SALT.STATES.FILE

OPERATIONS ON REGULAR FILES, SPECIAL FILES, DIRECTORIES, AND SYMLINKS

~~Salt States can aggressively manipulate files on a system. There are a number of ways in which files can be managed.~~

The states.file module enables you to alter configuration files to complete tasks such as upgrading operating systems, changing permissions, security settings, adding and configuring nodes.

Virtual name:

(heading 2) For new Salt users:

For further information about states, see

- Configuration management Getting Started tutorial https://docs.saltproject.io/en/getstarted/config/functions.html
- Configuration management and states tutorials https://docs.saltproject.io/en/latest/topics/states/index.html
- Writing Salt states https://docs.saltproject.io/en/latest/ref/states/writing.html

Here is a quick guide to common salt.states.file examples. For more examples, see the functions reference below.

- Manage regular files with the `file.managed` state
- Use py renderer as a templating option
- Specify a file with the the source parameter
- Use the names parameter to expand the contents of a single state
- Manage special files with the mknod function
- Manage directories with the directory function
- Use the symlink function
- Manage directories recursively with the recurse function
- Manage backup directories with the retention schedule function

(link to bullet, new heading) Manage regular files with the file.managed state

Regular files can be enforced with the file.managed state. This state downloads files from the salt master and places them on the target system. Managed files can be rendered as a jinja, mako, or wempy template, adding a dynamic component to file management. ~~An example of file.managed which makes use of the jinja templating system would look like this:~~

For example, here file.managed uses the jinja templating system:

```
/etc/http/conf/http.conf:
  file.managed:
    - source: salt://apache/http.conf
    - user: root
    - group: root
    - mode: 644
    - attrs: ai
    - template: jinja
    - defaults:
        custom_var: "default value"
        other_var: 123
{% if grains['os'] == 'Ubuntu' %}
    - context:
        custom_var: "override"
{% endif %}
```

(link to bullet, new heading) Use py renderer as a templating option

It is also possible to use the py renderer as a templating option. The template would be a Python script which would need to contain a function called run(), which returns a string. All arguments to the state will be made available to the Python script as globals. The returned string will be the contents of the managed file. ~~For example:~~

For example:

```
def run():
    lines = ['foo', 'bar', 'baz']
    lines.extend([source, name, user, context])  # Arguments as globals
    return '\n\n'.join(lines)
```

Note:

The `defaults` and `context` arguments require extra indentation (four spaces instead of the normal two) in order to create a nested dictionary. ~~More information.~~ See YAML idiosyncrasies for more information.

If using a template, any user-defined template variables in the file defined in`source` must be passed in using the `defaults` and/or `context` arguments. The ~~general~~ best practice is to place default values in `defaults`, with conditional overrides going into `context`, as seen above.

The template will receive a variable `custom_var`, which would be accessed in the template using `{{ custom_var }}`. If the operating system is Ubuntu, the value of the variable `custom_var` would be *override*, otherwise it is the default *default value*

(link to bullet, new heading) Specify a file with the the source parameter

The `source` parameter can be specified as a list. If this is done, then the first file to be matched will be the one that is used. This allows you to have a default file on which to fall back if the desired file does not exist on the Salt fileserver. Here's an example:

```
/etc/foo.conf:
  file.managed:
    - source:
      - salt://foo.conf.{{ grains['fqdn'] }}
      - salt://foo.conf.fallback
    - user: foo
    - group: users
    - mode: 644
    - attrs: i
    - backup: minion
```

Note:

Salt supports backing up managed files via the backup option. For more details on this functionality please review the backup_mode documentation.

The `source` parameter can also specify a file in another Salt environment. In this example `foo.conf` in the `dev` environment will be used instead.

```
/etc/foo.conf:
  file.managed:
    - source:
      - 'salt://foo.conf?saltenv=dev'
    - user: foo
    - group: users
    - mode: '0644'
    - attrs: i
```

~~Warning~~: Caution:

When using a mode that includes a leading zero you must wrap the value in single quotes. If the value is not wrapped in quotes it will be read by YAML as an integer and evaluated as an octal.

(link to bullet, new heading) Use the names parameter to expand the contents of a single state

The `names` parameter, which is part of the state compiler, can be used to expand the contents of a single state declaration into multiple, single state declarations. Each item in the `names` list receives its own individual state `name` and is converted into its own low-data structure. This is a convenient way to manage several files with similar attributes.

```
salt_master_conf:
  file.managed:
    - user: root
    - group: root
    - mode: '0644'
    - names:
      - /etc/salt/master.d/master.conf:
        - source: salt://saltmaster/master.conf
      - /etc/salt/minion.d/minion-99.conf:
        - source: salt://saltmaster/minion.conf
```

Note:

There is more documentation about this feature in the Names declaration section of the Highstate docs.

(link to bullet, new heading) Manage special files with the mknod function

Special files can be managed via the `mknod` function. This function will create and enforce the permissions on a special file. The function supports the creation of character devices, block devices, and FIFO pipes. The function will create the directory structure up to the special file if it is needed on the minion. The function will not overwrite or operate on (change major/minor numbers) existing special files with the exception of user, group, and permissions. In most cases the creation of some special files require root permissions on the minion. This would require that the minion to be run as the root user. Here is an example of a character device:

```
/var/named/chroot/dev/random:
  file.mknod:
    - ntype: c
    - major: 1
    - minor: 8
    - user: named
    - group: named
    - mode: 660
```

Here is an example of a block device:

```
/var/named/chroot/dev/loop0:
  file.mknod:
    - ntype: b
    - major: 7
    - minor: 0
    - user: named
    - group: named
    - mode: 660
```

Here is an example of a fifo pipe:

```
/var/named/chroot/var/log/logfifo:
  file.mknod:
    - ntype: p
    - user: named
    - group: named
    - mode: 660
```

(link to bullet, new heading) Manage directories with the directory function

Directories can be managed via the `directory` function. This function can create and enforce the permissions on a directory. A directory statement will look like this:

```
/srv/stuff/substuf:
  file.directory:
    - user: fred
    - group: users
    - mode: 755
    - makedirs: True
```

~~If you need~~ To enforce user and/or group ownership or permissions recursively on the directory's contents, ~~you can do so by adding~~ add a `recurse` directive:

```
/srv/stuff/substuf:
  file.directory:
    - user: fred
    - group: users
    - mode: 755
    - makedirs: True
    - recurse:
      - user
      - group
      - mode
```

As a default, `mode` will resolve to `dir_mode` and `file_mode`. ~~, to~~ To specify both directory and file permissions, use this form:

```
/srv/stuff/substuf:
  file.directory:
    - user: fred
    - group: users
    - file_mode: 744
    - dir_mode: 755
    - makedirs: True
    - recurse:
      - user
      - group
      - mode
```

(link to bullet, new heading) Use symlinks

~~Symlinks can be easily created;~~ The symlink function ~~is very simple and~~ only takes a few arguments:

```
/etc/grub.conf:
  file.symlink:
    - target: /boot/grub/grub.conf
```

(link to bullet, new heading) Manage directories recursively with the recurse function

Recursive directory management can also be set via the `recurse` function. Recursive directory management allows for a directory on the salt master to be recursively copied down to the minion. This is a great tool for deploying large code and configuration systems. Here is an example of a ~~A~~ state using `recurse:` ~~would look something like this:~~

```
/opt/code/flask:
  file.recurse:
    - source: salt://code/flask
    - include_empty: True
```

A more complex `recurse` example:

```
{% set site_user = 'testuser' %}
{% set site_name = 'test_site' %}
{% set project_name = 'test_proj' %}
{% set sites_dir = 'test_dir' %}

django-project:
  file.recurse:
    - name: {{ sites_dir }}/{{ site_name }}/{{ project_name }}
    - user: {{ site_user }}
    - dir_mode: 2775
    - file_mode: '0644'
    - template: jinja
    - source: salt://project/templates_dir
    - include_empty: True
```

(link to bullet, new heading) Manage backup directories with the retention schedule function

Retention scheduling can be applied to manage contents of backup directories. For example:

```
/var/backups/example_directory:
  file.retention_schedule:
    - strptime_format: example_name_%Y%m%dT%H%M%S.tar.bz2
    - retain:
        most_recent: 5
        first_of_hour: 4
        first_of_day: 14
        first_of_week: 6
        first_of_month: 6
        first_of_year: all
```

(heading) Functions reference

`salt.states.file.absent(name, **kwargs)`

~~Make sure that the named file or directory is absent. If it exists, it will be deleted. This will work to reverse any of the functions in the file state module. If a directory is supplied, it will be recursively deleted.~~

Ensures that a named file or directory, if it exists, will be deleted. This will remove any file nodes that can be created with the states.file module, including hard- and symlinks. If a directory is supplied, it will be recursively deleted.

If only the contents of the directory need to be deleted but not the directory itself, use `file.directory` with `clean=True`

`name`

The path which should be deleted.

`salt.states.file.`**`accumulated`**_(name, filename, text, **kwargs)_

~~Prepare accumulator which can be used in template in file.managed state. Accumulator dictionary becomes available in template. It can also be used in file.blockreplace.~~

Creates an accumulator which can be used in a template in `file.managed state`. The accumulator dictionary becomes available in the template. It can also be used with `file.blockreplace`.

You can use accumulators to record one or more lists of items such as users, services, addresses, configuration commands, etc.

`name`

Accumulator name

`filename`

~~Filename which would receive this accumulator (see file.managed state documentation about `name`~~)

Filename of file to which the accumulator is applied (see `file.managed` state documentation for `name`)

`text`

~~String or list for adding in accumulator~~

String or list to add to the accumulator

`require_in / watch_in`

~~One of them required for sure we fill up accumulator before we manage the file. Probably the same as filename~~

One of `require_in / watch_in` or both is required to initiate the accumulator.

Example:

Given the following:

```
animals_doing_things:
  file.accumulated:
    - filename: /tmp/animal_file.txt
    - text: ' jumps over the lazy dog.'
    - require_in:
      - file: animal_file

animal_file:
  file.managed:
    - name: /tmp/animal_file.txt
    - source: salt://animal_file.txt
    - template: jinja
```

~~One might write a template for `animal_file.txt` like the following:~~
You can write the following template for `animal_file.txt:`

```
The quick brown fox{% for animal in
accumulator['animals_doing_things'] %}{{ animal }}{% endfor %}
```

~~Collectively, the above states and template file will produce:~~
The above states and template file will return:

```
The quick brown fox jumps over the lazy dog.
```

Multiple accumulators can be "chained" together.

Tip:
You can have multiple accumulators running in a given Salt run - the name is important.
Whatever runs the accumulator must run after all the steps that put data into the accumulator.

Note:
The 'accumulator' data structure is a Python dictionary. ~~Do not expect any~~ You cannot loop over the keys in a deterministic order.

Example 2:

Here `accumulated` is used in an accounts formula to build a list of users authorized to log in to a given host. This file can be used to fulfill various purposes, including as a pam limiter.

Various teams or groups of users may or may not be authorized on a given host. Each team/group is configured in a different formula file.

The accounts/TEAM.sls contains:

```
include:
  - accounts.nhgri_authorized_users

accounts - teams - unix - nhgri_authorized_users add unixteam users:
    file.accumulated:
    - name: users
    - text: |
        {{ salt['pillar.get']('unixteam', [])|join('\n')|indent(8) }}
    - filename: /etc/security/nhgri_authorized_users
    - require_in:
        - file: accounts - nhgri_authorized_users -
create_nhgri_authorized_users
```

`local_users` are assigned to various hosts and added as follows:

```
accounts - auth-ad - el8 - add local_users pillar to
nhgri_authorized_users accumulator:

  file.accumulated:
    - name: users
    - filename: /etc/security/nhgri_authorized_users
    - text: |
        {{ salt['pillar.get']('local_users', [])|join('\n')|indent(8)
}}
    - require_in:
        - file: accounts - nhgri_authorized_users -
create_nhgri_authorized_users
```

~~(basically the same code as above, just in a different formula/sub-formula)~~
The above code is the same as the preceding code, with a different formula/sub-formula

~~At the very end, in its own accounts/sub-formula, we have the single state that writes it all out:~~
Here is the single state that writes it all out:

```
accounts - nhgri_authorized_users - create_nhgri_authorized_users:
  file.managed:
    - source: salt://accounts/files/nhgri_authorized_users.j2
    - name: /etc/security/nhgri_authorized_users
    - user: root
    - group: root
    - mode: "0600"
    - template: jinja
```

salt.states.file.**append***(name, text=None, makedirs=False, source=None,* source_hash=None, template='jinja', sources=None, source_hashes=None, defaults=None, context=None, ignore_whitespace=True)

Ensures that some text appears at the end of a file.

The text will not be appended if it already exists in the file. A single string of text or a list of strings may be appended.

