Brownie Documentation

Release 1.13.0

Ben Hauser

Jan 09, 2021

Contents

CHAPTER 1

Brownie

Brownie is a Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine.

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.

1.1 Features

- Full support for Solidity and Vyper
- Contract testing via pytest, including trace-based coverage evaluation
- Property-based and stateful testing via hypothesis
- Powerful debugging tools, including python-style tracebacks and custom error strings
- Built-in console for quick project interaction
- Support for ethPM packages

CHAPTER 2

Quickstart

This page provides a quick overview of how to use Brownie. It relies mostly on examples and assumes a level of familiarity with Python and smart contract dvelopment. For more in-depth content, you should read the documentation sections under “Getting Started” in the table of contents.

If you have any questions about how to use Brownie, feel free to ask on Ethereum StackExchange or join us on Gitter.

2.1 Creating a New Project

Main article: Creating 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
```

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 will create a `token/` subdirectory, and download the template project within it.

2.2 Exploring the Project

Main article: Structure of a Project

Each Brownie project uses the following structure:

- `contracts/`: Contract sources
- `interfaces/`: Interface sources
- `scripts/`: Scripts for deployment and interaction

- `tests/`: Scripts for testing the project

The following directories are also created, and used internally by Brownie for managing the project. You should not edit or delete files within these folders.

- `build/`: Project data such as compiler artifacts and unit test results
- `reports/`: JSON report files for use in the GUI

2.3 Compiling your Contracts

> *Main article: Compiling Contracts*

To compile your project:

```
$ brownie compile
```

You will see the following output:

```
Brownie - Python development framework for Ethereum

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

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

Note: Brownie automatically compiles any new or changed source files each time it is loaded. You do not need to manually run the compiler.

2.4 Core Functionality

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 directory, load it by typing:

```
$ 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: 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 any class or method to view information on it's functionality.

2.4.1 Accounts

Main article: Gas Strategies

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
```

2.4.2 Contracts

Main article: Working with Contracts

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

```
>>> Token
[]

>>> Token.deploy
<ContractConstructor object 'Token.constructor(string _symbol, string _name, uint256 _
↪decimals, uint256 _totalSupply)'>

>>> t = Token.deploy("Test Token", "TST", 18, 1e20, {'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'>
```

(continues on next page)

(continued from previous page)

```
>>> t.balanceOf(accounts[1])
1000000000000000000000

>>> t.transfer
<ContractTx object 'transfer(address _to, uint256 _value)'>

>>> t.transfer(accounts[2], 1e20, {'from': accounts[1]})

Transaction sent: 0xcd98225a77409b8d81023a3a4be15832e763cd09c74ff431236bfc6d56a74532
Transaction confirmed - block: 2   gas spent: 51241
<Transaction object
↪'0xcd98225a77409b8d81023a3a4be15832e763cd09c74ff431236bfc6d56a74532'>

>>> t.balanceOf(accounts[1])
900000000000000000000

>>> t.balanceOf(accounts[2])
100000000000000000000
```

When a contract source includes NatSpec documentation, you can view it via the `ContractCall.info` method:

```
>>> t.transfer.info()
transfer(address _to, uint256 _value)
  @dev transfer token for a specified address
  @param _to The address to transfer to.
  @param _value The amount to be transferred.
```

2.4.3 Transactions

Main article: Inspecting and Debugging Transactions

The `TransactionReceipt` object contains all relevant information about a transaction, as well as various methods to aid in debugging.

```
>>> tx = Token[0].transfer(accounts[1], 1e18, {'from': accounts[0]})

Transaction sent: 0x0d96e8ceb555616fca79dd9d07971a9148295777bb767f9aa5b34ede483c9753
Token.transfer confirmed - block: 2   gas used: 51019 (33.78%)

>>> tx
<Transaction object
↪'0x0d96e8ceb555616fca79dd9d07971a9148295777bb767f9aa5b34ede483c9753'>
```

Use `TransactionReceipt.events` to examine the events that fired:

```
>>> len(tx.events)
1

>>> 'Transfer' in tx.events
True

>>> tx.events['Transfer']
{
    'from': "0x4fe357adbdb4c6c37164c54640851d6bff9296c8",
```

(continues on next page)

(continued from previous page)

```
    'to': "0xfae9bc8a468ee0d8c84ec00c8345377710e0f0bb",
    'value': "1000000000000000000",
}
```

To inspect the transaction trace:

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

For information on why a transaction reverted:

```
>>> tx = Token[0].transfer(accounts[1], 1e18, {'from': accounts[3]})

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

>>> tx.traceback()
Traceback for '0x5ff198f3a52250856f24792889b5251c120a9ecfb8d224549cb97c465c04262a':
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);
```

2.5 Writing Scripts

Main article: 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 how 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:

```python
from brownie import *

def main():
    Token.deploy("Test Token", "TEST", 18, 1e23, {'from': accounts[0]})
```

2.6 Testing your Project

Main article: Writing Unit Tests

Brownie uses the `pytest` framework for contract testing.

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

```
$ brownie test
```

2.6.1 Fixtures

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's automatically generated fixtures:

```python
def test_transfer(Token, accounts):
    token = Token.deploy("Test Token", "TST", 18, 1e20, {'from': accounts[0]})
    assert token.totalSupply() == 1e20

    token.transfer(accounts[1], 1e19, {'from': accounts[0]})
    assert token.balanceOf(accounts[1]) == 1e19
    assert token.balanceOf(accounts[0]) == 9e19
```

See the *Pytest Fixtures* section for a complete list of fixtures.

2.6.2 Handling Reverted Transactions

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

```python
import brownie

def test_transfer_reverts(accounts, Token):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, 1e23)
    with brownie.reverts():
        token.transfer(accounts[1], 1e24, {'from': accounts[0]})
```

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

```python
import brownie

def test_transfer_reverts(accounts, Token):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, 1e23)
    with brownie.reverts("Insufficient Balance"):
        token.transfer(accounts[1], 1e24, {'from': accounts[0]})
```

2.6.3 Isolating Tests

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. Because the `token` fixture uses a session scope, the transaction to deploy the contract is only executed once.

```python
import pytest
from brownie import accounts


@pytest.fixture(scope="module")
def token(Token):
    yield Token.deploy("Test Token", "TST", 18, 1e20, {'from': accounts[0]})


def test_transferFrom(fn_isolation, token):
    token.approve(accounts[1], 6e18, {'from': accounts[0]})
    token.transferFrom(accounts[0], accounts[2], 5e18, {'from': accounts[1]})

    assert token.balanceOf(accounts[2]) == 5e18
    assert token.balanceOf(accounts[0]) == 9.5e19
    assert token.allowance(accounts[0], accounts[1]) == 1e18


def test_balance_allowance(fn_isolation, token):
    assert token.balanceOf(accounts[0]) == 1e20
    assert token.allowance(accounts[0], accounts[1]) == 0
```

CHAPTER 3

Installing Brownie

The recommended way to install Brownie is via pipx. Pipx is a tool to help you install and run end-user applications written in Python. It's roughly similar to macOS's `brew`, JavaScript's `npx`, and Linux's `apt`.

`pipx` installs Brownie into a virtual environment and makes it available directly from the commandline. Once installed, you will never have to activate a virtual environment prior to using Brownie.

`pipx` does not ship with Python. If you have not used it before you will probably need to install it.

To install `pipx`:

```
python3 -m pip install --user pipx
python3 -m pipx ensurepath
```

Note: You may need to restart your terminal after installing `pipx`.

To install Brownie using `pipx`:

```
pipx install eth-brownie
```

Once installation is complete, type `brownie` to verify that it worked:

```
$ brownie
Brownie - Python development framework for Ethereum

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

3.1 Other Installation Methods

You can also install Brownie via `pip`, or clone the repository and use `setuptools`. If you install via one of these methods, we highly recommend using `venv` and installing into a virtual environment.

To install via `pip`:

```
pip install eth-brownie
```

To clone the github repository and install via `setuptools`:

```
git clone https://github.com/eth-brownie/brownie.git
cd brownie
python3 setup.py install
```

3.2 Dependencies

Brownie has the following dependencies:

- python3 version 3.6 or greater, python3-dev
- ganache-cli - tested with version 6.11.0

3.2.1 Tkinter

The Brownie GUI is built using the Tk GUI toolkit. Both Tk and tkinter are available on most Unix platforms, as well as on Windows systems.

Tk is not a strict dependency for Brownie. However, if it is not installed on your system you will receive an error when attempting to load the GUI.

You can use the following command to check that Tk has been correctly installed:

```
python -m tkinter
```

This should open a simple window and display the installed version number.

For installation instructions read Installing TK within the TK Documentation.

CHAPTER 4

Creating a New Project

The first step to using Brownie is to initialize a new project. This can be done in two ways:

1. Create an empty project using `brownie init`.
2. Create a project from an existing template using `brownie bake`.

4.1 Creating an Empty Project

To initialize an empty project, start by creating a new folder. From within that folder, type:

```
$ brownie init
```

An empty *project structure* is created within the folder.

4.2 Creating a Project from a Template

You can 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.

Mixes are automatically created within a subfolder of their name. To initialize the `token` mix:

```
$ brownie bake token
```

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

4.2.1 React Template

React-Mix is a bare-bones implementation of Create React App configured to work with Brownie. You can use it as a starting point for building your own React frontend for your dApp.

To initialize from this mix:

```
$ brownie bake react
```

See the React-Mix repo for more information on how to use React with Brownie.

4.2.2 Continuous Integration Template

Github-Actions-Mix is a template preconfigured for use with Github Actions continuous integration, as well as other useful tools.

To initialize from this mix:

```
$ brownie bake github-actions
```

See the Github-Actions-Mix repo for a detailed explanation of how to configure and use the tools within this template.

CHAPTER 5

Structure of a Project

Every Brownie project includes the following folders:

- `contracts/`: Contract sources
- `interfaces/`: Interface sources
- `scripts/`: Scripts for deployment and interaction
- `tests/`: Scripts for testing the project

The following folders are also created, and used internally by Brownie for managing the project. You should not edit or delete files within these folders.

- `build/`: Project data such as compiler artifacts and unit test results
- `reports/`: JSON report files for use in the GUI

See *The Build Folder* for more information about Brownie internal project folders.

If you require a different organization for your project, you can adjust the subdirectory names within the project *configuration file*.

5.1 contracts/

The `contracts` folder holds all contract source files for the project. Each time Brownie is run, it checks for new or modified files within this folder. If any are found, they are compiled and included within the project.

Contracts may be written in Solidity (with a `.sol` extension) or Vyper (with a `.vy` extension).

5.2 interfaces/

The `interfaces` folder holds interface source files that may be referenced by contract sources, but which are not considered to be primary components of the project. Adding or modifying an interface source onlys triggers a recompile if the interface is required by a contract.

Interfaces may be written in Solidity (`.sol`) or Vyper (`.vy`), or supplied as a JSON encoded ABI (`.json`).

5.3 `scripts/`

The `scripts` folder holds Python scripts used for deploying contracts, or to automate common tasks and interactions. These scripts are executed via the `brownie run` command.

See the *Brownie Scripts* documentation for more information on Brownie scripts.

5.4 `tests/`

The `tests` folder holds Python scripts used for testing a project. Brownie uses the pytest framework for unit testing.

See *Brownie Pytest* documentation for more information on testing a project.

CHAPTER 6

Compiling Contracts

To compile all of the contract sources within the `contracts/` subfolder of a project:

```
$ brownie compile
```

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

If one or more contracts are unable to compile, Brownie raises an exception with information about why the compilation failed. You cannot use Brownie with a project as long as compilation is failing. You can temporarily exclude a file or folder from compilation by adding an underscore (_) to the start of the name.

6.1 Supported Languages

Brownie supports Solidity (`>=0.4.22`) and Vyper (`>=0.1.0-beta.16`). The file extension determines which compiler is used:

- Solidity: `.sol`
- Vyper: `.vy`

6.2 Interfaces

Project contracts can import interfaces from the `interfaces/` subfolder. Interfaces are not considered primary components of a project. Adding or modifying an interface only triggers a recompile if a contract is dependent upon that interface.

The `interfaces/` folder is of particular use in the following situations:

1. When using Vyper, where interfaces are not necessarily compilable source code and so cannot be included in the `contracts/` folder.

2. When using Solidity and Vyper in the same project, or multiple versions of Solidity, where compatibility issues prevent contracts from directly referencing one another.

Interfaces may be written in Solidity (.sol) or Vyper (.vy). Vyper contracts are also able to directly import JSON encoded ABI (.json) files.

6.3 Compiler Settings

Compiler settings may be declared in the *configuration file* of a project. When no configuration file is present or settings are omitted, Brownie uses the following default values:

```yaml
compiler:
    evm_version: null
    solc:
        version: null
        optimizer:
            enabled: true
            runs: 200
    vyper:
        version: null
```

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

6.3.1 Setting the Compiler Version

Note: Brownie supports Solidity versions >=0.4.22 and Vyper versions >=0.1.0-beta.16.

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

When the compiler version is not explicitly declared, 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, and set all interface pragmas as >=0.4.22.

6.3.2 The EVM Version

By default evm_version is set to null. Brownie sets the ruleset based on the compiler:

- **byzantium**: Solidity <=0.5.4
- **petersburg**: Solidity >=0.5.5 <=0.5.12
- **istanbul**: Solidity >=0.5.13, Vyper

You can also set the EVM version manually. Valid options are byzantium, constantinople, petersburg and istanbul. You can also use the Ethereum Classic rulesets atlantis and agharta, which are converted to their Ethereum equivalents prior to being passed to the compiler.

See the Solidity EVM documentation or Vyper EVM documentation for more info on the different EVM versions and how they affect compilation.

6.3.3 Compiler Optimization

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

Values given under `compiler.solc.optimizer` in the project *configuration file* are passed directly to the compiler. This way you can modify specific optimizer settings. For example, to enable common subexpression elimination and the YUL optimizer:

```
compiler:
    solc:
        optimizer:
            details:
                cse: true
                yul: true
```

See the Solidity documentation for information on the optimizer and it's available settings.

6.3.4 Path Remappings

The Solidity compiler allows path remappings. Brownie exposes this functionality via the `compiler.solc.remappings` field in the configuration file:

```
compiler:
    solc:
        remappings:
          - zeppelin=/usr/local/lib/open-zeppelin/contracts/
          - github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/
```

Each value under `remappings` is a string in the format `prefix=path`. A remapping instructs the compiler to search for a given prefix at a specific path. For example:

```
github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/
```

This remapping instructs the compiler to search for anything starting with `github.com/ethereum/dapp-bin/` under `/usr/local/lib/dapp-bin`.

Brownie automatically ensures that all remapped paths are allowed. You do not have to declare `allow_paths`.

> **Warning:** Brownie does not detect modifications to files that are imported from outside the root folder of your project. You must manually recompile your project when an external source file changes.

Remapping Installed Packages

Remappings can also be applied to installed packages. For example:

```
compiler:
    solc:
        remappings:
          - "@openzeppelin=OpenZeppelin/openzeppelin-contracts@3.0.0"
```

With the `OpenZeppelin/openzeppelin-contracts@3.0.0` package installed, and the above remapping added to the configuration file, both of the following import statements point to the same location:

```
import "OpenZeppelin/openzeppelin-contracts@3.0.0/contracts/math/SafeMath.sol";
```

```
import "@openzeppelin/contracts/math/SafeMath.sol";
```

6.4 Installing the Compiler

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

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

```
>>> from brownie.project.compiler import install_vyper
>>> install_vyper("0.2.4")
```

CHAPTER 7

Interacting with your Contracts

Brownie has three main components that you can use while developing your project:

1. The *console* is useful for quick testing and debugging.
2. *Scripts* allow you to automate common tasks and handle deployments.
3. *Tests* help to ensure that your contracts are executing as intended.

7.1 Using 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 directory, load it by typing:

```
$ brownie console
```

Brownie will compile the contracts, launch or attach to the local test environment, 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 any class or method to view information on it's functionality.

7.2 Writing Scripts

Along with the console, you can write scripts for quick testing or to automate common processes. Scripting is also useful when deploying your contracts to a non-local network.

Scripts are stored in the `scripts/` directory within your project.

7.2.1 Layout of a Script

Brownie scripts use standard Python syntax, but there are a few things to keep in mind in order for them to execute properly.

Import Statements

Unlike the console where all of Brownie's objects are already available, in a script you must first import them. The simplest way to do this is via a wildcard import:

```
from brownie import *
```

This imports the instantiated project classes into the local namespace and gives access to the *Brownie API* in exactly the same way as if you were using the console.

Alternatively you may wish to only import exactly the classes and methods required by the script. For example:

```
from brownie import Token, accounts
```

This makes available the *accounts* and *Token* containers, which is enough to deploy a contract.

Functions

Each script can contain as many functions as you'd like. When executing a script, brownie attempts to run the `main` function if no other function name is given.

7.2.2 Running Scripts

To execute a script from the command line:

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

From the console, you can use the `run` method:

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

You can also import and call the script directly:

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

7.2.3 Examples

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 Token, accounts

def main():
    Token.deploy("Test Token", "TST", 18, 1e23, {'from': accounts[0]})
```

And here is an expanded version of the same script, that includes a simple method for distributing tokens.

```
from brownie import Token, accounts

def main():
    token = Token.deploy("Test Token", "TST", 18, 1e23, {'from': accounts[0]})
    return token

def distribute_tokens(sender=accounts[0], receiver_list=accounts[1:]):
    token = main()
    for receiver in receiver_list:
        token.transfer(receiver, 1e18, {'from': sender})
```

7.3 Writing Tests

Brownie leverages `pytest` and `hypothesis` to provide a robust framework for testing your contracts.

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

```
$ brownie test
```

To learn more about writing tests in Brownie, you should start by reviewing the *Brownie Pytest documentation*.

CHAPTER 8

Brownie Package Manager

Brownie allows you to install other projects as packages. Some benefits of packages include:

- Easily importing and building upon code ideas written by others
- Reducing duplicated code between projects
- Writing unit tests that verify interactions between your project and another project

The Brownie package manager is available from the commandline:

```
$ brownie pm
```

8.1 Installing a Package

Brownie supports package installation from ethPM and Github.

8.1.1 Installing from Github

The easiest way to install a package is from a Github repository. Brownie considers a Github repository to be a package if meets the following criteria:

- The repository must have one or more tagged versions.
- The repository must include a `contracts/` folder containing one or more Solidity or Vyper source files.

A repository does not have to implement Brownie in order to function as a package. Many popular projects using frameworks such as Truffle or Embark can be added as Brownie packages.

To install a package from Github you must use a package ID. A package ID is comprised of the name of an organization, a repository, and a version tag. Package IDs are not not case sensitive.

```
[ORGANIZATION]/[REPOSITORY]@[VERSION]
```

Examples

To install OpenZeppelin contracts version `3.0.0`:

```
$ brownie pm install OpenZeppelin/openzeppelin-contracts@3.0.0
```

To install AragonOS version `4.0.0`:

```
$ brownie pm install aragon/aragonos@4.0.0
```

8.1.2 Installing from ethPM

The Ethereum Package Manager (ethPM) is a decentralized package manager used to distribute EVM smart contracts and projects.

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.

To obtain an ethPM package, you must know both the package name and the address of the registry where it is available. This information is communicated through a registry URI. Registry URIs use the following format:

```
ethpm://[CONTRACT_ADDRESS]:[CHAIN_ID]/[PACKAGE_NAME]@[VERSION]
```

The Snake Charmers maintain an ethPM registry explorer where you can obtain registry URIs.

Examples

To install OpenZeppelin's Math package, served from the Snake Charmers Zeppelin registry:

```
$ brownie pm install ethpm://zeppelin.snakecharmers.eth:1/math@1.0.0
```

To install v2 of the Compound Protocol, served from the Snake Charmers DeFi registry:

```
$ brownie pm install ethpm://defi.snakecharmers.eth:1/compound@1.1.0
```

8.2 Working with Packages

8.2.1 Viewing Installed Packages

Use `brownie pm list` to view currently installed packages. After installing all of the examples given above, the output looks something like this:

```
$ brownie pm list
Brownie - Python development framework for Ethereum

The following packages are currently installed:

OpenZeppelin
└─OpenZeppelin/openzeppelin-contracts@3.0.0

```

(continues on next page)

(continued from previous page)

```
aragon
└─aragon/aragonOS@4.0.0

zeppelin.snakecharmers.eth
└─zeppelin.snakecharmers.eth/access@1.0.0

defi.snakecharmers.eth
└─defi.snakecharmers.eth/compound@1.1.0
```

8.2.2 Cloning a Package

Use `brownie pm clone [path]` to copy the contents of a package into another folder. The package will be cloned to the current directory if [path] is ommited. This is useful for exploring the filestructure of a package, or when you wish to build a project on top of an existing package.

To copy the Aragon package to the current folder:

```
$ brownie pm export aragon/aragonOS@4.0.0
```

8.3 Using Packages in your Project

8.3.1 Importing Sources from a Package

You can import sources from an installed package in the same way that you would a source within your project. The root path is based on the name of the package and can be obtained via `brownie pm list`.

For example, to import `SafeMath` from OpenZeppelin contracts:

```
import "OpenZeppelin/openzeppelin-contracts@3.0.0/contracts/math/SafeMath.sol";
```

You can modify the import path with the `remappings` field in your project configuration file. See *Remapping Installed Packages* for more information.

8.3.2 Using Packages in Tests

The `pm` fixture provides access to installed packages during testing. It returns a *`Project`* object when called with a project ID:

```
def test_with_compound_token(pm):
    compound = pm('defi.snakecharmers.eth/compound@1.1.0').CToken
```

See the *unit test documentation* for more detailed information.

8.3.3 Declaring Project Dependencies

Dependencies are declared by adding a `dependencies` field to your project *configuration file*:

```
dependencies:
    - aragon/aragonOS@4.0.0
    - defi.snakecharmers.eth/compound@1.1.0
```

Brownie attempts to install any listed dependencies prior to compiling a project. This is useful when your project may be used outside of your local environment.

CHAPTER 9

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.

Note: If you receive an error when attempting to load the GUI, you probably do not have Tk installed on your system. See the *Tk installation instrucions* for more detailed information.

