

---

# **Hammurabi Documentation**

***Release 0.7.3***

**Gábor Boros**

**Jun 25, 2020**



# CONTENTS

<b>1</b>	<b>Hammurabi</b>	<b>1</b>
1.1	Features . . . . .	1
1.2	Installation . . . . .	2
1.3	Configuration . . . . .	2
1.4	Command line options . . . . .	2
1.5	Usage examples . . . . .	2
1.6	Custom Rules . . . . .	5
1.7	Community . . . . .	5
1.8	Contributing . . . . .	6
1.9	Why Hammurabi? . . . . .	6
<b>2</b>	<b>Installation</b>	<b>7</b>
2.1	Stable release . . . . .	7
2.2	From sources . . . . .	7
<b>3</b>	<b>Configuration</b>	<b>9</b>
3.1	Overview . . . . .	9
3.2	Hammurabi configuration . . . . .	9
3.3	Pillar configuration . . . . .	10
<b>4</b>	<b>Rules</b>	<b>11</b>
4.1	Base rule . . . . .	11
4.2	Attributes . . . . .	12
4.3	Directories . . . . .	13
4.4	Files . . . . .	15
4.5	Ini files . . . . .	18
4.6	JSON files . . . . .	21
4.7	Operations . . . . .	25
4.8	Templates . . . . .	27
4.9	Text files . . . . .	27
4.10	YAML files . . . . .	29
<b>5</b>	<b>Preconditions</b>	<b>35</b>
5.1	Base precondition . . . . .	35
5.2	Attributes . . . . .	35
5.3	Directories . . . . .	37
5.4	Files . . . . .	38
5.5	Text files . . . . .	39
<b>6</b>	<b>Reporters</b>	<b>41</b>
6.1	Base reporter . . . . .	41

6.2	Formatted reporters . . . . .	41
<b>7</b>	<b>hammurabi</b>	<b>43</b>
7.1	hammurabi package . . . . .	43
<b>8</b>	<b>Contributing</b>	<b>99</b>
8.1	Types of Contributions . . . . .	99
8.2	Get Started! . . . . .	100
8.3	Pull Request Guidelines . . . . .	101
8.4	Releasing . . . . .	101
<b>9</b>	<b>Vulnerabilities</b>	<b>103</b>
9.1	Reporting vulnerabilities . . . . .	103
<b>10</b>	<b>Credits</b>	<b>105</b>
10.1	Development Lead . . . . .	105
10.2	Contributors . . . . .	105
<b>11</b>	<b>CHANGELOG</b>	<b>107</b>
11.1	Unreleased . . . . .	107
11.2	0.7.3 - 2020-05-25 . . . . .	107
11.3	0.7.2 - 2020-05-25 . . . . .	107
11.4	0.7.1 - 2020-05-22 . . . . .	107
11.5	0.7.0 - 2020-04-28 . . . . .	108
11.6	0.6.0 - 2020-04-06 . . . . .	109
11.7	0.5.0 - 2020-03-31 . . . . .	109
11.8	0.4.0 - 2020-03-31 . . . . .	109
11.9	0.3.1 - 2020-03-26 . . . . .	110
11.10	0.3.0 - 2020-03-25 . . . . .	111
11.11	0.2.0 - 2020-03-23 . . . . .	111
11.12	0.1.2 - 2020-03-18 . . . . .	112
11.13	0.1.1 - 2020-03-17 . . . . .	113
11.14	0.1.0 - 2020-03-12 . . . . .	113
<b>12</b>	<b>Indices and tables</b>	<b>115</b>
	<b>Python Module Index</b>	<b>117</b>
	<b>Index</b>	<b>119</b>

## **HAMMURABI**

Mass changes made easy.

Hammurabi is an extensible CLI tool responsible for enforcing user-defined rules on a git repository.

### **1.1 Features**

Hammurabi integrates well with both git and Github to make sure that the execution happens on a separate branch and the committed changes are pushed to the target repository. After pushing to the target repository, a pull request will be opened.

Hammurabi supports several operations (Rules) by default. These Rules can do

- file and directory operations like copy, move, create or delete
- manipulation of attributes like ownership or access permissions change
- file and directory manipulations
- piped rule execution (output of a rule is the input of the next rule)
- children rule execution (output of a rule is the input of the upcoming rules)
- creating files from Jinja2 templates

Supported file formats:

- plain text
- ini
- json
- yaml (basic, single document operations)

Upcoming file format support:

- toml

- `hocon`

## 1.2 Installation

Hammurabi can be installed by running `pip install hammurabi` and it requires Python 3.7.0+ to run. This is the preferred method to install Hammurabi, as it will always install the most recent stable release. If you don't have `pip` installed, this [Python installation guide](#) can guide you through the process.

## 1.3 Configuration

For configuration instructions, please visit the [documentation](#) site.

## 1.4 Command line options

```
hammurabi [OPTIONS] COMMAND [ARGS]...
```

Hammurabi is an extensible CLI tool responsible **for** enforcing user-defined rules on a git repository.

Find more information at: <https://hammurabi.readthedocs.io/latest/>

Options:

<code>-c, --config PATH</code>	Set the configuration file. [default: <code>pyproject.toml</code> ]
<code>--repository TEXT</code>	Set the remote repository. Required format: <code>owner/repository</code>
<code>--github-token TEXT</code>	Set github access token
<code>--log-level [DEBUG INFO WARNING ERROR]</code>	Set logging level.
<code>--help</code>	Show this message and exit.

Commands:

<code>describe</code>	Show details of a specific resource or group of resources.
<code>enforce</code>	Execute all registered Law.
<code>get</code>	Show a specific resource or group of resources.
<code>version</code>	Print Hammurabi version.

## 1.5 Usage examples

In every case, make sure that you clone the target repository prior using Hammurabi. After cloning the repository, always set the current working directory to the target's path. Hammurabi will not clone the target repository or change its execution directory.

### 1.5.1 Enforce registered laws

```
$ hammurabi enforce
[INFO] 2020-14-07 16:31 - Checkout branch "hammurabi"
[INFO] 2020-14-07 16:31 - Executing law "L001"
[INFO] 2020-14-07 16:31 - Running task for "configure file exists"
[INFO] 2020-14-07 16:31 - Rule "configure file exists" finished successfully
[INFO] 2020-14-07 16:31 - Running task for "Minimum clang version is set"
[INFO] 2020-14-07 16:31 - Rule "Minimum clang version is set" finished successfully
[INFO] 2020-14-07 16:31 - Running task for "Minimum icc version is set"
[INFO] 2020-14-07 16:31 - Rule "Minimum icc version is set" finished successfully
[INFO] 2020-14-07 16:31 - Running task for "Minimum lessc version is set"
[INFO] 2020-14-07 16:31 - Rule "Minimum lessc version is set" finished successfully
[INFO] 2020-14-07 16:31 - Running task for "Maximum lessc version is set"
[INFO] 2020-14-07 16:31 - Rule "Maximum lessc version is set" finished successfully
[INFO] 2020-14-07 16:31 - Pushing changes
[INFO] 2020-14-07 16:35 - Checking for opened pull request
[INFO] 2020-14-07 16:35 - Opening pull request
```

### 1.5.2 Listing available laws

```
$ hammurabi get laws
- Unicorn config set up properly
```

### 1.5.3 Get info about a law by its name

```
$ hammurabi get law "Unicorn config set up properly"
Unicorn config set up properly

Change the unicorn configuration based on our learnings
described at: https://google.com/?q=unicorn.

If the unicorn configuration does not exist, create a
new one configuration file.
```

### 1.5.4 Get all registered (root) rules

```
$ hammurabi get rules
- Rule 1
- Rule 5
```

### 1.5.5 Get a rule by its name

```
$ hammurabi get rule "Rule 1"
Rule 1
```

Ensure that a file exists. If the file does not exists,  
this `:class:`hammurabi.rules.base.Rule`` will create it.

Due to the file is already created by `:func:`pre_task_hook``  
there is no need to **do** anything just **return** the input parameter.

### 1.5.6 Describe a law by its name

```
$ hammurabi describe law "Gunicorn config set up properly"
Gunicorn config set up properly
```

Change the gunicorn configuration based on our learnings  
described at: <http://docs.gunicorn.org/en/latest/configure.html>.

If the gunicorn configuration does not exist, create a  
new one configuration file.

Rules:

```
--> Rule 1
--> Rule 2
--> Rule 3
--> Rule 4
--> Rule 5
```

### 1.5.7 Describe a rule by its name

```
$ hammurabi describe rule "Rule 1"
Rule 1
```

Ensure that a file exists. If the file does not exists,  
this `:class:`hammurabi.rules.base.Rule`` will create it.

Due to the file is already created by `:func:`pre_task_hook``  
there is no need to **do** anything just **return** the input parameter.

Chain:

```
--> Rule 1
--> Rule 2
--> Rule 3
--> Rule 4
```



## 1.5.8 Getting the execution order of laws and rules

```
$ hammurabi get order
- Unicorn config set up properly
--> Rule 1
--> Rule 2
--> Rule 3
--> Rule 4
--> Rule 5
```

## 1.6 Custom Rules

Although the project aims to support as many general operations as it can, the need for adding custom rules may arise.

To extend Hammurabi with custom rules, you will need to inherit a class from `Rule` and define its abstract methods.

The following example will show you how to create and use a custom rule. For more reference please check how the existing rules are implemented.

```
# custom.py
import shutil
import logging
from hammurabi.mixins import GitMixin
from hammurabi.rules.base import Rule

class CustomOwnerChanged(Rule, GitMixin):
    """
    Change the ownership of a file or directory to <original user>:admin.
    """

    def __init__(self, name: str, path: Optional[Path] = None, **kwargs):
        super().__init__(name, path, **kwargs)

    def post_task_hook(self):
        self.git_add(self.param)

    def task(self) -> Path:
        # Since ``Rule`` is setting its 2nd parameter to ``self.param``,
        # we can use ``self.param`` to access the target file's path.
        logging.debug('Changing group of "%s" to admin', str(self.param))
        shutil.chown(self.param, group="admin")
        return self.param
```

## 1.7 Community

If you need help or you would like to be part of the Hammurabi community, join us on [discord](#).

## 1.8 Contributing

Hurray, You reached this section, which means you are ready to contribute.

Please read our contributing [guideline](#). This guideline will walk you through how can you successfully contribute to Hammurabi.

### 1.8.1 Installation

For development you will need [poetry](#) and [pre-commit](#). After poetry installed, simply run *poetry install*. This command will both create the virtualenv and install development dependencies for you.

### 1.8.2 Useful make Commands

Command	Description
help	Print available make commands
clean	Remove all artifacts
clean-build	Remove build artifacts
clean-mypy	Remove mypy artifacts
clean-pyc	Remove Python artifacts
clean-test	Remove test artifacts
docs	Generate Sphinx documentation
format	Run several formatters
lint	Run several linters after format
test	Run all tests with coverage
test-unit	Run unit tests with coverage
test-integration	Run integration tests with coverage

## 1.9 Why Hammurabi?

Hammurabi was the sixth king in the Babylonian dynasty, which ruled in central Mesopotamia from c. 1894 to 1595 B.C.

The Code of Hammurabi was one of the earliest and most complete written legal codes and was proclaimed by the Babylonian king Hammurabi, who reigned from 1792 to 1750 B.C. Hammurabi expanded the city-state of Babylon along the Euphrates River to unite all of southern Mesopotamia. The Hammurabi code of laws, a collection of 282 rules, established standards for commercial interactions and set fines and punishments to meet the requirements of justice. Hammurabi's Code was carved onto a massive, finger-shaped black stone stele (pillar) that was looted by invaders and finally rediscovered in 1901.

## INSTALLATION

### 2.1 Stable release

To install Hammurabi, run this command in your terminal:

```
$ pip install hammurabi
```

This is the preferred method to install Hammurabi, as it will always install the most recent stable release.

If you don't have `pip` installed, this [Python installation guide](#) can guide you through the process.

### 2.2 From sources

The sources for Hammurabi can be downloaded from the [Github repo](#).

You can either clone the public repository:

```
$ git clone git://github.com/gabor-boros/hammurabi
```

Or download the [tarball](#):

```
$ curl -OL https://github.com/gabor-boros/hammurabi/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```



## CONFIGURATION

### 3.1 Overview

### 3.2 Hammurabi configuration

You can set the following options in your `pyproject.toml` config file's `[hammurabi]` section. Config option marked with \* (asterisk) is mandatory (set by CLI argument or project config). Hammurabi can be configured through environment variables too. To use an environment variable based config option set the `HAMMURABI_<CONFIG_OPTION>` where `<CONFIG_OPTION>` is in uppercase and matches one of the options below.

Config option	Description	Default value
<code>pillar_config *</code>	location of pillar config	None
<code>pillar_name</code>	name of the pillar variable	pillar
<code>log_level</code>	logging level of the program	INFO
<code>log_path</code>	path to the log file or None	./hammurabi.log
<code>log_format</code>	format of the log lines	BASIC_FORMAT
<code>repository</code>	git repository (owner/repo)	None
<code>git_branch_name</code>	working branch name	hammurabi
<code>dry_run</code>	enforce without any modification	False
<code>rule_can_abort</code>	if a rule fails it aborts the whole execution	False
<code>report_name</code>	report file's name to generate	hammurabi_report.json

For HTTPS git remotes do not forget to set the `GIT_USERNAME` and `GIT_PASSWORD` environment variables. For SSH git remotes please add your ssh key before using Hammurabi.

#### 3.2.1 Examples

Example content of the `pyproject.toml` file.

```
[hammurabi]
pillar_config = "/tmp/config/global_config.py"
working_dir = "/tmp/clones/hammurabi"
repository = "gabor-boros/hammurabi"
git_branch_name = "custom-branch-name"
log_level = "WARNING"
log_file = "/var/log/hammurabi.log"
log_format = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
```

(continues on next page)

(continued from previous page)

```
rule_can_abort = true
report_name = "hammurabi_report.json"
```

## 3.3 Pillar configuration

The pillar needs no configuration. All the thing the developer must do is creating a *hammurabi.pillar.Pillar* object and registering the laws to it.

### 3.3.1 Using custom rules

Custom rules are not different from built-in one. In case of a custom rule, just import and use it.

### 3.3.2 Examples

```
>>> from hammurabi import Law, Pillar
>>> from mycompany.rules import MyCustomRule
>>>
>>> meaning_of_life = Law(
>>>     name="...",
>>>     description="...",
>>>     rules=[MyCustomRule]
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(meaning_of_life)
```

## 4.1 Base rule

```
class hammurabi.rules.base.Rule(name: str, param: Any, preconditions: Iterable[hammurabi.preconditions.base.Precondition] = (),  
                                pipe: Optional[Rule] = None, children: Iterable[Rule] = ())
```

Abstract class which describes the bare minimum and helper functions for Rules. A rule defines what and how should be executed. Since a rule can have piped and children rules, the “parent” rule is responsible for those executions. This kind of abstraction allows to run both piped and children rules sequentially in a given order.

Example usage:

```
>>> from typing import Optional
>>> from pathlib import Path
>>> from hammurabi import Rule
>>> from hammurabi.mixins import GitMixin
>>>
>>> class SingleFileRule(Rule, GitMixin):
>>>     def __init__(self, name: str, path: Optional[Path] = None, **kwargs) -> _
↳None:
>>>         super().__init__(name, path, **kwargs)
>>>
>>>     def post_task_hook(self):
>>>         self.git_add(self.param)
>>>
>>>     @abstractmethod
>>>     def task(self) -> Path:
>>>         pass
```

### Parameters

- **name** (*str*) – Name of the rule which will be used for printing
- **param** (*Any*) – Input parameter of the rule will be used as `self.param`
- **preconditions** (*Iterable["Rule"]*) – “Boolean Rules” which returns a truthy or falsy value
- **pipe** (*Optional["Rule"]*) – Pipe will be called when the rule is executed successfully
- **children** (*Iterable["Rule"]*) – Children will be executed after the piped rule if there is any

**Warning:** Preconditions can be used in several ways. The most common way is to run “Boolean Rules” which takes a parameter and returns a truthy or falsy value. In case of a falsy return, the precondition will fail and the rule will not be executed.

If any modification is done by any of the rules which are used as a precondition, those changes will be committed.

## 4.2 Attributes

### 4.2.1 OwnerChanged

```
class hammurabi.rules.attributes.OwnerChanged(name: str, path: Optional[pathlib.Path] =  
None, new_value: Optional[str] = None,  
**kwargs)
```

Change the ownership of a file or directory.

The new ownership of a file or directory can be set in three ways. To set only the user use `new_value="username"`. To set only the group use `new_value=":group_name"` (please note the colon :). It is also possible to set both username and group at the same time by using `new_value="username:group_name"`.

Example usage:

```
>>> from pathlib import Path  
>>> from hammurabi import Law, Pillar, OwnerChanged  
>>>  
>>> example_law = Law(  
>>>     name="Name of the law",  
>>>     description="Well detailed description what this law does.",  
>>>     rules=(  
>>>         OwnerChanged(  
>>>             name="Change ownership of nginx config",  
>>>             path=Path("./nginx.conf"),  
>>>             new_value="www:web_admin"  
>>>         ),  
>>>     )  
>>> )  
>>>  
>>> pillar = Pillar()  
>>> pillar.register(example_law)
```

### 4.2.2 ModeChanged

```
class hammurabi.rules.attributes.ModeChanged(name: str, path: Optional[pathlib.Path] =  
None, new_value: Optional[int] = None,  
**kwargs)
```

Change the mode of a file or directory.

Supported modes:



Config option	Description
stat.S_ISUID	Set user ID on execution.
stat.S_ISGID	Set group ID on execution.
stat.S_ENFMT	Record locking enforced.
stat.S_ISVTX	Save text image after execution.
stat.S_IREAD	Read by owner.
stat.S_IWRITE	Write by owner.
stat.S_IEXEC	Execute by owner.
stat.S_IRWXU	Read, write, and execute by owner.
stat.S_IRUSR	Read by owner.
stat.S_IWUSR	Write by owner.
stat.S_IXUSR	Execute by owner.
stat.S_IRWXG	Read, write, and execute by group.
stat.S_IRGRP	Read by group.
stat.S_IWGRP	Write by group.
stat.S_IXGRP	Execute by group.
stat.S_IRWXO	Read, write, and execute by others.
stat.S_IROTH	Read by others.
stat.S_IWOTH	Write by others.
stat.S_IXOTH	Execute by others.

Example usage:

```
>>> import stat
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, ModeChanged
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         ModeChanged(
>>>             name="Update script must be executable",
>>>             path=Path("./scripts/update.sh"),
>>>             new_value=stat.S_IXGRP | stat.S_IXGRP | stat.S_IXOTH
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

## 4.3 Directories

### 4.3.1 DirectoryExists

**class** hammurabi.rules.directories.**DirectoryExists** (*name:* *str*, *path:* *Optional[pathlib.Path]* = *None*, *\*\*kwargs*)

Ensure that a directory exists. If the directory does not exists, make sure the directory is created.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryExists(
>>>             name="Create secrets directory",
>>>             path=Path("./secrets")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.3.2 DirectoryNotExists

**class** hammurabi.rules.directories.**DirectoryNotExists** (*name: str, path: Optional[pathlib.Path] = None, \*\*kwargs*)

Ensure that the given directory does not exists.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryNotExists(
>>>             name="Remove unnecessary directory",
>>>             path=Path("./temp")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.3.3 DirectoryEmptied

**class** hammurabi.rules.directories.**DirectoryEmptied** (*name: str, path: Optional[pathlib.Path] = None, \*\*kwargs*)

Ensure that the given directory's content is removed. Please note the difference between emptying a directory and recreating it. The latter results in lost ACLs, permissions and modes.

