ScriptEngine Documentation

Jan 27, 2023

Contents:

CHAPTER 1

Introduction

ScriptEngine is a lightweight and extensible framework and tool for executing scripts written in YAML. The main purpose of ScriptEngine is to replace shell scripts in situations where highly configurable and modular scripts are needed. ScriptEngine relies on YAML and makes extensive use of Jinja2 templating to provide a comprehensive scripting language. This allows users to put focus on the description of tasks, rather than their implementation. The description of complex tasks should be nearly as short and clear as for simpler ones.

The ScriptEngine concept separates scripts (*what* to do) from the implementation of tasks (*how* to do things), and also from the actual execution environment (the script engine instances). This modularity allows the provision of different execution models (e.g. sequential or parallel; local or remote) within the same framework.

ScriptEngine provides a concise set of basic tasks that users can use to write their scripts. However, extensibility is a fundamental principle for ScriptEngine. Complementary task sets are provided as Python packages that can be loaded into ScriptEngine at run time. Thus, the capabilities of ScriptEngine scripts extend to features provided by task set packages available from package repositories or locally.

ScriptEngine tasks are simple Python classes with a `run()` method, which means that it is rather easy to build new tasks sets for particular needs. And because everything is based on Python3, there is a wide range of modules available to get the actual work done easily and efficiently.

ScriptEngine is strongly influenced by Ansible. In fact, ScriptEngine was developed because it met almost, but not quite, the needs of the author. Here are some important differences:

- Tasks are executed on a single host, often the host where the ScriptEngine command line tool is runnning. This is not an implication of any ScriptEngine concept, though. It would be possible to implement ScriptEngine instances that execute tasks remotely, or on multiple hosts.
- Contrary to Ansible, ScriptEngine tasks specify actions, not states.
- ScriptEngine is *not* focused on system administration. Although this is not part of the underlying concept, it will most probably reflect in the selection of available tasks.
- ScriptEngine's implementation of tasks is extremely lightweight, they're basically single Python functions. Almost every user should be able to add new tasks to ScriptEngine.

ScriptEngine is easily installed from the *Python Package Index* (PyPI) with the pip package manager.

CHAPTER 2

Installation

2.1 Prerequisites

ScriptEngine needs Python version 3.6 or newer. To check the version of your default Python installation, run:

```
> python --version
```

If the version is 3.6 or larger, everything is fine. If the default Python version is 2, check if Python3 is still avalable:

```
> python3 --version
```

If that returns a verison >=3.6, you need to specify the Python3 interpreter explicitely in the installation below.

2.2 Install in a Python Virtual Environment

ScriptEngine is preferably installed in a Python virtual environment. There are different ways to create virtual environments, for example with virtualenv or venv. The `venv` module is part of the Python standard library and has been recommended for creating virtual environments since Python 3.5, Hence, it is used here to explain the ScriptEngine installation.

Load the `venv` module from the `python` executable (if your default Python is version 3, otherwise use `python3`) to create a virtual environment for ScriptEngine:

```
> python -m venv .se
```

The `.se` argument is just an arbitrary name for the virtual environment and the corresponding directory. You can chose any name, but it can be convenient to chose a hidden directory.

Activate the created virtual environment:

```
> source .se/bin/activate
```

The `venv` module will also install the `pip` package manager in the virtual environment. Once the virtual environment is activated, use `pip` to install ScriptEngine, along with it's dependencies, from the Python Package Index (PyPI):

```
(.se)> pip install scriptengine
```

Test the ScriptEngine installation with:

```
(.se)> se --version
```

which should display the ScriptEngine version.

2.3 Install under Anaconda

It is sometimes preferable to install ScriptEngine within Anaconda, a Python distribution that provides a rich set of packages for scientific computing. In particular, this is needed if additional ScriptEngine task packages have dependencies that can only be satisfied with the conda package manager.

Assuming that the `conda` command is available, create and activate a virtual environment (named `se` in the following example) for ScriptEngine:

```
> conda create -n se
> activate se
```

ScriptEngine is available as package from the `conda-forge` channel, so installation is as easy as

```
(se)> conda install -c conda-forge scriptengine
```

Test if ScriptEngine works in your `conda` environment:

```
(se)> se --version
```

2.4 Development installation

ScriptEngine can be installed directly from a local directory. This can be useful for testing own developments or changes that have not yet been published as a package on PyPi. For example, ScriptEngine can be installed from a clone of the Github repository:

```
(.se)> git clone https://github.com/uwefladrich/scriptengine.git
(.se)> cd scriptengine
(.se)> pip install -e .
```

This will install ScriptEngine along with it's dependencies very similar to installing from PyPI. However, any changes made in the local directory will immediately affect the ScriptEngine installation.

