pave Documentation

Release 0.68 beta

Mike Miller

March 09, 2014

Contents

1 Contents **3**

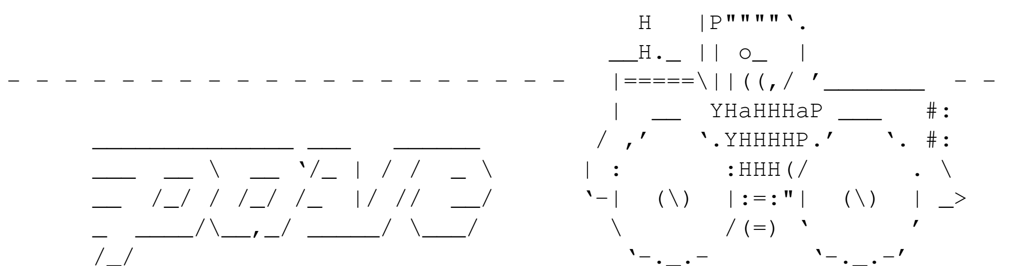
1.1 Intro . . . 3
1.2 Pavefile Reference . . . 41
1.3 Pavefile Reference Part II . . . 46
1.4 Command-line Reference . . . 60
1.5 Developer Guide . . . 62
1.6 FAQ - (Not So) Frequently Asked Questions . . . 75

Python Module Index **79**

Simple push-based deployment and remote host management, leveraging fabric. No servers, few dependencies.

CHAPTER 1

Contents



tl;dr

Simple push-based deployment and remote host management, leveraging fabric. No servers, few dependencies.

1.1 Intro

There are also slides on this topic, located here.

1.1.1 Another one?

Been looking for the "perfect" config management tool to deploy software to networked boxes... and still looking. I've tried but didn't choose these:

bcfg, cfengine, puppet, chef, saltstack, ansible, etc.

I found them powerful and complex. They required daemons running, had their own languages, unique terminology, XML, Ruby, or Jinja dependencies. Despite reading the word "simple" multiple times, each had a varingly steep learning curve, and dozens if not hundreds of pages of abstract or "enterprisey" docs.

For my modest needs, ansible seemed the best fit (ssh-based and no servers), but there were a few things bugging me. Finally realized I was telling it how to do things, though I really wanted the tool to *figure it out itself*.

Likewise, I stumbled across fabric early on, and like most loved it—at first. Unfortunately as a script gets longer it dawns on you why people say, *"don't use it for configuration management!"* Intentions, implementation details, and code all get mixed up in the same script and things get ugly fast. Cuisine (a fabric addon) appeared promising, but it looked mainly to shorten the fabfile. I wanted to go further and completely separate my setup from general implementation code.

Maybe I could whip one up myself? I started looking into how to build such a system. Thankfully after a day or so I realized the building blocks are already in fabric. Why not just commandeer it for my own evil purposes? *</grin>*

So here is pave.

1.1.2 How does it work?

Pave logs you into your networked hosts using SSH and configures them to your liking "automatically" using the command line. Just tell it *what* to do. It'll check first and *won't make changes* unless it needs to.

Don't have to write any code for common tasks. Don't have to know anything about Python or the OS you're using (in theory—You'll get the most out of it if you know how to use the default shell installed on your hosts, such as bash.) No gnarly/obscure runtimes to install, no root servers taking up memory/listening/waiting to be cracked, no new languages, nor weird lingo to learn.

Just a config file, a command, ... and space for a new parking lot.

1.1.3 Use

Requirements

An OS from the last 5 years or so, with Python 2.6+ (locally) and sshd (remotely) ready to go.

Development is primarily on Ubuntu clients with **Ubuntu Server 12.04 (Precise LTS)** as the target (with some recent attention to Fedora, CentOS, Gentoo, Rasbian, and OSX), but aims to work everywhere practical, Unix and Windows.

Installation & Setup

Need pip?

```
sudo apt-get install python-pip   # Or yum, brew, etc.
```

"Express" install:

```
sudo pip install pave
```

Custom install options:

> **OSX**: (Not yet tested, alpha quality) Install pip and the command-line developer tools if they are not already (type `easy_install pip` and `gcc` at a terminal), since pave needs paramiko/pycrypto and the latter needs to be built.
>
> **Windows**: (Not yet tested, alpha quality) Install pip and pycrypto. Type `pip install pave[win]` in an Admin prompt.
>
> **Debian Linux**: Note if pycrypto (ssh) needs to be built, or you'd like to build libyaml (for speed), do this first:
>
> ```
> sudo apt-get install build-essential python-dev libyaml-dev
> ```
>
> For the dev version:
>
> ```
> sudo pip install \
> https://bitbucket.org/mixmastamyk/pave/get/default.zip
> ```
>
> Also, note that pip is not required; You could also download the package manually from PyPI or bitbucket. Unpack the archive, `cd` into the folder created, then:
>
> ```
> sudo ./setup.py install    # manual install
> ```

> Like typing in passwords over and over? No? ;)
> Do you have an ssh key-pair set up? No?
>
> ```
> ssh-keygen -t rsa
> ssh-copy-id [user@]machine
> ```
>
> Pave can do the steps above if you'd like, however for "piece" of mind it's better to do it yourself.
>
> You may need to paste the matching line from `~/.ssh/id_rsa.pub` into your EC2 or other cloud console... if you're into that sort of thing.

