*source_text*)

Identify if the provided source text is a vyper interface

Parameters **`source_text`** – (`str`) The full source code

Returns (`bool`) If the provided source code is a Vyper interface

`solidbyte.compile.vyper.`**`source_extract`**(*source_text*, *start_ln*, *end_ln*)

Extract a section of source code given start and end line numbers.

Parameters

- **`source_text`** – (`str`) The full source code
- **`start_ln`** – (`int`) The start line number
- **`end_ln`** – (`int`) The end line number

Returns (`str`) The source code snippet

`solidbyte.compile.vyper.`**`vyper_funcs_from_source`**(*source_text*)

Generate an AST and pull all function defs from it

Parameters **`source_text`** – (`str`) The full source code

Returns (`list`) The source code definition of the functions

`solidbyte.compile.vyper.`**`vyper_import_to_file_paths`**`(`*workdir*, *importpath*`)`

Resolve a Vyper import path to a file

Parameters

- **`workdir`** – (`pathlib.Path`) The Path to a directory to search
- **`importpath`** – (`str`) The vyper import statement to resolve

Returns (`pathlib.path`) The Path to the file the import resolves to

`solidbyte.compile.`**`compile_all`**`()`

Compile all contracts in the current project directory

`console` Module

The *console* module of Solidbyte.

`class` `solidbyte.console.`**`SolidbyteConsole`**`(`*_locals=None*, *filename='<console>'*, *network_name=None*, *histfile=PosixPath('/home/docs/.solidbyte-history')*, *web3=None*`)`

`__init__``(`*_locals=None*, *filename='<console>'*, *network_name=None*, *histfile=PosixPath('/home/docs/.solidbyte-history')*, *web3=None*`)`

Constructor.

The optional locals argument will be passed to the InteractiveInterpreter base class.

The optional filename argument should specify the (file)name of the input stream; it will show up in tracebacks.

`interact``(`*banner=None*, *exitmsg='bye!'*`)`

Closely emulate the interactive Python console.

The optional banner argument specifies the banner to print before the first interaction; by default it prints a banner similar to the one printed by the real Python interpreter, followed by the current class name in parentheses (so as not to confuse this with the real interpreter – since it's so close!).

The optional exitmsg argument specifies the exit message printed when exiting. Pass the empty string to suppress printing an exit message. If exitmsg is not given or None, a default message is printed.

`deploy` Module

The `deploy` module

Deployer

Ethereum deployment functionality

`class` `solidbyte.deploy.`**`Deployer`**`(`*network_name: str*, *account: str = None*, *project_dir: Union[pathlib.Path, str] = None*`)`

The big ugly black box of an object that handles deployment in one giant muddy process but it tries to be useful to various parts of the system and represent the current state of the entire project's deployments.

The primary purpose of this object is to know if a deployment is necessary, and to handle the deployment of all contracts if necessary.

Example:

```
from solidbyte.deploy import Deployer
d = Deployer('test', '0xdeadbeef00000000000000000000000000000000',
             Path('/path/to/my/project'))
assert d.check_needs_deploy() == True
d.deploy()
```

`__init__`(*network_name: str*, *account: str = None*, *project_dir: Union[pathlib.Path, str] = None*)
Initialize the Deployer. Get it juiced up. Make the machine shudder.

Parameters

- **`network_name`** – (`str`) The name of of the network, as defined in networks.yml.
- **`account`** – (`str`) The address of the account to deploy with.
- **`project_dir`** – (`Path`/`str`) The project directory, if not pwd.

`artifacts`
Returns the ABI and Bytecode artifacts, generated from the build direcotry.

Parameters **`force`** – (`bool`) Force load, don't just rely on cached dicts.

Returns (`AttrDict`) An AttrDict representing all available contracts

`check_needs_deploy`(*name: str = None*) → bool
Check if any contracts need to be deployed

Parameters **`name`** – (`str`) The name of a contract if checking a specific.

Returns (`bool`) if deployment is required

`contracts`
Returns instantiated Contract objects to provide to the deploy scripts.

Parameters **`force`** – (`bool`) Force load, don't just rely on cached data.

`contracts_to_deploy`() → Set[str]
Return a Set of contract names that need deployment

`deploy`() → bool
Deploy the contracts with magic lol

Returns (`bool`) if deployment succeeded. Fails miserably if it didn't.