Example usage:

```

>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, DirectoryEmptied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryEmptied(
>>>             name="Empty results directory",
>>>             path=Path("./test-results")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

## 4.4 Files

### 4.4.1 FileExists

**class** hamurabi.rules.files.**FileExists** (*name: str, path: Optional[pathlib.Path] = None,*  
*\*\*kwargs*)

Ensure that a file exists. If the file does not exists, make sure the file is created.

Example usage:

```

>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, FileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileExists(
>>>             name="Create service descriptor",
>>>             path=Path("./service.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

### 4.4.2 FilesExist

**class** `hammurabi.rules.files.FilesExist` (*name: str, paths: Optional[Iterable[pathlib.Path]] =*  
*()*, *\*\*kwargs*)

Ensure that all files exists. If the files does not exists, make sure the files are created.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FilesExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FilesExist(
>>>             name="Create test files",
>>>             paths=[
>>>                 Path("./file_1"),
>>>                 Path("./file_2"),
>>>                 Path("./file_3"),
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.4.3 FileNotExists

**class** `hammurabi.rules.files.FileNotExists` (*name: str, path: Optional[pathlib.Path] =*  
*None*, *\*\*kwargs*)

Ensure that the given file does not exists. If the file exists remove it, otherwise do nothing and return the original path.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileNotExists(
>>>             name="Remove unused file",
>>>             path=Path("./debug.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

#### 4.4.4 FilesNotExist

**class** `hammurabi.rules.files.FilesNotExist` (*name: str, paths: Optional[Iterable[pathlib.Path]] = (), \*\*kwargs*)

Ensure that the given files does not exist. If the files exist remove them, otherwise do nothing and return the original paths.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FilesNotExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FilesNotExist(
>>>             name="Remove several files",
>>>             paths=[
>>>                 Path("./file_1"),
>>>                 Path("./file_2"),
>>>                 Path("./file_3"),
>>>             ]
>>>         ),
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

#### 4.4.5 FileEmptied

**class** `hammurabi.rules.files.FileEmptied` (*name: str, path: Optional[pathlib.Path] = None, \*\*kwargs*)

Remove the content of the given file, but keep the file. Please note the difference between emptying a file and recreating it. The latter results in lost ACLs, permissions and modes.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileEmptied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileEmptied(
>>>             name="Empty the check log file",
>>>             path=Path("/var/log/service/check.log")
>>>         ),
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

## 4.5 Ini files

### 4.5.1 SectionExists

```
class hammurabi.rules.ini.SectionExists (name: str, path: Optional[pathlib.Path] = None,  
                                           target: Optional[str] = None, options: Iter-  
                                           able[Tuple[str, Any]] = (), add_after: bool = True,  
                                           **kwargs)
```

Ensure that the given config section exists. If needed, the rule will create a config section with the given name, and optionally the specified options. In case options are set, the config options will be assigned to that config sections.

Similarly to `hammurabi.rules.text.LineExists`, this rule is able to add a section before or after a target section. The limitation compared to `LineExists` is that the `SectionExists` rule is only able to add the new entry exactly before or after its target.

Example usage:

```
>>> from pathlib import Path  
>>> from hammurabi import Law, Pillar, SectionExists  
>>>  
>>> example_law = Law(  
>>>     name="Name of the law",  
>>>     description="Well detailed description what this law does.",  
>>>     rules=(  
>>>         SectionExists(  
>>>             name="Ensure section exists",  
>>>             path=Path("./config.ini"),  
>>>             section="polling",  
>>>             target="add_after_me",  
>>>             options=(  
>>>                 ("interval", "2s"),  
>>>                 ("abort_on_error", True),  
>>>             ),  
>>>         ),  
>>>     ),  
>>> )  
>>> pillar = Pillar()  
>>> pillar.register(example_law)
```

**Warning:** When `options` parameter is set, make sure you are using an iterable tuple. The option keys must be strings, but there is no limitation for the value. It can be set to anything what the parser can handle. For more information on the parser, please visit the documentation of [configupdater](#).

### 4.5.2 SectionNotExists

```
class hammurabi.rules.ini.SectionNotExists (name: str, path: Optional[pathlib.Path]
                                             = None, section: Optional[str] = None,
                                             **kwargs)
```

Make sure that the given file not contains the specified line. When a section removed, all the options belonging to it will be removed too.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionNotExists(
>>>             name="Ensure section removed",
>>>             path=Path("./config.ini"),
>>>             section="invalid",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.5.3 SectionRenamed

```
class hammurabi.rules.ini.SectionRenamed (name: str, path: Optional[pathlib.Path] = None,
                                             new_name: Optional[str] = None, **kwargs)
```

Ensure that a section is renamed. None of its options will be changed.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionRenamed(
>>>             name="Ensure section renamed",
>>>             path=Path("./config.ini"),
>>>             section="polling",
>>>             new_name="fetching",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.5.4 OptionsExist

**class** `hammurabi.rules.ini.OptionsExist` (*name: str, path: Optional[pathlib.Path] = None, options: Iterable[Tuple[str, Any]] = None, force\_value: bool = False, \*\*kwargs*)

Ensure that the given config option exists. If needed, the rule will create a config option with the given value. In case the `force_value` parameter is set to `True`, the original values will be replaced by the give ones.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionsExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionsExist(
>>>             name="Ensure options are changed",
>>>             path=Path("./config.ini"),
>>>             section="fetching",
>>>             options=(
>>>                 ("interval", "2s"),
>>>                 ("abort_on_error", True),
>>>             ),
>>>             force_value=True,
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** When using the `force_value` parameter, please note that all the existing option values will be replaced by those set in `options` parameter.

### 4.5.5 OptionsNotExist

**class** `hammurabi.rules.ini.OptionsNotExist` (*name: str, path: Optional[pathlib.Path] = None, options: Iterable[str] = (), \*\*kwargs*)

Remove one or more option from a section.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionsNotExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionsNotExist(
>>>             name="Ensure options are removed",
>>>             path=Path("./config.ini"),
>>>             section="invalid",
```

(continues on next page)



(continued from previous page)

```

>>>         options=(
>>>             "remove",
>>>             "me",
>>>             "please",
>>>         )
>>>     ),
>>> )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

## 4.5.6 OptionRenamed

**class** `hammurabi.rules.ini.OptionRenamed` (*name: str, path: Optional[pathlib.Path] = None, option: Optional[str] = None, new\_name: Optional[str] = None, \*\*kwargs*)

Ensure that an option of a section is renamed.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionRenamed(
>>>             name="Rename an option",
>>>             path=Path("./config.ini"),
>>>             section="my_section",
>>>             option="typo",
>>>             new_name="correct",
>>>         ),
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

## 4.6 JSON files

### 4.6.1 JSONKeyExists

**class** `hammurabi.rules.json.JSONKeyExists` (*name: str, path: Optional[pathlib.Path] = None, key: str = "", value: Union[None, list, dict, str, int, float] = None, \*\*kwargs*)

Ensure that the given key exists. If needed, the rule will create a key with the given name, and optionally the specified value. In case the value is set, the value will be assigned to the key. If no value is set, the key will be created with an empty value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONKeyExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONKeyExists(
>>>             name="Ensure service descriptor has stack",
>>>             path=Path("./service.json"),
>>>             key="stack",
>>>             value="my-awesome-stack",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** Compared to `hammurabi.rules.text.LineExists`, this rule is NOT able to add a key before or after a target.

## 4.6.2 JSONKeyNotExists

**class** `hammurabi.rules.json.JSONKeyNotExists` (*name: str, path: Optional[pathlib.Path] = None, key: str = "", \*\*kwargs*)

Ensure that the given key not exists. If needed, the rule will remove a key with the given name, including its value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONKeyNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONKeyNotExists(
>>>             name="Ensure outdated_key is removed",
>>>             path=Path("./service.json"),
>>>             key="outdated_key",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.6.3 JSONKeyRenamed

```
class hammurabi.rules.json.JSONKeyRenamed(name: str, path: Optional[pathlib.Path] =
None, key: str = "", new_name: str = "",
**kwargs)
```

Ensure that the given key is renamed. In case the key can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename they will fail otherwise.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONKeyRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONKeyRenamed(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.json"),
>>>             key="development.depends_on",
>>>             value="dependencies",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.6.4 JSONValueExists

```
class hammurabi.rules.json.JSONValueExists(name: str, path: Optional[pathlib.Path] =
None, key: str = "", value: Union[None, list,
dict, str, int, float] = None, **kwargs)
```

Ensure that the given key has the expected value(s). In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

This rule is special in the way that the value can be almost anything. For more information please read the warning below.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONValueExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONValueExists(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.json"),
>>>             key="development.dependencies",
>>>             value=["service1", "service2", "service3"],
>>>         ),
>>>     )
>>>     # Or
```

(continues on next page)

(continued from previous page)

```

>>>     JSONValueExists(
>>>         name="Add infra alerting to existing alerting components",
>>>         path=Path("./service.json"),
>>>         key="development.alerting",
>>>         value={"infra": "#slack-channel-2"},
>>>     ),
>>>     # Or
>>>     JSONValueExists(
>>>         name="Add support info",
>>>         path=Path("./service.json"),
>>>         key="development.supported",
>>>         value=True,
>>>     ),
>>>     # Or even
>>>     JSONValueExists(
>>>         name="Make sure that no development branch is set",
>>>         path=Path("./service.json"),
>>>         key="development.branch",
>>>         value=None,
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**Warning:** Since the value can be anything from `None` to a list of lists, and rule piping passes the 1st argument (path) to the next rule the value parameter can not be defined in `__init__` before the path. Hence the value parameter must have a default value. The default value is set to `None`, which translates to the following:

Using the `JSONValueExists` rule and not assigning value to value parameter will set the matching key's value to `None` by default in the document.

#### 4.6.5 JSONValueNotExists

```

class hamurabi.rules.json.JSONValueNotExists (name: str, path: Optional[pathlib.Path] =
                                              None, key: str = "", value: Union[str, int,
                                              float] = None, **kwargs)

```

Ensure that the key has no value given. In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

Compared to `hamurabi.rules.json.JSONValueExists`, this rule can only accept simple value for its value parameter. No `list`, `dict`, or `None` can be used.

Based on the key's value's type if the value contains (or equals for simple types) value provided in the value parameter the value is:

1. Set to `None` (if the key's value's type is not a dict or list)
2. Removed from the list (if the key's value's type is a list)
3. Removed from the dict (if the key's value's type is a dict)

Example usage:

```

>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, JSONValueNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONValueNotExists(
>>>             name="Remove decommissioned service from dependencies",
>>>             path=Path("./service.json"),
>>>             key="development.dependencies",
>>>             value="service4",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

## 4.7 Operations

### 4.7.1 Moved

**class** hamurabi.rules.operations.**Moved**(name: str, path: Optional[pathlib.Path] = None, destination: Optional[pathlib.Path] = None, \*\*kwargs)

Move a file or directory from “A” to “B”.

Example usage:

```

>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, Moved
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Moved(
>>>             name="Move pyproject.toml to its place",
>>>             path=Path("/tmp/generated/pyproject.toml.template"),
>>>             destination=Path("./pyproject.toml"), # Notice the rename!
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

## 4.7.2 Renamed

**class** `hammurabi.rules.operations.Renamed` (*name: str, path: Optional[pathlib.Path] = None, new\_name: Optional[str] = None, \*\*kwargs*)

This rule is a shortcut for `hammurabi.rules.operations.Moved`. Instead of destination path a new name is required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename pyproject.toml.bkp",
>>>             path=Path("/tmp/generated/pyproject.toml.bkp"),
>>>             new_name="pyproject.toml",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

## 4.7.3 Copied

**class** `hammurabi.rules.operations.Copied` (*name: str, path: Optional[pathlib.Path] = None, destination: Optional[pathlib.Path] = None, \*\*kwargs*)

Ensure that the given file or directory is copied to the new path.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Copied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Copied(
>>>             name="Create backup file",
>>>             path=Path("./service.yaml"),
>>>             destination=Path("./service.bkp.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

## 4.8 Templates

### 4.8.1 TemplateRendered

```
class hammurabi.rules.templates.TemplateRendered (name: str, template: Optional[pathlib.Path] = None, destination: Optional[pathlib.Path] = None, context: Optional[Dict[str, Any]] = None, **kwargs)
```

Render a file from a Jinja2 template. In case the destination file not exists, this rule will create it, otherwise the file will be overridden.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, TemplateRendered
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         TemplateRendered(
>>>             name="Create gunicorn config from template",
>>>             template=Path("/tmp/templates/gunicorn.conf.py"),
>>>             destination=Path("./gunicorn.conf.py"),
>>>             context={
>>>                 "keepalive": 65
>>>             },
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

## 4.9 Text files

### 4.9.1 LineExists

```
class hammurabi.rules.text.LineExists (name: str, path: Optional[pathlib.Path] = None, text: Optional[str] = None, target: Optional[str] = None, position: int = 1, respect_indentation: bool = True, ensure_trailing_newline: bool = False, **kwargs)
```

Make sure that the given file contains the required line. This rule is capable for inserting the expected text before or after the unique target text respecting the indentation of its context.

The default behaviour is to insert the required text exactly after the target line, and respect its indentation. Please note that `text` and `target` parameters are required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineExists, IsLineNotExists
>>>
```

(continues on next page)

(continued from previous page)

```

>>> gunicorn_config = Path("./gunicorn.conf.py")
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineExists(
>>>             name="Extend gunicorn config",
>>>             path=gunicorn_config,
>>>             text="keepalive = 65",
>>>             target=r"^bind.*",
>>>             preconditions=[
>>>                 IsLineNotExists(path=gunicorn_config, criteria=r"^keepalive.*
↵")
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**Note:** The indentation of the target text will be extracted by a simple regular expression. If a more complex regexp is required, please inherit from this class.

## 4.9.2 LineNotExists

**class** `hammurabi.rules.text.LineNotExists` (*name: str, path: Optional[pathlib.Path] = None, text: Optional[str] = None, \*\*kwargs*)

Make sure that the given file not contains the specified line.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineNotExists(
>>>             name="Remove keepalive",
>>>             path=Path("./gunicorn.conf.py"),
>>>             text="keepalive = 65",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```



### 4.9.3 LineReplaced

```
class hammurabi.rules.text.LineReplaced(name: str, path: Optional[pathlib.Path] = None,
                                         text: Optional[str] = None, target: Optional[str]
                                         = None, respect_indentation: bool = True,
                                         **kwargs)
```

Make sure that the given text is replaced in the given file.

The default behaviour is to replace the required text with the exact same indentation that the target line has. This behaviour can be turned off by setting the `respect_indentation` parameter to `False`. Please note that `text` and `target` parameters are required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineReplaced
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineReplaced(
>>>             name="Replace typo using regex",
>>>             path=Path("./unicorn.conf.py"),
>>>             text="keepalive = 65",
>>>             target=r"^kepalive.*",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

---

**Note:** The indentation of the target text will be extracted by a simple regular expression. If a more complex regexp is required, please inherit from this class.

---

**Warning:** This rule will replace all the matching lines in the given file. Make sure the given target regular expression is tested before the rule used against production code.

## 4.10 YAML files

### 4.10.1 YAMLKeyExists

```
class hammurabi.rules.yaml.YAMLKeyExists(name: str, path: Optional[pathlib.Path] = None,
                                           key: str = "", value: Union[None, list, dict, str, int,
                                                                    float] = None, **kwargs)
```

Ensure that the given key exists. If needed, the rule will create a key with the given name, and optionally the specified value. In case the value is set, the value will be assigned to the key. If no value is set, the key will be created with an empty value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLKeyExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLKeyExists(
>>>             name="Ensure service descriptor has stack",
>>>             path=Path("./service.yaml"),
>>>             key="stack",
>>>             value="my-awesome-stack",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** Compared to `hammurabi.rules.text.LineExists`, this rule is NOT able to add a key before or after a target.

## 4.10.2 YAMLKeyNotExists

**class** `hammurabi.rules.yaml.YAMLKeyNotExists` (*name: str, path: Optional[pathlib.Path] = None, key: str = "", \*\*kwargs*)

Ensure that the given key not exists. If needed, the rule will remove a key with the given name, including its value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLKeyNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLKeyNotExists(
>>>             name="Ensure outdated_key is removed",
>>>             path=Path("./service.yaml"),
>>>             key="outdated_key",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.10.3 YAMLKeyRenamed

```
class hammurabi.rules.yaml.YAMLKeyRenamed(name: str, path: Optional[pathlib.Path] =
                                           None, key: str = "", new_name: str = "",
                                           **kwargs)
```

Ensure that the given key is renamed. In case the key can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename they will fail otherwise.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLKeyRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLKeyRenamed(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.yaml"),
>>>             key="development.depends_on",
>>>             value="dependencies",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

### 4.10.4 YAMLValueExists

```
class hammurabi.rules.yaml.YAMLValueExists(name: str, path: Optional[pathlib.Path] =
                                           None, key: str = "", value: Union[None, list,
                                           dict, str, int, float] = None, **kwargs)
```

Ensure that the given key has the expected value(s). In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

This rule is special in the way that the value can be almost anything. For more information please read the warning below.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLValueExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLValueExists(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.yaml"),
>>>             key="development.dependencies",
>>>             value=["service1", "service2", "service3"],
>>>         ),
>>>     )
>>>     # Or
```

(continues on next page)

(continued from previous page)

```

>>>     YAMLValueExists(
>>>         name="Add infra alerting to existing alerting components",
>>>         path=Path("./service.yaml"),
>>>         key="development.alerting",
>>>         value={"infra": "#slack-channel-2"},
>>>     ),
>>>     # Or
>>>     YAMLValueExists(
>>>         name="Add support info",
>>>         path=Path("./service.yaml"),
>>>         key="development.supported",
>>>         value=True,
>>>     ),
>>>     # Or even
>>>     YAMLValueExists(
>>>         name="Make sure that no development branch is set",
>>>         path=Path("./service.yaml"),
>>>         key="development.branch",
>>>         value=None,
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**Warning:** Since the value can be anything from `None` to a list of lists, and rule piping passes the 1st argument (path) to the next rule the value parameter can not be defined in `__init__` before the path. Hence the value parameter must have a default value. The default value is set to `None`, which translates to the following:

Using the `YAMLValueExists` rule and not assigning value to value parameter will set the matching key's value to `None` by default in the document.

#### 4.10.5 YAMLValueNotExists

```

class hamurabi.rules.yaml.YAMLValueNotExists (name: str, path: Optional[pathlib.Path] =
                                              None, key: str = "", value: Union[str, int,
                                              float] = None, **kwargs)

```

Ensure that the key has no value given. In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

Compared to `hamurabi.rules.yaml.YAMLValueExists`, this rule can only accept simple value for its value parameter. No `list`, `dict`, or `None` can be used.

Based on the key's value's type if the value contains (or equals for simple types) value provided in the value parameter the value is:

1. Set to `None` (if the key's value's type is not a dict or list)
2. Removed from the list (if the key's value's type is a list)
3. Removed from the dict (if the key's value's type is a dict)

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLValueNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLValueNotExists(
>>>             name="Remove decommissioned service from dependencies",
>>>             path=Path("./service.yaml"),
>>>             key="development.dependencies",
>>>             value="service4",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```



## PRECONDITIONS

### 5.1 Base precondition

**class** `hammurabi.preconditions.base.Precondition` (*name: Optional[str] = None, param: Optional[Any] = None*)

This class which describes the bare minimum and helper functions for Preconditions. A precondition defines what and how should be checked/validated before executing a Rule. Since preconditions are special rules, all the functions available what can be used for `hammurabi.rules.base.AbstractRule`.

As said, preconditions are special from different angles. While this is not true for Rules, Preconditions will always have a name, hence giving a name to a Precondition is not necessary. In case no name given to a precondition, the name will be the name of the class and "precondition" suffix.

Example usage:

```
>>> import logging
>>> from typing import Optional
>>> from pathlib import Path
>>> from hammurabi import Precondition
>>>
>>> class IsFileExists(Precondition):
>>>     def __init__(self, path: Optional[Path] = None, **kwargs) -> None:
>>>         super().__init__(param=path, **kwargs)
>>>
>>>     def task(self) -> bool:
>>>         return self.param and self.param.exists()
```

#### Parameters

- **name** (*Optional[str]*) – Name of the rule which will be used for printing
- **param** (*Any*) – Input parameter of the rule will be used as `self.param`

### 5.2 Attributes

#### 5.2.1 IsOwnedBy

**class** `hammurabi.preconditions.attributes.IsOwnedBy` (*path: pathlib.Path, owner: str, \*\*kwargs*)

Check if the given file or directory has the required ownership.

To check only the user use `owner="username"`. To check only the group use `owner=":group_name"` (please note the colon :). It is also possible to check both username and group at the same time by using `owner="username:group_name"`.

Example usage:

```
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, Renamed, IsOwnedBy
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename pyproject.toml if owned by gabor",
>>>             path=Path("./pyproject.toml"),
>>>             new_name="gabor-pyproject.toml"
>>>             preconditions=[
>>>                 IsOwnedBy(path=Path("./pyproject.toml"), owner="gabor")
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

#### Parameters

- **path** (*Path*) – Input file's path
- **owner** (*str*) – Owner user and/or group of the file/directory separated by colon

### 5.2.2 IsNotOwnedBy

**class** hamurabi.preconditions.attributes.**IsNotOwnedBy** (*path: pathlib.Path, owner: str, \*\*kwargs*)  
Opposite of *hamurabi.preconditions.attributes.IsOwnedBy*.

### 5.2.3 HasMode

**class** hamurabi.preconditions.attributes.**HasMode** (*path: pathlib.Path, mode: int, \*\*kwargs*)

Check if the given file or directory has the required permissions/mode.

You can read more about the available modes at <https://docs.python.org/3/library/stat.html>.

Example usage:

```
>>> import stat
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, Renamed, HasMode
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
```

(continues on next page)



(continued from previous page)

```

>>> rules=(
>>>     Renamed(
>>>         name="Rename pyproject.toml if owned by gabor",
>>>         path=Path("./pyproject.toml"),
>>>         new_name="gabor-pyproject.toml"
>>>         preconditions=[
>>>             HasMode(path=Path("scripts/run_unittests.sh"), mode=stat.S_
↪ IXOTH)
>>>         ]
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

### Parameters

- **path** (*Path*) – Input file's path
- **mode** (*str*) – The desired mode to check

## 5.2.4 HasNoMode

**class** `hammurabi.preconditions.attributes.HasNoMode` (*path: pathlib.Path, mode: int, \*\*kwargs*)  
 Opposite of `hammurabi.preconditions.attributes.HasMode`.