Creating El Pavefile

The pavefile is what tells pave what to do. This file is named `./pave.yml` by default, and "hello world" might look like this:

```
main:
    user: ubuntu
    sudo: True
    targets:
        - 192.168.2.1            # a list/str of hosts/groups

# passwords:                     # use only if not using keys
#   - ask password SSH Password  # default passwd for fabric

tasks:
    - packages:
        if-platform: Debian
        install: sysvbanner

    - packages:
        if-platform: Redhat
        install: banner

    - banner "Hello!"            # world
```

To create this file now, you may use the "skeleton" shortcut:

```
pave -S
```

Each section above declares what you'd like pave to do, and the details under each must be indented *with spaces*. Comments are written following the # character. The only required section is `main` which contains options for pave. Lists and mappings (dicts/hashes) are welcome in appropriate places. The format is called yaml, and syntax details can be found here. You don't need to know much about it for now, just keep to the form above.

Common tasks already have their own modules, such as `packages:` above. These modules handle mundane details so you don't need to be concerned. Don't reinvent the wheel by putting platform-dependent commands into an ad-hoc tasklist—That's what we're trying to avoid.

Reference docs can be found on the next page and examples here. How to target groups of hosts and do other necessary things like declare variables, render templates, create users, configure the OS, etc, etc. is demonstrated.

Running it

```
pave
```

Now, that's my definition of simple. ;) Of course pave does less than the big-guns but handles quite a lot already and will only grow. Consult the references and `pave -h` for many options.

You should see output similar to this (if you use `pave -V`). (Verbose logs from the last five runs are kept in `~/.cache/pave/logs/`.)

```
 INFO    main: Version 0.60, Python: 2.7.4, fabric: 1.6.1, ./pave.yml
 INFO    transport: Connected (version 2.0, client OpenSSH_5.9p1)
 INFO    transport: Authentication (publickey) successful!
 INFO    main: found a(n) Ubuntu system.
 INFO    packages: install: sysvbanner

[snip install-text]

[192.168.2.1] out: sudo password:
[192.168.2.1] out: #     #                                    ###
[192.168.2.1] out: #     #  ######  #       #         ####    ###
[192.168.2.1] out: #     #  #       #       #        #    #   ###
[192.168.2.1] out: #######  #####   #       #        #    #    #
[192.168.2.1] out: #     #  #       #       #        #    #
[192.168.2.1] out: #     #  #       #       #        #    #   ###
[192.168.2.1] out: #     #  ######  ######  ######    ####    ###

 INFO    main: {u'192.168.2.1': (2, 0)}
```

The numbers at the end mean pave made two changes and had zero errors on the remote host.

"Exxxxcellent, Smithers."

License

Licensed under the GPL, version 3+. A commercial license may also be available in the future.

Next »»

(Looking for a very simple build tool to complement pave? BuildIt was designed for that purpose.)

This concludes the introductory readme. Kindly continue on to the next chapter of the full docs. You may also go to the table of contents instead.

Pavefile Reference Syntax and layout of pave's configuration file is described here. For information on the various sections and task module library, skip ahead to the next chapter, *Pavefile Reference Part II*.

Syntax At the most basic level, a pavefile is expected to be encoded in UTF-8 format and named `./pave.yml` by default.

Pave utilizes the user-friendly, human-optimized YAML file format with a few extensions. If you haven't encountered it yet, you may wish you had found it earlier. There's not much to learn to get started, less than other formats.

No verbose "angle-bracket" tax, less punctuation and synchronization necessary than XML or JSON, and no harsh limitations like .ini files.

YAML Resources When in doubt I wander over to the sites below:

- http://en.wikipedia.org/wiki/Yaml#Language_elements
- http://pyyaml.org/wiki/PyYAMLDocumentation#YAMLsyntax

General Layout Comments start with the "#" character, and indentation must be done *with spaces*. The form `name: value` represents a dictionary mapping, often called a hash, or associative array. A value is retrieved by its name. The `- item` form represents a list, an expandable array indexed by number.

