Brownie Documentation

Release v1.3.1

Ben Hauser

Dec 31, 2020

Contents

CHAPTER 1

Brownie

Brownie is a Python framework for Ethereum smart contract testing, interaction and deployment.

Note: All code starting with `$` is meant to be run on your terminal. Code starting with >>> is meant to run inside the Brownie console.

Note: This project relies heavily upon `web3.py` and the documentation assumes a basic familiarity with it. You may wish to view the Web3.py docs if you have not used it previously.

Brownie has several uses:

- **Testing**: Unit test your project with `pytest`, and evaluate test coverage through stack trace analysis. We make *no promises*.
- **Debugging**: Get detailed information when a transaction reverts, to help you locate and solve the issue quickly.
- **Interaction**: Write scripts or use the console to interact with your contracts on the main-net, or for quick testing in a local environment.
- **Deployment**: Automate the deployment of many contracts onto the blockchain, and any transactions needed to initialize or integrate the contracts.

CHAPTER 2

Quickstart

This page will walk you through the basics of using Brownie. Please review the rest of the documentation to learn more about specific functionality.

2.1 Installing Brownie

2.1.1 Dependencies

Before installing Brownie, make sure you have the following dependencies:

- ganache-cli
- pip
- python3 version 3.6 or greater, python3-dev, python3-tk

As brownie relies on py-solc-x, you do not need solc installed locally but you must install all required solc dependencies.

2.1.2 Installation

The easiest way to install Brownie is via pip.

```
$ pip install eth-brownie
```

You can also clone the github repository and use setuptools for the most up-to-date version.

```
$ python3 setup.py install
```

2.2 Initializing a New Project

The first step to using Brownie is to initialize a new project. To do this, create an empty folder and then type:

```
$ brownie init
```

This will create the following project structure within the folder:

- `build/`: Compiled contracts and test data
- `contracts/`: Contract source code
- `scripts/`: Scripts for deployment and interaction
- `tests/`: Scripts for testing your project
- `brownie-config.yaml`: *Configuration file* for the project

You can also initialize "Brownie mixes", simple templates to build your project upon. For the examples in this document we will use the token mix, which is a very basic ERC-20 implementation:

```
$ brownie bake token
```

This creates a new folder `token/` and deploys the project inside it.

2.3 Compiling your Contracts

To compile your project:

```
$ brownie compile
```

You will see the following output:

```
Brownie v1.0.0 - Python development framework for Ethereum

Compiling contracts...
Optimizer: Enabled  Runs: 200
- Token.sol...
- SafeMath.sol...
Brownie project has been compiled at token/build/contracts
```

Once a contract has been complied, it will only be recompiled if the source file has changed.

You can change the compiler version and optimization settings by editting the *config file*.

2.4 Interacting with your Project

Brownie provides two ways to interact with your project:

- The **console** is useful for quick testing and debugging as you develop
- Via **scripts** that handle deployments and to automate common tasks

2.4.1 The Console

The console is an easy to use command-line environment for debugging and testing as you develop. It is almost identical the standard python interpreter. To open it:

```
$ brownie console
```

Brownie will compile your contracts, start the local RPC client, and give you a command prompt. From here you may interact with the network with the full range of functionality offered by the *Brownie API*.

Hint: Within the console, the builtin `dir` is modified to only display public methods and attributes. It is a valuable tool for exploring Brownie's functionality as you are getting started.

You can also call `help` for detailed information on most objects.

Access to local accounts is through `accounts`, a list-like object that contains `Account` objects capable of making transactions.

Here is an example of checking a balance and transfering some ether:

```
>>> accounts[0]
<Account object '0xC0BcE0346d4d93e30008A1FE83a2Cf8CfB9Ed301'>
>>> accounts[1].balance()
100000000000000000000
>>> accounts[0].transfer(accounts[1], "10 ether")

Transaction sent: 0x124ba3f9f9e5a8c5e7e559390bebf8dfca998ef32130ddd114b7858f255f6369
Transaction confirmed - block: 1   gas spent: 21000
<Transaction object
 ↪'0x124ba3f9f9e5a8c5e7e559390bebf8dfca998ef32130ddd114b7858f255f6369'>
>>> accounts[1].balance()
110000000000000000000
```

Brownie creates a `ContractContainer` object for each contract in your project. They are list-like objects used to deploy new contracts.

Here is an example of deploying a contract:

```
>>> Token
[]
>>> Token.deploy
<ContractConstructor object 'Token.constructor(string _symbol, string _name, uint256 _
 ↪decimals, uint256 _totalSupply)'>
>>> t = Token.deploy("Test Token", "TST", 18, "1000 ether", {'from': accounts[1]})

Transaction sent: 0x2e3cab83342edda14141714ced002e1326ecd8cded4cd0cf14b2f037b690b976
Transaction confirmed - block: 1   gas spent: 594186
Contract deployed at: 0x5419710735c2D6c3e4db8F30EF2d361F70a4b380
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>>
>>> t
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
```

When a contact is deployed you are returned a `Contract` object that can be used to interact with it. This object is also added to the `ContractContainer`.

`Contract` objects contain class methods for performing calls and transactions. In this example we are checking a token balance and transfering tokens:

```
>>> t
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>> t.balanceOf(accounts[1])
10000000000000000000

>>> t.transfer
<ContractTx object 'transfer(address _to, uint256 _value)'>
>>> t.transfer(accounts[2], "100 ether", {'from': accounts[1]})

Transaction sent: 0xcd98225a77409b8d81023a3a4be15832e763cd09c74ff431236bfc6d56a74532
Transaction confirmed - block: 2   gas spent: 51241
<Transaction object
 ↪'0xcd98225a77409b8d81023a3a4be15832e763cd09c74ff431236bfc6d56a74532'>
>>>
>>> t.balanceOf(accounts[1])
9000000000000000000
>>> t.balanceOf(accounts[2])
100000000000000000000
```

See *Project Interaction via the Console* for more information on available objects and how they function.

2.4.2 Writing Scripts

You can write scripts to automate contract deployment and interaction. By placing `from brownie import *` at the beginning of your script, you can access objects identically to the way you would in the console.

To execute the `main` function in a script, store it in the `scripts/` folder and type:

```
$ brownie run [script name]
```

Within the token project, you will find an example script at scripts/token.py that is used for deployment:

```
1 from brownie import *
2
3 def main():
4     accounts[0].deploy(Token, "Test Token", "TEST", 18, "1000 ether")
```

2.5 Testing your Project

Brownie uses the `pytest` framework for contract testing.

Tests should be stored in the `tests/` folder. To run the full suite:

```
$ pytest tests/
```

Brownie provides pytest fixtures to allow you to interact with your project and to aid in testing. To use a fixture, add an argument with the same name to the inputs of your test function.

Here is an example test function using Brownie fixtures:

```
1 def test_transfer(Token, accounts):
2     token = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")
3     assert token.totalSupply() == "1000 ether"
4     token.transfer(accounts[1], "0.1 ether", {'from': accounts[0]})
```

(continues on next page)

(continued from previous page)

```
    assert token.balanceOf(accounts[1]) == "0.1 ether"
    assert token.balanceOf(accounts[0]) == "999.9 ether"
```

Transactions that revert raise a `VirtualMachineError` exception. To write assertions around this you can use `pytest.reverts` as a context manager, which functions very similarly to `pytest.raises`:

```
import pytest

def test_transferFrom_reverts(Token, accounts):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")
    with pytest.reverts():
        token.transferFrom(accounts[0], accounts[3], "10 ether", {'from': accounts[1]}
↪)
```

Test isolation is handled through the `module_isolation` and `fn_isolation` fixtures:

- `module_isolation` resets the local chain before and after completion of the module, ensuring a clean environment for this module and that the results of it will not affect subsequent modules.
- `fn_isolation` additionally takes a snapshot of the chain before running each test, and reverts to it when the test completes. This allows you to define a common state for each test, reducing repetitive transactions.

This example uses isolation and a shared setup fixture:

```
import pytest
from brownie import accounts

@pytest.fixture(scope="module")
def token(Token):
    t = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")
    yield t

def test_transferFrom(fn_isolation, token):
    token.approve(accounts[1], "6 ether", {'from': accounts[0]})
    token.transferFrom(accounts[0], accounts[2], "5 ether", {'from': accounts[1]})
    assert token.balanceOf(accounts[2]) == "5 ether"
    assert token.balanceOf(accounts[0]) == "995 ether"
    assert token.allowance(accounts[0], accounts[1]) == "1 ether"

def test_balance_allowance(fn_isolation, token):
    assert token.balanceOf(accounts[0]) == "1000 ether"
    assert token.allowance(accounts[0], accounts[1]) == 0
```

Brownie monitors which files have changed since the test suite was last executed. Tests that are properly isolated can be skipped if none of the contracts or related test files have changed. To enable this, include the `--update` flag when running `pytest`.

See *Unit Testing with Pytest* for more information on available fixtures, and other features and options related to unit testing.

2.6 Analyzing Test Coverage

Test coverage is calculated by generating a map of opcodes associated with each statement and branch of the source code, and then analyzing the stack trace of each transaction to see which opcodes executed.

To check test coverage:

```
$ pytest tests/ --coverage
```

To view detailed results, first load the Brownie GUI:

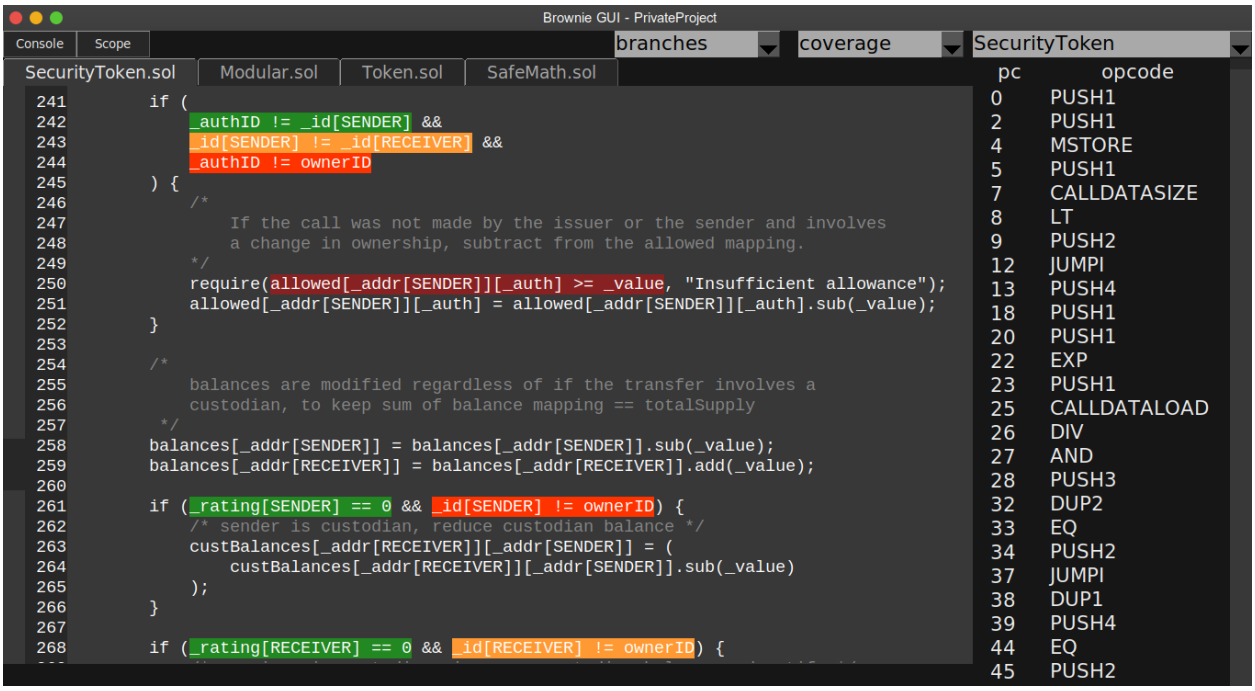
```
$ brownie gui
```

Next:

- In the upper-right drop box, select a contract to view.
- In the drop box immediately left of the contract selection, select "coverage". Then left of that, choose to view either the "statement" or "branch" coverage report.

Relevant code will be highlighted in different colors:

- Green code was executed during the tests
- Yellow branch code executed, but only evaluated truthfully
- Orange branch code executed, but only evaluated falsely
- Red code did not execute during the tests



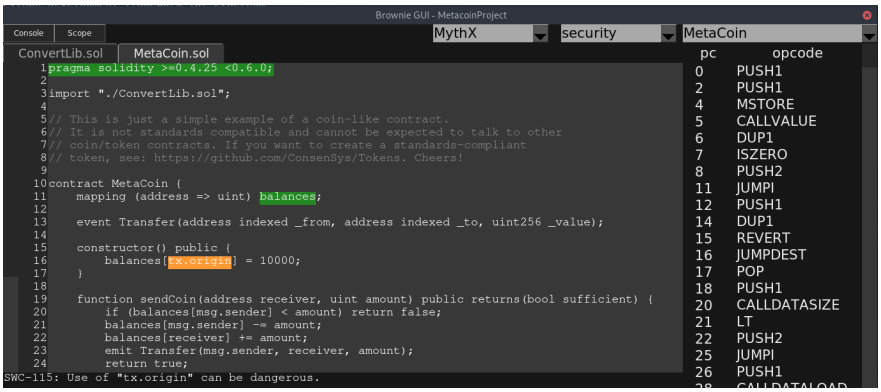
See *Coverage Evaluation* for more information.

2.7 Scanning for Security Vulnerabilities

To prevent vulnerabilities from being introduced to the code base, Brownie a includes plugin that integrates automated security scans using the MythX analysis API. Simply run `brownie analyze` on your compiled project directory. This will send the compiled build artifacts to MythX for analysis. By default no login is required and the analysis is going to be executed as a trial user. To access more vulnerability information, register for free on the MythX website and pass your login data via environment variables or command line arguments.

```
Brownie v1.0.0 - Python development framework for Ethereum

Usage: brownie analyze [options] [--async | --interval=<sec>]

Options:
--gui                       Launch the Brownie GUI after analysis
--full                      Perform a full scan (MythX Pro required)
--interval=<sec>            Result polling interval in seconds [default: 3]
--async                     Do not poll for results, print job IDs and exit
--access-token=<string>     The JWT access token from the MythX dashboard
--eth-address=<string>      The address of your MythX account
--password=<string>         The password of your MythX account
--help -h                   Display this message

Use the "analyze" command to submit your project to the MythX API for
smart contract security analysis.

To authenticate with the MythX API, it is recommended that you provide
the MythX JWT access token. It can be obtained on the MythX dashboard
site in the profile section. They should be passed through the environment
variable "MYTHX_ACCESS_TOKEN". If that is not possible, it can also be
passed explicitly with the respective command line option.

Alternatively, you have to provide a username/password combination. It
is recommended to pass them through the environment variables as
"MYTHX_ETH_ADDRESS" and "MYTHX_PASSWORD".

You can also choose to not authenticate and submit your analyses as a free
trial user. No registration required! To see your past analyses, get access
to deeper vulnerability detection, and a neat dashboard, register at
https://mythx.io/. Any questions? Hit up dominik.muhs@consensys.net or contact
us on the website!
```

Once the analysis is done, the vulnerabilities are stored in the `reports/` directory. With `brownie analyze --gui` the GUI can be started automatically once the analysis has finished.



CHAPTER 3

Installing Brownie

The easiest way to install Brownie is via pip.

```
$ pip install eth-brownie
```

You can also clone the github repository and use setuptools for the most up-to-date version.

```
$ python3 setup.py install
```

Once you have installed, type `brownie` to verify that it worked:

```
$ brownie
Brownie v1.0.0 - Python development framework for Ethereum

Usage:  brownie <command> [<args>...] [options <args>]
```

3.1 Dependencies

Brownie has the following dependencies:

- ganache-cli
- pip
- python3 version 3.6 or greater, python3-dev, python3-tk

As brownie relies on py-solc-x, you do not need solc installed locally but you must install all required solc dependencies.

CHAPTER 4

Initializing a New Project

The first step to using Brownie is to initialize a new project. To do this, create a new empty folder and then type:

```
$ brownie init
```

This will create the following project structure within the folder:

- `build/`: Compiled contracts and test data
- `contracts/`: Contract source code
- `reports/`: JSON report files for use in the *Viewing Coverage Data*
- `scripts/`: Scripts for deployment and interaction
- `tests/`: Scripts for testing your project
- `brownie-config.yaml`: Configuration file for the project

You can also initialize "Brownie mixes", simple templates to build your project upon. For many examples within the Brownie documentation we will use the token mix, which is a very basic ERC-20 implementation:

```
$ brownie bake token
```

This creates a new folder `token/` and deploys the project inside it.

CHAPTER 5

The Ethereum Package Manager

The Ethereum Package Manager (ethPM) is a decentralized package manager used to distribute EVM smart contracts and projects. It has similar goals to most package managers found in any given programming language:

- Easily import and build upon core ideas written by others.
- Distribute the ideas that you've written and/or deployed, making them easily consumable for tooling and the community at large.

At its core, an ethPM package is a JSON object containing the ABI, source code, bytecode, deployment data and any other information that combines together to compose the smart contract idea. The ethPM specification defines a schema to store all of this data in a structured JSON format, enabling quick and efficient transportation of smart contract ideas between tools and frameworks which support the specification.

Brownie supports ethPM, offering the following functionality:

1. ethPM packages may be used to obtain deployment data, providing easy *interaction with existing contracts* on the main-net or testnets.
2. Package source files may be *installed within a Brownie project*, to be inherited by existing contracts or used as a starting point when building something new.
3. Packages can be generated from Brownie projects and *released on ethPM registries*, for simple and verified distribution.

5.1 Registry URIs

To obtain an ethPM package, you must know both the package name and the address of the registry where it is available. The simplest way to communicate this information is through a registry URI. Registry URIs adhere to the following format:

For example, here is a registry URI for the popular OpenZeppelin Math package, served by the Snake Charmers Zeppelin registry:

5.2 Working with ethPM Packages

The `brownie ethpm` command-line interface is used to add and remove packages to a Brownie project, as well as to generate a package from a project.

5.2.1 Installing a Package

To install an ethPM package within a Brownie project:

```
$ brownie ethpm install [registry-uri]
```

This will add all of the package sources files into the project `contracts/` folder.

If a package contains a source with an identical filename to one in your project, Brownie raises a `FileExistsError` unless the contents of the two files are identical, or the `overwrite` flag is set to `True`.

5.2.2 Listing Installed Packages

To view a list of currently installed packages within a project:

```
$ brownie ethpm list
Brownie v1.2.0 - Python development framework for Ethereum

Found 2 installed packages:
  ├─access@1.0.0
  └─math@1.0.0
```

Any packages that are installed from a registry are also saved locally. To view a list of all locally available ethPM packages, and the registries they were downloaded from:

```
$ brownie ethpm all
Brownie v1.1.0 - Python development framework for Ethereum

erc1319://erc20.snakecharmers.eth
  └─dai-dai@1.0.0

erc1319://zeppelin.snakecharmers.eth
  ├─access@1.0.0
  ├─gns@1.0.0
  └─math@1.0.0
```

5.2.3 Removing a Package

Removing an installed package from a Brownie project will delete any of that package's sources files, as long as they are not also required by another package.

To remove a package, either delete all of it's source files or use the following command:

```
$ brownie ethpm remove [package-name]
```

5.2.4 Unlinking a Package

You may wish to install a package as a starting point upon which you build your own project, and in doing so make changes to the package sources. This will cause Brownie to flag the package as "modified" and raise warnings when performing certain actions. You can silence these warnings by unlinking the package - deleting Brownie's record that it is an ethPM package without removing the source files.

To unlink a package:

```
$ brownie ethpm remove [package-name]
```

5.3 Creating and Releasing a Package

Brownie allows you to generate an ethPM package from your project and publish it to a registry. Packages generated by Brownie will **always** include:

- All contract source files within the project
- The name, ABI, bytecode and compiler settings for each contract in the project

Depending upon the configuartion, they may also **optionally** include:

- Addresses of deployed contracts instances across each network
- References to other ethPM packages that this package requires

The process of releasing a package is:

1. Set all required fields within the `ethpm-config.yaml` configuration file.
2. Generate the package manifest and verify the contents.
3. Pin the manifest and sources to IPFS and publish the manifest URI to an ethPM registry.

Important: Ensure that all import statements within your source files use relative file paths (beginning with `./`). If you use absolute paths, your package is more likely to have namespace collisions when imported into other projects.

5.3.1 Step 1: Package Configuration Settings

To create a package you must first set all required fields within the `ethpm-config.yaml` file in the root folder of your project. If this file is not present in your project, the following command will generate it:

```
$ brownie ethpm all
```

Required Settings

The following settings must have a non-`null` value in order to generate a package.

`package_name`
: The `package_name` field defines a human readable name for the package. It must begin with a lowercase letter and be comprised of only lowercase letters, numeric characters, dashes and underscores. Package names must not exceed 255 characters in length.

  Link: ethPM specification: package name

version
The `version` field defines the version number for the package. All versions should conform to the semver versioning specificaion.

Link: ethPM specification: version

`settings.`**deployment_networks**
The `deployment_networks` field is a list of networks that should be included in the package's `deployments` field. The name of each network must correspond to that of a network listed in the *project configuration file*.

In order for a deployment to be included:

- *Persistence* must be enabled for that network
- The bytecode of the deployed contract must be identical to the bytecode generated from the source code currently present in the project's `contracts/` folder

You can use a wildcard `*` to include deployments on all networks, or `False` to not include any deployments.

Link: ethPM specification: deployments

`settings.`**include_dependencies**
The `include_dependencies` field is a boolean to indicate how package dependencies should be handled.

- if `True`, Brownie will generate a standalone package without any listed dependencies.
- if `False`, Brownie will list all package dependencies within the manifest, and only include as much data about them as is required by the `deployments` field.

Note that you cannot set `include_dependencies` to `False` while your package contains dependency source files that have been modified. In this situation you must first *unlink* the modified packages.

Link: ethPM specification: build dependencies

Optional Settings

meta
The `meta` field, and all it's subfields, provides metadata about the package. This data is not integral for package installation, but may be important or convenient to provide.

Any fields that are left blank will be omitted. You can also add additional fields, they will be included within the package.

Link: ethPM specification: package meta

Example Configuration

Here is an example configuration for `ethpm-config.yaml`:

```
# required fields
package_name: nftoken
version: 1.0.1
settings:
  deployment_networks:
    - mainnet
include_dependencies: false

# optional fields
meta:
```

(continues on next page)

(continued from previous page)

```
  description: A non-fungible implementation of the ERC20 standard, allowing scalable␣
↪NFT transfers with fixed gas costs.
  authors:
    - Ben Hauser
    - Gabriel Shapiro
  license: MIT
  keywords:
    - ERC20
    - ERC721
    - NFT
  links:
    repository: https://github.com/iamdefinitelyahuman/nftoken
```

5.3.2 Step 2: Creating the Manifest

Once you have set the required fields in the configuration file, you can create a manifest with the following command:

```
$ brownie ethpm create
```

The manifest is saved locally as `manifest.json` in the project root folder. Note that this saved copy is not tightly packed and so does not strictly adhere the ethPM specification. This is not the final copy to be pinned to IPFS, rather it is a human-readable version that you can use to verify it's contents before releasing.

Once you have confirmed that the included fields are consistent with what you would like to publish, you are ready to release.

5.3.3 Step 3: Releasing the Package

There are two steps in releasing a package:

1. Pinning the manifest and related sources to IPFS.

   Brownie uses Infura's public IPFS gateway to interact with IPFS. Note that pinning files to IPFS can be a very slow proess. If you receive a timeout error, simply repeat the request. Files that have been successfully pinned will not need to be re-pinned.

2. Calling the release function of an ethPM registry with details of the package.

   Brownie broadcasts this transaction on the "mainnet" network as defined in the *project configuration file*. The account that you send the transaction from must be approved to call `release` in the registry, otherwise it will fail. Depending on your use case you may wish to run your own registry, or include your files within an existing one. See the ethPM documentation for more information.

To release a package:

```
$ brownie ethpm release [registry] [account]
```

You must include the following arguments:

- `registry`: the address of an ethPM registry on the main-net
- `account`: the address that the transaction is sent from. It can be given as an alias to a local account, or as a hex string if the address is unlocked within the connected node.

Once the package is successfully released, Brownie provides you with a registry URI that you can share with others so they can easily access your package:

```
$ brownie ethpm release erc20.snakecharmers.eth registry_owner
Brownie v1.1.0 - Python development framework for Ethereum

Generating manifest and pinning assets to IPFS...
Pinning "NFToken.sol"...
Pinning "NFMintable.sol"...
Pinning manifest...

Releasing nftoken@1.0.1 on "erc20.snakecharmers.eth"...
Enter the password for this account: *****

SUCCESS: nftoken@1.0.1 has been released!

URI: erc1319://erc20.snakecharmers.eth:1/nftoken@1.0.1
```

5.4 Interacting with Package Deployments

You can load an entire package as a *Project* object, which includes *Contract* instances for any contracts deployed on the currently active network:

```
>>> from brownie.project import from_ethpm
>>> maker = from_ethpm("erc1319://erc20.snakecharmers.eth:1/dai-dai@1.0.0")
>>> maker
<TempProject object 'dai-dai'>
>>> maker.dict()
{
    'DSToken': [<DSToken Contract object '0x89d24A6b4CcB1B6fAA2625fE562bDD9a23260359'>
↪]
}
```

Or, create a *Contract* object to interact with a deployed instance of a specific contract within a package:

```
>>> from brownie import network, Contract
>>> network.connect('mainnet')
>>> ds = Contract("DSToken", manifest_uri="erc1319://erc20.snakecharmers.eth:1/dai-
↪dai@1.0.0")
>>> ds
<DSToken Contract object '0x89d24A6b4CcB1B6fAA2625fE562bDD9a23260359'>
```

If the package does not include deployment information for the currently active network, a `ContractNotFound` exception is raised.