## 5.3 Directories

### 5.3.1 IsDirectoryExists

**class** `hammurabi.preconditions.directories.IsDirectoryExists` (*path: pathlib.Path, \*\*kwargs*)

Check if the given directory exists.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed, IsDirectoryExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename the dir if an other one exists",
>>>             path=Path("old-name"),
>>>             new_name="new-name",
>>>             preconditions=[
>>>                 IsDirectoryExists(path=Path("other-dir"))
>>>             ]
>>>         ),
>>>     ),
>>> )

```

(continues on next page)

(continued from previous page)

```
>>> )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Parameters** `path` (*Path*) – Input directory's path

## 5.3.2 IsDirectoryNotExists

**class** `hammurabi.preconditions.directories.IsDirectoryNotExists` (*path: path-lib.Path, \*\*kwargs*)

Opposite of `hammurabi.preconditions.directories.IsDirectoryExists`.

## 5.4 Files

### 5.4.1 IsFileExists

**class** `hammurabi.preconditions.files.IsFileExists` (*path: pathlib.Path, \*\*kwargs*)

Check if the given file exists.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed, IsFileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename the file if an other one exists",
>>>             path=Path("old-name"),
>>>             new_name="new-name",
>>>             preconditions=[
>>>                 IsFileExists(path=Path("other-file"))
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Parameters** `path` (*Path*) – Input files's path

## 5.4.2 IsFileNotExists

**class** `hammurabi.preconditions.files.IsFileNotExists` (*path: pathlib.Path, \*\*kwargs*)  
 Opposite of `hammurabi.preconditions.files.IsFileExists`.

## 5.5 Text files

### 5.5.1 IsLineExists

**class** `hammurabi.preconditions.text.IsLineExists` (*path: pathlib.Path, criteria: str, \*\*kwargs*)

Check if the given line exists.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed, IsLineExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename the file if an other one exists",
>>>             path=Path("old-name"),
>>>             new_name="new-name",
>>>             preconditions=[
>>>                 IsLineExists(path=Path("other-file"), criteria=r"^string=some-
↪value$")
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

#### Parameters

- **path** (*Path*) – Input files's path
- **criteria** (*str*) – Regexp of the desired line

### 5.5.2 IsLineNotExists

**class** `hammurabi.preconditions.text.IsLineNotExists` (*path: pathlib.Path, criteria: str, \*\*kwargs*)

Opposite of `hammurabi.preconditions.text.IsLineExists`.



## REPORTERS

### 6.1 Base reporter

**class** `hammurabi.reporters.base.Reporter` (*laws: List[hammurabi.law.Law]*)

Abstract class which describes the bare minimum and helper functions for Reporters. A reporter can generate different outputs from the results of the execution. Also, reporters can be extended by additional data which may not contain data for every execution like GitHub pull request url. The report file's name set by `report_name` config parameter.

**Note:** Reporters measures the execution time for the complete execution from checking out the git branch until the pull request creation finished. Although the completion time is measured, it is not detailed for the rules. At this moment measuring execution time of rules is not planned.

Example usage:

```
>>> from hammurabi.reporters.base import Reporter
>>>
>>>
>>> class JSONReporter(Reporter):
>>>     def report(self) -> str:
>>>         return self._get_report().json()
```

**Parameters** `laws` (*Iterable[Law]*) – Iterable Law objects which will be included to the report

### 6.2 Formatted reporters

#### 6.2.1 JSONReporter

**class** `hammurabi.reporters.json.JSONReporter` (*laws: List[hammurabi.law.Law]*)

Generate reports in JSON format and write into file. JSONReporter is the default reporter of the pillar. The example below shows the way how to replace a reporter which could base on the JSONReporter.

The report will be written into the configured report file. The report file's name set by `report_name` config parameter.

Example usage:

```
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, OwnerChanged
>>> from my_company import MyJSONReporter
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OwnerChanged(
>>>             name="Change ownership of nginx config",
>>>             path=Path("./nginx.conf"),
>>>             new_value="www:web_admin"
>>>         ),
>>>     )
>>> )
>>>
>>> # override pillar's default JSONReporter reporter
>>> pillar = Pillar(reporter_class=MyJSONReporter)
```

## HAMMURABI

## 7.1 hammurabi package

### 7.1.1 Subpackages

## hammurabi.preconditions package

## Submodules

## hammurabi.preconditions.attributes module

This module contains the definition of Preconditions which are related to attributes of a file or directory.

This module contains the definition of Preconditions which are related to attributes of a file or directory.

```
class hammurabi.preconditions.attributes.HasMode(path: pathlib.Path, mode: int,  
                                                  **kwargs)
```

Bases: *hammurabi.preconditions.base.Precondition*

Check if the given file or directory has the required permissions/mode.

You can read more about the available modes at <https://docs.python.org/3/library/stat.html>.

Example usage:

```
>>> import stat
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, Renamed, HasMode
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename pyproject.toml if owned by gabor",
>>>             path=Path("./pyproject.toml"),
>>>             new_name="gabor-pyproject.toml"
>>>             preconditions=[
>>>                 HasMode(path=Path("scripts/run_unittests.sh"), mode=stat.S_
↳ IXOTH)
>>>             ],
>>>         ),
>>>     )
>>>
```

(continues on next page)

(continued from previous page)

```
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Parameters**

- **path** (*Path*) – Input file’s path
- **mode** (*str*) – The desired mode to check

**made\_changes****name****param****task** () → bool

Check if the given mode is set on the file or directory.

**Returns** Returns True if the desired mode is set**Return type** bool

```
class hammurabi.preconditions.attributes.HasNoMode (path:  pathlib.Path, mode:  int,
                                                    **kwargs)
```

Bases: *hammurabi.preconditions.attributes.HasMode*Opposite of *hammurabi.preconditions.attributes.HasMode*.**made\_changes****name****param****task** () → bool

Check if the given mode is not set on the file or directory.

**Returns** Returns True if the desired mode is not set**Return type** bool

```
class hammurabi.preconditions.attributes.IsNotOwnedBy (path:  pathlib.Path, owner:
                                                         str, **kwargs)
```

Bases: *hammurabi.preconditions.attributes.IsOwnedBy*Opposite of *hammurabi.preconditions.attributes.IsOwnedBy*.**made\_changes****name****param****task** () → bool

Check if the ownership does not meet the requirements.

**Returns** Returns True if the owner matches**Return type** bool

```
class hammurabi.preconditions.attributes.IsOwnedBy (path:  pathlib.Path, owner:  str,
                                                    **kwargs)
```

Bases: *hammurabi.preconditions.base.Precondition*

Check if the given file or directory has the required ownership.



To check only the user use `owner="username"`. To check only the group use `owner=":group_name"` (please note the colon `:`). It is also possible to check both username and group at the same time by using `owner="username:group_name"`.

Example usage:

```
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, Renamed, IsOwnedBy
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename pyproject.toml if owned by gabor",
>>>             path=Path("./pyproject.toml"),
>>>             new_name="gabor-pyproject.toml"
>>>             preconditions=[
>>>                 IsOwnedBy(path=Path("./pyproject.toml"), owner="gabor")
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

#### Parameters

- **path** (*Path*) – Input file's path
- **owner** (*str*) – Owner user and/or group of the file/directory separated by colon

#### made\_changes

**name**

**param**

**task** () → bool

Check if the ownership meets the requirements.

**Returns** Returns True if the owner matches

**Return type** bool

### hammurabi.preconditions.base module

This module contains the definition of Preconditions which describes what to do with the received parameter and does the necessary changes. The preconditions are used to enable developers skipping or enabling rules based on a set of conditions.

**Warning:** The precondition is for checking that a rule should or shouldn't run, not for breaking/aborting the execution. To indicate a precondition failure as an error in the logs, create a precondition which raises an exception if the requirements doesn't match.

```
class hammurabi.preconditions.base.Precondition(name: Optional[str] = None, param:
                                                Optional[Any] = None)
    Bases: hammurabi.rules.abstract.AbstractRule, abc.ABC
```

This class which describes the bare minimum and helper functions for Preconditions. A precondition defines what and how should be checked/validated before executing a Rule. Since preconditions are special rules, all the functions available what can be used for `hammurabi.rules.base.AbstractRule`.

As said, preconditions are special from different angles. While this is not true for Rules, Preconditions will always have a name, hence giving a name to a Precondition is not necessary. In case no name given to a precondition, the name will be the name of the class and "precondition" suffix.

Example usage:

```
>>> import logging
>>> from typing import Optional
>>> from pathlib import Path
>>> from hammurabi import Precondition
>>>
>>> class IsFileExists(Precondition):
>>>     def __init__(self, path: Optional[Path] = None, **kwargs) -> None:
>>>         super().__init__(param=path, **kwargs)
>>>
>>>     def task(self) -> bool:
>>>         return self.param and self.param.exists()
```

#### Parameters

- **name** (*Optional[str]*) – Name of the rule which will be used for printing
- **param** (*Any*) – Input parameter of the rule will be used as `self.param`

**execute()** → bool

Execute the precondition.

**Raise** `AssertionError`

**Returns** `None`

**made\_changes**

**name**

**param**

**abstract task()** → bool

Abstract method representing how a `hammurabi.rules.base.Precondition.task()` must be parameterized. Any difference in the parameters or return type will result in pylint/mypy errors.

To be able to use the power of `pipe` and `children`, return something which can be generally used for other rules as in input.

**Returns** Returns an output which can be used as an input for other rules

**Return type** `Any` (usually same as `self.param`'s type)

## hammurabi.preconditions.directories module

This module contains the definition of Preconditions which are related to directories.

**class** `hammurabi.preconditions.directories.IsDirectoryExists` (*path:* `pathlib.Path`,  
*\*\*kwargs*)

Bases: `hammurabi.preconditions.base.Precondition`

Check if the given directory exists.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed, IsDirectoryExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename the dir if an other one exists",
>>>             path=Path("old-name"),
>>>             new_name="new-name",
>>>             preconditions=[
>>>                 IsDirectoryExists(path=Path("other-dir"))
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Parameters** `path` (*Path*) – Input directory's path

**made\_changes**

**name**

**param**

**task** () → bool

Check if the given directory exists.

**Returns** Returns True if the directory exists

**Return type** bool

**class** `hammurabi.preconditions.directories.IsDirectoryNotExists` (*path:* `pathlib.Path`,  
*\*\*kwargs*)

Bases: `hammurabi.preconditions.directories.IsDirectoryExists`

Opposite of `hammurabi.preconditions.directories.IsDirectoryExists`.

**made\_changes**

**name**

**param**

**task()** → bool

Check if the given directory not exists.

**Returns** Returns True if the directory not exists

**Return type** bool

## hammurabi.preconditions.files module

Files preconditions module contains simple preconditions used for checking file existence.

**class** `hammurabi.preconditions.files.IsFileExists` (*path: pathlib.Path, \*\*kwargs*)

Bases: `hammurabi.preconditions.base.Precondition`

Check if the given file exists.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed, IsFileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename the file if an other one exists",
>>>             path=Path("old-name"),
>>>             new_name="new-name",
>>>             preconditions=[
>>>                 IsFileExists(path=Path("other-file"))
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Parameters** `path` (*Path*) – Input files's path

**made\_changes**

**name**

**param**

**task()** → bool

Check if the given file exists.

**Returns** Returns True if the file exists

**Return type** bool

**class** `hammurabi.preconditions.files.IsFileNotExists` (*path: pathlib.Path, \*\*kwargs*)

Bases: `hammurabi.preconditions.files.IsFileExists`

Opposite of `hammurabi.preconditions.files.IsFileExists`.

**made\_changes**

```
name
param
task () → bool
    Check if the given file not exists.

Returns Returns True if the file not exists
Return type bool
```

## hammurabi.preconditions.text module

This module contains the definition of Preconditions which are related to general text files.

```
class hammurabi.preconditions.text.IsLineExists (path:  pathlib.Path,  criteria:  str,
                                                **kwargs)
    Bases: hammurabi.preconditions.base.Precondition
```

Check if the given line exists.

Example usage:

```
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, Renamed, IsLineExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename the file if an other one exists",
>>>             path=Path("old-name"),
>>>             new_name="new-name",
>>>             preconditions=[
>>>                 IsLineExists(path=Path("other-file"), criteria=r"^string=some-
↵value$")
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

## Parameters

- **path** (*Path*) – Input files's path
- **criteria** (*str*) – Regexp of the desired line

**made\_changes**

name

param

**task** ( ) → bool  
Check if the given line exists.

**Returns** Returns True if the line exists

**Return type** bool

```
class hammurabi.preconditions.text.IsLineNotExists (path: pathlib.Path, criteria: str,  
                                                    **kwargs)  
    Bases: hammurabi.preconditions.text.IsLineExists  
    Opposite of hammurabi.preconditions.text.IsLineExists.  
    made_changes  
    name  
    param  
    task () → bool  
        Check if the given line not exists.  
        Returns Returns True if the line not exists  
        Return type bool
```

## Module contents

### hammurabi.reporters package

#### Submodules

#### hammurabi.reporters.base module

This module contains the definition of Reporters which is responsible for exposing the execution results in several formats.

```
class hammurabi.reporters.base.AdditionalData  
    Bases: pydantic.main.BaseModel  
    Additional data which may not be set for every execution.  
    finished: str = None  
    pull_request_url: str = None  
    started: str = None
```

```
class hammurabi.reporters.base.LawItem  
    Bases: pydantic.main.BaseModel  
    LawItem represents the basic summary of a low attached to a rule.  
    description: str = None  
    name: str = None
```

```
class hammurabi.reporters.base.Report  
    Bases: pydantic.main.BaseModel  
    The report object which contains all the necessary and optional data for the report will be generated.  
    additional_data: AdditionalData = None  
    failed: List[RuleItem] = None  
    passed: List[RuleItem] = None  
    skipped: List[RuleItem] = None
```

**class** `hammurabi.reporters.base.Reporter` (*laws: List[hammurabi.law.Law]*)

Bases: `abc.ABC`

Abstract class which describes the bare minimum and helper functions for Reporters. A reporter can generate different outputs from the results of the execution. Also, reporters can be extended by additional data which may not contain data for every execution like GitHub pull request url. The report file's name set by `report_name` config parameter.

**Note:** Reporters measures the execution time for the complete execution from checking out the git branch until the pull request creation finished. Although the completion time is measured, it is not detailed for the rules. At this moment measuring execution time of rules is not planned.

Example usage:

```
>>> from hammurabi.reporters.base import Reporter
>>>
>>> class JSONReporter(Reporter):
>>>     def report(self) -> str:
>>>         return self._get_report().json()
```

**Parameters** `laws` (*Iterable[Law]*) – Iterable Law objects which will be included to the report

**abstract** `report()` → Any

Do the actual reporting based on the report assembled.

**class** `hammurabi.reporters.base.RuleItem`

Bases: `pydantic.main.BaseModel`

RuleItem represents the registered rule and its status.

The rule (as normally) has the status of the execution which can be passed, failed or skipped.

**law:** `LawItem` = None

**name:** `str` = None

## hammurabi.reporters.json module

**class** `hammurabi.reporters.json.JSONReporter` (*laws: List[hammurabi.law.Law]*)

Bases: `hammurabi.reporters.base.Reporter`

Generate reports in JSON format and write into file. JSONReporter is the default reporter of the pillar. The example below shows the way how to replace a reporter which could base on the JSONReporter.

The report will be written into the configured report file. The report file's name set by `report_name` config parameter.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OwnerChanged
>>> from my_company import MyJSONReporter
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
```

(continues on next page)

(continued from previous page)

```
>>> rules=(
>>>     OwnerChanged(
>>>         name="Change ownership of nginx config",
>>>         path=Path("./nginx.conf"),
>>>         new_value="www:web_admin"
>>>     ),
>>> )
>>> )
>>>
>>> # override pillar's default JSONReporter reporter
>>> pillar = Pillar(reporter_class=MyJSONReporter)
```

**report ()** → None

Do the actual reporting based on the report assembled in JSON format. The report will be written into the configured report file.

## Module contents

### hammurabi.rules package

#### Submodules

#### hammurabi.rules.abstract module

This module contains the definition of the `AbstractRule` which describes what is shared between Rules and Preconditions.

**class** `hammurabi.rules.abstract.AbstractRule` (*name: str, param: Any*)

Bases: `abc.ABC`

Abstract class which describes the common behaviour for any kind of rule even it is a `hammurabi.rules.base.Rule` or `hammurabi.rules.base.Precondition`

#### Parameters

- **name** (*str*) – Name of the rule which will be used for printing
- **param** (*Any*) – Input parameter of the rule will be used as `self.param`

#### property description

Return the description of the `hammurabi.rules.base.Rule.task()` based on its docstring.

**Returns** Stripped description of `hammurabi.rules.base.Rule.task()`

**Return type** `str`

---

**Note:** As of this property returns the docstring of `hammurabi.rules.base.Rule.task()` method, it worth to take care of its description when initialized.

---

#### property documentation

Return the documentation of the rule based on its name, docstring and the description of its task.

**Returns** Concatenation of the rule's name, docstring, and task description

**Return type** `str`



---

**Note:** As of this method returns the name and docstring of the rule it worth to take care of its name and description when initialized.

---

**made\_changes**

**name**

**param**

**post\_task\_hook()**

Run code after the `hammurabi.rules.base.Rule.task()` has been performed. To access the parameter passed to the rule, always use `self.param` for `hammurabi.rules.base.Rule.post_task_hook()`.

---

**Note:** This method can be used for execution of git commands like git add, or double checking a modification made.

---

**Warning:** This method is not called in dry run mode.

**pre\_task\_hook()** → None

Run code before performing the `hammurabi.rules.base.Rule.task()`. To access the parameter passed to the rule, always use `self.param` for `hammurabi.rules.base.Rule.pre_task_hook()`.

**Warning:** This method is not called in dry run mode.

**abstract task()** → Any

Abstract method representing how a `hammurabi.rules.base.Rule.task()` or `hammurabi.preconditions.base.Precondition.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

To be able to use the power of `pipe` and `children`, return something which can be generally used for other rules as in input.

**Returns** Returns an output which can be used as an input for other rules

**Return type** Any (usually same as `self.param`'s type)

---

**Note:** Although it is a good practice to return the same type for the output that the input has, but this is not the case for “Boolean Rules”. “Boolean Rules” should return True (or truthy) or False (or falsy) values.

---

**static validate** (*val: Any, cast\_to: Optional[Any] = None, required=False*) → Any

Validate and/or cast the given value to another type. In case the existence of the value is required or casting failed an exception will be raised corresponding to the failure.

**Parameters**

- **val** (*Any*) – Value to validate
- **cast\_to** (*Any*) – Type in which the value should be returned
- **required** (*bool*) – Check that the value is not falsy

**Raise** `ValueError` if the given value is required but falsy

**Returns** Returns the value in its original or casted type

**Return type** Any

Example usage:

```
>>> from typing import Optional
>>> from pathlib import Path
>>> from hammurabi import Rule
>>>
>>> class MyAwesomeRule(Rule):
>>>     def __init__(self, name: str, param: Optional[Path] = None):
>>>         self.param = self.validate(param, required=True)
>>>
>>>     # Other method definitions ...
>>>
```

## hammurabi.rules.attributes module

Attributes module contains file and directory attribute manipulation rules which can be handy after creating new files or directories or even when adding execute permissions for a script in the project.

```
class hammurabi.rules.attributes.ModeChanged(name: str, path: Optional[pathlib.Path] =
                                             None, new_value: Optional[int] = None,
                                             **kwargs)
```

Bases: `hammurabi.rules.attributes.SingleAttributeRule`

Change the mode of a file or directory.