Use `append` when you have configuration files such as resolv.conf or networks where the system facility expects to see a specific immutable code block at the end of the config.

`name`
~~The location of the file to append to.~~
The location of the file to which the text will be appended.

`text`
The text to be appended, which can be a single string or a list of strings.

`makedirs`
If the file is located in a path without a parent directory, then the state will fail. If `makedirs` is set to `True`, then the parent directories will be created to facilitate the creation of the named file. Defaults to `False`.

`source`
A single source file to append. This source file can be hosted on either the Salt master server, or on an HTTP or FTP server. Both HTTPS and HTTP are supported as well as downloading directly from Amazon S3 compatible URLs with both pre-configured and automatic IAM credentials (see s3.get state documentation). File retrieval from Openstack Swift object storage is supported via swift://container/object_path URLs (see swift.get documentation).
For files hosted on the salt file server, if the file is located on the master in the directory named spam, and is called eggs, the source string is salt://spam/eggs.

If the file is hosted on an HTTP or FTP server, the `source_hash` argument is also required.

`source_hash`
This can be one of the following:

1. A source hash string.
2. The URI of a file that contains source hash strings.

The function accepts the first-encountered long unbroken alphanumeric string of correct length as a valid hash, in order from most secure to least secure:

```
Type    Length
======  ======
sha512     128
sha384      96
sha256      64
sha224      56
sha1        40
md5         32
```

See the source_hash parameter description for the `file.managed` function for more details and examples.

`template`
The named templating engine that will be used to render the appended-to file. Defaults to `jinja`. The following templates are supported:

- `cheetah`
- `genshi`
- `jinja`
- `mako`
- `py`
- `wempy`

`sources`
A list of source files to append. If the files are hosted on an HTTP or FTP server, the source_hashes argument is also required.

`source_hashes`
A list of source_hashes corresponding to the sources list specified in the sources argument.

`defaults`
Default context passed to the template.

`context`
Overrides default context variables passed to the template.

`ignore_whitespace`

~~New in version 2015.8.4.~~

~~Spaces and Tabs in text are ignored by default, when searching for the appending content, one space or multiple tabs are the same for salt. Set this option to False if you want to change this behavior.~~

Spaces and tabs in text are ignored by default when searching for the content to be appended. One space or multiple tabs are the same for Salt. Set this option to `False` to change this behavior.

~~Multi-line example:~~

Example - Multi-line:

```
/etc/motd:
  file.append:
    - text: |
        Thou hadst better eat salt with the Philosophers of Greece,
        than sugar with the Courtiers of Italy.
           - Benjamin Franklin
```

Example - Multiple lines of text:

```
/etc/motd:
  File.append:
   - text:
- Trust no one unless you have eaten much salt with him.
 - "Salt is born of the purest of parents: the sun and the sea."
```

Example - Gather text from multiple template files:

```
/etc/motd:
  file:f
    - append
    - template: jinja
    - sources:
      - salt://motd/devops-messages.tmpl
      - salt://motd/hr-messages.tmpl
      - salt://motd/general-messages.tmpl
```

~~New in version 0.9.5.~~

salt.states.file.**blockreplace**(*name, marker_start='#--start managed zone --', marker_end='#-- end managed zone --', source=None, source_hash=None, template='jinja', sources=None, source_hashes=None, defaults=None, context=None, content='', append_if_not_found=False, prepend_if_not_found=False, backup='.bak', show_changes=True, append_newline=None, insert_before_match=None, insert_after_match=None*)

~~Maintains~~ Allows an edit in a file in a ~~zone~~ section delimited by two line markers.

Blockreplace can be used in configuration files where the system facility expects to see a specific code block, and designated blocks to substitute your own configuration are provided.

~~New in version 2014.1.0.~~
~~Changed in version 2017.7.5,2018.3.1: append_newline argument added. Additionally, to improve idempotence, if the string represented by marker_end is found in the middle of the line, the content preceding the marker will be removed when the block is replaced. This allows one to remove append_newline: False from the SLS and have the block properly replaced if the end of the content block is immediately followed by themarker_end (i.e. no newline before the marker).~~

A block of content delimited by comments can help you manage several lines entries without worrying about ~~old entries removal~~ the removal of old entries. This can help you ~~maintaining~~ maintain an un-managed file containing manual edits.

~~**Note:**~~ **Tip**:
This function will store two copies of the file in-memory (the original version and the edited version) in order to detect changes and only edit the targeted file if necessary.
Additionally, you can use `file.accumulated` and target this state. All accumulated data dictionaries' content will be added in the content block.

`name`
Filesystem path to the file to be edited.

`marker_start`
The line content identifying~~ a line as ~~the start of the content block. Note that the whole line containing this marker will be considered, so whitespace or extra content before or after the marker is included in the final output.

`marker_end`
The line content identifying the end of the content block. As of versions 2017.7.5 and 2018.3.1, everything up to the text matching the marker will be replaced, so it's important to ensure that your marker includes the beginning of the text you wish to replace.

`content`
The content to be ~~used~~ placed between the two lines identified by `marker_start` and `marker_end`.

`source`
The source file to download to the minion, this source file can be hosted on either the Salt master server, or on an HTTP or FTP server. Both HTTPS and HTTP are supported as well as downloading directly from Amazon S3 compatible URLs with both pre-configured and automatic IAM credentials. (see s3.get state documentation)

(newline) File retrieval from Openstack Swift object storage is supported via swift://container/object_path URLs, see swift.get documentation. For files hosted on the Salt file server, if the file is located on the master in the directory named spam, and is called eggs, the source string is salt://spam/eggs.

(newline) If `source` is left blank or `None` (use ~ in YAML), the file will be created as an empty file and the content will not be managed. This is also the case when a file already exists and the source is undefined; the contents of the file will not be changed or managed.

If the file is hosted on a HTTP or FTP server then the `source_hash` argument is also required. A list of sources can also be passed in to provide a default source and a set of fallbacks. The first source in the list that is found to exist will be used and subsequent entries in the list will be ignored.

```
file_override_example:
  file.blockreplace:
    - name: /etc/example.conf
    - source:
      - salt://file_that_does_not_exist
      - salt://file_that_exists
```

`source_hash`
This can be one of the following:

1. ~~a~~ A source hash string.
2. ~~the~~ The URI of a file that contains source hash strings.

The function accepts the first encountered long unbroken alphanumeric string of correct length as a valid hash, in order from most secure to least secure:

```
Type     Length
======   ======
sha512      128
sha384       96
sha256       64
```

sha224	56
sha1	40
md5	32

See the source_hash parameter description for the `file.managed` function for more details and examples.

`template`
Specifies the templating engine ~~to be~~ used to render the downloaded file. The following engines are supported:

- `cheetah`
- `genshi`
- `jinja`
- `mako`
- `py`
- `wempy`

`context`
Overrides default context variables passed to the template.

`defaults`
Default context passed to the template.

`append_if_not_found`
If markers are not found and this option is set to `True`, the content block will be appended to the file.

`prepend_if_not_found`
If markers are not found and this option is set to `True`, the content block will be prepended to the file.

`insert_before_match`
If markers are not found, this parameter can be set to a regex which will insert the block before the first found occurrence in the file.
~~New in version 3001.~~

`insert_after_match`
If markers are not found, this parameter can be set to a regex which will insert the block after the first found occurrence in the file.
~~New in version 3001.~~

`backup`
The file extension to use for a backup of the file if any edit is made. Set this to `False` to skip making a backup.

`dry_run`
If `True`, do not make any edits to the file and simply return the changes that would be made.

`show_changes`
Controls how changes are presented. If `True`, the Changes section of the state return will contain a unified diff of the changes made. If `False`, then it will contain a boolean (`True` if any changes were made, otherwise `False`).

`append_newline`
Controls whether or not a newline is appended to the content block. If the value of this argument is `True` then a newline will be added to the content block. If it is `False`, then a newline will not be added to the content block. If it is unspecified, then a newline will only be added to the content block if it does not already end in a newline.
~~New in version 2017.7.5,2018.3.1.~~

Example of usage with an accumulator and with a variable:

```
{% set myvar = 42 %}
hosts-config-block-{{ myvar }}:
  file.blockreplace:
    - name: /etc/hosts
    - marker_start: "# START managed zone {{ myvar }} -DO-NOT-EDIT-"
    - marker_end: "# END managed zone {{ myvar }} --"
    - content: 'First line of content'
    - append_if_not_found: True
    - backup: '.bak'
    - show_changes: True

hosts-config-block-{{ myvar }}-accumulated1:
  file.accumulated:
    - filename: /etc/hosts
    - name: my-accumulator-{{ myvar }}
    - text: "text 2"
    - require_in:
      - file: hosts-config-block-{{ myvar }}

hosts-config-block-{{ myvar }}-accumulated2:
  file.accumulated:
    - filename: /etc/hosts
    - name: my-accumulator-{{ myvar }}
    - text: |
         text 3
         text 4
    - require_in:
      - file: hosts-config-block-{{ myvar }}

will generate and maintain a block of content in /etc/hosts:
# START managed zone 42 -DO-NOT-EDIT-
First line of content
text 2
text 3
text 4
# END managed zone 42 –
```

salt.states.file.cached(*name*, *source_hash=''*, *source_hash_name=None*, *skip_verify=False*, *saltenv='base'*, *use_etag=False*)

~~New in version 2017.7.3.~~
~~Changed in version 3005.~~

Ensures that a file is saved to the minion's cache. This state is primarily invoked by other states to ensure that we do not re-download a source file if we do not need to.

`name`
The URL of the file to be cached. To cache a file from an environment other than base, either use the `saltenv` argument or include the `saltenv` in the URL (e.g. `salt://path/to/file.conf?saltenv=dev`).

Note:
A list of URLs is not supported, this must be a single URL. If a local file is passed here, then the state will ~~obviously~~ not try to download anything, but it will compare a hash if one is specified.

`source_hash`
See the documentation for this same argument in the `file.managed` state.

Note:
For remote files not originating from the salt:// fileserver, such as http(s) or ftp servers, this state will not re-download the file if the locally-cached copy matches this hash. This is done to prevent unnecessary downloading on repeated runs of this state. To update the cached copy of a file, it is necessary to update this hash.

`source_hash_name`
See the documentation for this same argument in the `file.managed` state.

`skip_verify`

See the documentation for this same argument in the `file.managed` state.

Note:
Setting this to True will result in a copy of the file being downloaded from a remote (http(s), ftp, etc.) source each time the state is run.

`saltenv`
Used to specify the environment from which to download a file from the Salt fileserver (i.e. those with salt:// URL).

`use_etag`
If `True`, remote http/https file sources will attempt to use the ETag header to determine if the remote file needs to be downloaded. This provides a lightweight mechanism for promptly refreshing files changed on a web server without requiring a full hash comparison via the source_hash parameter.
~~New in version 3005.~~

Examples:

This state will in most cases not be useful in SLS files, but it is useful when writing a state or remote-execution module that needs to make sure that a file at a given URL has been downloaded to the cachedir. One example of this is in the archive.extracted state:

```
result = __states__['file.cached'](source_match,
                                   source_hash=source_hash,
                                   source_hash_name=source_hash_name,
                                   skip_verify=skip_verify,
                                   saltenv=__env__)
```

This will return a dictionary containing the state's return data, including a result key which will state whether or not the state was successful. Note that this will not catch exceptions, so it is best used within a try/except.

Once this state has been run from within another state or remote-execution module, the actual location of the cached file can be obtained using cp.is_cached:

```
cached = __salt__['cp.is_cached'](source_match, saltenv=__env__)
```

This function will return the cached path of the file, or an empty string if the file is not present in the minion cache.

salt.states.file.comment(***name**, **regex**, **char='#'**, **backup='.bak'**, **ignore_missing=False***)

~~New in version 0.9.5.~~

~~Changed in version 3005.~~

Comments out specified lines in a file.

For example, when a package ships a default configuration file, but doesn't support overrides in separate files, you can comment-out a single item.

`name`
The full path to the file to be edited.

`regex`
A regular expression used to find the lines that are to be commented; this pattern will be wrapped in parenthesis and will move any preceding/trailing ^ or $ characters outside the parenthesis (e.g., the pattern ^foo$ will be rewritten as ^(foo)$). ~~Note that you _need_ the leading ^, otherwise each time you run highstate, another comment char will be inserted.~~

Note: The leading ^ is required, otherwise each time you run highstate, another comment char will be inserted.

`char`
The character to be inserted at the beginning of a line in order to comment it out

`backup`
~~The file will be backed up before edit~~ Before the file is changed a backup file will be generated with this file extension

~~Warning~~: **Caution**:
This backup will be overwritten each time sed / comment / uncomment is called. ~~Meaning~~ t The backup will only be useful after the first invocation.