CHAPTER 3

Command Line Interface

The ScriptEngine command line interface is provided by the `se` script. The command provides basic help when running it with the `-h` or `--help` flag:

```
> se --help
usage: se [-h] [-V] [--loglevel {debug,info,warning,error,critical}]
          [--nocolor]
          files [files ...]

ScriptEngine command line tool

positional arguments:
  files                 YAML file(s) to read

optional arguments:
  -h, --help            show this help message and exit
  -V, --version         show ScriptEngine version and exit
  --loglevel {debug,info,warning,error,critical}
                        The minimum level of log messages that is displayed.
                        For verbose output, use "debug". For minimal output,
                        use "error" or "critical".
  --nocolor             do not use colored terminal output

Available ScriptEngine tasks: hpc.slurm.sbatch, base.chdir, base.command,
base.context, base.copy, base.echo, base.exit, base.find, base.getenv,
base.include, base.link, base.make_dir, base.move, base.remove,
base.task_timer, base.template, base.time
```

Using the `-V` or `--version` option displayes the current ScriptEngine version:

```
> se --version
0.8.5
```

One or more scripts can be passed to ScriptEngine as arguments:

```
> se hello.yml
2021-03-18 09:54:40 INFO [se.cli] Logging configured and started
2021-03-18 09:54:40 INFO [se.task:echo <37f189f217>] Hello, world!
Hello, world!
```

As seen above, ScriptEngine and ScriptEngine tasks provide information during the run via logging, usually as output to the terminal. The amount of information can be controlled by setting the `--loglevel`:

```
> se --loglevel debug hello.yml
2021-03-18 09:58:04 INFO [se.cli] Logging configured and started
2021-03-18 09:58:04 DEBUG [se.cli] Loaded task base.chdir: Chdir from scriptengine.
↪tasks.base.chdir
2021-03-18 09:58:04 DEBUG [se.cli] Loaded task base.command: Command from␣
↪scriptengine.tasks.base.command
[...]
2021-03-18 09:58:04 DEBUG [se.cli] Loaded task base.time: Time from scriptengine.
↪tasks.base.time
2021-03-18 09:58:04 DEBUG [se.task:echo <aa66c50601>] Created task: Echo: Hello,␣
↪world!
2021-03-18 09:58:04 INFO [se.task:echo <aa66c50601>] Hello, world!
Hello, world!

> se --loglevel error hello.yml
Hello, world!
```

Usually, the logging output is colored (not seen in this documentation). This is done on the terminal with ANSI escape codes, but it may be undesired, for example, when the output is stored in a logfile. Hence, colored output can be switched of with the `--nocolor` argument.

As seen in the output of `se --help` above, ScriptEngine lists all available task in the current installation. ScriptEngine uses dynamic task loading (see Concepts) and additional task can be installed from Python packages. In the example above, all tasks from the build-in `base.*` package are available. Furthermore, the `hpc.slurm.sbatch` task is provided, which comes from the `scriptengine-tasks-hpc` (go there) Python package.

Note that ScriptEngine task names follow a namespace scheme to prevent name clashes for tasks from different packages.

CHAPTER 4

Concepts

Working with ScriptEngine does not require knowledge about it's underlying implementation architecture in detail, but it is useful to understand a few basic concepts and terms. It is, for example, good to know how *tasks* and *scripts* are understood in the ScriptEngine world. Furthermore, a few technical topics are important when using ScriptEngine, in particular some details about the YAML format and how templating with Jinja2 works.

4.1 Tasks

A task is the basic unit of work for ScriptEngine. It is the building block for scripts (see *Scripts* section) and also the individual item a ScriptEngine instance (see even later) will handle.

Tasks "do things". This can be simple things, like copying a file or writing a message on the terminal. It could also be more complex things, involving more complex computations, file operations, or interactions with services. But whatever the actual complexity of a task is, it will be hidden. A task is listed in a script by it's name and some arguments, and later it is run by a ScriptEngine instance to do it's actual work.

A number of different tasks are available when using ScriptEngine. In fact, one of the main ideas with ScriptEngine is that it should be easy to develop and provide new tasks for anything that users may need. Hence, a task should be available for most of what users would want to do with ScriptEngine.

However, tasks are not "baked" into the ScriptEngine code. Instead, they are loaded, at run time, from separate Python modules. Thus, tasks can easily be provided by other packages and made available to the user when ScriptEngine is run. ScriptEngine comes with a basic package of tasks that are often used, but it can be adapted to a wide range of fields by providing specialised task packages.

Technically, a task is a Python class that is derived from the `Task` class that ScriptEngine provides. In fact, it is possible to use ScriptEngine from within other Python programs, without ever defining tasks in YAML scripts.

4.2 Jobs

Similar to tasks, *jobs* are units of work for ScriptEngine. Jobs can extend tasks in two ways:

- jobs can join a number of tasks into a sequence, and
- jobs can add contitionals and loops to tasks.

Corresponding to these two cases, jobs use the special `do` keyword to specify sequences of tasks (see *Do*), and/or `when` or `loop` clauses clauses to specify *Conditionals* and *Loops*, respectively.