9.1 Getting Started

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

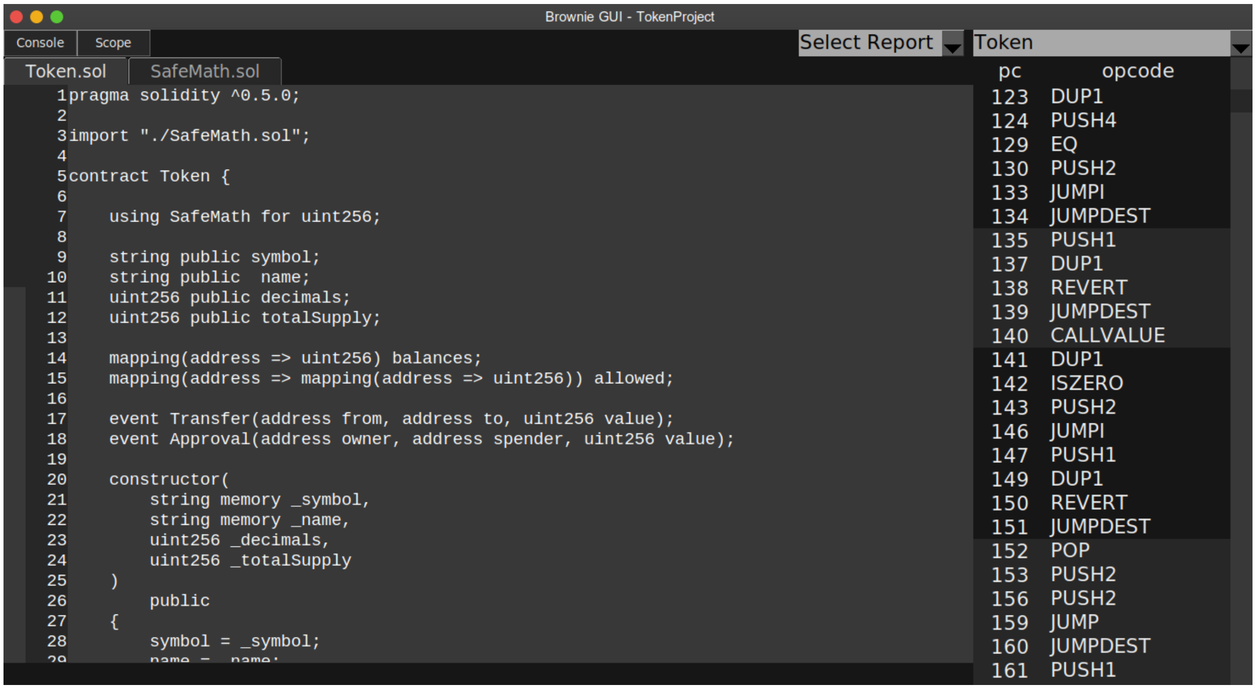
```
$ brownie gui
```

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`:



```
pragma solidity ^0.5.0;

import "./SafeMath.sol";

contract Token {

    using SafeMath for uint256;

    string public symbol;
    string public  name;
    uint256 public decimals;
    uint256 public totalSupply;

    mapping(address => uint256) balances;
    mapping(address => mapping(address => uint256)) allowed;

    event Transfer(address from, address to, uint256 value);
    event Approval(address owner, address spender, uint256 value);

    constructor(
        string memory _symbol,
        string memory _name,
        uint256 _decimals,
        uint256 _totalSupply
    )
        public
    {
        symbol = _symbol;
        name = _name;
```

pc	opcode
123	DUP1
124	PUSH4
129	EQ
130	PUSH2
133	JUMPI
134	JUMPDEST
135	PUSH1
137	DUP1
138	REVERT
139	JUMPDEST
140	CALLVALUE
141	DUP1
142	ISZERO
143	PUSH2
146	JUMPI
147	PUSH1
149	DUP1
150	REVERT
151	JUMPDEST
152	POP
153	PUSH2
156	PUSH2
159	JUMP
160	JUMPDEST
161	PUSH1

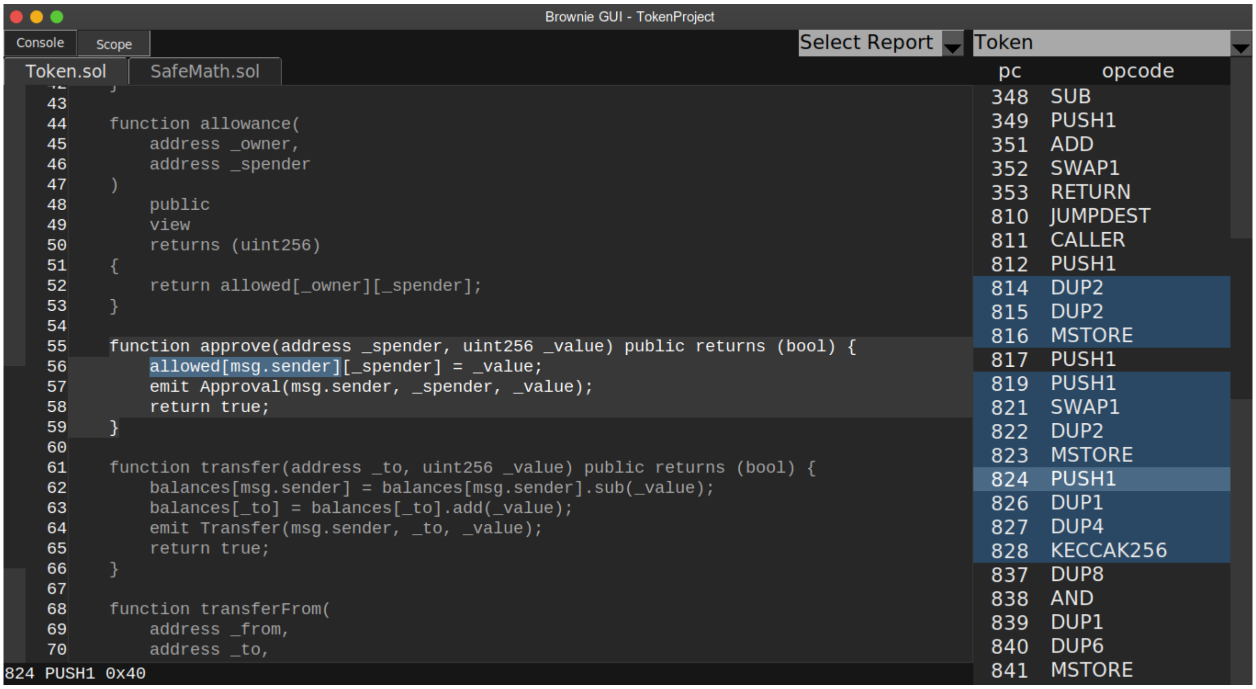
9.2 Working with Opcodes

9.2.1 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.



```

    function allowance(
        address _owner,
        address _spender
    )
        public
        view
        returns (uint256)
    {
        return allowed[_owner][_spender];
    }

    function approve(address _spender, uint256 _value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;
        emit Approval(msg.sender, _spender, _value);
        return true;
    }

    function transfer(address _to, uint256 _value) public returns (bool) {
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(_value);
        emit Transfer(msg.sender, _to, _value);
        return true;
    }

    function transferFrom(
        address _from,
        address _to,
```

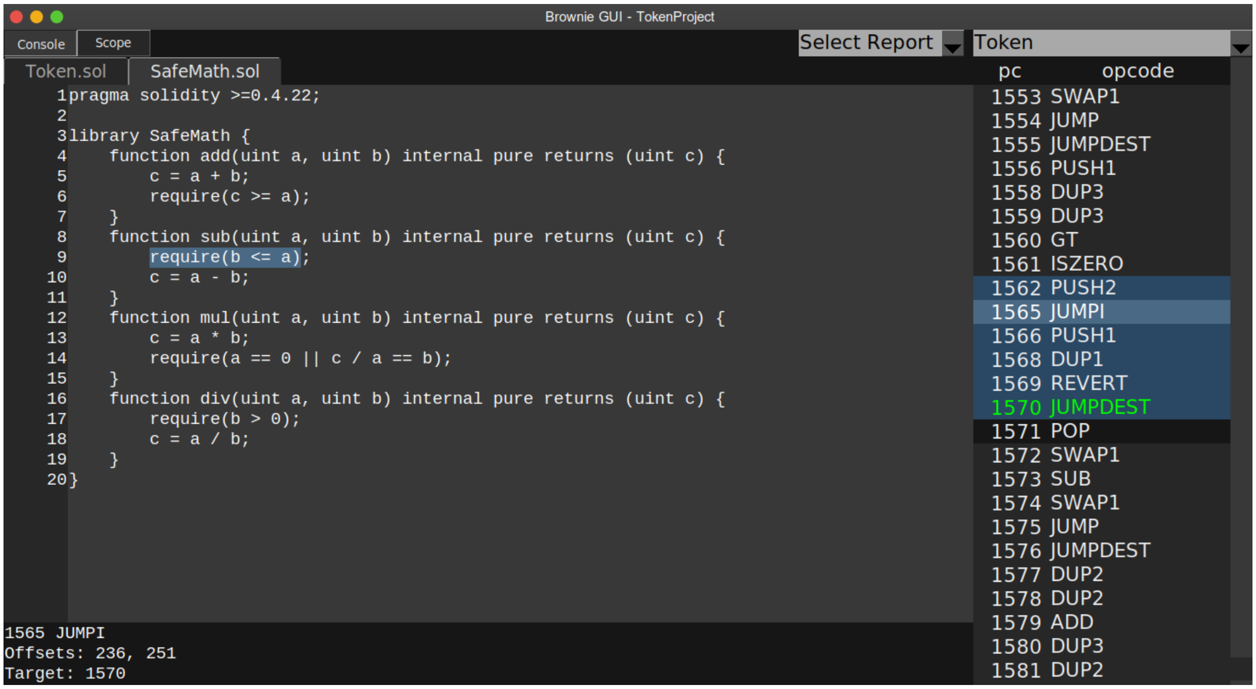
```
824 PUSH1 0x40
```

pc	opcode
348	SUB
349	PUSH1
351	ADD
352	SWAP1
353	RETURN
810	JUMPDEST
811	CALLER
812	PUSH1
814	DUP2
815	DUP2
816	MSTORE
817	PUSH1
819	PUSH1
821	SWAP1
822	DUP2
823	MSTORE
824	PUSH1
826	DUP1
827	DUP4
828	KECCAK256
837	DUP8
838	AND
839	DUP1
840	DUP6
841	MSTORE

9.2.2 Jump Instructions

Click the `Console` button in the top left (or press 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.



```
pragma solidity >=0.4.22;

library SafeMath {
    function add(uint a, uint b) internal pure returns (uint c) {
        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;
        require(a == 0 || c / a == b);
    }
    function div(uint a, uint b) internal pure returns (uint c) {
        require(b > 0);
        c = a / b;
    }
}
```

```
1553 SWAP1
1554 JUMP
1555 JUMPDEST
1556 PUSH1
1558 DUP3
1559 DUP3
1560 GT
1561 ISZERO
1562 PUSH2
1565 JUMPI
1566 PUSH1
1568 DUP1
1569 REVERT
1570 JUMPDEST
1571 POP
1572 SWAP1
1573 SUB
1574 SWAP1
1575 JUMP
1576 JUMPDEST
1577 DUP2
1578 DUP2
1579 ADD
1580 DUP3
1581 DUP2
```

```
1565 JUMPI
Offsets: 236, 251
Target: 1570
```

9.2.3 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.

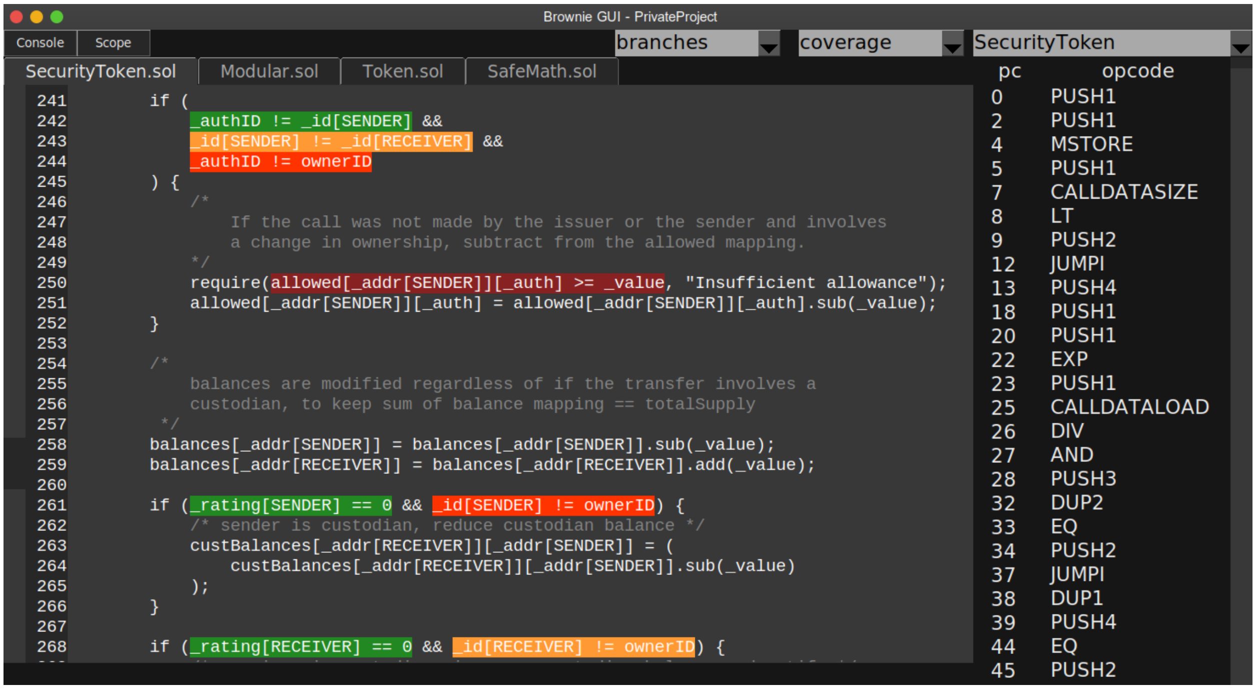
9.3 Viewing Reports

Actions such as coverage evaluation and security analysis produce report files within the reports/ directory of your project. To examine a report:

1. click on the drop-down list in the upper right that says “Select Report”
2. Select the report file you wish to view.
3. A new drop-down list will appear where you can select which report to display.

Some reports will include additional information that is displayed in the GUI console when you hover the mouse over a related section.

Here is an example of a coverage analysis report:



```
        if (
            _authID != _id[SENDER] &&
            _id[SENDER] != _id[RECEIVER] &&
            _authID != ownerID
        ) {
            /*
                If the call was not made by the issuer or the sender and involves
                a change in ownership, subtract from the allowed mapping.
            */
            require(allowed[_addr[SENDER]][_auth] >= _value, "Insufficient allowance");
            allowed[_addr[SENDER]][_auth] = allowed[_addr[SENDER]][_auth].sub(_value);
        }

        /*
            balances are modified regardless of if the transfer involves a
            custodian, to keep sum of balance mapping == totalSupply
         */
        balances[_addr[SENDER]] = balances[_addr[SENDER]].sub(_value);
        balances[_addr[RECEIVER]] = balances[_addr[RECEIVER]].add(_value);

        if (_rating[SENDER] == 0 && _id[SENDER] != ownerID) {
            /* sender is custodian, reduce custodian balance */
            custBalances[_addr[RECEIVER]][_addr[SENDER]] = (
                custBalances[_addr[RECEIVER]][_addr[SENDER]].sub(_value)
            );
        }

        if (_rating[RECEIVER] == 0 && _id[RECEIVER] != ownerID) {
```

```
pc   opcode
0    PUSH1
2    PUSH1
4    MSTORE
5    PUSH1
7    CALLDATASIZE
8    LT
9    PUSH2
12   JUMPI
13   PUSH4
18   PUSH1
20   PUSH1
22   EXP
23   PUSH1
25   CALLDATALOAD
26   DIV
27   AND
28   PUSH3
32   DUP2
33   EQ
34   PUSH2
37   JUMPI
38   DUP1
39   PUSH4
44   EQ
45   PUSH2
```

9.4 Report JSON Format

Third party tools can generate reports for display in the Brownie GUI. Reports must be saved in the `reports/` directory of a project. Brownie expects reports to be JSON encoded and 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 10

Working with 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]
<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]
```

The *Account.balance* method is used to check the balance of an account. The value returned is denominated in *wei*.

```
>>> accounts[1].balance()
100000000000000000000
```

The *Account.transfer* method is used to send ether between accounts and perform other simple transactions. As shown in the example below, the amount to transfer may be specified as a string that is converted by *Wei*.

```
>>> accounts[0].transfer(accounts[1], "10 ether")

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

10.1 Generating, Adding, and Unlocking Accounts

Newly added accounts are automatically appended to the `Accounts` container.

The `Accounts.add` method is used to randomly generate a new account:

```
>>> accounts.add()
mnemonic: 'rice cement vehicle ladder end engine tiger gospel toy inspire steel teach'
<LocalAccount '0x7f1eCD32aF08635A3fB3128108F6Eb0956Efd532'>
```

You can optionally specify a private key to access a specific account:

```
>>> accounts.add('0xca751356c37a98109fd969d8e79b42d768587efc6ba35e878bc8c093ed95d8a9')
<LocalAccount '0xf6c0182eFD54830A87e4020E13B8E4C82e2f60f0'>
```

In a development environment, it is possible to send transactions from an address without having that addresses private key. To create an `Account` object from an arbitrary address, use the `Accounts.at` method and include `force=True` as a keyword argument:

```
>>> accounts.at('0x79B2f0CbED2a565C925A8b35f2B402710564F8a2', force=True)
<Account '0x79B2f0CbED2a565C925A8b35f2B402710564F8a2'>
```

See *Account Management* for more information on working with accounts.

10.2 Broadcasting Multiple Transactions

Broadcasting a transaction is normally a *blocking action* - Brownie waits until the transaction has confirmed before continuing. One way to broadcast transactions without blocking is to set `required_confs = 0`. This immediately returns a pending `TransactionReceipt` and continues without waiting for a confirmation. Additionally, setting `silent = True` suppresses the console output.

```
>>> transactions = [
        accounts[0].transfer(accounts[i], "1 ether", required_confs=0, silent=True)
        for i in range(1, 4)
    ]
>>> [tx.status for tx in transactions]
[1, -1, -1]
```

These transactions are initially pending (`status == -1`) and appear yellow in the console.

10.3 Replacing Transactions

The `TransactionReceipt.replace` method can be used to replace underpriced transactions while they are still pending:

```
>>> tx = accounts[0].transfer(accounts[1], 100, required_confs=0, gas_price="1 gwei")
Transaction sent: 0xc1aab54599d7875fc1fe8d3e375abb0f490cbb80d5b7f48cedaa95fa726f29be
    Gas price: 13.0 gwei   Gas limit: 21000   Nonce: 3
<Transaction object
 ↪'0xc1aab54599d7875fc1fe8d3e375abb0f490cbb80d5b7f48cedaa95fa726f29be'>

>>> tx.replace(1.1)
Transaction sent: 0x9a525e42b326c3cd57e889ad8c5b29c88108227a35f9763af33dccd522375212
    Gas price: 14.3 gwei   Gas limit: 21000   Nonce: 3
<Transaction '0x9a525e42b326c3cd57e889ad8c5b29c88108227a35f9763af33dccd522375212'>
```

All pending transactions are available within the `history` object. As soon as one transaction confirms, the remaining dropped transactions are removed. See the documentation on *accessing transaction history* for more info.

CHAPTER 11

Working with Contracts

11.1 Deploying Contracts

Each time Brownie is loaded it will automatically compile your project and create `ContractContainer` objects for each deployable contract. This object is a container used to access individual deployments. It is also used to deploy new contracts.

```
>>> Token
[]
>>> type(Token)
<class 'brownie.network.contract.ContractContainer'>
>>> Token.deploy
<ContractConstructor object 'Token.constructor(string _symbol, string _name, uint256 _
↪decimals, uint256 _totalSupply)'>
```

`ContractContainer.deploy` is used to deploy a new contract.

```
>>> Token.deploy
<ContractConstructor object 'Token.constructor(string _symbol, string _name, uint256 _
↪decimals, uint256 _totalSupply)'>
```

It must be called with the contract constructor arguments, and a dictionary of *transaction parameters* containing a `from` field that specifies which `Account` to deploy the contract from.

```
>>> Token.deploy("Test Token", "TST", 18, 1e23, {'from': accounts[1]})

Transaction sent: 0x2e3cab83342edda14141714ced002e1326ecd8cded4cd0cf14b2f037b690b976
Transaction confirmed - block: 1   gas spent: 594186
Contract deployed at: 0x5419710735c2D6c3e4db8F30EF2d361F70a4b380
<Token Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>
```

Calling `ContractContainer.deploy` returns a `ProjectContract` object. The returned object is also appended to the `ContractContainer`.

```
>>> t = Token.deploy("Test Token", "TST", 18, 1e23, {'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'>]
```

11.1.1 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.

```
>>> MetaCoin.deploy({'from': accounts[0]})
  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

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

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

11.2 Interacting with your Contracts

Once a contract has been deployed, you can interact with it via via *calls* and *transactions*.

- **Transactions** are broadcast to the network and recorded on the blockchain. They cost ether to run, and are able to alter the state to the blockchain.
- **Calls** are used to execute code on the network without broadcasting a transaction. They are free to run, and cannot alter the state of the blockchain in any way. Calls are typically used to retrieve a storage value from a contract using a getter method.

You may call or send a transaction to any public function within a contract. However, depending on the code, there is always a preferred method:

- In Solidity, callable methods are labelled as view or pure
- In Vyper, callable methods include the @constant decorator.

All public contract methods are available from the *ProjectContract* object via class methods of the same name.

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

When a contract source includes NatSpec documentation, you can view it via the *ContractCall.info* method:

```
>>> Token[0].transfer.info()
transfer(address _to, uint256 _value)
  @dev transfer token for a specified address
  @param _to The address to transfer to.
  @param _value The amount to be transferred.
```

11.2.1 Transactions

State-changing contract methods are called via a *ContractTx* object. This object performs a transaction and returns a *TransactionReceipt*.

You may optionally include a dictionary of *transaction parameters* as the final argument. If you do not do this, or do not specify a `from` value within the parameters, the transaction is sent from the same address that deployed the contract.

```
>>> Token[0].transfer(accounts[1], 1e18, {'from': accounts[0]})

Transaction sent: 0x6e557594e657faf1270235bf4b3f27be7f5a3cb8a9c981cfffb12133cbaa165e
Token.transfer confirmed - block: 4   gas used: 51019 (33.78%)
<Transaction object
↪'0x6e557594e657faf1270235bf4b3f27be7f5a3cb8a9c981cfffb12133cbaa165e'>
```

If you wish to call the contract method without a transaction, use the *ContractTx.call* method.

```
>>> Token[0].transfer.call(accounts[1], 1e18, {'from': accounts[0]})
True
```

Transaction Parameters

When executing a transaction to a contract, you can optionally include a `dict` of transaction parameters as the final input. It may contain the following values:

- `from`: the *Account* that the transaction it sent from. If not given, the transaction is sent from the account that deployed the contract.
- `gas_limit`: The amount of gas provided for transaction execution, in wei. If not given, the gas limit is determined using `web3.eth.estimateGas`.
- `gas_buffer`: A multiplier applied to `web3.eth.estimateGas` when setting gas limit automatically. `gas_limit` and `gas_buffer` cannot be given at the same time.
- `gas_price`: The gas price for the transaction, in wei. If not given, the gas price is set according to `web3.eth.gasPrice`.
- `amount`: The amount of Ether to include with the transaction, in wei.
- `nonce`: The nonce for the transaction. If not given, the nonce is set according to `web3.eth.getTransactionCount` while taking pending transactions from the sender into account.

- `required_confs`: The required *confirmations* before the *TransactionReceipt* is processed. If none is given, defaults to 1 confirmation. If 0 is given, immediately returns a pending *TransactionReceipt*, while waiting for a confirmation in a separate thread.
- `allow_revert`: Boolean indicating whether the transaction should be broadacsted when it is expected to revert. If not set, the default behaviour is to allow reverting transactions in development and disallow them in a live environment.

All currency integer values can also be given as strings that will be converted by *Wei*.

Hint: When working in development environment, the `from` field can be any address given as a string. In this way you can broadcast a transaction from an address without having it's private key. It is even possible to send transactions from contracts!

11.2.2 Calls

Contract methods that do not alter the state are called via a *ContractCall* object. This object will call the contract method without broadcasting a transaction, and return the result.