CHAPTER 6

Compiling Contracts

To compile a project:

```
$ brownie compile
```

Each time the compiler runs, Brownie compares hashes of the contract source code against the existing compiled versions. If a contract has not changed it will not be recompiled. If you wish to force a recompile of the entire project, use `brownie compile --all`.

Note: All of a project's contract sources must be placed inside the `contracts/` folder. Attempting to import sources from outside this folder will result in a compiler error.

6.1 Compiler Settings

Settings for the compiler are found in `brownie-config.yaml`:

```
solc:
    version: 0.5.10
    evm_version: null
    optimize: true
    runs: 200
    minify_source: false
```

Modifying any compiler settings will result in a full recompile of the project.

6.1.1 Setting the Compiler Version

Note: Brownie supports Solidity versions `>=0.4.22`.

If a compiler version is set in the configuration file, all contracts in the project are compiled using that version. It is installed automatically if not already present. The version should be given as a string in the format `0.x.x`.

If the version is set to `null`, Brownie looks at the version pragma of each contract and uses the latest matching compiler version that has been installed. If no matching version is found, the most recent release is installed.

Setting the version via pragma allows you to use multiple versions in a single project. When doing so, you may encounter compiler errors when a contract imports another contract that is meant to compile on a higher version. A good practice in this situation is to import interfaces rather than actual contracts when possible, and set all interface pragmas as `>=0.4.22`.

6.1.2 The EVM Version

By default, `evm_version` is set to `null`. Brownie uses `byzantium` when compiling versions `<=0.5.4` and `petersburg` for `>=0.5.5`.

If you wish to use a newer compiler version on a network that has not yet forked you can set the EVM version manually. Valid options are `byzantium`, `constantinople` and `petersburg`.

See the Solidity documentation for more info on the different EVM versions.

6.1.3 Compiler Optimization

Compiler optimization is enabled by default. Coverage evaluation was designed using optimized contracts - there is no need to disable it during testing.

See the Solidity documentation for more info on the `solc` optimizer.

6.1.4 Source Minification

If `minify_source` is `true`, the contract source is minified before compiling. Each time Brownie is loaded it will then minify the current source code before checking the hashes to determine if a recompile is necessary. This allows you to modify code formatting and comments without triggering a recompile, at the cost of increased load times from recalculating source offsets.

6.2 Installing the Compiler

If you wish to manually install a different version of `solc`:

```
>>> from brownie.project.compiler import install_solc
>>> install_solc("0.5.10")
```

CHAPTER 7

Project Interaction via the Console

The console is useful when you want to interact directly with contracts deployed on a non-local chain, or for quick testing as you develop. It's also a great starting point to familiarize yourself with Brownie's functionality.

The console feels very similar to a regular python interpreter. From inside a project folder, load it by typing:

```
$ brownie console
```

Brownie will compile the contracts, launch or attach to *The Local RPC Client*, and then give you a command prompt. From here you may interact with the network with the full range of functionality offered by the *Brownie API*.

Hint: You can call the builtin `dir` method to see available methods and attributes for any class. Classes, methods and attributes are highlighted in different colors.

You can also call `help` on most classes and methods to get detailed information on how they work.

7.1 Accounts

The *Accounts* container (available as `accounts` or just `a`) allows you to access all your local accounts.

```
>>> accounts
['0xC0BcE0346d4d93e30008A1FE83a2Cf8CfB9Ed301',
 ↪'0xf414d65808f5f59aE156E51B97f98094888e7d92',
 ↪'0x055f1c2c9334a4e57ACF2C4d7ff95d03CA7d6741',
 ↪'0x1B63B4495934bC1D6Cb827f7a9835d316cdBB332',
 ↪'0x303E8684b9992CdFA6e9C423e92989056b6FC04b',
 ↪'0x5eC14fDc4b52dE45837B7EC8016944f75fF42209',
 ↪'0x22162F0D8Fd490Bde6Ffc9425472941a1a59348a',
 ↪'0x1DA0dcC27950F6070c07F71d1dE881c3C67CEAab',
 ↪'0xa4c7f832254eE658E650855f1b529b2d01C92359',
 ↪'0x275CAe3b8761CEdc5b265F3241d07d2fEc51C0d8']
>>> accounts[0]
```

(continues on next page)

(continued from previous page)

```
<Account object '0xC0BcE0346d4d93e30008A1FE83a2Cf8CfB9Ed301'>
```

Each individual account is represented by an *Account* object that can perform actions such as querying a balance or sending ETH.

```
>>> accounts[0]
<Account object '0xC0BcE0346d4d93e30008A1FE83a2Cf8CfB9Ed301'>
>>> dir(accounts[0])
[address, balance, deploy, estimate_gas, nonce, transfer]
>>> accounts[1].balance()
100000000000000000000
>>> accounts[0].transfer(accounts[1], "10 ether")

Transaction sent: 0x124ba3f9f9e5a8c5e7e559390bebf8dfca998ef32130ddd114b7858f255f6369
Transaction confirmed - block: 1   gas spent: 21000
<Transaction object
↪'0x124ba3f9f9e5a8c5e7e559390bebf8dfca998ef32130ddd114b7858f255f6369'>
>>> accounts[1].balance()
110000000000000000000
```

You can import accounts with `accounts.add`, which takes a private key as the only argument. If you do not enter a private key one is randomly generated.

```
>>> len(accounts)
10
>>> accounts.add("ce7594141801cf9b81b7ccb09e30395fc9e9e5940b1c01eed6434588bd726f94")
<Account object '0x405De4AeCb9c1cE75152F82F956E09F4eda3b351'>
>>> len(accounts)
11
>>> accounts[10]
<Account object '0x405De4AeCb9c1cE75152F82F956E09F4eda3b351'>
>>> accounts.add()
<Account object '0xc1b3a737C147E8d85f600F8082f42F0511ED5278'>
>>> len(accounts)
12
```

Imported accounts may be saved with an identifier and then loaded again at a later date. Account data is saved in a standard json keystore file that is compatible with most wallets.

```
>>> accounts.add()
<LocalAccount object '0xa9c2DD830DfFE8934fEb0A93BAbcb6e823e1FF05'>
>>> accounts[-1].save('my_account')
Enter the password to encrypt this account with:
Saved to brownie/data/accounts/my_account.json
>>> accounts.load('my_account')
Enter the password for this account:
<LocalAccount object '0xa9c2DD830DfFE8934fEb0A93BAbcb6e823e1FF05'>
```

7.2 Contracts

7.2.1 Deploying

Each deployable contract and library has a *ContractContainer* class, used to deploy new contracts and access already existing ones.

To deploy a contract, call the `ContractContainer.deploy` method with the constructor arguments, with a dictionary of transaction parameters as the final argument. The dictionary must include a `from` value that specifies the `Account` to deploy the contract from.

A *Contract* object is returned and also appended to the `ContractContainer`.

```
>>> type(Token)
<class 'brownie.network.contract.ContractContainer'>
>>> Token
[]
>>> Token.deploy
<ContractConstructor object 'Token.constructor(string _symbol, string _name, uint256 _
↪decimals, uint256 _totalSupply)'>
>>> t = Token.deploy("Test Token", "TST", 18, "1000 ether", {'from': accounts[1]})

Transaction sent: 0x2e3cab83342edda14141714ced002e1326ecd8cded4cd0cf14b2f037b690b976
Transaction confirmed - block: 1   gas spent: 594186
Contract deployed at: 0x5419710735c2D6c3e4db8F30EF2d361F70a4b380
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>>
>>> t
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>> Token
[<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>]
>>> Token[0]
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
```

Alternatively, you can deploy from `Account` with the `ContractContainer` as the first argument.

```
>>> Token
[]
>>> t = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")

Transaction sent: 0x2e3cab83342edda14141714ced002e1326ecd8cded4cd0cf14b2f037b690b976
Transaction confirmed - block: 1   gas spent: 594186
Contract deployed at: 0x5419710735c2D6c3e4db8F30EF2d361F70a4b380
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>>
>>> t
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>> Token
[<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>]
>>> Token[0]
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
```

You can also use `ContractContainer.at` to create a new `Contract` object for an already deployed contract.

```
>>> Token.at("0x5419710735c2D6c3e4db8F30EF2d361F70a4b380")
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
```

7.2.2 Unlinked Libraries

If a contract requires a library, Brownie will automatically link to the most recently deployed one. If the required library has not been deployed yet an `UndeployedLibrary` exception is raised.

```
>>> accounts[0].deploy(MetaCoin)
  File "brownie/network/contract.py", line 167, in __call__
    f"Contract requires '{library}' library but it has not been deployed yet"
UndeployedLibrary: Contract requires 'ConvertLib' library but it has not been␣
↪deployed yet

>>> accounts[0].deploy(ConvertLib)
Transaction sent: 0xff3f5cff35c68a73658ad367850b6fa34783b4d59026520bd61b72b6613d871c
ConvertLib.constructor confirmed - block: 1   gas used: 95101 (48.74%)
ConvertLib deployed at: 0x08c4C7F19200d5636A1665f6048105b0686DFf01
<ConvertLib Contract object '0x08c4C7F19200d5636A1665f6048105b0686DFf01'>

>>> accounts[0].deploy(MetaCoin)
Transaction sent: 0xd0969b36819337fc3bac27194c1ff0294dd65da8f57c729b5efd7d256b9ecfb3
MetaCoin.constructor confirmed - block: 2   gas used: 231857 (69.87%)
MetaCoin deployed at: 0x8954d0c17F3056A6C98c7A6056C63aBFD3e8FA6f
<MetaCoin Contract object '0x8954d0c17F3056A6C98c7A6056C63aBFD3e8FA6f'>
```

7.2.3 Accessing Contract Methods

External and public contract methods are callable from the `Contract` object via class methods of the same name. Arguments given to these objects are converted using the methods outlined in the *Type Conversions* section of the API documentation.

```
>>> Token[0].transfer
<ContractTx object 'transfer(address _to, uint256 _value)'>
>>> Token[0].balanceOf
<ContractCall object 'balanceOf(address _owner)'>
```

Transactions

For state changing contract methods, the related class method is *ContractTx*. Calls to this object perform a transaction and return a *TransactionReceipt* object. If you wish to call the contract method without a transaction, use the `ContractTx.call` method.

For transactions you can optionally include a dictionary of transaction parameters as the final argument. If you omit this or do not specify a `from` value, the transaction will be sent from the same address that deployed the contract.

```
>>> Token[0].transfer(accounts[1], "1 ether", {'from': accounts[0]})

Transaction sent: 0x6e557594e657faf1270235bf4b3f27be7f5a3cb8a9c981cfffb12133cbaa165e
Token.transfer confirmed - block: 4   gas used: 51019 (33.78%)
<Transaction object
↪'0x6e557594e657faf1270235bf4b3f27be7f5a3cb8a9c981cfffb12133cbaa165e'>
>>> Token[0].transfer.call(accounts[1], "1 ether", {'from': accounts[0]})
True
```

Calls

If the contract method has a state mutability of `view` or `pure`, the related class method type is *ContractCall*. Calling this object will call the contract method and return the result. If you wish to access the method via a transaction you can use `ContractCall.transact`.

```
>>> Token[0].balanceOf(accounts[0])
10000000000000000000
>>> tx = Token[0].balanceOf.transact(accounts[0])

Transaction sent: 0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8
Token.balanceOf confirmed - block: 3   gas used: 23222 (18.85%)
<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>
>>> tx.return_value
10000000000000000000
```

7.3 Ether Values

Brownie uses the *Wei* class when a value is meant to represent an amount of ether. `Wei` is a subclass of `int` that converts strings, scientific notation and hex strings into wei denominated integers:

```
>>> Wei("1 ether")
1000000000000000000
>>> Wei("12.49 gwei")
12490000000
>>> Wei("0.029 shannon")
29000000
>>> Wei(8.38e32)
838000000000000000000000000000000
```

It also converts other values to `Wei` before performing comparisons, addition or subtraction:

```
>>> Wei(1e18) == "1 ether"
True
>>> Wei("1 ether") < "2 ether"
True
>>> Wei("1 ether") - "0.75 ether"
250000000000000000
```

Whenever a Brownie method takes an input referring to an amount of ether, the given value is converted to `Wei`. Balances and `uint`/`int` values returned in contract calls and events are given in `Wei`.

```
>>> accounts[0].balance()
100000000000000000000
>>> type(accounts[0].balance())
<class 'brownie.convert.Wei'>
```

7.4 Transactions

Each transaction returns a *TransactionReceipt* object. This object contains all relevant information about the transaction, as well as various methods to aid in debugging if it reverted.

```
>>> tx = Token[0].transfer(accounts[1], "1 ether", {'from': accounts[0]})

Transaction sent: 0xa7616a96ef571f1791586f570017b37f4db9decb1a5f7888299a035653e8b44b
Token.transfer confirmed - block: 2   gas used: 51019 (33.78%)
<Transaction object
↪'0xa7616a96ef571f1791586f570017b37f4db9decb1a5f7888299a035653e8b44b'>
```

(continues on next page)

(continued from previous page)

```
>>> tx
<Transaction object
↪'0xa7616a96ef571f1791586f570017b37f4db9decb1a5f7888299a035653e8b44b'>
```

To get human-readable information on a transaction, use `TransactionReceipt.info()`.

```
>>> tx.info()

Transaction was Mined
---------------------
Tx Hash: 0xa7616a96ef571f1791586f570017b37f4db9decb1a5f7888299a035653e8b44b
From: 0x4FE357AdBdB4C6C37164C54640851D6bff9296C8
To: 0xDd18d6475A7C71Ee33CEBE730a905DbBd89945a1
Value: 0
Function: Token.transfer
Block: 2
Gas Used: 51019 / 151019 (33.8%)

Events In This Transaction
--------------------------
Transfer
    from: 0x4fe357adbdb4c6c37164c54640851d6bff9296c8
    to: 0xfae9bc8a468ee0d8c84ec00c8345377710e0f0bb
    value: 1000000000000000000
```

7.4.1 Accessing Event Data

Events are stored at `TransactionReceipt.events` using the *EventDict* class. `EventDict` hybrid container with both dict-like and list-like properties.

Note: Event data is still available when a transaction reverts.

```
>>> tx.events
{
    'CountryModified': [
        {
            'country': 1,
            'limits': (0,0,0,0,0,0,0,0),
            'minrating': 1,
            'permitted': True
        },
        {
            'country': 2,
            'limits': (0,0,0,0,0,0,0,0),
            'minrating': 1,
            'permitted': True
        }
    ],
    'MultiSigCallApproved': [
        {
            'callHash':
↪"0x0013ae2e37373648c5161d81ca78d84e599f6207ad689693d6e5938c3ae4031d",
            'callSignature': "0xa513efa4",
```

(continues on next page)

(continued from previous page)

```
            'caller': "0xF9c1fd2f0452FA1c60B15f29cA3250DfcB1081b9",
            'id': "0x8be1198d7f1848ebeddb3f807146ce7d26e63d3b6715f27697428ddb52db9b63"
        }
    ]
}
```

Use it as a dict for looking at specific events when the sequence they are fired in does not matter:

```
>>> len(tx.events)
3
>>> len(tx.events['CountryModified'])
2
>>> 'MultiSigCallApproved' in tx.events
True
>>> tx.events['MultiSigCallApproved']
{
    'callHash': "0x0013ae2e37373648c5161d81ca78d84e599f6207ad689693d6e5938c3ae4031d",
    'callSignature': "0xa513efa4",
    'caller': "0xF9c1fd2f0452FA1c60B15f29cA3250DfcB1081b9",
    'id': "0x8be1198d7f1848ebeddb3f807146ce7d26e63d3b6715f27697428ddb52db9b63"
}
```

Or as a list when the sequence is important, or more than one event of the same type was fired:

```
>>> tx.events[1].name
'CountryModified'
>>> tx.events[1]
{
    'country': 1,
    'limits': (0,0,0,0,0,0,0,0),
    'minrating': 1,
    'permitted': True
}
```

7.4.2 Reverted Transactions

When a transaction reverts in the console you are still returned a `TransactionReceipt`, but it will show as reverted. If an error string is given, it will be displayed in brackets and highlighted in red.

```
>>> tx = Token[0].transfer(accounts[1], "1 ether", {'from': accounts[3]})

Transaction sent: 0x5ff198f3a52250856f24792889b5251c120a9ecfb8d224549cb97c465c04262a
Token.transfer confirmed (Insufficient Balance) - block: 2   gas used: 23858 (19.26%)
<Transaction object
↪'0x5ff198f3a52250856f24792889b5251c120a9ecfb8d224549cb97c465c04262a'>
```

You can use `TransactionReceipt.error()` to see the section of the source code that caused the revert:

```
>>> tx.error()
File "contracts/Token.sol", line 62, in function transfer
    }

    function transfer(address _to, uint256 _value) public returns (bool) {
        require(balances[msg.sender] >= _value, "Insufficient Balance");
        balances[msg.sender] = balances[msg.sender].sub(_value);
```

(continues on next page)

(continued from previous page)

```
        balances[_to] = balances[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
```

Or `TransactionReceipt.traceback()` for a full traceback leading up to the revert:

```
>>> tx.traceback()
Traceback for '0x9542e92a904e9d345def311ea52f22c3191816c6feaf7286f9b48081ab255ffa':
Trace step 99, program counter 1699:
  File "contracts/Token.sol", line 67, in Token.transfer:
    balances[msg.sender] = balances[msg.sender].sub(_value);
Trace step 110, program counter 1909:
  File "contracts/SafeMath.sol", line 9, in SafeMath.sub:
    require(b <= a);
```

You can also call `TransactionReceipt.call_trace()` to see all the contract jumps, internal and external, that occured during the transaction. This method is available for all transactions, not only those that reverted.

```
>>> tx = Token[0].transferFrom(accounts[2], accounts[3], "10000 ether")

Transaction sent: 0x0d96e8ceb555616fca79dd9d07971a9148295777bb767f9aa5b34ede483c9753
Token.transferFrom confirmed (reverted) - block: 4   gas used: 25425 (26.42%)

>>> tx.call_trace()
Call trace for '0x0d96e8ceb555616fca79dd9d07971a9148295777bb767f9aa5b34ede483c9753':
Token.transfer 0:244  (0x4A32104371b05837F2A36dF6D850FA33A92a178D)
   Token.transfer 72:226
   SafeMath.sub 100:114
   SafeMath.add 149:165
```

See *Debugging Tools* for more information on debugging reverted transactions.

7.4.3 Unconfirmed Transactions

If you are working on a chain where blocks are not mined automatically, you can press `CTRL-C` while waiting for a transaction to confirm and return to the console. You will still be returned a `TransactionReceipt`, however it will be marked as pending (printed in yellow). A notification is displayed when the transaction confirms.

If you send another transaction from the same account before the previous one has confirmed, it will still broadcast with the next sequential nonce.

7.4.4 Accessing Historic Transactions

The *brownie.network.state* object, available as `history`, holds all the transactions that have been broadcasted. You can use it to access `TransactionReceipt` objects if you did not assign them a unique name when making the call.

```
>>> history
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>, <Transaction␣
↪object '0xa7616a96ef571f1791586f570017b37f4db9decb1a5f7888299a035653e8b44b'>]
```

7.5 The Local Test Environment

Brownie is designed to use ganache-cli as a local development environment. Functionality such as snapshotting and time travel is accessible via the *Rpc* object, available as `rpc`:

```
>>> rpc
<brownie.network.rpc.Rpc object at 0x7f720f65fd68>
```

`Rpc` is useful when you need to perform tests dependent on time:

```
>>> rpc.time()
1557151189
>>> rpc.sleep(100)
>>> rpc.time()
1557151289
```

Or for returning to a previous state during tests:

```
>>> rpc.snapshot()
Snapshot taken at block height 4
>>> accounts[0].balance()
100000000000000000000
>>> accounts[0].transfer(accounts[1], "10 ether")

Transaction sent: 0xd5d3b40eb298dfc48721807935eda48d03916a3f48b51f20bcded372113e1dca
Transaction confirmed - block: 5   gas used: 21000 (100.00%)
<Transaction object
↪'0xd5d3b40eb298dfc48721807935eda48d03916a3f48b51f20bcded372113e1dca'>

>>> accounts[0].balance()
89999580000000000000
>>> rpc.revert()
Block height reverted to 4
>>> accounts[0].balance()
100000000000000000000
```

See *The Local RPC Client* for more information on how to use `Rpc`.

CHAPTER 8

Unit Testing with Pytest

Brownie utilizes the `pytest` framework for unit testing. You may wish to view the pytest documentation if you have not used it previously.

Test scripts are stored in the `tests/` folder of your project. To run the complete test suite:

```
$ pytest tests
```

8.1 Brownie Pytest Fixtures

Brownie provides pytest fixtures which allow you to interact with your project. To use a fixture, add an argument with the same name to the inputs of your test function.

8.1.1 Session Fixtures

These fixtures provide quick access to Brownie objects that are frequently used during testing. If you are unfamiliar with these objects, you may wish to read *Project Interaction via the Console*.

accounts
: Yields an *Accounts* container for the active project, used to interact with your local Eth accounts.

```
def test_account_balance(accounts):
    assert accounts[0].balance() == "100 ether"
```

a
: Short form of the `accounts` fixture.

```
def test_account_balance(a):
    assert a[0].balance() == "100 ether"
```

history
: Yields a *TxHistory* container for the active project, used to access transaction data.

```
def test_account_balance(accounts, history):
    accounts[0].transfer(accounts[1], "10 ether")
    assert len(history) == 1
```

rpc

Yields an *Rpc* object, used for interacting with the local test chain.

```
def test_account_balance(accounts, rpc):
    balance = accounts[1].balance()
    accounts[0].transfer(accounts[1], "10 ether")
    assert accounts[1].balance() == balance + "10 ether"
    rpc.reset()
    assert accounts[1].balance() == balance
```

web3

Yields a *Web3* object.

```
def test_account_balance(accounts, web3):
    height = web3.eth.blockNumber
    accounts[0].transfer(accounts[1], "10 ether")
    assert web3.eth.blockNumber == height + 1
```

If you are accessing the same object across many tests in the same module, you may prefer to import it from the `brownie` package instead of accessing it via fixtures. The following two examples will work identically:

```
def test_account_balance(accounts):
    assert accounts[0].balance() == "100 ether"

def test_account_nonce(accounts):
    assert accounts[0].nonce == 0
```

```
from brownie import accounts

def test_account_balance():
    assert accounts[0].balance() == "100 ether"

def test_account_nonce():
    assert accounts[0].nonce == 0
```

8.1.2 Contract Fixtures

Brownie creates dynamically named fixtures to access each *ContractContainer* object within a project. Fixtures are generated for all deployable contracts and libraries.

For example - if your project contains a contract named `Token`, there will be a `Token` fixture available.

```
from brownie import accounts

def test_token_deploys(Token):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")
    assert token.name() == "Test Token"
```

8.2 Handling Reverted Transactions

When running tests, transactions that revert raise a `VirtualMachineError` exception. To write assertions around this you can use `pytest.reverts` as a context manager. It functions very similarly to pytest.raises.

```
import pytest

def test_transfer_reverts(Token):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")
    with pytest.reverts():
        token.transfer(accounts[1], "2000 ether", {'from': accounts[0]})
```

You may optionally supply a string as an argument. If given, the error string returned by the transaction must match it in order for the test to pass.

```
import pytest

def test_transfer_reverts(Token):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")
    with pytest.reverts("Insufficient Balance"):
        token.transfer(accounts[1], "9001 ether", {'from': accounts[0]})
```

8.2.1 Developer Revert Comments

Each revert string adds a minimum 20000 gas to your contract deployment cost, and increases the cost for a function to execute. Including a revert string for every `require` and `revert` statement is often impractical and sometimes simply not possible due to the block gas limit.

For this reason, Brownie allows you to include revert strings as source code comments that are not included in the bytecode but still accessible via `TransactionReceipt.revert_msg`. You write tests that target a specific `require` or `revert` statement without increasing gas costs.

Revert string comments must begin with `// dev:` in order for Brownie to recognize them. Priority is always given to compiled revert strings. Some examples:

```
function revertExamples(uint a) external {
    require(a != 2, "is two");
    require(a != 3); // dev: is three
    require(a != 4, "cannot be four"); // dev: is four
    require(a != 5); // is five
}
```

- Line 2 will use the given revert string `"is two"`
- Line 3 will substitute in the string supplied on the comments: `"dev:  is three"`
- Line 4 will use the given string `"cannot be four"` and ignore the subsitution string.
- Line 5 will have no revert string. The comment did not begin with `"dev:"` and so is ignored.

If the above function is executed in the console:

```
>>> tx = test.revertExamples(3)
Transaction sent: 0xd31c1c8db46a5bf2d3be822778c767e1b12e0257152fcc14dcf7e4a942793cb4
test.revertExamples confirmed (dev: is three) - block: 2   gas used: 31337 (6.66%)
<Transaction object
↪'0xd31c1c8db46a5bf2d3be822778c767e1b12e0257152fcc14dcf7e4a942793cb4'>
```

(continues on next page)

(continued from previous page)

```
>>> tx.revert_msg
'dev: is three'
```

8.3 Isolating Tests

8.3.1 Module Isolation

In most cases you will want to isolate your tests from one another by resetting the local environment in between modules. Brownie provides the `module_isolation` fixture to accomplish this. This fixture calls `Rpc.reset()` before and after completion of the module, ensuring a clean environment for this module and that the results of it will not affect subsequent modules.

The `module_isolation` fixture is **always the first module-scoped fixture to execute**.

To apply the fixture to all tests in a module, include the following fixture within the module:

```
import pytest

@pytest.fixture(scope="module", autouse=True)
def setup(module_isolation):
    pass
```

You can also place this fixture in a conftest.py file to apply it across many modules.

8.3.2 Function Isolation

Brownie provides the function scoped `fn_isolation` fixture, used to isolate individual test functions. This fixture takes a snapshot of the local environment before running each test, and revert to it after the test completes.