4.3 Scripts

Scripts, in the ScriptEngine terminology, are collections of tasks, described in the YAML language. Usually, a script is a YAML file that represents a list of tasks. How specific tasks are represented in YAML is usually intuitive, as shown in the following example:

```
- base.context:
    planet: Earth
- base.echo:
    msg: "Hello, {{planet}}!"
```

This little script is the inevitable "Hello world" example in ScriptEngine! It is run like this:

```
> se hello-world.yml
```

provided the little YAML snippet was stored in a file called `hello-world.yml`. Please not that ScriptEngine uses a dot notation for the naming of tasks. This allows tasks from different task packages to coexist without conflict. The tasks in the above example come from the `base` task package, which is an integral part of ScriptEngine. Nevertheless, beside being always available, there is no technical difference between the `base` task package and others.

4.4 ScriptEngine Instances

Most of the time, the difference between the term ScriptEngine in general and a ScriptEngine instance in particular is not very important. However, technically, scripts are given to a particular ScriptEngine instance for execution. Different types of ScriptEngine instances could exist, allowing for different execution models to be implemented. For example, a ScriptEngine instance could allow tasks to be run in parallel, or to be submitted to remote hosts for execution.

For now, ScriptEngine provides `SimpleScriptEngine`, which takes scripts and executes tasks sequentially, on the local host.

4.5 Task Context

An important concept in ScriptEngine is the task context, or short, the *context*. The context is, technically, a dictionary, i.e. a data structure of key, value pairs. ScriptEngine tasks can store and retrieve information from the context.

When a ScriptEngine instance is created, the context is initialised. Some information about the execution environment is stored by the ScriptEngine instance in the new context. Then, it is passed to every task that is executed. Usually, the context will be populated with information as tasks are processed.

We have already seen the usage of the context in the "Hello world" example above. The `context` task stored a parameter named `planet` in the context and the `echo` task used the information from the context to display it's message.

Since the context is a Python dictionary, it can store any Python data types. This is, for example, often used to structure information by storing further dictionaries in the context. Numbers and dates are other examples for useful data types for context information.

4.6 YAML

YAML syntax for lists:

```
- apple
- pear
- peach
- banana
```

Compact list syntax:

```
[apple, pear, peach, banana]
```

A YAML dictionary:

```
name: apple
color: green
price: 0.2
```

Compact syntax:

```
{name: apple, color: green, price: 0.2}
```

A list of dictionaries:

```
- name: apple
  color: green
  price: 0.2
- name: pear
  color: pink
  price: 0.4
- name: banana
  color: yellow
  price: 0.7
```

A dictionary with lists:

```
name: apple
color: green
price: 0.2
vitamins:
    - C
    - B6
    - B2
    - K
```

YAML treats all terms as objects of no particular type. However, the Python YAML parser will convert terms into Python objects of the appropriate type, for example:

```
number: 2
another_number: 3.21
```

(continues on next page)

(continued from previous page)

```
string: This is a string
another_string: "This is a quoted string"
a_date: 2020-08-13
```

4.7 Jinja2 Templating

CHAPTER 5

Writing Scripts

ScriptEngine scripts are written in YAML. A basic understanding is therefore needed about YAML syntax rules in order to write ScriptEngine scripts.

5.1 Simple ScriptEngine scripts

Each ScriptEngine task is a YAML dictionary and a ScriptEngine script is usually a list of tasks. Let's look at the simplest possible script:

```
base.exit:
```

This script is not particularly helpful, it doesn't do anything at all. Nevertheless, if you put this in a file, e.g. `exit.yml`, and run:

```
> se exit.yml
```

ScriptEngine will run the `exit` task and, well, exit. Technically, the script contains the YAML dictionary `{exit: null}`, which will be parsed by ScriptEngine into the `exit` task.

Let't look at a slightly more useful example:

```
base.echo:
    msg: Hello, world!
```

This is the inevitable Hello, world! example in ScriptEngine. Beside the name of the task (`base.echo`, i.e. the `echo` task from the `base` package) there is also an argument, `msg`. From a YAML perspective, all task arguments are the (key, value) pairs of a dictionary associated with the task name.

Most scripts will contain more than one task and therefore will contain a YAML list containing the tasks:

```
- base.context:
    planet: Earth
- base.echo:
    msg: "Hello, {{planet}}!"
```

This script will write "Hello, Earth!". There are two tasks: `base.context` and `base.echo`, which are the two elements of a YAML list.

5.2 Do

The job specifier `do` allows for grouping a *list of tasks* inside one single job (see *Jobs*). For example:

```
- do:
  - base.context:
      planet: Earth
  - base.echo:
      msg: "Hello, {{planet}}!"
```

This is most often used in order to apply a `when` clause or a `loop` to a sequence of tasks.