Set to `False/None` to not keep a backup.

`ignore_missing`
Ignore a failure to find the regex in the file. This is useful for scenarios where a line must only be commented if it is found in the file.

~~New in version 3005.~~

~~Usage:~~
Example:

```
/etc/fstab:
  file.comment:
    - regex: ^bind 127.0.0.1
```

`salt.states.file.`**`copy_`**`(`*`name`*`, `*`source`*`, `*`force=False`*`, `*`makedirs=False`*`, `*`preserve=False`*`, `*`user=None`*`, `*`group=None`*`, `*`mode=None`*`, `*`subdir=False`*`, `*`**kwargs`*`)`

Copies a file to the named path if the file defined by the source option exists on the minion. ~~copy it to the named path~~. The file will not be overwritten if it already exists, unless the `force` option is set to `True`.

Note:
This state only copies files from one location on a minion to another location on the same minion. For copying files from the master, use a `file.managed` state.

`name`
~~The location of the file to copy to~~
The location where the file is to be copied.

`source`
~~The location of the file to copy to the location specified with name~~
The current location of the file that will be copied to the location specified by name.

`force`
~~If the target location is present then the file will not be moved, specify "force: True" to overwrite the target file~~
If the target location is present then the file will not be moved. Specify `force: True` to overwrite the target file.

`makedirs`
~~If the target subdirectories don't exist create them~~
If the target subdirectories don't exist, create them by specifying `makedirs: True`

`preserve`
~~New in version 2015.5.0.~~
Set `preserve: True` to preserve user/group ownership and mode after copying. Default is `False`. If preserve is set to `True`, then user/group/mode attributes will be ignored.

`user`
~~New in version 2015.5.0.~~
~~The user to own the copied file, this defaults to the user salt is running as on the minion. If preserve is set to True, then this will be ignored~~

The user who will own the copied file. This defaults to the user Salt is running as on the minion. If preserve is set to `True`, then this will be ignored.

`group`

~~New in version 2015.5.0.~~

~~The group to own the copied file, this defaults to the group salt is running as on the minion. If preserve is set to True or on Windows this will be ignored~~

The group that will own the copied file. This defaults to the group Salt is running as on the minion. If preserve is set to `True` or on Windows this will be ignored.

`mode`

~~New in version 2015.5.0.~~

The permissions to set on the copied file, aka `644`, `'0775'`, `'4664'`. If preserve is set to True, then this will be ignored. Not supported on Windows. The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

`subdir`

~~New in version 2015.5.0.~~

If the name is a directory then place the file inside the named directory

Note:

The copy function accepts paths that are local to the Salt minion. This function does not support salt://, http://, or the other additional file paths that are supported by `states.file.managed` and `states.file.recurse`.

~~Usage:~~

Example:

```
# Use 'copy', not 'copy_'
/etc/example.conf:
  file.copy:
    - source: /tmp/example.conf
```

salt.states.file.decode(*name*, *encoded_data=None*, *contents_pillar=None*, *encoding_type='base64'*, *checksum='md5'*)

Decodes an encoded file and write it to disk.

~~New in version 2016.3.0.~~

`name`

Path of the file to be written.

`encoded_data`
The encoded file. Either this option or `contents_pillar` must be specified.

`contents_pillar`
A Pillar path to the encoded file. Uses the same path syntax as [`pillar.get`](). The [`hashutil.base64_encodefile`]() function can load encoded content into Pillar. Either this option or `encoded_data` must be specified.

`encoding_type`
The type of encoding.

`checksum`
The hashing algorithm to use to generate checksums. Wraps the [`hashutil.digest`]() execution function.

~~Usage:~~
Example:

```
write_base64_encoded_string_to_a_file:
  file.decode:
    - name: /tmp/new_file
    - encoding_type: base64
    - contents_pillar: mypillar:thefile

# or

write_base64_encoded_string_to_a_file:
  file.decode:
    - name: /tmp/new_file
    - encoding_type: base64
    - encoded_data: |
        Z2V0IHNhbHRlZAo=
```

Be careful with multi-line strings that the YAML indentation is correct. ~~E.g.~~,

For example:

```
write_base64_encoded_string_to_a_file:
  file.decode:
    - name: /tmp/new_file
    - encoding_type: base64
    - encoded_data: |
        {{ salt.pillar.get('path:to:data') | indent(8) }}
```

salt.states.file.**directory**(***name***, ***user=None***, ***group=None***, ***recurse=None***, ***max_depth=None***, ***dir_mode=None***, ***file_mode=None***, ***makedirs=False***, ***clean=False***, ***require=None***, ***exclude_pat=None***, ***follow_symlinks=False***, ***force=False***, ***backupname=None***, ***allow_symlink=True***, ***children_only=False***, ***win_owner=None***, ***win_perms=None***, ***win_deny_perms=None***, ***win_inheritance=True***, ***win_perms_reset=False***, *****kwargs***)

Ensures that a named directory is present and has the ~~right perms~~ correct permissions. `directory` can be used to individually create directories, setting permissions and ownership for each as needed.

`name`
~~The location to create or manage a directory, as an absolute path~~
The location where a directory is to be created or managed, as an absolute path

`user`
~~The user to own the directory; this defaults to the user salt is running as on the minion~~
The user who will own the directory. Defaults to the user salt is running as on the minion.

`group`
The group ownership set for the directory; ~~this~~ Defaults to the group salt is running as on the minion. On Windows, this is ignored.

`recurse`
Enforces user/group ownership and mode of directory recursively. Accepts a list of strings representing what you would like to recurse.
(newline)
If mode is defined, will recurse on both `file_mode` and `dir_mode` if they are defined. If `ignore_files` or `ignore_dirs` is included, files or directories will be left unchanged respectively. If silent is defined, individual file/directory change notifications will be suppressed.

Example:
```
/var/log/httpd:
  file.directory:
    - user: root
    - group: root
    - dir_mode: 755
    - file_mode: 644
    - recurse:
      - user
      - group
      - mode
```

Leave files or directories unchanged:

```
/var/log/httpd:
  file.directory:
    - user: root
    - group: root
    - dir_mode: 755
    - file_mode: 644
    - recurse:
      - user
      - group
      - mode
      - ignore_dirs
```

~~New in version 2015.5.0.~~

`max_depth`
Limits the recursion depth. The default is no `limit=None`. `max_depth` and `clean` are mutually exclusive.
~~New in version 2016.11.0.~~

`dir_mode / mode`
The permissions mode to set any directories created. Not supported on Windows.
The default mode for new files and directories corresponds to umask of salt process. For existing files and directories it's not enforced.

`file_mode`
The permissions mode to set any files created if 'mode' is run in 'recurse'. This defaults to `dir_mode`. Not supported on Windows. The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

`makedirs`
If the directory is located in a path without a parent directory, then the state will fail. If makedirs is set to `True`, then the parent directories will be created to facilitate the creation of the named file.

`clean`
Removes any files that are not referenced by a required file state. See examples below for more info. If this option is set then everything in this directory will be deleted unless it is required. `clean` and `max_depth` are mutually exclusive.

`require`
Requires other resources such as packages or files.

`exclude_pat`
When `clean` is set to `True`, exclude this pattern from the removal list and preserve it in the destination.

`follow_symlinks`
If the desired path is a symlink (or recurse is defined and a symlink is encountered while recursing), follow it and check the permissions of the directory/file to which the symlink points.
~~New in version 2014.1.4.~~
~~Changed in version 3001.1:~~ If set to `False` symlinks permissions are ignored on Linux systems because it does not support permissions modification. Symlinks permissions are always `0o777` on Linux.

`force`
If the name of the directory exists and is not a directory and force is set to `False`, the state will fail. If force is set to `True`, the file in the way of the directory will be deleted to make room for the directory, unless `backupname` is set, then it will be renamed.
~~New in version 2014.7.0.~~

`backupname`
If the name of the directory exists and is not a directory, it will be renamed to the `backupname`. If the `backupname` already exists and force is `False`, the state will fail. Otherwise, the `backupname` will be removed first.
~~New in version 2014.7.0.~~

`allow_symlink`
If allow_symlink is `True` and the specified path is a symlink, it will be allowed to remain if it points to a directory. If `allow_symlink` is `False` then the state will fail, unless force is also set to `True`, in which case it will be removed or renamed, depending on the value of the `backupname` argument.
~~New in version 2014.7.0.~~

`children_only`
If `children_only` is `True` the base of a path is excluded when performing a recursive operation. In case of /path/to/base, base will be ignored while all of /path/to/base/* are still operated on.

`win_owner`
The owner of the directory. If this is not passed, user will be used. If user is not passed, the account under which Salt is running will be used.

~~New in version 2017.7.0.~~

`win_perms`
A dictionary containing permissions to grant and their propagation. For example: `{'Administrators': {'perms': 'full_control', 'applies_to': 'this_folder_only'}}` Can be a single basic perm or a list of advanced perms. perms must be specified. `applies_to` is optional and defaults to this_folder_subfolder_files.
~~New in version 2017.7.0.~~

`win_deny_perms`
A dictionary containing permissions to deny and their propagation. For example: `{'Administrators': {'perms': 'full_control', 'applies_to': 'this_folder_only'}}` Can be a single basic perm or a list of advanced perms.
~~New in version 2017.7.0.~~

`win_inheritance`
Set to `True` to inherit permissions from the parent directory. Set to `False` ~~not to~~ to not inherit permission.
~~New in version 2017.7.0.~~

`win_perms_reset`
If `True` the existing DACL will be cleared and replaced with the settings defined in this function. If `False`, new entries will be appended to the existing DACL. Default is `False`.
~~New in version 2018.3.0.~~

Examples:

~~Here's an example using the above win_* parameters:~~
The following example uses the above `win_*` parameters:

```
create_config_dir:
  file.directory:
    - name: 'C:\config\'
    - win_owner: Administrators
    - win_perms:
        # Basic Permissions
        dev_ops:
          perms: full_control
        # List of advanced permissions
        appuser:
          perms:
            - read_attributes
            - read_ea
            - create_folders
            - read_permissions
          applies_to: this_folder_only
        joe_snuffy:
          perms: read
          applies_to: this_folder_files
    - win_deny_perms:
        fred_snuffy:
          perms: full_control
    - win_inheritance: False
```

For `clean: True` there is no mechanism that allows all states and modules to enumerate the files that they manage, so for `file.directory` to know what files are managed by Salt, a file state targeting managed files is required. ~~To use a contrived~~

For example, the following states will always have changes, despite the file named `okay` being created by a Salt state:

```
silly_way_of_creating_a_file:
  cmd.run:
     - name: mkdir -p /tmp/dont/do/this && echo "seriously" >
/tmp/dont/do/this/okay
     - unless: grep seriously /tmp/dont/do/this/okay

will_always_clean:
  file.directory:
    - name: /tmp/dont/do/this
    - clean: True
```

Because `cmd.run` has no way of communicating that it's creating a file, `will_always_clean` will remove the newly created file. ~~Of course,~~ Every time the states run the same thing will happen - the `silly_way_of_creating_a_file` will create the file and `will_always_clean` will always remove it, ~~Over and over again, no matter~~ regardless of how many times you run it.

To make this example work correctly, we need to add a file state that targets the file, and a require between the file states:

```
silly_way_of_creating_a_file:
  cmd.run:
     - name: mkdir -p /tmp/dont/do/this && echo "seriously" >
/tmp/dont/do/this/okay
     - unless: grep seriously /tmp/dont/do/this/okay
  file.managed:
     - name: /tmp/dont/do/this/okay
     - create: False
     - replace: False
     - require_in:
       - file: will_always_clean
```

Now there is a file state that clean can check, so running those states will work as expected. The file will be created with the specific contents, and clean will ignore the file because it is being managed by a Salt file state. Note that if `require_in` was placed under `cmd.run`, it would not work, because the requisite is for the cmd, not the file.

```
silly_way_of_creating_a_file:
  cmd.run:
     - name: mkdir -p /tmp/dont/do/this && echo "seriously" > 
/tmp/dont/do/this/okay
     - unless: grep seriously /tmp/dont/do/this/okay
     # This part should be under file.managed
     - require_in:
       - file: will_always_clean
  file.managed:
     - name: /tmp/dont/do/this/okay
     - create: False
     - replace: False
```

Any other state that creates a file as a result, for example `pkgrepo`, must have the resulting files referenced in a file state in order for `clean: True` to ignore them. Also note that the requisite (`require_in` vs `require`) works in both directions:

```
clean_dir:
  file.directory:
    - name: /tmp/a/better/way
    - require:
      - file: a_better_way

a_better_way:
  file.managed:
    - name: /tmp/a/better/way/truely
    - makedirs: True
    - contents: a much better way
```

Works ~~the same as~~ similarly as in this example:

```
clean_dir:
  file.directory:
    - name: /tmp/a/better/way
    - clean: True

a_better_way:
  file.managed:
    - name: /tmp/a/better/way/truely
    - makedirs: True
    - contents: a much better way
    - require_in:
      - file: clean_dir
```

A common mistake here is to forget that the state `name` and `id` are both required for requisites:

```
# Correct:
/path/to/some/file:
  file.managed:
    - contents: Cool
    - require_in:
      - file: clean_dir

# Incorrect
/path/to/some/file:
  file.managed:
    - contents: Cool
    - require_in:
      # should be `- file: clean_dir`
      - clean_dir

# Also incorrect
/path/to/some/file:
  file.managed:
    - contents: Cool
    - require_in:
      # should be `- file: clean_dir`
      - file
```

salt.states.file.exists(name, **kwargs)

Verifies that the named file or directory is present or exists. Ensures pre-requisites outside of Salt's purview (e.g., keytabs, private keys, etc.) have been previously satisfied before deployment.

