**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.

**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](https://eth-brownie.readthedocs.io/en/v1.19.3/structure.html#structure) is created within the folder.

**Creating a Project from a Template**

You can initialize “[Brownie mixes](https://github.com/brownie-mix)”, simple templates to build your project upon. For many examples within the Brownie documentation we will use the [token](https://github.com/brownie-mix/token-mix) 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.

**React Template**

[React-Mix](https://github.com/brownie-mix/react-mix) is a bare-bones implementation of [Create React App](https://create-react-app.dev/) 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](https://github.com/brownie-mix/react-mix) for more information on how to use React with Brownie.

**Continuous Integration Template**

[Github-Actions-Mix](https://github.com/brownie-mix/github-actions-mix) is a template preconfigured for use with [Github Actions](https://github.com/features/actions) continuous integration, as well as other useful tools.

To initialize from this mix:

$ brownie bake github-actions

See the [Github-Actions-Mix repo](https://github.com/brownie-mix/github-actions-mix) for a detailed explanation of how to configure and use the tools within this template.

**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](https://eth-brownie.readthedocs.io/en/v1.19.3/build-folder.html#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](https://eth-brownie.readthedocs.io/en/v1.19.3/config.html#config-project-structure).

**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).

**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 only triggers a recompile if the interface is required by a contract.

Interfaces may be written in [Solidity](https://solidity.readthedocs.io/en/latest/contracts.html#interfaces) (.sol) or [Vyper](https://vyper.readthedocs.io/en/latest/structure-of-a-contract.html" \l "interfaces) (.vy), or supplied as a [JSON encoded ABI](https://solidity.readthedocs.io/en/latest/abi-spec.html#json) (.json).

**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](https://eth-brownie.readthedocs.io/en/v1.19.3/interaction.html#scripts) documentation for more information on Brownie scripts.

**tests/**

The tests folder holds Python scripts used for testing a project. Brownie uses the [pytest](https://docs.pytest.org/en/latest/) framework for unit testing.

See [Brownie Pytest](https://eth-brownie.readthedocs.io/en/v1.19.3/tests-pytest-intro.html#pytest) documentation for more information on testing a project.

# 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.

## 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

## 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](https://solidity.readthedocs.io/en/latest/contracts.html#interfaces) (.sol) or [Vyper](https://vyper.readthedocs.io/en/latest/structure-of-a-contract.html" \l "interfaces) (.vy). Vyper contracts are also able to directly import [JSON encoded ABI](https://solidity.readthedocs.io/en/latest/abi-spec.html#json) (.json) files.

## Compiler Settings

Compiler settings may be declared in the [configuration file](https://eth-brownie.readthedocs.io/en/v1.19.3/config.html#config) of a project. When no configuration file is present or settings are omitted, Brownie uses the following default values:

**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.

### 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](https://solidity.readthedocs.io/en/latest/layout-of-source-files.html#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](https://solidity.readthedocs.io/en/latest/contracts.html#interfaces) rather than actual contracts, and set all interface pragmas as >=0.4.22.

### 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.

If needed, the EVM version can be different between Solidity and Vyper by setting evm\_version under solc or vyper.

See the [Solidity EVM documentation](https://solidity.readthedocs.io/en/latest/using-the-compiler.html#setting-the-evm-version-to-target) or [Vyper EVM documentation](https://vyper.readthedocs.io/en/latest/compiling-a-contract.html" \l "setting-the-target-evm-version) for more info on the different EVM versions and how they affect compilation.

### 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](https://eth-brownie.readthedocs.io/en/v1.19.3/config.html#config) 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](https://solidity.readthedocs.io/en/latest/using-the-compiler.html#input-description) and it’s [available settings](https://solidity.readthedocs.io/en/latest/using-the-compiler.html#input-description).

### 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";

## 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.vyper** **import** install\_vyper

**>>>** install\_vyper("0.2.4")