5.3 Loops

Jobs and tasks can be looped over. The simplest example is just a task with an added `loop` specifier, such as:

```
- base.echo:
    msg: "Looping over item, which is now {{item}}"
  loop: [1,2,3]
```

In this example, the `base.echo` task would be executed three times, with `item` taking the values of 1, 2, and 3. Here, the loop is specified by an explicit list in YAML inline notation. A conventional block format notation of the list works just the same:

```
- base.echo:
    msg: "Looping over item, which is now {{item}}"
  loop:
    - 1
    - 2
    - 3
```

The list can also be specified in a separate `base.context` task, as in:

```
- base.context:
    list: [1,2,3]
- base.echo:
    msg: "Looping over item, which is now {{item}}"
  loop: "{{list}}"
```

Note that the string defining the loop list must be enclosed in quotes because of the braces.

In all of the above examples, the loop index variable was not explicitely named, which means it takes on it's default name, `item`. The `item` variable is added to the context for all jobs or tasks within the loop and can be accessed using the usual syntax, as shown in the previous examples. After the loop is completed, the variable is removed from the context, i.e. it is *not* possible to access it from jobs or tasks that follow the loop.

It is possible to explicitely define another name to the loop index variable, by using an extended loop specifier. Here is an example:

```
- base.echo:
    msg: "Looping over the 'foo' variable: {{foo}}"
  loop:
    with: foo
    in:   [1,2,3,4]
```

In that example, the loop index variable is named `foo` and it is added to the context of all jobs and tasks defined in the loop, in the same manner as the default `item` variable.

In case a loop variable (explicitely given or `item`, by default) already exists in the context when a loop is entered, ScriptEngine will issue a warning about a colliding loop index variable. Nevertheless, the loop will still be processed, with the loop variable value *hiding* the value of the variable with the same name from outside the loop. After the loop has completed, the original value of the variable is restored.

It is also possible to nest loops:

```
- do:
    - base.echo:
        msg: "Nested loop: 'foo' is {{foo}} while 'bar' is {{bar}}"
      loop:
        with: foo
        in:   [1,2]
  loop:
    with: bar
    in:   [4,5,6]
```

In most cases, it will make sense to explicitely define the name of the loop index variables in nested loops, although it *is* possible to rely on the default variables. So the following example would work:

```
- do:
    - base.echo:
        msg: "Nested loop: 'item' is {{item}}"
      loop: [1,2]
  loop: [4,5,6]
```

Nevertheless, ScriptEngine will, again, issue a warning about a loop index variable collision. When using nested loops with the same loop index variable (explicitely or by default), the variable values from outer loops will not be accessible in the inner loops.

It is also possible to loop over dicts in ScriptEngine, like in the following example:

```
- base.echo:
    msg: "{{key}} is {{value}} years old."
  loop:
    in:
        Mary: 31
        Peter: 29
        Paul: 39
```

which would yield:

```
Mary is 31 years old.
Peter is 29 years old.
Paul is 39 years old.
```

The example shows that the extended loop specifier with `in:` must be used when looping over dicts, otherwise an *invalid loop decriptor error* error occurs. Futhermore, the example shows that the default loop variables for loops over dicts are `key` and `value`. If the dict loop should use other variables, their names can be given explicitely:

```
- base.echo:
    msg: "{{name}} is {{age}} years old."
  loop:
    with: [name, age]
    in:
        Mary: 31
        Peter: 29
        Paul: 39
```

In the same manner as for lists, loop dicts can be defined in the ScriptEngine context:

```
- base.context:
    people:
        Mary: 31
        Peter: 29
        Paul: 39
- base.echo:
    msg: '{{name}} is {{age}} years old.'
  loop:
    with: [name, age]
    in: '{{people}}'
```

5.4 Conditionals

It is possible to control that a given job runs exclusively under a certain condition, by using a `when` clause. Here is an example:

```
- base.context:
    year: 1963
- base.echo:
    msg: 'Peter, Paul and Mary most famous song'
  when: "{{year==1963}}"
```

Hint: Because dict keys are not ordered in YAML, the second task in the previous example is equivalent to:

```
- when: "{{year==1963}}"
  base.echo:
    msg: 'Peter, Paul and Mary most famous song'
```

Some might find it easier to read if the condition preceeds the task body.

The `when` clause can be combined with the `do` keyword, to execute a sequence of tasks conditionally:

```
- base.context:
    year: 1963
- when: "{{year==1963}}"
  do:
    - base.echo:
        msg: 'Puff, the magic dragon'
    - base.echo:
        msg: 'lives by the sea'
```

Note: There is no *else* clause in ScriptEngine. If the equivalent to an if-then-else logic is needed, two `when` clauses with complementary expressions must be used.

5.5 Special YAML Features

5.5.1 YAML constructors

PyYAML (the YAML implementation used by ScriptEngine) allows user-defined data types, which are indicated by a single exclamation mark (!). ScriptEngine makes use of this feature to implement some advanced features:

Noparse strings

Every time ScriptEngine reads a string argument value from a script, it parses the value with Jinja2 (to make substitutions from the context and other Jinja2 transformations) and, thereafter, once more with YAML (to create correct data types, e.g. numbers, lists, dicts).