In the example below, the assert statement in `test_isolated` passes because the state is reverted in between tests. If you remove the `isolation` fixture the test will fail.

```
import pytest

@pytest.fixture(autouse=True)
def isolation(fn_isolation):
    pass

def test_transfer(accounts):
    accounts[0].transfer(accounts[1], "10 ether")
    assert accounts[1].balance() == "110 ether"

def test_isolated(accounts):
    assert accounts[1].balance() == "100 ether"
```

8.3.3 Defining a Shared Initial State

The `fn_isolation` fixture is **always the first function-scoped fixture to execute**. A common pattern is to include one or more module-scoped setup fixtures that define the initial test conditions, and then use `fn_isolation` to revert to this base state at the start of each test. For example:

```
import pytest

@pytest.fixture(scope="module", autouse=True)
def token(Token, accounts):
    t = accounts[0].deploy(Token, "Test Token", "TST", 18, 1000)
    yield t

@pytest.fixture(autouse=True)
def isolation(fn_isolation):
    pass

def test_transfer(token, accounts):
    token.transfer(accounts[1], 100, {'from': accounts[0]})
    assert token.balanceOf(accounts[0]) == 900

def test_chain_reverted(token):
    assert token.balanceOf(accounts[0]) == 1000
```

The sequence of events in the above example is:

1. The setup phase of `module_isolation` runs, resetting the local environment.
2. The module-scoped `token` fixture runs, deploying a `Token` contract with a total supply of 1000 tokens.
3. The setup phase of the function-scoped `fn_isolation` fixture runs. A snapshot of the blockchain is taken.
4. `test_transfer` runs, transferring 100 tokens from `accounts[0]` to `accounts[1]`
5. The teardown phase of `fn_isolation` runs. The blockchain is reverted to it's state before `test_transfer`.
6. The setup phase of the `fn_isolation` fixture runs again. Another snapshot is taken - identical to the previous one.
7. `test_chain_reverted` runs. The assert statement passes because of the `fn_isolation` fixture.
8. The teardown phase of `fn_isolation` runs. The blockchain is reverted to it's state before `test_chain_reverted`.
9. The teardown phase of `module_isolation` runs, resetting the local environment.

Additionally, remember that **module-scoped fixtures will always execute prior to function-scoped**. New module-scoped fixtures can be introduced part way through a module, and in this way modify the setup snapshot. Expanding on the previous example:

```
import pytest

@pytest.fixture(scope="module", autouse=True)
def token(Token, accounts):
    t = accounts[0].deploy(Token, "Test Token", "TST", 18, 1000)
    yield t

@pytest.fixture(scope="module")
def transfer_tokens(token, accounts):
    token.transfer(accounts[1], 100, {'from': accounts[0]})

@pytest.fixture(autouse=True)
def isolation(fn_isolation):
    pass

```

(continues on next page)

(continued from previous page)

```
16 def test_transfer(token, accounts):
17     token.transfer(accounts[1], 100, {'from': accounts[0]})
18     assert token.balanceOf(accounts[0]) == 900
19
20 def test_chain_reverted(token):
21     assert token.balanceOf(accounts[0]) == 1000
22
23 def test_module_fixture_transfer(transfer_tokens, token):
24     token.transfer(accounts[1], 50, {'from': accounts[0]})
25     assert token.balanceOf(accounts[0]) == 850
26
27 def test_snapshot_altered(token):
28     assert token.balanceOf(accounts[0]) == 900
```

Let's look at the sequence of events, starting from the teardown of `test_chain_reverted` (step 8 in the previous example):

8. The teardown phase of `fn_isolation` runs. The blockchain is reverted to it's state before `test_chain_reverted`.
9. The module-scoped `transfer_tokens` fixture runs. 100 tokens are transferred to `accounts[1]`.
10. The setup phase of `fn_isolation` runs. A new snapshot is taken, this time including the transfer performed by `transfer_tokens`.
11. `test_module_fixture_transfer` runs. 50 tokens are transferred and the assert statement passes.
12. The teardown phase of `fn_isolation` runs. The state is reverted to immediately before `test_module_fixture_transfer` was run.
13. The setup phase of `fn_isolation` runs. Another snapshot is taken - identical to the previous one.
14. `test_snapshot_altered` runs. The assertion passes.
15. `fn_isolation` and then `module_isolation` perform their final teardowns. The local environment is reset and the module is completed.

8.4 Coverage Evaluation

Test coverage is calculated by generating a map of opcodes associated with each statement and branch of the source code, and then analyzing the stack trace of each transaction to see which opcodes executed. See "Evaluating Solidity Code Coverage via Opcode Tracing" for a more detailed explanation of how coverage evaluation works.

During coverage analysis, all contract calls are executed as transactions. This gives a more accurate coverage picture by allowing analysis of methods that are typically non-state changing. A snapshot is taken before each of these calls-as-transactions and the state is reverted immediately after, to ensure that the outcome of the test is not affected. For tests that involve many calls this can result in significantly slower execution time.

Note: Coverage analysis is stored on a per-transaction basis. If you repeat an identical transaction, Brownie will not have to analyze it. It is good to keep this in mind when designing setup fixtures, especially for large test suites.

8.4.1 Coverage Fixtures

Brownie provides fixtures that allow you to alter the behaviour of tests when coverage evaluation is active. They are useful for tests with many repetitive functions, to avoid the slowdown caused by `debug_traceTransaction` queries.

Both of these fixtures are function-scoped.

`no_call_coverage`
Coverage evaluation will not be performed on called contact methods during this test.

```
import pytest

@pytest.fixture(scope="module", autouse=True)
def token(Token, accounts):
    t = accounts[0].deploy(Token, "Test Token", "TST", 18, 1000)
    t.transfer(accounts[1], 100, {'from': accounts[0]})
    yield t

def test_normal(token):
    # this call is handled as a transaction, coverage is evaluated
    assert token.balanceOf(accounts[0]) == 900

def test_no_call_cov(Token, no_call_coverage):
    # this call happens normally, no coverage evaluation
    assert token.balanceOf(accounts[1]) == 100
```

`skip_coverage`
This test will be skipped if coverage evaluation is active.

8.5 Running Tests

Test scripts are stored in the `tests/` folder. Test discovery follows the standard pytest discovery rules.

To run the complete test suite:

```
$ pytest tests
```

Or to run a specific test:

```
$ pytest tests/test_transfer.py
```

Note: Because of Brownie's dynamically named contract fixtures, you cannot run `pytest` outside of the Brownie project folder.

Test results are saved at `build/tests.json`. This file holds the results of each test, coverage analysis data, and hashes that are used to determine if any related files have changed since the tests last ran. If you abort test execution early via a `KeyboardInterrupt`, results are only be saved for modules that fully completed.

8.5.1 Only Running Updated Tests

After the test suite has been run once, you can use the `--update` flag to only repeat tests where changes have occured:

```
$ pytest tests --update
```

A module must use the `module_isolation` or `fn_isolation` fixture in every test function in order to be skipped in this way.

The `pytest` console output will represent skipped tests with an "s", but it will be colored green or red to indicate if the test passed when it last ran.

If coverage analysis is also active, tests that previously completed but were not analyzed will be re-run. The final coverage report will include results for skipped modules.

Brownie compares hashes of the following items to check if a test should be re-run:

- The bytecode for every contract deployed during execution of the test
- The AST of the test module
- The AST of all `conftest.py` modules that are accessible to the test module

8.5.2 Evaluating Coverage

To check your unit test coverage, add the `--coverage` flag when running pytest:

```
$ pytest tests/ --coverage
```

When the tests complete, a report will display:

```
Coverage analysis:

  contract: Token - 82.3%
    SafeMath.add - 66.7%
    SafeMath.sub - 100.0%
    Token.<fallback> - 0.0%
    Token.allowance - 100.0%
    Token.approve - 100.0%
    Token.balanceOf - 100.0%
    Token.decimals - 0.0%
    Token.name - 100.0%
    Token.symbol - 0.0%
    Token.totalSupply - 100.0%
    Token.transfer - 85.7%
    Token.transferFrom - 100.0%

Coverage report saved at reports/coverage.json
```

Brownie outputs a % score for each contract method that you can use to quickly gauge your overall coverage level. A detailed coverage report is also saved in the project's `reports` folder, that can be viewed via the Brownie GUI. See *Viewing Coverage Data* for more information.

8.6 Configuration Settings

The following test configuration settings are available in `brownie-config.yaml`. These settings affect the behaviour of your tests.

```
pytest:
    gas_limit: 6721975
    default_contract_owner: false
    reverting_tx_gas_limit: 6721975
    revert_traceback: false
```

gas_limit
Replaces the default network gas limit.

reverting_tx_gas_limit
Replaces the default network setting for the gas limit on a tx that will revert.

default_contract_owner
If `True`, calls to contract transactions that do not specify a sender are broadcast from the same address that deployed the contract.

If `False`, contracts will not remember which account they were created by. You must explicitely declare the sender of every transaction with a transaction parameters dictionary as the last method argument.

revert_traceback
If `True`, unhandled `VirtualMachineError` exceptions will include a full transaction traceback. This is useful for debugging but slows test execution.

This can also be enabled from the command line with the `--revert-tb` flag.

CHAPTER 9

Debugging Tools

When using the console, transactions that revert still return a *TransactionReceipt* object. This object provides access to various attributes and methods that help you determine why it reverted.

Note: Debugging functionality relies on the debug_traceTransaction RPC method. If you are using Infura this endpoint is unavailable. Attempts to access this functionality will raise an `RPCRequestError`.

9.1 Revert Strings

The first step in determining why a transaction has failed is to look at the error string it returned (the "revert string"). This is available as `TransactionReceipt.revert_msg`, and is also displayed in the console output when the transaction confirms. Often this alone will be enough to understand what has gone wrong.

```
>>> tx = token.transfer(accounts[1], 11000, {'from': accounts[0]})

Transaction sent: 0xd31c1c8db46a5bf2d3be822778c767e1b12e0257152fcc14dcf7e4a942793cb4
SecurityToken.transfer confirmed (Insufficient Balance) - block: 13   gas used:␣
↪226266 (2.83%)
<Transaction object
↪'0xd31c1c8db46a5bf2d3be822778c767e1b12e0257152fcc14dcf7e4a942793cb4'>

>>> tx.revert_msg
'Insufficient Balance'
```

A good coding practice is to use one expression per `require` so your revert strings can be more precise. For example, if a transaction fails from the following require statement you cannot immediately tell whether it failed because of the balance or the allowance:

```
1 function transferFrom(address _from, address _to, uint _amount) external returns␣
  ↪(bool) {
2     require (allowed[_from][msg.sender] >= _amount && balance[_from] >= _amount);
```

By seperating the `require` expressions, unique revert strings are possible and determining the cause becomes trivial:

```
function transferFrom(address _from, address _to, uint _amount) external returns
↪(bool) {
    require (allowed[_from][msg.sender] >= _amount, "Insufficient allowance");
    require (balance[_from][] >= _amount, "Insufficient Balance");
```

9.2 Contract Source Code

You can call `TransactionReceipt.error()` to display the section of the contract source that caused the revert. Note that in some situations, particiarly where an `INVALID` opcode is raised, the source may not be available.

```
>>> tx.error()
Trace step 5197, program counter 9719:
    File "contracts/SecurityToken.sol", line 136, in SecurityToken._checkTransfer:
    require(balances[_addr[SENDER]] >= _value, "Insufficient Balance");
```

Sometimes the source that reverted is insufficient to determine what went wrong, for example if a SafeMath `require` failed. In this case you can call `TransactionReceipt.traceback()` to view a python-like traceback for the failing transaction. It shows source highlights at each jump leading up to the revert.

```
>>> tx.traceback()
Traceback for '0xd31c1c8db46a5bf2d3be822778c767e1b12e0257152fcc14dcf7e4a942793cb4':
Trace step 169, program counter 3659:
    File "contracts/SecurityToken.sol", line 156, in SecurityToken.transfer:
    _transfer(msg.sender, [msg.sender, _to], _value);
Trace step 5070, program counter 5666:
    File "contracts/SecurityToken.sol", lines 230-234, in SecurityToken._transfer:
    _addr = _checkTransfer(
        _authID,
        _id,
        _addr
    );
Trace step 5197, program counter 9719:
    File "contracts/SecurityToken.sol", line 136, in SecurityToken._checkTransfer:
    require(balances[_addr[SENDER]] >= _value, "Insufficient Balance");
```

9.3 Events

Brownie provides access to events that fired in reverted transactions. They are viewable via `TransactionReceipt.events` in the same way as events for successful transactions. If you cannot determine why a transaction reverted or are getting unexpected results, one approach is to add temporary logging events into your code to see the values of different variables during execution.

See the *events* section of *Project Interaction via the Console* for information on how event data is stored.

9.4 The Transaction Trace

The best way to understand exactly happened in a failing transaction is to generate and examine the transaction trace. This is available as a list of dictionaries at `TransactionReceipt.trace`, with several fields added to make it easier to understand.

Each step in the trace includes the following data:

```
{
    'address': "",  // address of the contract containing this opcode
    'contractName': "",  // contract name
    'depth': 0,  // the number of external jumps away the initially called contract␣
↪(starts at 0)
    'error': "",  // occurred error
    'fn': "",  // function name
    'gas': 0,  // remaining gas
    'gasCost': 0,  // cost to execute this opcode
    'jumpDepth': 1,  // number of internal jumps within the active contract (starts␣
↪at 1)
    'memory': [],  // execution memory
    'op': "",  // opcode
    'pc': 0,  // program counter
    'source': {
        'filename': "path/to/file.sol",  // path to contract source
        'offset': [0, 0]  // start:stop offset associated with this opcode
    },
    'stack': [],  // execution stack
    'storage': {}  // contract storage
}
```

9.5 Call Traces

Because the trace is often many thousands of steps long, it can be challenging to know where to begin when examining it. Brownie provides the `TransactionReceipt.call_trace()` method to view a complete map of every jump that occured in the transaction, along with associated trace indexes:

```
>>> tx.call_trace()
Call trace for '0xd31c1c8db46a5bf2d3be822778c767e1b12e0257152fcc14dcf7e4a942793cb4':
SecurityToken.transfer 0:5198  (0xea53cB8c11f96243CE3A29C55dd9B7D761b2c0BA)
└─SecurityToken._transfer 170:5198
    ├─IssuingEntity.transferTokens 608:4991 ␣
↪(0x40b49Ad1B8D6A8Df6cEdB56081D51b69e6569e06)
    │ ├─IssuingEntity.checkTransfer 834:4052
    │ │ ├─IssuingEntity._getID 959:1494
    │ │ │ └─KYCRegistrar.getID 1186:1331  (0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8)
    │ │ ├─IssuingEntity._getID 1501:1635
    │ │ ├─IssuingEntity._getID 1642:2177
    │ │ │ └─KYCRegistrar.getID 1869:2014  (0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8)
    │ │ ├─IssuingEntity._getInvestors 2305:3540
    │ │ │ └─KYCRegistrar.getInvestors 2520:3483 ␣
↪(0xa79269260195879dBA8CEFF2767B7F2B5F2a54D8)
    │ │ │   ├─KYCBase.isPermitted 2874:3003
    │ │ │   │ └─KYCRegistrar.isPermittedID 2925:2997
    │ │ │   └─KYCBase.isPermitted 3014:3143
    │ │ │     └─KYCRegistrar.isPermittedID 3065:3137
    │ │ └─IssuingEntity._checkTransfer 3603:4037
    │ ├─IssuingEntity._setRating 4098:4162
    │ ├─IssuingEntity._setRating 4204:4268
    │ ├─SafeMath32.add 4307:4330
    │ └─IssuingEntity._incrementCount 4365:4770
    │   ├─SafeMath32.add 4400:4423
```

(continues on next page)

(continued from previous page)

```
    │   ├─SafeMath32.add 4481:4504
    │   ├─SafeMath32.add 4599:4622
    │   └─SafeMath32.add 4692:4715
    └─SecurityToken._checkTransfer 5071:5198
```

Each line shows the active contract and function name, the trace indexes where the function is entered and exitted, and an address if the function was entered via an external jump. Functions that terminated with `REVERT` or `INVALID` opcodes are highlighted in red.

Calling `call_trace` provides an initial high level overview of the transaction execution path, which helps you to examine the individual trace steps in a more targetted manner.

CHAPTER 10

The Brownie GUI

Brownie includes a GUI for viewing test coverage data and analyzing the compiled bytecode of your contracts.

Parts of this section assume a level of familiarity with EVM bytecode. If you are looking to learn more about the subject, Alejandro Santander from OpenZeppelin has written an excellent guide - Deconstructing a Solidity Contract.

10.1 Getting Started

To open the GUI, run the following command from within your project folder:

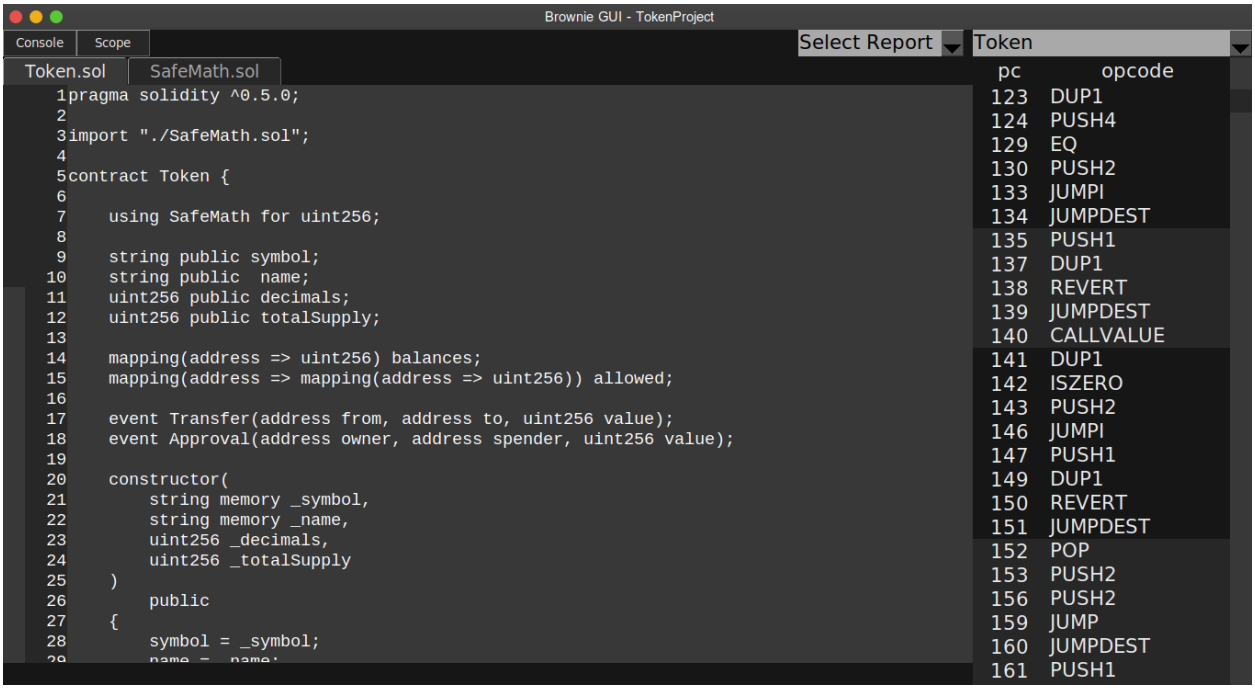
```
$ brownie gui
```

Or from the console:

```
>>> Gui()
```

Once loaded, the first thing you'll want to do is choose a contract to view. To do this, click on the drop-down list in the upper right that says "Select a Contract". You will see a list of every deployable contract within your project.

Once selected, the contract source code is displayed in the main window with a list of opcodes and program counters on the right. If the contract inherits from more than one source file, tabs will be available to switch between sources. For example, in the image below the `Token` contract includes both `Token.sol` and `SafeMath.sol`:



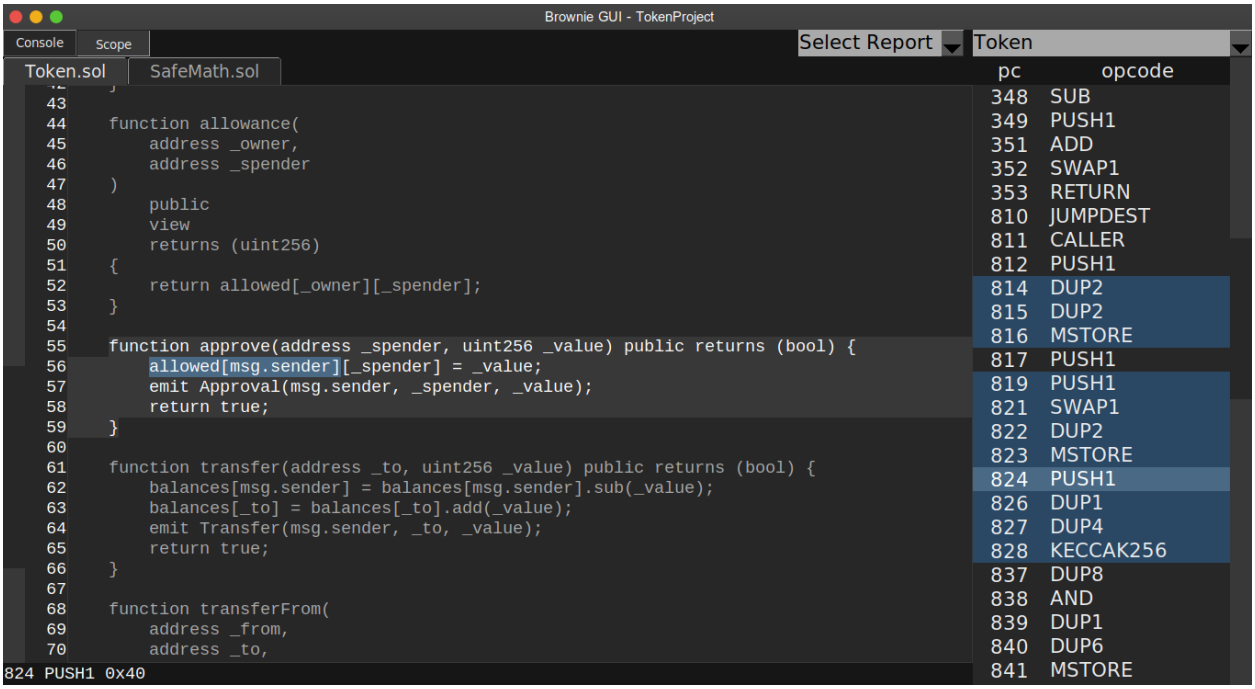
10.2 Working with Opcodes

10.2.1 Mapping Opcodes to Source

Highlighting a section of code will also highlight the instructions that are associated with it. Similarly, selecting on an instruction will highlight the related source.

Click the `Scope` button in the top left (or the `S` key) to filter the list of instructions such that only those contained within the highlighted source are shown.

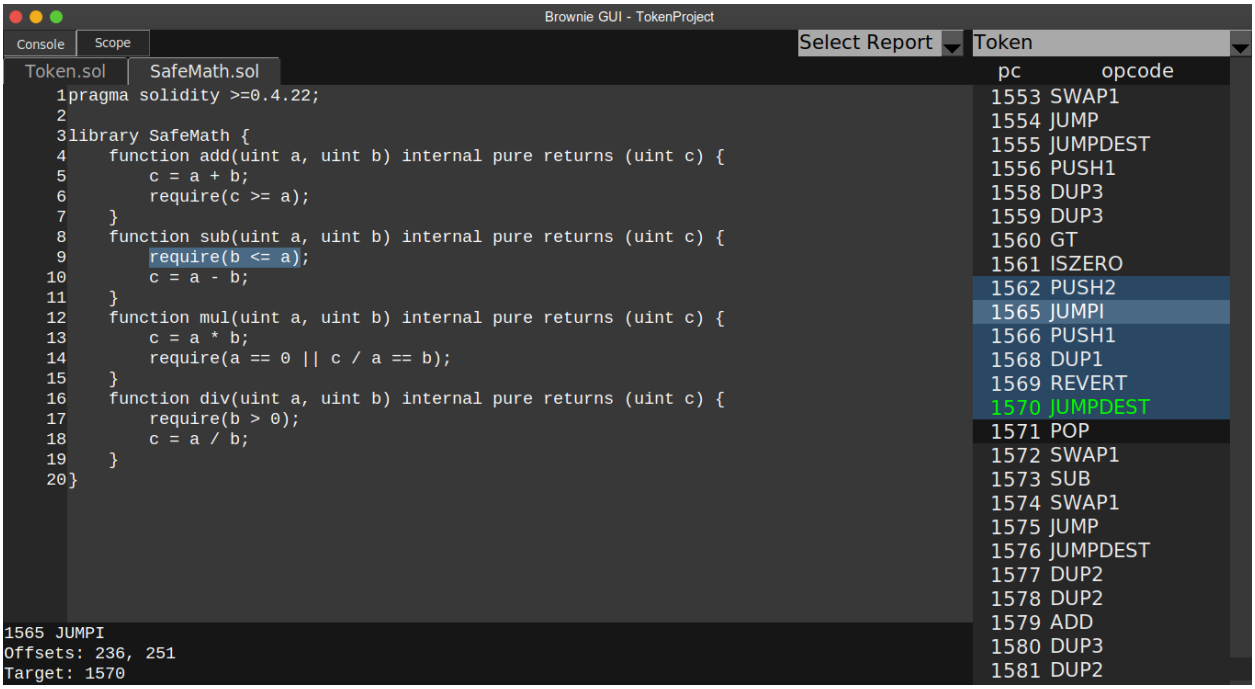
Note: Opcodes displayed with a dark background are not mapped to any source, or are mapped to the source of the entire contract. These are typically the result of compiler optimization or part of the initial function selector.



10.2.2 Jump Instructions

Click the `Console` button in the top left (or the `C` key) to expand the console. It shows more detailed information about the highlighted instruction.