This function does not create the file if it doesn't exist, ~~it~~ and will return an error.

Tip:
Use absent if you want to remove the file or directory, and use an execution module if you just want to check.

`name`
Absolute path which must exist.

`salt.states.file.`**`hardlink`***(name, target, force=False, makedirs=False, user=None, group=None, dir_mode=None, **kwargs)*

Creates a hard link. If the file already exists and is a hard link pointing to any location other than the specified target, the hard link will be replaced. If the hard link is a regular file or directory then the state will return `False`. If the regular file is desired to be replaced with a hard link, pass `force`: `True`

`name`
The location of the hard link to create.

`target`
The location that the hard link points to.

`force`
If the name of the hard link exists and force is set to `False`, the state will fail. If `force` is set to `True`, the file or directory in the way of the hard link file will be deleted to make room for the hard link, unless backupname is set, when it will be renamed

`makedirs`
If the location of the hard link does not already have a parent directory, then the state will fail. Setting `makedirs` to `True` will allow Salt to create the parent directory.

`user`
The user to own any directories made if `makedirs` is set to `True`. This defaults to the user Salt is running as on the minion.

`group`
The group ownership set on any directories made if `makedirs` is set to `True`. This defaults to the group Salt is running as on the minion. On Windows, this is ignored.

`dir_mode`
If directories are to be created, passing this option specifies the permissions for those directories.

Example:

Create hardlink:

```
  File.hardlink:
    - name: /tmp/new_hardlink
    - target: /tmp/oldfile
```

salt.states.file.keyvalue(*name, key=None, value=None, key_values=None, separator='=', append_if_not_found=False, prepend_if_not_found=False, search*

False, show_changes=True, ignore_if_missing=False, count=1, uncomment=None, key_ignore_case=False, value_ignore_case=False)

Enables Key/Value based editing of a file. A readable and easily-maintained function that enables modifications to configuration files.

~~New in version 3001.~~

This function differs from `file.replace` in that it is able to search for keys, followed by a customizable separator, and replace the value with the given value. Should the value be the same as the one already in the file, no changes will be made.

Either supply both key and value parameters, or supply a dictionary with key / value pairs. It is an error to supply both.

`name`
Name of the file ~~to~~ in which to search/replace ~~in~~.

`key`
Key to search for when ensuring a value. Use in combination with a value parameter.

`value`
Value to set for a given key. Use in combination with a key parameter.

`key_values`
Dictionary of key/value pairs to search and ensure values for. ~~and ensure values for~~. Used to specify multiple key/values at once.

`separator`
Separator which separates key from value.

`append_if_not_found`
Append the key/value to the end of the file if not found. Note that this takes precedence over

`prepend_if_not_found`.

`prepend_if_not_found`
Prepend the key/value to the beginning of the file if not found. Note that `append_if_not_found` takes precedence.

`show_changes`
Show a diff of the resulting removals and inserts.

`ignore_if_missing`
Return with success even if the file is not found (or not readable).

`count`
Number of occurrences to allow (and correct)~~,~~. Default is `1`. Set to -`1` to replace all, or set to `0` to remove all lines with this key regardless of its value.

Note:
Any additional occurrences after `count` are removed. A `count` of `-1` will only replace all occurrences that are currently uncommented already. Lines commented out will be left alone.

`uncomment`
Disregard and remove supplied leading characters when finding keys. When set to `None`, lines that are commented out are left ~~for what they are~~ as-is.

Note:
The argument to uncomment is not a prefix string. ~~Rather; I~~It is a set of characters, each of which are stripped.

`key_ignore_case`
Keys are matched case insensitively. When a value is changed the matched key is kept as-is.

`value_ignore_case`
Values are checked case insensitively, trying to set e.g. 'Yes' while the current value is 'yes', will not result in changes when `value_ignore_case` is set to `True`.

Examples:

An example of using `file.keyvalue` to ensure sshd does not allow for root to login with a password and ~~at the same time~~ while setting the login-gracetime to 1 minute and disabling all forwarding:

```
sshd_config_harden:
    file.keyvalue:
      - name: /etc/ssh/sshd_config
```

```
   - key_values:
     permitrootlogin: 'without-password'
     LoginGraceTime: '1m'
     DisableForwarding: 'yes'
   - separator: ' '
   - uncomment: '# '
   - key_ignore_case: True
   - append_if_not_found: True
```

The same example, ~~except for only ensuring~~ with `permitrootlogin` ~~is~~ set correctly. ~~Thus being able to~~ Therefore you can use the shorthand `key` and `value` parameters instead of `key_values`.

```
sshd_config_harden:
  file.keyvalue:
   - name: /etc/ssh/sshd_config
   - key: PermitRootLogin
   - value: without-password
   - separator: ' '
   - uncomment: '# '
   - key_ignore_case: True
   - append_if_not_found: True
```

Note:
~~Notice how~~ In the above example, the key is not matched case-sensitively – this way it will correctly identify both `'PermitRootLogin'` as well as `'permitrootlogin'`.

The following example has a Zabbix configuration file with many defaults, where you only want to keep track of two of the fields:

```
Configure Zabbix Agent:
 file.keyvalue:
  - name: /etc/zabbix/zabbix_agent2.conf
  - key_values:
    Server: zabbix
    Hostname: {{ salt.grains.get('id') }}
  - prepend_if_not_found: True
  - count: -1
```

`salt.states.file.line`*(name, content=None, match=None, mode=None, location=None, before=None, after=None, show_changes=True, backup=False, quiet=False, indent=True, create=False, user=None, group=None, file_mode=None)*

Allows line-focused editing of a file.

~~New in version 2015.8.0.~~

Note:
`file.line` exists for historic reasons, and is not generally recommended. file.replace is recommended.

`file.line` is most useful if you have single lines in a file, ~~potentially a~~ or config file that you ~~would like~~ want to manage. It can remove, add, and replace lines.

`name`
Filesystem path to the file to be edited.

`content`
Content of the line. Allowed to be empty if `mode=delete`.

`match`
Match the target line for an action by a fragment of a string or regular expression.
If neither before nor after are provided, and match is also None, match falls back to the content value.

`mode`
Defines how to edit a line. One of the following options is required:

- `ensure`
  If a line does not exist, it will be added. If before and afte rare specified either zero lines, or lines that contain the content line are allowed to be in between before and after. If there are lines, and none of them match then it will produce an error.
- `replace`
  If a line already exists, it will be replaced.
- `delete`
  Delete the line, if found.
- `insert`
  Nearly identical to `ensure`. If a line does not exist, it will be added.
  The differences are that multiple (and non-matching) lines are allowed between before and after, if they are specified. The line will always be inserted right before before. `insert` also allows the use of location to specify that the line should be added at the beginning or end of the file.

Note:

If `mode=insert` is used, at least one of the following options must also be defined: location, `before`, or `after`. If `location` is used, it takes precedence over the other two options.

`location`
In `mode=insert` only, whether to place the content at the beginning or end of a ~~the~~ file. If location is provided, before and after are ignored. Valid locations:

- `start`
  Place the content at the beginning of the file.
- `end`
  Place the content at the end of the file.

`before`
Regular expression or an exact case-sensitive fragment of the string. Will be tried as both a regex and a part of the line. Must match exactly one line in the file. This value is only used in ensure and insert modes. The content will be inserted just before this line, matching its indent unless `indent=False`.

`after`
Regular expression or an exact case-sensitive fragment of the string. Will be tried as both a regex and a part of the line. Must match exactly one line in the file. This value is only used in ensure and insert modes. The content will be inserted directly after this line, unless before is also provided. If before is not matched, indentation will match this line, unless `indent=False`.

`show_changes`
Outputs a unified diff of the old file and the new file. If `False` return a boolean if any changes were made. Default is `True`.

Note:
Using this option will store two copies of the file in-memory (the original version and the edited version) in order to generate the diff.

`backup`
Creates a backup of the original file with the extension:
"Year-Month-Day-Hour-Minutes-Seconds".

`quiet`
~~Do not raise any exceptions.~~ Turns off notifications about exception errors. For example, when the file to be edited does not exist. ~~E.g. ignore the fact that the file that is tried to be edited does not exist and nothing really happened.~~

`indent`
Keeps indentation with the previous line. This option is not considered when the delete mode is specified. Default is `True`.

`create`
Creates an empty file if doesn't exist.
~~New in version 2016.11.0.~~

`user`
Specifies the user to own the file, this defaults to the user Salt is running as on the minion.
~~New in version 2016.11.0.~~

`group`
Specifies the group ownership set for the file, this defaults to the group Salt is running as on the minion On Windows, this is ignored.
~~New in version 2016.11.0.~~

`file_mode`
Specifies the permissions to set on this file, aka `644`, `0775`, `4664`. Not supported on Windows.
~~New in version 2016.11.0.~~

If an equal sign (=) appears in an argument to a Salt command, it is interpreted as a keyword argument in the format of `key=val`. That processing can be bypassed in order to pass an equal sign through to the remote shell command by manually specifying the kwarg:

```
update_config:
  file.line:
    - name: /etc/myconfig.conf
    - mode: ensure
    - content: my key = my value
    - before: somekey.*?
```

Examples:

```
Here's a simple config file.
[some_config]
# Some config file
# this line will go away

here=False
away=True
goodybe=away
```

And an sls file:

```
remove_lines:
  file.line:
    - name: /some/file.conf
    - mode: delete
    - match: away
```

This will produce:

```
[some_config]
# Some config file

here=False
away=True
goodbye=away
```

If that state is executed 2 more times, this will be the result:

```
[some_config]
# Some config file

here=False
```

Given that original file with this state:

```
replace_things:
  file.line:
    - name: /some/file.conf
    - mode: replace
    - match: away
    - content: here
```

Three passes will this state will result in this file:

```
[some_config]
# Some config file
here

here=False
here
here
```

Each pass replacing the first line found.

Given this file:

```
insert after me
something
insert before me
```

The following state:

```
insert_a_line:
  file.line:
    - name: /some/file.txt
    - mode: insert
    - after: insert after me
    - before: insert before me
    - content: thrice
```

If this state is executed 3 times, the result will be:

```
insert after me
something
thrice
thrice
thrice
insert before me
```

If the mode is `ensure` instead, it will fail each time. To succeed, we need to remove the incorrect line between before and after:

```
insert after me
insert before me
```

With an ensure mode, this will insert `thrice` the first time and make no changes for subsequent calls. For something simple this is fine, but if you have instead blocks like this:

```
Begin SomeBlock
    foo = bar
End

Begin AnotherBlock
    another = value
End
```

And given this state:

```
ensure_someblock:
  file.line:
    - name: /some/file.conf
    - mode: ensure
    - after: Begin SomeBlock
    - content: this = should be my content
    - before: End
```

This will fail because there are multiple `End` lines. Without that problem, it still would fail because there is a non-matching line, `foo = bar`. Ensure only allows either zero, or the matching line present to be present in between `before` and `after`.

`salt.states.file.`**`managed`**(*name, source=None, source_hash='', source_hash_name=None, keep_source=True, user=None, group=None, mode=None, attrs=None,* *template=None, makedirs=False, dir_mode=None, context=None, replace=True, defaults=None, backup='', show_changes=True, create=True, contents=None, tmp_dir='', tmp_ext='', contents_pillar=None, contents_grains=None, contents_newline=True, contents_delimiter=':', encoding=None, encoding_errors='strict', allow_empty=True, follow_symlinks=True, check_cmd=None, skip_verify=False, selinux=None, win_owner=None, win_perms=None, win_deny_perms=None, win_inheritance=True, win_perms_reset=False, verify_ssl=True, use_etag=False, **kwargs)*

~~Manage a given file, this function allows for a file to be downloaded from the salt master and potentially run through a templating system.~~

`managed` allows you to create or control the contents and metadata of a regular file. It allows for a file to be downloaded from the Salt master and potentially run through a templating system.