However, this leads sometimes to undesired results. Consider the following `context` task:

```
base.context:
  first_name: Foo
  last_name: Bar
  full_name: "{{first_name}} {{last_name}}"
```

In the example, `full_name` gets assigned " " (a single space), because `first_name` and `last_name` are only effectively in the context *after* the `context` task has completed.

ScriptEngine can be instructed to skip parsing the `full_name` argument in this task, which would solve the problem in many cases, because when `full_name` is used later as (part of) any other argument, it is parsed again, thus substituting `first_name` and `last_name` at a later stage.

To avoid parsing of an argument, use the `!noparse` YAML constructor:

```
base.context:
  first_name: Foo
  last_name: Bar
  full_name: !noparse "{{first_name}} {{last_name}}"
```

which assignes the argument string `{{first_name}} {{last_name}}` literally to `full_name` and delays parsing until later, when `first_name` and `last_name` are available from the context.

Another situation were parsing needs to be avoided is:

```
base.echo:
  msg: "Foo: bar"
```

which would, unexpectedly, write "`{'Foo': 'bar'}` instead of `Foo: bar` because YAML parsing would turn the string into a dictionary. Similar issues would arise with other data types, like lists or dates/times. `!noparse` avoids the situation again:

```
base.echo:
  msg: !noparse "Foo: bar"
```

and stores the string `Foo: bar` literally in the context.

While `!noparse` solves problems in most cases, a finer control over the parsing is sometimes needed. It is possible to avoid either Jinja2 or YAML parsing exclusively by using `!noparse_jinja` or `!noparse_yaml`, respectively.

RRULEs

ScriptEngine supports recurrence rules for dates and times, as defined in RFC5545 and implemented in the Python dateutil module. To create an RRULE in a ScriptEngine script, use the `!rrule` constructor (for an explanation of the > operator and multi-line strings, see below):

```
base.context:
  schedule: !rrule >
      DTSTART:19900101
      RRULE:FREQ=YEARLY;UNTIL=20000101
```

which would create a schedule with 11 yearly events, starting on January 1st 1990 and extending until, including, 2000. The specification is turned into a `dateutil.rrule.rrule` object, which is (in the above example) stored in the context. It could be used elsewhere in the script to access, for example, the year of the first event:

```
base.echo:
  msg: "First event is in year {{schedule[0].year}}"
```

5.5.2 Multi-line strings

Multi-line strings are defined in YAML and not a special feature of ScriptEngine. They can be useful for writing scripts by allowing to split long strings and make scripte more readable, or make it possible to format output. This is an example for using multi-line strings to format output:

```
base.echo:
  msg: !noparse_yaml |
    This
    is a multi-line
    string
    with an answer: {{18+24}}.
```

YAML multi-line strings are either denoted by `|`, in which case they are preserving line breaks, or by >, in which case they are not.

Note that in the example above, it is necessary to add `!noparse_yaml` because ScriptEngine would re-parse the multi-line string otherwise, removing all line breaks. If there hadn't been a Jinja2 command in the string, just `!noparse` had been working as well.

5.6 Jinja2 filters

ScriptEngine defines a number of additional Jinja2 filters, which might be useful for writing scripts.

5.6.1 Filters to handle dates and times

datetime Converts a string to a `datetime.datetime` object, for example:

```
base.context:
    date_time: "{{ '2022-01-01 00:00:00' | datetime }}""
```

The format of the string defaults to `%Y-%m-%d %H:%M:%S`, but it can be changed:

```
base.context:
    date_time: "{{ '2022/01/01 00:00:00' | datetime('%Y/%m/%d %H:%M:%S') }}""
```

date Converts a string to a `datetime.date` object:

```
base.context:
    start_date: "{{ '2022-01-01 00:00:00' | date }}""
```

The format of the date string can be changed the same way as for the **datetime** filter.

5.6.2 Filters to handle paths and filenames

basename Returns the base name of a part (i.e. the path with all but the last part removed):

```
- base.context:
    file: /path/to/file.txt
- base.copy:
    src: "{{ file }}"
    dst: "/new/path/to/{{ file | basename }}"
```

dirname Returns the directory part of a path (i.e. the parth with the base name removed):

```
- base.context:
    file: /path/to/file.txt
- base.copy:
    src: "{{ file }}"
    dst: "{{ file | dirname }}/new_file.txt"
```

exists Returns true if the path exists, otherwise false:

```
when: "{{ '/path/to/file' | exists }}"
echo:
    msg: Yes, file exists!
```

path_join Composes path from components:

```
base.echo:
    msg: "{{ ['foo', 'bar.txt'] | path_join }}"
```

CHAPTER 6

The base task package

The ScriptEngine **base** task package collects essential tasks that are commonly needed to write scripts. The base task package is pre-installed in ScriptEngine and any task within the `base.*` namespace can be used without further installation or set up. Many useful ScriptEngine scripts can be written with just the base task package.