`deployed_contracts`
Contracts from MetaFile

`get_artifacts`(*force: bool = False*) → attrdict.dictionary.AttrDict
Returns the ABI and Bytecode artifacts, generated from the build direcotry.

Parameters **`force`** – (`bool`) Force load, don't just rely on cached dicts.

Returns (`AttrDict`) An AttrDict representing all available contracts

`get_contracts`(*force: bool = False*)
Returns instantiated Contract objects to provide to the deploy scripts.

Parameters **`force`** – (`bool`) Force load, don't just rely on cached data.

`refresh`(*force: bool = True*) → None
Return the available kwargs to give to user scripts

Parameters **`force`** – (`bool`) Don't rely on cache and reload everything.

Deployer Objects

Contract deployer

class `solidbyte.deploy.objects.`**Contract**(*name: str*, *network_name: str*, *from_account: str*, *metafile: MetaFile*, *web3: Web3 = None*)

The representation of a smart contract deployment state on a specific network.

This object is exposed to users' in their deploy script, instantiated for each of their contracts. The general use is to provide information about the contract state, a Web3 Contract instance and deploy it if necessary. There's a bunch of properties, but two primary methods to interact with the contract.

- **check_needs_deployment(bytecode: str) -> bool: This function will take the provided bytecode** and return whether or not there's been a change compared to the known, deployed bytecode.
- **deployed(*args: T, **kwargs: T) -> web3.eth.Contract: This is the primary interaction a user** has with this object in a deploy script. It will return an instantiated web3.eth.Contract instance and in the process, deploy the smart contract if necessary.

__init__(*name: str*, *network_name: str*, *from_account: str*, *metafile: MetaFile*, *web3: Web3 = None*)

Initialize the Contract

Parameters

- **`name`** – The name of. . . me. The name of the contract I represent.
- **`network_name`** – The name of of the network, as defined in networks.yml.
- **`from_account`** – The address of the account to deploy with.
- **`metafile`** – An instantiated MetaFile object.
- **`web3`** – An instantiated Web3 object

Example

```
>>> from solidbyte.deploy.objects import Contract
>>> MyContract = Contract(
...         'TestContract',
...         'test',
...         '0xdeadbeef00000000000000000000000000000000',
...         MetaFile(),
...         Web3(),
... )
>>> my_contract = MyContract.deployed()
```

abi

The latest deployed ABI

address

The latest deployed address

bytecode_hash

The latest deployed bytecode hash

check_needs_deployment(*bytecode: str*) → bool

Check if this contract has been changed since last deployment

NOTE: This method does not take into account dependencies. Check with Deployer

Parameters **`bytecode`** – The hex bytecode to compare to the latest known deployment.

Returns If the bytecode differs from the last known deployment.

Example

```
>>> from solidbyte.deploy.objects import Contract
>>> MyContract = Contract('test', '0xdeadbeef00000000000000000000000000000000
↪', {
...         'abi': [],
...         'bytecode': '0x1234...'
...         'name': 'MyContract'
...     }, {}, MetaFile())
>>> assert Mycontract.check_needs_deployment('0x2234...')
```

deployed(**args*, ***kwargs*)
Return an instantiated web3.eth.Contract tinstance and deploy the contract if necessary.

Parameters

- ***args** – Any args to provide the constructor.
- ****kargs** – Any kwargs to provide the constructor OR one of the following special kwargs: - gas: The gas limit for the deploy transaction. - gas_price: The gas price to use, in wei. - links: This is a dict of {name,address} of library links for the contract.

Returns If the bytecode differs from the last known deployment.

Example

```
>>> from solidbyte.deploy.objects import Contract
>>> MyContract = Contract('test', '0xdeadbeef00000000000000000000000000000000
↪', {
...         'abi': [],
...         'bytecode': '0x1234...'
...         'name': 'MyContract'
...     }, {}, MetaFile())
>>> contract = Mycontract.deploy(links={
            'MyLibrary': '0xdeadbeef000000000000000000000000000000001'
        })
>>> assert contract.functions.owner().call() == contract.from_account
```

is_deployed() → bool
Return if this contract has deployments

refresh() → None
Refresh metadata from MetaFile and the compiled artifacts

class solidbyte.deploy.objects.**ContractDependencyTree**
A tree of Leafs describing contract library dependencies