```
>>> Token[0].balanceOf(accounts[0])
1000000000000000000000
```

If you wish to access the method via a transaction you can use *ContractCall.transact*.

```
>>> 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
1000000000000000000000
```

11.3 Contracts Outside of your Project

When working in a *live environment* or *forked development network*, you can create *Contract* objects to interact with already-deployed contracts.

Contract objects may be created from interfaces within the `interfaces/` folder of your project, or by fetching information from a remote source such as a block explorer or ethPM registry.

11.3.1 Using Local Interfaces

The *InterfaceContainer* object (available as `interface`) provides access to the interfaces within your project's `interfaces/` folder.

For example, to create a *Contract* object from an interface named `Dai`:

```
>>> interface.Dai
<InterfaceConstructor 'Dai'>
```

(continues on next page)

(continued from previous page)

```
>>> interface.Dai("0x6B175474E89094C44Da98b954EedeAC495271d0F")
<Dai Contract object '0x6B175474E89094C44Da98b954EedeAC495271d0F'>
```

You can also use the `Contract.from_abi` classmethod to instatiate from an ABI as a dictionary:

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

11.3.2 Fetching from a Remote Source

Contract objects may also be created by fetching data from a remote source. For example, use `Contract.from_explorer` to create an object by querying Etherscan:

```
>>> Contract.from_explorer("0x6b175474e89094c44da98b954eedeac495271d0f")
Fetching source of 0x6B175474E89094C44Da98b954EedeAC495271d0F from api.etherscan.io...
<Dai Contract '0x6B175474E89094C44Da98b954EedeAC495271d0F'>
```

11.3.3 Persisting Contracts between Sessions

The data used to create `Contract` objects is stored in a local database and persists between sessions. After the initial creation via a *class method*, you can recreate an object by initializing `Contract` with an address:

```
>>> Contract("0x6b175474e89094c44da98b954eedeac495271d0f")
<Dai Contract '0x6B175474E89094C44Da98b954EedeAC495271d0F'>
```

Alternatively, `Contract.set_alias` allows you to create an alias for quicker access. Aliases also persist between sessions.

```
>>> contract = Contract("0x6b175474e89094c44da98b954eedeac495271d0f")
>>> contract.set_alias('dai')

>>> Contract('dai')
<Dai Contract '0x6B175474E89094C44Da98b954EedeAC495271d0F'>
```

CHAPTER 12

Interacting with the Blockchain

12.1 Accessing Block Information

The *Chain* object, available as `chain`, uses list-like syntax to provide access to block information:

```
>>> chain
<Chain object (chainid=1, height=10451202)>

>>> chain[2000000]
AttributeDict({
    'difficulty': 49824742724615,
    'extraData': '0xe4b883e5bda9e7a59ee4bb99e9b1bc',
    'gasLimit': 4712388,
    'gasUsed': 21000,
    'hash': '0xc0f4906fea23cf6f3cce98cb44e8e1449e455b28d684dfa9ff65426495584de6',
    'logsBloom':
↪'0x000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪',
    'miner': '0x61c808d82a3ac53231750dadc13c777b59310bd9',
    'nonce': '0x3b05c6d5524209f1',
    'number': 2000000,
    'parentHash': '0x57ebf07eb9ed1137d41447020a25e51d30a0c272b5896571499c82c33ecb7288
↪',
    'receiptRoot': '0x84aea4a7aad5c5899bd5cfc7f309cc379009d30179316a2a7baa4a2ea4a438ac
↪',
    'sha3Uncles': '0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347
↪',
    'size': 650,
    'stateRoot': '0x96dbad955b166f5119793815c36f11ffa909859bbfeb64b735cca37cbf10bef1',
    'timestamp': 1470173578,
    'totalDifficulty': 44010101827705409388,
    'transactions': [
↪'0xc55e2b90168af6972193c1f86fa4d7d7b31a29c156665d15b9cd48618b5177ef'],
    'transactionsRoot':
↪'0xb31f174d27b99cdae8e746bd138a01ce60d8dd7b224f7c60845914def05ecc58',
```

(continues on next page)

(continued from previous page)

```
    'uncles': [],
})

>>> web3.eth.blockNumber
10451202

>>> len(chain)
10451203  # always +1 to the current block number, because the first block is zero

>>> chain[0] == web3.eth.getBlock(0)
True

# for negative index values, the block returned is relative to the most recently␣
↪mined block
>>> chain[-1] == web3.eth.getBlock('latest')
True
```

12.2 Accessing Transaction Data

12.2.1 Local Transaction History

The *TxHistory* container, available as `history`, holds all the transactions that have been broadcasted during the Brownie session. You can use it to access *TransactionReceipt* objects if you did not assign them to a variable when making the call.

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

You can use *history.filter* to filter for specific transactions, either with key-value pairs or a lambda function:

```
>>> history.filter(sender=accounts[0], value="1 ether")
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>]

>>> history.filter(key=lambda k: k.nonce < 2)
[<Transaction '0x03569ee152b04ba5b55c2bf05f99f7ec153db715acfe0c1600f144ded58f31fe'>,
↪<Transaction '0x42193c0ff7007c6e2a5e5572a3c6b5706cd133d21e30e5826add3d971134504c'>]
```

12.2.2 Other Transactions

Use *chain.get_transaction* to get a *TransactionReceipt* object for any transaction:

```
>>> chain.get_transaction(
↪'0xf598d43ef34a48478f3bb0ad969c6735f416902c4eb1eb18ebebe0fca786105e')
<Transaction '0xf598d43ef34a48478f3bb0ad969c6735f416902c4eb1eb18ebebe0fca786105e'>
```

This also works for pending transactions. When the transaction has not yet confirmed, the transaction hash is displayed in yellow within the console.

12.3 Manipulating the Development Chain

Brownie is designed to use ganache-cli as a local development environment. Functionality such as mining, snapshotting and time travel is accessible via the *Chain* object.

12.3.1 Mining New Blocks

Ganache's default behavior is to mine a new block each time you broadcast a transaction. You can mine empty blocks with the *chain.mine* method:

```
>>> web3.eth.blockNumber
0
>>> chain.mine(50)
50
>>> web3.eth.blockNumber
50
```

12.3.2 Time Travel

You can call *chain.time* to view the current epoch time:

```
>>> chain.time()
1500000000
```

To fast forward the clock, call *chain.sleep*.

```
>>> chain.sleep(31337)

>>> chain.time()
1500031337
```

Note that sleeping does not mine a new block. Contract view functions that rely on `block.timestamp` will be unaffected until you perform a transaction or call *chain.mine*.

12.3.3 Snapshots

Use *chain.snapshot* to take a snapshot of the current state of the blockchain:

```
>>> chain.snapshot()

>>> 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 then return to this state later using `chain.revert`:

```
>>> accounts[0].balance()
89999580000000000000
>>> chain.revert()
4
>>> accounts[0].balance()
100000000000000000000
```

Reverting does not consume the 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 `chain.reset`.

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

12.3.4 Undo / Redo

Along with snapshotting, you can use `chain.undo` and `chain.redo` to move backward and forward through recent transactions. This is especially useful during *interactive test debugging*.

```
>>> accounts[0].transfer(accounts[1], "1 ether")
Transaction sent: 0x8c166b66b356ad7f5c58337973b89950f03105cdae896ac66f16cdd4fc395d05
  Gas price: 0.0 gwei   Gas limit: 6721975
  Transaction confirmed - Block: 1   Gas used: 21000 (0.31%)

<Transaction '0x8c166b66b356ad7f5c58337973b89950f03105cdae896ac66f16cdd4fc395d05'>

>>> chain.undo()
0

>>> chain.redo()
Transaction sent: 0x8c166b66b356ad7f5c58337973b89950f03105cdae896ac66f16cdd4fc395d05
  Gas price: 0.0 gwei   Gas limit: 6721975
  Transaction confirmed - Block: 1   Gas used: 21000 (0.31%)
```

Note that `chain.snapshot` and `chain.revert` clear the undo buffer.

CHAPTER 13

Inspecting and Debugging Transactions

The *TransactionReceipt* object provides information about a transaction, as well as various methods to aid in debugging.

```
>>> tx = Token[0].transfer(accounts[1], 1e18, {'from': accounts[0]})

Transaction sent: 0xa7616a96ef571f1791586f570017b37f4db9decb1a5f7888299a035653e8b44b
Token.transfer confirmed - block: 2   gas used: 51019 (33.78%)

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

To view human-readable information on a transaction, call the *TransactionReceipt.info* method.

```
>>> 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
```

13.1 Event Data

Data about events is available as `TransactionReceipt.events`. It is stored in an `EventDict` object; a hybrid container with both dict-like and list-like properties.

Hint: You can also view events that were emitted in a reverted transaction. When debugging it can be useful to create temporary events to examine local variables during the execution of a failed transaction.

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

Use it as a dictionary 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:

```
# name of the address
>>> tx.events[1].name
```

(continues on next page)

(continued from previous page)

```
'CountryModified'

# address where the event fired
>>> tx.events[1].address
"0xDd18d6475A7C71Ee33CEBE730a905DbBd89945a1"

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

13.2 Internal Transactions and Deployments

TransactionReceipt.internal_transfers provides a list of internal ether transfers that occurred during the transaction.

```
>>> tx.internal_transfers
[
    {
        "from": "0x79447c97b6543F6eFBC91613C655977806CB18b0",
        "to": "0x21b42413bA931038f35e7A5224FaDb065d297Ba3",
        "value": 100
    }
]
```

TransactionReceipt.new_contracts provides a list of addresses for any new contracts that were created during a transaction. This is useful when you are using a factory pattern.

```
>>> deployer
<Deployer Contract object '0x5419710735c2D6c3e4db8F30EF2d361F70a4b380'>

>>> tx = deployer.deployNewContract()
Transaction sent: 0x6c3183e41670101c4ab5d732bfe385844815f67ae26d251c3bd175a28604da92
  Gas price: 0.0 gwei   Gas limit: 79781
  Deployer.deployNewContract confirmed - Block: 4   Gas used: 79489 (99.63%)

>>> tx.new_contracts
["0x1262567B3e2e03f918875370636dE250f01C528c"]
```

To generate *Contract* objects from this list, use *ContractContainer.at*:

```
>>> tx.new_contracts
["0x1262567B3e2e03f918875370636dE250f01C528c"]
>>> Token.at(tx.new_contracts[0])
<Token Contract object '0x1262567B3e2e03f918875370636dE250f01C528c'>
```

13.3 Debugging Failed Transactions

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`.

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], 1e18, {'from': accounts[3]})

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

The error string is also available as *`TransactionReceipt.revert_msg`*.

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

You can also 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");
```

13.4 Inspecting the Trace

13.4.1 The Trace Object

The best way to understand exactly happened in a transaction is to generate and examine a 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
```

(continues on next page)

(continued from previous page)

```
    '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
}
```

13.4.2 Call Traces

When dealing with complex transactions the trace can be may thousands of steps long - it can be challenging to know where to begin examining it. Brownie provides the *TransactionReceipt.call_trace* method to view a complete map of every jump that occured in the transaction:

```
>>> tx.call_trace()
Call trace for '0x7824c6032966ca2349d6a14ec3174d48d546d0fb3020a71b08e50c7b31c1bcb1':
Initial call cost  [21228 gas]
LiquidityGauge.deposit  0:3103  [64010 / 128030 gas]
├── LiquidityGauge._checkpoint  83:1826  [-6420 / 7698 gas]
│   ├── GaugeController.get_period_timestamp  [STATICCALL]  119:384  [2511 gas]
│   ├── ERC20CRV.start_epoch_time_write  [CALL]  411:499  [1832 gas]
│   ├── GaugeController.gauge_relative_weight_write  [CALL]  529:1017  [3178 / 7190␣
↪gas]
│   │   └── GaugeController.change_epoch  697:953  [2180 / 4012 gas]
│   │       └── ERC20CRV.start_epoch_time_write  [CALL]  718:806  [1832 gas]
│   └── GaugeController.period  [STATICCALL]  1043:1336  [2585 gas]
├── LiquidityGauge._update_liquidity_limit  1929:2950  [45242 / 54376 gas]
│   ├── VotingEscrow.balanceOf  [STATICCALL]  1957:2154  [2268 gas]
│   └── VotingEscrow.totalSupply  [STATICCALL]  2180:2768  [6029 / 6866 gas]
│       └── VotingEscrow.supply_at  2493:2748  [837 gas]
└── ERC20LP.transferFrom  [CALL]  2985:3098  [1946 gas]
```

Each line shows the following information:

```
ContractName.functionName (external call opcode) start:stop [internal / total gas␣
↪used]
```

Where `start` and `stop` are the indexes of *TransactionReceipt.trace* where the function was entered and exited. *TransactionReceipt.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 and determine where things went wrong in a complex transaction.

Functions that terminated with `REVERT` or `INVALID` opcodes are highlighted in red.

For functions with no subcalls, the used gas is shown. Otherwise, the first gas number is the amount of gas used internally by this function and the second number is the total gas used by the function including all sub-calls. Gas refunds from deleting storage or contracts are shown as negative gas used. Note that overwriting an existing zero-value with another zero-value will incorrectly display a gas refund.

Calling *TransactionReceipt.call_trace* with `True` as an argument provides an expanded view:

```
>>> history[-1].call_trace(True)

Call trace for '0x7824c6032966ca2349d6a14ec3174d48d546d0fb3020a71b08e50c7b31c1bcb1':
Initial call cost  [21228 gas]
LiquidityGauge.deposit  0:3103  [64010 / 128030 gas]
├── LiquidityGauge._checkpoint  83:1826  [-6420 / 7698 gas]
│   │
│   ├── GaugeController.get_period_timestamp  [STATICCALL]  119:384  [2511 gas]
│   │       ├── address: 0x0C41Fc429cC21BC3c826efB3963929AEdf1DBb8e
│   │       ├── input arguments:
│   │       │   └── p: 0
│   │       └── return value: 1594574319
...
```

The expanded trace includes information about external subcalls, including:

- the target address
- the amount of ether transferred
- input arguments
- return values

For calls that revert, the revert reason is given in place of the return value:

```
>>> history[-1].call_trace(True)
...
└── ERC20LP.transferFrom  [CALL]  2985:3098  [1946 gas]
        ├── address: 0xd495633B90a237de510B4375c442C0469D3C161C
        ├── value: 0
        ├── input arguments:
        │   ├── _from: 0x9EC9431CCCCD2C73F0A2F68DC69A4A527AB5D809
        │   ├── _to: 0x5AE569698C5F986665018B6E1D92A71BE71DEF9A
        │   └── _value: 100000
        └── revert reason: Integer underflow
```

You can also access this information programmatically via the *TransactionReceipt.subcalls* attribute:

```
>>> history[-1].subcalls
[
    {
        'from': "0x5AE569698C5F986665018B6e1d92A71be71DEF9a",
        'function': "get_period_timestamp(int128)",
        'inputs': {
            'p': 0
        },
        'op': "STATICCALL",
        'return_value': (1594574319,),
        'to': "0x0C41Fc429cC21BC3c826efB3963929AEdf1DBb8e"
    },
...
```

CHAPTER 14

Data Types

Brownie uses custom data types to simplify working with common represented values.

14.1 Wei

The `Wei` class is used when a value is meant to represent an amount of Ether. It is a subclass of `int` capable of converting 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'>
```

14.2 Fixed

The *Fixed* class is used to handle Vyper decimal values. It is a subclass of `decimal.Decimal` that allows comparisons, addition and subtraction against strings, integers and *Wei*.

```
>>> Fixed(1)
Fixed('1')
>>> Fixed("3.1337")
Fixed('3.1337')
>>> Fixed("12.49 gwei")
Fixed('12490000000')
>>> Fixed("-1.23") == "-1.2300"
True
```

Attempting to assign, compare or perform arithmetic against a float raises a `TypeError`.

```
>>> Fixed(3.1337)
Traceback (most recent call last):
    File "<console>", line 1, in <module>
TypeError: Cannot convert float to decimal - use a string instead

>>> Fixed("-1.23") == -1.2300
Traceback (most recent call last):
    File "<console>", line 1, in <module>
TypeError: Cannot compare to floating point - use a string instead
```

CHAPTER 15

Gas Strategies

Gas strategies are objects that dynamically generate a gas price for a transaction. They can also be used to automatically replace pending transactions within the mempool.

Gas strategies come in three basic types:

- **Simple** strategies provide a gas price once, but do not replace pending transactions.
- **Block** strategies provide an initial price, and optionally replace pending transactions based on the number of blocks that have been mined since the first transaction was broadcast.
- **Time** strategies provide an initial price, and optionally replace pending transactions based on the amount of time that has passed since the first transaction was broadcast.

15.1 Using a Gas Strategy

To use a gas strategy, first import it from `brownie.network.gas.strategies`:

```
>>> from brownie.network.gas.strategies import GasNowStrategy
>>> gas_strategy = GasNowStrategy("fast")
```

You can then provide the object in the `gas_price` field when making a transaction:

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

When the strategy replaces a pending transaction, the returned *`TransactionReceipt`* object will be for the transaction that confirms.

During *non-blocking transactions*, all pending transactions are available within the *`history`* object. As soon as one transaction confirms, the remaining dropped transactions are removed.

15.2 Setting a Default Gas Strategy

You can use `network.gas_price` to set a gas strategy as the default for all transactions:

```
>>> from brownie.network import gas_price
>>> gas_price(gas_strategy)
```

15.3 Available Gas Strategies

class `brownie.network.gas.strategies.`**`LinearScalingStrategy`**(*initial_gas_price*, *max_gas_price*, *increment=1.125*, *time_duration=30*)

Time based scaling strategy for linear gas price increase.

- `initial_gas_price`: The initial gas price to use in the first transaction
- `max_gas_price`: The maximum gas price to use
- `increment`: Multiplier applied to the previous gas price in order to determine the new gas price
- `time_duration`: Number of seconds between transactions

```
>>> from brownie.network.gas.strategies import LinearScalingStrategy
>>> gas_strategy = LinearScalingStrategy("10 gwei", "50 gwei", 1.1)

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

class `brownie.network.gas.strategies.`**`ExponentialScalingStrategy`**(*initial_gas_price*, *max_gas_price*, *time_duration=30*)

Time based scaling strategy for exponential gas price increase.

The gas price for each subsequent transaction is calculated as the previous price multiplied by *1.1 ** n* where n is the number of transactions that have been broadcast. In this way the price increase starts gradually and ramps up until confirmation.

- `initial_gas_price`: The initial gas price to use in the first transaction
- `max_gas_price`: The maximum gas price to use
- `time_duration`: Number of seconds between transactions

```
>>> from brownie.network.gas.strategies import ExponentialScalingStrategy
>>> gas_strategy = ExponentialScalingStrategy("10 gwei", "50 gwei")

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

class `brownie.network.gas.strategies.`**`GasNowStrategy`**(*speed="fast"*)

Simple gas strategy for determing a price using the GasNow API.

- `speed`: The gas price to use based on the API call. Options are rapid, fast, standard and slow.

```
>>> from brownie.network.gas.strategies import GasNowStrategy
>>> gas_strategy = GasNowStrategy("fast")

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

class brownie.network.gas.strategies.**GasNowScalingStrategy**(*initial_speed="standard", max_speed="rapid", increment=1.125, block_duration=2*)

Block based scaling gas strategy using the GasNow API.

- `initial_speed`: The initial gas price to use when broadcasting the first transaction. Options are rapid, fast, standard and slow.
- `max_speed`: The maximum gas price to use when replacing the transaction. Options are rapid, fast, standard and slow.
- `increment`: A multiplier applied to the most recently used gas price in order to determine the new gas price. If the incremented value is less than or equal to the current `max_speed` rate, a new transaction is broadcasted. If the current rate for `initial_speed` is greater than the incremented rate, it is used instead.
- `block_duration`: The number of blocks to wait between broadcasting new transactions.

```
>>> from brownie.network.gas.strategies import GasNowScalingStrategy
>>> gas_strategy = GasNowScalingStrategy("fast", increment=1.2)

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

class brownie.network.gas.strategies.**GethMempoolStrategy**(*position=500, graphql_endpoint=None, block_duration=2*)

Block based scaling gas strategy using Geth's GraphQL interface.

In order to use this strategy you must be connecting via a Geth node with GraphQL enabled.

The yielded gas price is determined by sorting transactions in the mempool according to gas price, and returning the price of the transaction at *position*. This is the same technique used by the GasNow API.

- A position of 200 or less usually places a transaction within the mining block.
- A position of 500 usually places a transaction within the 2nd pending block.

```
>>> from brownie.network.gas.strategies import GethMempoolStrategy
>>> gas_strategy = GethMempoolStrategy(200)

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

15.4 Building your own Gas Strategy

To implement your own gas strategy you must subclass from one of the *gas strategy abstract base classes*.

CHAPTER 16

Writing Unit Tests

Brownie utilizes the pytest framework for unit testing. Pytest is a mature, feature-rich test framework. It lets you write small tests with minimal code, scales well for large projects, and is highly extendable.

To run your tests:

```
$ brownie test
```

This documentation provides a quick overview of basic pytest usage, with an emphasis on features that are relevent to Brownie. Many components of pytest are only explained partially - or not at all. If you wish to learn more about pytest you should review the official pytest documentation.

16.1 Getting Started

16.1.1 Test File Structure

Pytest performs a test discovery process to locate functions that should be included in your project's test suite.

1. Tests must be stored within the `tests/` directory of your project, or a subdirectory thereof.
2. Filenames must match `test_*.py` or `*_test.py`.

Within the test files, the following methods will be run as tests:

1. Functions outside of a class prefixed with `test`.
2. Class methods prefixed with `test`, where the class is prefixed with `Test` and does not include an `__init__` method.

16.1.2 Writing your First Test

The following example is a very simple test using Brownie and pytest, verifying that an account balance has correctly changed after performing a transaction.

```
from brownie import accounts

def test_account_balance():
    balance = accounts[0].balance()
    accounts[0].transfer(accounts[1], "10 ether", gas_price=0)

    assert balance - "10 ether" == accounts[0].balance()
```

16.2 Fixtures

A fixture is a function that is applied to one or more test functions, and is called prior to the execution of each test. Fixtures are used to setup the initial conditions required for a test.

Fixtures are declared using the `@pytest.fixture` decorator. To pass a fixture to a test, include the fixture name as an input argument for the test:

```
import pytest

from brownie import Token, accounts

@pytest.fixture
def token():
    return accounts[0].deploy(Token, "Test Token", "TST", 18, 1000)

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

In this example the `token` fixture is called prior to running `test_transfer`. The fixture returns a deployed *`Contract`* instance which is then used in the test.

Fixtures can also be included as dependencies of other fixtures:

```
import pytest

from brownie import Token, accounts

@pytest.fixture
def token():
    return accounts[0].deploy(Token, "Test Token", "TST", 18, 1000)

@pytest.fixture
def distribute_tokens(token):
    for i in range(1, 10):
        token.transfer(accounts[i], 100, {'from': accounts[0]})
```

16.2.1 Brownie Pytest Fixtures

Brownie provides fixtures that simplify interact with and testing your project. Most core Brownie functionality can be accessed via a fixture rather than an import statement. For example, here is the previous example using Brownie fixtures rather than imports:

```python
import pytest

@pytest.fixture
def token(Token, accounts):
    return accounts[0].deploy(Token, "Test Token", "TST", 18, 1000)

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

See the *Fixture and Marker Reference* for information about all available fixtures.