The base task package contains the following tasks, described in more detail below:

```
base.echo, base.chdir, base.command, base.context, base.context.from,
base.copy, base.exit, base.find, base.getenv, base.include, base.link,
base.make_dir, base.move, base.remove, base.setenv, base.task_timer,
base.template, base.time
```

6.1 base.echo

This is a very basic task that can be used to write customised messages:

```
base.echo:
    msg: <MESSAGE>
```

for example:

```
base.echo:
    msg: Hello, world!
```

or, provided that `planet` is set in the context (*base.context*):

```
base.echo:
    msg: "Hello, {{ planet }}!"
```

6.2 Working with the SE context

6.2.1 base.context

Stores or updates data (one or more, possibly nested, pairs of names and values) in the ScriptEngine context:

```
base.context:
    <NAME>: <VALUE>
    [...]
```

For example:

```
base.context:
    planet: Earth
    some_countries:
        - Norway
        - Sweden
        - Finnland
        - Danmark
    number: 4
```

The arguments of `base.context` will be *merged* into the ScriptEngine context. This means that data can be added to existing data structures, such as lists or dictionaries:

```
- base.context:
    mylist:
        - one
        - two
# mylist is ['one', 'two']
- base.context:
    mylist:
        - 3
        - 4
# mylist is now ['one', 'two', 3, 4]
```

In the case of conflicts, the arguments of `base.context` will overwrite the current values in the ScriptEngine context. This can be used to clean data:

```
- base.context:
    mylist:
        - one
        - two
# mylist is ['one', 'two']
- base.context:
    mylist: null  # "remove" the value if mylist
- base.context:
    mylist:
        - 3
        - 4
# mylist is now [3, 4]
```

6.2.2 base.context.from

This is an extension of `base.context`. It updates the ScriptEngine context in the same way, but it allows to "read" the context update from another source instead of explicitly specifying the name-value pairs as task arguments.

In particular, `base.context.from` accepts one of two arguments, `dict` or `file`. The arguments are mutual exclusive:

```
base.context.from:
    # exactly one of the two arguments:
    dict: <DICTIONARY>  # optional, mutual exclusive
    file: <FILE_NAME>  # optional, mutual exclusive
```

If given the `dict` argument, the context update is specified by the argument value, which must be a dictionary. This may sound rather similar to the standard `base.context`, but it allows greater flexibility because the argument value can be taken from the context itself. For example, one could implement overwriteable default settings using this feature:

```
# Let the user set preferred values
- base.context:
    user_config:
        foo: 5
# [... later (could be in another script) ...]
# Set default values
- base.context:
    foo: 1
    bar: 2
# Overwrite defaults with user preferences
- base.context.from:
    dict: "{{ user_config  }}"
# result: foo==5, bar==2
```

The `file` argument of `base.context.from` can be used to read context values from a YAML file:

```
# data.yml
foo: 4
bar: 5

# script.yml
- base.context.from:
    file: data.yml
```

When running the scripte with `se script.yml`, the context will contain `foo==4` and `bar==5`, provided that the file `data.yml` can be found in the current directory.

The only supported file format for the time being is YAML. The content of the file must be a, possibly nested, dictionary (i.e. single values or lists are not allowed).

6.3 Control flow

6.3.1 base.include

Reads and executes another ScriptEngine script. This is done *if* (see *Conditionals*) and *when* the `base.include` task is executed:

```
base.include:
    src: <PATH>
    ignore_not_found: <true or false>  # optional, default false
```

The script to be included is given by the `src` argument, which must be a path relative to

- the current working directory at the moment `base.include` is run,
- the original working directory when the `se` command was run, or
- any of the directories that the scripts given to the `se` command were in.

If `ignore_not_found` is `true`, only a warning is written in case the include script is not found. If it is `false` (the default) an error is raised in this case.

6.3.2 base.command

Executes an external command, with optional arguments:

```
- base.command:
    name: <COMMAND_NAME>
    args: <LIST_OF_ARGS>  # optional
    cwd: <PATH>  # optional
    stdout: [true|false|<STRING>]  # optional
    stderr: [true|false|<STRING>]  # optional
    ignore_error: [true|false]  # optional
```

When `cwd` (current work directory) is specified, the command is executed in the given directory:

```
- base.command:
    name: ls
    args: [-l]
    cwd: /tmp
```

When the `stdout` is given, it can be eiter true, false, or a string that makes for a valid name in the ScriptEngine context. If `stdout` is set to true (the default), then the standard output of the command is printed as log messages on the INFO level. When `stdout` is false, the standard output of the command is ignored.