Example

```
>>> deptree = ContractDependencyTree()
```

__init__()
Initialize self. See help(type(self)) for accurate signature.

add_dependent(*name: str*, *parent: str = None*) → solidbyte.deploy.objects.ContractLeaf
Add a child dependent

has_dependencies(*name: str*)
Check of name has dependencies

has_dependents(*name: str*)
Check of name has dependents

move(*name: str*, *new_parent: solidbyte.deploy.objects.ContractLeaf*) → solidbyte.deploy.objects.ContractLeaf
Move an element to be a child of another

search_tree(*name: str*) → Tuple[Optional[solidbyte.deploy.objects.ContractLeaf], int]
Search a tree for a named leaf

Parameters **name** – The name of the leaf to look for

class solidbyte.deploy.objects.**ContractLeaf**(*name: str*, *tree: solidbyte.deploy.objects.ContractDependencyTree*, *parent: Optional[solidbyte.deploy.objects.ContractLeaf] = None*)
A leaf object in the dependency tree

Definitions

- dependent: Leaves that this leaf depends on
- dependency: A leaf that depends on this leaf

__init__(*name: str*, *tree: solidbyte.deploy.objects.ContractDependencyTree*, *parent: Optional[solidbyte.deploy.objects.ContractLeaf] = None*) → None
Initialize self. See help(type(self)) for accurate signature.

add_dependent(*name: str*) → solidbyte.deploy.objects.ContractLeaf
Add a dependent leaf

attach_dependent(*el: solidbyte.deploy.objects.ContractLeaf*) → None
Attach an element to this Leaf as dependent

get_dependencies() → Set[solidbyte.deploy.objects.ContractLeaf]
Resolve and return all dependencies in a flat set

get_dependents() → Set[solidbyte.deploy.objects.ContractLeaf]
Resolve and return all dependents in a flat set

get_parent() → Optional[solidbyte.deploy.objects.ContractLeaf]
Return the parent ContractLeaf

has_dependencies() → bool
Does this leaf have dependencies?

has_dependents() → bool
Does this leaf have dependents

is_root() → bool
Is this the root leaf?

class solidbyte.deploy.objects.**Deployment**(*network: str, address: str, bytecode_hash: str, date: datetime.datetime, abi: List[Dict[str, Optional[Any]]]*)
representation of a simgle contract deployment

__init__(*network: str, address: str, bytecode_hash: str, date: datetime.datetime, abi: List[Dict[str, Optional[Any]]]*)
Initialize self. See help(type(self)) for accurate signature.

solidbyte.deploy.objects.**get_lineage**(*leaf: solidbyte.deploy.objects.ContractLeaf*) → Set[solidbyte.deploy.objects.ContractLeaf]
Climb a deptree and return all elements “above” the provided leaf

`script` Module

The *script* module of Solidbyte.

Functionality for running user scripts

`solidbyte.script.`**`get_availble_script_kwargs`**(*network*, *account: str = None*) → Dict[str, Any]
  Get a dict of the kwargs available for user scripts

`solidbyte.script.`**`get_contracts`**(*network: str*, *account: str = None*) → attrdict.dictionary.AttrDict
  Get a list of web3 contract instances.

`solidbyte.script.`**`run_script`**(*network: str*, *script: str*, *account: str = None*) → bool
  Runs a user script

`solidbyte.script.`**`run_scripts`**(*network: str, scripts: List[str], account: str = None*) → bool
  Run multiple user scripts

`templates` Module

The `templates` module

`templates.template` Module

The `templates.template` module

Abstract template class

`class` `solidbyte.templates.template.`**`Template`**(**args*, ***kwargs*)
  Template abstract

  `__init__`(**args*, ***kwargs*)
    Init the Template object. Arguments can be added by subclasses. The one used by Template are documented below.

    Parameters

    - **`dir_mode`** – (`int`) The directory mode permissions
    - **`pwd`** – (`pathlib.Path`) The current working directory

  `copy_template_file`(*dest_dir*, *subdir*, *filename*)
    Copy a file from the template module directory to dest

    Parameters

    - **`dest_dir`** – (`pathlib.Path`) - The destination directory in the project structure
    - **`subdir`** – (`pathlib.Path`) - The subdirectory under dest_dir
    - **`filename`** – (`str`) - The name of the destination file

    Returns (`str`) Destination path

  `create_dirs`()
    Create the project directory structure

  `initialize`()
    This method performs all steps necessary to build a template. It must be implemented by the Template subclass.

Built-in Templates

The built-in project templates for Solidbyte

Bare Template

The bare template that creates a minimum project structure.