- When you select a `JUMP` or `JUMPI` instruction, the console includes a "Target:" field that gives the program counter for the related `JUMPDEST`, where possible. The related `JUMPDEST` is also highlighted in green. Press the `J` key to show the instruction.
- When you select a `JUMPDEST` instruction, the console includes a "Jumps:" field that gives a list of program counters that point at the highlighted instruction. Each related `JUMP`/`JUMPI` is also highlighted in green.



10.2.3 Miscellaneous

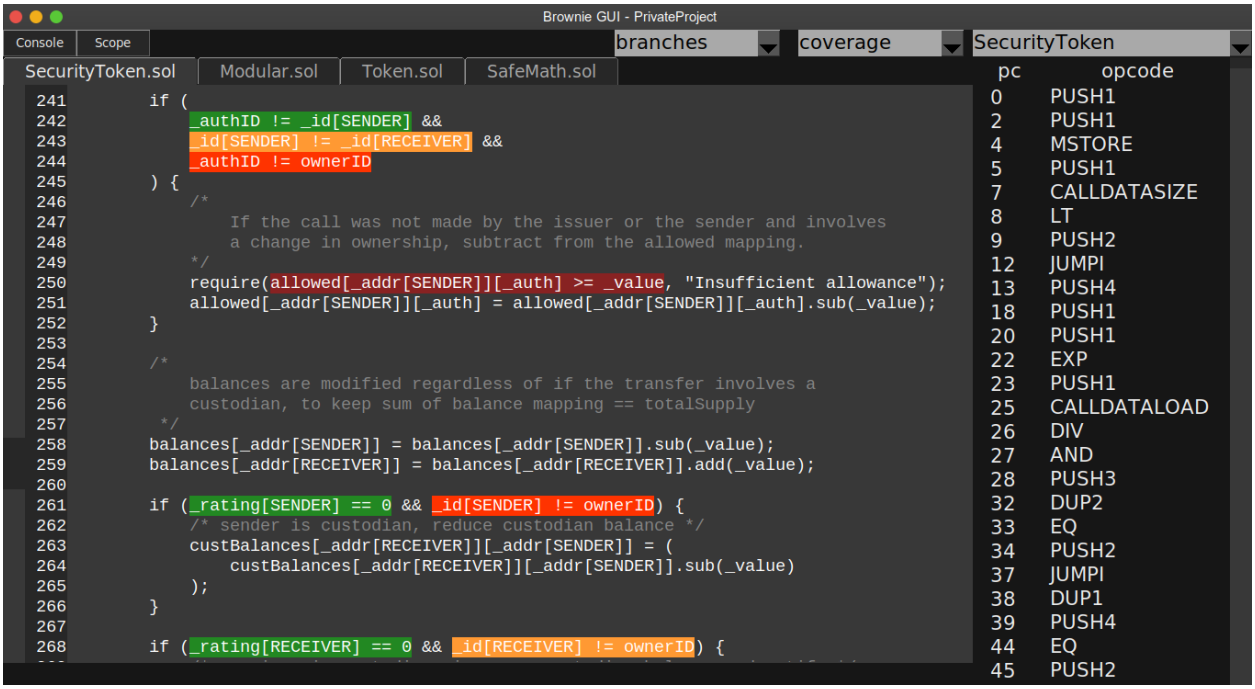
- Right clicking on an instruction will apply a yellow highlight to all instructions of the same opcode type.
- Press the `R` key to toggle highlight on all `REVERT` opcodes.

10.3 Viewing Coverage Data

For an in-depth look at your test coverage, click on the drop-down list in the upper right that says “Select Report” and choose “coverage”. A new drop-down list will appear where you can select which type of coverage data to view (branches or statements).

Relevant code will be highlighted in different colors:

- Green code was executed during the tests
- Yellow branch code executed, but only evaluated truthfully
- Orange branch code executed, but only evaluated falsely
- Red code did not execute during the tests

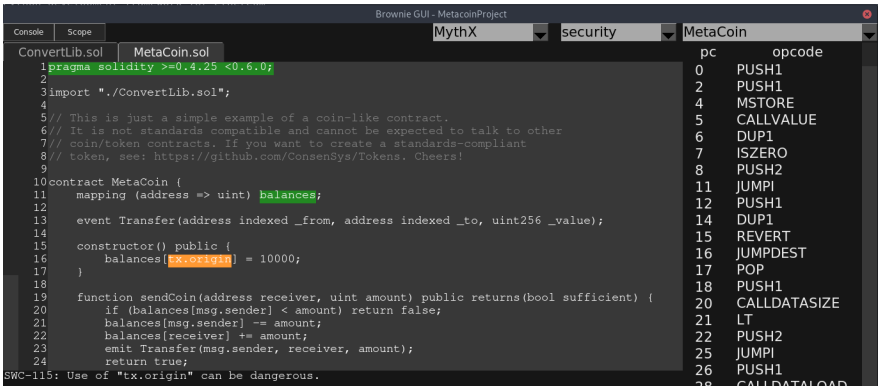


10.4 Viewing Security Report Data

Once the `brownie analyze` command has finished, the GUI will show a new security report. Select the `security` report and the `MythX` report type. If any vulnerabilities have been found, they will be highlighted based on their severity:

- Green Low severity (best practice violations)
- Yellow Medium severity (potential vulnerability), needs to be fixed
- Red High severity (critical, immediate danger of exploitation)

The report data can also be directly accessed in `reports/security.json`.



10.5 Report JSON Format

Project coverage data is saved to `reports/coverage.json` using Brownie's standard report format. Third party tools wishing to display information in the Brownie GUI can also save JSON files within the `reports/` folder.

Brownie expects JSON reports to use the following structure:

```
{
    "highlights": {
        // this name is shown in the report type drop-down menu
        "<Report Type>": {
            "ContractName": {
                "path/to/sourceFile.sol": [
                    // start offset, stop offset, color, optional message
                    [123, 440, "green", ""],
                    [502, 510, "red", ""],
                ]
            }
        }
    },
    "sha1": {} // optional, not yet implemented
}
```

The final item in each highlight offset is an optional message to be displayed. If included, the text given here will be shown in the GUI console when the user hovers the mouse over the highlight. To not show a message, set it to `""` or `null`.

CHAPTER 11

Deploying Contracts

Brownie lets you write scripts to interact with your project. Scripting is especially useful for deploying your contracts to the main-net, or for automating processes that you perform regularly.

Every script should begin with `from brownie import *`. This imports the instantiated project classes into the local namespace and gives access to the Brownie *Brownie API* in exactly the same way as if you were using the console.

To execute a script from the command line:

```
$ brownie run <script> [function]
```

Or from the console, use the `run` method:

```
>>> run('token') # executes the main() function within scripts/token.py
```

Or the import statement:

```
>>> from scripts.token import main
>>> main()
```

Scripts are stored in the `scripts/` folder. Each script can contain as many functions as you'd like. If no function name is given, brownie will attempt to run `main`.

Here is a simple example script from the `token` project, used to deploy the `Token` contract from `contracts/Token.sol` using `web3.eth.accounts[0]`.

```
from brownie import *

def main():
    accounts[0].deploy(Token, "Test Token", "TEST", 18, "1000 ether")
```

See the *Brownie API* documentation for available classes and methods when writing scripts.

11.1 Unlinked Libraries

If a contract requires a library, the most recently deployed one will be used automatically. If the required library has not been deployed yet an `UndeployedLibrary` exception is raised.

```
>>> accounts[0].deploy(MetaCoin)
  File "brownie/network/contract.py", line 167, in __call__
    f"Contract requires '{library}' library but it has not been deployed yet"
UndeployedLibrary: Contract requires 'ConvertLib' library but it has not been␣
↪deployed yet

>>> accounts[0].deploy(ConvertLib)
Transaction sent: 0xff3f5cff35c68a73658ad367850b6fa34783b4d59026520bd61b72b6613d871c
ConvertLib.constructor confirmed - block: 1   gas used: 95101 (48.74%)
ConvertLib deployed at: 0x08c4C7F19200d5636A1665f6048105b0686DFf01
<ConvertLib Contract object '0x08c4C7F19200d5636A1665f6048105b0686DFf01'>

>>> accounts[0].deploy(MetaCoin)
Transaction sent: 0xd0969b36819337fc3bac27194c1ff0294dd65da8f57c729b5efd7d256b9ecfb3
MetaCoin.constructor confirmed - block: 2   gas used: 231857 (69.87%)
MetaCoin deployed at: 0x8954d0c17F3056A6C98c7A6056C63aBFD3e8FA6f
<MetaCoin Contract object '0x8954d0c17F3056A6C98c7A6056C63aBFD3e8FA6f'>
```

CHAPTER 12

The Local RPC Client

Brownie is designed to use ganache-cli as a local development environment.

12.1 Launching and Connecting

The connection settings for the local RPC are outlined in `brownie-config.yaml`:

```
development:
    host: http://127.0.0.1
    reverting_tx_gas_limit: 6721975
    test_rpc:
        cmd: ganache-cli
        port: 8545
        gas_limit: 6721975
        accounts: 10
        evm_version: petersburg
        mnemonic: brownie
```

Brownie will launch or attach to the client when using any network that includes a `test-rpc` dictionary in it's settings.

Each time Brownie is loaded, it will first attempt to connect to the `host` address to determine if the RPC client is already active.

12.1.1 Client is Active

If able to connect to the `host` address, Brownie:

- Checks the current block height and raises an Exception if it is greater than zero
- Locates the process listening at the address and attaches it to the `Rpc` object
- Takes a snapshot

When Brownie is terminated:

- The RPC client is reverted based on the initial snapshot.

12.1.2 Client is not Active

If unable to connect to the `host` address, Brownie:

- Launches the client using the `test-rpc` command given in the configuration file
- Waits to see that the process loads successfully
- Confirms that it can connect to the new process
- Attaches the process to the `Rpc` object

When Brownie is terminated:

- The RPC client and any child processes are also terminated.

12.2 Common Interactions

You can interact with the RPC client using the *Rpc* object, which is automatically instantiated as `rpc`:

```
>>> rpc
<brownie.network.rpc.Rpc object at 0x7f720f65fd68>
```

12.2.1 Mining

To mine empty blocks, use `rpc.mine`.

```
>>> web3.eth.blockNumber
0
>>> rpc.mine(50)
Block height at 50
>>> web3.eth.blockNumber
50
```

12.2.2 Time

You can call `rpc.time` to view the current epoch time. To fast forward, call `rpc.sleep`.

```
>>> rpc.time()
1557151189
>>> rpc.sleep(100)
>>> rpc.time()
1557151289
```

12.2.3 Snapshots

`rpc.snapshot` takes a snapshot of the current state of the blockchain:

```
>>> rpc.snapshot()
Snapshot taken at block height 4
>>> accounts[0].balance()
100000000000000000000
>>> accounts[0].transfer(accounts[1], "10 ether")

Transaction sent: 0xd5d3b40eb298dfc48721807935eda48d03916a3f48b51f20bcded372113e1dca
Transaction confirmed - block: 5   gas used: 21000 (100.00%)
<Transaction object
↪'0xd5d3b40eb298dfc48721807935eda48d03916a3f48b51f20bcded372113e1dca'>
```

You can return to this state later using `rpc.revert`:

```
>>> accounts[0].balance()
89999580000000000000
>>> rpc.revert()
Block height reverted to 4
>>> accounts[0].balance()
100000000000000000000
```

Reverting does not consume a snapshot. You can return to the same snapshot as many times as needed. However, if you take a new snapshot the previous one is no longer accessible.

To return to the genesis state, use `rpc.reset`.

```
>>> web3.eth.blockNumber
6
>>> rpc.reset()
>>> web3.eth.blockNumber
0
```

CHAPTER 13

Using Non-Local Networks

In addition to using ganache-cli as a local development environment, Brownie can connect to non-local networks (i.e. any testnet/mainnet node that supports JSON RPC).

Warning: Before you go any further, consider that connecting to non-local networks can potentially expose your private keys if you aren't careful:

- When interacting with the mainnet, make sure you verify all of the details of any transactions before signing or sending. Brownie cannot protect you from sending ETH to the wrong address, sending too much, etc.
- Always protect your private keys. Don't leave them lying around unencrypted!

13.1 Registering with Infura

Before you can connect to a non-local network, you need access to an Ethereum node (whether your own local one or hosted) that supports JSON RPC (either HTTP, IPC, or web-sockets). Infura is a good option for accessing a hosted node. Once you register and create a project, Infura will provide you with a project ID as well as API URLs that can be leveraged to access the given network.

13.2 Network Configuration

13.2.1 Defining Non-Local Networks

The connection settings for non-local networks must be defined in `brownie-config.yaml`.

First, for each network you want to configure, create a new section in the network.networks section as below:

```
network:
    networks:
```

(continues on next page)

(continued from previous page)

```
        ropsten:
            host: http://ropsten.infura.io/v3/$WEB3_INFURA_PROJECT_ID
```

If using Infura, you can provide your project ID key as an environment variable or by modifying the `hosts` setting in the configuration file.

The environment variable is set to `WEB3_INFURA_PROJECT_ID` in the default configuration file. Use the following command to set the environment variable:

```
$ export WEB3_INFURA_PROJECT_ID=YourProjectID
```

13.2.2 Setting the Default Network

To modify the default network that Brownie connects to, update the network.default field as shown below:

```
network:
    default: ropsten
```

13.3 Launching and Connecting to Networks

13.3.1 Using the CLI

By default, Brownie will connect to whichever network is set as "default" in `brownie-config.yaml`. To connect to a different network, use the `--network` flag:

```
$ brownie --network ropsten
```

13.3.2 Using brownie.network

The `brownie.network` module conains methods that allow you to connect or disconnect from any network defined within the configuration file.

To connect to a network:

```
>>> network.connect('ropsten')
>>> network.is_connected()
True
>>> network.show_active()
'ropsten'
```

To disconnect:

```
>>> network.disconnect()
>>> network.is_connected()
False
```

13.4 Interacting with Non-Local Networks

There are several key differences in functionality between using a non-local network as opposed to a local develpment environment.

13.4.1 Contracts

ProjectContract

By default, Brownie stores information about contract deployments on non-local networks. `ProjectContract` instances will persist through the following actions:

- Disconnecting and reconnecting to the same network
- Closing and reloading a project
- Exiting and reloading Brownie
- Modifying a contract's source code - Brownie still retains the source for the deployed version

The following actions will remove locally stored data for a `ProjectContract`:

- Calling `ContractContainer.remove` or `ContractContainer.clear` will erase deployment information for the removed `ProjectContract` instances.
- Removing a contract source file from your project (or renaming it) will cause Brownie to delete all deployment information for the removed contract.

You can create a `ProjectContract` instance for an already-deployed contract with the *ContractContainer*'s `ContractContainer.at` method.

See *The Configuration File* for information on how to enable or disable persistence.

Contract

The *Contract* class (available as `brownie.Contract`) is used to interact with already deployed contracts that are not a part of your core project. You will need to provide an ABI as a `dict` generated from the compiled contract code.

```
>>> Contract('0x79447c97b6543F6eFBC91613C655977806CB18b0', "Token", abi)
<Token Contract object '0x79447c97b6543F6eFBC91613C655977806CB18b0'>
```

Once instantiated, all of the usual `Contract` attributes and methods can be used to interact with the deployed contract.

13.4.2 Accounts

Brownie will automatically load any unlocked accounts returned by a node. If you are using your own private node, you will be able to access your accounts in the same way you would in a local environment.

When connected to a hosted node such as Infura, local accounts must be added via the `Accounts.add` method:

```
>>> accounts.add('8fa2fdfb89003176a16b707fc860d0881da0d1d8248af210df12d37860996fb2')
<Account object '0xc1826925377b4103cC92DeeCDF6F96A03142F37a'>
>>> accounts[0].balance()
177227502990000000000
```

Once an account is added to the `Accounts` object, use *Account.save* to save the it to an encrypted keystore, and *Accounts.load* to open it for subsequent use.

13.4.3 Transactions

After broadcasting a transaction, Brownie will pause and wait until it confirms. If you are using the console you can press `Ctrl-C` to immediately receive the *TransactionReceipt* object. Note that `TransactionReceipt.status` will be `-1` until the transaction is mined, and many attributes and methods will not yet be available.

Debugging

Brownie's *debugging tools* rely upon the debug_traceTransaction RPC method which is not supported by Infura. Attempts to call it will result in a `RPCRequestError`. This means that the following `TransactionReceipt` attributes and methods are unavailable:

- `TransactionReceipt.return_value`
- `TransactionReceipt.trace`
- `TransactionReceipt.call_trace`
- `TransactionReceipt.traceback`
- `TransactionReceipt.source`

13.4.4 Rpc

The *Rpc* object is unavailable when working with non-local networks.

CHAPTER 14

The Configuration File

Every project has a file `brownie-config.yaml` that holds all the configuration settings. The defaut configuration is as follows.

```yaml
# Brownie configuration file
# https://eth-brownie.readthedocs.io/en/stable/config.html
network:
    default: development # the default network that brownie connects to
    settings:
        gas_limit: false
        gas_price: false
        persist: true
        reverting_tx_gas_limit: false  # if false, reverting tx's will raise without broadcasting
    networks:
        # any settings given here will replace the defaults
        development:
            host: http://127.0.0.1
            persist: false
            reverting_tx_gas_limit: 6721975
            test_rpc:
                cmd: ganache-cli
                port: 8545
                gas_limit: 6721975
                accounts: 10
                evm_version: petersburg
                mnemonic: brownie
        # set your Infura API token to the environment variable WEB3_INFURA_PROJECT_ID
        mainnet:
            host: https://mainnet.infura.io/v3/$WEB3_INFURA_PROJECT_ID
        goerli:
            host: https://goerli.infura.io/v3/$WEB3_INFURA_PROJECT_ID
        kovan:
            host: https://kovan.infura.io/v3/$WEB3_INFURA_PROJECT_ID
        rinkeby:
```

(continues on next page)

(continued from previous page)

```
            host: https://rinkeby.infura.io/v3/$WEB3_INFURA_PROJECT_ID
        ropsten:
            host: https://ropsten.infura.io/v3/$WEB3_INFURA_PROJECT_ID
pytest:
    # these settings replace the defaults when running pytest
    gas_limit: 6721975
    default_contract_owner: false
    reverting_tx_gas_limit: 6721975
    revert_traceback: false
compiler:
    solc:
        version: null
        evm_version: null
        optimize: true
        runs: 200
        minify_source: false
colors:
    key:
    value: bright blue
    callable: bright cyan
    module: bright blue
    contract: bright magenta
    contract_method: bright magenta
    string: bright magenta
    dull: dark white
    error: bright red
    success: bright green
    pending: bright yellow
```

When using the Brownie console or writing scripts, you can view and edit configuration settings through the `config` dict. Any changes made in this way are temporary and will be reset when you exit Brownie or reset the network.

Note: If you are experiencing errors or warnings related to the configuration file, delete it and then run `brownie init` from the root folder of your project. This will create a clean copy of the config file.

14.1 Settings

The following settings are available:

network
: Defines the available networks and how Brownie interacts with them.

  - `default`: The default network that brownie connects to when loaded. If a different network is required, you can override this setting with the `--network` flag in the command line.

  `network.`**settings**
  : Default settings for every network. The following properties can be set:

    - `gas_price`: The default gas price for all transactions. If left as `false` the gas price will be determined using `web3.eth.gasPrice`.
    - `gas_limit`: The default gas limit for all transactions. If left as `false` the gas limit will be determined using `web3.eth.estimateGas`.

- `persist`: If `True`, Brownie will remember information about deployed contracts in between sessions. This is enabled by default for all non-local networks.
- `reverting_tx_gas_limit`: The gas limit to use when a transaction would revert. If set to `false`, transactions that would revert will instead raise a `VirtualMachineError`.

`network.`**`networks`**

Settings specific to individual networks. All values outlined above in `settings` are also valid here and will override the defaults.

Additionally, you must include a host setting in order to connect to that network:

- `host`: The address of the RPC API you wish to connect to. You can include environment variables, they will be expanded when attempting connect. The default settings use Infura and look for the project ID token as `WEB3_INFURA_PROJECT_ID`.

`networks.`**`test_rpc`**

An optional dictionary outlining settings for how the local RPC client is loaded. If not included, Brownie will not attempt to launch or attach to the process. See *The Local RPC Client* for more details. `test-rpc` properties include:

- `cmd`: The command-line argument used to load the client. You can add any extra flags here as needed.
- `port`: Port the client should listen on.
- `gas_limit`: Block gas limit.
- `accounts`: The number of funded accounts in `web3.eth.accounts`.
- `evm_version`: The EVM version to compile for. If `null` the most recent one is used. Possible values are `byzantium`, `constantinople` and `petersburg`.
- `mnemonic`: Local accounts are derived from this mnemonic. If set to `null`, you will have different local accounts each time Brownie is run.
- `account_keys_path`: Optional path to save generated accounts and private keys as a JSON object

`compiler`

Compiler settings. See *compiler settings* for more information.

`compiler.`**`solc`**

Settings specific to the Solidity compiler. At present this is the only compiler supported by Brownie.

- `version`: The version of solc to use. Should be given as a string in the format `0.x.x`. If set to `null`, the version is set based on the contract pragma. Brownie supports solc versions `>=0.4.22`.
- `evm_version`: The EVM version to compile for. If `null` the most recent one is used. Possible values are `byzantium`, `constantinople` and `petersburg`.
- `optimize`: Set to `true` if you wish to enable compiler optimization.
- `runs`: The number of times the optimizer should run.
- `minify_source`: If `true`, contract source is minified before compiling.

`pytest`

Properties that only affect Brownie's configuration when running tests. See *test configuration settings* for more information.

- `gas_limit`: Replaces the default network gas limit.
- `default_contract_owner`: If `false`, deployed contracts will not remember the account that they were created by and you will have to supply a `from` kwarg for every contract transaction.
- `reverting_tx_gas_limit`: Replaces the default network setting for the gas limit on a tx that will revert.

- `revert_traceback`: if `true`, unhandled `VirtualMachineError` exceptions will include a full traceback for the reverted transaction.

`colors`

Defines the colors associated with specific data types when using Brownie. Setting a value as an empty string will use the terminal's default color.

CHAPTER 15

The Build Folder

Each project has a `build/` folder that contains various data files. If you are integrating a third party tool or hacking on the Brownie source code, it can be valuable to understand how these files are structured.

15.1 Compiler Artifacts

Brownie generates compiler artifacts for each contract within a project, which are stored in the `build/contracts` folder. The structure of these files are as follows:

```
{
    'abi': [], // contract ABI
    'allSourcePaths': [], // relative paths to every related contract source code file
    'ast': {}, // the AST object
    'bytecode': "0x00", // bytecode object as a hex string, used for deployment
    'bytecodeSha1': "", // hash of bytecode without final metadata
    'compiler': {}, // information about the compiler
    'contractName': "", // name of the contract
    'coverageMap': {}, // map for evaluating unit test coverage
    'deployedBytecode': "0x00", // bytecode as hex string after deployment
    'deployedSourceMap': "", // source mapping of the deployed bytecode
    'dependencies': [], // contracts and libraries that this contract inherits from␣
↪or is linked to
    'offset': [], // source code offsets for this contract
    'opcodes': "", // deployed contract opcodes list
    'pcMap': [], // program counter map
    'sha1': "", // hash of the contract source, used to check if a recompile is␣
↪necessary
    'source': "", // compiled source code as a string
    'sourceMap': "", // source mapping of undeployed bytecode
    'sourcePath': "", // relative path to the contract source code file
    'type': "" // contract, library, interface
}
```

This raw data is available within Brownie through the build module. If the contract was minified before compiling, Brownie will automatically adjust the source map offsets in `pcMap` and `coverageMap` to fit the current source.

```
>>> from brownie.project import build
>>> token_json = build.get("Token")
>>> token_json['contractName']
"Token"
```

15.1.1 Program Counter Map

Brownie generates an expanded version of the deployed source mapping that it uses for debugging and test coverage evaluation. It is structured as a dictionary of dictionaries, where each key is a program counter as given by `debug_traceTransaction`.

If a value is `false` or the type equivalent, the key is not included.

```
{
    'pc': {
        'op': "", // opcode string
        'path': "", // relative path to the contract source code
        'offset': [0, 0], // source code start and stop offsets
        'fn': str, // name of the related method
        'jump': "", // jump instruction as given in the sourceMap (i, o)
        'value': "0x00", // hex string value of the instruction
        'statement': 0, // statement coverage index
        'branch': 0 // branch coverage index
    }
}
```

15.1.2 Coverage Map

All compiler artifacts include a `coverageMap` which is used when evaluating test coverage. It is structured as a nested dictionary in the following format:

```
{
    "statements": {
        "/path/to/contract/file.sol": {
            "ContractName.functionName": {
                "index": [start, stop]  // source offsets
            }
        }
    },
    "branches": {
        "/path/to/contract/file.sol": {
            "ContractName.functionName": {
                "index": [start, stop, bool]  // source offsets, jump boolean
            }
        }
    }
}
```

- Each `statement` index exists on a single program counter step. The statement is considered to have executed when the corresponding opcode executes within a transaction.

- Each `branch` index is found on two program counters, one of which is always a `JUMPI` instruction. A transaction must run both opcodes before the branch is considered to have executed. Whether it evaluates true or false depends on if the jump occurs.

See *Coverage Map Indexes* for more information.

15.2 Deployment Artifacts

Each time a contract is deployed to a network where *persistence* is enabled, Brownie saves a copy of the :ref‘compiler artifact<build-folder-compiler>‘_ used for deployment. In this way accurate deployment data is maintained even if the contract’s source code is later modified.

Deployment artifacts are stored at:

```
build/deployments/[NETWORK_NAME]/[ADDRESS].json
```

When instantiating *Contract* objects from deployment artifacts, Brownie parses the files in order of creation time. If the `contractName` field in an artifact gives a name that longer exists within the project, the file is deleted.