Supported modes:

Config option	Description
stat.S_ISUID	Set user ID on execution.
stat.S_ISGID	Set group ID on execution.
stat.S_ENFMT	Record locking enforced.
stat.S_ISVTX	Save text image after execution.
stat.S_IREAD	Read by owner.
stat.S_IWRITE	Write by owner.
stat.S_IEXEC	Execute by owner.
stat.S_IRWXU	Read, write, and execute by owner.
stat.S_IRUSR	Read by owner.
stat.S_IWUSR	Write by owner.
stat.S_IXUSR	Execute by owner.
stat.S_IRWXG	Read, write, and execute by group.
stat.S_IRGRP	Read by group.
stat.S_IWGRP	Write by group.
stat.S_IXGRP	Execute by group.
stat.S_IRWXO	Read, write, and execute by others.
stat.S_IROTH	Read by others.
stat.S_IWOTH	Write by others.
stat.S_IXOTH	Execute by others.

Example usage:

```

>>> import stat
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, ModeChanged
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         ModeChanged(
>>>             name="Update script must be executable",
>>>             path=Path("./scripts/update.sh"),
>>>             new_value=stat.S_IXGRP | stat.S_IXGRP | stat.S_IXOTH
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

#### made\_changes

**name**

**param**

**task()** → pathlib.Path

Change the mode of the given file or directory.

**Returns** Return the input path as an output

**Return type** Path

**class** hammurabi.rules.attributes.**OwnerChanged** (*name: str, path: Optional[pathlib.Path] = None, new\_value: Optional[str] = None, \*\*kwargs*)

Bases: *hammurabi.rules.attributes.SingleAttributeRule*

Change the ownership of a file or directory.

The new ownership of a file or directory can be set in three ways. To set only the user use `new_value="username"`. To set only the group use `new_value=":group_name"` (please note the colon :). It is also possible to set both username and group at the same time by using `new_value="username:group_name"`.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OwnerChanged
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OwnerChanged(
>>>             name="Change ownership of nginx config",
>>>             path=Path("./nginx.conf"),
>>>             new_value="www:web_admin"
>>>         ),
>>>     )
>>> )
>>>

```

(continues on next page)

(continued from previous page)

```
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes****name****param****task()** → `pathlib.Path`

Change the ownership of the given file or directory. None of the new username or group name can contain colons, otherwise only the first two colon separated values will be used as username and group name.

**Returns** Return the input path as an output

**Return type** `Path`

```
class hammurabi.rules.attributes.SingleAttributeRule(name: str, path: Optional[pathlib.Path] = None,
new_value: Optional[str] = None, **kwargs)
```

Bases: `hammurabi.rules.common.SinglePathRule`

Extend `hammurabi.rules.base.Rule` to handle attributes of a single file or directory.

**made\_changes****name****param****post\_task\_hook()**

Run code after the `hammurabi.rules.base.Rule.task()` has been performed. To access the parameter passed to the rule, always use `self.param` for `hammurabi.rules.base.Rule.post_task_hook()`.

---

**Note:** This method can be used for execution of git commands like git add, or double checking a modification made.

---

**Warning:** This method is not called in dry run mode.

**abstract task()** → `Any`

Abstract method representing how a `hammurabi.rules.base.Rule.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check `hammurabi.rules.base.Rule.task()`.

## hammurabi.rules.base module

This module contains the definition of Rule which describes what to do with the received parameter and does the necessary changes.

The Rule is an abstract class which describes all the required methods and parameters, but it can be extended and customized easily by inheriting from it. A good example for this kind of customization is `hammurabi.rules.text.LineExists` which adds more parameters to `hammurabi.rules.files.SingleFileRule` which inherits from `hammurabi.rules.base.Rule`.

```
class hammurabi.rules.base.Rule (name: str, param: Any, preconditions: Iterable[hammurabi.preconditions.base.Precondition] = (),
                                pipe: Optional[Rule] = None, children: Iterable[Rule] = ())
```

Bases: `hammurabi.rules.abstract.AbstractRule`, `abc.ABC`

Abstract class which describes the bare minimum and helper functions for Rules. A rule defines what and how should be executed. Since a rule can have piped and children rules, the “parent” rule is responsible for those executions. This kind of abstraction allows to run both piped and children rules sequentially in a given order.

Example usage:

```
>>> from typing import Optional
>>> from pathlib import Path
>>> from hammurabi import Rule
>>> from hammurabi.mixins import GitMixin
>>>
>>> class SingleFileRule(Rule, GitMixin):
>>>     def __init__(self, name: str, path: Optional[Path] = None, **kwargs) -> _
    ↪None:
>>>         super().__init__(name, path, **kwargs)
>>>
>>>     def post_task_hook(self):
>>>         self.git_add(self.param)
>>>
>>>     @abstractmethod
>>>     def task(self) -> Path:
>>>         pass
```

### Parameters

- **name** (*str*) – Name of the rule which will be used for printing
- **param** (*Any*) – Input parameter of the rule will be used as `self.param`
- **preconditions** (*Iterable["Rule"]*) – “Boolean Rules” which returns a truthy or falsy value
- **pipe** (*Optional["Rule"]*) – Pipe will be called when the rule is executed successfully
- **children** (*Iterable["Rule"]*) – Children will be executed after the piped rule if there is any

**Warning:** Preconditions can be used in several ways. The most common way is to run “Boolean Rules” which takes a parameter and returns a truthy or falsy value. In case of a falsy return, the precondition will fail and the rule will not be executed.

If any modification is done by any of the rules which are used as a precondition, those changes will be committed.

**property can\_proceed**

Evaluate if a rule can continue its execution. In case the execution is called with `dry_run` config option set to true, this method will always return `False` to make sure not performing any changes. If preconditions are set, those will be evaluated by this method.

**Returns** Return with the result of evaluation

**Return type** bool

**Warning:** `hammurabi.rules.base.Rule.can_proceed()` checks the result of `self.preconditions`, which means the preconditions are executed. Make sure that you are not doing any modifications within rules used as preconditions, otherwise take extra attention for those rules.

**execute** (*param: Optional[Any] = None*)

Execute the rule's task, its piped and children rules as well.

The execution order of task, piped rule and children rules described in but not by `hammurabi.rules.base.Rule.get_rule_chain()`.

**Parameters** **param** (*Optional[Any]*) – Input parameter of the rule given by the user

**Raise** `AssertionError`

**Returns** `None`

---

**Note:** The input parameter can be optional because of the piped and children rules which are receiving the output of its parent. In this case the user is not able to set the param manually, since it is calculated.

---

**Warning:** If `self.can_proceed` returns `False` the whole execution will be stopped immediately and `AssertionError` will be raised.

**get\_execution\_order** () → List[Union[Rule, hammurabi.preconditions.base.Precondition]]

Same as `hammurabi.rules.base.Rule.get_rule_chain()` but for the root rule.

**get\_rule\_chain** (*rule: Rule*) → List[Union[Rule, hammurabi.preconditions.base.Precondition]]

Get the execution chain of the given rule. The execution order is the following:

- task (current rule's `hammurabi.rules.base.Rule.task()`)
- Piped rule
- Children rules (in the order provided by the iterator used)

**Parameters** **rule** (*hammurabi.rules.base.Rule*) – The rule which execution chain should be returned

**Returns** Returns the list of rules in the order above

**Return type** List[Rule]

**made\_changes**

**name**

**param**

**abstract task()** → Any

See the documentation of `hammurabi.rules.base.AbstractRule.task()`

## hammurabi.rules.common module

```
class hammurabi.rules.common.MultiplePathRule(name: str, paths: Optional[Iterable[pathlib.Path]] = (),
                                              **kwargs)
```

Bases: `hammurabi.rules.base.Rule`, `hammurabi.mixins.GitMixin`

Abstract class which extends `hammurabi.rules.base.Rule` to handle operations on multiple files.

**made\_changes**

**name**

**param**

**post\_task\_hook()**

Run code after the `hammurabi.rules.base.Rule.task()` has been performed. To access the parameter passed to the rule, always use `self.param` for `hammurabi.rules.base.Rule.post_task_hook()`.

---

**Note:** This method can be used for execution of git commands like git add, or double checking a modification made.

---

**Warning:** This method is not called in dry run mode.

**abstract task()** → Any

Abstract method representing how a `hammurabi.rules.base.Rule.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check `hammurabi.rules.base.Rule.task()`.

```
class hammurabi.rules.common.SinglePathRule(name: str, path: Optional[pathlib.Path] = None, **kwargs)
```

Bases: `hammurabi.rules.base.Rule`, `hammurabi.mixins.GitMixin`

Abstract class which extends `hammurabi.rules.base.Rule` to handle operations on a single directory.

**made\_changes**

**name**

**param**

**post\_task\_hook()**

Run code after the `hammurabi.rules.base.Rule.task()` has been performed. To access the parameter passed to the rule, always use `self.param` for `hammurabi.rules.base.Rule.post_task_hook()`.

---

**Note:** This method can be used for execution of git commands like git add, or double checking a modification made.

---

**Warning:** This method is not called in dry run mode.

**abstract task()** → Any

Abstract method representing how a `hammurabi.rules.base.Rule.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check `hammurabi.rules.base.Rule.task()`.

## hammurabi.rules.dictionaries module

Extend `hammurabi.rules.base.Rule` to handle parsed content manipulations dictionaries. Standalone these rules are not useful, but they are very handy when files should be manipulated like YAML or JSON which will be parsed as dict.

These rules are intentionally not exported directly through `hammurabi` as it is done for YAML or JSON rules. The reason, as it is mentioned above, these rules are not standalone rules. Also, it is intentional that these rules are not represented in the documentation's [Rules section](#).

**class** `hammurabi.rules.dictionaries.DictKeyExists` (*name: str, path: Optional[`pathlib.Path`] = None, key: str = "", value: Union[None, list, dict, str, int, float] = None, \*\*kwargs*)

Bases: `hammurabi.rules.dictionaries.SinglePathDictParsedRule`, `abc.ABC`

Ensure that the given key exists. If needed, the rule will create a key with the given name, and optionally the specified value. In case the value is set, the value will be assigned to the key. If no value is set, the key will be created with an empty value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar
>>> from hammurabi.rules.dictionaries import DictKeyExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DictKeyExists(
>>>             name="Ensure service descriptor has stack",
>>>             path=Path("./service.dictionary"),
>>>             key="stack",
>>>             value="my-awesome-stack",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** Compared to `hammurabi.rules.text.LineExists`, this rule is NOT able to add a key before or after a target.

**made\_changes**



**name****param****task** () → pathlib.Path

Ensure that the given key exists in the parsed file. If needed, create the key with the given name, and optionally the specified value.

**Returns** Return the input path as an output

**Return type** Path

```
class hammurabi.rules.dictionaries.DictKeyNotExists(name: str, path: Optional[pathlib.Path] = None, key: str = "", loader: Callable[[Any], Dict[str, Any]] = <class 'dict'>, **kwargs)
```

Bases: *hammurabi.rules.dictionaries.SinglePathDictParsedRule*, *abc.ABC*

Ensure that the given key not exists. If needed, the rule will remove a key with the given name, including its value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar
>>> from hammurabi.rules.dictionaries import DictKeyNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DictKeyNotExists(
>>>             name="Ensure outdated_key is removed",
>>>             path=Path("./service.dictionary"),
>>>             key="outdated_key",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes****name****param****task** () → pathlib.Path

Ensure that the given key does not exists in the parsed file.

**Returns** Return the input path as an output

**Return type** Path

```
class hammurabi.rules.dictionaries.DictKeyRenamed(name: str, path: Optional[pathlib.Path] = None, key: str = "", new_name: str = "", **kwargs)
```

Bases: *hammurabi.rules.dictionaries.SinglePathDictParsedRule*, *abc.ABC*

Ensure that the given key is renamed. In case the key can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename they will fail otherwise.

Example usage:

```
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar
>>> from hamurabi.rules.dictionaries import DictKeyRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DictKeyRenamed(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.dictionary"),
>>>             key="development.depends_on",
>>>             value="dependencies",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

**task()** → `pathlib.Path`

Ensure that the given key is renamed. In case the key can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename they will fail otherwise.

**Raises** `LookupError` raised if no key can be renamed or both the new and old keys are in the config file

**Returns** Return the input path as an output

**Return type** `Path`

```
class hamurabi.rules.dictionaries.DictValueExists (name: str, path: Optional[pathlib.Path] = None,
key: str = "", value: Union[None, list, dict, str, int, float] = None,
**kwargs)
```

Bases: `hamurabi.rules.dictionaries.SinglePathDictParsedRule`, `abc.ABC`

Ensure that the given key has the expected value(s). In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

This rule is special in the way that the value can be almost anything. For more information please read the warning below.

Example usage:

```
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar
```

(continues on next page)

(continued from previous page)

```

>>> from hamurabi.rules.dictionaries import DictValueExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DictValueExists(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.dictionary"),
>>>             key="development.dependencies",
>>>             value=["service1", "service2", "service3"],
>>>         ),
>>>         # Or
>>>         DictValueExists(
>>>             name="Add infra alerting to existing alerting components",
>>>             path=Path("./service.dictionary"),
>>>             key="development.alerting",
>>>             value={"infra": "#slack-channel-2"},
>>>         ),
>>>         # Or
>>>         DictValueExists(
>>>             name="Add support info",
>>>             path=Path("./service.dictionary"),
>>>             key="development.supported",
>>>             value=True,
>>>         ),
>>>         # Or even
>>>         DictValueExists(
>>>             name="Make sure that no development branch is set",
>>>             path=Path("./service.dictionary"),
>>>             key="development.branch",
>>>             value=None,
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**Warning:** Since the value can be anything from None to a list of lists, and rule piping passes the 1st argument (path) to the next rule the value parameter can not be defined in `__init__` before the path. Hence the value parameter must have a default value. The default value is set to None, which translates to the following:

Using the DictValueExists rule and not assigning value to value parameter will set the matching key's value to None` by default in the document.

**made\_changes**

**name**

**param**

**task()** → pathlib.Path

Ensure that the given key has the expected value(s). In case the key cannot be found, a LookupError exception will be raised to stop the execution.

**Warning:** Since the value can be anything from `None` to a list of lists, and rule piping passes the 1st argument (`path`) to the next rule the `value` parameter can not be defined in `__init__` before the `path`. Hence the `value` parameter must have a default value. The default value is set to `None`, which translates to the following:

Using the `DictValueExists` rule and not assigning value to `value` parameter will set the matching key's value to `None` by default in the document.

**Raises** `LookupError` raised if no key can be renamed or both the new and old keys are in the config file

**Returns** Return the input path as an output

**Return type** `Path`

```
class hammurabi.rules.dictionaries.DictValueNotExists(name: str, path: Optional[pathlib.Path] = None,
key: str = "", value: Union[str, int, float] = None, **kwargs)
```

Bases: `hammurabi.rules.dictionaries.SinglePathDictParsedRule`, `abc.ABC`

Ensure that the key has no value given. In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

Compared to `hammurabi.rules.dictionaries.DictValueExists`, this rule can only accept simple value for its `value` parameter. No list, dict, or `None` can be used.

Based on the key's value's type if the value contains (or equals for simple types) value provided in the `value` parameter the value is:

1. Set to `None` (if the key's value's type is not a dict or list)
2. Removed from the list (if the key's value's type is a list)
3. Removed from the dict (if the key's value's type is a dict)

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar
>>> from hammurabi.rules.dictionaries import DictValueNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DictValueNotExists(
>>>             name="Remove decommissioned service from dependencies",
>>>             path=Path("./service.dictionary"),
>>>             key="development.dependencies",
>>>             value="service4",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name****param****task()** → `pathlib.Path`

Ensure that the key has no value given. In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

Based on the key's value's type if the value contains (or equals for simple types) value provided in the `value` parameter the value is: 1. Set to `None` (if the key's value's type is not a dict or list) 2. Removed from the list (if the key's value's type is a list) 3. Removed from the dict (if the key's value's type is a dict)

**Returns** Return the input path as an output

**Return type** `Path`

```
class hammurabi.rules.dictionaries.SinglePathDictParsedRule (name: str, path: Optional[pathlib.Path]
                                                                = None, key:
                                                                str = "", loader:
                                                                Callable[[Any],
                                                                Dict[str, Any]]
                                                                = <class 'dict'>,
                                                                **kwargs)

Bases:      hammurabi.rules.common.SinglePathRule,      hammurabi.rules.mixins.
SelectorMixin
```

Extend `hammurabi.rules.base.Rule` to handle parsed content manipulations dictionaries. Standalone this rule is not useful, but it is very handy when files should be manipulated like YAML or JSON which will be parsed as dict. This rule ensures that the implementation will be the same for these rules, so the maintenance cost and effort is reduced.

Although this rule is not that powerful on its own, we would not like to make it an abstract class like `hammurabi.rules.base.Rule` because it can easily happen that at some point this rule will be a standalone rule.

**made\_changes****name****param****pre\_task\_hook()** → `None`

Parse the file for later use.

**abstract task()** → `pathlib.Path`

Abstract method representing how a `hammurabi.rules.base.Rule.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check `hammurabi.rules.base.Rule.task()`.

## hammurabi.rules.directories module

Directories module contains directory specific manipulation rules. Please note that those rules which can be used for files and directories are located in other modules like *hammurabi.rules.operations* or *hammurabi.rules.attributes*.

**class** `hammurabi.rules.directories.DirectoryEmptied` (*name:* *str*, *path:* *Optional[pathlib.Path]* = *None*, *\*\*kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given directory's content is removed. Please note the difference between emptying a directory and recreating it. The latter results in lost ACLs, permissions and modes.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryEmptied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryEmptied(
>>>             name="Empty results directory",
>>>             path=Path("./test-results")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

**task()** → *pathlib.Path*

Iterate through the entries of the given directory and remove them. If an entry is a file simply remove it, otherwise remove the whole subdirectory and its content.

**Returns** Return the input path as an output

**Return type** *Path*

**class** `hammurabi.rules.directories.DirectoryExists` (*name:* *str*, *path:* *Optional[pathlib.Path]* = *None*, *\*\*kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that a directory exists. If the directory does not exists, make sure the directory is created.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
```

(continues on next page)

(continued from previous page)

```

>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryExists(
>>>             name="Create secrets directory",
>>>             path=Path("./secrets")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name****param****task()** → pathlib.Path

Create the given directory if not exists.

**Returns** Return the input path as an output**Return type** Path

```

class hammurabi.rules.directories.DirectoryNotExists(name: str, path: Optional[pathlib.Path] = None,
**kwargs)

```

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given directory does not exists.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, DirectoryNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         DirectoryNotExists(
>>>             name="Remove unnecessary directory",
>>>             path=Path("./temp")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name****param****post\_task\_hook()**

Remove the given directory from git index.

Remove the given directory.

**Return type** Path

```
class hammurabi.rules.files.FileEmptied(name: str, path: Optional[pathlib.Path] = None,
```

Bases: `hammurabi.rules.common.SinglePathRule`

Remove the content of the given file, but keep the file. Please

[illegible]

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileEmptied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileEmptied(
>>>             name="Empty the check log file",
>>>             path=Path("/var/log/service/check.log")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

11111

task /

Remove the content of the given file. If the file does not exists this rule will create the file without content.

**Return type:** `Path`

... mulca f...

Bases: *hammurabi.rules.common.SinglePathRule*

From the  $G1s$  state. If the  $G1s$  does not emit, make sure the



```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileExists(
>>>             name="Create service descriptor",
>>>             path=Path("./service.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes**

**name**

**param**

**task()** → pathlib.Path

If the target file not exists, create the file to make sure we can manipulate it.

**Returns** The created/existing file's path

**Return type** Path

**class** hammurabi.rules.files.**FileNotExists** (*name: str, path: Optional[pathlib.Path] = None, \*\*kwargs*)

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given file does not exists. If the file exists remove it, otherwise do nothing and return the original path.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileNotExists(
>>>             name="Remove unused file",
>>>             path=Path("./debug.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes**

**name**

**param**

**post\_task\_hook()**

Remove the given file from git index.

**task()** → `pathlib.Path`

Remove the given file if exists, otherwise do nothing and return the original path.

**Returns** Return the removed file's path

**Return type** `Path`

**class** `hammurabi.rules.files.FilesExist` (*name: str, paths: Optional[Iterable[pathlib.Path]] = (), \*\*kwargs*)

Bases: `hammurabi.rules.common.MultiplePathRule`

Ensure that all files exists. If the files does not exists, make sure the files are created.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FilesExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FilesExist(
>>>             name="Create test files",
>>>             paths=[
>>>                 Path("./file_1"),
>>>                 Path("./file_2"),
>>>                 Path("./file_3"),
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

**task()** → `Iterable[pathlib.Path]`

If the target files not exist, create the files to make sure we can manipulate them.