`managed` is useful when you wish to control the contents of the entire file, instead of controlling a block (`file.blockreplace`), a comment (`file.comment`), or certain options (if a common config, `file.keyvalue`).

`name`
The location of the file to manage as an absolute path.

`source`
The source file to download to the minion. This source file can be hosted on either the salt master server (`salt://`), the salt minion local file system (`/`), or on an HTTP or FTP server (`http(s)://, ftp://`).

~~Both HTTPS and HTTP are supported as well as downloading directly from Amazon S3 compatible URLs with both pre-configured and automatic IAM credentials. (see s3.get state documentation) File retrieval from Openstack Swift object storage is supported via swift://container/object_path URLs, see swift.get documentation.~~

The following are supported:

- HTTPS and HTTP
- Downloading directly from Amazon S3 compatible URLs with both pre-configured and automatic IAM credentials (see s3.get state documentation)
- File retrieval from Openstack Swift object storage via swift://container/object_path URLs. See swift.get documentation.

For files hosted on the Salt file server, if the file is located on the master in the directory named spam, and is called eggs, the source string is salt://spam/eggs. (added newline)

If source is left blank or None (use ~ in YAML), the file will be created as an empty file and the content will not be managed. This is also the case when a file already exists and the source is undefined; the contents of the file will not be changed or managed. (added newline)

If source is left blank or None, please also set replaced to False to make your intention explicit. (added newline)

If the file is hosted on a HTTP or FTP server then the source_hash argument is also required. A list of sources can also be passed in to provide a default source and a set of fallbacks. The first source in the list that is found to exist will be used and subsequent entries in the list will be ignored. Source list functionality only supports local files and remote files hosted on the Salt master server or retrievable via HTTP, HTTPS, or FTP.

```
file_override_example:
  file.managed:
    - source:
      - salt://file_that_does_not_exist
      - salt://file_that_exists
```

source_hash

This can be one of the following:

1. A source hash string
2. The URI of a file that contains source hash strings

The function accepts the first encountered long unbroken alphanumeric string of correct length as a valid hash, in order from most secure to least secure:

Type	Length
sha512	128
sha384	96
sha256	64
sha224	56
sha1	40
md5	32

Using a Source Hash File

The file can contain several checksums for several files. Each line must contain both the file name and the hash. If no file name is matched, the first hash encountered will be used, otherwise the most secure hash with the correct source file name will be used.

When using a source hash file the `source_hash` argument needs to be a url. The standard download urls are supported, ftp, http, Salt etc:

Example:

```
tomdroid-src-0.7.3.tar.gz:
  file.managed:
    - name: /tmp/tomdroid-src-0.7.3.tar.gz
    - source:
https://launchpad.net/tomdroid/beta/0.7.3/+download/tomdroid-
src-0.7.3.tar.gz
    - source_hash:
https://launchpad.net/tomdroid/beta/0.7.3/+download/tomdroid-
src-0.7.3.hash
```

The following lines are all supported formats:

```
/etc/rc.conf ef6e82e4006dee563d98ada2a2a80a27
sha254c8525aee419eb649f0233be91c151178b30f0dff8ebbdcc8de71b1d5c8bcc06
a  /etc/resolv.conf
ead48423703509d37c4a90e6a0d53e143b6fc268
```

Debian file type `*.dsc` files are also supported.

Inserting the Source Hash in the SLS Data

The source_hash can be specified as a simple checksum~~, like so~~:

```
tomdroid-src-0.7.3.tar.gz:
  file.managed:
    - name: /tmp/tomdroid-src-0.7.3.tar.gz
    - source:
https://launchpad.net/tomdroid/beta/0.7.3/+download/tomdroid-src-0.7.
3.tar.gz
    - source_hash: 79eef25f9b0b2c642c62b7f737d4f53f
```

Note:

Releases prior to 2016.11.0 must also include the hash type, ~~like~~ as in the below example:

```
tomdroid-src-0.7.3.tar.gz:
  file.managed:
    - name: /tmp/tomdroid-src-0.7.3.tar.gz
    - source:
https://launchpad.net/tomdroid/beta/0.7.3/+download/tomdroid-src-0.7.
3.tar.gz
    - source_hash: md5=79eef25f9b0b2c642c62b7f737d4f53f
```

Known issues:

If the remote server URL has the hash file as an apparent sub-directory of the source file, the module will discover that it has already cached a directory where a file should be cached.

For example:

```
tomdroid-src-0.7.3.tar.gz:
  file.managed:
    - name: /tmp/tomdroid-src-0.7.3.tar.gz
    - source:
https://launchpad.net/tomdroid/beta/0.7.3/+download/tomdroid-src-0.7.
3.tar.gz
    - source_hash:
https://launchpad.net/tomdroid/beta/0.7.3/+download/tomdroid-src-0.7.
3.tar.gz/+md5
```

`source_hash_name`

When `source_hash` refers to a hash file, Salt will try to find the correct hash by matching the filename/URI associated with that hash. By default, Salt will look for the filename being managed. When managing a file at path `/tmp/foo.txt`, then the following line in a hash file would match:

```
acbd18db4cc2f85cedef654fccc4a4d8    foo.txt
```

However, sometimes a hash file will include multiple similar paths:

```
37b51d194a7513e45b56f6524f2d51f2    ./dir1/foo.txt
acbd18db4cc2f85cedef654fccc4a4d8    ./dir2/foo.txt
73feffa4b7f6bb68e44cf984c85f6e88    ./dir3/foo.txt
```

In cases like this, Salt may match the incorrect hash. This argument can be used to tell Salt which filename to match, to ensure that the correct hash is identified. (newline added)

For example:

```
/tmp/foo.txt:
  file.managed:
    - source: https://mydomain.tld/dir2/foo.txt
    - source_hash: https://mydomain.tld/hashes
    - source_hash_name: ./dir2/foo.txt
```

Note:

This argument must contain the full filename entry from the `checksum` file, as this argument is meant to disambiguate matches for multiple files that have the same basename. So, in the example above, ~~simply~~ using `foo.txt` would not match.
~~New in version 2016.3.5.~~

`keep_source`

Set to `False` to discard the cached copy of the source file once the state completes. This can be useful for larger files to keep them from taking up space in the minion cache. However, keep in mind that discarding the source file will result in the state needing to re-download the source file if the state is run again.
New in version 2017.7.3.

`user`
The user ~~to~~ who will own the file. ~~this~~ Defaults to the user Salt is running as on the minion.

`group`
The group ownership set for the file. ~~this~~ Defaults to the group Salt is running as on the minion. On Windows, this is ignored.

`mode`
The permissions to set on this file, e.g. `644, 0775, or 4664.`
The default mode for new files and directories corresponds to umask of the Salt process. The mode of existing files and directories will only be changed if `mode` is specified.

Note:
This option is not supported on Windows.

~~Changed in version 2016.11.0: This option can be set to keep, and Salt will keep the mode from the Salt fileserver. This is only supported when the source URL begins with salt://, or for files local to the minion. Because the source option cannot be used with any of the contents options, setting the mode to keep is also incompatible with the contents options.~~

Caution:
`keep` does not work with salt-ssh.
As a consequence of how the files are transferred to the minion, and the inability to connect back to the master with salt-ssh, Salt is unable to stat the file as it exists on the fileserver and thus cannot mirror the mode on the salt-ssh minion.

`attrs`
The attributes to have on this file, e.g. `a`, `i.` The attributes can be any or a combination of the following characters: `aAcCdDeijPsStTu`.

Note:
This option is not supported on Windows.
~~New in version 2018.3.0.~~

`template`
If this setting is applied, the named templating engine will be used to render the downloaded file. The following templates are supported:

- `cheetah`

- genshi
- jinja
- mako
- py
- wempy

makedirs
If set to True, then the parent directories will be created to facilitate the creation of the named file. If False, and the parent directory of the destination file doesn't exist, the state will fail.

dir_mode
If directories are to be created, passing this option specifies the permissions for those directories. If this is not set, directories will be assigned permissions by adding the execute bit to the mode of the files.

The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

replace
If set to False and the file already exists, the file will not be modified even if changes would otherwise be made. Permissions and ownership will still be enforced~~, however~~.

context
Overrides default context variables passed to the template.

defaults
Default context passed to the template.

backup
Overrides the default backup mode for this specific file. See backup_mode documentation for more details.

show_changes
Output a unified diff of the old file and the new file. If False return a boolean if any changes were made.

create
If set to False, then the file will only be managed if the file already exists on the system.

contents
Specify the contents of the file. Cannot be used in combination with source. Ignores hashes and does not use a templating engine. This value can be either a single string, a multiline YAML

string or a list of strings. If a list of strings, then the strings will be joined together with newlines in the resulting file.

For example, the below two example states would result in identical file contents:

```
/path/to/file1:
  file.managed:
    - contents:
      - This is line 1
      - This is line 2

/path/to/file2:
  file.managed:
    - contents: |
        This is line 1
        This is line 2
```

`contents_pillar`
~~New in version 0.17.0.~~

Changed in version 2016.11.0: `contents_pillar` can also be a list, and the pillars will be concatenated ~~together~~ to form one file.

Operates like contents, but draws from a value stored in pillar, using the pillar path syntax used in pillar.get. This is useful when the pillar value contains newlines, as referencing a pillar variable using a jinja/mako template can result in YAML formatting issues due to the newlines causing indentation mismatches. (newline added)

For example, the following could be used to deploy an SSH private key:

```
/home/deployer/.ssh/id_rsa:
  file.managed:
    - user: deployer
    - group: deployer
    - mode: 600
    - attrs: a
    - contents_pillar: userdata:deployer:id_rsa
```

This would populate `/home/deployer/.ssh/id_rsa` with the contents of `pillar['userdata']['deployer']['id_rsa']`. ~~An example of this pillar setup would be like so:~~

Here is an example of this pillar setup:

```
userdata:
  deployer:
    id_rsa: |
      -----BEGIN RSA PRIVATE KEY-----

MIIEowIBAAKCAQEAoQiwO3JhBquPAalQF9qP1lLZNXVjYMIswrMe2HcWUVBgh+vY

U7sCwx/dH6+VvNwmCoqmNnP+8gTPKGl1vgAObJAnMT623dMXjVKwnEagZPRJIxDy

B/HaAre9euNiY3LvIzBTWRSeMfT+rWvIKVBpvwlgGrfgz70m0pqxu+UyFbAGLin+
      GpxzZAMaFpZw4sSbIlRuissXZj/sHpQb8p9M5IeO4Z3rjkCP1cxI
      -----END RSA PRIVATE KEY-----
```

Note:
The private key above is shortened to keep the example brief, but shows how to do multiline strings in YAML. The key is followed by a pipe character, and the multiline string is indented two more spaces.

To avoid the hassle of creating an indented multiline YAML string, the file_tree external pillar can be used instead. However, this will not work for binary files in Salt releases before 2015.8.4.

`contents_grains`
~~New in version 2014.7.0.~~
Operates ~~like~~ in the same way as `contents`, but draws from a value stored in grains, using the grains path syntax used in grains.get. This functionality works similarly to `contents_pillar`, but with grains.

For example, the following could be used to deploy a "message of the day" file:

```
write_motd:
  file.managed:
    - name: /etc/motd
    - contents_grains: motd
```

This would populate `/etc/motd` file with the contents of the `motd` grain. The `motd` grain is not a default grain, and would need to be set prior to running the state:

```
salt '*' grains.set motd 'Welcome! This system is managed by Salt.'
```

`contents_newline`
~~New in version 2014.7.0. Changed in version 2015.8.4:~~ This option is now ignored if the contents being deployed contain binary data. If `True`, files managed using `contents`, `contents_pillar`, or `contents_grains` will have a newline added to the end of the file if one is not present. Setting this option to False will ensure the final line, or entry, does not contain a new line. If the last line or entry in the file does contain a new line already, this option will not remove it.

`contents_delimiter`
~~New in version 2015.8.4.~~

Can be used to specify an alternate delimiter for `contents_pillar` or `contents_grains`. This delimiter will be passed through to [`pillar.get`]() or [`grains.get`]() when retrieving the contents.

`encoding`
If specified, then the specified encoding will be used. Otherwise, the file will be encoded using the system locale (usually UTF-8). See https://docs.python.org/3/library/codecs.html#standard-encodings for the list of available encodings.

~~New in version 2017.7.0.~~

`encoding_errors`
Error encoding scheme. Default is `'strict'`. See https://docs.python.org/2/library/codecs.html#codec-base-classes for the list of available schemes.

~~New in version 2017.7.0.~~

`allow_empty`
~~New in version 2015.8.4.~~

If set to `False`, then the state will fail if the contents specified by `contents_pillar` or `contents_grains` are empty.