15.3 Test Results and Coverage Data

The `build/test.json` file holds information about unit tests and coverage evaluation. It has the following format:

```
{
    "contracts": {
        "contractName": "0xff" // Hash of the contract source
    },
    //
    "tests": {
        "tests/path/of/test_file.py": {
            "coverage": true, // Has coverage eval been performed for this module?
            "isolated": [], // List of contracts deployed when executing this module.
↪Used to determine if the tests must be re-run.
            "results": ".....", // Test results. Follows the same format as pytest's
↪output (.sfex)
            "sha1": "0xff", // Hash of the module
            "txhash": [] // List of transaction hashes generated when running this
↪module.
        },
    },
    // Coverage data for individual transactions
    "tx": {
        "0xff": { // Transaction hash
            "ContractName": {
                // Coverage map indexes (see below)
                "path/to/contract.sol": [
                    [], // statements
                    [], // branches that did not jump
                    []  // branches that did jump
                ]
            }
        }
    }
}
```

15.3.1 Coverage Map Indexes

In tracking coverage, Brownie produces a set of coverage map indexes for each transaction. They are represented as lists of lists, each list containing key values that correspond to that contract's *coverage map*. As an example, look at the following transaction coverage data:

```
{
    "ae6ccafbd0b0c8cf2eb623e390080854755f3fa7": {
        "Token": {
            // Coverage map indexes (see below)
            "contracts/Token.sol": [
                [1, 3],
                [],
                [5]
            ],
            "contracts/SafeMath.sol": [
                [8],
                [11],
                [11]
            ],
        }
    }
}
```

Here we see that within the `Token` contract:

- Statements 1 and 3 were executed in `"contracts/Token.sol"`, as well as statement 8 in `"contracts/SafeMath.sol"`
- In `"contracts/Token.sol"`, there were no branches that were seen and did not jump, branch 5 was seen and did jump
- In `"contracts/SafeMath.sol"`, branch 11 was seen both jumping and not jumping

To convert these indexes to source offsets, we check the *coverage map* for Token. For example, here is branch 11:

```
{
    "contracts/SafeMath.sol": {
        "SafeMath.add": {
            "11": [147, 153, true]
        }
    }
}
```

From this we know that the branch is within the `add` function, and that the related source code starts at position 147 and ends at 153. The final boolean indicates whether a jump means the branch evaluated truthfully of falsely - in this case, a jump means it evaluated `True`.

15.4 Installed ethPM Package Data

The `build/packages.json` file holds information about installed ethPM packages. It has the following format:

```
{
    "packages": {
        "package_name": {
            "manifest_uri": "ipfs://",  // ipfs URI of the package manifest
```

(continues on next page)

(continued from previous page)

```
            "registry_address": "",  // ethPM registry address the package was␣
↪installed from
            "version": ""  // package version string
        },
        ...
    },
    "sources": {
        "path/to/ContractFile.sol": {
            "md5": "",  // md5 hash of the source file at installation
            "packages": []  // installed packages that include this source file
        },
        ...
    }
}
```

CHAPTER 16

Brownie as a Python Package

Brownie can be imported as a package and used within regular Python scripts. This can be useful if you wish to incorporate a specific function or range of functionality within a greater project, or if you would like more granular control over how Brownie operates.

For quick reference, the following statements generate an environment and namespace identical to what you have when loading the Brownie console:

```
from brownie import *
p = project.load('my_projects/token', name="TokenProject")
p.load_config()
from brownie.project.TokenProject import *
network.connect('development')
```

16.1 Loading a Project

The `brownie.project` module is used to load a Brownie project.

```
>>> import brownie.project as project
>>> project.load('myprojects/token')
<Project object 'TokenProject'>
```

Once loaded, the `Project` object is available within `brownie.project`. This container holds all of the related `ContractContainer` objects.

```
>>> p = project.TokenProject
>>> p
<Project object 'TokenProject'>
>>> dict(p)
{'Token': <ContractContainer object 'Token'>, 'SafeMath': <ContractContainer object
↪'SafeMath'>}
>>> p.Token
<ContractContainer object 'Token'>
```

Alternatively, use a `from` import statement to import `ContractContainer` objects to the local namespace:

```
>>> from brownie.project.TokenProject import Token
>>> Token
<ContractContainer object 'Token'>
```

Importing with a wildcard will retrieve every available `ContractContainer`:

```
>>> from brownie.project.TokenProject import *
>>> Token
<ContractContainer object 'Token'>
>>> SafeMath
<ContractContainer object 'SafeMath'>
```

16.2 Loading Project Config Settings

When accessing Brownie via the regular Python interpreter, you must explicitely load configuration settings for a project:

```
>>> p = project.TokenProject
>>> p.load_config()
```

16.3 Accessing the Network

The `brownie.network` module contains methods for network interaction. The simplest way to connect is with the `network.connect` method:

```
>>> from brownie import network
>>> network.connect('development')
```

This method queries the network settings from the configuration file, launches the local RPC, and connects to it with a `Web3` instance. Alternatively, you can accomplish the same with these commands:

```
>>> from brownie.network import rpc, web3
>>> rpc.launch('ganache-cli')
>>> web3.connect('http://127.0.0.1:8545')
```

Once connected, the `accounts` container is automatically populated with local accounts.

```
>>> from brownie.network import accounts
>>> len(accounts)
0
>>> network.connect('development')
>>> len(accounts)
10
```

CHAPTER 17

Brownie API

The following classes and methods are available when writing brownie scripts or using the console.

Hint: From the console you can call `dir` to see available methods and attributes for any class. By default, callables are highlighed in cyan and attributes in blue. You can also call `help` on any class or method to view information on it's functionality.

17.1 Brownie API

17.1.1 brownie

The `brownie` package is the main package containing all of Brownie's functionality.

```
>>> from brownie import *
>>> dir()
['Gui', 'accounts', 'alert', 'brownie', 'check', 'compile_source', 'config', 'history
↪', 'network', 'project', 'rpc', 'web3', 'wei']
```

17.1.2 brownie.convert

The `convert` module contains methods relating to data conversion.

Type Conversions

The following classes and methods are used to convert arguments supplied to `ContractTx` and `ContractCall`.

`brownie.convert.`**`to_uint`**(*value*, *type_="uint256"*)

Converts a value to an unsigned integer. This is equivalent to calling `Wei` and then applying checks for over/underflows.

brownie.convert.**to_int**(*value*, *type_="int256"*)
Converts a value to a signed integer. This is equivalent to calling `Wei` and then applying checks for over/underflows.

brownie.convert.**to_bool**(*value*)
Converts a value to a boolean. Raises `ValueError` if the given value does not match a value in `(True, False, 0, 1)`.

brownie.convert.**to_address**(*value*)
Converts a value to a checksummed address. Raises `ValueError` if value cannot be converted.

brownie.convert.**to_bytes**(*value*, *type_="bytes32"*)
Converts a value to bytes. `value` can be given as bytes, a hex string, or an integer.

Raises `OverflowError` if the length of the converted value exceeds that specified by `type_`.

Pads left with `00` if the length of the converted value is less than that specified by `type_`.

```
>>> to_bytes('0xff','bytes')
b'\xff'
>>> to_bytes('0xff','bytes16')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xff'
```

brownie.convert.**to_string**(*value*)
Converts a value to a string.

brownie.convert.**bytes_to_hex**(*value*)
Converts a bytes value to a hex string.

```
>>> from brownie.convert import bytes_to_hex
>>> bytes_to_hex(b'\xff\x3a')
0xff3a
>>> bytes_to_hex('FF')
0xFF
>>> bytes_to_hex("Hello")
  File "brownie/types/convert.py", line 149, in bytes_to_hex
    raise ValueError("'{value}' is not a valid hex string".format(value))
ValueError: 'Hello' is not a valid hex string
```

Type Classes

For certain types of contract data, Brownie uses subclasses to assist with conversion and comparison.

class brownie.convert.**Wei**(*value*)
Integer subclass that converts a value to wei and allows comparisons, addition and subtraction using the same conversion.

`Wei` is useful for strings where you specify the unit, for large floats given in scientific notation, or where a direct conversion to `int` would cause inaccuracy from floating point errors.

Whenever a Brownie method takes an input referring to an amount of ether, the given value is converted to `Wei`. Balances and `uint`/`int` values returned in contract calls and events are given in `Wei`.

```
>>> from brownie import Wei
>>> Wei("1 ether")
1000000000000000000
>>> Wei("12.49 gwei")
12490000000
>>> Wei("0.029 shannon")
```

(continues on next page)

(continued from previous page)

```
29000000
>>> Wei(8.38e32)
838000000000000000000000000000000
>>> Wei(1e18) == "1 ether"
True
>>> Wei("1 ether") < "2 ether"
True
>>> Wei("1 ether") - "0.75 ether"
250000000000000000
```

class brownie.convert.**EthAddress**(*value*)

String subclass for address comparisons. Raises a `TypeError` when compared to a non-address.

Addresses returned from a contract call or as part of an event log are given in this type.

```
>>> from brownie.convert import EthAddress
>>> e = EthAddress("0x0035424f91fd33084466f402d5d97f05f8e3b4af")
'0x0035424f91Fd33084466f402d5d97f05f8E3b4af'
>>> e == "0x3506424F91fD33084466F402d5D97f05F8e3b4AF"
False
>>> e == "0x0035424F91fD33084466F402d5D97f05F8e3b4AF"
True
>>> e == "0x35424F91fD33084466F402d5D97f05F8e3b4AF"
Traceback (most recent call last):
File "brownie/convert.py", line 304, in _address_compare
    raise TypeError(f"Invalid type for comparison: '{b}' is not a valid address")
TypeError: Invalid type for comparison: '0x35424F91fD33084466F402d5D97f05F8e3b4AF
↪' is not a valid address

>>> e == "potato"
Traceback (most recent call last):
File "brownie/convert.py", line 304, in _address_compare
    raise TypeError(f"Invalid type for comparison: '{b}' is not a valid address")
TypeError: Invalid type for comparison: 'potato' is not a valid address
>>> type(e)
<class 'brownie.convert.EthAddress'>
```

class brownie.convert.**HexString**(*value*, *type_*)

Bytes subclass for hexstring comparisons. Raises `TypeError` if compared to a non-hexstring. Evaluates `True` for hex strings with the same value but differing leading zeros or capitalization.

All `bytes` values returned from a contract call or as part of an event log are given in this type.

```
>>> from brownie.convert import HexString
>>> h = HexString("0x00abcd", "bytes2")
"0xabcd"
>>> h == "0xabcd"
True
>>> h == "0x0000aBcD"
True
>>> h == "potato"
File "<console>", line 1, in <module>
File "brownie/convert.py", line 327, in _hex_compare
  raise TypeError(f"Invalid type for comparison: '{b}' is not a valid hex string")
TypeError: Invalid type for comparison: 'potato' is not a valid hex string
```

class brownie.network.return_value.**ReturnValue**

Tuple subclass with limited dict-like functionality. Used for iterable return values from contract calls or event

logs.

```
>>> result = issuer.getCountry(784)
>>> result
(1, (0, 0, 0, 0), (100, 0, 0, 0))
>>> result[2]
(100, 0, 0, 0)
>>> result.dict()
{
    '_count': (0, 0, 0, 0),
    '_limit': (100, 0, 0, 0),
    '_minRating': 1
}
>>> result['_minRating']
1
```

When checking equality, `ReturnValue` objects ignore the type of container compared against. Tuples and lists will both return `True` so long as they contain the same values.

```
>>> result = issuer.getCountry(784)
>>> result
(1, (0, 0, 0, 0), (100, 0, 0, 0))
>>> result == (1, (0, 0, 0, 0), (100, 0, 0, 0))
True
>>> result == [1, [0, 0, 0, 0], [100, 0, 0, 0]]
True
```

classmethod `ReturnValue.`**dict**`()`
Returns a `dict` of the named values within the object.

classmethod `ReturnValue.`**items**`()`
Returns a set-like object providing a view on the object's named items.

classmethod `ReturnValue.`**keys**`()`
Returns a set-like object providing a view on the object's keys.

Internal Methods

Formatting Contract Data

The following methods are used to convert multiple values based on a contract ABI specification. Values are formatted via calls to the methods outlined under *type conversions*, and where appropriate *type classes* are applied.

`brownie.convert.`**_format_input**(*abi*, *inputs*) → 'ReturnValue'
Formats inputs based on a contract method ABI.

Returns

- `abi`: A contract method ABI as a dict.
- `inputs`: List or tuple of values to format.

Returns a tuple subclass (brownie.convert.ReturnValue) of values formatted for use by `ContractTx` or `ContractCall`.

Each value in `inputs` is converted using the one of the methods outlined in *Type Conversions*.

```
>>> from brownie.convert import format_input
>>> abi = {'constant': False, 'inputs': [{'name': '_to', 'type': 'address'}, {
↪'name': '_value', 'type': 'uint256'}], 'name': 'transfer', 'outputs': [{'name':
↪'', 'type': 'bool'}], 'payable': False, 'stateMutability': 'nonpayable', 'type
↪': 'function'}
>>> format_input(abi, ["0xB8c77482e45F1F44dE1745F52C74426C631bDD52","1 ether"])
('0xB8c77482e45F1F44dE1745F52C74426C631bDD52', 1000000000000000000)
```

brownie.convert.**_format_output**(*abi*, *outputs*) → 'ReturnValue'

Standardizes outputs from a contract call based on the contract's ABI.

Returns a tuple sublcass (brownie.convert.ReturnValue).

- `abi`: A contract method ABI as a dict.
- `outputs`: List or tuple of values to format.

Each value in `outputs` is converted using the one of the methods outlined in *Type Conversions*.

This method is called internally by `ContractCall` to ensure that contract output formats remain consistent, regardless of the RPC client being used.

```
>>> from brownie.convert import format_output
>>> abi = {'constant': True, 'inputs': [], 'name': 'name', 'outputs': [{'name': '
↪', 'type': 'string'}], 'payable': False, 'stateMutability': 'view', 'type':
↪'function'}
>>> format_output(abi, ["0x5465737420546f6b656e"])
('Test Token',)
```

brownie.convert.**_format_event**(*event*)

Standardizes outputs from an event fired by a contract.

- `event`: Decoded event data as given by the `decode_event` or `decode_trace` methods of the eth-event package.

The given event data is mutated in-place and returned. If an event topic is indexed, the type is changed to `bytes32` and `" (indexed)"` is appended to the name.

17.1.3 brownie.exceptions

The `exceptions` module contains all Brownie `Exception` classes.

exception brownie.exceptions.**CompilerError**

Raised by the compiler when there is an error within a contract's source code.

exception brownie.exceptions.**ContractExists**

Raised when attempting to create a new `Contract` or `ContractABI` object, when one already exists for the given address.

exception brownie.exceptions.**ContractNotFound**

Raised when attempting to access a `Contract` or `ContractABI` object that no longer exists because the local network was reverted.

exception brownie.exceptions.**EventLookupError**

Raised during lookup errors by `EventDict` and `_EventItem`.

exception brownie.exceptions.**IncompatibleEVMVersion**

Raised when attempting to deploy a contract that was compiled to target an EVM version that is imcompatible than the currently active local RPC client.

exception brownie.exceptions.**IncompatibleSolcVersion**
Raised when a project requires a version of solc that is not installed or not supported by Brownie.

exception brownie.exceptions.**InvalidManifest**
Raised when attempting to process an improperly formatted ethPM package.

exception brownie.exceptions.**MainnetUndefined**
Raised when an action requires interacting with the main-net, but no "mainnet" network is defined in brownie-config.yaml.

exception brownie.exceptions.**NamespaceCollision**
Raised by project.sources when the multiple source files contain a contract with the same name.

exception brownie.exceptions.**PragmaError**
Raised when a contract has no pragma directive, or a pragma which requires a version of solc that cannot be installed.

exception brownie.exceptions.**ProjectAlreadyLoaded**
Raised by project.load_project if a project has already been loaded.

exception brownie.exceptions.**ProjectNotFound**
Raised by project.load_project when a project cannot be found at the given path.

exception brownie.exceptions.**UndeployedLibrary**
Raised when attempting to deploy a contract that requires an unlinked library, but the library has not yet been deployed.

exception brownie.exceptions.**UnknownAccount**
Raised when the Accounts container cannot locate a specified Account object.

exception brownie.exceptions.**UnsetENSName**
Raised when an ENS name is unset (resolves to 0x00).

exception brownie.exceptions.**RPCConnectionError**
Raised when the RPC process is active and web3 is connected, but Brownie is unable to communicate with it.

exception brownie.exceptions.**RPCProcessError**
Raised when the RPC process fails to launch successfully.

exception brownie.exceptions.**RPCRequestError**
Raised when a direct request to the RPC client has failed, such as a snapshot or advancing the time.

exception brownie.exceptions.**VirtualMachineError**
Raised when a contract call causes the EVM to revert.

17.1.4 brownie._config

The _config module handles all Brownie configuration settings. It is not designed to be accessed directly. If you wish to view or modify config settings while Brownie is running, import brownie.config which will return a ConfigDict with the active settings:

```
>>> from brownie import config
>>> type(config)
<class 'brownie._config.ConfigDict'>
>>> config['network_defaults']
{'name': 'development', 'gas_limit': False, 'gas_price': False}
```

ConfigDict

class brownie._config.**ConfigDict**

Subclass of dict that prevents adding new keys when locked. Used to hold config file settings.

```
>>> from brownie.types import ConfigDict
>>> s = ConfigDict({'test': 123})
>>> s
{'test': 123}
```

ConfigDict Internal Methods

classmethod ConfigDict.**_lock**()

Locks the `ConfigDict`. When locked, attempts to add a new key will raise a `KeyError`.

```
>>> s._lock()
>>> s['other'] = True
Traceback (most recent call last):
File "brownie/types/types.py", line 18, in __setitem__
  raise KeyError("{} is not a known config setting".format(key))
KeyError: 'other is not a known config setting'
>>>
```

classmethod ConfigDict.**_unlock**()

Unlocks the `ConfigDict`. When unlocked, new keys can be added.

```
>>> s._unlock()
>>> s['other'] = True
>>> s
{'test': 123, 'other': True}
```

classmethod ConfigDict.**_copy**()

Returns a copy of the object as a `dict`.

17.1.5 brownie._singleton

class brownie._singleton.**_Singleton**

Internal metaclass used to create singleton objects. Instantiating a class derived from this metaclass will always return the same instance, regardless of how the child class was imported.

17.2 Network API

The `network` package holds classes for interacting with the Ethereum blockchain. This is the most extensive package within Brownie and contains the majority of the user-facing functionality.

17.2.1 brownie.network.main

The `main` module contains methods for conncting to or disconnecting from the network. All of these methods are available directly from `brownie.network`.

main.**connect**(*network: str = None*, *launch_rpc: bool = True*) → None
Connects to the network. Network settings are retrieved from `brownie-config.yaml`

- `network`: The network to connect to. If `None`, connects to the default network as specified in the config file.
- `launch_rpc`: If `True` and the configuration for this network includes `test_rpc` settings, attempts to launch or attach to a local RPC client. See *The Local RPC Client* for detailed information on the sequence of events in this process.

Calling this method is favored over calling `web3.connect` and `rpc.launch` or `rpc.attach` individually.

```
>>> from brownie import network
>>> network.connect('development')
```

main.**disconnect**(*kill_rpc: bool = True*) → None
Disconnects from the network.

The `Web3` provider is cleared, the active network is set to `None` and the local RPC client is terminated if it was launched as a child process.

```
>>> from brownie import network
>>> network.disconnect()
```

main.**is_connected**() → bool
Returns `True` if the `Web3` object is connected to the network.

```
>>> from brownie import network
>>> network.is_connected()
True
```

main.**show_active**() → Optional[str]
Returns the name of the network that is currently active, or `None` if not connected.

```
>>> from brownie import network
>>> network.show_active()
'development'
```

main.**gas_limit**(**args: Tuple[Union[int, str, bool, None]]*) → Union[int, bool]
Gets and optionally sets the default gas limit.

- If no argument is given, the current default is displayed.
- If an integer value is given, this will be the default gas limit.
- If set to `None`, `True` or `False`, the gas limit is determined automatically via `web3.eth.estimateGas`.

Returns `False` if the gas limit is set automatically, or an `int` if it is set to a fixed value.

```
>>> from brownie import network
>>> network.gas_limit()
False
>>> network.gas_limit(6700000)
6700000
>>> network.gas_limit(None)
False
```

main.**gas_price**(**args: Tuple[Union[int, str, bool, None]]*) → Union[int, bool]
Gets and optionally sets the default gas price.

- If an integer value is given, this will be the default gas price.
- If set to `None`, `True` or `False`, the gas price is determined automatically via `web3.eth.getPrice`.

Returns `False` if the gas price is set automatically, or an `int` if it is set to a fixed value.

```
>>> from brownie import network
>>> network.gas_price()
False
>>> network.gas_price(10000000000)
10000000000
>>> network.gas_price("1.2 gwei")
1200000000
>>> network.gas_price(False)
False
```

17.2.2 brownie.network.account

The `account` module holds classes for interacting with Ethereum accounts for which you control the private key.

Classes in this module are not meant to be instantiated directly. The `Accounts` container is available as `accounts` (or just `a`) and will create each `Account` automatically during initialization. Add more accounts using `Accounts.add`.

Accounts

class brownie.network.account.**Accounts**

List-like *Singleton* container that holds all of the available accounts as `Account` or `LocalAccount` objects. When printed it will display as a list.

```
>>> from brownie.network import accounts
>>> accounts
[<Account object '0x7Ebaa12c5d1EE7fD498b51d4F9278DC45f8D627A'>, <Account object
↪'0x186f79d227f5D819ACAB0C529031036D11E0a000'>, <Account object
↪'0xC53c27492193518FE9eBff00fd3CBEB6c434Cf8b'>, <Account object
↪'0x2929AF7BBCde235035ED72029c81b71935c49e94'>, <Account object
↪'0xb93538FEb07b3B8433BD394594cA3744f7ee2dF1'>, <Account object
↪'0x1E563DBB05A10367c51A751DF61167dE99A4d0A7'>, <Account object
↪'0xa0942deAc0885096D8400D3369dc4a2dde12875b'>, <Account object
↪'0xf427a9eC1d510D77f4cEe4CF352545071387B2e6'>, <Account object
↪'0x2308D528e4930EFB4aF30793A3F17295a0EFa886'>, <Account object
↪'0x2fb37EB570B1eE8Eda736c1BD1E82748Ec3d0Bf1'>]
>>> dir(accounts)
[add, at, clear, load, remove]
```

Accounts Methods

classmethod Accounts.**add**(*priv_key=None*)

Creates a new `LocalAccount` with private key `priv_key`, appends it to the container, and returns the new account instance. If no private key is entered, one is randomly generated via `os.urandom(8192)`.

```
>>> accounts.add()
<Account object '0xb094716BC0E9D3F3Fb42FF928bd76618435FeeAA'>
```

(continues on next page)

(continued from previous page)

```
>>> accounts.add('8fa2fdfb89003176a16b707fc860d0881da0d1d8248af210df12d37860996fb2
↪')
<Account object '0xc1826925377b4103cC92DeeCDF6F96A03142F37a'>
```

classmethod Accounts.**at**(*address*)

Given an address as a string, returns the corresponding `Account` or `LocalAccount` from the container.

```
>>> accounts.at('0xc1826925377b4103cC92DeeCDF6F96A03142F37a')
<Account object '0xc1826925377b4103cC92DeeCDF6F96A03142F37a'>
```

classmethod Accounts.**clear**()

Empties the container.

```
>>> accounts.clear()
```

classmethod Accounts.**load**(*filename=None*)

Decrypts a keystore file and returns a `LocalAccount` object.

Brownie will first attempt to find the keystore file as a path relative to the loaded project. If not found, it will look in the `brownie/data/accounts` folder within the Brownie package.

If filename is `None`, returns a list of available keystores in `brownie/data/accounts`.

```
>>> accounts.load()
['my_account']
>>> accounts.load('my_account')
Enter the password for this account:
<LocalAccount object '0xa9c2DD830DfFE8934fEb0A93BAbcb6e823e1FF05'>
```

classmethod Accounts.**remove**(*address*)

Removes an address from the container. The address may be given as a string or an `Account` instance.

```
>>> accounts.remove('0xc1826925377b4103cC92DeeCDF6F96A03142F37a')
```

Accounts Internal Methods

classmethod Accounts.**_reset**()

Called by *rpc._notify_registry* when the local chain has been reset. All `Account` objects are recreated.

classmethod Accounts.**_revert**(*height*)

Called by *rpc._notify_registry* when the local chain has been reverted to a block height greater than zero. Adjusts `Account` object nonce values.

Account

class brownie.network.account.**Account**

An ethereum address that you control the private key for, and so can send transactions from. Generated automatically from `web3.eth.accounts` and stored in the `Accounts` container.

```
>>> accounts[0]
<Account object '0x7Ebaa12c5d1EE7fD498b51d4F9278DC45f8D627A'>
>>> dir(accounts[0])
[address, balance, deploy, estimate_gas, nonce, transfer]
```

Account Attributes

`Account.`**`address`**
The public address of the account. Viewable by printing the class, you do not need to call this attribute directly.

```
>>> accounts[0].address
'0x7Ebaa12c5d1EE7fD498b51d4F9278DC45f8D627A'
```

`Account.`**`nonce`**
The current nonce of the address.

```
>>> accounts[0].nonce
0
```

Account Methods

`classmethod` `Account.`**`balance`**`()`
Returns the current balance at the address, in *wei*.

```
>>> accounts[0].balance()
100000000000000000000
>>> accounts[0].balance() == "100 ether"
True
```

`classmethod` `Account.`**`deploy`**(*contract*, **args*, *amount=None*, *gas_limit=None*, *gas_price=None*)
Deploys a contract.

- `contract`: A `ContractContainer` instance of the contract to be deployed.
- `*args`: Contract constructor arguments.
- `amount`: Amount of ether to send with the transaction. The given value is converted to *wei*.
- `gas_limit`: Gas limit for the transaction. The given value is converted to *wei*. If none is given, the price is set using `eth_estimateGas`.
- `gas_price`: Gas price for the transaction. The given value is converted to *wei*. If none is given, the price is set using `eth_gasPrice`.