**Returns** The created/existing files' path

**Return type** `Iterable[Path]`

**class** `hammurabi.rules.files.FilesNotExist` (*name: str, paths: Optional[Iterable[pathlib.Path]] = (), \*\*kwargs*)

Bases: `hammurabi.rules.common.MultiplePathRule`

Ensure that the given files does not exist. If the files exist remove them, otherwise do nothing and return the original paths.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FilesNotExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FilesNotExist(
>>>             name="Remove several files",
>>>             paths=[
>>>                 Path("./file_1"),
>>>                 Path("./file_2"),
>>>                 Path("./file_3"),
>>>             ]
>>>         ),
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes**

**name**

**param**

**post\_task\_hook()**

Remove the given files from git index.

**task()** → Iterable[pathlib.Path]

Remove all existing files.

**Returns** Return the removed files' paths

**Return type** Iterable[Path]

## hammurabi.rules.ini module

Ini module is an extension for text rules tailor made for .ini/.cfg files. The main difference lies in the way it works. First, the .ini/.cfg file is parsed, then the modifications are made on the already parsed file.

```

class hammurabi.rules.ini.OptionRenamed(name: str, path: Optional[pathlib.Path] = None,
                                         option: Optional[str] = None, new_name: Op-
                                         tional[str] = None, **kwargs)

```

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Ensure that an option of a section is renamed.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(

```

(continues on next page)

(continued from previous page)

```

>>>         OptionRenamed(
>>>             name="Rename an option",
>>>             path=Path("./config.ini"),
>>>             section="my_section",
>>>             option="typo",
>>>             new_name="correct",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name****param****task()** → pathlib.Path

Rename an option of a section. In case a section can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if dependant rules will fail otherwise.

**Raises** `LookupError` raised if no section found or both the old and new option names are found

**Returns** Return the input path as an output

**Return type** Path

```

class hammurabi.rules.ini.OptionsExist(name: str, path: Optional[pathlib.Path] = None,
                                         options: Iterable[Tuple[str, Any]] = None,
                                         force_value: bool = False, **kwargs)

```

Bases: `hammurabi.rules.ini.SingleConfigFileRule`

Ensure that the given config option exists. If needed, the rule will create a config option with the given value. In case the `force_value` parameter is set to `True`, the original values will be replaced by the give ones.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionsExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionsExist(
>>>             name="Ensure options are changed",
>>>             path=Path("./config.ini"),
>>>             section="fetching",
>>>             options=(
>>>                 ("interval", "2s"),
>>>                 ("abort_on_error", True),
>>>             ),
>>>             force_value=True,
>>>         ),
>>>     )
>>> )

```

(continues on next page)

(continued from previous page)

```
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** When using the `force_value` parameter, please note that all the existing option values will be replaced by those set in `options` parameter.

**made\_changes**

**name**

**param**

**task()** → `pathlib.Path`

Remove one or more option from a section. In case a section can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if dependant rules will fail otherwise.

**Raises** `LookupError` raised if no section can be renamed

**Returns** Return the input path as an output

**Return type** `Path`

**class** `hammurabi.rules.ini.OptionsNotExist` (*name: str, path: Optional[`pathlib.Path`] = None, options: Iterable[str] = (), \*\*kwargs*)

Bases: `hammurabi.rules.ini.SingleConfigFileRule`

Remove one or more option from a section.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, OptionsNotExist
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         OptionsNotExist(
>>>             name="Ensure options are removed",
>>>             path=Path("./config.ini"),
>>>             section="invalid",
>>>             options=(
>>>                 "remove",
>>>                 "me",
>>>                 "please",
>>>             )
>>>         ),
>>>     ),
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

**task** () → pathlib.Path

Remove one or more option from a section. In case a section can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if dependant rules will fail otherwise.

**Raises** `LookupError` raised if no section can be renamed

**Returns** Return the input path as an output

**Return type** Path

```
class hammurabi.rules.ini.SectionExists(name: str, path: Optional[pathlib.Path] = None,
                                         target: Optional[str] = None, options: Iter-
                                         able[Tuple[str, Any]] = (), add_after: bool = True,
                                         **kwargs)
```

Bases: `hammurabi.rules.ini.SingleConfigFileRule`

Ensure that the given config section exists. If needed, the rule will create a config section with the given name, and optionally the specified options. In case options are set, the config options will be assigned to that config sections.

Similarly to `hammurabi.rules.text.LineExists`, this rule is able to add a section before or after a target section. The limitation compared to `LineExists` is that the `SectionExists` rule is only able to add the new entry exactly before or after its target.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionExists(
>>>             name="Ensure section exists",
>>>             path=Path("./config.ini"),
>>>             section="polling",
>>>             target="add_after_me",
>>>             options=(
>>>                 ("interval", "2s"),
>>>                 ("abort_on_error", True),
>>>             ),
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** When options parameter is set, make sure you are using an iterable tuple. The option keys must be strings, but there is no limitation for the value. It can be set to anything what the parser can handle. For more information on the parser, please visit the documentation of [configupdater](#).

**made\_changes**

**name****param****task** () → pathlib.Path

Ensure that the given config section exists. If needed, create a config section with the given name, and optionally the specified options.

**Returns** Return the input path as an output

**Return type** Path

```
class hammurabi.rules.ini.SectionNotExists(name: str, path: Optional[pathlib.Path]
                                           = None, section: Optional[str] = None,
                                           **kwargs)
```

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Make sure that the given file not contains the specified line. When a section removed, all the options belonging to it will be removed too.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionNotExists(
>>>             name="Ensure section removed",
>>>             path=Path("./config.ini"),
>>>             section="invalid",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes****name****param****task** () → pathlib.Path

Remove the given section including its options from the config file.

**Returns** Return the input path as an output

**Return type** Path

```
class hammurabi.rules.ini.SectionRenamed(name: str, path: Optional[pathlib.Path] = None,
                                          new_name: Optional[str] = None, **kwargs)
```

Bases: *hammurabi.rules.ini.SingleConfigFileRule*

Ensure that a section is renamed. None of its options will be changed.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, SectionRenamed
```

(continues on next page)

(continued from previous page)

```

>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         SectionRenamed(
>>>             name="Ensure section renamed",
>>>             path=Path("./config.ini"),
>>>             section="polling",
>>>             new_name="fetching",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name****param****task()** → `pathlib.Path`

Rename the given section to a new name. None of its options will be changed. In case a section can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename they will fail otherwise.

**Raises** `LookupError` if we can not decide or can not find what should be renamed

**Returns** Return the input path as an output

**Return type** `Path`

```

class hammurabi.rules.ini.SingleConfigFileRule (name: str, path: Optional[pathlib.Path]
                                                = None, section: Optional[str] = None,
                                                **kwargs)

```

Bases: `hammurabi.rules.common.SinglePathRule`

Extend `hammurabi.rules.base.Rule` to handle parsed content manipulations on a single file.

**made\_changes****name****param****pre\_task\_hook()** → `None`

Parse the configuration file for later use.

**abstract task()** → `Any`

Abstract method representing how a `hammurabi.rules.base.Rule.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check `hammurabi.rules.base.Rule.task()`.



## hammurabi.rules.json module

This module adds JSON file support. JSON module is an extension for text rules tailor made for .json files. The main difference lies in the way it works. First, the .json file is parsed, then the modifications are made on the already parsed file.

```
class hammurabi.rules.json.JSONKeyExists (name: str, path: Optional[pathlib.Path] = None,
                                           key: str = "", value: Union[None, list, dict, str, int,
                                                                    float] = None, **kwargs)

Bases:      hammurabi.rules.dictionaries.DictKeyExists, hammurabi.rules.json.
             SingleJSONFileRule
```

Ensure that the given key exists. If needed, the rule will create a key with the given name, and optionally the specified value. In case the value is set, the value will be assigned to the key. If no value is set, the key will be created with an empty value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONKeyExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONKeyExists(
>>>             name="Ensure service descriptor has stack",
>>>             path=Path("./service.json"),
>>>             key="stack",
>>>             value="my-awesome-stack",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** Compared to `hammurabi.rules.text.LineExists`, this rule is NOT able to add a key before or after a target.

**made\_changes**

**name**

**param**

```
class hammurabi.rules.json.JSONKeyNotExists (name: str, path: Optional[pathlib.Path] =
                                              None, key: str = "", **kwargs)

Bases:      hammurabi.rules.dictionaries.DictKeyNotExists, hammurabi.rules.json.
             SingleJSONFileRule
```

Ensure that the given key not exists. If needed, the rule will remove a key with the given name, including its value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONKeyNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONKeyNotExists(
>>>             name="Ensure outdated_key is removed",
>>>             path=Path("./service.json"),
>>>             key="outdated_key",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

```
class hammurabi.rules.json.JSONKeyRenamed(name: str, path: Optional[pathlib.Path] =
                                           None, key: str = "", new_name: str = "",
                                           **kwargs)
```

Bases: *hammurabi.rules.dictionaries.DictKeyRenamed*, *hammurabi.rules.json.SingleJSONFileRule*

Ensure that the given key is renamed. In case the key can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename they will fail otherwise.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONKeyRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONKeyRenamed(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.json"),
>>>             key="development.depends_on",
>>>             value="dependencies",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

```
class hammurabi.rules.json.JSONValueExists(name: str, path: Optional[pathlib.Path] =
                                           None, key: str = "", value: Union[None, list,
                                           dict, str, int, float] = None, **kwargs)

Bases: hammurabi.rules.dictionaries.DictValueExists, hammurabi.rules.json.
SingleJSONFileRule
```

Ensure that the given key has the expected value(s). In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

This rule is special in the way that the value can be almost anything. For more information please read the warning below.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONValueExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONValueExists(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.json"),
>>>             key="development.dependencies",
>>>             value=["service1", "service2", "service3"],
>>>         ),
>>>         # Or
>>>         JSONValueExists(
>>>             name="Add infra alerting to existing alerting components",
>>>             path=Path("./service.json"),
>>>             key="development.alerting",
>>>             value={"infra": "#slack-channel-2"},
>>>         ),
>>>         # Or
>>>         JSONValueExists(
>>>             name="Add support info",
>>>             path=Path("./service.json"),
>>>             key="development.supported",
>>>             value=True,
>>>         ),
>>>         # Or even
>>>         JSONValueExists(
>>>             name="Make sure that no development branch is set",
>>>             path=Path("./service.json"),
>>>             key="development.branch",
>>>             value=None,
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** Since the value can be anything from `None` to a list of lists, and rule piping passes the 1st argument (path) to the next rule the value parameter can not be defined in `__init__` before the path. Hence the value parameter must have a default value. The default value is set to `None`, which translates

to the following:

Using the `JSONValueExists` rule and not assigning value to value parameter will set the matching key's value to `None` by default in the document.

**made\_changes**

**name**

**param**

```
class hammurabi.rules.json.JSONValueNotExists (name: str, path: Optional[pathlib.Path] =
                                             None, key: str = "", value: Union[str, int,
                                             float] = None, **kwargs)

Bases: hammurabi.rules.dictionaries.DictValueNotExists, hammurabi.rules.json.
SingleJSONFileRule
```

Ensure that the key has no value given. In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

Compared to `hammurabi.rules.json.JSONValueExists`, this rule can only accept simple value for its value parameter. No list, dict, or `None` can be used.

Based on the key's value's type if the value contains (or equals for simple types) value provided in the value parameter the value is:

1. Set to `None` (if the key's value's type is not a dict or list)
2. Removed from the list (if the key's value's type is a list)
3. Removed from the dict (if the key's value's type is a dict)

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, JSONValueNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         JSONValueNotExists(
>>>             name="Remove decommissioned service from dependencies",
>>>             path=Path("./service.json"),
>>>             key="development.dependencies",
>>>             value="service4",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

```
class hammurabi.rules.json.SingleJSONFileRule (name: str, path: Optional[pathlib.Path] =
                                             None, key: str = "", **kwargs)

Bases: hammurabi.rules.dictionaries.SinglePathDictParsedRule
```

Extend `hammurabi.rules.dictionaries.SinglePathDictParsedRule` to handle parsed content manipulations on a single JSON file.

**made\_changes**

**name**

**param**

**abstract task()** → `pathlib.Path`

Abstract method representing how a `hammurabi.rules.base.Rule.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check `hammurabi.rules.base.Rule.task()`.

## hammurabi.rules.mixins module

**class** `hammurabi.rules.mixins.SelectorMixin`

Bases: `object`

This mixin contains the helper function to get a value from dict by a css selector like selector path. (example. `path.to.key`)

**get\_by\_selector** (*data: Any, key\_path: Union[str, List[str]]*) → `Dict[str, Any]`

Get a key's value by a selector and traverse the path.

### Parameters

- **data** (`hammurabi.rules.mixins.Any`) – The loaded YAML data into dict
- **key\_path** (`Union[str, List[str]]`) – Path to the key in a selector format (example. `path.to.the.key` or `["path", "to", "the", "key"]`)

**Returns** Return the value belonging to the selector

**Return type** `hammurabi.rules.mixins.Any`

**set\_by\_selector** (*loaded\_data: Any, key\_path: Union[str, List[str]], value: Union[None, list, dict, str, int, float], delete: bool = False*) → `Any`

Set a value by the key selector and traverse the path.

### Parameters

- **loaded\_data** (`hammurabi.rules.mixins.Any`) – The loaded YAML data into dict
- **key\_path** (`Union[str, List[str]]`) – Path to the key in a selector format (example. `path.to.the.key` or `["path", "to", "the", "key"]`)
- **value** (`Union[None, list, dict, str, int, float]`) – The value set for the key
- **delete** (`bool`) – Indicate if the key should be deleted

**Returns** The modified YAML data

**Return type** `hammurabi.rules.mixins.Any`

## hammurabi.rules.operations module

Operations module contains common file/directory operation which can be handy when need to move, rename or copy files.

```
class hammurabi.rules.operations.Copied(name: str, path: Optional[pathlib.Path] =  
                                         None, destination: Optional[pathlib.Path] = None,  
                                         **kwargs)
```

Bases: *hammurabi.rules.common.SinglePathRule*

Ensure that the given file or directory is copied to the new path.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Copied
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Copied(
>>>             name="Create backup file",
>>>             path=Path("./service.yaml"),
>>>             destination=Path("./service.bkp.yaml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

**post\_task\_hook()**

Add the destination and not the original path.

**task()** → `pathlib.Path`

Copy the given file or directory to a new place.

**Returns** Returns the path of the copied file/directory

**Return type** `Path`

```
class hammurabi.rules.operations.Moved(name: str, path: Optional[pathlib.Path] = None,  
                                         destination: Optional[pathlib.Path] = None,  
                                         **kwargs)
```

Bases: *hammurabi.rules.common.SinglePathRule*

Move a file or directory from “A” to “B”.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Moved
>>>
>>> example_law = Law(
```

(continues on next page)

(continued from previous page)

```

>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Moved(
>>>             name="Move pyproject.toml to its place",
>>>             path=Path("/tmp/generated/pyproject.toml.template"),
>>>             destination=Path("./pyproject.toml"), # Notice the rename!
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name****param****post\_task\_hook()**

Add both the new and old git objects.

**task()** → `pathlib.Path`

Move the given path to the destination. In case the file got a new name when destination is provided, the file/directory will be moved to its new place with its new name.

**Returns** Returns the new destination of the file/directory

**Return type** `Path`

**class** `hammurabi.rules.operations.Renamed` (*name: str, path: Optional[`pathlib.Path`] = None, new\_name: Optional[str] = None, \*\*kwargs*)

Bases: `hammurabi.rules.operations.Moved`

This rule is a shortcut for `hammurabi.rules.operations.Moved`. Instead of destination path a new name is required.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, Renamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         Renamed(
>>>             name="Rename pyproject.toml.bkp",
>>>             path=Path("/tmp/generated/pyproject.toml.bkp"),
>>>             new_name="pyproject.toml",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name**

**param**

## hammurabi.rules.templates module

Templates module contains rules which are capable to create a new file based on a Jinja2 template by rendering it.

```
class hammurabi.rules.templates.TemplateRendered(name: str, template: Optional[pathlib.Path] = None, destination: Optional[pathlib.Path] = None, context: Optional[Dict[str, Any]] = None, **kwargs)
```

Bases: `hammurabi.rules.common.SinglePathRule`

Render a file from a Jinja2 template. In case the destination file not exists, this rule will create it, otherwise the file will be overridden.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, TemplateRendered
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         TemplateRendered(
>>>             name="Create unicorn config from template",
>>>             template=Path("/tmp/templates/unicorn.conf.py"),
>>>             destination=Path("./unicorn.conf.py"),
>>>             context={
>>>                 "keepalive": 65
>>>             },
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

**post\_task\_hook()**

Add the destination and not the original path.

**task()** → `pathlib.Path`

Render a file from a Jinja2 template. In case the destination file not exists, this rule will create it, otherwise the file will be overridden.

**Returns** Returns the path of the rendered file

**Return type** `Path`



## hammurabi.rules.text module

Text module contains simple but powerful general file content manipulations. Combined with other simple rules like `hammurabi.rules.files.FileExists` or `hammurabi.rules.attributes.ModeChanged` almost anything can be achieved. Although any file's content can be changed using these rules, for common file formats like ini, yaml or json dedicated rules are created.

**class** `hammurabi.rules.text.LineExists` (*name: str, path: Optional[pathlib.Path] = None, text: Optional[str] = None, target: Optional[str] = None, position: int = 1, respect\_indentation: bool = True, ensure\_trailing\_newline: bool = False, \*\*kwargs*)

Bases: `hammurabi.rules.common.SinglePathRule`

Make sure that the given file contains the required line. This rule is capable for inserting the expected text before or after the unique target text respecting the indentation of its context.

The default behaviour is to insert the required text exactly after the target line, and respect its indentation. Please note that `text``` and ```target` parameters are required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineExists, IsLineNotExists
>>>
>>> gunicorn_config = Path("./gunicorn.conf.py")
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineExists(
>>>             name="Extend gunicorn config",
>>>             path=gunicorn_config,
>>>             text="keepalive = 65",
>>>             target=r"^bind.*",
>>>             preconditions=[
>>>                 IsLineNotExists(path=gunicorn_config, criteria=r"^keepalive.*
↪")
>>>             ]
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Note:** The indentation of the target text will be extracted by a simple regular expression. If a more complex regexp is required, please inherit from this class.

**made\_changes**

**name**

**param**

**task** () → `pathlib.Path`

Make sure that the given file contains the required line. This rule is capable for inserting the expected rule before or after the unique target text respecting the indentation of its context.

**Raises** LookupError

**Returns** Returns the path of the modified file

**Return type** Path

```
class hammurabi.rules.text.LineNotExists(name: str, path: Optional[pathlib.Path] = None,  
                                         text: Optional[str] = None, **kwargs)
```

Bases: *hammurabi.rules.common.SinglePathRule*

Make sure that the given file not contains the specified line.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineNotExists(
>>>             name="Remove keepalive",
>>>             path=Path("./unicorn.conf.py"),
>>>             text="keepalive = 65",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

**task** () → pathlib.Path

Make sure that the given file not contains the specified line.

**Returns** Returns the path of the modified file

**Return type** Path

```
class hammurabi.rules.text.LineReplaced(name: str, path: Optional[pathlib.Path] = None,  
                                         text: Optional[str] = None, target: Optional[str]  
                                         = None, respect_indentation: bool = True,  
                                         **kwargs)
```

Bases: *hammurabi.rules.common.SinglePathRule*

Make sure that the given text is replaced in the given file.

The default behaviour is to replace the required text with the exact same indentation that the target line has. This behaviour can be turned off by setting the `respect_indentation` parameter to False. Please note that text and target parameters are required.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, LineReplaced
>>>
>>> example_law = Law(
```

(continues on next page)

(continued from previous page)

```

>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         LineReplaced(
>>>             name="Replace typo using regex",
>>>             path=Path("./unicorn.conf.py"),
>>>             text="keepalive = 65",
>>>             target=r"^keepalive.*",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**Note:** The indentation of the target text will be extracted by a simple regular expression. If a more complex regexp is required, please inherit from this class.

**Warning:** This rule will replace all the matching lines in the given file. Make sure the given target regular expression is tested before the rule used against production code.

**made\_changes**

**name**

**param**

**task** () → pathlib.Path

Make sure that the given text is replaced in the given file.

**Raises** LookupError if we can not decide or can not find what should be replaced

**Returns** Returns the path of the modified file

**Return type** Path

## hammurabi.rules.yaml module

This module adds YAML file support. YAML module is an extension for text rules tailor made for .yaml/.yml files. The main difference lies in the way it works. First, the .yaml/.yml file is parsed, then the modifications are made on the already parsed file.

```

class hammurabi.rules.yaml.SingleDocumentYAMLFileRule(name: str, path: Optional[pathlib.Path] = None,
                                                         key: str = "", **kwargs)

```

Bases: *hammurabi.rules.dictionaries.SinglePathDictParsedRule*

Extend *hammurabi.rules.dictionaries.SinglePathDictParsedRule* to handle parsed content manipulations on a single YAML file.