`follow_symlinks`
New in version 2014.7.0.
If the desired path is a symlink, follow it and make changes to the file to which the symlink points.

`check_cmd`
~~New in version 2014.7.0.~~
The specified command will be run with an appended argument of a temporary file containing the new managed contents. If the command exits with a zero status the new managed contents will be written to the managed destination. If the command exits with a nonzero exit code, the state will fail and no changes will be made to the file.

For example, the following could be used to verify sudoers before making changes:

```
/etc/sudoers:
  file.managed:
    - user: root
    - group: root
    - mode: 0440
    - attrs: i
    - source: salt://sudoers/files/sudoers.jinja
    - template: jinja
    - check_cmd: /usr/sbin/visudo -c -f
```

NOTE: This check_cmd functions differently than the requisitecheck_cmd.

tmp_dir

Directory for temp file created by check_cmd. Useful for checkers dependent on config file location (e.g. daemons restricted to their own config directories by an apparmor profile).

```
/etc/dhcp/dhcpd.conf:
  file.managed:
    - user: root
    - group: root
    - mode: 0755
    - tmp_dir: '/etc/dhcp'
    - contents: "# Managed by Salt"
    - check_cmd: dhcpd -t -cf
```

tmp_ext

Suffix for temp file created by check_cmd. Useful for checkers dependent on config file extension (e.g. the init-checkconf upstart config checker).

```
/etc/init/test.conf:
 file.managed:
    - user: root
    - group: root
    - mode: 0440
    - tmp_ext: '.conf'
    - contents:
      - 'description "Salt Minion"'
      - 'start on started mountall'
      - 'stop on shutdown'
      - 'respawn'
      - 'exec salt-minion'
    - check_cmd: init-checkconf -f
```

skip_verify

If True, hash verification of remote file sources (http://, https://, ftp://) will be skipped, and the source_hash argument will be ignored.

~~New in version 2016.3.0~~.

selinux

Allows setting the selinux user, role, type, and range of a managed file

```
/tmp/selinux.test
  file.managed:
    - user: root
    - selinux:
        seuser: system_u
        serole: object_r
        setype: system_conf_t
        seranage: s0
```

~~New in version 3000.~~

`win_owner`
The owner of the directory. If this is not passed, `user` will be used. If `user` is not passed, the account under which Salt is running will be used.
~~New in version 2017.7.0.~~

`win_perms`
A dictionary containing permissions to grant and their propagation. For example: `{'Administrators': {'perms': 'full_control'}}` Can be a single basic perm or a list of advanced perms. perms must be specified. `applies_to` does not apply to file objects.
~~New in version 2017.7.0.~~

`win_deny_perms`
A dictionary containing permissions to deny and their propagation. For example: `{'Administrators': {'perms': 'full_control'}}` Can be a single basic perm or a list of advanced perms. perms must be specified. applies_to does not apply to file objects.
~~New in version 2017.7.0.~~

`win_inheritance`
Set to `True` to inherit permissions from the parent directory, `False` ~~not to~~ to not inherit permission.
~~New in version 2017.7.0.~~

`win_perms_reset`
If `True` the existing DACL will be cleared and replaced with the settings defined in this function. If `False`, new entries will be appended to the existing DACL. Default is `False`.
~~New in version 2018.3.0.~~

Here's an example using the above `win_`* parameters:

```
create_config_file:
  file.managed:
    - name: C:\config\settings.cfg
    - source: salt://settings.cfg
    - win_owner: Administrators
  - win_perms:
    # Basic Permissions
    dev_ops:
     perms: full_control
    # List of advanced permissions
    appuser:
     perms:
      - read_attributes
      - read_ea
      - create_folders
      - read_permissions
    joe_snuffy:
     perms: read
  - win_deny_perms:
    fred_snuffy:
     perms: full_control
  - win_inheritance: False
```

`verify_ssl`
If `False`, remote https file sources (https://) and `source_hash` will not attempt to validate the servers certificate. Default is `True`.
~~New in version 3002.~~

`use_etag`
If `True`, remote http/https file sources will attempt to use the ETag header to determine if the remote file needs to be downloaded. This provides a lightweight mechanism for promptly refreshing files changed on a web server without requiring a full hash comparison via the `source_hash` parameter.
~~New in version 3005.~~

salt.states.file.**missing**(*name*, ***kwargs*)

~~Verify~~ Verifies that the named file or directory is missing. This returns `True` only if the named file is missing but does not remove the file if ~~it is~~ present.

Tip:
Useful if you don’t want to write jinja. Use `absent` if you want to remove the file or directory, and use an execution module if you just want to check.

`name`

Absolute path which must NOT exist.

`salt.states.file.`**`mknod`***(name, ntype, major=0, minor=0, user=None, group=None, mode='0600')*

For creating special devices in Linux, such as block devices and hardware links. Creates a special file similar to the 'nix `mknod` command.

Note:

`mknod` is rarely used, and is typically used if creating a fifo. Creating special devices is restricted, and usually only done as root. If the state is executed as none other than root on a minion, you may receive a permission error.

The supported device types are `p` (fifo pipe), `c` (character device), and `b` (block device). Provide the major and minor numbers when specifying a character device or block device. A fifo pipe does not require this information. The command will create the necessary dirs if needed. If a file of the same name not of the same type/major/minor exists, it will not be overwritten or unlinked (deleted).
(newline)
This is logically in place as a safety measure because you can really shoot yourself in the foot here and it is the behavior of 'nix `mknod`. It is also important to note that not just anyone can create special devices. Usually this is only done as root. If the state is executed as none other than root on a minion, you may receive a permission error.

`name`

Name of the file.

`ntype`

Node type 'p' (fifo pipe), 'c' (character device), or 'b' (block device).

`major`

Major number of the device does not apply to a fifo pipe.

`minor`

Minor number of the device does not apply to a fifo pipe.

`user`

~~owning user~~ User who owns ~~of~~ the device/pipe.

```
group
```

~~owning group~~ Group that owns ~~of~~ the device/pipe

```
mode
```

Permissions on the device/pipe.

~~Usage:~~

Examples:

```
New hdd:
     File.mknod:
       - name: /dev/fifo
       - user: root
       - group: root
       - mode: 660
       - ntype: p


/dev/chr:
  file.mknod:
    - ntype: c
    - major: 180
    - minor: 31
    - user: root
    - group: root
    - mode: 660

/dev/blk:
  file.mknod:
    - ntype: b
    - major: 8
    - minor: 999
    - user: root
    - group: root
    - mode: 660

/dev/fifo:
  file.mknod:
    - ntype: p
    - user: root
    - group: root
    - mode: 660
```

~~New in version 0.17.0.~~

`salt.states.file.`**`mod_beacon`***(name, **kwargs)*

This is an internal function, not for use by end users. Creates a beacon to monitor a file based on a beacon state argument. It is used so that files can automatically add beacons based on state files. For further information, see the beacon documentation and https://github.com/saltstack/salt-enhancement-proposals/pull/40 .

~~**Note:**~~
~~This state exists to support special handling of the beacon state argument for supported state functions. It should not be called directly.~~

`salt.states.file.`**`mod_run_check_cmd`***(cmd, filename, **check_cmd_opts)*

This is an internal function, not for use by end users. Executes the check_cmd logic.
~~`Mod_run_check_cmd` is used internally by other states.~~

Returns a result dict if `check_cmd` succeeds (`check_cmd == 0`), otherwise returns `True`.

This internal function is used when `check_cmd` is used with a file module for validating files. Also useful if commands need to be issued that can be used to check, validate, and use such as visudo for sudoers.

Example:

```
Setup sudo:
  File.managed:
    - name: /etc/sudoers
    - source: salt://sudo/files/sudoers
    - user: root
    - group: root
    - mode: 440
    - check_cmd: visudo -c -s -f
```

`salt.states.file.`**`not_cached`***(name, saltenv='base')*

~~New in version 2017.7.3.~~

Internal function, not for end users. Ensures that a file is not present in the minion's cache, deleting it if found. This state is primarily invoked by other states to ensure that a fresh copy is fetched.

`name`
The URL of the file to be removed from cache. To remove a file from cache in an environment other than `base`, either use the `saltenv` argument or include the `saltenv` in the URL (e.g.`salt://path/to/file.conf?saltenv=dev`).

Note:
A list of URLs is not supported, this must be a single URL. If a local file is passed here, the state will take no action.

`saltenv`
Used to specify the environment from which to download a file from the Salt fileserver (i.e. those with `salt://` URL).

`salt.states.file.`**`patch`**(*name, source=None, source_hash=None, source_hash_name=None, skip_verify=False, template=None, context=None, defaults=None, options='', reject_file=None, strip=None, saltenv=None, **kwargs*)

Ensures that a patch has been applied to the specified file or directory.

~~Changed in version 2019.2.0: The `hash` and `dry_run_first` options are now ignored, as the logic which determines whether or not the patch has already been applied no longer requires them. Additionally, this state now~~ Supports patch files that modify more than one file. To use these ~~sort of~~ patches, specify a directory (and, if necessary, the strip option) instead of a file.

Note:
A suitable patch executable must be available on the minion.~~ Also, keep in mind that~~ The pre-check this state does to determine whether or not changes need to be made will create a temp file and send all patch output to that file. ~~This means that,~~ Therefore in the event that the patch would not have applied cleanly, the comment included in the state results will reference a temp file that will no longer exist once the state finishes running.

`name`
The file or directory to which the patch should be applied.

`source`
The patch file to apply.
~~Changed in version 2019.2.0:~~ The source can now be from any file source supported by Salt (salt://, http://, https://, ftp://, etc.). Templating is also now supported.

`source_hash`
Works the same ~~way~~ as in [`file.managed`](file.managed).
~~New in version 2019.2.0.~~

`source_hash_name`
Works the same ~~way~~ as in [`file.managed`](file.managed).
~~New in version 2019.2.0.~~

`skip_verify`
Works the same ~~way~~ as in [`file.managed`](file.managed).
~~New in version 2019.2.0.~~

`template`
Works the same ~~way~~ as in [`file.managed`](file.managed).
~~New in version 2019.2.0.~~

`context`
Works the same ~~way~~ as in [`file.managed`](file.managed).
~~New in version 2019.2.0.~~

`defaults`
Works the same ~~way~~ as in [`file.managed`](file.managed).
New in version 2019.2.0.

`options`
Extra options to pass to patch. This should not be necessary in most cases.

Note:
For best results, short opts should be separate from one another. The `-N` and `-r`, and `-o` options are used internally by this state and cannot be used here. Additionally, instead of using `-pN` or `--strip=N`, use the strip option documented below.

`reject_file`
If specified, any rejected hunks will be written to this file. If not specified, then they will be written to a temp file which will be deleted when the state finishes running.

`important`
The parent directory must exist. Also, this will overwrite the file if it is already present.
~~New in version 2019.2.0.~~

`strip`
Number of directories to strip from paths in the patch file. For example, using the below SLS would instruct Salt to use -p1 when applying the patch:

```
/etc/myfile.conf:
  file.patch:
    - source: salt://myfile.patch
    - strip: 1
```

~~New in version 2019.2.0: In previous versions, -p1 would need to be passed as part of the options value.~~

`saltenv`
Specify the environment from which to retrieve the patch file indicated by the source parameter. If not provided, this defaults to the environment from which the state is being executed.

Note:
Ignored when the patch file is from a non-salt:// source.

~~Usage~~ Example:

```
# Equivalent to ``patch --forward /opt/myfile.txt myfile.patch``
/opt/myfile.txt:
  file.patch:
    - source: salt://myfile.patch
```

`salt.states.file.`**`prepend`***(name, text=None, makedirs=False, source=None, source_hash=None, template='jinja', sources=None, source_hashes=None, defaults=None, context=None, header=None)*
Ensures that ~~some~~ specified text appears at the beginning of a file.

The text will not be prepended again if it already exists in the file. You may specify a single line of text or a list of lines to append.

`name`
The location of the file to ~~append to~~ which the new text will be appended.

`text`
The text to be appended. ~~which~~ Text can be a single string or a list of strings.

`makedirs`
If the file is located in a path without a parent directory, then the state will fail. If `makedirs` is set to `True`, then the parent directories will be created to facilitate the creation of the named file. Defaults to `False`.

`source`
A single source file to append. This source file can be hosted on either the Salt master server, or on an HTTP or FTP server. Both HTTPS and HTTP are supported as well as downloading directly from Amazon S3 compatible URLs with both pre-configured and automatic IAM credentials (see s3.get state documentation). File retrieval from Openstack Swift object storage is supported via swift://container/object_path URLs (see swift.get documentation).

For files hosted on the salt file server, if the file is located on the master in the directory named spam, and is called eggs, the source string is salt://spam/eggs.

If the file is hosted on an HTTP or FTP server, the `source_hash` argument is also required.