16.2.2 Fixture Scope

The default behaviour for a fixture is to execute each time it is required for a test. By adding the `scope` parameter to the decorator, you can alter how frequently the fixture executes. Possible values for scope are: `function`, `class`, `module`, or `session`.

Expanding upon our example:

```python
import pytest

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

def test_approval(token, accounts):
    token.approve(accounts[1], 500, {'from': accounts[0]})
    assert token.allowance(accounts[0], accounts[1]) == 500

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

By applying a `module` scope to the the `token` fixture, the contract is only deployed once and the same *`Contract`* instance is used for both `test_approval` and `test_transfer`.

Fixture of higher-scopes (such as `session` or `module`) are always instantiated before lower-scoped fixtures (such as `function`). The relative order of fixtures of same scope follows the declared order in the test function and honours dependencies between fixtures. The only exception to this rule is isolation fixtures, which are expained below.

16.2.3 Isolation Fixtures

In many cases you will want isolate your tests from one another by resetting the local environment. Without isolation, it is possible that the outcome of a test will be dependent on actions performed in a previous test.

Brownie provides two fixtures that are used to handle isolation:

- *`module_isolation`* is a module scoped fixture. It 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`* is function scoped. It 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.

Isolation fixtures are **always the first fixture within their scope to execute**. You can be certain that any action performed within a fuction-scoped fixture will happend *after* the isolation snapshot.

To apply an isolation fixture to all tests in a module, require it in another fixture and include the autouse parameter:

```
import pytest

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

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

16.2.4 Defining a Shared Initial State

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.

16.3 Markers

A marker is a decorator applied to a test function. Markers are used to pass meta data about the test which is accessible by fixtures and plugins.

To apply a marker to a specific test, use the `@pytest.mark` decorator:

```python
@pytest.mark.foo
def test_with_example_marker():
    pass
```

To apply markers at the module level, add the `pytestmark` global variable:

```python
import pytest

pytestmark = [pytest.mark.foo, pytest.mark.bar]
```

Along with the standard pytest markers, Brownie provides additional markers specific to smart contract testing. See the *markers reference* section of the documentation for more information.

16.4 Handling Reverted Transactions

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

```python
import brownie

def test_transfer_reverts(accounts, Token):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, 1e23)
    with brownie.reverts():
        token.transfer(accounts[1], 1e24, {'from': accounts[0]})
```

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

```python
import brownie

def test_transfer_reverts(accounts, Token):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, 1e23)
    with brownie.reverts("Insufficient Balance"):
        token.transfer(accounts[1], 1e24, {'from': accounts[0]})
```

16.4.1 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 Solidity, or `# dev:` in Vyper. Priority is always given to compiled revert strings. Some examples:

```
1 function revertExamples(uint a) external {
2     require(a != 2, "is two");
3     require(a != 3); // dev: is three
4     require(a != 4, "cannot be four"); // dev: is four
5     require(a != 5); // is five
6 }
```

- 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'>

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

When there is an error string included in the code, you can still access the dev revert reason via *TransactionReceipt.dev_revert_msg*:

```
>>> tx = test.revertExamples(4)
Transaction sent: 0xd9e0fb1bd6532f6aec972fc8aef806a8d8b894349cf5c82c487335625db8d0ef
test.revertExamples confirmed (cannot be four) - block: 3   gas used: 31337 (6.66%)
<Transaction object
↪'0xd9e0fb1bd6532f6aec972fc8aef806a8d8b894349cf5c82c487335625db8d0ef'>

>>> tx.revert_msg
'cannot be four'

>>> tx.dev_revert_msg
'dev: is four'
```

16.5 Parametrizing Tests

The `@pytest.mark.parametrize` marker enables parametrization of arguments for a test function. Here is a typical example of a parametrized test function, checking that a certain input results in an expected output:

```python
import pytest

@pytest.mark.parametrize('amount', [0, 100, 500])
def test_transferFrom_reverts(token, accounts, amount):
    token.approve(accounts[1], amount, {'from': accounts[0]})
    assert token.allowance(accounts[0], accounts[1]) == amount
```

In the example the `@parametrize` decorator defines three different values for `amount`. The `test_transferFrom_reverts` is executed three times using each of them in turn.

You can achieve a similar effect with the `@given` decorator to automatically generate parametrized tests from a defined range:

```
from brownie.test import given, strategy

@given(amount=strategy('uint', max_value=1000)
def test_transferFrom_reverts(token, accounts, amount):
    token.approve(accounts[1], amount, {'from': accounts[0]})
    assert token.allowance(accounts[0], accounts[1]) == amount
```

This technique is known as *property-based testing*. To learn more, read *Property-Based Testing*.

16.6 Testing against Other Projects

The `pm` fixture provides access to packages that have been installed with the *Brownie package manager*. Using this fixture, you can write test cases that verify interactions between your project and another project.

`pm` is a function that accepts a project ID as an argument and returns a *`Project`* object. This way you can deploy contracts from the package and deliver them as fixtures to be used in your tests:

```
@pytest.fixture(scope="module")
def compound(pm, accounts):
    ctoken = pm('defi.snakecharmers.eth/compound@1.1.0').CToken
    yield ctoken.deploy({'from': accounts[0]})
```

Be sure to add required testing packages to your project *dependency list*.

16.7 Running Tests

To run the complete test suite:

```
$ brownie test
```

Or to run a specific test:

```
$ brownie test tests/test_transfer.py
```

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.

16.7.1 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:

```
$ brownie test --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

16.7.2 Interactive Debugging

The `--interactive` flag allows you to debug your project while running your tests:

```
$ brownie test --interactive
```

When using interactive mode, Brownie immediately prints the traceback for each failed test and then opens a console. You can interact with the deployed contracts and examine the transaction history to help determine what went wrong.

- Deployed *`ProjectContract`* objects are available within their associated *`ContractContainer`*
- *`TransactionReceipt`* objects are in the *`TxHistory`* container, available as `history`
- Use *`chain.undo`* and *`chain.redo`* to move backward and forward through recent transactions

Once you are finished, type `quit()` to continue with the next test.

See *Inspecting and Debugging Transactions* for more information on Brownie's debugging functionality.

16.7.3 Evaluating Gas Usage

To generate a gas profile report, add the `--gas` flag:

```
$ brownie test --gas
```

When the tests complete, a report will display:

```
Gas Profile:
Token <Contract>
   ├─ constructor   -  avg: 1099591  low: 1099591  high: 1099591
   ├─ transfer      -  avg:   43017  low:   43017  high:   43017
   └─ approve       -  avg:   21437  low:   21437  high:   21437
Storage <Contract>
   ├─ constructor   -  avg:  211445  low:  211445  high:  211445
   └─ set           -  avg:   21658  low:   21658  high:   21658
```

16.7.4 Evaluating Coverage

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

```
$ brownie test --coverage
```

When the tests complete, a report will display:

```
contract: Token - 80.8%
  Token.allowance - 100.0%
  Token.approve - 100.0%
  Token.balanceOf - 100.0%
  Token.transfer - 100.0%
  Token.transferFrom - 100.0%
  SafeMath.add - 75.0%
  SafeMath.sub - 75.0%
  Token.<fallback> - 0.0%

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 Reports* for more information.

You can exclude specific contracts or source files from this report by modifying your project's *configuration file*.

16.7.5 Using `xdist` for Distributed Testing

Brownie is compatible with the pytest-xdist plugin, allowing you to parallelize test execution. In large test suites this can greatly reduce the total runtime.

You may wish to read an overview of how xdist works if you are unfamiliar with the plugin.

To run your tests in parralel, include the `-n` flag:

```
$ brownie test -n auto
```

Tests are distributed to workers on a per-module basis. An *isolation fixture* must be applied to every test being executed, or `xdist` will fail after collection. This is because without proper isolation it is impossible to ensure consistent behaviour between test runs.

CHAPTER 17

Fixture and Marker Reference

Brownie includes custom *fixtures* and *markers* that can be used when testing your project.

17.1 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 the documentation listed under “Core Functionality” in the table of contents.

`accounts`

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

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

`a`

Short form of the `accounts` fixture.

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

`chain`

Yields an `Chain` object, used to access block data and interact with the local test chain.

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

    chain.reset()
    assert accounts[1].balance() == balance
```

`Contract`

Yields the `Contract` class, used to interact with contracts outside of the active project.

```python
@pytest.fixture(scope="session")
def dai(Contract):
    yield Contract.from_explorer(
↪"0x6B175474E89094C44Da98b954EedeAC495271d0F")
```

history

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

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

interface

Yields the *InterfaceContainer* object for the active project, which provides access to project interfaces.

```python
@pytest.fixture(scope="session")
def dai(interface):
    yield interface.Dai("0x6B175474E89094C44Da98b954EedeAC495271d0F")
```

pm

Callable fixture that provides access to *Project* objects, used for testing against installed packages.

```python
@pytest.fixture(scope="module")
def compound(pm, accounts):
    ctoken = pm('defi.snakecharmers.eth/compound@1.1.0').CToken
    yield ctoken.deploy({'from': accounts[0]})
```

state_machine

Yields the *state_machine* method, used for running a *stateful test*.

```python
def test_stateful(Token, accounts, state_machine):
    token = Token.deploy("Test Token", "TST", 18, 1e23, {'from':␣
↪accounts[0]})

    state_machine(StateMachine, accounts, token)
```

web3

Yields a *Web3* object.

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

17.2 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.

```python
def test_token_deploys(Token, accounts):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, 1e24)
    assert token.name() == "Test Token"
```

17.3 Isolation Fixtures

Isolation fixtures are used ensure a clean test environment when running tests, and to prevent the results of a test from affecting subsequent tests. See *Isolation Fixtures* for information on how to use these fixtures.

`module_isolation`

Resets the local chain before running and after completing the test module.

`fn_isolation`

Takes a snapshot of the chain before running a test and reverts to it after the test completes.

17.4 Markers

Brownie provides the following *markers* for use within your tests:

`pytest.mark.`**`require_network`**`(`*network_name*`)`

Mark a test so that it only runs if the active network is named `network_name`. This is useful when you have some tests intended for a local development environment and others for a forked mainnet.

```
@pytest.mark.require_network("mainnet-fork")
def test_almost_in_prod():
    pass
```

`pytest.mark.`**`no_call_coverage`**

Only evaluate coverage for transactions made during this test, not calls.

This marker is useful for speeding up slow tests that involve many calls to the same view method.

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

@pytest.mark.no_call_coverage
def test_no_call_cov(Token):
    # this call is handled as a call, the test execution is quicker
    assert token.balanceOf(accounts[1]) == 100
```

`pytest.mark.`**`skip_coverage`**

Skips a test if coverage evaluation is active.

```
@pytest.mark.skip_coverage
def test_heavy_lifting():
    pass
```

CHAPTER 18

Property-Based Testing

Brownie utilizes the `hypothesis` framework to allow for property-based testing.

Much of the content in this section is based on the official hypothesis.works website. To learn more about property-based testing, you may wish to read this series of introductory articles or view the official Hypothesis documentation.

18.1 What is Property-Based Testing?

Property-based testing is a powerful tool for locating edge cases and discovering faulty assumptions within your code.

The core concept behind property-based testing is that rather than writing a test for a single scenario, you write tests that describe a range of scenarios and then let your computer explore the possibilities for you rather than having to hand-write every one yourself.

The basic process consists of:

1. Choose a function within your smart contract that you wish to test.
2. Specify a range of inputs for this function that should always yield the same result.
3. Call the function with random data from your specification.
4. Make an assertion about the result.

Using this technique, each test is run many times with different arbitrary data. If an example is found where the assertion fails, an attempt is made to find the simplest case possible that still causes the problem. This example is then stored in a database and repeated in each subsequent tests to ensure that once the issue is fixed, it stays fixed.

18.2 Writing Tests

To begin writing property-based tests, import the following two methods:

```
from brownie.test import given, strategy
```

`brownie.test.`**`given`**`()`

A decorator for turning a test function that accepts arguments into a randomized test.

When using Brownie, this is the main entry point to property-based testing. This is a thin wrapper around `hypothesis.given`, the API is identical.

> **Warning:** Be sure to import `@given` from Brownie and not directly from Hypothesis. Importing the function directly can cause issues with test isolation.

`brownie.test.`**`strategy`**`()`

A method for creating *test strategies* based on ABI types.

A test using Hypothesis consists of two parts: A function that looks like a normal pytest test with some additional arguments, and a `@given` decorator that specifies how to those arguments are provided.

Here is a basic example, testing the `transfer` function of an ERC20 token contract.

```python
from brownie import accounts
from brownie.test import given, strategy

@given(value=strategy('uint256', max_value=10000))
def test_transfer_amount(token, value):
    balance = token.balanceOf(accounts[0])
    token.transfer(accounts[1], value, {'from': accounts[0]})

    assert token.balanceOf(accounts[0]) == balance - value
```

When this test runs:

1. The setup phase of all pytest fixtures are executed in their regular order.
2. A snapshot of the current chain state is taken.
3. `strategy` generates a random integer value and assigns it to the `amount` keyword argument.
4. The test is executed.
5. The chain is reverted to the snapshot taken in step 2.
6. Steps 3-5 are repeated 50 times, or until the test fails.
7. The teardown phase of all pytest fixtures are executed in their normal order.

It is possible to supply multiple strategies via `@given`. In the following example, we add a `to` argument using an address strategy.

```python
from brownie import accounts
from brownie.test import given, strategy

@given(
    to=strategy('address', exclude=accounts[0]),
    value=strategy('uint256', max_value=10000),
)
def test_transfer_amount(token, to, value):
    balance = token.balanceOf(accounts[0])
    token.transfer(to, value, {'from': accounts[0]})

    assert token.balanceOf(accounts[0]) == balance - value
    assert token.balanceOf(to) == value
```

18.3 Strategies

The key object in every test is a *strategy*. A strategy is a recipe for describing the sort of data you want to generate. Brownie provides a `strategy` method that generates strategies for any given ABI type.

```
>>> from brownie.test import strategy
>>> strategy('uint8')
integers(min_value=0, max_value=255)
```

Each strategy object contains an `example` method that you can call in the console to explore the types of data that will be generated.

```
>>> st = strategy('uint8')
>>> st.example()
243
>>> st.example()
77
```

`strategy` accepts different keyword arguments depending on the ABI type.

18.3.1 Type Strategies

The following strategies correspond to types within Solidity and Vyper.

Address

Base strategy: `hypothesis.strategies.sampled_from`

`address` strategies yield *`Account`* objects from the *`Accounts`* container.

Optional keyword arguments:

- `length`: The number of *`Account`* objects to include in the strategy. If the *`Accounts`* container holds less than this number of objects, the entire container is used.
- `excludes`: An object, iterable or callable used to filter strategy results.

```
>>> strategy('address')
sampled_from(accounts)

>>> strategy('address').example()
<Account '0x33A4622B82D4c04a53e170c638B944ce27cffce3'>
```

Bool

Base strategy: `hypothesis.strategies.booleans`

`bool` strategies yield `True` or `False`.

This strategy does not accept any keyword arguments.

```
>>> strategy('bool')
booleans()

>>> strategy('bool').example()
True
```

Bytes

Base strategy: hypothesis.strategies.binary

bytes strategies yield byte strings.

All bytes strategies accept the following keyword arguments:

- excludes: An object, iterable or callable used to filter strategy results.

For fixed length values (bytes1` ...``bytes32) the strategy always generates bytes of exactly the given length. For dynamic bytes arrays (bytes), the minimum and maximum length may be specified using keyord arguments:

- min_size: Minimum length for each returned value. The default value is 1.
- max_size: Maximum length for each returned value. The default value is 64.

```
>>> strategy('bytes32')
binary(min_size=32, max_size=32)
>>> strategy('bytes', max_size=16)
binary(min_size=1, max_size=16)

>>> strategy('bytes8').example()
b'\xb8\xd6\xaa\xcbR\x0f\xb88'
```

Decimal

Base strategy: hypothesis.strategies.decimals

decimal strategies yield decimal.Decimal instances.

Optional keyword arguments:

- min_value: The maximum value to return. The default is -2**127 (the lower bound of Vyper's decimal type). The given value is converted to *Fixed*.
- max_value: The maximum value to return. The default is 2**127-1 (the upper bound of Vyper's decimal type). The given value is converted to *Fixed*.
- places: The number of decimal points to include. The default value is 10.
- excludes: An object, iterable or callable used to filter strategy results.

```
>>> strategy('decimal')
decimals(min_value=-170141183460469231731687303715884105728, max_
↪value=170141183460469231731687303715884105727, places=10)

>>> strategy('decimal').example()
Decimal('44.8234019327')
```

Integer

Base strategy: hypothesis.strategies.integers

int and uint strategies yield integer values.

Optional keyword arguments:

- min_value: The maximum value to return. The default is the lower bound for the given type. The given value is converted to *Wei*.

- max_value: The maximum value to return. The default is the upper bound for the given type. The given value is converted to *Wei*.
- excludes: An object, iterable or callable used to filter strategy results.

```
>>> strategy('uint32')
integers(min_value=0, max_value=4294967295)
>>> strategy('int8')
integers(min_value=-128, max_value=127)
>>> strategy('uint', min_value="1 ether", max_value="25 ether")
integers(min_value=1000000000000000000, max_value=25000000000000000000)

>>> strategy('uint').example()
156806085
```

String

Base strategy: hypothesis.strategies.text

string strategies yield unicode text strings.

Optional keyword arguments:

- min_size: Minimum length for each returned value. The default value is 0.
- max_size: Maximum length for each returned value. The default value is 64.
- excludes: An object, iterable or callable used to filter strategy results.

```
>>> strategy('string')
text(max_size=64)
>>> strategy('string', min_size=12, max_size=23)
text(min_size=12, max_size=23)

>>> strategy('string').example()
'\x02\x14\x01\U0009b3c5'
```

18.3.2 Sequence Strategies

Along with the core strategies, Brownie also offers strategies for generating array or tuple sequences.

Array

Base strategy: hypothesis.strategies.lists

Array strategies yield lists of strategies for the base array type. It is possible to generate arrays of both fixed and dynamic length, as well as multidimensional arrays.

Optional keyword arguments:

- min_length: The minimum number of items inside a dynamic array. The default value is 1.
- max_length: The maximum number of items inside a dynamic array. The default value is 8.
- unique: If True, each item in the list will be unique.

For multidimensional dynamic arrays, `min_length` and `max_length` may be given as a list where the length is equal to the number of dynamic dimensions.

You can also include keyword arguments for the base type of the array. They will be applied to every item within the generated list.

```
>>> strategy('uint32[]')
lists(elements=integers(min_value=0, max_value=4294967295), min_length=1, max_
↪length=8)
>>> strategy('uint[3]', max_value=42)
lists(elements=integers(min_value=0, max_value=42), min_length=3, max_length=3)

>>> strategy('uint[3]', max_value=42).example()
[16, 23, 14]
```

Tuple

Base strategy: `hypothesis.strategies.tuples`

Tuple strategies yield tuples of mixed strategies according to the given type string.

This strategy does not accept any keyword arguments.

```
>>> strategy('(int16,bool)')
tuples(integers(min_value=-32768, max_value=32767), booleans())
>>> strategy('(uint8,(bool,bytes4))')
tuples(integers(min_value=0, max_value=255), tuples(booleans(), binary(min_size=4,␣
↪max_size=4)))

>>> strategy('(uint16,bool)').example()
(47628, False)
```

18.3.3 Contract Strategies

The `contract_strategy` function is used to draw from *`ProjectContract`* objects within a *`ContractContainer`*.

`brownie.test.`**`contract_strategy`**`(`*contract_name*`)`

Base strategy: `hypothesis.strategies.sampled_from`

A strategy to access *`ProjectContract`* objects.

- `contract_name`: The name of the contract, given as a string

```
>>> ERC20
[<ERC20 Contract '0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87'>, <ERC20 Contract
↪'0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6'>]

>>> from brownie.test import contract_strategy
>>> contract_strategy('ERC20')
sampled_from(ERC20)

>>> contract_strategy('ERC20').example()
<ERC20 Contract '0x602C71e4DAC47a042Ee7f46E0aee17F94A3bA0B6'>
```

18.3.4 Other Strategies

All of the strategies that Brownie provides are based on core strategies from the `hypothesis.strategies` library. If you require something more specific or complex than Brownie offers, you can also directly use hypothesis strategies.

See the Hypothesis strategy documentation for more information on available strategies and how they can be customized.

18.4 Settings

Depending on the scope and complexity of your tests, it may be necessary to modify the default settings for how property-based tests are run.

The mechanism for doing this is the `hypothesis.settings` object. You can set up a `@given` based test to use this using a settings decorator:

```python
from brownie.test import given
from hypothesis settings

@given(strategy('uint256'))
@settings(max_examples=500)
def test_this_thoroughly(x):
    pass
```

You can also affect the settings permanently by adding a `hypothesis` field to your project's `brownie-config.yaml` file:

```yaml
   hypothesis:
       max_examples: 500

See the :ref:`Configuration File<config>` documentation for more information.
```

18.4.1 Available Settings

Note: See the Hypothesis settings documentation for a complete list of available settings. This section only lists settings where the default value has been changed from the Hypothesis default.

`deadline`

The number of milliseconds that each individual example within a test is allowed to run. Tests that take longer than this time will be considered to have failed.

Because Brownie test times can vary widely, this property has been disabled by default.

default-value: `None`

`max_examples`

The maximum number of times a test will be run before considering it to have passed.

For tests involving many complex transactions you may wish to reduce this value.

default-value: `50`

`report_multiple_bugs`

Because Hypothesis runs each test many times, it can sometimes find multiple bugs in a single run. Reporting all of them at once can be useful, but also produces significantly longer and less descriptive output when compared to reporting a single error.

default-value: `False`

`stateful_step_count`

The maximum number of rules to execute in a stateful program before ending the run and considering it to have passed.

For more complex state machines you may wish to increase this value - however you should keep in mind that this can result in siginificantly longer execution times.

default-value: `10`

CHAPTER 19

Stateful Testing

Stateful testing is a more advanced method of *property-based testing* used to test complex systems. In a stateful test you define a number of actions that can be combined together in different ways, and Hypothesis attempts to find a sequence of those actions that results in a failure. This is useful for testing complex contracts or contract-to-contract interactions where there are many possible states.

Brownie utilizes the `hypothesis` framework to allow for stateful testing.

Much of the content in this section is based on the official hypothesis.works website. To learn more about stateful testing, you may wish to read the following articles:

- Rule Based Stateful Testing by David R. MacIver
- Solving the Water Jug Problem from Die Hard 3 with TLA+ and Hypothesis by Nicholas Chammas
- Hypothesis Documentation on stateful testing

Warning: This functionality is still under development and should be considered experimental. Use common sense when evaluating the results, and if you encounter any problems please open an issue on Github.

19.1 Rule-based State Machines

A state machine is a class used within stateful testing. It defines the initial test state, a number of actions outlining the structure that the test will execute in, and invariants that should not be violated during execution.