A basic file will look like this, with *section* names at the far left:

```
vars:
    name: value

main:                               # starts a map of a map, to
    name1:  value1                  # strings, or a
    name2:  True                    # boolean, or ...

tasks:                              # a map to a list of
    - "list item 2"                 # quoted-strings, or
    - 42                            # numbers, or ...
    - name3:  |                     # text block, that preserves newlines
              ls -l /foo
              echo %name
    - name4:  >                     # text block, wrapped. May omit > char.
        This form is good for long command lines.
```

Variables ... are discussed in depth on the next page (at *vars*) but the short story is that they are created in the `vars:` section (see above) and can be expanded in string values with `%varname` style syntax.

Quoting

- If a declaration or command-line string contains positional arguments and an argument needs to contain whitespace, then it should be quoted just like it would in a shell. Also:
- Use double quotes and C/Python-style escape sequences to denote control (or difficult to type) characters in strings:

  ```
  "\n", "\x##" and "\u####"
  ```

- Conversely, use the "stronger" single quotes when a string has backslashes that need to be sent literally, such as definition of regular expressions. If not you'll get errors, such as "`s is not a valid escape sequence`.":

  ```
  - '#\s*password_encryption password_encryption'
  ```

- Quoting a quoted string (with the other kind of quotes) may need to be done if the string starts with a quote but ends early, :/ e.g.:

  ```
  - ' "-s /tmp/memcached.socket" /etc/memcached.conf '
  ```

Blocks, Braces

- If you'd like to avoid one level of the quoting mess above, block styles (see yaml docs above) are the way to go. They are useful for longer text (or command-lines) with special characters or regexes.
- A block is signified by use of an optional block character, > or |, and newline followed by the indented body. To enter a block that will be wrapped (newlines converted to spaces) you may omit the character, or add it explicitly. This one is good for long command-lines:

  ```
  important:                  # Add > char here to explicitly wrap.
      sudo foo install
      https://github.com/developer/repository/tarball/master
  ```

  When newlines need to be preserved, use the | char, which is helpful when writing haiku I suppose:

  ```
  status: |
      I wakey wakey.
      Production release today.
      No breaky breaky.
  ```

- If using brace variable expansion (python string.format), use quotes for strings when they begin with a brace character:

  ```
  - '{user}'
  ```

Conditional Execution

Poor Man's Conditionals If your needs in the area are modest, or you'd prefer to avoid external dependencies, you'll likely be able to squeak by with using one of the alternatives below. Separate pavefiles with includes are an option as well.

There are a few types of simple conditionals supported in the pavefile, executed either locally or remotely. They are supported by all standard modules. Add these to items under the `tasks:` section as needed.

- *Local* conditionals:
  - `if: VARNAME COMPARISON VALUE` This clause will compare a variable from the `vars:` section to a value locally, and if true, execution of the attached item will continue. E.g.:

    ```
    - scp:
        - if: inst_ssl == True
          do: # ...
    ```

    This is the only place where the variable name needn't be dereferenced with the `%` character. If there were whitespace in an unquoted variable for example, an error would occur, however this convenience-form avoids it and is easier to type.

    Comparisons are currently done as strings only, meaning 'True' equals a boolean True, while supported comparisons are == and `!=`.

- *Remote* conditionals:
  - `if-exec: EXECUTABLE_STATEMENT` Execute this statement to do reconnaissance at the *remote* end before primary task execution. If the statement returns with a exit status code of 0 (meaning true), the primary statement is run also. Conversely, a return code of >= 1 (false) will skip the task. Change/Skip events are handled automatically.

    ```
    - run:
        - if-exec: test ! -f media/js/site.js.gz
          do: # ...
    ```

- *Manual* remote conditionals
  - Similar to `if-exec:` above, use `test` (aka `[ ... ]`), `grep` or other command-line program before the primary statement. See the Events section below on manual event dispatch.
- *Remote* conditional handling II:
  - `if-platform: PLATFORM if-platform: KEY COMPARISON VALUE` Make decisions based on what we're talking to:

    ```
    - packages:
        if-platform: Ubuntu
        install: sysvbanner

    - packages:
        if-platform: arch[0] == 64bit
        install: ia32-libs
    ```

    The available platform keys with example values are given below. The single token PLATFORM format is usually a shortcut to a simplified version of `linux_dist[0]` on Linux:

    ```
    arch:         ['64bit','ELF']
    encoding:     'UTF-8'
    linux_dist:   ['Ubuntu','13.10','saucy']
    machine:      'x86_64'
    node:         'darkstar.home.com'
    os_release:   '3.11.0-15-generic'
    proc:         'x86_64'
    py_vers:      '2.7.5+'
    py_vert:      ['2','7','5+']
    system:       'Linux'
    version:      '#23-Ubuntu SMP Mon Dec 9 18:17:04 UTC 2013'
    ```

    Additional example values:

    ```
    mac_vers:     ['10.9.1', ['', '', ''], 'x86_64']  # system: 'Darwin'
    win_vers:     ['XP', '5.1.2600', 'SP3', 'Uniprocessor Free']
    ```

    This information may be found under `~/.cache/pave/platform.json` on the client and each target. It is also logged and printed to the console at debug level when pave is run.