`source_hash`
This can be one of the following:

1. A source hash string
2. The URI of a file that contains source hash strings

The function accepts the first encountered long unbroken alphanumeric string of correct length as a valid hash, in order from most secure to least secure:

Type	Length
sha512	128
sha384	96
sha256	64
sha224	56
sha1	40
md5	32

See the `source_hash` parameter description for `file.managed` function for more details and examples.

`template`
The named templating engine will be used to render the appended-to file. Defaults to jinja.

The following templates are supported:

- `cheetah`
- `genshi`
- `jinja`
- `mako`
- `py`
- `wempy`

`sources`
A list of source files to append. If the files are hosted on an HTTP or FTP server, the `source_hashes` argument is also required.

`source_hashes`
A list of `source_hashes` corresponding to the `sources` list specified in the `sources` argument.

`defaults`
Default context passed to the template.

`context`
Overrides default context variables passed to the template.

`ignore_whitespace`
~~New in version 2015.8.4.~~
Spaces and Tabs in text are ignored by default, when searching for the appending content, one space or multiple tabs are the same for Salt. Set this option to `False` if you want to change this behavior.

Multi-line example:

```
/etc/motd:
  file.prepend:
    - text: |
      Thou hadst better eat salt with the Philosophers of Greece,
        than sugar with the Courtiers of Italy.
        - Benjamin Franklin
```

Multiple lines of text:

```
/etc/motd:
  file.prepend:
    - text:
      - Trust no one unless you have eaten much salt with him.
      - "Salt is born of the purest of parents: the sun and the sea."
```

Optionally, require the text to appear exactly as specified (order and position). Combine with multi-line or multiple lines of input.

```
/etc/motd:
  file.prepend:
    - header: True
    - text:
      - This will be the very first line in the file.
      - The 2nd line, regardless of duplicates elsewhere in the file.
```

```
        - These will be written anew if they do not appear verbatim.
```

Gather text from multiple template files:

```
/etc/motd:
  file:
    - prepend
    - template: jinja
    - sources:
      - salt://motd/devops-messages.tmpl
      - salt://motd/hr-messages.tmpl
      - salt://motd/general-messages.tmpl
```

~~New in version 2014.7.0.~~

`salt.states.file.`**recurse**(*name*, *source*, *keep_source=True*, *clean=False*, *require=None*, *user=None*, *group=None*, *dir_mode=None*, *file_mode=None*, *sym_mode=None*, *template=None*, *context=None*, *replace=True*, *defaults=None*, *include_empty=False*, *backup=''*, *include_pat=None*, *exclude_pat=None*, *maxdepth=None*, *keep_symlinks=False*, *force_symlinks=False*, *win_owner=None*, *win_perms=None*, *win_deny_perms=None*, *win_inheritance=True*, ***kwargs*)

~~Recurse through a subdirectory on the master and copy said subdirectory over to the specified path.~~ Copies a directory from the salt master fileserver to the minion.

`name`
The directory to set the recursion in.

`source`
The source directory. This directory is located on the Salt master file server and is specified with the salt:// protocol. If the directory is located on the master in the directory named spam, and is called eggs, the source string is salt://spam/eggs

`keep_source`
Set to `False` to discard the cached copy of the source file once the state completes. This can be useful for larger files to keep them from taking up space in the minion cache. ~~However, keep in mind that~~ Discarding the source file will result in the state needing to re-download the source file if the state is run again.
~~New in version 2017.7.3.~~

`clean`

Make sure that only files that are set up by Salt and required by this function are kept. If this option is set then everything in this directory will be deleted unless it is required.

`require`
Require other resources such as packages or files.

`user`
The user ~~to~~ who will own the directory. This defaults to the user Salt is running as on the minion.

`group`
The group ownership set for the directory. This defaults to the group Salt is running as on the minion. On Windows, this is ignored.

`dir_mode`
The permissions mode to set on any directories created.
The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

Note:
This option is not supported on Windows.

`file_mode`
The permissions mode to set ~~on~~ for any files created.
The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

Note:
This option is not supported on Windows.
~~Changed in version 2016.11.0:~~ This option can be set to keep, and Salt will keep the mode from the Salt fileserver. This is only supported when the source URL begins with salt://, or for files local to the minion. Because the source option cannot be used with any of the contents options, setting the mode to keep is also incompatible with the contents options.

`sym_mode`
The permissions mode to set ~~on~~ for any symlink created.
The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

Note:
This option is not supported on Windows.

`template`
If this setting is applied, the named templating engine will be used to render the downloaded file. The following templates are supported:

- `cheetah`
- `genshi`
- `jinja`
- `mako`
- `py`
- `wempy`

Note:
The template option is required when recursively applying templates.

`replace`
If set to `False` and the file already exists, the file will not be modified even if changes would otherwise be made. Permissions and ownership will still be enforced~~, however~~.

`context`
Overrides default context variables passed to the template.

`defaults`
Default context passed to the template.

`include_empty`
Set this to True if empty directories should also be created (default is `False`)

`backup`
Overrides the default backup mode for all replaced files. See backup_mode documentation for more details.

`include_pat`
When copying, include only this pattern, or list of patterns, from the source. Default is glob match; if prefixed with 'E@', then regexp match.
Example:

```
- include_pat: hello*       :: glob matches 'hello01', 'hello02'
                               ... but not 'otherhello'
- include_pat: E@hello      :: regexp matches 'otherhello',
                               'hello01' ...
```

Changed in version 3001: List patterns are now supported

```
- include_pat:
    - hello01
    - hello02
```

`exclude_pat`

Exclude this pattern, or list of patterns, from the source when copying. If both `include_pat` and `exclude_pat` are supplied, then it will apply conditions cumulatively. i.e. first select based on `include_pat`, and then within that result apply `exclude_pat`.
Also, when `'clean=True'`, exclude this pattern from the removal list and preserve in the destination.

For example:

```
- exclude_pat: APPDATA*                  :: glob matches APPDATA.01,
                                            APPDATA.02,.. for exclusion
- exclude_pat: E@(APPDATA)|(TEMPDATA)  :: regexp matches APPDATA
                                            or TEMPDATA for exclusion
```

~~Changed in version 3001:~~ Example: List patterns are ~~now~~ supported

```
- exclude_pat:
    - APPDATA.01
    - APPDATA.02
```

`maxdepth`

When copying, only copy paths which are of depth maxdepth from the source path.

For example:

```
- maxdepth: 0      :: Only include files located in the source
                      directory
- maxdepth: 1      :: Only include files located in the source
                      or immediate subdirectories
```

`keep_symlinks`

Keep symlinks when copying from the source. This option will cause the copy operation to terminate at the symlink. If desired behavior is similar to rsync, then set this to `True`.

`force_symlinks`

Force symlink creation. ~~This option will force the symlink creation.~~ If a file or directory is obstructing symlink creation it will be recursively removed so that symlink creation can proceed. This option is usually not needed except in special circumstances.

`win_owner`

The owner of the symlink and directories if `makedirs` is `True`. If this is not passed, `user` will be used. If `user` is not passed, the account under which Salt is running will be used.
~~New in version 2017.7.7.~~

`win_perms`

A dictionary containing permissions to grant.
~~New in version 2017.7.7.~~

`win_deny_perms`
A dictionary containing permissions to deny.
~~New in version 2017.7.7.~~

`win_inheritance`
Use `True` to inherit permissions from parent, otherwise use `False`.
~~New in version 2017.7.7.~~

salt.states.file.rename(*name, source, force=False, makedirs=False, **kwargs*)

If the source file exists on the system, rename it ~~to~~ as the named file. The named file will not be overwritten if it already exists unless the `force` option is set to `True`.

`name`
The location of the file to ~~rename to~~ be renamed.

`source`
The location of the file to move to the location specified ~~with~~ by `name`.

`force`
If the target location is present then the file will not be moved~~,~~. Specify "`force`: `True`" to overwrite the target file.

`makedirs`
~~If the target subdirectories don't exist create them~~
Create subdirectories if they don’t exist.

salt.states.file.replace(*name, pattern, repl, count=0, flags=8, bufsize=1, append_if_not_found=False, prepend_if_not_found=False, not_found_content=None, backup='.bak', show_changes=True, ignore_if_missing=False, backslash_literal=False*)

Maintain an edit in a file.

~~New in version 0.17.0.~~

`name`
Filesystem path to the file to be edited. If a symlink is specified, it will be resolved to its target.

`pattern`
A regular expression, to be matched using Python's `re.search()`.

Note:
If you need to match a literal string that contains regex special characters, you may want to use Salt's custom Jinja filter, `regex_escape`.

```
{{ 'http://example.com?foo=bar%20baz' | regex_escape }}
```

`repl`
The replacement text.

`count`
Maximum number of pattern occurrences to be replaced. Defaults to 0. If `count` is a positive integer n, no more than n occurrences will be replaced, otherwise all occurrences will be replaced.

`flags`
A list of flags defined in the re module documentation from the Python standard library. Each list item should be a string that will correlate to the human-friendly flag name. ~~E.g.~~ For example, `['IGNORECASE', 'MULTILINE']`. Optionally, `flags` may be an int, with a value corresponding to the XOR (`|`) of all the desired flags. Defaults to 8 (which equates to `['MULTILINE']`).

Note:
`file.replace` reads the entire file as a string to support multiline regex patterns. Therefore, when using anchors such as `^` or `$` in the pattern, those anchors may be relative to the line OR relative to the file. The default for `file.replace` is to treat anchors as relative to the line, which is implemented by setting the default value of `flags` to `['MULTILINE']`. When overriding the default value for `flags`, if `'MULTILINE'` is not present then anchors will be relative to the file. If the desired behavior is for anchors to be relative to the line, then simply add `'MULTILINE'` to the list of flags.

`bufsize`

Specifies how much of the file to buffer into memory at once. The default value `1` processes one line at a time. The special value `file` may be specified which will read the entire file into memory before processing.

`append_if_not_found`
If set to `True`, and pattern is not found, then the content will be appended to the file.
~~New in version 2014.7.0.~~

`prepend_if_not_found`
If set to `True` and pattern is not found, then the content will be prepended to the file.
~~New in version 2014.7.0.~~

`not_found_content`
Content to use for append/prepend if not found. If `None` (default), uses `repl`. Useful when `repl` uses references to group in pattern.
~~New in version 2014.7.0.~~

`backup`
The file extension to use for a backup of the file before editing. Set to `False` to skip making a backup.

`show_changes`
Output a unified diff of the old file and the new file. If `False`, return a boolean if any changes were made. Returns a boolean or a string.

`ignore_if_missing`
~~New in version 2016.3.4.~~
Controls what to do if the file is missing. If set to `False`, the state will display an error raised by the execution module. If set to `True`, the state will ~~simply~~ report no changes.

`backslash_literal`
~~New in version 2016.11.7.~~ Interpret backslashes as literal backslashes for the repl ~~and not escape~~ without escaping characters. This will help when using append/prepend so that the backslashes are not interpreted for the `repl` on the second run of the state.

For complex regex patterns, it can be useful to avoid the need for complex quoting and escape sequences by making use of YAML's multiline string syntax:

```
complex_search_and_replace:
  file.replace:
    # <...snip...>
    - pattern: |
        CentOS \(2.6.32[^\\n]+\\n\s+root[^\\n]+\\n\)+
```

Note:
When using YAML multiline string syntax in `pattern:`,, make sure to also use that syntax in the `repl:` part, or you might ~~loose~~ lose line feeds.

When regex capture groups are used in pattern:, their captured value is available for reuse in the repl: part as a backreference (ex. `\1`).

```
add_login_group_to_winbind_ssh_access_list:
  file.replace:
    - name: '/etc/security/pam_winbind.conf'
    - pattern: '^(require_membership_of = )(.*)$'
    - repl: '\1\2,append-new-group-to-line'
```

Note:
The `file.replace` state uses Python's `re` module. For more advanced options, see https://docs.python.org/2/library/re.html

salt.states.file.retention_schedule(*name, retain, strptime_format=None, timezone=None*)

Applies retention scheduling to backup storage directory.
~~New in version 2016.11.0.~~

`name`
The filesystem path to the directory containing backups to be managed.

`retain`
Deletes ~~the~~ backups ~~except for~~ other than the ones ~~we~~ you want to keep. The N below should be an integer but may also be the special value of `all`, which keeps all files matching the criteria. All of the retain options default to `None`, which means to not keep files based on this criteria.

- `most_recent N`
  Keep the most recent N files.
- `first_of_hour N`
  For the last N hours from now, keep the first file after the hour.
- `first_of_day N`
  For the last N days from now, keep the first file after midnight. See also `timezone`.

- `first_of_week N`
  For the last N weeks from now, keep the first file after Sunday midnight.
- `first_of_month N`
  For the last N months from now, keep the first file after the start of the month.
- `first_of_year N`
  For the last N years from now, keep the first file after the start of the year.