Note: Unlike regular Hypothesis state machines, Brownie state machines should not subclass `RuleBasedStateMachine`.

19.1.1 Rules

At the core of every state machine are one or more *rules*. Rules are class methods that are very similar to `@given` based tests; they receive values drawn from strategies and pass them to a user defined test function. The key difference is that where `@given` based tests run independently, rules can be chained together - a single stateful test run may involve multiple rule invocations, which may interact in various ways.

Any state machine method named `rule` or beginning with `rule_` is treated as a rule.

```
class StateMachine:

    def rule_one(self):
        # performs a test action

    def rule_two(self):
        # performs another, different test action
```

19.1.2 Initializers

There is also a special type of rule known as an *initializer*. These are rules that are guaranteed to be executed at most one time at the beginning of a run (i.e. before any normal rule is called). They may be called in any order, or not at all, and the order will vary from run to run.

Any state machine method named `initialize` or beginning with `initialize_` is treated as an initializer.

```
class StateMachine:

    def initialize(self):
        # this method may or may not be called prior to rule_two

    def rule(self):
        # once this method is called, initialize will not be called during the test run
```

19.1.3 Strategies

A state machine should contain one or more *strategies*, in order to provide data to it's rules.

Strategies must be defined at the class level, typically before the first function. They can be given any name.

Similar to how fixtures work within pytest tests, state machine rules receive strategies by referencing them within their arguments. This is shown in the following example:

```
class StateMachine:

    st_uint = strategy('uint256')
    st_bytes32 = strategy('bytes32')

    def initialize(self, st_uint):
        # this method draws from the uint256 strategy

    def rule(self, st_uint, st_bytes32):
        # this method draws from both strategies

    def rule_two(self, value="st_uint", othervalue="st_uint"):
        # this method draws from the same strategy twice
```

19.1.4 Invariants

Along with rules, a state machine often defines *invariants*. These are properties that should remain unchanged, regardless of any actions performed by the rules. After each rule is executed, every invariant method is always called to ensure that the test has not failed.

Any state machine method named `invariant` or beginning with `invariant_` is treated as an invariant. Invariants are meant for verifying correctness of state; they cannot receive strategies.

```
class StateMachine:

    def rule_one(self):
        pass

    def rule_two(self):
        pass

    def invariant(self):
        # assertions in this method should always pass regardless
        # of actions in both rule_one and rule_two
```

19.1.5 Setup and Teardown

A state machine may optionally include setup and teardown procedures. Similar to pytest fixtures, setup and teardown methods are available to execute logic on a per-test and per-run basis.

classmethod `StateMachine.`**`__init__`**(*cls*, **args*)

This method is called once, prior to the chain snapshot taken before the first test run. It is run as a class method - changes made to the state machine will persist through every run of the test.

`__init__` is the only method that can be used to pass external data into the state machine. In the following example, we use it to pass the *accounts* fixture, and a deployed instance of a token contract:

```
class StateMachine:

    def __init__(cls, accounts, token):
        cls.accounts = accounts
        cls.token = token


def test_stateful(Token, accounts, state_machine):
    token = Token.deploy("Test Token", "TST", 18, 1e23, {'from': accounts[0]})

    # state_machine forwards all the arguments to StateMachine.__init__
    state_machine(StateMachine, accounts, token)
```

classmethod `StateMachine.`**`setup`**(*self*)

This method is called at the beginning of each test run, immediately after chain is reverted to the snapshot. Changes applied during `setup` will only have an effect for the upcoming run.

classmethod `StateMachine.`**`teardown`**(*self*)

This method is called at the end of each successful test run, prior to the chain revert. `teardown` is not called if the run fails.

classmethod `StateMachine.`**`teardown_final`**(*cls*)

This method is called after the final test run has completed and the chain has been reverted. `teardown_final` is called regardless of whether the test passed or failed.

19.2 Test Execution Sequence

A Brownie stateful test executes in the following sequence:

1. The setup phase of all pytest fixtures are executed in their regular order.
2. If present, the `StateMachine.__init__` method is called.
3. A snapshot of the current chain state is taken.
4. If present, the `StateMachine.setup` method is called.
5. Zero or more `StateMachine` initialize methods are called, in no particular order.
6. One or more `StateMachine` rule methods are called, in no particular order.
7. After each initialize and rule, every `StateMachine` invariant method is called.
8. If present, the `StateMachine.teardown` method is called.
9. The chain is reverted to the snapshot taken in step 3.
10. Steps 4-9 are repeated 50 times, or until the test fails.
11. If present, the `StateMachine.teardown_final` method is called.
12. The teardown phase of all pytest fixtures are executed in their normal order.

19.3 Writing Stateful Tests

To write a stateful test:

1. Create a state machine class.
2. Create a regular pytest-style test that includes the *`state_machine`* fixture.
3. Within the test, call *`state_machine`* with the state machine as the first argument.

`brownie.test.stateful.`**`state_machine`**`(`*state_machine_class*, **args*, *settings=None*`)`

Executes a stateful test.

- `state_machine_class`: A state machine class to be used in the test. Be sure to pass the class itself, not an instance of the class.
- `*args`: Any arguments given here will be passed to the state machine's `__init__` method.
- `settings`: An optional `dict` of *Hypothesis settings* that will replace the defaults for this test only.

This method is available as a pytest fixture *`state_machine`*.

19.3.1 Basic Example

As a basic example, we will create a state machine to test the following Vyper `Depositer` contract. This is very simple contract with two functions and a public mapping. Anyone can deposit ether for another account using the `deposit_for` method, or withdraw deposited ether using `withdraw_from`.

```
deposited: public(HashMap[address, uint256])

@external
@payable
```

(continues on next page)

(continued from previous page)

```
def deposit_for(_receiver: address) -> bool:
    self.deposited[_receiver] += msg.value
    return True

@external
def withdraw_from(_value: uint256) -> bool:
    assert self.deposited[msg.sender] >= _value, "Insufficient balance"
    self.deposited[msg.sender] = _value
    send(msg.sender, _value)
    return True
```

If you looked closely you may have noticed a major issue in the contract code. If not, don't worry! We're going to find it using our test.

Here is a state machine and test function we can use to test the contract.

```
import brownie
from brownie.test import strategy

class StateMachine:

    value = strategy('uint256', max_value="1 ether")
    address = strategy('address')

    def __init__(cls, accounts, Depositer):
        # deploy the contract at the start of the test
        cls.accounts = accounts
        cls.contract = Depositer.deploy({'from': accounts[0]})

    def setup(self):
        # zero the deposit amounts at the start of each test run
        self.deposits = {i: 0 for i in self.accounts}

    def rule_deposit(self, address, value):
        # make a deposit and adjust the local record
        self.contract.deposit_for(address, {'from': self.accounts[0], 'value': value})
        self.deposits[address] += value

    def rule_withdraw(self, address, value):
        if self.deposits[address] >= value:
            # make a withdrawal and adjust the local record
            self.contract.withdraw_from(value, {'from': address})
            self.deposits[address] -= value
        else:
            # attempting to withdraw beyond your balance should revert
            with brownie.reverts("Insufficient balance"):
                self.contract.withdraw_from(value, {'from': address})

    def invariant(self):
        # compare the contract deposit amounts with the local record
        for address, amount in self.deposits.items():
            assert self.contract.deposited(address) == amount


def test_stateful(Depositer, accounts, state_machine):
    state_machine(StateMachine, accounts, Depositer)
```

When this test is executed, it will call `rule_deposit` and `rule_withdraw` using random data from the given strategies until it encounters a state which violates one of the assertions. If this happens, it repeats the test in an attempt to find the shortest path and smallest data set possible that reproduces the error. Finally it saves the failing conditions to be used in future tests, and then delivers the following output:

```
    def invariant(self):
        for address, amount in self.deposits.items():
>           assert self.contract.deposited(address) == amount
E           AssertionError: assert 0 == 1

Falsifying example:
state = BrownieStateMachine()
state.rule_deposit(address=<Account '0x33A4622B82D4c04a53e170c638B944ce27cffce3'>,␣
↪value=1)
state.rule_withdraw(address=<Account '0x33A4622B82D4c04a53e170c638B944ce27cffce3'>,␣
↪value=0)
state.teardown()
```

From this we can see the sequence of calls leading up to the error, and that the failed assertion is that `self.contract.deposited(address)` is zero, when we expected it to be one. We can infer that the contract is incorrectly adjusting balances within the withdraw function. Looking at that function:

```
9   @external
10  def withdraw_from(_value: uint256) -> bool:
11      assert self.deposited[msg.sender] >= _value, "Insufficient balance"
12      self.deposited[msg.sender] = _value
13      send(msg.sender, _value)
14      return True
```

On line 12, rather than subtracting `_value`, the balance is being *set* to `_value`. We found the bug!

19.3.2 More Examples

Here are some links to repositories that make use of stateful testing. If you have a project that you would like included here, feel free to edit this document and open a pull request, or let us know about it on Gitter.

- celioggr/erc20-pbt : A testing framework based in Property-based testing for assessing the correctness and compliance of ERC-20 contracts.
- iamdefinitelyahuman/NFToken: A non-fungible implementation of the ERC20 standard.
- apguerrera/DreamFrames: Buy and sell frames in movies.
- curvefi/curve-dao-contracts: Vyper contracts used by Curve DAO

19.4 Running Stateful Tests

By default, stateful tests are included when you run your test suite. There is no special action required to invoke them.

You can choose to exclude stateful tests, or to *only* run stateful tests, with the `--stateful` flag. This can be useful to split the test suite when setting up continuous integration.

To only run stateful tests:

```
$ brownie test --stateful true
```

To skip stateful tests:

```
$ brownie test --stateful false
```

When a stateful test is active the console shows a spinner that rotates each time a run of the test has finished. If the color changes from yellow to red, it means the test has failed and hypothesis is now searching for the shortest path to the failure.

CHAPTER 20

Coverage Evaluation

To check your unit test coverage:

```
$ brownie test --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*.

20.1 Viewing Coverage Data

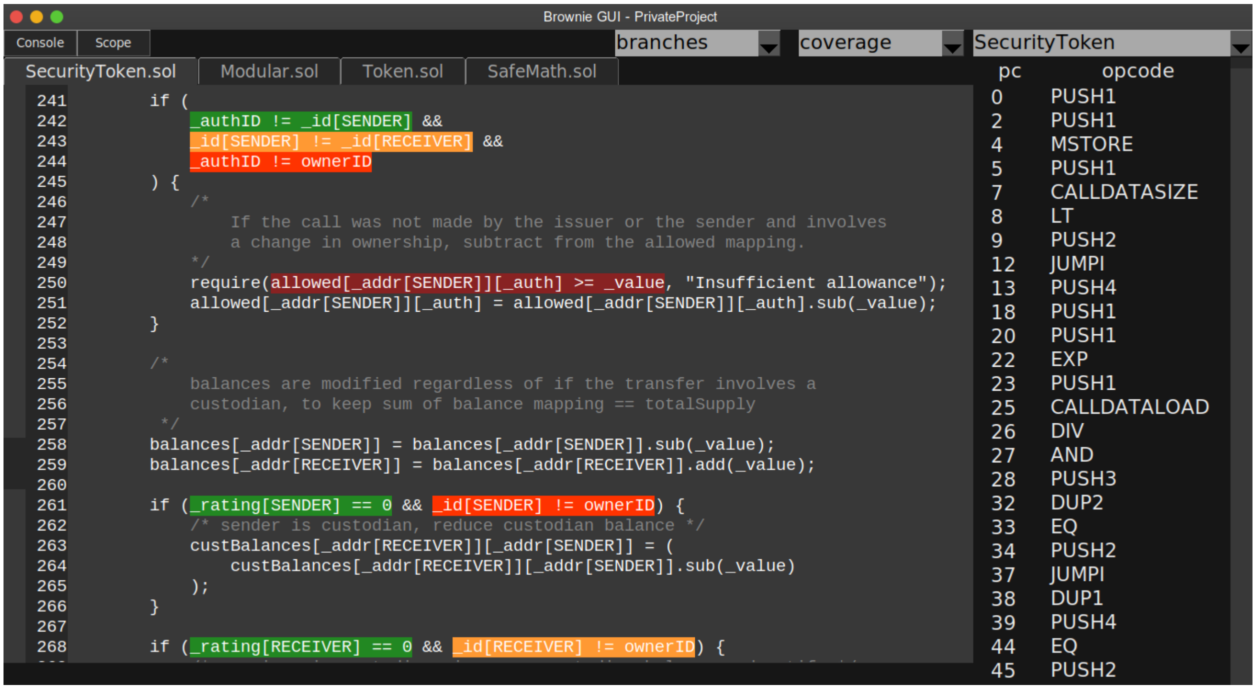
For an in-depth examination of your test coverage, first open the Brownie GUI:

```
brownie gui
```

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



```
        if (
            _authID != _id[SENDER] &&
            _id[SENDER] != _id[RECEIVER] &&
            _authID != ownerID
        ) {
            /*
                If the call was not made by the issuer or the sender and involves
                a change in ownership, subtract from the allowed mapping.
            */
            require(allowed[_addr[SENDER]][_auth] >= _value, "Insufficient allowance");
            allowed[_addr[SENDER]][_auth] = allowed[_addr[SENDER]][_auth].sub(_value);
        }

        /*
            balances are modified regardless of if the transfer involves a
            custodian, to keep sum of balance mapping == totalSupply
         */
        balances[_addr[SENDER]] = balances[_addr[SENDER]].sub(_value);
        balances[_addr[RECEIVER]] = balances[_addr[RECEIVER]].add(_value);

        if (_rating[SENDER] == 0 && _id[SENDER] != ownerID) {
            /* sender is custodian, reduce custodian balance */
            custBalances[_addr[RECEIVER]][_addr[SENDER]] = (
                custBalances[_addr[RECEIVER]][_addr[SENDER]].sub(_value)
            );
        }

        if (_rating[RECEIVER] == 0 && _id[RECEIVER] != ownerID) {
```

```
pc     opcode
0      PUSH1
2      PUSH1
4      MSTORE
5      PUSH1
7      CALLDATASIZE
8      LT
9      PUSH2
12     JUMPI
13     PUSH4
18     PUSH1
20     PUSH1
22     EXP
23     PUSH1
25     CALLDATALOAD
26     DIV
27     AND
28     PUSH3
32     DUP2
33     EQ
34     PUSH2
37     JUMPI
38     DUP1
39     PUSH4
44     EQ
45     PUSH2
```

20.2 How Coverage Evaluation Works

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.

20.3 Improving Performance

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.

Some things to keep in mind that can help to reduce your test runtime when evaluating coverage:

1. Coverage is analyzed on a per-transaction basis, and the results are cached. If you repeat an identical transaction, Brownie will not analyze it the 2nd time. Keep this in mind when designing and sequencing setup fixtures.
2. For tests that involve many calls to the same getter method, use the *no_call_coverage* marker to significantly speed execution.

3. Omit very complex tests altogether with the `skip_coverage` marker.

4. If possible, always run your tests in parralel with *xdist*.

You can use the `--durations` flag to view a profile of your slowest tests. You may find good candidates for optimization, or the use of the `no_call_coverage` and `skip_coverage` fixtures.

CHAPTER 21

Security Analysis with MythX

Brownie is integrated with the MythX analysis API to allow automated security scans of your project.

MythX is a smart contract security service that scans your project for vulnerabilities using static analysis, dynamic analysis, and symbolic execution. It runs in three modes:

1. **Quick mode** which is effective at finding bad coding patterns and low complexity-bugs (available to free users)
2. **Standard mode** which takes longer to run, but can locate complex security issues (available to Dev users)
3. **Deep mode** which takes even longer to run, but is able to find deep, hidden vulnerabilities (available to Pro users)

MythX offers both free and paid services. To learn more about how it works you may wish to read MythX Pro Security Analysis Explained by Bernhard Mueller.

21.1 Authentication

Before you can submit your contracts for analysis you must sign up for a MythX account. Next, login to your account and obtain a JWT token so you can authenticate to the API.

The preferred way to pass your JWT token is via the `MYTHX_API_KEY` environment variable. You can set it with the following command:

```
$ export MYTHX_API_KEY=YourToken
```

If this is not possible, you may also pass it via the `--api-key` commandline option:

```
$ brownie analyze --api-key=<string>
```

21.2 Scanning for Vulnerabilities

To quickly scan your project for vulnerabilities:

```
$ brownie analyze
```

This will send the compiled build artifacts to MythX for analysis. You will receive updates on the status of the scan; the entire process should take around three minutes.

To perform a standard scan:

```
$ brownie analyze --mode=standard
```

Note that a deep scan requires authentication and takes approximately half an hour to complete.

If you include the `--async` flag Brownie will submit the job, output the pending ID and exit. You can view the finished report later through the MythX dashboard.

21.3 Viewing Analysis Results

Once analysis is finished, data about any vulnerabilities is stored in the `reports/` directory within your project. The report can be viewed using the *Brownie GUI*, or by logging into the MythX dashboard.

To view your report in the GUI, first open the GUI:

```
brownie gui
```

Alternatively, the `--gui` flag can be passed to the `analyze` subcommand to open the Brownie GUI right away after the analysis results have been received.

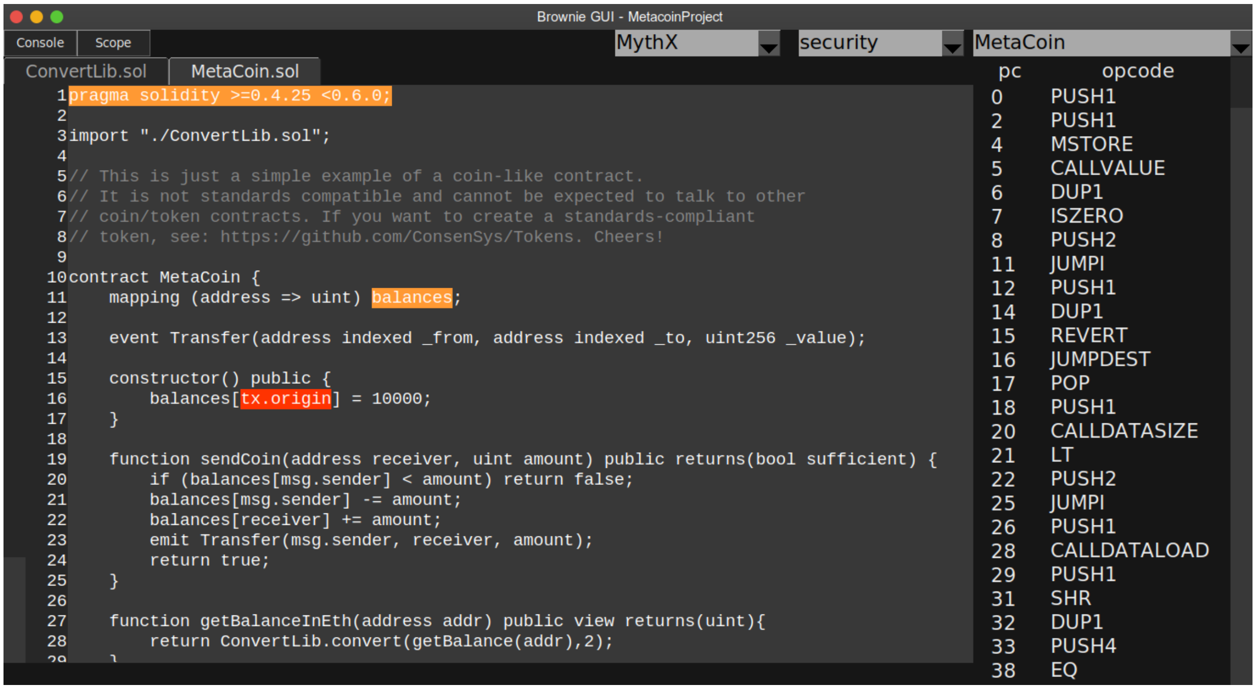
```
brownie analyze --gui
```

Click on the drop-down list in the upper right that says "Select Report" and choose "security". Then choose `MythX` in the new dropdown that appears.

If any vulnerabilities have been found, they will be highlighted based on their severity:

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

You can expand the console by clicking the `Console` button in the top left (or pressing the `C` key). Hovering the mouse over a vulnerability will displayed a more detailed explanation from the SWC registry.



```
pragma solidity >=0.4.25 <0.6.0;

import "./ConvertLib.sol";

// This is just a simple example of a coin-like contract.
// It is not standards compatible and cannot be expected to talk to other
// coin/token contracts. If you want to create a standards-compliant
// token, see: https://github.com/ConsenSys/Tokens. Cheers!

contract MetaCoin {
    mapping (address => uint) balances;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    constructor() public {
        balances[tx.origin] = 10000;
    }

    function sendCoin(address receiver, uint amount) public returns(bool sufficient) {
        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Transfer(msg.sender, receiver, amount);
        return true;
    }

    function getBalanceInEth(address addr) public view returns(uint){
        return ConvertLib.convert(getBalance(addr),2);
    }
```

pc	opcode
0	PUSH1
2	PUSH1
4	MSTORE
5	CALLVALUE
6	DUP1
7	ISZERO
8	PUSH2
11	JUMPI
12	PUSH1
14	DUP1
15	REVERT
16	JUMPDEST
17	POP
18	PUSH1
20	CALLDATASIZE
21	LT
22	PUSH2
25	JUMPI
26	PUSH1
28	CALLDATALOAD
29	PUSH1
31	SHR
32	DUP1
33	PUSH4
38	EQ

CHAPTER 22

Deployment Basics

Once your project is ready to be deployed to a persistent chain (such as the Etherem mainnet or a testnet), Brownie can be used to handle the deployments.

It is important to remember that blockchains are *permanent* and *immutable*. Once your project has been deployed there is no going back. For this reason, we highly recommend the following process when deploying to the mainnet:

1. Create a deployment script
2. Test the script on your local development environment
3. Test the script again on one of the public test networks and verify that it executed as intended
4. Use the script to deploy your project to the mainnet

Once deployment is complete you may also create an ethPM package to simplify the process for other developers who wish to interact with your project.

22.1 Writing a Deployment Script

Deployment scripts function in the same way as any other *Brownie script*, but there are a couple of things to keep in mind when writing one for a non-local network:

1. Unless you are using your own node you will have to unlock a local account prior to deploying. This is handled within the script by calling `Accounts.load`. If you have not yet added a local account to Brownie, read the documentation on *local account management*.
2. Most networks require that you to pay gas to miners. If no values are specified Brownie will calculate the gas price and limit automatically, but in some cases you may wish to manually declare these values.

Here is an small example script that unlocks a local account and uses it to deploy a `Token` contract.

```
from brownie import Token, accounts

def main():
```

(continues on next page)

(continued from previous page)

```
    acct = accounts.load('deployment_account')
    Token.deploy("My Real Token", "RLT", 18, 1e28, {'from': acct})
```

22.2 Running your Deployment Script

In order to execute your script in a live environment, you must include the `--network` flag in the command line. For example, to connect to the ropsten network and run `scripts/deploy.py`:

```
$ brownie run deploy.py --network ropsten
```

Remember that transactions are not confirmed immediately on live networks. You will see a notification on the status of each transaction, however the script will take some time to complete.

See the documentation on *network management* for more information on how to define and connect to live networks.

22.3 The Deployment Map

Brownie will maintain a `map.json` file in your `build/deployment/` folder that lists all deployed contracts on live networks, sorted by chain and contract name.