When `stdout` is a name, the standard output of the command is stored, under that name, in the ScriptEngine context, for example:

```
- base.command:
    name: echo
    args: [ Hello, World! ]
    stdout: message
- base.echo:
    msg: "Command returned: {{message}}"
```

Note that the standard output is always returned as a list of lines, even if there is only one line (as in the example above). This is often desired, for example when using the command output in a loop. However, if one wanted to extract the first (and only) line in the example above, Jinja2 syntax could be used:

```
- echo:
    msg: "Command returned: {{message|first}}"
```

Note: When tasks update the ScriptEngine context, the changes are always *merged* (see *base.context*). This implies, among other things, that list items are *appended* if the list is already defined in the context. This mechanism applies also to `base.command` and consequently output lines are appended to the context variable if it already exist.

The `stderr` argument works exactly as `stdout`, but for standard error output.

If the command returns a non-zero exit code, ScriptEngine writes the exit code as log message (on the ERROR level) and stops with an error. However, if `ignore_error` is true and the command returns a non-zero exit code, the exit

code of the command is logged at the WARNING level instead and ScriptEngine continues. The default value for `ignore_error` is false.

6.3.3 base.exit

Requests ScriptEngine to stop, optionally displaying a customised message:

```
- base.exit:
    msg: <MESSAGE>  # optional
```

If the `msg` argument is not given, a default message is printed.

6.4 Shell environment

6.4.1 base.chdir

This task changes the current working directory:

```
base.chdir:
    path: <PATH>
```

for example:

```
- base.getenv:
    home: HOME
- base.chdir:
    path: "{{ home }}"
```

6.4.2 base.getenv

Reads one or more environment variables and stores the values in the ScriptEngine context:

```
- base.getenv:
    <CONTEXT_PARAMETER>: <ENV_VAR_NAME>
    [...]
```

for example:

```
- base.getenv:
    name: USER
    home: HOME
- base.echo:
    msg: "I am {{ name }} and {{ home }} is my castle."
```

When a requested environment variable does not exist, a warning is given and no corresponding context changes are made.

Warning: Only simple, non-nested context parameters (without dots) can be used in `base.getenv`

6.4.3 base.setenv

Sets one or more environment variables from values of the ScriptEngine context:

```
- base.setenv:
    <ENV_VAR_NAME>: <CONTEXT_PARAMETER>
    [...]
```

The following example:

```
- base.context:
    libs: /path/to/libraries
- base.setenv:
    LD_LIBRARY_PATH:  "{{ libs }}"
    FOO: 1
    bar: two
```

will set the environment variables `$LD_LIBRARY_PATH` to `"/path/to/libraries"`, `$FOO` to `"1"` and `$bar` to `"two"`.

Note: Environment variables are always strings! Thus, all values are converted to strings before they are assigned. In the above example, the number `1` is converted to the string `"1"` before it is assigned to the environment variable `$FOO`.

Warning: Only simple, non-nested context parameters can be used in `base.setenv`

6.5 Basic file operations

The ScriptEngine base task package provides tasks to create, copy/move/link and remove files and directories, as described in detail below in this section.

Whenever it makes sense, the tasks will accept as their argument values single file or directory names, as well as YAML lists of such. Furhtermore, instead of full names, also Unix shell wildcard expressions are accepted.

6.5.1 base.make_dir

Creates a new directory at the given `path`:

```
base.make_dir:
    path: <PATH>
```

If `path` already exists, an info message is displayed (no warning or error). When `path` is a file or symbolic link, an error occurs.

A list of names is accepted for the `path` argument.

6.5.2 base.copy

This task copies the file or directory given by `src` to `dst`:

```
- base.copy:
    src: <PATH>
    dst: <PATH>
    ignore_not_found: <true or false>  # optional, default false
```

If `src` is a file and `dst` is a directory, the `src` file is copied into the `dst/` directory. If `src` is a directory, `dst` must be a directory as well and `src` is copied recursively into `dst/`. When a directory is copied, symbolic links are preserved.

When copying a file and the `dst` exists already, it is overwritten and a waring is issued. Copying a directory when `dst` already exists results in an error. An error occurs if `src` does not exist, unless `ignore_not_found` is `true`.

A list of names or wildcard expressions is accepted for the `src` argument, provided that `dst` is a directory.

6.5.3 base.link

Creates a symbolic link with name given by `link`, which is pointing to the path given by `target`:

```
base.link:
    target: <PATH>
    link: <PATH>
```

When the `target` does not exist, the link is still created and a warning is issued.

When `link` is a directory, a symbolic link with the same base name as `target` is created within the `link` directory, pointing to `target`.

A list of names or wildcard expressions is accepted for the `target` argument, provided that `link` is a directory. A symbolic link for each `target` is created in that case in the `link` directory.

> **Warning:** In ScriptEngine versions up to 0.13.1 the arguments of `base.link` were named `src` (now `target`) and `dst` (now `link`). The use of these argument names is deprecated and will be invalid in future versions of ScriptEngine. The use of directories for `dst` or lists and wildcards for `src` was not available in those versions.