Returns a `Contract` instance upon success. If the transaction reverts or you do not wait for a confirmation, a `TransactionReceipt` is returned instead.

```
>>> Token
[]
>>> t = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")

Transaction sent:␣
↪0x2e3cab83342edda14141714ced002e1326ecd8cded4cd0cf14b2f037b690b976
Transaction confirmed - block: 1   gas spent: 594186
Contract deployed at: 0x5419710735c2D6c3e4db8F30EF2d361F70a4b380
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>>
>>> t
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>> Token
[<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>]
>>> Token[0]
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
```

classmethod Account.**estimate_gas**(*to*, *amount*, *data=""*)

Estimates the gas required to perform a transaction. Raises a VirtualMachineError if the transaction would revert.

The returned value is given as an int denominated in wei.

- to: Recipient address. Can be an Account instance or string.
- amount: Amount of ether to send. The given value is converted to *wei*.
- data: Transaction data hexstring.

```
>>> accounts[0].estimate_gas(accounts[1], "1 ether")
21000
```

classmethod Account.**transfer**(*self*, *to*, *amount*, *gas_limit=None*, *gas_price=None*, *data=""*)

Broadcasts a transaction from this account.

- to: Recipient address. Can be an Account instance or string.
- amount: Amount of ether to send. The given value is converted to *wei*.
- gas_limit: Gas limit for the transaction. The given value is converted to *wei*. If none is given, the price is set using eth_estimateGas.
- gas_price: Gas price for the transaction. The given value is converted to *wei*. If none is given, the price is set using eth_gasPrice.
- data: Transaction data hexstring.

Returns a TransactionReceipt instance.

```
>>> accounts[0].transfer(accounts[1], "1 ether")

Transaction sent:␣
↪0x0173aa6938c3a5e50b6dc7b4d38e16dab40811ab4e00e55f3e0d8be8491c7852
Transaction confirmed - block: 1   gas used: 21000 (100.00%)
<Transaction object
↪'0x0173aa6938c3a5e50b6dc7b4d38e16dab40811ab4e00e55f3e0d8be8491c7852'>
```

LocalAccount

class brownie.network.account.**LocalAccount**

Functionally identical to Account. The only difference is that a LocalAccount is one where the private key was directly inputted, and so is not found in web3.eth.accounts.

Note: Resetting the RPC client will delete all LocalAccount objects from the Accounts container.

```
>>> accounts.add()
<LocalAccount object '0x716E8419F2926d6AcE07442675F476ace972C580'>
>>> accounts[-1]
<LocalAccount object '0x716E8419F2926d6AcE07442675F476ace972C580'>
```

LocalAccount Attributes

LocalAccount.**public_key**

The local account's public key as a string.

```
>>> accounts[-1].public_key

↪'0x34b51e2913f5771acdddea7d353404f844b02a39ad4003c08afaa729993c43e890181327beaf352d81424cd277f
↪'
```

LocalAccount.**private_key**
The local account's private key as a string.

```
>>> accounts[-1].private_key
'0xd289bec8d9ad145aead13911b5bbf01936cbcd0efa0e26d5524b5ad54a61aeb8'
```

LocalAccount Methods

classmethod LocalAccount.**save**(*filename*, *overwrite=False*)
Saves the account's private key in an encrypto keystore file.

If the filename does not include a folder, the keystore is saved in the `brownie/data/accounts` folder within the Brownie package.

Returns the absolute path to the keystore file, as a string.

```
>>> accounts[-1].save('my_account')
Enter the password to encrypt this account with:
/python3.6/site-packages/brownie/data/accounts/my_account.json
>>>
>>> accounts[-1].save('~/my_account.json')
Enter the password to encrypt this account with:
/home/computer/my_account.json
```

PublicKeyAccount

class brownie.network.account.**PublicKeyAccount**
Object for interacting with an Ethereum account where you do not control the private key. Can be used to check balances or to send ether to that address.

```
>>> from brownie.network.account import PublicKeyAccount
>>> pub = PublicKeyAccount("0x14b0Ed2a7C4cC60DD8F676AE44D0831d3c9b2a9E")
<PublicKeyAccount object '0x14b0Ed2a7C4cC60DD8F676AE44D0831d3c9b2a9E'>
```

Along with regular addresses, `PublicKeyAccount` objects can be instantiated using ENS domain names. The returned object will have the resolved address.

```
>>> PublicKeyAccount("ens.snakecharmers.eth")
<PublicKeyAccount object '0x808B53bF4D70A24bA5cb720D37A4835621A9df00'>
```

classmethod PublicKeyAccount.**balance**()
Returns the current balance at the address, in *wei*.

```
>>> pub.balance()
1000000000000000000
```

PublicKeyAccount.**nonce**
The current nonce of the address.

```
>>> accounts[0].nonce
0
```

17.2.3 brownie.network.alert

The `alert` module is used to set up notifications and callbacks based on state changes in the blockchain.

Alert

Alerts and callbacks are handled by creating instances of the `Alert` class.

class brownie.network.alert.**Alert**(*fn*, *args=None*, *kwargs=None*, *delay=2*, *msg=None*, *callback=None*, *repeat=False*)

An alert object. It is active immediately upon creation of the instance.

- `fn`: A callable to check for the state change.
- `args`: Arguments to supply to the callable.
- `kwargs`: Keyword arguments to supply to the callable.
- `delay`: Number of seconds to wait between checking for changes.
- `msg`: String to display upon change. The string will have `.format(initial_value, new_value)` applied before displaying.
- `callback`: A callback function to call upon a change in value. It should accept two arguments, the initial value and the new value.
- `repeat`: If `False`, the alert will terminate after the first time it first. if `True`, it will continue to fire with each change until it is stopped via `Alert.stop()`. If an `int` value is given, it will fire a total of `n+1` times before terminating.

Alerts are **non-blocking**, threading is used to monitor changes. Once an alert has finished running it cannot be restarted.

A basic example of an alert, watching for a changed balance:

```
>>> from brownie.network.alert import Alert
>>> Alert(accounts[1].balance, msg="Account 1 balance has changed from {} to {}")
<brownie.network.alert.Alert object at 0x7f9fd25d55f8>

>>> alert.show()
[<brownie.network.alert.Alert object at 0x7f9fd25d55f8>]
>>> accounts[2].transfer(accounts[1], "1 ether")

Transaction sent:␣
↪0x912d6ac704e7aaac01be159a4a36bbea0dc0646edb205af95b6a7d20945a2fd2
Transaction confirmed - block: 1   gas spent: 21000
<Transaction object
↪'0x912d6ac704e7aaac01be159a4a36bbea0dc0646edb205af95b6a7d20945a2fd2'>
ALERT: Account 1 balance has changed from 100000000000000000000 to␣
↪101000000000000000000
```

This example uses the alert's callback function to perform a token transfer, and sets a second alert to watch for the transfer:

```
>>> alert.new(accounts[3].balance, msg="Account 3 balance has changed from {} to
↪{}")
<brownie.network.alert.Alert object at 0x7fc743e415f8>

>>> def on_receive(old_value, new_value):
...     accounts[2].transfer(accounts[3], new_value-old_value)

>>> alert.new(accounts[2].balance, callback=on_receive)
<brownie.network.alert.Alert object at 0x7fc743e55cf8>
>>> accounts[1].transfer(accounts[2],"1 ether")

Transaction sent:␣
↪0xbd1bade3862f181359f32dac02ffd1d145fdfefc99103ca0e3d28ffc7071a9eb
Transaction confirmed - block: 1   gas spent: 21000
<Transaction object
↪'0xbd1bade3862f181359f32dac02ffd1d145fdfefc99103ca0e3d28ffc7071a9eb'>

Transaction sent:␣
↪0x8fcd15e38eed0a5c9d3d807d593b0ea508ba5abc892428eb2e0bb0b8f7dc3083
Transaction confirmed - block: 2   gas spent: 21000
ALERT: Account 3 balance has changed from 100000000000000000000 to␣
↪101000000000000000000
```

classmethod Alert.**is_alive**()

Returns a boolean indicating if an alert is currently running.

```
>>> a.is_alive()
True
```

classmethod Alert.**wait**(*timeout=None*)

Blocks until an alert has completed firing or the timeout value is reached. Similar to `Thread.join()`.

```
>>> a.wait()
```

classmethod Alert.**stop**(*wait=True*)

Stops the alert.

```
>>> alert_list = alert.show()
[<brownie.network.alert.Alert object at 0x7f9fd25d55f8>]
>>> alert_list[0].stop()
>>> alert.show()
[]
```

Module Methods

alert.**new**(*fn*, *args=[]*, *kwargs={}*, *delay=0.5*, *msg=None*, *callback=None*, *repeat=False*)

Alias for creating a new `Alert` instance.

```
>>> from brownie import alert
>>> alert.new(accounts[3].balance, msg="Account 3 balance has changed from {} to
↪{}")
<brownie.network.alert.Alert object at 0x7fc743e415f8>
```

alert.**show**()

Returns a list of all currently active alerts.

```
>>> alert.show()
[<brownie.network.alert.Alert object at 0x7f9fd25d55f8>]
```

alert.**stop_all**()
Stops all currently active alerts.

```
>>> alert.show()
[<brownie.network.alert.Alert object at 0x7f9fd25d55f8>]
>>> alert.stop_all()
>>> alert.show()
[]
```

17.2.4 brownie.network.contract

The contract module contains classes for deploying and interacting with smart contracts.

When a project is loaded, Brownie automatically creates ContractContainer instances from on the files in the contracts/ folder. New ProjectContract instances are created via methods in the container.

If you wish to interact with a contract outside of a project where only the ABI is available, use the Contract class.

Arguments supplied to calls or transaction methods are converted using the methods outlined in the *convert* module.

Note: On networks where persistence is enabled, ProjectContract instances will remain between sessions. Use ContractContainer.remove to delete these objects when they are no longer needed. See *Interacting with Non-Local Networks* for more information.

ContractContainer

class brownie.network.contract.**ContractContainer**
A list-like container class that holds all ProjectContract instances of the same type, and is used to deploy new instances of that contract.

```
>>> Token
[]
>>> dir(Token)
[abi, at, bytecode, deploy, remove, signatures, topics, tx]
```

ContractContainer Attributes

ContractContainer.**abi**
The ABI of the contract.

```
>>> Token.abi
[{'constant': True, 'inputs': [], 'name': 'name', 'outputs': [{'name': '', 'type
↪': 'string'}], 'payable': False, 'stateMutability': 'view', 'type': 'function'},
↪ {'constant': False, 'inputs': [{'name': '_spender', 'type': 'address'}, {'name
↪': '_value', 'type': 'uint256'}], 'name': 'approve', 'outputs': [{'name': '',
↪'type': 'bool'}], 'payable': False, 'stateMutability': 'nonpayable', 'type':
↪'function'}, ... ]
```

`ContractContainer.`**`bytecode`**
The bytecode of the contract, without any applied constructor arguments.

```
>>> Token.bytecode

↪'60806040523480156100105760008fd5b506040516107873803806107878339810160409081528151602080840151
↪...
```

`ContractContainer.`**`signatures`**
A dictionary of bytes4 signatures for each contract method.

If you have a signature and need to find the method name, use `ContractContainer.get_method`.

```
>>> Token.signatures
{
    'allowance': "0xdd62ed3e",
    'approve': "0x095ea7b3",
    'balanceOf': "0x70a08231",
    'decimals': "0x313ce567",
    'name': "0x06fdde03",
    'symbol': "0x95d89b41",
    'totalSupply': "0x18160ddd",
    'transfer': "0xa9059cbb",
    'transferFrom': "0x23b872dd"
}
>>> Token.signatures.keys()
dict_keys(['name', 'approve', 'totalSupply', 'transferFrom', 'decimals',
↪'balanceOf', 'symbol', 'transfer', 'allowance'])
>>> Token.signatures['transfer']
0xa9059cbb
```

`ContractContainer.`**`topics`**
A dictionary of bytes32 topics for each contract event.

```
>>> Token.topics
{
    'Approval':
↪"0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925",
    'Transfer':
↪"0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef"
}
>>> Token.topics.keys()
dict_keys(['Transfer', 'Approval'])
>>> Token.topics['Transfer']
0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef
```

ContractContainer Methods

`classmethod` `ContractContainer.`**`deploy`**`(`*`*args`*`)`
Deploys the contract.

- `*args`: Contract constructor arguments.

You can optionally include a dictionary of transaction parameters as the final argument. If you omit this or do not specify a `'from'` value, the transaction will be sent from the same address that deployed the contract.

If the contract requires a library, the most recently deployed one will be used. If the required library has not been deployed yet an `UndeployedLibrary` exception is raised.

Returns a `ProjectContract` object upon success.

In the console if the transaction reverts or you do not wait for a confirmation, a `TransactionReceipt` is returned instead.

```
>>> Token
[]
>>> Token.deploy
<ContractConstructor object 'Token.constructor(string,string,uint256,uint256)'>
>>> t = Token.deploy("Test Token", "TST", 18, "1000 ether", {'from': accounts[1]})

Transaction sent:␣
↪0x2e3cab83342edda14141714ced002e1326ecd8cded4cd0cf14b2f037b690b976
Transaction confirmed - block: 1   gas spent: 594186
Contract deployed at: 0x5419710735c2D6c3e4db8F30EF2d361F70a4b380
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>>
>>> t
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
>>> Token
[<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>]
>>> Token[0]
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
```

classmethod ContractContainer.**at**(*address*, *owner=None*)

Returns a `ProjectContract` instance.

- `address`: Address where the contract is deployed. Raises a ValueError if there is no bytecode at the address.
- `owner`: `Account` instance to set as the contract owner. If transactions to the contract do not specify a `'from'` value, they will be sent from this account.

```
>>> Token
[<Token Contract object '0x79447c97b6543F6eFBC91613C655977806CB18b0'>]
>>> Token.at('0x79447c97b6543F6eFBC91613C655977806CB18b0')
<Token Contract object '0x79447c97b6543F6eFBC91613C655977806CB18b0'>
>>> Token.at('0xefb1336a2E6B5dfD83D4f3a8F3D2f85b7bfb61DC')
File "brownie/lib/console.py", line 82, in _run
    exec('_result = ' + cmd, self.__dict__, local_)
File "<string>", line 1, in <module>
File "brownie/lib/components/contract.py", line 121, in at
    raise ValueError("No contract deployed at {}".format(address))
ValueError: No contract deployed at 0xefb1336a2E6B5dfD83D4f3a8F3D2f85b7bfb61DC
```

classmethod ContractContainer.**get_method**(*calldata*)

Given the call data of a transaction, returns the name of the contract method as a string.

```
>>> tx = Token[0].transfer(accounts[1], 1000)

Transaction sent:␣
↪0xc1fe0c7c8fd08736718aa9106662a635102604ea6db4b63a319e43474de0b420
Token.transfer confirmed - block: 3   gas used: 35985 (26.46%)
<Transaction object
↪'0xc1fe0c7c8fd08736718aa9106662a635102604ea6db4b63a319e43474de0b420'>
>>> tx.input
0xa9059cbb000000000000000000000000066ace0365c25329a407002d22908e25adeacb9bb000000000000000000000000
>>> Token.get_method(tx.input)
transfer
```

classmethod ContractContainer.**remove**(*address*)
Removes a contract instance from the container.

```
>>> Token
[<Token Contract object '0x79447c97b6543F6eFBC91613C655977806CB18b0'>]
>>> Token.remove('0x79447c97b6543F6eFBC91613C655977806CB18b0')
>>> Token
[]
```

ContractContainer Internal Methods

classmethod ContractContainer.**_reset**()
Called by *rpc._notify_registry* when the local chain has been reset. All Contract objects are removed from the container and marked as *reverted*.

classmethod ContractContainer.**_revert**(*height*)
Called by *rpc._notify_registry* when the local chain has been reverted to a block height greater than zero. Any Contract objects that no longer exist are removed from the container and marked as *reverted*.

Contract and ProjectContract

Contract and ProjectContract are both used to call or send transactions to smart contracts.

- Contract objects are instantiated directly and only require an ABI. They are used for calls to existing contracts that exist outside of a project.
- ProjectContract objects are created by calls to ContractContainer.deploy. Because they are compiled and deployed directly by Brownie, they provide much greater debugging capability.

These classes have identical APIs.

class brownie.network.contract.**Contract**(*name*, *address=None*, *abi=None*, *manifest_uri=None*, *owner=None*)
A deployed contract. This class allows you to call or send transactions to the contract.

- name: The name of the contract.
- address: Address of the contract. Required unless a manifest_uri is given.
- abi: ABI of the contract. Required unless a manifest_uri is given.
- manifest_uri: EthPM registry manifest uri. If given, the ABI (and optionally the contract address) are retrieved from here.
- owner: An optional Account instance. If given, transactions to the contract are sent broadcasted from this account by default.

```
>>> from brownie import Contract
>>> Contract('0x79447c97b6543F6eFBC91613C655977806CB18b0', "Token", abi)
<Token Contract object '0x79447c97b6543F6eFBC91613C655977806CB18b0'>
```

class brownie.network.contract.**ProjectContract**
A deployed contract that is part of an active Brownie project. Along with making calls and transactions, this object allows access to Brownie's full range of debugging and testing capability.

```
>>> Token[0]
<Token Contract object '0x79447c97b6543F6eFBC91613C655977806CB18b0'>
```

(continues on next page)

(continued from previous page)

```
>>> dir(Token[0])
[abi, allowance, approve, balance, balanceOf, bytecode, decimals, name,␣
↪signatures, symbol, topics, totalSupply, transfer, transferFrom, tx]
```

Contract Attributes

`Contract.`**`bytecode`**
The bytecode of the deployed contract, including constructor arguments.

```
>>> Token[0].bytecode

↪'6080604052600436106100985763ffffffff7c0100000000000000000000000000000000000000000000000000000000000000
↪..
```

`Contract.`**`tx`**
The `TransactionReceipt` of the transaction that deployed the contract. If the contract was not deployed during this instance of brownie, it will be `None`.

```
>>> Token[0].tx
<Transaction object
↪'0xcede03c7e06d2b4878438b08cd0cf4515942b3ba06b3cfd7019681d18bb8902c'>
```

Contract Methods

`classmethod` `Contract.`**`balance`**`()`
Returns the current balance at the contract address, in *wei*.

```
>>> Token[0].balance
0
```

Contract Internal Attributes

`Contract.`**`_reverted`**
Boolean. Once set to to `True`, any attempt to interact with the object raises a `ContractNotFound` exception. Set as a result of a call to *rpc._notify_registry*.

ContractCall

`class` `brownie.network.contract.`**`ContractCall`**`(`*args*`)`
Calls a non state-changing contract method without broadcasting a transaction, and returns the result. `args` must match the required inputs for the method.

The expected inputs are shown in the method's `__repr__` value.

Inputs and return values are formatted via methods in the *convert* module. Multiple values are returned inside a *ReturnValue*.

```
>>> Token[0].allowance
<ContractCall object 'allowance(address,address)'>
```

(continues on next page)

(continued from previous page)

```
>>> Token[0].allowance(accounts[0], accounts[2])
0
```

ContractCall Attributes

`ContractCall.`**`abi`**
The contract ABI specific to this method.

```
>>> Token[0].allowance.abi
{
    'constant': True,
    'inputs': [{'name': '_owner', 'type': 'address'}, {'name': '_spender', 'type
↪': 'address'}],
    'name': "allowance",
    'outputs': [{'name': '', 'type': 'uint256'}],
    'payable': False,
    'stateMutability': "view",
    'type': "function"
}
```

`ContractCall.`**`signature`**
The bytes4 signature of this method.

```
>>> Token[0].allowance.signature
'0xdd62ed3e'
```

ContractCall Methods

`classmethod` `ContractCall.`**`transact`**`(*args)`
Sends a transaction to the method and returns a `TransactionReceipt`.

```
>>> tx = Token[0].allowance.transact(accounts[0], accounts[2])

Transaction sent:␣
↪0xc4f3a0addfe1e475c2466f30c750ca7a60450132b07102af610d8d56f170046b
Token.allowance confirmed - block: 2   gas used: 24972 (19.98%)
<Transaction object
↪'0xc4f3a0addfe1e475c2466f30c750ca7a60450132b07102af610d8d56f170046b'>
>>> tx.return_value
0
```

ContractTx

`class` `brownie.network.contract.`**`ContractTx`**`(*args)`
Broadcasts a transaction to a potentially state-changing contract method. Returns a `TransactionReceipt`.

The given `args` must match the required inputs for the method. The expected inputs are shown in the method's `__repr__` value.

Inputs are formatted via methods in the *convert* module.

You can optionally include a dictionary of transaction parameters as the final argument. If you omit this or do not specify a `'from'` value, the transaction will be sent from the same address that deployed the contract.

```
>>> Token[0].transfer
<ContractTx object 'transfer(address,uint256)'>
>>> Token[0].transfer(accounts[1], 100000, {'from':accounts[0]})

Transaction sent:␣
↪0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0
Transaction confirmed - block: 2   gas spent: 51049
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
```

ContractTx Attributes

`ContractTx.`**`abi`**
The contract ABI specific to this method.

```
>>> Token[0].transfer.abi
{
    'constant': False,
    'inputs': [{'name': '_to', 'type': 'address'}, {'name': '_value', 'type':
↪'uint256'}],
    'name': "transfer",
    'outputs': [{'name': '', 'type': 'bool'}],
    'payable': False,
    'stateMutability': "nonpayable",
    'type': "function"
}
```

`ContractTx.`**`signature`**
The bytes4 signature of this method.

```
>>> Token[0].transfer.signature
'0xa9059cbb'
```

ContractTx Methods

`classmethod` `ContractTx.`**`call`**`(`**args*`)`
Calls the contract method without broadcasting a transaction, and returns the result.

Inputs and return values are formatted via methods in the *convert* module. Multiple values are returned inside a *ReturnValue*.

```
>>> Token[0].transfer.call(accounts[2], 10000, {'from': accounts[0]})
True
```

`classmethod` `ContractTx.`**`encode_input`**`(`**args*`)`
Returns a hexstring of ABI calldata that can be used to call the method with the given arguments.

```
>>> calldata = Token[0].transfer.encode_input(accounts[1], 1000)
0xa9059cbb000000000000000000000000d36bdba474b5b442310a5bfb989903020249bba000000000000000000000000
>>> accounts[0].transfer(Token[0], 0, data=calldata)

Transaction sent:␣
↪0x8dbf15878104571669f9843c18afc40529305ddb842f94522094454dcde22186
```

(continues on next page)

(continued from previous page)

```
Token.transfer confirmed - block: 2   gas used: 50985 (100.00%)
<Transaction object
↪'0x8dbf15878104571669f9843c18afc40529305ddb842f94522094454dcde22186'>
```

classmethod ContractTx.**decode_output**(*hexstr*)

Decodes raw hexstring data returned by this method.

```
>>>  Token[0].balanceOf.decode_output(
↪"0x0000000000000000000000000000000000000000000000003635c9adc5dea00000")
10000000000000000000
```

OverloadedMethod

class brownie.network.contract.**OverloadedMethod**(*address*, *name*, *owner*)

When a contract uses overloaded function names, the ContractTx or ContractCall objects are stored inside a dict-like OverloadedMethod container.

```
>>> erc223 = ERC223Token[0]
>>> erc223.transfer
<OverloadedMethod object 'ERC223Token.transfer'>
```

Individual methods are mapped to keys that correspond to the function input types. Input types can be given as a single comma-seperated string or a tuple of strings. uint and uint256 are equivalent.

```
>>> erc223.transfer['address,uint']
<ContractTx object 'transfer(address,uint256)'>

>>> erc223.transfer['address', 'uint256', 'uint256']
<ContractTx object 'transfer(address,uint256,uint256)'>
```

17.2.5 brownie.network.event

The event module contains classes and methods related to decoding transaction event logs. It is largely a wrapper around eth-event.

Brownie stores encrypted event topics in brownie/data/topics.json. The JSON file is loaded when this module is imported.

EventDict

class brownie.types.types.**EventDict**

Hybrid container type that works as a dict and a list. Base class, used to hold all events that are fired in a transaction.

When accessing events inside the object:

- If the key is given as an integer, events are handled as a list in the order that they fired. An _EventItem is returned for the specific event that fired at the given position.
- If the key is given as a string, a _EventItem is returned that contains all the events with the given name.

```
>>> tx
<Transaction object
↪'0xf1806643c21a69fcfa29187ea4d817fb82c880bcd7beee444ef34ea3b207cebe'>
>>> tx.events
{
    'CountryModified': [
        {
            'country': 1,
            'limits': (0, 0, 0, 0, 0, 0, 0, 0),
            'minrating': 1,
            'permitted': True
        },
            'country': 2,
            'limits': (0, 0, 0, 0, 0, 0, 0, 0),
            'minrating': 1,
            'permitted': True
        }
    ],
    'MultiSigCallApproved': {
        'callHash':
↪"0x0013ae2e37373648c5161d81ca78d84e599f6207ad689693d6e5938c3ae4031d",
        'caller': "0xf9c1fd2f0452fa1c60b15f29ca3250dfcb1081b9"
    }
}
>>> tx.events['CountryModified']
[
    {
        'country': 1,
        'limits': (0, 0, 0, 0, 0, 0, 0, 0),
        'minrating': 1,
        'permitted': True
    },
        'country': 2,
        'limits': (0, 0, 0, 0, 0, 0, 0, 0),
        'minrating': 1,
        'permitted': True
    }
]
>>> tx.events[0]
{
    'callHash':
↪"0x0013ae2e37373648c5161d81ca78d84e599f6207ad689693d6e5938c3ae4031d",
    'caller': "0xf9c1fd2f0452fa1c60b15f29ca3250dfcb1081b9"
}
```

classmethod EventDict.**count**(*name*)

Returns the number of events that fired with the given name.

```
>>> tx.events.count('CountryModified')
2
```

classmethod EventDict.**items**()

Returns a set-like object providing a view on the object's items.

classmethod EventDict.**keys**()

Returns a set-like object providing a view on the object's keys.

classmethod EventDict.**values**()

Returns an object providing a view on the object's values.