```
{
  "1": {
    "SolidityStorage": [
      "0x73B74F5f1d1f7A00d8c33bFbD09744eD90220D12",
      "0x189a7fBB0038D4b55Bd03840be0B0a38De034089"
    ],
    "VyperStorage": [
      "0xF104A50668c3b1026E8f9B0d9D404faF8E42e642"
    ]
  }
}
```

The list for each contract is sorted by the block number of the deployment with the most recent deployment first.

22.4 Interacting with Deployed Contracts

Brownie saves information about contract deployments on live networks. Once a contract has been deployed, the generated *`ProjectContract`* instance will still be available in future Brownie sessions.

The following actions will NOT remove locally stored deployment data:

- 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 deployment data within your project:

- Calling *`ContractContainer.remove`* will erase deployment information for the removed *`ProjectContract`* instances.

- Removing or renaming a contract source file within your project will cause Brownie to delete all deployment information for the removed contract.
- Deleting the `build/deployments/` directory will erase all information about deployed contracts.

To restore a deleted *ProjectContract* instance, or generate one for a deployment that was handled outside of Brownie, use the *ContractContainer.at* method.

22.5 Verifying Deployments on Etherscan

Brownie features automatic source code verification for solidity contracts on all networks supported by etherscan. To verify a contract while deploying it, add the `publish_source=True` argument:

```
acct = accounts.load('deployment_account')
Token.deploy("My Real Token", "RLT", 18, 1e28, {'from': acct}, publish_source=True)
```

Verifying already deployed contracts is also possible as long as you set the identical compiler settings:

```
token = Token.at("0x114A107C1931de1d5023594B14fc19d077FC4dfD")
Token.publish_source(token)
```

Warning: Make sure all your source files use the same compiler version, otherwise the verification will fail.

22.6 Saving Deployments on Development Networks

If you need deployment artifacts on a development network, set *dev_deployment_artifacts* to `true` in the in the project's `brownie-config.yaml` file.

These temporary deployment artifacts and the corresponding entries in *the deployment map* will be removed whenever you (re-) load a project or connect, disconnect, revert or reset your local network.

If you use a development network that is not started by brownie - for example an external instance of ganache - the deployment artifacts will not be deleted when disconnecting from that network. However, the network will be reset and the deployment artifacts deleted when you connect to such a network with brownie.

CHAPTER 23

Network Management

Brownie can be used with both development and live environments.

- A **development** environment is a local, temporary network used for testing and debugging. Brownie uses Ganache for development environments.
- A **live** environment is a non-local, persistent blockchain. This term is used to refer to both the Ethereum mainnet and testnets.

23.1 Network Configuration

Networks settings are handled using the command-line:

```
$ brownie networks
```

23.1.1 Viewing Existing Networks

Networks are broadly categorized as "development" (local, ephemeral environments) and "live" (non-local, persistent environments). Live networks are additionally categorized by chain (Ethereum, ETC, etc).

Type `brownie networks list` to view a list of existing networks:

```
$ brownie networks list
Brownie - Python development framework for Ethereum

The following networks are declared:

Ethereum
  ├─Mainnet (Infura): mainnet
  ├─Ropsten (Infura): ropsten
  ├─Rinkeby (Infura): rinkeby
  ├─Goerli (Infura): goerli
```

(continues on next page)

(continued from previous page)

```
  └─Kovan (Infura): kovan

Ethereum Classic
  ├─Mainnet: etc
  └─Kotti: kotti

Development
  ├─Ganache-CLI: development
  └─Ganache-CLI (Mainnet Fork): mainnet-fork
```

23.1.2 Adding a New Network

To add a new network:

```
$ brownie networks add [environment] [id] host=[host] [KEY=VALUE, ...]
```

When declaring a new network, the following fields must always be included:

- `environment`: the category that the network should be placed in, e.g. "Ethereum", "Ethereum Classic", or "Development"
- `id`: a unique identifier for the network, e.g. "mainnet"
- `host`: the address of the node to connect to, e.g. `https://mainnet.infura.io/v3/1234567890abcdef`

The following fields are optional:

- `name` A longer name to use for the network. If not given, `id` is used.
- `timeout`: The number of seconds to wait for a response when making an RPC call. Defaults to 30.

There are additional required and optional fields that are dependent on the type of network.

Live Networks

Live networks **must** include the following fields:

- `chainid`: The chain ID for a network. Live networks with the same chain ID share local data about *contract deployments*. See chainid.network for a list of chain IDs.

The following fields are optional for live networks:

- `explorer`: API url used by *`Contract.from_explorer`* to fetch source code. If this field is not given, you will not be able to fetch source code when using this network.

Development Networks

Development networks **must** include the following fields:

- `cmd`: The command used to launch the local RPC client, e.g. `ganache-cli`.

The following optional fields may be given for development networks, which are passed into Ganache as commandline arguments:

- `port`: The port to connect to. If not given as a unique field, it should be included within the host path.
- `gas_limit`: The block gas limit. Defaults to 6721925.

- `accounts`: The number of funded, unlocked accounts. Default 10.
- `mnemonic`: A mnemonic to use when generating local accounts.
- `chain_id`: The chain id as integer used for `eth_chainId` and the `CHAINID` opcode. If no value is given, defaults to the chain id of the forked network or to 1337 and 1 respectively if no fork is specified.
- `network_id`: The network id as integer used by ganache to identify itself. Defaults to the current timestamp or the network id of the forked chain.
- `evm_version`: The EVM ruleset to use. Default is the most recent available.
- `fork`: If given, the local client will fork from another currently running Ethereum client. The value may be an HTTP location and port of the other client, e.g. `http://localhost:8545`, or the ID of a production network, e.g. `mainnet`. See *Using a Forked Development Network*.
- `block_time`: The time waited between mining blocks. Defaults to instant mining.
- `default_balance`: The starting balance for all unlocked accounts. Can be given as unit string like “1000 ether” or “50 gwei” or as an number **in Ether**. Will default to 100 ether.
- `time`: Date (ISO 8601) that the first block should start. Use this feature, along with *`Chain.sleep`* to test time-dependent code. Defaults to the current time.
- `unlock`: A single address or a list of addresses to unlock. These accounts are added to the *`Accounts`* container and can be used as if the private key is known. Also works in combination with `fork` to send transactions from any account.

Note: These optional commandline fields can also be specified on a project level in the project’s `brownie-config.yaml` file. See the *configuration files*.

Note: `brownie networks list true` shows a detailed view of existing networks, including all configuration fields. This can be useful for defining fields of a new network.

23.1.3 Setting the Default Network

To modify the default network, add the‘‘networks.default‘‘ field to your project configuration file:

```
networks:
    default: ropsten
```

If a configuration file does not exist you will have to create one. See the documentation on *configuration files* for more information.

23.2 Launching and Connecting

23.2.1 Using the CLI

By default, Brownie will launch and connect to `ganache-cli` each time it is loaded. To connect to a different network, use the `--network` flag with the ID of the network you wish to connect to:

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

23.2.2 Using brownie.network

The *brownie.network* module contains methods that allow you to connect or disconnect from any already-defined network.

To connect to a network:

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

To disconnect:

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

23.3 Live Networks

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

> **Warning:** Before you go any further, consider that connecting to a live network 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!

23.3.1 Personal Node vs Hosted Node

To interact with a live network you must connect to a node. You can either run your own node, or connect to a hosted node.

Running your Own Node

Clients such as Geth or Parity can be used to run your own Ethereum node, that Brownie can then connect to. Having your node gives you complete control over which RPC endpoints are available and ensures you have a private and dedicated connection to the network. Unfortunately, keeping a node operating and synced can be a challenging task.

If you wish to learn more about running a node, ethereum.org provides a list of resources that you can use to get started.

Using a Hosted Node

Services such as Infura provide public access to Ethereum nodes. This is a much simpler option than running your own, but it is not without limitations:

1. Some RPC endpoints may be unavailable. In particular, Infura does not provide access to the debug_traceTransaction method. For this reason, Brownie's *debugging tools* will not work when connected via Infura.
2. Hosted nodes do not provide access to accounts - this would be a major security hazard! You will have to manually unlock your own *local account* before you can make a transaction.

Using Infura

To Infura you need to register for an account. Once you have signed up, login and create a new project. You will be provided with a project ID, as well as API URLs that can be leveraged to access the network.

To connect to Infura using Brownie, store your project ID as an environment variable named `WEB3_INFURA_PROJECT_ID`. You can do so with the following command:

```
$ export WEB3_INFURA_PROJECT_ID=YourProjectID
```

23.4 Using a Forked Development Network

Ganache allows you create a development network by forking from an live network. This is useful for testing interactions between your project and other projects that are deployed on the main-net.

Brownie's `mainnet-fork` network uses Infura to create a development network that forks from the main-net. To connect with the console:

```
$ brownie console --network mainnet-fork
```

In this mode, you can use *`Contract.from_explorer`* to fetch sources and interact with contracts on the network you have forked from.

Note: Forking from Infura can be *very slow*. If you are using this mode extensively, it may be useful to run your own Geth node.

CHAPTER 24

Account Management

When connecting to a remote network via a hosted node such as Infura, the *Accounts* container will be empty. Before you can perform any transactions you must add a local account to Brownie.

When we use the term *local* it implies that the account exists locally on your system, as opposed to being available directly in the node. Local accounts are stored in encrypted JSON files known as *keystores*. If you want to learn more about keystore files, you can read If you want to understand the contents of your json file you can read "What is an Ethereum keystore file?" by Julien Maffre.

You can manage your locally available accounts via the commandline:

```
$ brownie accounts
```

24.1 Generating a New Account

To generate a new account using the command line:

```
$ brownie accounts generate <id>
```

You will be asked to choose a password for the account. Brownie will then generate a random private key, and make the account available as `<id>`.

24.2 Importing from a Private Key

To add a new account via private key:

```
$ brownie accounts new <id>
```

You will be asked to input the private key, and to choose a password. The account will then be available as `<id>`.

24.3 Importing from a Keystore

You can import an existing JSON keystore into Brownie using the commandline:

```
$ brownie accounts import <id> <path>
```

Once imported the account is available as `<id>`.

24.4 Exporting a Keystore

To export an existing account as a JSON keystore file:

```
$ brownie accounts export <id> <path>
```

The exported account will be saved at `<path>`.

24.5 Unlocking Accounts

In order to access a local account from a script or console, you must first unlock it. This is done via the *`Accounts.load`* method:

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

Once the account is unlocked it will be available for use within the *`Accounts`* container.

24.6 Unlocking Accounts on Development Networks

On a local or forked development network you can unlock and use any account, even if you don't have the corresponding private key. To do so, add the account to the `unlock` setting in a project's *configuration file*:

```
networks:
    development:
        cmd_settings:
            unlock:
                - 0x7E1E3334130355799F833ffec2D731BCa3E68aF6
                - 0x0063046686E46Dc6F15918b61AE2B121458534a5
```

The unlocked accounts are automatically added to the *`Accounts`* container. Note that you might need to fund the unlocked accounts manually.

CHAPTER 25

The Configuration File

You can modify Brownie's default behaviours by creating an optional configuration file.

The configuration file must be saved as `brownie-config.yaml`. If saved in the root directory of a project it will be loaded whenever that project is active. If saved in your home path, it will always be loaded.

All configuration fields are optional. You can copy from the examples below and modify the settings as required.

25.1 Default Configuration

The following example shows all configuration settings and their default values:

```yaml
project_structure:
    build: build
    contracts: contracts
    interfaces: interfaces
    reports: reports
    scripts: scripts
    tests: tests

networks:
    default: development
    development:
        gas_limit: max
        gas_buffer: 1
        gas_price: 0
        reverting_tx_gas_limit: max
        default_contract_owner: true
        cmd_settings: null
    live:
        gas_limit: auto
        gas_buffer: 1.1
        gas_price: auto
        reverting_tx_gas_limit: false
```

(continues on next page)

(continued from previous page)

```
        default_contract_owner: false

compiler:
    evm_version: null
    solc:
        version: null
        optimizer:
            enabled: true
            runs: 200
        remappings: null
    vyper:
        version: null

console:
    show_colors: true
    color_style: monokai
    auto_suggest: true
    completions: true

reports:
    exclude_paths: null
    exclude_contracts: null

hypothesis:
    deadline: null
    max_examples: 50
    report_multiple_bugs: False
    stateful_step_count: 10
    phases:
        explicit: true
        reuse: true
        generate: true
        target: true
        shrink: true

autofetch_sources: false
dependencies: null
dev_deployment_artifacts: false
```

25.2 Settings

25.2.1 Project Structure

Project subdirectory names. Include these fields if you wish to modify the default structure of your project.

`project_structure.`**`build`**

Project subdirectory that stores data such as compiler artifacts and unit test results.

default value: `build`

`project_structure.`**`contracts`**

Project subdirectory that stores contract source files.

default value: `contracts`

project_structure.**interfaces**

Project subdirectory that stores interface source files and ABIs.

default value: `interfaces`

project_structure.**reports**

Project subdirectory that stores JSON report files.

default value: `reports`

project_structure.**scripts**

Project subdirectory that stores scripts for deployment and interaction.

default value: `scripts`

project_structure.**tests**

Project subdirectory that stores unit tests.

default value: `tests`

25.2.2 Networks

default

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

default value: `development`

networks.**development**

This setting is only available for development networks.

cmd_settings

Additional commandline parameters, which are passed into Ganache as commandline arguments. These settings will update the network specific settings defined in *network management* whenever the project with this configuration file is active.

The following example shows all commandline settings with their default value. `fork` and `unlock` have no default values. `network_id` and `time` will default to the current timestamp or time respectively. See *adding a development network* for more details on the arguments.

```
networks:
    development:
        gas_limit: max
        gas_buffer: 1
        gas_price: 0
        reverting_tx_gas_limit: max
        default_contract_owner: true
        cmd_settings:
            port: 8545
            gas_limit: 6721975
            accounts: 10
            chain_id: 1337
            network_id: 1588949648
            evm_version: istanbul
            fork: null
            mnemonic: brownie
            block_time: 0
            default_balance: 100
            time: 2020-05-08T14:54:08+0000
            unlock: null
```

`networks.`**`live`**

Default settings for development and live environments.

`gas_limit`

The default gas limit for all transactions. If set to `auto` the gas limit is determined using `web3.eth.estimateGas`. If set to `max`, the block gas limit is used.

development default: `max`

live default: `auto`

`gas_buffer`

A modifier applied to `web3.eth.estimateGas` when determining gas price automatically.

development default: `1`

live default: `1.1`

`gas_price`

The default gas price for all transactions. If set to `auto` the gas price is determined using `web3.eth.gasPrice`.

development default: `0`

live default: `auto`

`default_contract_owner`

If `false`, deployed contracts will not remember the account that they were created by. Every transaction will require a `from` kwarg.

`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`*.

development default: `max`

live default: `false`

25.2.3 Compiler

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

`evm_version`

The EVM version to compile for. If `null` the most recent one is used. Possible values are `byzantium`, `constantinople`, `petersburg`, `istanbul`, `atlantis` and `agharta`.

default value: `null`

`compiler.`**`solc`**

Settings specific to the Solidity compiler.

`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`.

default value: `null`

`optimizer`

Optimizer settings to be passed to the Solidity compiler. Values given here are passed into the compiler with no reformatting. See the Solidity documentation for a list of possible values.

`remappings`

Optional field used to supply *path remappings*.

```
remappings:
  - zeppelin=/usr/local/lib/open-zeppelin/contracts/
  - github.com/ethereum/dapp-bin/=/usr/local/lib/dapp-bin/
```

`compiler.`**`vyper`**

Settings specific to the Vyper compiler.

`version`

The version of vyper 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 vyper versions `>=0.1.0-beta.16`.

default value: `null`

25.2.4 Console

`show_colors`

Enable or disable colorful output.

default value: `true`

`color_style`

Set the Pygments color style used within the console and throughout Brownie.

You can view a gallery of popular styles here.

default value: `monokai`

`auto_suggest`

Enable or disable type hints for contract function inputs.

default value: `true`

`completions`

Enable or disable autocompletion.

default value: `true`

25.2.5 Reports

Settings related to reports such as coverage data and gas profiles.

`exclude_paths`

Paths or glob patterns of source files to be excluded from report data.

default value: `null`

```
reports:
    exclude_paths:
        - contracts/mocks/**/*.*
        - contracts/SafeMath.sol
```

`exclude_contracts`

Contract names to be excluded from report data.

default value: `null`

```
reports:
    exclude_contracts:
        - SafeMath
        - Owned
```

25.2.6 Hypothesis

Default settings for *property-based* and *stateful* test execution. See the Hypothesis settings documentation for a complete list of available settings.

```
hypothesis:
    deadline: null
    max_examples: 50
    report_multiple_bugs: False
    stateful_step_count: 10
    deadline: null
    phases:
        explicit: true
        reuse: true
        generate: true
        target: true
        shrink: true
```

25.2.7 Other Settings

`autofetch_sources`

If enabled, Brownie will always attempt to fetch source code for unknown addresses using *`Contract.from_explorer`*.

default value: `false`

`dependencies`

A list of packages that a project depends on. Brownie will attempt to install all listed dependencies prior to compiling the project.

```
dependencies:
    - aragon/aragonOS@4.0.0
    - defi.snakecharmers.eth/compound@1.1.0
```

See the *Brownie Package Manager* to learn more about package dependencies.

`dev_deployment_artifacts`

If enabled, Brownie will save deployment artifacts for contracts deployed on development networks and will include the "dev" network on the deployment map.

This is useful if another application, such as a front end framework, needs access to deployment artifacts while you are on a development network.

default value: `false`

CHAPTER 26

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.

26.1 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': {}, // map of source ids and the path to the related source 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
    'language': "", // source code language (Solidity or Vyper)
    '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
}
```

The build/interfaces folder contains compiler artifacts generated from project interfaces. These files use a similar structure, but only contain some of the fields listed above.

Note: The allSourcePaths field is used to map <SOURCE_ID> references to their actual paths.

26.1.1 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': "<SOURCE_ID>", // id of the related 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
    }
}
```

26.1.2 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": {
        "<SOURCE_ID>": {
            "ContractName.functionName": {
                "index": [start, stop]  // source offsets
            }
        }
    },
    "branches": {
        "<SOURCE_ID>": {
            "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.

26.2 Deployment Artifacts

Each time a contract is deployed to a network where *persistence* is enabled, Brownie saves a copy of the *compiler artifact* 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.

26.3 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)
                "<SOURCE_ID>": [
                    [], // statements
                    [], // branches that did not jump
                    []  // branches that did jump
                ]
            }
        }
    }
}
```

26.3.1 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)
            "<SOURCE_ID>": [
                [1, 3],
                [],
                [5]
            ],
            "<SOURCE_ID>": [
                [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:

```
{
    "<SOURCE_ID>": {
        "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`.

26.4 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 27

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')
```

27.1 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'>
```

27.2 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()
```

27.3 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 28

Brownie API

This section provides a complete overview of the Brownie API. It includes all public classes and methods as well as limited internal documentation.

If you have not yet viewed the documentation under "Core Functionality" within the table of contents, you may wish to start there before exploring the API docs.

Hint: From the console 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 any class or method to view information on it's functionality.

28.1 Brownie API

28.1.1 brownie

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

```
>>> from brownie import *
>>> dir()
['Contract', 'Fixed', 'Wei', 'accounts', 'alert', 'compile_source', 'config', 'history
↪', 'network', 'project', 'rpc', 'run', 'web3']
```

28.1.2 brownie.exceptions

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

Exceptions

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* object, when one already exists for the given address.

exception brownie.exceptions.**ContractNotFound**
Raised when attempting to access a *Contract* 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.

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

exception brownie.exceptions.**ProjectNotFound**
Raised by *project.load* 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.**UnsupportedLanguage**
Raised when attempting to compile a language that Brownie does not support.

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.

Warnings

exception brownie.exceptions.**BrownieCompilerWarning**

Raised by *Contract.from_explorer* when a contract cannot be compiled, or compiles successfully but produces unexpected bytecode.

exception brownie.exceptions.**BrownieEnvironmentWarning**

Raised on unexpected environment conditions.

exception brownie.exceptions.**InvalidArgumentWarning**

Raised on non-critical, invalid arguments passed to a method, function or config file.

28.1.3 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 "<console>", line 1, in <module>
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.

28.1.4 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.

28.2 Convert API

The convert package contains methods and classes for representing and converting data.

28.2.1 brownie.convert.main

The main module contains methods for data conversion. Methods within this module can all be imported directly from the convert package.

brownie.convert.**to_uint**(*value*, *type_str="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_str="int256"*)

Converts a value to a signed integer. This is equivalent to calling *Wei* and then applying checks for over/underflows.

brownie.convert.**to_decimal**(*value*)

Converts a value to a decimal fixed point and applies bounds according to Vyper's decimal type.

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_str="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_str.

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

```
>>> from brownie.convert import to_bytes
>>> 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.

28.2.2 brownie.convert.datatypes

The datatypes module contains subclasses that Brownie uses to assist with conversion and comparison.

EthAddress

class brownie.convert.datatypes.**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 "<console>", line 1, in <module>
TypeError: Invalid type for comparison: '0x35424F91fD33084466F402d5D97f05F8e3b4AF
↪' is not a valid address

>>> e == "potato"
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: Invalid type for comparison: 'potato' is not a valid address

>>> type(e)
<class 'brownie.convert.EthAddress'>
```

Fixed

class brownie.convert.datatypes.**Fixed**(*value*)

decimal.Decimal subclass that allows comparisons, addition and subtraction against strings, integers and *Wei*.

Fixed is used for inputs and outputs to Vyper contracts that use the decimal type.

Attempting comparisons or arithmetic against a float raises a TypeError.

```
>>> from brownie import Fixed
>>> Fixed(1)
Fixed('1')
>>> Fixed(3.1337)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: Cannot convert float to decimal - use a string instead

>>> Fixed("3.1337")
Fixed('3.1337')
```

(continues on next page)

(continued from previous page)

```
>>> Fixed("12.49 gwei")
Fixed('12490000000')
>>> Fixed("-1.23") == -1.2300
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: Cannot compare to floating point - use a string instead

>>> Fixed("-1.23") == "-1.2300"
True
```

HexString

class `brownie.convert.datatypes.`**`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"
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: Invalid type for comparison: 'potato' is not a valid hex string
```

ReturnValue

class `brownie.convert.datatypes.`**`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.

Wei

class `brownie.convert.datatypes.`**Wei**(*value*)

Integer subclass that converts a value to wei (the smallest unit of Ether, equivalent to 10^{-18} Ether) 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")
29000000
>>> Wei(8.38e32)
838000000000000000000000000000000
>>> Wei(1e18) == "1 ether"
True
>>> Wei("1 ether") < "2 ether"
True
>>> Wei("1 ether") - "0.75 ether"
250000000000000000
```

classmethod `Wei.`**to**(*unit*)

Returns a `Fixed` number converted to the specified unit.

Attempting a conversion to an unknown unit raises a `TypeError`.

```
>>> from brownie import Wei
>>> Wei("20 gwei").to("ether")
Fixed('2.0000000000E-8')
```

28.2.3 `brownie.convert.normalize`

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

`normalize.`**`format_input`**`(`*abi*, *inputs*`)`

Formats inputs based on a contract method ABI.