Create a bare project template

class `solidbyte.templates.templates.bare.`**`BareTemplate`**`(`**args*, ***kwargs*`)`

`__init__``(`**args*, ***kwargs*`)`
Init the Template object. Arguments can be added by subclasses. The one used by Template are documented below.

Parameters

- **`dir_mode`** – (`int`) The directory mode permissions
- **`pwd`** – (`pathlib.Path`) The current working directory

`create_deployment``()`
Create the deploy file

`create_networks``()`
Create the networks.yml file

`initialize``()`
Initialize the template and create a bare project structure

`solidbyte.templates.templates.bare.`**`get_template_instance`**`(`**args*, ***kwargs*`)`
Return a bare template

ERC20 Template

The ERC20 template that creates a ready to go token.

Create a project template with an ERC20 contract and accompanying tests

class `solidbyte.templates.templates.erc20.`**`ERC20Template`**`(`**args*, ***kwargs*`)`

`__init__``(`**args*, ***kwargs*`)`
Init the Template object. Arguments can be added by subclasses. The one used by Template are documented below.

Parameters

- **`dir_mode`** – (`int`) The directory mode permissions
- **`pwd`** – (`pathlib.Path`) The current working directory

`create_contracts``()`
Create the contract source files

`create_deployment``()`
Create the deploy module and script

create_networks()
  Create the networks.yml file

create_tests()
  Create the test files

initialize()
  Create a project structure for an ERC20 token

solidbyte.templates.templates.erc20.**get_template_instance**(**args*, ***kwargs*)
  Return an ERC20 template

Templates

Every template should at a minimum implement this function that returns an instance of `solidbyte.templates.Template`.

```
def get_template_instance(*args, **kwargs):
    pass
```

For more details, see the `solidbyte.templates.templates.bare.Bare` template.

solidbyte.templates.**get_templates**()
  Return all available templates **DEPRECIATED**

solidbyte.templates.**init_template**(*name*, *dir_mode=493*, *pwd=None*)
  Initialize and return a Template instance with name

solidbyte.templates.**lazy_load_templates**(*force_load=False*)
  Import all templates and stuff them into the `TEMPLATES` global

`testing` Module

The *testing* module of Solidbyte.

2.1.4 MetaFile

MetaFile is a representation of the `metafile.json` file.

Store and retrieve metdata about a contract for this project

Example JSON structure:

```
{
  "contracts": [
    {
      "name": "ExampleContract",
      "networks": {
        "1": {
          "deployedHash": "0xdeadbeef...",
          "deployedInstances": [
            {
              "hash": "0xdeadbeef...",
              "date": "2018-10-21 00:00:00T-7",
              "address": "0xdeadbeef...",
            }
```

(continues on next page)

(continued from previous page)

```
          ]
        }
      }
    }
  ],
  "seenAccounts": [
    "0xdeadbeef..."
  ],
  "defaultAccount": "0xdeadbeef..."
}
```

class solidbyte.common.metafile.**MetaFile**(*filename_override: Union[pathlib.Path, str] = None, project_dir: Union[pathlib.Path, str] = None, read_only: bool = False*)

Class representing the project metafile

__init__(*filename_override: Union[pathlib.Path, str] = None, project_dir: Union[pathlib.Path, str] = None, read_only: bool = False*) → None

Initialize self. See help(type(self)) for accurate signature.

account_known(*address: str*) → bool

Check if an account is known

add_account(*address: str*) → None

Add an account to seenAccounts

backup(*outfile*) → bool

Backup the metafile.json and verify

cleanup(*dry_run: bool = False*) → List[Tuple[str, str, str]]

Cleanup metafile.json of test deployments. In practice, this means any deployments with a network_id > 100, as the last semi-official network_id is 99.

Returns a list of tuple. Tuples are (name, network_id).

get_all_contracts() → List[attrdict.dictionary.AttrDict]

return all meta data for all contracts

get_contract(*name*) → attrdict.dictionary.AttrDict

Get the meta data for a contract

get_default_account() → Optional[str]

Get the default account

set_default_account(*address*) → None

Set the default account

solidbyte.common.metafile.**autoload**(*f: Callable*) → Callable

MetaFile method decorator to automatically load the metafile before method execution

solidbyte.common.metafile.**autosave**(*f: Callable*) → Callable

MetaFile method decorator to automatically save the metafile after method execution

2.1.5 NetworksYML

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index