Internal Classes and Methods

_EventItem

class brownie.types.types.**_EventItem**

Hybrid container type that works as a dict and a list. Represents one or more events with the same name that were fired in a transaction.

Instances of this class are created by `EventDict`, it is not intended to be instantiated directly.

When accessing events inside the object:

- If the key is given as an integer, events are handled as a list in the order that they fired. An `_EventItem` is returned for the specific event that fired at the given position.
- If the key is given as a string, `_EventItem` assumes that you wish to access the first event contained within the object. `event['value']` is equivalent to `event[0]['value']`.

All values within the object are formatted by methods outlined in the *convert* module.

```
>>> event = tx.events['CountryModified']
<Transaction object
↪'0xf1806643c21a69fcfa29187ea4d817fb82c880bcd7beee444ef34ea3b207cebe'>
>>> event
[
    {
        'country': 1,
        'limits': (0, 0, 0, 0, 0, 0, 0, 0),
        'minrating': 1,
        'permitted': True
    },
        'country': 2,
        'limits': (0, 0, 0, 0, 0, 0, 0, 0),
        'minrating': 1,
        'permitted': True
    }
]
>>> event[0]
{
    'country': 1,
    'limits': (0, 0, 0, 0, 0, 0, 0, 0),
    'minrating': 1,
    'permitted': True
}
>>> event['country']
1
>>> event[1]['country']
2
```

_EventItem.**name**

The name of the event(s) contained within this object.

```
>>> tx.events[2].name
CountryModified
```

_EventItem.**pos**
A tuple giving the absolute position of each event contained within this object.

```
>>> event.pos
(1, 2)
>>> event[1].pos
(2,)
>>> tx.events[2] == event[1]
True
```

classmethod _EventItem.**items**()
Returns a set-like object providing a view on the items in the first event within this object.

classmethod _EventItem.**keys**()
Returns a set-like object providing a view on the keys in the first event within this object.

classmethod _EventItem.**values**()
Returns an object providing a view on the values in the first event within this object.

Internal Methods

brownie.network.event.**_get_topics**(*abi*)
Generates encoded topics from the given ABI, merges them with those already known in `topics.json`, and returns a dictioary in the form of `{'Name': "encoded topic hexstring"}`.

```
>>> from brownie.network.event import _get_topics
>>> abi = [{'name': 'Approval', 'anonymous': False, 'type': 'event', 'inputs': [{
↪'name': 'owner', 'type': 'address', 'indexed': True}, {'name': 'spender', 'type
↪': 'address', 'indexed': True}, {'name': 'value', 'type': 'uint256', 'indexed':␣
↪False}]}, {'name': 'Transfer', 'anonymous': False, 'type': 'event', 'inputs': [{
↪'name': 'from', 'type': 'address', 'indexed': True}, {'name': 'to', 'type':
↪'address', 'indexed': True}, {'name': 'value', 'type': 'uint256', 'indexed':␣
↪False}]}]
>>> _get_topics(abi)
{'Transfer': '0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef',
↪ 'Approval': '0x8c5be1e5ebec7d5bd14f71427d1e84f3dd0314c0f7b2291e5b200ac8c7c3b925
↪'}
```

brownie.network.event.**_decode_logs**(*logs*)
Given an array of logs as returned by `eth_getLogs` or `eth_getTransactionReceipt` RPC calls, returns an *EventDict*.

```
>>> from brownie.network.event import _decode_logs
>>> tx = Token[0].transfer(accounts[1], 100)

Transaction sent:␣
↪0xfefc3b7d912ed438b312414fb31d94ff757970f4d2e74dd0950d5c58cc23fdb1
Token.transfer confirmed - block: 2   gas used: 50993 (33.77%)
<Transaction object
↪'0xfefc3b7d912ed438b312414fb31d94ff757970f4d2e74dd0950d5c58cc23fdb1'>
>>> e = _decode_logs(tx.logs)
>>> repr(e)
<brownie.types.types.EventDict object at 0x7feed74aebe0>
>>> e
{
    'Transfer': {
```

(continues on next page)

(continued from previous page)

```
        'from': "0x1ce57af3672a16b1d919aeb095130ab288ca7456",
        'to': "0x2d72c1598537bcf4a4af97668b3a24e68b7d0cc5",
        'value': 100
    }
}
```

brownie.network.event.**_decode_trace**(*trace*)

Given the `structLog` from a `debug_traceTransaction` RPC call, returns an *EventDict*.

```
>>> from brownie.network.event import _decode_trace
>>> tx = Token[0].transfer(accounts[2], 1000, {'from': accounts[3]})

Transaction sent:␣
↪0xc6365b065492ea69ad3cbe26039a45a68b2e9ab9d29c2ff7d5d9162970b176cd
Token.transfer confirmed (Insufficient Balance) - block: 2   gas used: 23602 (19.
↪10%)
<Transaction object
↪'0xc6365b065492ea69ad3cbe26039a45a68b2e9ab9d29c2ff7d5d9162970b176cd'>
>>> e = _decode_trace(tx.trace)
>>> repr(e)
<brownie.types.types.EventDict object at 0x7feed74aebe0>
>>> e
{}
```

17.2.6 brownie.network.state

The `state` module contains classes to record transactions and contracts as they occur on the blockchain.

TxHistory

class brownie.network.state.**TxHistory**

List-like *Singleton* container that contains *TransactionReceipt* objects. Whenever a transaction is broadcast, the `TransactionReceipt` is automatically added.

```
>>> from brownie.network.state import TxHistory
>>> history = TxHistory()
>>> history
[]
>>> dir(history)
[copy, from_sender, of_address, to_receiver]
```

TxHistory Attributes

TxHistory.**gas_profile**

A dict that tracks gas cost statistics for contract function calls over time.

```
>>> history.gas_profile
{
    'Token.constructor': {
        'avg': 742912,
        'count': 1,
```

(continues on next page)

(continued from previous page)

```
        'high': 742912,
        'low': 742912
    },
    'Token.transfer': {
        'avg': 43535,
        'count': 2,
        'high': 51035,
        'low': 36035
    }
}
```

TxHistory Methods

classmethod TxHistory.**copy**()

Returns a shallow copy of the object as a `list`.

```
>>> history
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>]
>>> c = history.copy()
>>> c
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>]
>>> type(c)
<class 'list'>
```

classmethod TxHistory.**from_sender**(*account*)

Returns a list of transactions where the sender is `account`.

```
>>> history.from_sender(accounts[1])
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>]
```

classmethod TxHistory.**to_receiver**(*account*)

Returns a list of transactions where the receiver is `account`.

```
>>> history.to_receiver(accounts[2])
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>]
```

classmethod TxHistory.**of_address**(*account*)

Returns a list of transactions where `account` is the sender or receiver.

```
>>> history.of_address(accounts[1])
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>]
```

TxHistory Internal Methods

classmethod TxHistory.**_reset**()

Called by *rpc._notify_registry* when the local chain has been reset. All `TransactionReceipt` objects are removed from the container.

classmethod TxHistory.**_revert**(*height*)
Called by *rpc._notify_registry* when the local chain has been reverted to a block height greater than zero. Any TransactionReceipt objects that no longer exist are removed from the container.

Internal Methods

The internal methods in the state module are primarily used for tracking and adjusting Contract instances whenever the local RPC network is reverted or reset.

brownie.network.state.**_add_contract**(*contract*)
Adds a Contract or ProjectContract object to the global contract record.

brownie.network.state.**_find_contract**(*address*)
Given an address, returns the related Contract or ProjectContract object. If none exists, returns None.

This method is used internally by Brownie to locate a ProjectContract when the project it belongs to is unknown.

brownie.network.state.**_remove_contract**(*contract*)
Removes a Contract or ProjectContract object to the global contract record.

brownie.network.state.**_get_current_dependencies**()
Returns a list of the names of all currently deployed contracts, and of every contract that these contracts are dependent upon.

Used during testing to determine which contracts must change before a test needs to be re-run.

17.2.7 brownie.network.rpc

The rpc module contains the Rpc class, which is used to interact with ganache-cli when running a local RPC environment.

Note: Account balances, contract containers and transaction history are automatically modified when the local RPC is terminated, reset or reverted.

Rpc

class brownie.network.rpc.**Rpc**
Singleton object for interacting with ganache-cli when running a local RPC environment. When using the console or writing tests, an instance of this class is available as rpc.

```
>>> from brownie import rpc
>>> rpc
<lib.components.eth.Rpc object at 0x7ffb7cbab048>
>>> dir(rpc)
[is_active, kill, launch, mine, reset, revert, sleep, snapshot, time]
```

Rpc Methods

classmethod Rpc.**launch**(*cmd*)
Launches the local RPC client as a subprocess. cmd is the command string requiried to run it.

If the process cannot load successfully, raises brownie.RPCProcessError.

If a provider has been set in `Web3` but is unable to connect after launching, raises a `brownie.RPCConnectionError`.

```
>>> rpc.launch('ganache-cli')
Launching 'ganache-cli'...
```

classmethod `Rpc.`**attach**(*laddr*)

Attaches to an already running RPC client.

`laddr`: Address that the client is listening at. Can be supplied as a string `"http://127.0.0.1:8545"` or tuple `("127.0.0.1", 8545)`.

Raises a `ProcessLookupError` if the process cannot be found.

```
>>> rpc.attach('http://127.0.0.1:8545')
```

classmethod `Rpc.`**kill**(*exc=True*)

Kills the RPC subprocess. Raises `SystemError` if `exc` is `True` and the RPC is not currently active.

```
>>> rpc.kill()
Terminating local RPC client...
```

Note: Brownie registers this method with the atexit module. It is not necessary to explicitly kill `Rpc` before terminating a script or console session.

---

classmethod `Rpc.`**reset**()

Resets the RPC to the genesis state by loading a snapshot. This is NOT equivalent to calling `rpc.kill` and then `rpc.launch`.

```
>>> rpc.reset()
```

classmethod `Rpc.`**is_active**()

Returns a boolean indicating if the RPC process is currently active.

```
>>> rpc.is_active()
False
>>> rpc.launch()
>>> rpc.is_active()
True
```

classmethod `Rpc.`**is_child**()

Returns a boolean indicating if the RPC process is a child process of Brownie. If the RPC is not currently active, returns `False`.

```
>>> rpc.is_child()
True
```

classmethod `Rpc.`**evm_version**()

Returns the currently active EVM version as a string.

```
>>> rpc.evm_version()
'petersburg'
```

classmethod `Rpc.`**evm_compatible**(*version*)

Returns a boolean indicating if the given `version` is compatible with the currently active EVM version.

```
>>> rpc.evm_compatible('byzantium')
True
```

classmethod Rpc.**time**()

Returns the current epoch time in the RPC as an integer.

```
>>> rpc.time()
1550189043
```

classmethod Rpc.**sleep**(*seconds*)

Advances the RPC time. You can only advance the time by whole seconds.

```
>>> rpc.time()
1550189043
>>> rpc.sleep(100)
>>> rpc.time()
1550189143
```

classmethod Rpc.**mine**(*blocks=1*)

Forces new blocks to be mined.

```
>>> web3.eth.blockNumber
0
>>> rpc.mine()
Block height at 1
>>> web3.eth.blockNumber
1
>>> rpc.mine(3)
Block height at 4
>>> web3.eth.blockNumber
4
```

classmethod Rpc.**snapshot**()

Creates a snapshot at the current block height.

```
>>> rpc.snapshot()
Snapshot taken at block height 4
```

classmethod Rpc.**revert**()

Reverts the blockchain to the latest snapshot. Raises `ValueError` if no snapshot has been taken.

```
>>> rpc.snapshot()
Snapshot taken at block height 4
>>> accounts[0].balance()
100000000000000000000
>>> accounts[0].transfer(accounts[1], "10 ether")

Transaction sent:␣
↪0xd5d3b40eb298dfc48721807935eda48d03916a3f48b51f20bcded372113e1dca
Transaction confirmed - block: 5   gas used: 21000 (100.00%)
<Transaction object
↪'0xd5d3b40eb298dfc48721807935eda48d03916a3f48b51f20bcded372113e1dca'>
>>> accounts[0].balance()
89999580000000000000
>>> rpc.revert()
Block height reverted to 4
```

(continues on next page)

(continued from previous page)

```
>>> accounts[0].balance()
10000000000000000000
```

Rpc Internal Methods

classmethod Rpc.**_internal_snap**()
Takes an internal snapshot at the current block height.

classmethod Rpc.**_internal_revert**()
Reverts to the most recently taken internal snapshot.

Note: When calling this method, you must ensure that the user has not had a chance to take their own snapshot since `_internal_snap` was called.

Internal Methods

class brownie.network.rpc.**_revert_register**(*obj*)
Registers an object to be called whenever the local RPC is reset or reverted. Objects that register must include `_revert` and `_reset` methods in order to receive these callbacks.

class brownie.network.rpc.**_notify_registry**(*height*)
Calls each registered object's `_revert` or `_reset` method after the local state has been reverted.

17.2.8 brownie.network.transaction

The `transaction` module contains the `TransactionReceipt` class and related internal methods.

TransactionReceipt

class brownie.network.transaction.**TransactionReceipt**
An instance of this class is returned whenever a transaction is broadcasted. When printed in the console, the transaction hash will appear yellow if the transaction is still pending or red if the transaction caused the EVM to revert.

Many of the attributes return `None` while the transaction is still pending.

```
>>> tx = Token[0].transfer
<ContractTx object 'transfer(address,uint256)'>
>>> Token[0].transfer(accounts[1], 100000, {'from':accounts[0]})

Transaction sent:␣
↪0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0
Transaction confirmed - block: 2   gas spent: 51049
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> dir(tx)
[block_number, call_trace, contract_address, contract_name, error, events, fn_
↪name, gas_limit, gas_price, gas_used, info, input, logs, nonce, receiver␣
↪sender, status, txid, txindex, value]
```

(continues on next page)

(continued from previous page)

TransactionReceipt Attributes

`TransactionReceipt.`**`block_number`**
The block height at which the transaction confirmed.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.block_number
2
```

`TransactionReceipt.`**`contract_address`**
The address of the contract deployed as a result of this transaction, if any.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.contract_address
None
```

`TransactionReceipt.`**`contract_name`**
The name of the contract that was called or deployed in this transaction.

```
>>> tx
<Transaction object
↪'0xcdd07c6235bf093e1f30ac393d844550362ebb9b314b7029667538bfaf849749'>
>>> tx.contract_name
Token
```

`TransactionReceipt.`**`events`**
An *EventDict* of decoded event logs for this transaction.

Note: If you are connected to an RPC client that allows for `debug_traceTransaction`, event data is still available when the transaction reverts.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.events
{
    'Transfer': {
        'from': "0x94dd96c7e6012c927537cd789c48c42a1d1f790d",
        'to': "0xc45272e89a23d1a15a24041bce7bc295e79f2d13",
        'value': 100000
    }
}
```

`TransactionReceipt.`**`fn_name`**
The name of the function called by the transaction.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.fn_name
'transfer'
```

`TransactionReceipt.`**`gas_limit`**

The gas limit of the transaction, in wei as an `int`.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.gas_limit
150921
```

`TransactionReceipt.`**`gas_price`**

The gas price of the transaction, in wei as an `int`.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.gas_price
2000000000
```

`TransactionReceipt.`**`gas_used`**

The amount of gas consumed by the transaction, in wei as an `int`.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.gas_used
51049
```

`TransactionReceipt.`**`input`**

The complete calldata of the transaction as a hexstring.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.input

↪'0xa9059cbb000000000000000000000000031d504908351d2d87f3d6111f491f0b52757b592000000000000000000000
↪'
```

`TransactionReceipt.`**`logs`**

The raw event logs for the transaction. Not available if the transaction reverts.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.logs
[AttributeDict({'logIndex': 0, 'transactionIndex': 0, 'transactionHash': HexBytes(
↪'0xa8afb59a850adff32548c65041ec253eb64e1154042b2e01e2cd8cddb02eb94f'),
↪'blockHash': HexBytes(
↪'0x0b93b4cf230c9ef92b990de9cd62611447d83d396f1b13204d26d28bd949543a'),
↪'blockNumber': 6, 'address': '0x79447c97b6543F6eFBC91613C655977806CB18b0', 'data
↪':
↪'0x000000000000000000000000006b5132740b834674c3277aafa2c27898cbe740f6000000000000000000000000031d
↪', 'topics': [HexBytes(
↪'0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef')], 'type':
↪'mined'})]
```

(continues on next page)

(continued from previous page)

TransactionReceipt.**modified_state**

Boolean indicating if this transaction resuled in any state changes on the blockchain.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.modified_state
True
```

TransactionReceipt.**nonce**

The nonce of the transaction.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.nonce
2
```

TransactionReceipt.**receiver**

The address the transaction was sent to, as a string.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.receiver
'0x79447c97b6543F6eFBC91613C655977806CB18b0'
```

TransactionReceipt.**revert_msg**

The error string returned when a transaction causes the EVM to revert, if any.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.revert_msg
None
```

TransactionReceipt.**return_value**

The value returned from the called function, if any. Only available if the RPC client allows `debug_traceTransaction`.

If more then one value is returned, they are stored in a *ReturnValue*.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.return_value
True
```

TransactionReceipt.**sender**

The address the transaction was sent from. Where possible, this will be an Account instance instead of a string.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
```

(continues on next page)

(continued from previous page)

```
>>> tx.sender
<Account object '0x6B5132740b834674C3277aAfa2C27898CbE740f6'>
```

`TransactionReceipt.`**`status`**

The status of the transaction: -1 for pending, 0 for failed, 1 for success.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.status
1
```

`TransactionReceipt.`**`trace`**

An expanded transaction trace structLog, returned from the debug_traceTransaction RPC endpoint. If you are using Infura this attribute is not available.

Along with the standard data, the structLog also contains the following additional information:

- `address`: The address of the contract that executed this opcode
- `contractName`: The name of the contract
- `fn`: The name of the function
- `jumpDepth`: The number of jumps made since entering this contract. The initial function has a value of 1.
- `source`: The path and offset of the source code associated with this opcode.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> len(tx.trace)
239
>>> tx.trace[0]
{
    'address': "0x79447c97b6543F6eFBC91613C655977806CB18b0",
    'contractName': "Token",
    'depth': 0,
    'error': "",
    'fn': "Token.transfer",
    'gas': 128049,
    'gasCost': 22872,
    'jumpDepth': 1,
    'memory': [],
    'op': "PUSH1",
    'pc': 0,
    'source': {
        'filename': "contracts/Token.sol",
        'offset': [53, 2053]
    },
    'stack': [],
    'storage': {
    }
}
```

`TransactionReceipt.`**`txid`**

The transaction hash.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.txid
'0xa8afb59a850adff32548c65041ec253eb64e1154042b2e01e2cd8cddb02eb94f'
```

`TransactionReceipt.`**`txindex`**

The integer of the transaction's index position in the block.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.txindex
0
```

`TransactionReceipt.`**`value`**

The value of the transaction, in *wei*.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.value
0
```

TransactionReceipt Methods

`classmethod` `TransactionReceipt.`**`info`**`()`

Displays verbose information about the transaction, including event logs and the error string if a transaction reverts.

```
>>> tx = accounts[0].transfer(accounts[1], 100)
<Transaction object
↪'0x2facf2d1d2fdfa10956b7beb89cedbbe1ba9f4a2f0592f8a949d6c0318ec8f66'>
>>> tx.info()

Transaction was Mined
---------------------
Tx Hash: 0x2facf2d1d2fdfa10956b7beb89cedbbe1ba9f4a2f0592f8a949d6c0318ec8f66
From: 0x5fe657e72E76E7ACf73EBa6FA07ecB40b7312d80
To: 0x5814fC82d51732c412617Dfaecb9c05e3B823253
Value: 100
Block: 1
Gas Used: 21000

   Events In This Transaction
   --------------------------
   Transfer
      from: 0x5fe657e72E76E7ACf73EBa6FA07ecB40b7312d80
      to: 0x31d504908351d2d87f3d6111f491f0b52757b592
      value: 100
```

`classmethod` `TransactionReceipt.`**`call_trace`**`()`

Returns the sequence of contracts and functions called while executing this transaction, and the step indexes where each new method is entered and exitted. Any functions that terminated with `REVERT` or `INVALID` opcodes are highlighted in red.

```
>>> tx = Token[0].transferFrom(accounts[2], accounts[3], "10000 ether")

Transaction sent:␣
↪0x0d96e8ceb555616fca79dd9d07971a9148295777bb767f9aa5b34ede483c9753
Token.transferFrom confirmed (reverted) - block: 4   gas used: 25425 (26.42%)

>>> tx.call_trace()
Call trace for '0x0d96e8ceb555616fca79dd9d07971a9148295777bb767f9aa5b34ede483c9753
↪':
Token.transfer 0:244  (0x4A32104371b05837F2A36dF6D850FA33A92a178D)
└─Token.transfer 72:226
  ├─SafeMath.sub 100:114
  └─SafeMath.add 149:165
```

classmethod TransactionReceipt.**traceback**()

Returns an error traceback for the transaction, similar to a regular python traceback. If the transaction did not revert, returns an empty string.

```
>>> tx = >>> Token[0].transfer(accounts[1], "100000 ether")

Transaction sent:␣
↪0x9542e92a904e9d345def311ea52f22c3191816c6feaf7286f9b48081ab255ffa
Token.transfer confirmed (reverted) - block: 5   gas used: 23956 (100.00%)
<Transaction object
↪'0x9542e92a904e9d345def311ea52f22c3191816c6feaf7286f9b48081ab255ffa'>

>>> tx.traceback()
Traceback for '0x9542e92a904e9d345def311ea52f22c3191816c6feaf7286f9b48081ab255ffa
↪':
Trace step 99, program counter 1699:
  File "contracts/Token.sol", line 67, in Token.transfer:
    balances[msg.sender] = balances[msg.sender].sub(_value);
Trace step 110, program counter 1909:
  File "contracts/SafeMath.sol", line 9, in SafeMath.sub:
    require(b <= a);
```

classmethod TransactionReceipt.**error**(*pad=3*)

Displays the source code that caused the first revert in the transaction, if any.

- pad: Number of unrelated liness of code to include before and after the relevant source

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.error()
Source code for trace step 86:
  File "contracts/SafeMath.sol", line 9, in SafeMath.sub:

        c = a + b;
        require(c >= a);
    }
    function sub(uint a, uint b) internal pure returns (uint c) {
        require(b <= a);
        c = a - b;
    }
    function mul(uint a, uint b) internal pure returns (uint c) {
        c = a * b;
```

classmethod TransactionReceipt.**source**(*idx*, *pad=3*)

Displays the associated source code for a given stack trace step.

- idx: Stack trace step index
- pad: Number of unrelated liness of code to include before and after the relevant source

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.source(86)
Source code for trace step 86:
  File "contracts/SafeMath.sol", line 9, in SafeMath.sub:

        c = a + b;
        require(c >= a);
    }
    function sub(uint a, uint b) internal pure returns (uint c) {
        require(b <= a);
        c = a - b;
    }
    function mul(uint a, uint b) internal pure returns (uint c) {
        c = a * b;
```

17.2.9 brownie.network.web3

The web3 module contains a slightly modified version of the web3.py Web3 class that is used throughout various Brownie modules for RPC communication.

Web3

See the Web3 API documentation for detailed information on all the methods and attributes available here. This document only outlines methods that differ from the normal Web3 public interface.

class brownie.network.web3.**Web3**

Brownie subclass of Web3. An instance is created at brownie.network.web3.web and available for import from the main package.

```
>>> from brownie import web3
>>>
```

Web3 Methods

classmethod Web3.**connect**(*uri*)

Connects to a provider. uri can be the path to a local IPC socket, a websocket address beginning in ws:// or a URL.

```
>>> web3.connect('https://127.0.0.1:8545')
>>>
```

classmethod Web3.**disconnect**()

Disconnects from a provider.

```
>>> web3.disconnect()
>>>
```

Web3 Attributes

classmethod Web3.**chain_uri**()
Returns a BIP122 blockchain URI for the active chain.

```
>>> web3.chain_uri
'blockchain://a82ff4a4184a7b9e57aba1ae1ef91214c7d14a1040f4e1df8c0ec95f87a5bb62/
↪block/66760b538b3f02f6fbd4a745b3943af9fda982f2b8b26b502180ed96b2c7f52d'
```

classmethod Web3.**genesis_hash**()
Returns the hash of the genesis block for the active chain, as a string without a *0x* prefix.

```
>>> web3.genesis_hash
'41941023680923e0fe4d74a34bdac8141f2540e3ae90623718e47d66d1ca4a2d'
```

Web3 Internals

Web3.**_mainnet**
Provides access to a Web3 instance connected to the mainnet network as defined in the configuration file. Used internally for ENS and ethPM lookups.

Raises MainnetUndefined if the mainnet network is not defined.

Internal Methods

brownie.network.web3.**_resolve_address**(*address*)
Used internally for standardizing address inputs. If address is a string containing a . Brownie will attempt to resolve an ENS domain name address. Otherwise, returns the result of *brownie.convert.to_address*.

17.3 Project API

The project package contains methods for initializing, loading and compiling Brownie projects, and container classes to hold the data.

When Brownie is loaded from within a project folder, that project is automatically loaded and the ContractContainer objects are added to the __main__ namespace. Unless you are working with more than one project at the same time, there is likely no need to directly interact with the top-level Project object or any of the methods within this package.

Only the project.main module contains methods that directly interact with the filesystem.

17.3.1 brownie.project.main

The main module contains the high-level methods and classes used to create, load, and close projects. All of these methods are available directly from brownie.project.

Project

The `Project` class is the top level container that holds all objects related to a Brownie project.

Project Methods

classmethod `Project.load()` → None
Compiles the project source codes, instantiates `ContractContainer` objects, and populates the namespace.