`strptime_format`
A python strptime format string used to first match the filenames of backups and then parse the filename to determine the datetime of the file. See also: https://docs.python.org/2/library/datetime.html#datetime.datetime.strptime Defaults to `None`, which considers all files in the directory to be backups eligible for deletion and uses `os.path.getmtime()` to determine the datetime.

`timezone`
The timezone to use when determining midnight. This is only used when datetime is pulled from `os.path.getmtime()`. Defaults to `None` which uses the timezone from the locale.

~~Usage example:~~ Example:

```
/var/backups/example_directory:
  file.retention_schedule:
    - retain:
        most_recent: 5
        first_of_hour: 4
        first_of_day: 7
        first_of_week: 6    # NotImplemented yet.
        first_of_month: 6
        first_of_year: all
    - strptime_format: example_name_%Y%m%dT%H%M%S.tar.bz2
    - timezone: None
```

`salt.states.file.serialize`*(name, dataset=None, dataset_pillar=None, user=None, group=None, mode=None, backup='', makedirs=False, show_changes=True, create=True, merge_if_exists=False, encoding=None, encoding_errors='strict', serializer=None, serializer_opts=None, deserializer_opts=None, **kwargs)*

Serializes a dataset and stores it in ~~into~~ a managed file. Useful for sharing simple configuration files.

`name`

The location of the file to create.

`dataset`
The dataset that will be serialized.

`dataset_pillar`
Operates like `dataset`, but draws from a value stored in pillar, using the pillar path syntax ~~used~~ in pillar.get. This is useful when the pillar value contains newlines, as referencing a pillar variable using a jinja/mako template can result in YAML formatting issues due to the newlines causing indentation mismatches.
~~New in version 2015.8.0.~~

`serializer` (or formatter)
Write the data ~~as~~ in this format. See the list of serializer modules for supported output formats.
~~Changed in version 3002: `serializer` argument added as an alternative to `formatter`. Both are accepted, but using both will result in an error.~~ Similar to `formatter`. Using both `formatter` and `serializer` will result in an error.

`encoding`
~~If specified, then the specified encoding will be used.~~ The encoding to be used if specified. Otherwise, the file will be encoded using the system locale (usually UTF-8). See https://docs.python.org/3/library/codecs.html#standard-encodings for the list of available encodings.
~~New in version 2017.7.0.~~

`encoding_errors`
Error encoding scheme. Default is `` `'strict'` ``. See https://docs.python.org/2/library/codecs.html#codec-base-classes for the list of available schemes.
~~New in version 2017.7.0.~~

`user`
The user to own the directory~~,~~ . ~~this d~~ Defaults to the user Salt is running as on the minion.

`group`
The group ownership set for the directory~~.~~. ~~this d~~ Defaults to the group Salt is running as on the minion.

`mode`
The permissions to set on this file, e.g. `644`, `0775`, or `4664`.
The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

Note:

This option is not supported on Windows.

`backup`
Overrides the default backup mode for ~~this specific~~ the specified file.

`makedirs`
Create parent directories for the destination file.
~~New in version 2014.1.3.~~

`show_changes`
Output a unified diff of the old file and the new file. If `False` return a boolean if any changes were made.

`create`
Default is `True`, if create is set to `False` then the file will only be managed if the file already exists on the system.

`merge_if_exists`
Default is `False`~~,~~. If `merge_if_exists` is `True` then the existing file will be parsed and the dataset passed in will be merged with the existing content.
~~New in version 2014.7.0.~~

`serializer_opts`
Pass through options to serializer.

For example:

```
/etc/dummy/package.yaml
  file.serialize:
    - serializer: yaml
    - serializer_opts:
      - explicit_start: True
      - default_flow_style: True
      - indent: 4
```

The valid opts are the additional opts (i.e. not the data being serialized) for the function used to serialize the data. Documentation for the these functions can be found in the list below:

- For yaml: `yaml.dump()`
- For json: `json.dumps()`
- For python: `pprint.pformat()`
- For msgpack: Run `python -c 'import msgpack; help(msgpack.Packer)'` to see the available options (`encoding`, `unicode_errors`, etc.)

`deserializer_opts`

Like `serializer_opts` above, but only used when merging with an existing file (i.e. when merge_if_exists is set to `True`).

The options specified here will be passed to the deserializer to load the existing data, before merging with the specified data and re-serializing.

```
/etc/dummy/package.yaml
  file.serialize:
    - serializer: yaml
    - serializer_opts:
      - explicit_start: True
      - default_flow_style: True
      - indent: 4
    - deserializer_opts:
      - encoding: latin-1
    - merge_if_exists: True
```

The valid opts are the additional opts (i.e. not the data being deserialized) for the function used to deserialize the data. Documentation for the these functions can be found in the list below:

- For yaml: [`yaml.load()`]
- For json: [`json.loads()`]

However, note that not all arguments are supported. For example, when deserializing JSON, arguments like `parse_float` and `parse_int` which accept a callable object cannot be handled in an SLS file.
~~New in version 2019.2.0.~~

For example, this state:

```
/etc/dummy/package.json:
  file.serialize:
    - dataset:
        name: naive
        description: A package using naive versioning
        author: A confused individual <iam@confused.com>
        dependencies:
          express: '>= 1.2.0'
          optimist: '>= 0.1.0'
        engine: node 0.4.1
    - serializer: json
```

will manage the file /etc/dummy/package.json:

```
{
  "author": "A confused individual <iam@confused.com>",
  "dependencies": {
    "express": ">= 1.2.0",
    "optimist": ">= 0.1.0"
  },
  "description": "A package using naive versioning",
  "engine": "node 0.4.1",
  "name": "naive"
}
```

`salt.states.file.`**`shortcut`**(*name, target, arguments=None, working_dir=None, description=None, icon_location=None, force=False, backupname=None, makedirs=False, user=None, **kwargs*)

Creates a Windows shortcut.

If the file already exists and is a shortcut pointing to any location other than the specified target, the shortcut will be replaced. If it is a regular file or directory then the state will return `False`. If the regular file or directory is desired to be replaced with a shortcut pass force: True, if it is to be renamed, pass a backupname.

`name`
The location of the shortcut to create. Must end with either ".lnk" or ".url."

`target`
The location that the shortcut points to.

`arguments`
Any arguments to pass in the shortcut.

`working_dir`
Working directory in which to execute target.

`description`
Description to set on shortcut.

`icon_location`
Location of shortcut's icon.

`force`
If the name of the shortcut exists and is not a file and force is set to `False`, the state will fail. If force is set to `True`, the link or directory in the way of the shortcut file will be deleted to make room for the shortcut, unless backupname is set, when it will be renamed.

`backupname`
If the name of the shortcut exists and is not a file, it will be renamed to the backupname. If the backupname already exists and force is `False`, the state will fail. Otherwise, the backupname will be removed first.

`makedirs`
If the location of the shortcut does not already have a parent directory then the state will fail, setting makedirs to `True` will allow Salt to create the parent directory. Setting this to `True` will also create the parent for backupname if necessary.

`user`
The user to own the file. This defaults to the user Salt is running as on the minion
The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

`salt.states.file.``symlink`*(name, target, force=False, backupname=None, makedirs=False, user=None, group=None, mode=None, win_owner=None, win_perms=None, win_deny_perms=None, win_inheritance=None, **kwargs)*

Creates a symbolic link (symlink, soft link).

If the file already exists and is a symlink pointing to any location other than the specified target, the symlink will be replaced. If an entry with the same name exists then the state will return False. If the existing entry is desired to be replaced with a symlink pass force: True, if it is to be renamed, pass a backupname.

`name`
The location of the symlink to create.

`target`
The location that the symlink points to.

`force`
If the name of the symlink exists and is not a symlink and force is set to `False`, the state will fail. If force is set to `True`, the existing entry in the way of the symlink file will be deleted to make room for the symlink, unless backupname is set, when it will be renamed.
~~Changed in version 3000:~~ `force` will ~~now~~ remove all types of existing file system entries, not just files, directories and symlinks.

`backupname`

If the name of the symlink exists and is not a symlink, it will be renamed to the `backupname`. If the `backupname` already exists and force is `False`, the state will fail. Otherwise, the `backupname` will be removed first. An absolute path OR a basename file/directory name must be provided. The latter will be placed relative to the symlink destination's parent directory.

`makedirs`
If the location of the symlink does not already have a parent directory then the state will fail, setting makedirs to `True` will allow Salt to create the parent directory.

`user`
The user to own the file, this defaults to the user Salt is running as on the minion.

`group`
The group ownership set for the file, this defaults to the group Salt is running as on the minion. On Windows, this is ignored.

`mode`
The permissions to set on this file, aka `644`, `0775`, `4664`. Not supported on Windows.
The default mode for new files and directories corresponds to umask of Salt process. For existing files and directories it's not enforced.

`win_owner`
The owner of the symlink and directories if makedirs is `True`. If this is not passed, user will be used. If user is not passed, the account under which Salt is running will be used.
~~New in version 2017.7.7.~~

`win_perms`
A dictionary containing permissions to grant.
~~New in version 2017.7.7.~~

`win_deny_perms`
A dictionary containing permissions to deny.
~~New in version 2017.7.7.~~

`win_inheritance`
Set to `True` to inherit permissions from parent, otherwise `False`.
~~New in version 2017.7.7.~~

salt.states.file.tidied(*name, age=0, matches=None, rmdirs=False, size=0, exclude=None, full_path_match=False, followlinks=False, time_comparison='atime', **kwargs*)

~~Changed in version 3005.~~

Removes unwanted files based on specific criteria. Multiple criteria are OR'd together, so a file that is too large but is not old enough will still get tidied. If neither age nor size is given all files which match a pattern in matches will be removed.

Note:
The regex patterns in this function are used in re.match(), so there is an implicit "beginning of string" anchor (^) in the regex and it is unanchored at the other end unless explicitly entered ($).

`name`
The directory tree that should be tidied.

`age`
Maximum age in days after which files are considered for removal.

`matches`
List of regular expressions to restrict what gets removed. Default: ['.*']

`rmdirs`
~~Whether or not it is~~ `True` if allowed to remove directories; `False` if not allowed.

`size`
Maximum allowed file size. Files greater or equal to this size are removed. Doesn't apply to directories or symbolic links

`exclude`
List of regular expressions to filter the matches parameter and better control what gets removed.
~~New in version 3005.~~

`full_path_match`
Match the matches and exclude regex patterns against the entire file path instead of just the file or directory name. Default: `False`
~~New in version 3005.~~

`followlinks`
This module will not descend into subdirectories which are pointed to by symbolic links. If you wish to force it to do so, you may give this option the value True. Default: False

~~New in version 3005.~~

time_comparison
Default: atime. Options: atime/mtime/ctime. This value is used to set the type of time comparison made using age. The default is to compare access times (atime) or the last time the file was read. A comparison by modification time (mtime) uses the last time the contents of the file was changed. The ctime parameter is the last time the contents, owner, or permissions of the file were changed.
~~New in version 3005.~~

```
cleanup:
  file.tidied:
    - name: /tmp/salt_test
    - rmdirs: True
    - matches:
      - foo
      - b.*r
```

salt.states.file.touch(*name, atime=None, mtime=None, makedirs=False*)

Replicates the 'nix "touch" command to create a new empty file or update the atime and mtime of an existing file.

Note that if you just want to create a file and don't care about atime or mtime, you should use file.managed instead, as it is more feature-complete. (Just leave out the source/template/contents arguments, and it will just create the file and/or check its permissions, without messing with contents)

name
Name of the file.

atime
atime of the file.

mtime
mtime of the file.

makedirs
Specifies whether ~~we should~~ to create the parent directory/directories in order to touch the file.

~~Usage:~~ Example:

```
/var/log/httpd/logrotate.empty:
  file.touch
```

~~New in version 0.9.5.~~

salt.states.file.uncomment(*name, regex, char='#', backup='.bak'*)

Uncomments specified commented lines in a file

`name`
The full path to the file to be edited.

`regex`
A regular expression used to find the lines that are to be uncommented. This regex should not include the comment character. A leading ^character will be stripped for convenience (for easily switching between comment() and uncomment()). The regex will be searched for from the beginning of the line, ignoring leading spaces (we prepend '^[t]*')

`char`
The character to remove in order to uncomment a line.

`backup`
The file will be backed up before edit with this file extension.

Warning:
This backup will be overwritten each time sed / comment / uncommentis called. Meaning the backup will only be useful after the first invocation.
Set to `False`/`None` to not keep a backup.

~~Usage:~~ Example:

```
/etc/adduser.conf:
  file.uncomment:
    - regex: EXTRA_GROUPS
```

~~New in version 0.9.5.~~