**made\_changes**

**name**

**param**

**abstract task()** → `pathlib.Path`

Abstract method representing how a `hammurabi.rules.base.Rule.task()` must be parameterized. Any difference in the parameters will result in pylint/mypy errors.

For more details please check `hammurabi.rules.base.Rule.task()`.

```
class hammurabi.rules.yaml.YAMLKeyExists(name: str, path: Optional[pathlib.Path] = None,
                                         key: str = "", value: Union[None, list, dict, str, int,
                                                                float] = None, **kwargs)
```

Bases: `hammurabi.rules.dictionaries.DictKeyExists`, `hammurabi.rules.yaml.SingleDocumentYAMLFileRule`

Ensure that the given key exists. If needed, the rule will create a key with the given name, and optionally the specified value. In case the value is set, the value will be assigned to the key. If no value is set, the key will be created with an empty value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLKeyExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLKeyExists(
>>>             name="Ensure service descriptor has stack",
>>>             path=Path("./service.yaml"),
>>>             key="stack",
>>>             value="my-awesome-stack",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** Compared to `hammurabi.rules.text.LineExists`, this rule is NOT able to add a key before or after a target.

**made\_changes**

**name**

**param**

```
class hammurabi.rules.yaml.YAMLKeyNotExists(name: str, path: Optional[pathlib.Path] =
                                             None, key: str = "", **kwargs)
```

Bases: `hammurabi.rules.dictionaries.DictKeyNotExists`, `hammurabi.rules.yaml.SingleDocumentYAMLFileRule`

Ensure that the given key not exists. If needed, the rule will remove a key with the given name, including its value.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLKeyNotExists
```

(continues on next page)

(continued from previous page)

```

>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLKeyNotExists(
>>>             name="Ensure outdated_key is removed",
>>>             path=Path("./service.yaml"),
>>>             key="outdated_key",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name****param**

```

class hammurabi.rules.yaml.YAMLKeyRenamed(name: str, path: Optional[pathlib.Path] =
None, key: str = "", new_name: str = "",
**kwargs)

```

Bases: `hammurabi.rules.dictionaries.DictKeyRenamed`, `hammurabi.rules.yaml.SingleDocumentYAMLFileRule`

Ensure that the given key is renamed. In case the key can not be found, a `LookupError` exception will be raised to stop the execution. The execution must be stopped at this point, because if other rules depending on the rename they will fail otherwise.

Example usage:

```

>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLKeyRenamed
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLKeyRenamed(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.yaml"),
>>>             key="development.depends_on",
>>>             value="dependencies",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)

```

**made\_changes****name****param**

```
class hamurabi.rules.yaml.YAMLValueExists(name: str, path: Optional[pathlib.Path] =
                                          None, key: str = "", value: Union[None, list,
                                          dict, str, int, float] = None, **kwargs)

Bases: hamurabi.rules.dictionaries.DictValueExists, hamurabi.rules.yaml.
SingleDocumentYAMLFileRule
```

Ensure that the given key has the expected value(s). In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

This rule is special in the way that the value can be almost anything. For more information please read the warning below.

Example usage:

```
>>> from pathlib import Path
>>> from hamurabi import Law, Pillar, YAMLValueExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLValueExists(
>>>             name="Ensure service descriptor has dependencies",
>>>             path=Path("./service.yaml"),
>>>             key="development.dependencies",
>>>             value=["service1", "service2", "service3"],
>>>         ),
>>>         # Or
>>>         YAMLValueExists(
>>>             name="Add infra alerting to existing alerting components",
>>>             path=Path("./service.yaml"),
>>>             key="development.alerting",
>>>             value={"infra": "#slack-channel-2"},
>>>         ),
>>>         # Or
>>>         YAMLValueExists(
>>>             name="Add support info",
>>>             path=Path("./service.yaml"),
>>>             key="development.supported",
>>>             value=True,
>>>         ),
>>>         # Or even
>>>         YAMLValueExists(
>>>             name="Make sure that no development branch is set",
>>>             path=Path("./service.yaml"),
>>>             key="development.branch",
>>>             value=None,
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** Since the value can be anything from `None` to a list of lists, and rule piping passes the 1st argument (path) to the next rule the value parameter can not be defined in `__init__` before the path. Hence the value parameter must have a default value. The default value is set to `None`, which translates

to the following:

Using the `YAMLValueExists` rule and not assigning value to value parameter will set the matching key's value to `None` by default in the document.

**made\_changes**

**name**

**param**

```
class hammurabi.rules.yaml.YAMLValueNotExists (name: str, path: Optional[pathlib.Path] =
                                             None, key: str = "", value: Union[str, int,
                                             float] = None, **kwargs)

Bases: hammurabi.rules.dictionaries.DictValueNotExists, hammurabi.rules.yaml.
SingleDocumentYAMLFileRule
```

Ensure that the key has no value given. In case the key cannot be found, a `LookupError` exception will be raised to stop the execution.

Compared to `hammurabi.rules.yaml.YAMLValueExists`, this rule can only accept simple value for its value parameter. No list, dict, or `None` can be used.

Based on the key's value's type if the value contains (or equals for simple types) value provided in the value parameter the value is:

1. Set to `None` (if the key's value's type is not a dict or list)
2. Removed from the list (if the key's value's type is a list)
3. Removed from the dict (if the key's value's type is a dict)

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, YAMLValueNotExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         YAMLValueNotExists(
>>>             name="Remove decommissioned service from dependencies",
>>>             path=Path("./service.yaml"),
>>>             key="development.dependencies",
>>>             value="service4",
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**made\_changes**

**name**

**param**

## Module contents

### 7.1.2 Submodules

#### 7.1.3 hammurabi.config module

**class** hammurabi.config.CommonSettings

Bases: pydantic.env\_settings.BaseSettings

Common settings which applies to both TOML and CLI configuration of Hammurabi.

Pillar configuration is intentionally not listed since it is represented as a string in the TOML configuration, but used the parsed variable in the CLI configuration.

**class** Config

Bases: object

BaseSettings' config describing how the settings will be handled. The given `env_prefix` will make sure that settings can be read from environment variables starting with `HAMMURABI_`.

`env_prefix = 'hammurabi_'`

`dry_run: bool = None`

`git_base_name: str = None`

`git_branch_name: str = None`

`report_name: Path = None`

`repository: str = None`

`rule_can_abort: bool = None`

**class** hammurabi.config.Config

Bases: object

Simple configuration object which used across Hammurabi. The *Config* loads the given `pyproject.toml` according to PEP-518.

**Warning:** When trying to use GitHub based laws without an initialized GitHub client (or invalid token), a warning will be thrown at the beginning of the execution. In case a PR open is attempted, a `RuntimeError` will be raised

**load()**

Handle configuration loading from project toml file and make sure the configuration are initialized and merged. Also, make sure that logging is set properly. Before loading the configuration, it is a requirement to set the `HAMMURABI_SETTINGS_PATH` as it will contain the path to the `toml` file what Hammurabi expects. This is needed for cases when the 3rd party rules would like to read the configuration of Hammurabi.

... note:

The `HAMMURABI_SETTINGS_PATH` environment variable is set by the CLI by default, so there is no need to set if no 3rd party rules are used or those rules are not loading config.

**Raises** Runtime error if `HAMMURABI_SETTINGS_PATH` environment variable is not set or an invalid git repository was given.



```

class hammurabi.config.Settings
    Bases: hammurabi.config.CommonSettings

    CLI related settings which are directly needed for the execution.

    pillar: object = None

class hammurabi.config.TOMLSettings
    Bases: hammurabi.config.CommonSettings

    TOML Project configuration settings. Most of the fields are used to compose other configuration fields like
    github_token or pillar.

    github_token: str = None
    log_format: str = None
    log_level: str = None
    log_path: Optional[Path] = None
    pillar_config: Path = None
    pillar_name: str = None

```

## 7.1.4 hammurabi.exceptions module

```

exception hammurabi.exceptions.AbortLawError
    Bases: Exception

```

Custom exception to make sure that own exception types are caught by the Law's execution.

```

exception hammurabi.exceptions.PreconditionFailedError
    Bases: Exception

```

Custom exception representing a failed precondition. In case a precondition failed, there is no need to raise an error and report the rule as a failure. The precondition is for checking that a rule should or shouldn't run; not for breaking the execution.

## 7.1.5 hammurabi.helpers module

```

hammurabi.helpers.full_strip(value: str) → str
    Strip every line.

```

## 7.1.6 hammurabi.law module

This module contains the definition of Law which is responsible for the execution of its registered Rules. Every Law can have multiple rules to execute.

In case a rule raises an exception the execution may abort and none of the remaining rules will be executed neither pipes or children. An abort can cause an inconsistent state or a dirty git branch. If `rule_can_abort` config is set to True, the whole execution of the `:class:hammurabi.pillar.Pillar` will be aborted and the original exception will be re-raised.

```

class hammurabi.law.Law(name: str, description: str, rules: Iterable[hammurabi.rules.base.Rule],
                        preconditions: Iterable[hammurabi.preconditions.base.Precondition] =
                        ())
    Bases: hammurabi.mixins.GitMixin

```

A Law is a collection of Rules which is responsible for the rule execution and git committing.

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileExists(
>>>             name="Create pyproject.toml",
>>>             path=Path("./pyproject.toml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

#### **property can\_proceed**

Evaluate if the execution can be continued. If preconditions are set, those will be evaluated by this method.

**Returns** Return with the result of evaluation

**Return type** bool

**Warning:** `hammurabi.rules.base.Rule.can_proceed()` checks the result of `self.preconditions`, which means the preconditions are executed. Make sure that you are not doing any modifications within rules used as preconditions, otherwise take extra attention for those rules.

**commit()** → None

Commit the changes made by registered rules and add a meaningful commit message.

Example commit message:

```
Migrate to next generation project template
* Create pyproject.toml
* Add meta info from setup.py to pyproject.toml
* Add existing dependencies
* Remove requirements.txt
* Remove setup.py
```

#### **property documentation**

Get the name and description of the Law object.

**Returns** Return the name and description of the law as its documentation

**Return type** str

**enforce()** → None

Execute all registered rule. If `rule_can_abort` config option is set to `True`, all the rules will be aborted and an exception will be raised.

When the whole execution chain is finished, the changes will be committed except the failed ones.

---

**Note:** Failed rules and their chain (excluding prerequisites) will be added to the pull request description.

---

**Raises** `AbortLawError`

**property failed\_rules**

Return the rules which did modifications and failed.

**Returns** Return the failed rules

**Return type** `Union[Tuple[()], Tuple[Rule]]`

**get\_execution\_order** () → `List[Union[hammurabi.rules.base.Rule, hammurabi.preconditions.base.Precondition]]`

Get the execution order of the registered rules. The order will contain the pipes and children as well.

This helper function is useful in debugging and information gathering.

**Returns** Return the execution order of the rules

**Return type** `List[Rule]`

**property passed\_rules**

Return the rules which did modifications and not failed.

**Returns** Return the passed rules

**Return type** `Tuple[Rule, ..]`

**property skipped\_rules**

Return the rules which neither modified the code nor failed.

**Returns** Return the skipped rules

**Return type** `Tuple[Rule, ..]`

## 7.1.7 hammurabi.main module

## 7.1.8 hammurabi.mixins module

Mixins module contains helpers for both laws and rules. Usually this file will contain Git commands related helpers. Also, this module contains the extensions for several online git based VCS.

**class** `hammurabi.mixins.GitHubMixin`

Bases: `hammurabi.mixins.GitMixin`, `hammurabi.mixins.PullRequestHelperMixin`

Extending `hammurabi.mixins.GitMixin` to be able to open pull requests on GitHub after changes are pushed to remote.

**create\_pull\_request** () → `Optional[str]`

Create a PR on GitHub after the changes are pushed to remote. The pull request details (repository, branch) are set by the project configuration. The mapping of the details and configs:

Detail	Configuration
repo	repository (owner/repository format)
base	git_base_name
branch	git_branch_name

**Returns** Return the open (and updated) or opened PR's url

**Return type** `Optional[str]`

**class** `hammurabi.mixins.GitMixin`

Bases: `object`

Simple mixin which contains all the common git commands which are needed to push a change to an online VCS like GitHub or GitLab. This mixin could be used by `hammurabi.law.Law`s`, `:class:`hammurabi.rules.base` or any rules which can make modifications during its execution.

**static** `checkout_branch()` → `None`

Perform a simple git checkout, to not pollute the default branch and use that branch for the pull request later. The branch name can be changed in the config by setting the `git_branch_name` config option.

The following command is executed:

```
git checkout -b <branch name>
```

**git\_add** (*param: `pathlib.Path`*) → `None`

Add file contents to the index.

**Parameters** `param` (*Path*) – Path to add to the index

The following command is executed:

```
git add <path>
```

**git\_commit** (*message: str*) → `None`

Commit the changes on the checked out branch.

**Parameters** `message` (*str*) – Git commit message

The following command is executed:

```
git commit -m "<commit message>"
```

**git\_remove** (*param: `pathlib.Path`*) → `None`

Remove files from the working tree and from the index.

**Parameters** `param` (*Path*) – Path to remove from the working tree and the index

The following command is executed:

```
git rm <path>
```

**static** `push_changes()` → `None`

Push the changes with the given branch set by `git_branch_name` config option to the remote origin.

The following command is executed:

```
git push origin <branch name>
```

**class** `hammurabi.mixins.PullRequestHelperMixin`

Bases: `object`

Give helper classes for pull request related operations

**generate\_pull\_request\_body** (*pillar*) → `str`

Generate the body of the pull request based on the registered laws and rules. The pull request body is markdown formatted.

**Parameters** `pillar` (*`hammurabi.pillar.Pillar`*) – Pillar configuration

**Returns** Returns the generated pull request description

**Return type** `str`

### 7.1.9 hammurabi.pillar module

Pillar module is responsible for handling the whole execution chain including executing the registered laws, pushing the changes to the VCS and creating a pull request. All the laws registered to the pillar will be executed in the order of the registration.

```
class hammurabi.pillar.Pillar (reporter_class:  Type[hammurabi.reporters.base.Reporter]  =
                                <class 'hammurabi.reporters.json.JSONReporter'>)
    Bases: hammurabi.mixins.GitHubMixin
```

Pillar is responsible for the execution of the chain of laws and rules.

All the registered laws and rules can be retrieved using the `laws` and `rules` properties, or if necessary single laws and rules can be accessed using the resource's name as a parameter for `get_law` or `get_rule` methods.

As a final step, pillar will prepare its reporter for report generation. For more information about reporters, check *hammurabi.reporters.base.Reporter* and *hammurabi.reporters.json.JSONReporter*.

**Parameters** `reporter_class` (*Type[Reporter]*) – The reporter class used for generating the reports

**enforce** ()

Run all the registered laws and rules one by one. This method is responsible for executing the registered laws, push changes to the git origin and open the pull request.

This method glues together the lower level components and makes sure that the execution of laws and rules can not be called more than once at the same time for a target.

**get\_law** (*name: str*) → *hammurabi.law.Law*

Get a law by its name. In case of no Laws are registered or the law can not be found by its name, a `StopIteration` exception will be raised.

**Parameters** `name` (*str*) – Name of the law which will be used for the lookup

**Raises** `StopIteration` exception if Law not found

**Returns** Return the searched law

**Return type** *hammurabi.law.Law*

**get\_rule** (*name: str*) → *hammurabi.rules.base.Rule*

Get a registered rule (and its pipe/children) by the rule's name.

This helper function is useful in debugging and information gathering.

**Parameters** `name` (*str*) – Name of the rule which will be used for the lookup

**Raises** `StopIteration` exception if Rule not found

**Returns** Return the rule in case of a match for the name

**Return type** *Rule*

**property laws**

Return the registered laws in order of the registration.

**register** (*law: hammurabi.law.Law*)

Register the given Law to the Pillar. The order of the registration does not matter. The laws should never depend on each other.

**Parameters** `law` (*hammurabi.law.Law*) – Initialized Law which should be registered

Example usage:

```
>>> from pathlib import Path
>>> from hammurabi import Law, Pillar, FileExists
>>>
>>> example_law = Law(
>>>     name="Name of the law",
>>>     description="Well detailed description what this law does.",
>>>     rules=(
>>>         FileExists(
>>>             name="Create pyproject.toml",
>>>             path=Path("./pyproject.toml")
>>>         ),
>>>     )
>>> )
>>>
>>> pillar = Pillar()
>>> pillar.register(example_law)
```

**Warning:** The laws should never depend on each other, because the execution may not happen in the same order the laws were registered. Instead, organize the depending rules in one law to resolve any dependency conflicts.

### property rules

Return all the registered laws' rules.

## 7.1.10 Module contents

## CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

### 8.1 Types of Contributions

#### 8.1.1 Report Bugs

Report bugs at <https://github.com/gabor-boros/hammurabi/issues>.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

#### 8.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

#### 8.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it. In case you added a new Rule or Precondition, do not forget to add them to the docs as well.

#### 8.1.4 Write Documentation

Hammurabi could always use more documentation, whether as part of the official Hammurabi docs, in docstrings, or even on the web in blog posts, articles, and such.

### 8.1.5 Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/gabor-boros/hammurabi/issues>.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 8.2 Get Started!

Ready to contribute? Here's how to set up *hammurabi* for local development.

1. Fork the *hammurabi* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/hammurabi.git
```

3. Install your local copy. Assuming you have poetry installed, this is how you set up your fork for local development:

```
$ cd hammurabi/  
$ poetry install
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass linters and the tests:

```
$ poetry shell  
$ make lint  
$ make test
```

You will need make not just for executing the command, but to build (and test) the documentations page as well.

6. Commit your changes and push your branch to GitHub:

```
$ git add .  
$ git commit -m "Your detailed description of your changes."  
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.



## 8.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.7 and 3.8.

## 8.4 Releasing

A reminder for the maintainers on how to release. Make sure all your changes are committed (including an entry in CHANGELOG.rst).

After all, create a tag and a release on GitHub. The rest will be handled by Travis.

Please follow this checklist for the release:

1. Make sure that formatters are not complaining (`make format` returns 0)
2. Make sure that linters are not complaining (`make lint` returns 0)
3. Update CHANGELOG.rst - do not forget to update the unreleased link comparison
4. Update version in `pyproject.toml`, `CHANGELOG.rst`, `docs/conf.py` and `hammurabi/__init__.py`
5. Create a new Release on GitHub with a detailed release description based on the previous releases.



## **VULNERABILITIES**

---

**Note:** Important! In case you found vulnerability or security issue in one of the libraries we use or somewhere else in the code, please contact us via e-mail at [gabor.brs@gmail.com](mailto:gabor.brs@gmail.com). Please do not use this channel for support.

---

### **9.1 Reporting vulnerabilities**

#### **9.1.1 What is vulnerability?**

Vulnerability is a cyber-security term that refers to a flaw in a system that can leave it open to attack. The vulnerability may also refer to any type of weakness in a computer system itself, in a set of procedures, or in anything that leaves information security exposed to a threat. - by [techopedia](#)

#### **9.1.2 In case you found a vulnerability**

In case you found vulnerability or security issue in one of the libraries we use or somewhere else in the code, please do not publish it, instead, contact us via e-mail at [gabor.brs@gmail.com](mailto:gabor.brs@gmail.com). We will take the necessary steps to fix the issue. We are handling the vulnerabilities privately.

To make report processing easier, please consider the following:

- Use clear and expressive subject
- Have a short, clear, and direct description including the details
- Include OWASP link, CVE references or links to other public advisories and standards
- Add steps on how to reproduce the issue
- Describe your environment
- Attach screenshots if applicable

---

**Note:** This [article](#) is a pretty good resource on how to report vulnerabilities.

---

In case you have any further questions regarding vulnerability reporting, feel free to open an [issue](#) on GitHub.



## CREDITS

### 10.1 Development Lead

- Gábor Boros (@gabor-boros)

### 10.2 Contributors

Special thanks to Péter Turi (@turip) for the initial idea.

Check the whole list of contributors [here](#).



## CHANGELOG

All notable changes to this project will be documented in this file. The format is based on [Keep a Changelog](#), and this project adheres to [Semantic Versioning](#).