6.5.4 base.move

Moves files or directories (the latter recursively) from `src` to `dst`:

```
base.move:
    src: <PATH>
    dst: <PATH>
    ignore_not_found: <true or false>  # optional, default false
```

If `dst` is a directory, `src` is moved *into* `dst`. If `src` does not exists, an error occurs unless `ignore_not_found` is true.

Provided that `dst` is a directory, `src` may be a list or wildcard expression.

6.5.5 base.remove

Removes a file, link, or directory:

```
base.remove:
    path: <PATH>
    ignore_not_found: <true or false>  # optional, default false
```

Directories are recursively deleted, effectively removing all files and subdirectories that it contains. When `path` does not exist, an error occurs, unless `ignore_not_found` is `true`.

A list of names or wildcard expressions is accepted for `path`.

6.6 Other file operations

6.6.1 base.template

Runs the template file given by `src` through the Jinja2 Template Engine and saves the result as a file at `dst`:

```
base.template:
    src: <PATH>
    dst: <PATH>
    executable: [true|false]  # optional
```

ScriptEngine searches for the template file (`src`) in the following directories, in the order given:

1. `.`
2. `./templates`
3. `{{ se.cli.cwd }}`
4. `{{ se.cli.cwd }}/templates`

where `.` is the current directory at the time when the `template` task is executed and `{{se.cli.cwd}}` is the original working directory, the working directory at the time when the ScriptEngine command line tool was called.

The ScriptEngine context is passed to the Jinja2 template engine when the template is rendered, which means that all context parameters can be referred to in the template.

If the `executable` argument is true (the default being false), the destination file will get executable permissions. The setting of permissions will respect the user's umask.

6.6.2 base.find

This task can be used to find files or directories in the file system:

```
base.find:
    path: <PATH>
    pattern: <SEARCH_PATTERN>  # optional, default "*"
    type: <FILE_OR_DIR>  # optional, default "file"
    depth: <NUMBER>  # optional, default -1
    set: <CONTEXT_PARAMETER>  # optional, default "result"
```

Files and directories are searched starting at `path` and decending at most `depth` levels down the directory hierarchy. If `depth` is less than zero, no limit is used for the search. Files and directories are matched against the Unix shell-style wildcards `pattern`, which may include:

```
*       matches everything
?       matches any single character
[seq]   matches any character in seq
[!seq]  matches any character not in seq
```

If `type` is `file` (default) or `dir`, then `base.find` will search for files or directories, respectively.

Note: `base.find` supports Unix shell-type wildcards, not regular expressions.

6.7 Timing

6.7.1 base.time

This task can be used to measure absolute or elapsed time in ScriptEngine scripts:

```
base.time:
    set: <CONTEXT_NAME>
    since: <DATETIME>  # optional
```

The `set` argument specifies a name under which the time is stored in the context. Only simple names, without dots, may be used.

If the `since` argument is used, it must represent a `datetime` object and the elapsed time since this reference is measured:

```
- base.time:
    set: start
- base.echo:
    msg: Hello, world!
- base.time:
    set: elapsed_time
    since: "{{ start }}"
```

6.7.2 base.task_timer

This task can be used to control ScriptEngine's build-in timing feature. It is used as:

```
base.task_timer:
    mode: <TIMING_MODE>
    logging: <LOGLEVEL>  # optional
```

where:

```
TIMING_MODE is one of
    false:       Timing is switched off.
    'basic':     Each task is timed, log messages are written according to
                 'logging' argument.
    'classes':   As for 'basic', plus times are accumulated for task
                 classes.
    'instances': As for 'classes', plus times are accumulated for each
                 individual task instance.
```

and:

```
LOGLEVEL is one of
    false:   No time logging after each task
    'info':  Logging to the info logger
    'debug': Logging to the debug logger
```

Note: Switching off logging (`LOGLEVEL: false`) does not affect the collection of timing data for the tasks.

6.8 Context task

Stores or updates data (one or more, possibly nested, pairs of names and values) in the ScriptEngine context.

Usage:

```
base.context:
    <NAME>: <VALUE>
    [...]
```

Example:

```
base.context:
    planet: Earth
    some_countries:
        - Norway
        - Sweden
        - Finnland
        - Danmark
    number: 4
```

The arguments of `base.context` will be *merged* into the ScriptEngine context. This means that data can be added to existing data structures, such as lists or dictionaries:

```
- base.context:
    mylist:
        - one
        - two
# mylist is ['one', 'two']
- base.context:
    mylist:
        - 3
        - 4
# mylist is now ['one', 'two', 3, 4]
```

In the case of conflicts, the arguments of `base.context` will overwrite the current values in the ScriptEngine context. This can be used to clean data:

```
- base.context:
    mylist:
        - one
        - two
# mylist is ['one', 'two']
- base.context:
    mylist: null  # "remove" the value if mylist
```

(continues on next page)

(continued from previous page)

```
- base.context:
    mylist:
        - 3
        - 4
# mylist is now [3, 4]
```

CHAPTER 7

Indices and tables

- genindex