Projects are typically loaded via `brownie.project.load()`, but if you have a `Project` object that was previously closed you can reload it using this method.

classmethod `Project.load_config()` → None
Updates the configuration settings from the `brownie-config.yaml` file within this project's root folder.

classmethod `Project.close`(*raises: bool = True*) → None
Removes this object and the related `ContractContainer` objects from the namespace.

```
>>> from brownie.project import TokenProject
>>> TokenProject.close()
>>> TokenProject
NameError: name 'TokenProject' is not defined
```

classmethod `Project.dict()`
Returns a dictionary of `ContractContainer` objects.

```
>>> from brownie.project import TokenProject
>>> TokenProject.dict()
{
   'Token': [],
   'SafeMath': []
}
```

TempProject

`TempProject` is a simplified version of `Project`, used to hold contracts that are compiled via `main.compile_sources`. Instances of this class are not included in the list of active projects or automatically placed anywhere within the namespace.

Module Methods

`main.check_for_project`(*path: Union[str, 'Path']*) → Optional[Path]
Checks for an existing Brownie project within a folder and it's parent folders, and returns the base path to the project as a `Path` object. Returns `None` if no project is found.

Accepts a path as a str or a `Path` object.

```
>>> from brownie import project
>>> Path('.').resolve()
PosixPath('/my_projects/token/build/contracts')
>>> project.check_for_project('.')
PosixPath('/my_projects/token')
```

`main.get_loaded_projects()` → List
Returns a list of currently loaded `Project` objects.

```
>>> from brownie import project
>>> project.get_loaded_projects()
[<Project object 'TokenProject'>, <Project object 'OtherProject'>]
```

main.**new**(*project_path="."*, *ignore_subfolder=False*)
Initializes a new project at the given path. If the folder does not exist, it will be created.

Returns the path to the project as a string.

```
>>> from brownie import project
>>> project.new('/my_projects/new_project')
'/my_projects/new_project'
```

main.**from_brownie_mix**(*project_name*, *project_path=None*, *ignore_subfolder=False*)
Initializes a new project via a template. Templates are downloaded from the Brownie Mix github repo.

If no path is given, the project will be initialized in a subfolder of the same name.

Returns the path to the project as a string.

```
>>> from brownie import project
>>> project.from_brownie_mix('token')
Downloading from https://github.com/brownie-mix/token-mix/archive/master.zip...
'my_projects/token'
```

main.from_ethpm(uri):
Generates a `TempProject` from an ethPM package.

- `uri`: ethPM manifest URI. Format can be ERC1319 or IPFS.

main.**load**(*project_path=None*, *name=None*)
Loads a Brownie project and instantiates various related objects.

- `project_path`: Path to the project. If `None`, attempts to find one using `check_for_project('.')`.
- `name`: Name to assign to the project. If None, the name is generated from the name of the project folder.

Returns a `Project` object. The same object is also available from within the `project` module namespce.

```
>>> from brownie import project
>>> project.load('/my_projects/token')
[<Project object 'TokenProject'>]
>>> project.TokenProject
<Project object 'TokenProject'>
>>> project.TokenProject.Token
<ContractContainer object 'Token'>
```

main.**compile_source**(*source*, *solc_version=None*, *optimize=True*, *runs=200*, *evm_version=None*)
Compiles the given Solidity source code string and returns a `TempProject` object.

```
>>> from brownie import compile_source
>>> container = compile_source('''pragma solidity 0.4.25;

contract SimpleTest {

  string public name;

  constructor (string _name) public {
```

(continues on next page)

(continued from previous page)

```
    name = _name;
  }
}'''
>>>
>>> container
<TempProject object>
>>> container.SimpleTest
<ContractContainer object 'SimpleTest'>
```

17.3.2 brownie.project.build

The `build` module contains classes and methods used internally by Brownie to interact with files in a project's `build/contracts` folder.

Build

The `Build` object is a container that stores and manipulates build data loaded from the `build/contracts/` files of a specific project. It is instantiated automatically when a project is opened, and available within the *Project* object as `Project._build`.

```
>>> from brownie.project import TokenProject
>>> TokenProject._build
<brownie.project.build.Build object at 0x7fb74cb1b2b0>
```

Build Methods

classmethod Build.**get**(*contract_name*)

Returns build data for the given contract name.

```
>>> from brownie.project import build
>>> build.get('Token')
{...}
```

classmethod Build.**items**(*path=None*)

Provides an list of tuples in the format `('contract_name', build_json)`, similar to calling `dict.items`. If a path is given, only contracts derived from that source file are returned.

```
>>> from brownie.project import build
>>> for name, data in build.items():
...     print(name)
Token
SafeMath
```

classmethod Build.**contains**(*contract_name*)

Checks if a contract with the given name is in the currently loaded build data.

```
>>> from brownie.project import build
>>> build.contains('Token')
True
```

classmethod Build.**get_dependents**(*contract_name*)
Returns a list of contracts that inherit or link to the given contract name. Used by the compiler when determining which contracts to recompile based on a changed source file.

```
>>> from brownie.project import build
>>> build.get_dependents('Token')
['SafeMath']
```

classmethod Build.**expand_build_offsets**(*build_json*)
Given a build json as a dict, expands the minified offsets to match the original source code.

Build Internal Methods

classmethod Build.**_add**(*build_json*)
Adds a contract's build data to the container.

classmethod Build.**_remove**(*contract_name*)
Removes a contract's build data from the container.

classmethod Build.**_generate_revert_map**(*pcMap*)
Adds a contract's dev revert strings to the revert map and it's `pcMap`. Called internally when adding a new contract.

The revert map is dict of tuples, where each key is a program counter that contains a `REVERT` or `INVALID` operation for a contract in the active project. When a transaction reverts, the dev revert string can be determined by looking up the final program counter in this mapping.

Each value is a 5 item tuple of: `("path/to/source", (start, stop), "function name", "dev: revert string", self._source)`

When two contracts have differing values for the same program counter, the value in the revert map is set to `False`. If a transaction reverts with this pc, the entire trace must be queried to determine which contract reverted and get the dev string from it's `pcMap`.

Internal Methods

The following methods exist outside the scope of individually loaded projects.

build.**_get_dev_revert**(*pc*)
Given the program counter from a stack trace that caused a transaction to revert, returns the *commented dev string* (if any). Used by `TransactionReceipt`.

```
>>> from brownie.project import build
>>> build.get_dev_revert(1847)
"dev: zero value"
```

build.**_get_error_source_from_pc**(*pc*)
Given the program counter from a stack trace that caused a transaction to revert, returns the highlighted relevent source code and the name of the method that reverted.

Used by `TransactionReceipt` when generating a `VirtualMachineError`.

17.3.3 brownie.project.compiler

The `compiler` module contains methods for compiling contracts, and formatting the compiled data. This module is used internally whenever a Brownie project is loaded.

In most cases you will not need to call methods in this module directly. Instead you should use `project.load` to compile your project initially and `project.compile_source` for adding individual, temporary contracts. Along with compiling, these methods also add the returned data to `project.build` and return `ContractContainer` objects.

Module Methods

`compiler.`**`set_solc_version`**(*version*)

Sets the `solc` version. If the requested version is not available it will be installed.

```
>>> from brownie.project import compiler
>>> compiler.set_solc_version("0.4.25")
Using solc version v0.4.25
```

`compiler.`**`install_solc`**(**versions*)

Installs one or more versions of `solc`.

```
>>> from brownie.project import compiler
>>> compiler.install_solc("0.4.25", "0.5.10")
```

`compiler.`**`compile_and_format`**(*contract_sources*, *solc_version=None*, *optimize=True*, *runs=200*, *evm_version=None*, *minify=False*, *silent=True*, *allow_paths=None*)

Given a dict in the format `{'path': "source code"}`, compiles the contracts and returns the formatted build data.

- `contract_sources`: `dict` in the format `{'path': "source code"}`
- `solc_version`: solc version to compile with. If `None`, each contract is compiled with the latest installed version that matches the pragma.
- `optimize`: Toggle compiler optimization
- `runs`: Number of compiler optimization runs
- `evm_version`: EVM version to target. If `None` the compiler default is used.
- `minify`: Should contract sources be minified?
- `silent`: Toggle console verbosity
- `allow_paths`: Import path, passed to *solc* as an additional path that contract files may be imported from

Calling this method is roughly equivalent to the following:

```
>>> from brownie.project import compiler

>>> input_json = compiler.generate_input_json(contract_sources)
>>> output_json = compiler.compile_from_input_json(input_json)
>>> build_json = compiler.generate_build_json(input_json, output_json)
```

`compiler.`**`find_solc_versions`**(*contract_sources*, *install_needed=False*, *install_latest=False*, *silent=True*)

Analyzes contract pragmas and determines which solc version(s) to use.

- `contract_sources`: `dict` in the format `{'path': "source code"}`
- `install_needed`: if `True`, solc is installed when no installed version matches a contract pragma
- `install_latest`: if `True`, solc is installed when a newer version is available than the installed one
- `silent`: enables verbose reporting

Returns a dict of {'version': ["path", "path", ..]}.

compiler.**find_best_solc_version**(*contract_sources*, *install_needed=False*, *install_latest=False*, *silent=True*)

Analyzes contract pragmas and finds the best version compatible with all sources.

- contract_sources: dict in the format {'path': "source code"}
- install_needed: if True, solc is installed when no installed version matches a contract pragma
- install_latest: if True, solc is installed when a newer version is available than the installed one
- silent: enables verbose reporting

Returns a dict of {'version': ["path", "path", ..]}.

compiler.**generate_input_json**(*contract_sources*, *optimize=True*, *runs=200*, *evm_version=None*, *minify=False*)

Generates a standard solc input JSON as a dict.

compiler.**compile_from_input_json**(*input_json*, *silent=True*, *allow_paths=None*)

Compiles from an input JSON and returns a standard solc output JSON as a dict.

compiler.**generate_build_json**(*input_json*, *output_json*, *compiler_data={}*, *silent=True*)

Formats input and output compiler JSONs and returns a Brownie build JSON dict.

- input_json: Compiler input JSON dict
- output_json: Computer output JSON dict
- compiler_data: Additional compiler data to include
- silent: Toggles console verbosity

Internal Methods

compiler.**_format_link_references**(*evm*)

Standardizes formatting for unlinked library placeholders within bytecode. Used internally to ensure that unlinked libraries are represented uniformly regardless of the compiler version used.

- evm: The 'evm' object from a compiler output JSON.

compiler.**_get_bytecode_hash**(*bytecode*)

Removes the final metadata from a bytecode hex string and returns a hash of the result. Used to check if a contract has changed when the source code is modified.

compiler.**_expand_source_map**(*source_map*)

Returns an uncompressed source mapping as a list of lists where no values are omitted.

```
>>> from brownie.project.compiler import expand_source_map
>>> expand_source_map("1:2:1:-;:9;2:1:2;;;")
[[1, 2, 1, '-'], [1, 9, 1, '-'], [2, 1, 2, '-'], [2, 1, 2, '-'], [2, 1, 2, '-'],␣
↪[2, 1, 2, '-']]
```

compiler.**_generate_coverage_data**(*source_map_str*, *opcodes_str*, *contract_node*, *stmt_nodes*, *branch_nodes*, *has_fallback*)

Generates the program counter and coverage maps that are used by Brownie for debugging and test coverage evaluation.

Takes the following arguments:

- source_map_str: deployed source mapping as given by the compiler
- opcodes_str: deployed bytecode opcodes string as given by the compiler

- `contract_node`: py-solc-ast contract node object
- `stmt_nodes`: list of statement node objects from `compiler.get_statment_nodes`
- `branch_nodes`: list of branch node objects from `compiler.get_branch_nodes`
- `has_fallback`: Bool, does this contract contain a fallback method?

Returns:

- `pc_list`: program counter map
- `statement_map`: statement coverage map
- `branch_map`: branch coverage map

`compiler._get_statement_nodes`(*source_nodes*)
Given a list of AST source node objects from py-solc-ast, returns a list of statement nodes. Used to generate the statement coverage map.

`compiler._get_branch_nodes`(*source_nodes*)
Given a list of AST source node objects from py-solc-ast, returns a list of branch nodes. Used to generate the branch coverage map.

17.3.4 `brownie.project.ethpm`

The `ethpm` module contains methods for interacting with ethPM manifests and registries. See *The Ethereum Package Manager* for more detailed information on how to access this functionality.

Module Methods

`ethpm.get_manifest`(*uri*)
Fetches an ethPM manifest and processes it for use with Brownie. A local copy is also stored if the given URI follows the ERC1319 spec.

- `uri`: URI location of the manifest. Can be IPFS or ERC1319.

`ethpm.process_manifest`(*manifest*, *uri*)
Processes a manifest for use with Brownie.

- `manifest`: ethPM manifest
- `uri`: IPFS uri of the package

`ethpm.get_deployment_addresses`(*manifest*, *contract_name*, *genesis_hash*)
Parses a manifest and returns a list of deployment addresses for the given contract and chain.

- `manifest`: ethPM manifest
- `contract_name`: Name of the contract
- `genesis_block`: Genesis block hash for the chain to return deployments on. If `None`, the currently active chain will be used.

`ethpm.get_installed_packages`(*project_path*)
Returns information on installed ethPM packages within a project.

- `project_path`: Path to the root folder of the project

Returns:

- `[(project name, version), ..]` of installed packages

- `[(project name, version), ..]` of installed-but-modified packages

ethpm.**install_package**(*project_path*, *uri*, *replace_existing*)
Installs an ethPM package within the project.

- `project_path`: Path to the root folder of the project
- `uri`: manifest URI, can be erc1319 or ipfs
- `replace_existing`: if True, existing files will be overwritten when installing the package

Returns the package name as a string.

ethpm.**remove_package**(*project_path*, *package_name*, *delete_files*)
Removes an ethPM package from a project.

- `project_path`: Path to the root folder of the project
- `package_name`: name of the package
- `delete_files`: if `True`, source files related to the package are deleted. Files that are still required by other installed packages will not be deleted.

Returns a boolean indicating if the package was installed.

ethpm.**create_manifest**(*project_path*, *package_config*, *pin_assets=False*, *silent=True*)
Creates a manifest from a project, and optionally pins it to IPFS.

- `project_path`: Path to the root folder of the project
- `package_config`: Configuration settings for the manifest
- `pin_assets`: if `True`, all source files and the manifest will be uploaded onto IPFS via Infura.

Returns: `(generated manifest, ipfs uri of manifest)`

ethpm.**verify_manifest**(*package_name*, *version*, *uri*)
Verifies the validity of a package at a given IPFS URI.

- `package_name`: Package name
- `version`: Package version
- `uri`: IPFS uri

Raises `InvalidManifest` if the manifest is not valid.

ethpm.**release_package**(*registry_address*, *account*, *package_name*, *version*, *uri*)
Creates a new release of a package at an ERC1319 registry.

- `registry_address`: Address of the registry
- `account`: `Account` object used to broadcast the transaction to the registry
- `package_name`: Name of the package
- `version`: Package version
- `uri`: IPFS uri of the package

Returns the `TransactionReceipt` of the registry call to release the package.

17.3.5 brownie.project.scripts

The `scripts` module contains methods for comparing, importing and executing python scripts related to a project.

scripts.**run**(*script_path*, *method_name="main"*, *args=None*, *kwargs=None*, *project=None*)
Imports a project script, runs a method in it and returns the result.

script_path: path of script to import method_name: name of method in the script to run args: method args kwargs: method kwargs project: Project object that should available for import into the script namespace

```
>>> from brownie import run
>>> run('token')

Running 'scripts.token.main'...

Transaction sent:␣
↪0xeb9dfb6d97e8647f824a3031bc22a3e523d03e2b94674c0a8ee9b3ff601f967b
Token.constructor confirmed - block: 1   gas used: 627391 (100.00%)
Token deployed at: 0x8dc446C44C821F27B333C1357990821E07189E35
```

Internal Methods

scripts.**_get_ast_hash**(*path*)
Returns a hash based on the AST of a script and any scripts that it imports. Used to determine if a project script has been altered since it was last run.

path: path of the script

```
>>> from brownie.project.scripts import get_ast_hash
>>> get_ast_hash('scripts/deploy.py')
'12b57e7bb8d88e3f289e27ba29e5cc28eb110e45'
```

17.3.6 brownie.project.sources

The sources module contains classes and methods to access project source code files and information about them.

Sources

The Sources object provides access to the contracts/ files for a specific project. It is instantiated automatically when a project is opened, and available within the *Project* object as Project._sources.

```
>>> from brownie.project import TokenProject
>>> TokenProject._sources
<brownie.project.sources.Sources object at 0x7fb74cb1bb70>
```

classmethod Sources.**get**(*name*)
Returns the source code file for the given name. name can be a path or a contract name.

```
>>> from brownie.project import sources
>>> sources.get('SafeMath')
"pragma solidity ^0.5.0; ..."
```

classmethod Sources.**get_path_list**()
Returns a list of contract source paths for the active project.

```
>>> from brownie.project import sources
>>> sources.get_path_list()
['contracts/Token.sol', 'contracts/SafeMath.sol']
```

classmethod Sources.**get_contract_list**()
Returns a list of contract names for the active project.

```
>>> from brownie.project import sources
>>> sources.get_contract_list()
['Token', 'SafeMath']
```

classmethod Sources.**get_source_path**(*contract_name*)
Returns the path to the file where a contract is located.

```
>>> from brownie.project import sources
>>> sources.get_source_path('Token')
'contracts/Token.sol'
```

classmethod Sources.**expand_offset**(*contract_name*, *offset*)
Converts a minified offset to one that matches the current source code.

```
>>> from brownie.project import sources
>>> sources.expand_offset("Token", [1258, 1466])
(2344, 2839)
```

Module Methods

sources.**minify**(*source*)
Given contract source as a string, returns a minified version and an offset map used internally to translate minified offsets to the original ones.

```
>>> from brownie.project import sources
>>> token_source = sources.get('Token')
>>> source.minify(token_source)
"pragma solidity^0.5.0;\nimport"./SafeMath.sol";\ncontract Token{\nusing SafeMath␣
↪for uint256; ..."
```

sources.**is_inside_offset**(*inner*, *outer*)
Returns a boolean indicating if the first offset is contained completely within the second offset.

```
>>> from brownie.project import sources
>>> sources.is_inside_offset([100, 200], [100, 250])
True
```

sources.**get_hash**(*source*, *contract_name*, *minified*)
Returns a sha1 hash generated from a contract's source code.

17.4 Test API

The test package contains classes and methods for running tests and evaluating test coverage.

This functionality is typically accessed via pytest. See *Unit Testing with Pytest*.

17.4.1 brownie.test.plugin

The plugin module contains classes and methods used in the Brownie Pytest plugin. It defines custom fixtures and handles integration into the Pytest workflow.

Pytest Fixtures

Brownie includes the following fixtures for use with pytest.

Note: These fixtures are only available when pytest is run from inside a Brownie project folder.

Session Fixtures

These fixtures provide access to objects related to the project being tested.

plugin.**accounts**
Session scope. Yields an instantiated *Accounts* container for the active project.

plugin.**a**
Session scope. Short form of the accounts fixture.

plugin.**history**
Session scope. Yields an instantiated *TxHistory* object for the active project.

plugin.**rpc**
Session scope. Yields an instantiated *Rpc* object.

plugin.**web3**
Session scope. Yields an instantiated *Web3* object.

Isolation Fixtures

These fixtures are used to effectively isolate tests. If included on every test within a module, that module may now be skipped via the --update flag when none of the related files have changed since it was last run.

plugin.**module_isolation**
Module scope. When used, this fixture is always applied before any other module-scoped fixtures.

Resets the local environment before starting the first test and again after completing the final test.

plugin.**fn_isolation**(*module_isolation*)
Function scope. When used, this fixture is always applied before any other function-scoped fixtures.

Applies the module_isolation fixture, and additionally takes a snapshot prior to running each test which is then reverted to after the test completes. The snapshot is taken immediately after any module-scoped fixtures are applied, and before all function-scoped ones.

Coverage Fixtures

These fixtures alter the behaviour of tests when coverage evaluation is active.

plugin.**no_call_coverage**
Function scope. Coverage evaluation will not be performed on called contact methods during this test.

plugin.**skip_coverage**
Function scope. If coverage evaluation is active, this test will be skipped.

RevertContextManager

The RevertContextManager closely mimics the behaviour of pytest.raises.

class plugin.**RevertContextManager**(*revert_msg=None*)
Context manager used to handle VirtualMachineError exceptions. Raises AssertionError if no transaction has reverted when the context closes.

- revert_msg: Optional. Raises an AssertionError if the transaction does not revert with this error string.

Available as pytest.reverts.

```
import pytest
from brownie import accounts

def test_transfer_reverts(Token, accounts):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, "1000 ether")
    with pytest.reverts():
        token.transfer(account[2], "10000 ether", {'from': accounts[1]})
```

17.4.2 brownie.test.output

The output module contains methods for formatting and displaying test output.

Internal Methods

output.**_save_coverage_report**(*build*, *coverage_eval*, *report_path*)
Generates and saves a test coverage report for viewing in the GUI.

- build: Project *Build* object
- coverage_eval: Coverage evaluation dict
- report_path: Path to save to. If the path is a folder, the report is saved as coverage.json.

output.**_print_gas_profile**()
Formats and prints a gas profile report.

output.**_print_coverage_totals**(*build*, *coverage_eval*)
Formats and prints a coverage evaluation report.

- build: Project *Build* object
- coverage_eval: Coverage evaluation dict

output.**_get_totals**(*build*, *coverage_eval*)
Generates an aggregated coverage evaluation dict that holds counts and totals for each contract function.

- build: Project *Build* object
- coverage_eval: Coverage evaluation dict

Returns:

```
{ "ContractName": {
    "statements": {
        "path/to/file": {
            "ContractName.functionName": (count, total), ..
        }, ..
    },
    "branches" {
        "path/to/file": {
            "ContractName.functionName": (true_count, false_count, total), ..
        }, ..
    }
}
```

output.**_split_by_fn**(*build*, *coverage_eval*)

Splits a coverage eval dict so that coverage indexes are stored by contract function. The returned dict is no longer compatible with other methods in this module.

- `build`: Project *Build* object
- `coverage_eval`: Coverage evaluation dict
- Original format: `{"path/to/file": [index, ..], .. }`
- Returned format: `{"path/to/file": { "ContractName.functionName": [index, .. ], .. }`

output.**_get_highlights**(*build*, *coverage_eval*)

Returns a highlight map formatted for display in the GUI.

- `build`: Project *Build* object
- `coverage_eval`: Coverage evaluation dict

Returns:

```
{
    "statements": {
        "ContractName": {"path/to/file": [start, stop, color, msg], .. },
    },
    "branches": {
        "ContractName": {"path/to/file": [start, stop, color, msg], .. },
    }
}
```

See the *Viewing Security Report Data* for more info on the return format.

17.4.3 brownie.test.coverage

The `coverage` module is used storing and accessing coverage evaluation data.

Module Methods

coverage.**get_coverage_eval**()

Returns all coverage data, active and cached.

coverage.**get_merged_coverage_eval**()

Merges and returns all active coverage data as a single dict.

coverage.**clear**()
Clears all coverage eval data.

Internal Methods

coverage.**add_transaction**(*txhash*, *coverage_eval*)
Adds coverage eval data.

coverage.**add_cached_transaction**(*txhash*, *coverage_eval*)
Adds coverage data to the cache.

coverage.**check_cached**(*txhash*, *active=True*)
Checks if a transaction hash is present within the cache, and if yes includes it in the active data.

coverage.**get_active_txlist**()
Returns a list of coverage hashes that are currently marked as active.

coverage.**clear_active_txlist**()
Clears the active coverage hash list.

17.4.4 brownie.test._manager

The _manager module contains the TestManager class, used internally by Brownie to determine which tests should run and to load and save the test results.

17.5 Utils API

The utils package contains utility classes and methods that are used throughout Brownie.

17.5.1 brownie.utils.color

The color module contains the Color class, used for to apply color and formatting to text before printing.

Color

class brownie.utils.color.**Color**
The Color class is used to apply color and formatting to text before displaying it to the user. It is primarily used within the console. An instance of Color is available at brownie.utils.color:

```
>>> from brownie.utils import color
>>> color
<brownie.utils.color.Color object at 0x7fa9ec851ba8>
```

Color is designed for use in formatted string literals. When called or accessed like a list, it returns an ANSI escape code for the given color:

```
>>> color('red')
'\x1b[0;31m'
>>> color['red']
'\x1b[0;31m'
```

You can also prefix any color with "bright" or "dark":

```
>>> color('bright red')
'\x1b[0;1;31m'
>>> color('dark red')
'\x1b[0;2;31m'
```

Calling it with no values or Converting to a string returns the base color code:

```
>>> color()
'\x1b[0;m'
>>> str(color)
'\x1b[0;m'
```

Color Methods

classmethod `Color.`**`pretty_dict`**(*value*, *_indent=0*) → str
Given a `dict`, returns a colored and formatted string suitable for printing.

- `value`: `dict` to format
- `_indent`: used for recursive internal calls, should always be left as `0`

classmethod `Color.`**`pretty_sequence`**(*value*, *_indent=0*) → str
Given a sequence (`list`, `tuple`, `set`), returns a colored and formatted string suitable for printing.

- `value`: Sequence to format
- `_indent`: used for recursive internal calls, should always be left as `0`

classmethod `Color.`**`format_tb`**(*exc*, *filename=None*, *start=None*, *stop=None*) → str
Given a raised `Exception`, returns a colored and formatted string suitable for printing.

- `exc`: An `Exception` object
- `filename`: An optional path as a string. If given, only lines in the traceback related to this filename will be displayed.
- `start`: Optional. If given, the displayed traceback not include items prior to this index.
- `stop`: Optional. If given, the displayed traceback not include items beyond this index.

classmethod `Color.`**`format_syntaxerror`**(*exc*) → str
Given a raised `SyntaxError`, returns a colored and formatted string suitable for printing.

- `exc`: A `SyntaxError` object.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N