- `abi`: A contract method ABI as a dict.
- `inputs`: List or tuple of values to format. Each value is converted using one of the methods outlined in *brownie.convert.main*.

Returns a list of values formatted for use by `ContractTx` or `ContractCall`.

```
>>> from brownie.convert.normalize 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)
```

`normalize.`**`format_output`**`(`*abi*, *outputs*`)`

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

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

Returns a *`ReturnValue`* container where each value has been formatted using the one of the methods outlined in *brownie.convert.main*.

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

```
>>> from brownie.convert.normalize 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',)
```

`normalize.`**`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.

28.2.4 `brownie.convert.utils`

The `utils` module contains helper methods used by other methods within the `convert` package.

`utils.`**`get_int_bounds`**`(`*type_str*`)`

Given an integer type string, returns the lower and upper bound for that data type.

`utils.`**`get_type_strings`**`(`*abi_params*, *substitutions*`)`
Converts a list of parameters from an ABI into a list of type strings.

28.3 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.

28.3.1 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 = None*, *launch_rpc = True*`)`
Connects to a network.

- `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.

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 = True*`)`
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`**`()`
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*`)`
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 auto, 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("auto")
False
```

main.**gas_buffer**(**args*)

Gets and optionally sets the default gas buffer.

- If no argument is given, the current default is displayed.
- If an integer or float value is given, this will be the default gas buffer.
- If None is given, the gas buffer is set to 1 (disabled).

```
>>> from brownie import network
>>> network.gas_buffer()
1.1
>>> network.gas_buffer(1.25)
1.25
>>> network.gas_buffer(None)
1
```

main.**gas_price**(**args*)

Gets and optionally sets the default gas price.

- If an integer value is given, this will be the default gas price.
- If set to auto, the gas price is determined automatically via web3.eth.gasPrice.

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("auto")
False
```

28.3.2 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 Attributes

Accounts.**default**

Default account that is used for deploying contracts. Initially set to `None`.

Note that the default account used to send contract transactions is the one that deployed the contract, not `accounts.default`.

```
>>> accounts.default = accounts[1]
```

Accounts Methods

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

Creates a new *LocalAccount* with private key `private_key`, appends it to the container, and returns the new account instance.

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

When no private key is given a new one is randomly generated. A seed phrase for the account is also printed to the console.

```
>>> accounts.add()
mnemonic: 'buffalo cinnamon glory chalk require inform strike ginger crop sell␣
↪hidden cart'
<LocalAccount '0xf293C5E0b22802Bf5DCef3FB8112EaA4cA54fcCF'>
```

classmethod Accounts.**at**(*address, force=False*)

Given an address as a string, returns the corresponding *Account* or *LocalAccount* from the container. If `force=True`, returns and adds the account even if it is not found in the container. Use this if an account is unlocked by external means.

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

classmethod Accounts.**clear**()

Empties the container.

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

classmethod Accounts.**from_mnemonic**(*mnemonic*, *count=1*, *offset=0*)

Generates one or more *LocalAccount* objects from a seed phrase.

- `mnemonic` : Space-separated list of BIP39 mnemonic seed words
- `count` : The number of *LocalAccount* objects to create
- `offset` : The initial account index to create accounts from

If `count` is greater than 1, a list of *LocalAccount* objects are returned.

```
>>> a.from_mnemonic('buffalo cinnamon glory chalk require inform strike ginger␣
↪crop sell hidden cart')
<LocalAccount '0xf293C5E0b22802Bf5DCef3FB8112EaA4cA54fcCF'>
```

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 *state._notify_registry* when the local chain has been reset. All *Account* objects are recreated.

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

Called by *state._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.`**`gas_used`**

The cumulative gas amount paid for transactions from this account.

```
>>> accounts[0].gas_used
21000
```

`Account.`**`nonce`**

The current nonce of the address.

```
>>> accounts[0].nonce
1
```

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*, *nonce=None*, *required_confs=1*, *allow_revert=False*, *silent=False*, *publish_source=False*`)`

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 `web3.eth.estimateGas`.
- `gas_buffer`: A multiplier applied to `web3.eth.estimateGas` when setting gas limit automatically. `gas_limit` and `gas_buffer` cannot be given at the same time.
- `gas_price`: Gas price for the transaction. The given value is converted to `Wei`. If none is given, the price is set using `web3.eth.gasPrice`.

- `nonce`: Nonce for the transaction. If none is given, the nonce is set using `web3.eth.getTransactionCount` while also considering any pending transactions of the Account.
- `required_confs`: The required *`confirmations`* before the *`TransactionReceipt`* is processed. If none is given, defaults to 1 confirmation. If 0 is given, immediately returns a pending *`TransactionReceipt`* instead of a *`Contract`* instance, while waiting for a confirmation in a separate thread.
- `allow_revert`: When `True`, forces the deployment of a contract, even if a revert reason is detected.
- `silent`: When `True`, suppresses any console output for the deployment.
- `publish_source`: When `True`, attempts to verify the source code on etherscan.io.

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=None*, *amount=0*, *gas_price=None*, *data=""*)

Estimates the gas required to perform a transaction. Raises a func:*VirtualMachineError <brownie.exceptions.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`*.
- `gas_price`: Gas price of the transaction.
- `data`: Transaction data hexstring.

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

classmethod `Account.`**`get_deployment_address`**(*nonce=None*)

Return the address where a contract will be deployed from this account, if the deployment transaction uses the given nonce.

If nonce is *None*, the nonce of the next transaction is used.

```
>>> accounts[0].get_deployment_address()
'0xd495633B90a237de510B4375c442C0469D3C161C'
```

classmethod Account.**transfer**(*self*, *to=None*, *amount=0*, *gas_limit=None*, *gas_price=None*, *data=None*, *nonce=None*, *required_confs=1*, *allow_revert=None*, *silent=False*)

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 `web3.eth.estimateGas`.
- `gas_buffer`: A multiplier applied to `web3.eth.estimateGas` when setting gas limit automatically. `gas_limit` and `gas_buffer` cannot be given at the same time.
- `gas_price`: Gas price for the transaction. The given value is converted to *Wei*. If none is given, the price is set using `web3.eth.gasPrice`.
- `data`: Transaction data hexstring.
- `nonce`: Nonce for the transaction. If none is given, the nonce is set using `web3.eth.getTransactionCount` while also considering any pending transactions of the Account.
- `required_confs`: The required *confirmations* before the *TransactionReceipt* is processed. If none is given, defaults to 1 confirmation. If 0 is given, immediately returns a pending *TransactionReceipt*, while waiting for a confirmation in a separate thread.
- `allow_revert`: Boolean indicating whether the transaction should be broadacsted when it is expected to revert. If not set, the default behaviour is to allow reverting transactions in development and disallow them in a live environment.
- `silent`: Toggles console verbosity. If `True` is given, suppresses all console output for this transaction.

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'>
```

You can also deploy contracts by omitting the `to` field. Note that deploying with this method does not automatically create a *Contract* object.

```
>>> deployment_bytecode = "0x6103f056600035601c52740100..."
>>> accounts[0].transer(data=deployment_bytecode)
Transaction sent:␣
↪0x2b33315f7f9ec86d27112ea6dffb69b6eea1e582d4b6352245c0ac8e614fe06f
  Gas price: 0.0 gwei   Gas limit: 6721975
  Transaction confirmed - Block: 1   Gas used: 268460 (3.99%)
  UnknownContract deployed at: 0x3194cBDC3dbcd3E11a07892e7bA5c3394048Cc87
<Transaction '0x2b33315f7f9ec86d27112ea6dffb69b6eea1e582d4b6352245c0ac8e614fe06f'>
```

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 *Account* 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
```

28.3.3 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()
```

(continues on next page)

(continued from previous page)

```
[<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()
[]
```

28.3.4 `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 the documentation on *persistence* 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

↪'608060405234801561001057600080fd5b506040516107873803806107878339810160409081528151602080840151
↪...
```

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 `dict` 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*, *publish_source=False*)

Deploys the contract.

- `*args`: Contract constructor arguments.
- `publish_source`: When `True`, attempts to verify the source code on etherscan.io.

You can optionally include a `dict` 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 <brownie.exceptions.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 new `Contract` or `ProjectContract` object. The object is also appended to the container.

- `address`: Address where the contract is deployed.
- `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.

This method compares the bytecode at the given address with the deployment bytecode for the given `ContractContainer`. A `ProjectContract` is returned if the bytecodes match, a `Contract` otherwise.

Raises `ContractNotFound` if there is no code at the given address.

```
>>> 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_)
```

(continues on next page)

(continued from previous page)

```
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.**publish_source**(*contract*, *silent=False*)

Verifies the source code on etherscan.io for a `Project Contract` belonging to the container.

- `contract`: The `Project Contract` you intend to verify
- `silent`: When True, suppresses all console output of the call.

classmethod ContractContainer.**decode_input**(*calldata*)

Given the call data of a transaction, returns the function signature as a string and the decoded input arguments.

Raises `ValueError` if the call data cannot be decoded.

```
>>> Token.decode_input(
↪'0xa9059cbb00000000000000000000000009dc9431cccd2c73f0a2f68dc69a4a527ab5d809000000000000000000
↪')
("transfer(address,uint256)", ['0x9DC9431CcCCD2C73F0a2F68Dc69A4a527aB5d809',␣
↪10000])
```

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
0xa9059cbb00000000000000000000000066ace0365c25329a407002d22908e25adeacb9bb0000000000000000000000
>>> 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 *`state._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 *`state._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. They are used for interaction with already-deployed 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 greater debugging capability.

These classes have identical APIs.

class `brownie.network.contract.`**`Contract`**(*address_or_alias*, *owner=None*)

A deployed contract that is not part of a Brownie project.

- `address_or_alias`: Address of the contract.
- `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 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'>
>>> dir(Token[0])
[abi, allowance, approve, balance, balanceOf, bytecode, decimals, name,␣
↪signatures, symbol, topics, totalSupply, transfer, transferFrom, tx]
```

Contract Classmethods

New `Contract` objects are created with one of the following class methods.

classmethod `Contract.`**`from_abi`**(*name*, *address*, *abi*, *owner=None*)

Create a new `Contract` object from an address and an ABI.

- `name`: The name of the contract.
- `address`: Address of the contract.
- `abi`: ABI of the contract. Required unless a `manifest_uri` is given.
- `owner`: An optional `Account` instance. If given, transactions to the contract are sent broadcasted from this account by default.

Creating a `Contract` from an ABI will allow you to call or send transactions to the contract, but functionality such as debugging will not be available.

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

classmethod `Contract.`**`from_ethpm`**(*name*, *manifest_uri*, *address=None*, *owner=None*)

Create a new `Contract` object from an ethPM manifest.

- name: The name of the contract. Must be present within the manifest.
- manifest_uri: EthPM registry manifest uri.
- address: Address of the contract. Only Required if more than one deployment named name is included in the manifest.
- owner: An optional *Account* instance. If given, transactions to the contract are sent broadcasted from this account by default.

```
>>> from brownie import network, Contract
>>> network.connect('mainnet')
>>> Contract("DSToken", manifest_uri="ethpm://erc20.snakecharmers.eth:1/dai-dai@1.
↪0.0")
<DSToken Contract object '0x89d24A6b4CcB1B6fAA2625fE562bDD9a23260359'>
```

classmethod Contract.**from_explorer**(*address*, *as_proxy_for=None*, *owner=None*)

Create a new Contract object from source code fetched from a block explorer such as EtherScan or Blockscout.

- address: Address of the contract.
- as_proxy_for: Address of the implementation contract, if address is a proxy contract. The generated object sends transactions to address, but uses the ABI and NatSpec of as_proxy_for. This field is only required when the block explorer API does not provide an implementation address.
- owner: An optional *Account* instance. If given, transactions to the contract are sent broadcasted from this account by default.

If the deployed bytecode was generated using a compatible compiler version, Brownie will attempt to recompile it locally. If successful, most debugging functionality will be available.

```
>>> Contract.from_explorer("0x6b175474e89094c44da98b954eedeac495271d0f")
Fetching source of 0x6B175474E89094C44Da98b954EedeAC495271d0F from api.etherscan.
↪io...
<Dai Contract '0x6B175474E89094C44Da98b954EedeAC495271d0F'>
```

Contract Attributes

Contract.**alias**

User-defined alias applied to this Contract object. Can be used to quickly restore the object in future sessions.

```
>>> Token.alias
'mytoken'
```

Contract.**bytecode**

The bytecode of the deployed contract, including constructor arguments.

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

↪'6080604052600436106100985763ffffffff7c0100000000000000000000000000000000000000000000000000000
↪..
```

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
```

classmethod `Contract.`**`set_alias`**`(`*alias*`)`

Apply a unique alias this object. The alias can be used to restore the object in future sessions.

- `alias`: An alias to apply, given as a string. If `None`, any existing alias is removed.

Raises `ValueError` if the given alias is invalid or already in use on another contract.

```
>>> Token.set_alias('mytoken')

>>> Token.alias
'mytoken'
```

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 *`state._notify_registry`*.

ContractCall

class `brownie.network.contract.`**`ContractCall`**`(`**args*, *block_identifier=None*`)`

Calls a non state-changing contract method without broadcasting a transaction, and returns the result. `args` must match the required inputs for the method.

- `args`: Input arguments for the call. The expected inputs are shown in the method's `__repr__` value.
- `block_identifier`: A block number or hash that the call is executed at. If `None`, the latest block is used. Raises *ValueError* if this value is too far in the past and you are not using an archival node.

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)'>
>>> 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.**info**()

Display NatSpec documentation documentation for the given method.

```
>>> Token[0].allowance.info()
allowance(address _owner, address _spender)
  @dev Function to check the amount of tokens than an owner
       allowed to a spender.
  @param _owner address The address which owns the funds.
  @param _spender address The address which will spend the funds.
  @return A uint specifying the amount of tokens still available
          for the spender.
```

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 `dict` 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*, *block_identifier=None*`)`

Calls the contract method without broadcasting a transaction, and returns the result.

- `args`: Input arguments for the call. The expected inputs are shown in the method's `__repr__` value.
- `block_identifier`: A block number or hash that the call is executed at. If `None`, the latest block is used. Raises *ValueError* if this value is too far in the past and you are not using an archival node.

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.`**`decode_input`**`(`*hexstr*`)`

Decodes hexstring input data for this method.

```
>>>  Token[0].transfer.decode_input(
↪"0xa9059cbb000000000000000000000000d36bdba474b5b442310a5bfb989903020249bba0000000000000000000
↪")
("0xd36bdba474b5b442310a5bfb989903020249bba", 1000)
```

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

Decodes raw hexstring data returned by this method.

```
>>>  Token[0].balanceOf.decode_output(
↪"0x00000000000000000000000000000000000000000000003635c9adc5dea00000")
1000000000000000000000
```

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)
0xa9059cbb000000000000000000000000d36bdba474b5b442310a5bfb989903020249bba0000000000000000000000
>>> accounts[0].transfer(Token[0], 0, data=calldata)

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

classmethod ContractTx.**info**()

Display NatSpec documentation documentation for the given method.

```
>>> Token[0].transfer.info()
transfer(address _to, uint256 _value)
  @dev transfer token for a specified address
  @param _to The address to transfer to.
  @param _value The amount to be transferred.
```

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)'>
```

When a contract only contains one method with the given name and number of arguements, OverloadedMethod may be called directly. When more than one method is present, a ValueError is raised.

```
>>> erc223.transfer(accounts[0], "1 ether")
Transaction sent:␣
↪0x8dbf15878104571669f9843c18afc40529305ddb842f94522094454dcde22186
ERC223.transfer confirmed - block: 2   gas used: 50985 (100.00%)
<Transaction object
↪'0x8dbf15878104571669f9843c18afc40529305ddb842f94522094454dcde22186'>
```

InterfaceContainer

class brownie.network.contract.**InterfaceContainer**

Container class that provides access to interfaces within a project.

This object is created and populated with *InterfaceConstructor* objects when a Brownie project is opened. It is available as `interface` within the console and as a pytest fixture.

```
>>> interface
<brownie.network.contract.InterfaceContainer object at 0x7fa239bf0d30>
```

InterfaceConstructor

class brownie.network.contract.**InterfaceConstructor**(*address*, *owner=None*)

Constructor to create *Contract* objects from a project interface.

- `address_or_alias`: Address of the deployed contract.
- `owner`: An optional *Account* instance. If given, transactions to the contract are sent broadcasted from this account by default.

When a project is loaded, an `InterfaceConstructor` is generated from each interface file within the `interfaces/` folder of the project. These objects are stored as *InterfaceContainer* members.

```
>>> interface.Dai
<InterfaceConstructor 'Dai'>

>>> interface.Dai("0x6B175474E89094C44Da98b954EedeAC495271d0F")
<Dai Contract object '0x6B175474E89094C44Da98b954EedeAC495271d0F'>
```

InterfaceConstructor Attributes

InterfaceConstructor.**abi**

The interface ABI as a `dict`.

28.3.5 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.network.event.**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, an *_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.network.event.**_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
```

(continues on next page)

(continued from previous page)

```
}
>>> 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.**address**

The address where the event was fired. If the object contains multiple events, this value is set to `None`.

```
>>> tx.events[2].address
"0x2d72c1598537bcf4a4af97668b3a24e68b7d0cc5"
```

_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 dictiorary 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': {
        '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
{}
```

28.3.6 `brownie.network.gas`

The `gas` module contains gas strategy classes, as well as abstract base classes for buiding your own gas strategies.

Gas Strategies

class brownie.network.gas.strategies.**ExponentialScalingStrategy**(*initial_gas_price*, *max_gas_price*, *time_duration=30*)

Time based scaling strategy for exponential gas price increase.

The gas price for each subsequent transaction is calculated as the previous price multiplied by *1.1 ** n* where n is the number of transactions that have been broadcast. In this way the price increase starts gradually and ramps up until confirmation.

- `initial_gas_price`: The initial gas price to use in the first transaction
- `max_gas_price`: The maximum gas price to use
- `time_duration`: Number of seconds between transactions

```
>>> from brownie.network.gas.strategies import ExponentialScalingStrategy
>>> gas_strategy = ExponentialScalingStrategy("10 gwei", "50 gwei")

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

class brownie.network.gas.strategies.**GasNowStrategy**(*speed="fast"*)

Gas strategy for determing a price using the GasNow API.

- `speed`: The gas price to use based on the API call. Options are rapid, fast, standard and slow.

```
>>> from brownie.network.gas.strategies import GasNowStrategy
>>> gas_strategy = GasNowStrategy("fast")

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

class brownie.network.gas.strategies.**GasNowScalingStrategy**(*initial_speed="standard", max_speed="rapid", increment=1.125, block_duration=2*)

Block based scaling gas strategy using the GasNow API.

- `initial_speed`: The initial gas price to use when broadcasting the first transaction. Options are rapid, fast, standard and slow.
- `max_speed`: The maximum gas price to use when replacing the transaction. Options are rapid, fast, standard and slow.
- `increment`: A multiplier applied to the most recently used gas price in order to determine the new gas price. If the incremented value is less than or equal to the current `max_speed` rate, a new transaction is broadcasted. If the current rate for `initial_speed` is greater than the incremented rate, it is used instead.
- `block_duration`: The number of blocks to wait between broadcasting new transactions.

```
>>> from brownie.network.gas.strategies import GasNowScalingStrategy
>>> gas_strategy = GasNowScalingStrategy("standard", increment=1.125, block_
↪duration=2)

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

class brownie.network.gas.strategies.**GethMempoolStrategy**(*position=500, graphql_endpoint=None, block_duration=2*)

Block based scaling gas strategy using Geth's GraphQL interface.

In order to use this strategy you must be connecting via a Geth node with GraphQL enabled.

The yielded gas price is determined by sorting transactions in the mempool according to gas price, and returning the price of the transaction at *position*. This is the same technique used by the GasNow API.

- A position of 200 or less usually places a transaction within the mining block.
- A position of 500 usually places a transaction within the 2nd pending block.

```
>>> from brownie.network.gas.strategies import GethMempoolStrategy
>>> gas_strategy = GethMempoolStrategy(200)

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

class brownie.network.gas.strategies.**LinearScalingStrategy**(*initial_gas_price*, *max_gas_price*, *increment=1.125*, *time_duration=30*)

Time based scaling strategy for linear gas price increase.

- `initial_gas_price`: The initial gas price to use in the first transaction
- `max_gas_price`: The maximum gas price to use
- `increment`: Multiplier applied to the previous gas price in order to determine the new gas price
- `time_duration`: Number of seconds between transactions

```
>>> from brownie.network.gas.strategies import LinearScalingStrategy
>>> gas_strategy = LinearScalingStrategy("10 gwei", "50 gwei", 1.1)

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

Gas Strategy ABCs

Abstract base classes for building your own gas strategies.

Simple Strategies

class brownie.network.gas.bases.**SimpleGasStrategy**

Abstract base class for simple gas strategies.

Simple gas strategies are called once to provide a dynamically genreated gas price at the time a transaction is broadcasted. Transactions using simple gas strategies are not automatically rebroadcasted.

Simple Strategy Abstract Methods

To implement a simple gas strategy, subclass *SimpleGasStrategy* and include the following method:

SimpleGasStrategy.**get_gas_price**(*self*) → int:

Return the gas price for a transaction.

Scaling Strategies

class brownie.network.gas.bases.**BlockGasStrategy**(*duration=2*)

Abstract base class for block-based gas strategies.

Block gas strategies are called every `duration` blocks and can be used to automatically rebroadcast a pending transaction with a higher gas price.

class brownie.network.gas.bases.**TimeGasStrategy**(*duration=30*)

Abstract base class for time-based gas strategies.

Time gas strategies are called every `duration` seconds and can be used to automatically rebroadcast a pending transaction with a higher gas price.

Scaling Strategy Abstract Methods

To implement a scaling strategy, subclass one of the above ABCs and implement the following generator function:

`BlockGasStrategy.`**`get_gas_price`**`(`*self*`)` → Generator[int]:

Generator function that yields a new gas price each time it is called.

The produced generator is called every `duration` seconds while a transaction is still pending. Each call must yield a new gas price as an integer. If the newly yielded value is at least 10% higher than the current gas price, the transaction is rebroadcasted with the new gas price.

28.3.7 `brownie.network.state`

The `state` module contains classes to record transactions and contracts as they occur on the blockchain.

Classes in `state` are not meant to be instantiated directly. *`TxHistory`* and *`Chain`* objects are available as `history` and `chain` in the console and as pytest fixtures.

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,
        'high': 742912,
        'low': 742912
    },
    'Token.transfer': {
        'avg': 43535,
        'count': 2,
        'high': 51035,
        'low': 36035
```

(continues on next page)

(continued from previous page)

```
    }
}
```

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.**filter**(*key=None*, ***kwargs*)

Return a filtered list of transactions.