### 11.1 Unreleased

#### 11.2 0.7.3 - 2020-05-25

##### 11.2.1 Fixed

- Fix updating existing pull request issue pt. 3

#### 11.3 0.7.2 - 2020-05-25

##### 11.3.1 Fixed

- Fix updating existing pull request issue pt. 2

#### 11.4 0.7.1 - 2020-05-22

##### 11.4.1 Fixed

- Fix recursive directory removal issue
- Fix updating existing pull request issue
- Fix wrong default value in config documentation

### 11.4.2 Changed

- Bump hypothesis to 5.15.1
- Bump toml to 0.10.1
- Bump flake8 to 3.8.1
- Bump pylint to 2.5.2

## 11.5 0.7.0 - 2020-04-28

### 11.5.1 Added

- Implement `__repr` and `__str__` for `Law`, `Rule` and `Precondition` objects
- Add logging related configuration options to customize logging
- Add dictionary parsed rules as a base for YAML and JSON rules
- Extend the documentations by the new dictionary rules
- Add community discord link

### 11.5.2 Changed

- Unify log message styles
- Adjust logging levels
- Use dictionary parsed rules as a base for YAML and JSON rules
- Reduced the method complexity of `DictValueExists` and `DictValueNotExists` rules
- Reduced the method complexity of `Rule` execution
- Reduced the method complexity of `Law` execution
- Reduced the method complexity of `LineExists` task execution
- Reduced the method complexity of `SectionExists` task execution
- Improve `LineExists` rule to make sure text can be added at the end of file even the file has no trailing newline
- Bump click to 7.1.2
- Bump pylint to 2.5.0
- Bump pydantic to 1.5.1
- Bump hypothesis to 5.10.4
- Bump jinja2 to 2.11.2
- Bump coverage to 5.1
- Bump gitpython to 3.1.1



### 11.5.3 Removed

- Remove `criteria` fields since Hammurabi now supports preconditions and it breaks the API uniformity

## 11.6 0.6.0 - 2020-04-06

### 11.6.1 Added

- New precondition `IsOwnedBy / IsNotOwnedBy`
- New precondition `HasMode / HasNoMode`
- New precondition `IsDirectoryExists / IsDirectoryNotExists`
- New precondition `IsFileExists / IsFileNotExists`
- New precondition `IsLineExists / IsLineNotExists`
- Add preconditions for `Law` class
- Add JSON file support

### 11.6.2 Changed

- Added return value type hint to `pre_task_hook`
- `_get_by_selector / _set_by_selector` became `public methods (get_by_selector / set_by_selector)`

## 11.7 0.5.0 - 2020-03-31

### 11.7.1 Fixed

- Add untracked files as well to the index

### 11.7.2 Removed

- Remove lock file creation since it is useless

## 11.8 0.4.0 - 2020-03-31

### 11.8.1 Added

- Added `Reporter` and `JSONReporter` classes to be able to expose execution results
- Add new config option `report_name` to the available settings
- New exception type `PreconditionFailedError` indicating that the precondition failed and no need to raise an error

## 11.8.2 Changed

- Make sure children and pipe can be set at the same time
- Simplify yaml key rename logic
- `SectionRenamed` not raises error if old section name is not represented but the new one
- `OptionRenamed` not raises error if old option name is not represented but the new one
- `LineReplaced` not raises error if old line is not represented but the new one
- Remove redundant way of getting rules of a law (<https://github.com/gabor-boros/hammurabi/issues/45>)
- GitHub mixin now returns the URL of the open PR's URL; if an existing PR found, that PR's URL will be returned
- Pillar prepare its Reporter for report generation
- Pillar has a new argument to set the pillar's reporter easily
- CLI's enforce command now calls the Pillar's prepared Reporter to do the report
- "No changes made by" messages now info logs instead of warnings
- Commit changes only if the Law has passing rules
- If `PreconditionFailedError` raised, do not log error messages, log a warning instead
- `LineExists` will not raise an exception if multiple targets found, instead it will select the last match as target
- Have better PR description formatting

## 11.8.3 Fixed

- Fixed a dictionary traversal issue regarding yaml file support
- Fixed "Failed Rules" formatting of PR description by removing `\xa0` character
- Fixed no Rule name in PR description if the Law did not change anything issue
- Fixed nested rule indentation PR description markup
- Fixed an issue with `LineReplaced`, if the input file is empty, raise an exception

## 11.9 0.3.1 - 2020-03-26

### 11.9.1 Fixed

- Make sure the lost ini file fix is back lost by merge conflict resolution

## 11.10 0.3.0 - 2020-03-25

### 11.10.1 Added

- Add Yaml file support (<https://github.com/gabor-boros/hammurabi/pull/24>)

### 11.10.2 Changed

- Make sure `SectionExists` adds the section even if no target given (<https://github.com/gabor-boros/hammurabi/pull/21>)
- Apply PEP-561 (<https://github.com/gabor-boros/hammurabi/pull/19>)

### 11.10.3 Fixed

- Fixed an ini section rename issue (<https://github.com/gabor-boros/hammurabi/pull/24>)

### 11.10.4 Removed

- Updated CONTRIBUTING.rst to remove the outdated stub generation

## 11.11 0.2.0 - 2020-03-23

### 11.11.1 Added

- Render files from Jinja2 templates (`TemplateRendered` rule)
- Add new `Precondition` base class (<https://github.com/gabor-boros/hammurabi/pull/9>)
- Add Code of Conduct to meet community requirements (<https://github.com/gabor-boros/hammurabi/pull/10>)
- New section in the documentations for Rules and Preconditions (<https://github.com/gabor-boros/hammurabi/pull/11>)
- Collect failed rules for every law (`Law.failed_rules`) (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Add chained rules to PR body (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Add failed rules to PR body (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Throw a warning when no GitHub client is initialized (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Raise runtime error when no GitHub client is initialized, but PR creation called (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Guess owner/repository based on the origin url of the working directory (<https://github.com/gabor-boros/hammurabi/pull/13>)

### 11.11.2 Changed

- Add stub formatting to Makefile's *stubs* command
- Extract common methods of `Precondition` and `Rule` to a new `AbstractRule` class (<https://github.com/gabor-boros/hammurabi/pull/9>)
- Extended CONTRIBUTING guidelines to include a notice for adding Rules and Preconditions (<https://github.com/gabor-boros/hammurabi/pull/11>)
- Refactor package structure and extract preconditions to separate submodule (<https://github.com/gabor-boros/hammurabi/pull/11>)
- Pull request body generation moved to the common `GitMixin` class (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Pillar will always create lock file in the working directory (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Call `expandvar` and `expanduser` of configuration files (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Hammurabi only works in the current working directory (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Read settings (`pyproject.toml`) path from `HAMMURABI_SETTINGS_PATH` environment variable (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Fix version handling in docs

### 11.11.3 Fixed

- Remove faulty author of git committing (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Only attempt to create a PR if there is no PR from Hammurabi (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Fix double committing issue (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Fix committing of laws when nothing changed (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Fixed several CLI arguments related issues (<https://github.com/gabor-boros/hammurabi/pull/13>)
- Fixed a typo in the Bug issue template of GitHub (<https://github.com/gabor-boros/hammurabi/pull/13>)

### 11.11.4 Removed

- Removed target directory setting from config and CLI (<https://github.com/gabor-boros/hammurabi/pull/13>)

## 11.12 0.1.2 - 2020-03-18

### 11.12.1 Changed

- Extended Makefile to generate stubs
- Extend documentation how to generate and update stubs
- Update how to release section of CONTRIBUTING.rst

## 11.13 0.1.1 - 2020-03-17

### 11.13.1 Changed

- Moved unreleased section of CHANGELOG to the top
- Updated changelog entries to contain links for release versions
- Updated CONTRIBUTING document to mention changelog links
- Refactored configuration handling (<https://github.com/gabor-boros/hammurabi/pull/5>)

### 11.13.2 Fixed

- Fixed wrong custom rule example in the README
- Smaller issues around git committing and pushing (<https://github.com/gabor-boros/hammurabi/pull/5>)

## 11.14 0.1.0 - 2020-03-12

### 11.14.1 Added

- **Basic file manipulations**
  - Create file
  - Create files
  - Remove file
  - Remove files
  - Empty file
- **Basic directory manipulations**
  - Create directory
  - Remove directory
  - Empty directory
- **Basic file and directory operations**
  - Change owner
  - Change mode
  - Move file or directory
  - Copy file or directory
  - Rename file or directory
- **Plain text/general file manipulations**
  - Add line
  - Remove line
  - Replace line

- **INI file specific manipulations**

- Add section
- Remove section
- Rename section
- Add option
- Remove option
- Rename option

- **Miscellaneous**

- Initial documentation
- CI/CD integration

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`





## PYTHON MODULE INDEX

### h

- hammurabi, 98
- hammurabi.config, 92
- hammurabi.exceptions, 93
- hammurabi.helpers, 93
- hammurabi.law, 93
- hammurabi.main, 95
- hammurabi.mixins, 95
- hammurabi.pillar, 97
- hammurabi.preconditions, 50
- hammurabi.preconditions.attributes, 43
- hammurabi.preconditions.base, 45
- hammurabi.preconditions.directories, 47
- hammurabi.preconditions.files, 48
- hammurabi.preconditions.text, 49
- hammurabi.reporters, 52
- hammurabi.reporters.base, 50
- hammurabi.reporters.json, 51
- hammurabi.rules, 92
- hammurabi.rules.abstract, 52
- hammurabi.rules.attributes, 54
- hammurabi.rules.base, 57
- hammurabi.rules.common, 59
- hammurabi.rules.dictionaries, 60
- hammurabi.rules.directories, 66
- hammurabi.rules.files, 68
- hammurabi.rules.ini, 71
- hammurabi.rules.json, 77
- hammurabi.rules.mixins, 81
- hammurabi.rules.operations, 82
- hammurabi.rules.templates, 84
- hammurabi.rules.text, 85
- hammurabi.rules.yaml, 87



## A

AbortLawError, 93  
 AbstractRule (class in *hammurabi.rules.abstract*), 52  
 additional\_data (hammurabi.reporters.base.Report attribute), 50  
 AdditionalData (class in *hammurabi.reporters.base*), 50

## C

can\_proceed() (*hammurabi.law.Law* property), 94  
 can\_proceed() (*hammurabi.rules.base.Rule* property), 57  
 checkout\_branch() (*hammurabi.mixins.GitMixin* static method), 96  
 commit() (*hammurabi.law.Law* method), 94  
 CommonSettings (class in *hammurabi.config*), 92  
 CommonSettings.Config (class in *hammurabi.config*), 92  
 Config (class in *hammurabi.config*), 92  
 Copied (class in *hammurabi.rules.operations*), 82  
 create\_pull\_request() (*hammurabi.mixins.GitHubMixin* method), 95

## D

description (*hammurabi.reporters.base.LawItem* attribute), 50  
 description() (*hammurabi.rules.abstract.AbstractRule* property), 52  
 DictKeyExists (class in *hammurabi.rules.dictionaries*), 60  
 DictKeyNotExists (class in *hammurabi.rules.dictionaries*), 61  
 DictKeyRenamed (class in *hammurabi.rules.dictionaries*), 61  
 DictValueExists (class in *hammurabi.rules.dictionaries*), 62  
 DictValueNotExists (class in *hammurabi.rules.dictionaries*), 64

DirectoryEmptied (class in *hammurabi.rules.directories*), 66  
 DirectoryExists (class in *hammurabi.rules.directories*), 66  
 DirectoryNotExists (class in *hammurabi.rules.directories*), 67  
 documentation() (*hammurabi.law.Law* property), 94  
 documentation() (*hammurabi.rules.abstract.AbstractRule* property), 52  
 dry\_run (*hammurabi.config.CommonSettings* attribute), 92

## E

enforce() (*hammurabi.law.Law* method), 94  
 enforce() (*hammurabi.pillar.Pillar* method), 97  
 env\_prefix (*hammurabi.config.CommonSettings.Config* attribute), 92  
 execute() (*hammurabi.preconditions.base.Precondition* method), 46  
 execute() (*hammurabi.rules.base.Rule* method), 58

## F

failed (*hammurabi.reporters.base.Report* attribute), 50  
 failed\_rules() (*hammurabi.law.Law* property), 95  
 FileEmptied (class in *hammurabi.rules.files*), 68  
 FileExists (class in *hammurabi.rules.files*), 68  
 FileNotExists (class in *hammurabi.rules.files*), 69  
 FilesExist (class in *hammurabi.rules.files*), 70  
 FilesNotExist (class in *hammurabi.rules.files*), 70  
 finished (*hammurabi.reporters.base.AdditionalData* attribute), 50  
 full\_strip() (in module *hammurabi.helpers*), 93

## G

generate\_pull\_request\_body() (*hammurabi.mixins.PullRequestHelperMixin* method), 96  
 get\_by\_selector() (*hammurabi.rules.mixins.SelectorMixin* method),

81  
[get\\_execution\\_order\(\)](#) (*hammurabi.law.Law method*), 95  
[get\\_execution\\_order\(\)](#) (*hammurabi.rules.base.Rule method*), 58  
[get\\_law\(\)](#) (*hammurabi.pillar.Pillar method*), 97  
[get\\_rule\(\)](#) (*hammurabi.pillar.Pillar method*), 97  
[get\\_rule\\_chain\(\)](#) (*hammurabi.rules.base.Rule method*), 58  
[git\\_add\(\)](#) (*hammurabi.mixins.GitMixin method*), 96  
[git\\_base\\_name](#) (*hammurabi.config.CommonSettings attribute*), 92  
[git\\_branch\\_name](#) (*hammurabi.config.CommonSettings attribute*), 92  
[git\\_commit\(\)](#) (*hammurabi.mixins.GitMixin method*), 96  
[git\\_remove\(\)](#) (*hammurabi.mixins.GitMixin method*), 96  
[github\\_token](#) (*hammurabi.config.TOMLSettings attribute*), 93  
[GitHubMixin](#) (*class in hammurabi.mixins*), 95  
[GitMixin](#) (*class in hammurabi.mixins*), 95

## H

[hammurabi](#) (*module*), 98  
[hammurabi.config](#) (*module*), 92  
[hammurabi.exceptions](#) (*module*), 93  
[hammurabi.helpers](#) (*module*), 93  
[hammurabi.law](#) (*module*), 93  
[hammurabi.main](#) (*module*), 95  
[hammurabi.mixins](#) (*module*), 95  
[hammurabi.pillar](#) (*module*), 97  
[hammurabi.preconditions](#) (*module*), 50  
[hammurabi.preconditions.attributes](#) (*module*), 43  
[hammurabi.preconditions.base](#) (*module*), 45  
[hammurabi.preconditions.directories](#) (*module*), 47  
[hammurabi.preconditions.files](#) (*module*), 48  
[hammurabi.preconditions.text](#) (*module*), 49  
[hammurabi.reporters](#) (*module*), 52  
[hammurabi.reporters.base](#) (*module*), 50  
[hammurabi.reporters.json](#) (*module*), 51  
[hammurabi.rules](#) (*module*), 92  
[hammurabi.rules.abstract](#) (*module*), 52  
[hammurabi.rules.attributes](#) (*module*), 54  
[hammurabi.rules.base](#) (*module*), 57  
[hammurabi.rules.common](#) (*module*), 59  
[hammurabi.rules.dictionaries](#) (*module*), 60  
[hammurabi.rules.directories](#) (*module*), 66  
[hammurabi.rules.files](#) (*module*), 68  
[hammurabi.rules.ini](#) (*module*), 71  
[hammurabi.rules.json](#) (*module*), 77

[hammurabi.rules.mixins](#) (*module*), 81  
[hammurabi.rules.operations](#) (*module*), 82  
[hammurabi.rules.templates](#) (*module*), 84  
[hammurabi.rules.text](#) (*module*), 85  
[hammurabi.rules.yaml](#) (*module*), 87  
[HasMode](#) (*class in hammurabi.preconditions.attributes*), 43  
[HasNoMode](#) (*class in hammurabi.preconditions.attributes*), 44

## I

[IsDirectoryExists](#) (*class in hammurabi.preconditions.directories*), 47  
[IsDirectoryNotExists](#) (*class in hammurabi.preconditions.directories*), 47  
[IsFileExists](#) (*class in hammurabi.preconditions.files*), 48  
[IsFileNotExists](#) (*class in hammurabi.preconditions.files*), 48  
[IsLineExists](#) (*class in hammurabi.preconditions.text*), 49  
[IsLineNotExists](#) (*class in hammurabi.preconditions.text*), 50  
[IsNotOwnedBy](#) (*class in hammurabi.preconditions.attributes*), 44  
[IsOwnedBy](#) (*class in hammurabi.preconditions.attributes*), 44

## J

[JSONKeyExists](#) (*class in hammurabi.rules.json*), 77  
[JSONKeyNotExists](#) (*class in hammurabi.rules.json*), 77  
[JSONKeyRenamed](#) (*class in hammurabi.rules.json*), 78  
[JSONReporter](#) (*class in hammurabi.reporters.json*), 51  
[JSONValueExists](#) (*class in hammurabi.rules.json*), 78  
[JSONValueNotExists](#) (*class in hammurabi.rules.json*), 80

## L

[Law](#) (*class in hammurabi.law*), 93  
[law](#) (*hammurabi.reporters.base.RuleItem attribute*), 51  
[LawItem](#) (*class in hammurabi.reporters.base*), 50  
[laws\(\)](#) (*hammurabi.pillar.Pillar property*), 97  
[LineExists](#) (*class in hammurabi.rules.text*), 85  
[LineNotExists](#) (*class in hammurabi.rules.text*), 86  
[LineReplaced](#) (*class in hammurabi.rules.text*), 86  
[load\(\)](#) (*hammurabi.config.Config method*), 92  
[log\\_format](#) (*hammurabi.config.TOMLSettings attribute*), 93  
[log\\_level](#) (*hammurabi.config.TOMLSettings attribute*), 93

`log_path` (*hammurabi.config.TOMLSettings* attribute),  
93

## M

`made_changes` (*hammurabi.preconditions.attributes.HasMode* attribute), 44

`made_changes` (*hammurabi.preconditions.attributes.HasNoMode* attribute), 44

`made_changes` (*hammurabi.preconditions.attributes.IsNotOwnedBy* attribute), 44

`made_changes` (*hammurabi.preconditions.attributes.IsOwnedBy* attribute), 45

`made_changes` (*hammurabi.preconditions.base.Precondition* attribute), 46

`made_changes` (*hammurabi.preconditions.directories.IsDirectoryExists* attribute), 47

`made_changes` (*hammurabi.preconditions.directories.IsDirectoryNotExists* attribute), 47

`made_changes` (*hammurabi.preconditions.files.IsFileExists* attribute), 48

`made_changes` (*hammurabi.preconditions.files.IsFileNotExists* attribute), 48

`made_changes` (*hammurabi.preconditions.text.IsLineExists* attribute), 49

`made_changes` (*hammurabi.preconditions.text.IsLineNotExists* attribute), 50

`made_changes` (*hammurabi.rules.abstract.AbstractRule* attribute), 53

`made_changes` (*hammurabi.rules.attributes.ModeChanged* attribute), 55

`made_changes` (*hammurabi.rules.attributes.OwnerChanged* attribute), 56

`made_changes` (*hammurabi.rules.attributes.SingleAttributeRule* attribute), 56

`made_changes` (*hammurabi.rules.base.Rule* attribute), 58

`made_changes` (*hammurabi.rules.common.MultiplePathRule* attribute), 59

`made_changes` (*hammurabi.rules.common.SinglePathRule* attribute), 59

`made_changes` (*hammurabi.rules.dictionaries.DictKeyExists* attribute), 60

`made_changes` (*hammurabi.rules.dictionaries.DictKeyNotExists* attribute), 61

`made_changes` (*hammurabi.rules.dictionaries.DictKeyRenamed* attribute), 62

`made_changes` (*hammurabi.rules.dictionaries.DictValueExists* attribute), 63

`made_changes` (*hammurabi.rules.dictionaries.DictValueNotExists* attribute), 64

`made_changes` (*hammurabi.rules.dictionaries.SinglePathDictParsedRule* attribute), 65