Each keyword argument corresponds to a `TransactionReceipt` attribute. Only transactions where every attributes matches the given value are returned.

```
>>> history.filter(sender=accounts[0], value="1 ether")
[<Transaction object
↪'0xe803698b0ade1598c594b2c73ad6a656560a4a4292cc7211b53ffda4a1dbfbe8'>]
```

You can also use `key` to prodive a function or lambda. It should receive one argument, a `TransactionReceipt`, and return a boolean indicating if the object is to be included in the result.

```
>>> history.filter(key=lambda k: k.nonce < 2)
[<Transaction '0x03569ee152b04ba5b55c2bf05f99f7ec153db715acfe0c1600f144ded58f31fe
↪'>, <Transaction
↪'0x42193c0ff7007c6e2a5e5572a3c6b5706cd133d21e30e5826add3d971134504c'>]
```

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'>]
```

classmethod `TxHistory.`**`wait`**`(`*key=None*, ***kwargs*`)`

Wait for pending transactions to confirm.

This method iterates over a list of transactions generated by *`TxHistory.filter`*, waiting until each transaction has confirmed. If no arguments are given, all transactions within the container are used.

TxHistory Internal Methods

classmethod `TxHistory.`**`_reset`**`()`

Called by *`state._notify_registry`* when the local chain has been reset. All *`TransactionReceipt`* objects are removed from the container.

classmethod `TxHistory.`**`_revert`**`(`*height*`)`

Called by *`state._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.

Chain

class `brownie.network.state.`**`Chain`**

List-like *`Singleton`* used to access chain information and perform actions such as snapshotting, rewinds and time travel.

```
>>> from brownie.network.state import Chain
>>> chain = Chain()
>>> chain
<Chain object (chainid=1, height=10451202)>
```

You can use list indexing the access specific blocks. For negative index values, the block returned is relative to the most recently mined block. For example, `chain[-1]` returns the most recently mined block.

```
>>> web3.eth.blockNumber
10451202

>>> len(chain)
10451203  # always +1 to the current block number, because the first block is zero

>>> chain[0] == web3.eth.getBlock(0)
True

>>> chain[-1] == web3.eth.getBlock('latest')
True
```

Chain Attributes

`Chain.`**`height`**

The current block height.

```
>>> chain.height
10451202
```

Chain.**id**

The chain ID value for the active network. Returns `None` if no chain ID is available.

```
>>> chain.id
1
```

Chain Methods

Chain.**get_transaction**(*txid*)

Return a *TransactionReceipt* object for the given transaction hash.

This function is non-blocking. Pending transaction return immediately.

Raises `TransactionNotFound` if the transaction does not exist.

```
>>> chain.get_
↪transaction(0xf598d43ef34a48478f3bb0ad969c6735f416902c4eb1eb18ebebe0fca786105e)
<Transaction '0xf598d43ef34a48478f3bb0ad969c6735f416902c4eb1eb18ebebe0fca786105e'>
```

Chain.**new_blocks**(*height_buffer*, *poll_interval*)

Generator for iterating over new blocks.

`height_buffer`: The number of blocks behind "latest" to return. A higher value means more delayed results but less likelihood of uncles. `poll_interval`: Maximum interval between querying for a new block, if the height has not changed. Set this lower to detect uncles more frequently.

```
count = 0
for block in chain.new_blocks():
    print(block.number)
    count += 1
    if count == 5:
        break
```

Chain.**time**()

Return the current epoch time in the RPC as an integer.

```
>>> chain.time()
1550189043
```

Chain.**sleep**(*seconds*)

Advance the RPC time. You can only advance the time by whole seconds.

```
>>> chain.time()
1550189043
>>> chain.sleep(100)
>>> chain.time()
1550189143
```

Chain.**mine**(*blocks=1*, *timestamp=None*, *timedelta=None*)

Mine one or more empty blocks.

- `blocks`: Number of blocks to mine
- `timestamp`: Timestamp of the final block being mined. If multiple blocks are mined, they will be mined at equal intervals starting from *chain.time* and ending at `timestamp`.
- `timedelta`: Timedelta for the final block to be mined. If given, the final block will have a timestamp of `chain.time() + timedelta`.

Returns the block height after all new blocks have been mined.

```
>>> web3.eth.blockNumber
0
>>> chain.mine()
1
>>> chain.mine(3)
4
```

Chain.**snapshot**()

Create a snapshot at the current block height.

```
>>> chain.snapshot()
```

Chain.**revert**()

Revert the blockchain to the latest snapshot. Raises `ValueError` if no snapshot has been taken.

```
>>> chain.snapshot()
>>> 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
>>> chain.revert()
4
>>> accounts[0].balance()
100000000000000000000
```

Chain.**reset**()

Reset the local environment to the initial state when Brownie was loaded. This action is performed using a snapshot - it is NOT equivalent to calling *`rpc.kill`* and then *`rpc.launch`*.

Returns the block height after resetting.

```
>>> chain.reset()
0
```

Chain.**undo**(*num=1*)

Undo one or more recent transactions.

- `num`: Number of transactions to undo

Once undone, a transaction can be repeated using *`Chain.redo`*. Calling *`Chain.snapshot`* or *`Chain.revert`* clears the undo buffer.

Returns the block height after all undo actions are complete.

```
>>> web3.eth.blockNumber
3
>>> chain.undo()
2
```

Chain.**redo**(*num=1*)

Redo one or more recently undone transactions.

- `num`: Number of transactions to redo

Returns the block height after all redo actions are complete.

```
>>> web3.eth.blockNumber
2
>>> chain.redo()
Transaction sent:␣
↪0x8c166b66b356ad7f5c58337973b89950f03105cdae896ac66f16cdd4fc395d05
  Gas price: 0.0 gwei   Gas limit: 6721975
  Transaction confirmed - Block: 3   Gas used: 21000 (0.31%)

3
```

Internal Methods

The internal methods in the `state` module are used for tracking and adjusting the contents of various container objects when the local RPC network is reverted or reset.

`brownie.network.state.`**`_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.

`brownie.network.state.`**`_notify_registry`**`(`*height*`)`

Calls each registered object's `_revert` or `_reset` method after the local state has been reverted.

`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 function 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.

28.3.8 `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 *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.**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()
'istanbul'
```

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
```

28.3.9 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]
```

TransactionReceipt Attributes

TransactionReceipt.**block_number**

The block height at which the transaction confirmed.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
```

(continues on next page)

(continued from previous page)

```
>>> tx.block_number
2
```

TransactionReceipt.**confirmations**

The number of blocks mined since the transaction was confirmed, including the block the transaction was mined in: `block_height - tx.block_number + 1`.

```
>>> tx
<Transaction '0x8c166b66b356ad7f5c58337973b89950f03105cdae896ac66f16cdd4fc395d05'>
>>> tx.confirmations
11
```

TransactionReceipt.**contract_address**

The address of the contract deployed in this transaction, if the transaction was a deployment.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.contract_address
None
```

For contracts deployed as the result of calling another contract, see *`TransactionReceipt.new_contracts`*.

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.**dev_revert_msg**

The *developer revert comment* returned when a transaction causes the EVM to revert, if any.

```
>>> tx
<Transaction object
↪'0xd9e0fb1bd6532f6aec972fc8aef806a8d8b894349cf5c82c487335625db8d0ef'>
>>> tx.dev_revert_msg
'dev: is four'
```

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': {
```

(continues on next page)

(continued from previous page)

```
        '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

↪'0xa9059cbb00000000000000000000000031d504908351d2d87f3d6111f491f0b52757b5920000000000000000000
↪'
```

TransactionReceipt.**internal_transfers**

A list of all internal ether transfers that occurred during the transaction. Transfers are sequenced in the order they took place, and represented as dictionaries containing the following fields:

- `from`: Sender address
- `to`: Receiver address
- `value`: Amount of ether that was transferred in *Wei*

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.internal_transfers
[
    {
        "from": "0x79447c97b6543F6eFBC91613C655977806CB18b0",
        "to": "0x21b42413bA931038f35e7A5224FaDb065d297Ba3",
        "value": 100
    }
]
```

`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
↪':
↪'0x0000000000000000000000006b5132740b834674c3277aafa2c27898cbe740f60000000000000000000000031d
↪', 'topics': [HexBytes(
↪'0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef')], 'type':
↪'mined'})]
```

`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.`**`new_contracts`**

A list of new contract addresses that were deployed during this transaction, as the result of contract call.

```
>>> tx = Deployer.deploy_new_contract()
Transaction sent:␣
↪0x6c3183e41670101c4ab5d732bfe385844815f67ae26d251c3bd175a28604da92
  Gas price: 0.0 gwei   Gas limit: 79781
  Deployer.deploy_new_contract confirmed - Block: 4   Gas used: 79489 (99.63%)

>>> tx.new_contracts
["0x1262567B3e2e03f918875370636dE250f01C528c"]
```

`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.**subcalls**

A list of dictionaries providing information about subcalls that occured during the transaction.

The following fields are always included:

- `from`: Address where the call originated
- `to`: Address being called
- `op`: Instruction used to make the call

The following fields are included when the source code for `to` is known:

- `function`: Signature of the function being called
- `inputs`: Dictionary of decoded input arguments in the call

One of the following fields is included, depending on how the call ends:

- `return_value`: A tuple of decoded return values, if the call ended with `RETURN`
- `revert_msg`: The given error message, if the call ended in a `REVERT` or `INVALID` instruction
- `selfdestruct`: Set to `True` if the call ended in a `SELFDESTRUCT` instruction

```
>>> history[-1].subcalls
[
    {
        'from': "0x5AE569698C5F986665018B6e1d92A71be71DEF9a",
        'function': "get_period_timestamp(int128)",
        'inputs': {
            'p': 0
        },
        'op': "STATICCALL",
        'return_value': (1594574319,),
        'to': "0x0C41Fc429cC21BC3c826efB3963929AEdf1DBb8e"
    },
...
```

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'>
>>> tx.sender
<Account object '0x6B5132740b834674C3277aAfa2C27898CbE740f6'>
```

TransactionReceipt.**status**

An `IntEnum` object representing the status of the transaction:

- `1`: Successful
- `0`: Reverted
- `-1`: Pending
- `-2`: Dropped

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.status
1
```

TransactionReceipt.**timestamp**

The timestamp of the block that this transaction was included in.

```
>>> tx
<Transaction object
↪'0xac54b49987a77805bf6bdd78fb4211b3dc3d283ff0144c231a905afa75a06db0'>
>>> tx.timestamp
1588957325
```

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

`TransactionReceipt.`**`replace`**`(`*increment=None*, *gas_price=None*`)`

Broadcast an identical transaction with the same nonce and a higher gas price.

Exactly one of the following arguments must be provided:

- `increment`: Multiplier applied to the gas price of the current transaction in order to determine a new gas price
- `gas_price`: Absolute gas price to use in the replacement transaction

Returns a *`TransactionReceipt`* object.

```
>>> tx = accounts[0].transfer(accounts[1], 100, required_confs=0, gas_price="1␣
↪gwei")
Transaction sent:␣
↪0xc1aab54599d7875fc1fe8d3e375abb0f490cbb80d5b7f48cedaa95fa726f29be
  Gas price: 13.0 gwei   Gas limit: 21000   Nonce: 3
<Transaction object
↪'0xc1aab54599d7875fc1fe8d3e375abb0f490cbb80d5b7f48cedaa95fa726f29be'>

>>> tx.replace(1.1)
Transaction sent:␣
↪0x9a525e42b326c3cd57e889ad8c5b29c88108227a35f9763af33dccd522375212
  Gas price: 14.3 gwei   Gas limit: 21000   Nonce: 3
<Transaction '0x9a525e42b326c3cd57e889ad8c5b29c88108227a35f9763af33dccd522375212'>
```

`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`**`(`*expand=False*`)`

Display the complete sequence of contracts and functions called while execiting this transaction.

Each line is formatted as:

```
ContractName.functionName  (external call opcode)  start:stop  [internal / total␣
↪gas used]
```

- `start:stop` are index values for the *`TransactionReceipt.trace`*, showing where the call begins and ends
- for calls that include subcalls, gas use is displayed as `[gas used in this frame / gas used in this frame + subcalls]`
- Calls that terminate with a `REVERT` or `INVALID` instruction are highlighted in red

```
>>> tx.call_trace()
Call trace for '0x7824c6032966ca2349d6a14ec3174d48d546d0fb3020a71b08e50c7b31c1bcb1
↪':
Initial call cost  [21228 gas]
LiquidityGauge.deposit  0:3103  [64010 / 128030 gas]
├── LiquidityGauge._checkpoint  83:1826  [-6420 / 7698 gas]
│   ├── GaugeController.get_period_timestamp  [STATICCALL]  119:384  [2511 gas]
│   ├── ERC20CRV.start_epoch_time_write  [CALL]  411:499  [1832 gas]
│   ├── GaugeController.gauge_relative_weight_write  [CALL]  529:1017  [3178 /␣
↪7190 gas]
│   │   └── GaugeController.change_epoch  697:953  [2180 / 4012 gas]
│   │       └── ERC20CRV.start_epoch_time_write  [CALL]  718:806  [1832 gas]
│   └── GaugeController.period  [STATICCALL]  1043:1336  [2585 gas]
├── LiquidityGauge._update_liquidity_limit  1929:2950  [45242 / 54376 gas]
│   ├── VotingEscrow.balanceOf  [STATICCALL]  1957:2154  [2268 gas]
│   └── VotingEscrow.totalSupply  [STATICCALL]  2180:2768  [6029 / 6866 gas]
│       └── VotingEscrow.supply_at  2493:2748  [837 gas]
└── ERC20LP.transferFrom  [CALL]  2985:3098  [1946 gas]
```

Setting `expand=True` displays an expanded call trace that also includes function inputs and return values for all external calls.

```
>>> history[-1].call_trace(True)

Call trace for '0x7824c6032966ca2349d6a14ec3174d48d546d0fb3020a71b08e50c7b31c1bcb1
↪':
Initial call cost  [21228 gas]
LiquidityGauge.deposit  0:3103  [64010 / 128030 gas]
├── LiquidityGauge._checkpoint  83:1826  [-6420 / 7698 gas]
│   │
│   ├── GaugeController.get_period_timestamp  [STATICCALL]  119:384  [2511 gas]
│   │       ├── address: 0x0C41Fc429cC21BC3c826efB3963929AEdf1DBb8e
│   │       ├── input arguments:
│   │       │   └── p: 0
│   │       └── return value: 1594574319
│   │
...
```

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
```

(continues on next page)

(continued from previous page)

```
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;
    }
```

(continues on next page)

(continued from previous page)

```
    function mul(uint a, uint b) internal pure returns (uint c) {
        c = a * b;
```

classmethod TransactionReceipt.**wait**(*n*)

Will wait for n *confirmations* of the transaction. This has no effect if n is less than the current amount of confirmations.

```
>>> tx
<Transaction '0x830b842e24efae712b67dddd97633356122c36e6cf2193fcf9f7dc635c4cbe2f'>
>>> tx.wait(2)
This transaction already has 3 confirmations.
>>> tx.wait(6)
Required confirmations: 6/6
  Transaction confirmed - Block: 17   Gas used: 21000 (0.31%)
```

28.3.10 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*, *timeout=30*)

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'
```

classmethod Web3.**supports_traces**()

Boolean indicating if the currently connected node client supports the debug_traceTransaction RPC endpoint.

```
>>> web3.supports_traces
True
```

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 *convert.to_address*.

28.4 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.

28.4.1 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

class brownie.project.main.**Project**

Top level container that holds all objects related to a Brownie project.

Project Methods

classmethod Project.**load**()

Collects project source files, compiles new or updated contracts, instantiates *ContractContainer* objects, and populates the namespace.

Projects are typically loaded via *project.load*, but if you have a *Project* object that was previously closed you can reload it using this method.

classmethod Project.**load_config**()

Updates the configuration settings from the `brownie-config.yaml` file within this project's root folder.

classmethod Project.**close**(*raises = True*)

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

class brownie.project.main.**TempProject**

Simplified version of *Project*, used to hold contracts that are compiled via *project.compile_source*. 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*)

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**()

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 source code string and returns a `TempProject` object.

If Vyper source code is given, the contract name will be `Vyper`.

```
>>> from brownie import compile_source
>>> container = compile_source('''pragma solidity 0.4.25;

contract SimpleTest {

  string public name;

```

(continues on next page)

(continued from previous page)

```
  constructor (string _name) public {
    name = _name;
  }
}'''
>>>
>>> container
<TempProject object>
>>> container.SimpleTest
<ContractContainer object 'SimpleTest'>
```

main.**install_package**(*package_id*)

Install a package.

See the *Brownie Package Manager* documentation for more information on packages.

- `package_id`: Package identifier or ethPM URI

28.4.2 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

class brownie.project.build.**Build**

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']
```

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`*.

28.4.3 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*, *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.
- 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*, *language="Solidity"*`)`

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

28.4.4 `brownie.project.ethpm`

The `ethpm` module contains methods for interacting with ethPM manifests and registries. See the ethpm 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 ethpm, 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.

28.4.5 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'
```

28.4.6 brownie.project.sources

The `sources` module contains classes and methods to access project source code files and information about them.

Sources

`class` `brownie.project.sources.`**`Sources`**

The `Sources` object provides access to the `contracts/` and `interfaces/` files for a specific project. It is instantiated automatically when a project is loaded, 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 sorted list of contract source paths for the project.

```
>>> from brownie.project import sources
>>> sources.get_path_list()
['contracts/SafeMath.sol', 'contracts/Token.sol', 'interfaces/IToken.sol']
```

classmethod Sources.**get_contract_list**()

Returns a sorted list of contract names for the project.

```
>>> from brownie.project import sources
>>> sources.get_contract_list()
['SafeMath', 'Token']
```

classmethod Sources.**get_interface_list**()

Returns a sorted list of interface names for the project.

```
>>> from brownie.project import sources
>>> sources.get_interface_list()
['IToken']
```

classmethod Sources.**get_interface_hashes**()

Returns a dict of interface hashes in the form of `{'interfaceName': "hash"}`

classmethod Sources.**get_interface_sources**()

Returns a dict of interfaces sources in the form `{'path/to/interface': "source code"}`

classmethod Sources.**get_source_path**(*contract_name*)

Returns the path to the file where a contract or interface is located.

```
>>> from brownie.project import sources
>>> sources.get_source_path('Token')
'contracts/Token.sol'
```

Module Methods

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_contracts**(*full_source*)

Given a Solidity contract source as a string, returns a `dict` of source code for individual contracts.

```
>>> from brownie.project.sources import get_contracts
>>> get_contracts('''
... pragma solidity 0.5.0;
...
... contract Foo {
...     function bar() external returns (bool) {
```

(continues on next page)

(continued from previous page)

```
...         return true;
...     }
... }
...
... library Bar {
...     function baz(uint a, uint b) external pure returns (uint) {
...         return a + b;
...     }
... }''')
{
    'Foo': 'contract Foo {\n    function bar() external returns (bool) {\n        ␣
↪return true;\n    }\n}',
    'Bar': 'library Bar {\n    function baz(uint a, uint b) external pure returns␣
↪(uint) {\n        return a + b;\n    }\n}'
}
```

`sources.`**`get_pragma_spec`**(*source*, *path=None*)

Returns an NpmSpec object representing the first pragma statement found within a source file.

Raises *`PragmaError`* on failure. If `path` is not `None`, it will be included in the error string.

28.5 Test API

The `test` package contains classes and methods for running tests and evaluating test coverage.

This functionality is typically accessed via pytest. See *Writing Unit Tests*.

28.5.1 `brownie.test.fixtures`

The `fixtures` module contains custom fixtures provided by the Brownie Pytest plugin.

Pytest Fixtures

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.

`fixtures.`**`accounts`**

Session scope. Yields an instantiated *`Accounts`* container for the active project.

`fixtures.`**`a`**

Session scope. Short form of the *`accounts`* fixture.

`fixtures.`**`Contract`**

Session scope. Yields the *`Contract`* class, used to interact with contracts outside of the active project.

`fixtures.`**`history`**

Session scope. Yields an instantiated *`TxHistory`* object for the active project.

`fixtures.`**`rpc`**
Session scope. Yields an instantiated *`Rpc`* object.

`fixtures.`**`state_machine`**
Session scope. Yields the *`state_machine`* method, used to launc rule-based state machine tests.

`fixtures.`**`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.

`fixtures.`**`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.

`fixtures.`**`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.

28.5.2 `brownie.test.strategies`

The `strategies` module contains the *`strategy`* method, and related internal methods for generating Hypothesis search strategies.

`strategies.`**`strategy`**(*type_str*, ***kwargs*)
Returns a Hypothesis `SearchStrategy` based on the value of `type_str`. Depending on the type of strategy, different `kwargs` are available.

See the *Strategies* section for information on how to use this method.

28.5.3 `brownie.test.stateful`

The `stateful` module contains the *`state_machine`* method, and related internal classes and methods for performing stateful testing.

`stateful.`**`state_machine`**(*state_machine_class*, **args*, *settings=None*, ***kwargs*)
Executes a stateful test.

- `state_machine_class`: A state machine class to be used in the test. Be sure to pass the class itself, not an instance of the class.
- `*args`: Any arguments given here will be passed to the state machine's `__init__` method.
- `settings`: An optional dictionary of *Hypothesis settings* that will replace the defaults for this test only.

See the *Stateful Testing* section for information on how to use this method.

28.5.4 brownie.test.plugin

The `plugin` module is the entry point for the Brownie pytest plugin. It contains two `pytest` hook point methods that are used for setting up the plugin. The majority of the plugin functionality is handled by a *plugin manager* which is instantiated in the `pytest_configure` method.

28.5.5 brownie.test.manager

The `manager` module contains Brownie classes used internally to manage the Brownie pytest plugin.

Plugin Managers

One of these classes is instantiated in the `pytest_configure` method of `brownie.test.plugin`. Which is used depends on whether or not pytest-xdist is active.

class manager.base.**PytestBrownieBase**

Base class that is inherited by all Brownie plugin managers.

class manager.runner.**PytestBrownieRunner**

Runner plugin manager, used when `xdist` is not active.

class manager.runner.**PytestBrownieXdistRunner**

`xdist` runner plugin manager. Inherits from *PytestBrownieRunner*.

class manager.master.**PytestBrownieMaster**

`xdist` master plugin manager.

RevertContextManager

The `RevertContextManager` behaves similarly to `pytest.raises`.

class brownie.test.plugin.**RevertContextManager**(*revert_msg=None*, *dev_revert_msg=None*, *revert_pattern=None*, *dev_revert_pattern=None*)

Context manager used to handle *VirtualMachineError* exceptions. Raises `AssertionError` if no transaction has reverted when the context closes.

- `revert_msg`: Optional. Raises if the transaction does not revert with this error string.
- `dev_revert_msg`: Optional. Raises if the transaction does not revert with this *dev revert string*.
- `revert_pattern`: Regex pattern to compare against the transaction error string. Raises if the error string does not fully match the regex (partial matches are not allowed).
- `dev_revert_pattern`: Regex pattern to compare against the transaction dev revert string.

This class is available as `brownie.reverts` when `pytest` is active.

```python
import brownie

def test_transfer_reverts(Token, accounts):
    token = accounts[0].deploy(Token, "Test Token", "TST", 18, 1e23)
    with brownie.reverts():
        token.transfer(account[2], 1e24, {'from': accounts[1]})
```

28.5.6 `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. The report is grouped by contracts and functions are sorted by average gas used.

`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 *Report JSON Format* for more info on the return format.

28.5.7 `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.

28.6 Utils API

The `utils` package contains utility classes and methods that are used throughout Brownie.

28.6.1 `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 it returns an ANSI escape code for the given color:

```
>>> 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