`made_changes` (*hammurabi.rules.directories.DirectoryEmptied* attribute), 66

`made_changes` (*hammurabi.rules.directories.DirectoryExists* attribute), 67

`made_changes` (*hammurabi.rules.directories.DirectoryNotExists* attribute), 67

`made_changes` (*hammurabi.rules.files.FileEmptied* attribute), 68

`made_changes` (*hammurabi.rules.files.FileExists* attribute), 69

`made_changes` (*hammurabi.rules.files.FileNotExists* attribute), 69

`made_changes` (*hammurabi.rules.files.FilesExist* attribute), 70

`made_changes` (*hammurabi.rules.files.FilesNotExist* attribute), 71

`made_changes` (*hammurabi.rules.ini.OptionRenamed* attribute), 72

`made_changes` (*hammurabi.rules.ini.OptionsExist* attribute), 73

`made_changes` (*hammurabi.rules.ini.OptionsNotExist* attribute), 73

`made_changes` (*hammurabi.rules.ini.SectionExists* attribute), 74

`made_changes` (*hammurabi.rules.ini.SectionNotExists* attribute), 75

`made_changes` (*hammurabi.rules.ini.SectionRenamed* attribute), 76

`made_changes` (*hammurabi.rules.ini.SingleConfigFileRule* attribute), 76

- tribute), 76
- made\_changes (hammurabi.rules.json.JSONKeyExists attribute), 77
- made\_changes (hammurabi.rules.json.JSONKeyNotExists attribute), 78
- made\_changes (hammurabi.rules.json.JSONKeyRenamed attribute), 78
- made\_changes (hammurabi.rules.json.JSONValueExists attribute), 80
- made\_changes (hammurabi.rules.json.JSONValueNotExists attribute), 80
- made\_changes (hammurabi.rules.json.SingleJSONFileRule attribute), 81
- made\_changes (hammurabi.rules.operations.Copied attribute), 82
- made\_changes (hammurabi.rules.operations.Moved attribute), 83
- made\_changes (hammurabi.rules.operations.Renamed attribute), 83
- made\_changes (hammurabi.rules.templates.TemplateRendered attribute), 84
- made\_changes (hammurabi.rules.text.LineExists attribute), 85
- made\_changes (hammurabi.rules.text.LineNotExists attribute), 86
- made\_changes (hammurabi.rules.text.LineReplaced attribute), 87
- made\_changes (hammurabi.rules.yaml.SingleDocumentYAMLFileRule attribute), 87
- made\_changes (hammurabi.rules.yaml.YAMLKeyExists attribute), 88
- made\_changes (hammurabi.rules.yaml.YAMLKeyNotExists attribute), 89
- made\_changes (hammurabi.rules.yaml.YAMLKeyRenamed attribute), 89
- made\_changes (hammurabi.rules.yaml.YAMLValueExists attribute), 91
- made\_changes (hammurabi.rules.yaml.YAMLValueNotExists attribute), 91
- ModeChanged (class in hammurabi.rules.attributes), 54
- Moved (class in hammurabi.rules.operations), 82
- MultiplePathRule (class in hammurabi.rules.common), 59
- ## N
- name (hammurabi.preconditions.attributes.HasMode attribute), 44
- name (hammurabi.preconditions.attributes.HasNoMode attribute), 44
- name (hammurabi.preconditions.attributes.IsNotOwnedBy attribute), 44
- name (hammurabi.preconditions.attributes.IsOwnedBy attribute), 45
- name (hammurabi.preconditions.base.Precondition attribute), 46
- name (hammurabi.preconditions.directories.IsDirectoryExists attribute), 47
- name (hammurabi.preconditions.directories.IsDirectoryNotExists attribute), 47
- name (hammurabi.preconditions.files.IsFileExists attribute), 48
- name (hammurabi.preconditions.files.IsFileNotExists attribute), 48
- name (hammurabi.preconditions.text.IsLineExists attribute), 49
- name (hammurabi.preconditions.text.IsLineNotExists attribute), 50
- name (hammurabi.reporters.base.LawItem attribute), 50
- name (hammurabi.reporters.base.RuleItem attribute), 51
- name (hammurabi.rules.abstract.AbstractRule attribute), 53
- name (hammurabi.rules.attributes.ModeChanged attribute), 55
- name (hammurabi.rules.attributes.OwnerChanged attribute), 56
- name (hammurabi.rules.attributes.SingleAttributeRule attribute), 56
- name (hammurabi.rules.base.Rule attribute), 58
- name (hammurabi.rules.common.MultiplePathRule attribute), 59
- name (hammurabi.rules.common.SinglePathRule attribute), 59
- name (hammurabi.rules.dictionaries.DictKeyExists attribute), 60
- name (hammurabi.rules.dictionaries.DictKeyNotExists attribute), 61
- name (hammurabi.rules.dictionaries.DictKeyRenamed attribute), 62
- name (hammurabi.rules.dictionaries.DictValueExists attribute), 63
- name (hammurabi.rules.dictionaries.DictValueNotExists attribute), 64
- name (hammurabi.rules.dictionaries.SinglePathDictParsedRule attribute), 65



- name (*hammurabi.rules.directories.DirectoryEmptied attribute*), 66
- name (*hammurabi.rules.directories.DirectoryExists attribute*), 67
- name (*hammurabi.rules.directories.DirectoryNotExists attribute*), 67
- name (*hammurabi.rules.files.FileEmptied attribute*), 68
- name (*hammurabi.rules.files.FileExists attribute*), 69
- name (*hammurabi.rules.files.FileNotExists attribute*), 69
- name (*hammurabi.rules.files.FilesExist attribute*), 70
- name (*hammurabi.rules.files.FilesNotExist attribute*), 71
- name (*hammurabi.rules.ini.OptionRenamed attribute*), 72
- name (*hammurabi.rules.ini.OptionsExist attribute*), 73
- name (*hammurabi.rules.ini.OptionsNotExist attribute*), 73
- name (*hammurabi.rules.ini.SectionExists attribute*), 74
- name (*hammurabi.rules.ini.SectionNotExists attribute*), 75
- name (*hammurabi.rules.ini.SectionRenamed attribute*), 76
- name (*hammurabi.rules.ini.SingleConfigFileRule attribute*), 76
- name (*hammurabi.rules.json.JSONKeyExists attribute*), 77
- name (*hammurabi.rules.json.JSONKeyNotExists attribute*), 78
- name (*hammurabi.rules.json.JSONKeyRenamed attribute*), 78
- name (*hammurabi.rules.json.JSONValueExists attribute*), 80
- name (*hammurabi.rules.json.JSONValueNotExists attribute*), 80
- name (*hammurabi.rules.json.SingleJSONFileRule attribute*), 81
- name (*hammurabi.rules.operations.Copied attribute*), 82
- name (*hammurabi.rules.operations.Moved attribute*), 83
- name (*hammurabi.rules.operations.Renamed attribute*), 83
- name (*hammurabi.rules.templates.TemplateRendered attribute*), 84
- name (*hammurabi.rules.text.LineExists attribute*), 85
- name (*hammurabi.rules.text.LineNotExists attribute*), 86
- name (*hammurabi.rules.text.LineReplaced attribute*), 87
- name (*hammurabi.rules.yaml.SingleDocumentYAMLFileRule attribute*), 87
- name (*hammurabi.rules.yaml.YAMLKeyExists attribute*), 88
- name (*hammurabi.rules.yaml.YAMLKeyNotExists attribute*), 89
- name (*hammurabi.rules.yaml.YAMLKeyRenamed attribute*), 89
- name (*hammurabi.rules.yaml.YAMLValueExists attribute*), 91
- name (*hammurabi.rules.yaml.YAMLValueNotExists attribute*), 91
- ## O
- OptionRenamed (*class in hammurabi.rules.ini*), 71
- OptionsExist (*class in hammurabi.rules.ini*), 72
- OptionsNotExist (*class in hammurabi.rules.ini*), 73
- OwnerChanged (*class in hammurabi.rules.attributes*), 55
- ## P
- param (*hammurabi.preconditions.attributes.HasMode attribute*), 44
- param (*hammurabi.preconditions.attributes.HasNoMode attribute*), 44
- param (*hammurabi.preconditions.attributes.IsNotOwnedBy attribute*), 44
- param (*hammurabi.preconditions.attributes.IsOwnedBy attribute*), 45
- param (*hammurabi.preconditions.base.Precondition attribute*), 46
- param (*hammurabi.preconditions.directories.IsDirectoryExists attribute*), 47
- param (*hammurabi.preconditions.directories.IsDirectoryNotExists attribute*), 47
- param (*hammurabi.preconditions.files.IsFileExists attribute*), 48
- param (*hammurabi.preconditions.files.IsFileNotExists attribute*), 49
- param (*hammurabi.preconditions.text.IsLineExists attribute*), 49
- param (*hammurabi.preconditions.text.IsLineNotExists attribute*), 50
- param (*hammurabi.rules.abstract.AbstractRule attribute*), 53
- param (*hammurabi.rules.attributes.ModeChanged attribute*), 55
- param (*hammurabi.rules.attributes.OwnerChanged attribute*), 56
- param (*hammurabi.rules.attributes.SingleAttributeRule attribute*), 56
- param (*hammurabi.rules.base.Rule attribute*), 58
- param (*hammurabi.rules.common.MultiplePathRule attribute*), 59
- param (*hammurabi.rules.common.SinglePathRule attribute*), 59
- param (*hammurabi.rules.dictionaries.DictKeyExists attribute*), 61
- param (*hammurabi.rules.dictionaries.DictKeyNotExists attribute*), 61
- param (*hammurabi.rules.dictionaries.DictKeyRenamed attribute*), 62
- param (*hammurabi.rules.dictionaries.DictValueExists attribute*), 63

param (hammurabi.rules.dictionaries.DictValueNotExists attribute), 65	param (hammurabi.rules.yaml.SingleDocumentYAMLFileRule attribute), 87
param (hammurabi.rules.dictionaries.SinglePathDictParseRule attribute), 65	param (hammurabi.rules.yaml.YAMLKeyExists attribute), 88
param (hammurabi.rules.directories.DirectoryEmptied attribute), 66	param (hammurabi.rules.yaml.YAMLKeyNotExists attribute), 89
param (hammurabi.rules.directories.DirectoryExists attribute), 67	param (hammurabi.rules.yaml.YAMLKeyRenamed attribute), 89
param (hammurabi.rules.directories.DirectoryNotExists attribute), 67	param (hammurabi.rules.yaml.YAMLValueExists attribute), 91
param (hammurabi.rules.files.FileEmptied attribute), 68	param (hammurabi.rules.yaml.YAMLValueNotExists attribute), 91
param (hammurabi.rules.files.FileExists attribute), 69	passed (hammurabi.reporters.base.Report attribute), 50
param (hammurabi.rules.files.FileNotExists attribute), 69	passed_rules () (hammurabi.law.Law property), 95
param (hammurabi.rules.files.FilesExist attribute), 70	Pillar (class in hammurabi.pillar), 97
param (hammurabi.rules.files.FilesNotExist attribute), 71	pillar (hammurabi.config.Settings attribute), 93
param (hammurabi.rules.ini.OptionRenamed attribute), 72	pillar_config (hammurabi.config.TOMLSettings attribute), 93
param (hammurabi.rules.ini.OptionsExist attribute), 73	pillar_name (hammurabi.config.TOMLSettings attribute), 93
param (hammurabi.rules.ini.OptionsNotExist attribute), 74	post_task_hook () (hammurabi.rules.abstract.AbstractRule method), 53
param (hammurabi.rules.ini.SectionExists attribute), 75	post_task_hook () (hammurabi.rules.attributes.SingleAttributeRule method), 56
param (hammurabi.rules.ini.SectionNotExists attribute), 75	post_task_hook () (hammurabi.rules.common.MultiplePathRule method), 59
param (hammurabi.rules.ini.SectionRenamed attribute), 76	post_task_hook () (hammurabi.rules.common.SinglePathRule method), 59
param (hammurabi.rules.ini.SingleConfigFileRule attribute), 76	post_task_hook () (hammurabi.rules.directories.DirectoryNotExists method), 67
param (hammurabi.rules.json.JSONKeyExists attribute), 77	post_task_hook () (hammurabi.rules.files.FileNotExists method), 69
param (hammurabi.rules.json.JSONKeyNotExists attribute), 78	post_task_hook () (hammurabi.rules.files.FilesNotExist method), 71
param (hammurabi.rules.json.JSONKeyRenamed attribute), 78	post_task_hook () (hammurabi.rules.operations.Copied method), 82
param (hammurabi.rules.json.JSONValueExists attribute), 80	post_task_hook () (hammurabi.rules.operations.Moved method), 83
param (hammurabi.rules.json.JSONValueNotExists attribute), 80	post_task_hook () (hammurabi.rules.templates.TemplateRendered method), 84
param (hammurabi.rules.json.SingleJSONFileRule attribute), 81	pre_task_hook () (hammurabi.rules.abstract.AbstractRule method), 53
param (hammurabi.rules.operations.Copied attribute), 82	
param (hammurabi.rules.operations.Moved attribute), 83	
param (hammurabi.rules.operations.Renamed attribute), 83	
param (hammurabi.rules.templates.TemplateRendered attribute), 84	
param (hammurabi.rules.text.LineExists attribute), 85	
param (hammurabi.rules.text.LineNotExists attribute), 86	
param (hammurabi.rules.text.LineReplaced attribute), 87	



[pre\\_task\\_hook\(\)](#) ([hammurabi.rules.dictionaries.SinglePathDictParsedRule](#) [method](#)), [65](#)  
[pre\\_task\\_hook\(\)](#) ([hammurabi.rules.ini.SingleConfigFileRule](#) [method](#)), [76](#)  
[Precondition](#) ([class](#) in [hammurabi.preconditions.base](#)), [45](#)  
[PreconditionFailedError](#), [93](#)  
[pull\\_request\\_url](#) ([hammurabi.reporters.base.AdditionalData](#) [attribute](#)), [50](#)  
[PullRequestHelperMixin](#) ([class](#) in [hammurabi.mixins](#)), [96](#)  
[push\\_changes\(\)](#) ([hammurabi.mixins.GitMixin](#) [static method](#)), [96](#)

## R

[register\(\)](#) ([hammurabi.pillar.Pillar](#) [method](#)), [97](#)  
[Renamed](#) ([class](#) in [hammurabi.rules.operations](#)), [83](#)  
[Report](#) ([class](#) in [hammurabi.reporters.base](#)), [50](#)  
[report\(\)](#) ([hammurabi.reporters.base.Reporter](#) [method](#)), [51](#)  
[report\(\)](#) ([hammurabi.reporters.json.JSONReporter](#) [method](#)), [52](#)  
[report\\_name](#) ([hammurabi.config.CommonSettings](#) [attribute](#)), [92](#)  
[Reporter](#) ([class](#) in [hammurabi.reporters.base](#)), [50](#)  
[repository](#) ([hammurabi.config.CommonSettings](#) [attribute](#)), [92](#)  
[Rule](#) ([class](#) in [hammurabi.rules.base](#)), [57](#)  
[rule\\_can\\_abort](#) ([hammurabi.config.CommonSettings](#) [attribute](#)), [92](#)  
[RuleItem](#) ([class](#) in [hammurabi.reporters.base](#)), [51](#)  
[rules\(\)](#) ([hammurabi.pillar.Pillar](#) [property](#)), [98](#)

## S

[SectionExists](#) ([class](#) in [hammurabi.rules.ini](#)), [74](#)  
[SectionNotExists](#) ([class](#) in [hammurabi.rules.ini](#)), [75](#)  
[SectionRenamed](#) ([class](#) in [hammurabi.rules.ini](#)), [75](#)  
[SelectorMixin](#) ([class](#) in [hammurabi.rules.mixins](#)), [81](#)  
[set\\_by\\_selector\(\)](#) ([hammurabi.rules.mixins.SelectorMixin](#) [method](#)), [81](#)  
[Settings](#) ([class](#) in [hammurabi.config](#)), [92](#)  
[SingleAttributeRule](#) ([class](#) in [hammurabi.rules.attributes](#)), [56](#)  
[SingleConfigFileRule](#) ([class](#) in [hammurabi.rules.ini](#)), [76](#)  
[SingleDocumentYAMLFileRule](#) ([class](#) in [hammurabi.rules.yaml](#)), [87](#)

[SingleJSONFileRule](#) ([class](#) in [hammurabi.rules.json](#)), [80](#)  
[SinglePathDictParsedRule](#) ([class](#) in [hammurabi.rules.dictionaries](#)), [65](#)  
[SinglePathRule](#) ([class](#) in [hammurabi.rules.common](#)), [59](#)  
[skipped](#) ([hammurabi.reporters.base.Report](#) [attribute](#)), [50](#)  
[skipped\\_rules\(\)](#) ([hammurabi.law.Law](#) [property](#)), [95](#)  
[started](#) ([hammurabi.reporters.base.AdditionalData](#) [attribute](#)), [50](#)

## T

[task\(\)](#) ([hammurabi.preconditions.attributes.HasMode](#) [method](#)), [44](#)  
[task\(\)](#) ([hammurabi.preconditions.attributes.HasNoMode](#) [method](#)), [44](#)  
[task\(\)](#) ([hammurabi.preconditions.attributes.IsNotOwnedBy](#) [method](#)), [44](#)  
[task\(\)](#) ([hammurabi.preconditions.attributes.IsOwnedBy](#) [method](#)), [45](#)  
[task\(\)](#) ([hammurabi.preconditions.base.Precondition](#) [method](#)), [46](#)  
[task\(\)](#) ([hammurabi.preconditions.directories.IsDirectoryExists](#) [method](#)), [47](#)  
[task\(\)](#) ([hammurabi.preconditions.directories.IsDirectoryNotExists](#) [method](#)), [47](#)  
[task\(\)](#) ([hammurabi.preconditions.files.IsFileExists](#) [method](#)), [48](#)  
[task\(\)](#) ([hammurabi.preconditions.files.IsFileNotExists](#) [method](#)), [49](#)  
[task\(\)](#) ([hammurabi.preconditions.text.IsLineExists](#) [method](#)), [49](#)  
[task\(\)](#) ([hammurabi.preconditions.text.IsLineNotExists](#) [method](#)), [50](#)  
[task\(\)](#) ([hammurabi.rules.abstract.AbstractRule](#) [method](#)), [53](#)  
[task\(\)](#) ([hammurabi.rules.attributes.ModeChanged](#) [method](#)), [55](#)  
[task\(\)](#) ([hammurabi.rules.attributes.OwnerChanged](#) [method](#)), [56](#)  
[task\(\)](#) ([hammurabi.rules.attributes.SingleAttributeRule](#) [method](#)), [56](#)  
[task\(\)](#) ([hammurabi.rules.base.Rule](#) [method](#)), [58](#)  
[task\(\)](#) ([hammurabi.rules.common.MultiplePathRule](#) [method](#)), [59](#)  
[task\(\)](#) ([hammurabi.rules.common.SinglePathRule](#) [method](#)), [60](#)  
[task\(\)](#) ([hammurabi.rules.dictionaries.DictKeyExists](#) [method](#)), [61](#)  
[task\(\)](#) ([hammurabi.rules.dictionaries.DictKeyNotExists](#) [method](#)), [61](#)

`task()` (*hammurabi.rules.dictionaries.DictKeyRenamed* static method), 53  
method), 62

`task()` (*hammurabi.rules.dictionaries.DictValueExists* **Y**  
method), 63 *YAMLKeyExists* (class in *hammurabi.rules.yaml*), 88

`task()` (*hammurabi.rules.dictionaries.DictValueNotExists* *YAMLKeyNotExists* (class in *hammurabi.rules.yaml*),  
method), 65 88

`task()` (*hammurabi.rules.dictionaries.SinglePathDictParsedRule*  
method), 65 *YAMLKeyRenamed* (class in *hammurabi.rules.yaml*), 89

`task()` (*hammurabi.rules.directories.DirectoryEmptied*  
method), 66 *YAMLValueExists* (class in *hammurabi.rules.yaml*), 89

`task()` (*hammurabi.rules.directories.DirectoryExists* *YAMLValueNotExists* (class in *hammurabi.rules.yaml*), 91  
method), 67

`task()` (*hammurabi.rules.directories.DirectoryNotExists*  
method), 67

`task()` (*hammurabi.rules.files.FileEmptied* method), 68

`task()` (*hammurabi.rules.files.FileExists* method), 69

`task()` (*hammurabi.rules.files.FileNotExists* method),  
70

`task()` (*hammurabi.rules.files.FilesExist* method), 70

`task()` (*hammurabi.rules.files.FilesNotExist* method),  
71

`task()` (*hammurabi.rules.ini.OptionRenamed* method),  
72

`task()` (*hammurabi.rules.ini.OptionsExist* method), 73

`task()` (*hammurabi.rules.ini.OptionsNotExist* method),  
74

`task()` (*hammurabi.rules.ini.SectionExists* method), 75

`task()` (*hammurabi.rules.ini.SectionNotExists*  
method), 75

`task()` (*hammurabi.rules.ini.SectionRenamed* method),  
76

`task()` (*hammurabi.rules.ini.SingleConfigFileRule*  
method), 76

`task()` (*hammurabi.rules.json.SingleJSONFileRule*  
method), 81

`task()` (*hammurabi.rules.operations.Copied* method),  
82

`task()` (*hammurabi.rules.operations.Moved* method),  
83

`task()` (*hammurabi.rules.templates.TemplateRendered*  
method), 84

`task()` (*hammurabi.rules.text.LineExists* method), 85

`task()` (*hammurabi.rules.text.LineNotExists* method),  
86

`task()` (*hammurabi.rules.text.LineReplaced* method),  
87

`task()` (*hammurabi.rules.yaml.SingleDocumentYAMLFileRule*  
method), 87

*TemplateRendered* (class in *hammurabi.rules.templates*), 84

*TOMLSettings* (class in *hammurabi.config*), 93

## V

`validate()` (*hammurabi.rules.abstract.AbstractRule*