Templating For more robust conditional execution, there is the option of using a templating system to build your pavefile on the fly. Remote execution and platform checks are not available with this method, but continue with those previously mentioned. Currently jinja2 is supported.

Usage:

0. Install it if it has not been already, e.g.:

   ```
   sudo apt-get install python-jinja2      # yum, brew, pip, etc.
   ```

1. Create a pavefile with variables (`vars:` section) at the top. Do not include jinja markup in this section. Assign variables (with standard expansions if desired).
2. Add jinja markup to the rest of the document as needed, saving with a .j2 file extension.
3. Run pave with the `-f/--filename` command-line parameter, using a .j2 extension, e.g.:

   pave -f paradise.j2

Events Events are simple signals that are output during task execution. Modules or commands use them to tell pave what has occurred. Events are generated and handled *automatically* in all modules in the included library, with one exception. When bare strings (commands) are children of `tasks:`, `run:` events must be dispatched manually (see below).

Three types of events are tracked by pave.

Change or Skip

When a change has been made or pre-task test is passed (meaning a change needs to be made) a "change event" is generated.

A skip event is given instead, if a task decides the change is not currently necessary.

Error

Error events are a third kind of event that may occur. These will halt execution on the affected host if the `warn-only:` option is disabled in the `main:` section.

Change and error events are counted and returned to the user at the end of a completed task list.

Manual Dispatch from Command-Strings

- Typically a successful command returns a `0` status code, failure non-zero.
- To harness these follow a command-string with,
  - `" && echo CHANGED"`, which will register a change event,
  - `" || echo SKIPPED"`, a skip event, and/or
  - `" || true "`, to prevent an unimportant error from halting execution.

  Here's an example that combines a manual conditional with manual event dispatch:

  ```
  - "[ '%mode' == 'dev' ] && wget %url && echo CHANGED || echo SKIPPED"
  ```

  A multi-line statement can be sent as well (assuming the remote shell supports it) and you have correctly marked it as a yaml text block (see | or > chars in General Layout)
- CHANGED wins over SKIPPED if the both are output somehow.

Using Custom Modules A custom or third-party module to use in the `tasks:` section should follow the interface and format of the included library modules, as discussed in the Developer Guide.

To use such a module, you may move it to a location in python's `sys.path` (the current directory "." may already be there). For convenience the folder `~/.config/pave/lib/` is added to the path automatically, and as such is a good place to put custom modules.

Alternatively, you can modify the path to contain additional folders per project. This can be done by adding the path to `/main/sys.path` option in the pavefile, or setting the PYTHONPATH environment variable.

Disabling Sections/Tasks By convention, all data under `disabled:` is ignored—A place to temporarily bypass tasks without having to futz around commenting them.

Task selection, using the `-s/--select` command-line parameter is also available as an alternative. See the Command-line Reference.

Full Examples

- https://bitbucket.org/mixmastamyk/pave/src/default/examples/

Next, kindly continue on to the next chapter, *Pavefile Reference Part II*, for details on the standard sections and task module library.

Pavefile Reference Part II Sections and Task Library available to pave's configuration file are described here. For syntax, conditionals, and general information, see the previous chapter *Pavefile Reference*.

Sections A section is a top-level portion of a pavefile, which is organized hierarchically by indentation.

Names of sections are given by the labels at the far left in the first column. Names and the data (typically indented) below it are considered the complete section. Therefore, a name in the first column ends the preceding section and starts a new one:

```
# end previous section
foo:            # The "foo" section
    - bar
    - baz
# start next
```

The standard sections described next are handled by pave itself. Tasks implemented by library modules are described afterwards under Tasks Library and are placed in sub-sections under `tasks`.

---

vars A fine place to define variables, `name:  value` style.

When pave encounters `name` constructs in a string value it will replace them with a value given in this section. This expansion syntax is described in depth here, with the exception that pave uses the **percent** character to avoid clashing with shell scripting.

Pavefile strings support %-based substitutions, using the following rules:

- Expansion can be avoided by doubling the `%` character.
- `%identifier` names a substitution placeholder matching a mapping key of "identifier", which must spell a pave variable. The name must be a letter or underscore, followed by a letter, digit, or underscore. The first non-identifier character after the % character terminates this placeholder specification.
- `%{identifier}` is equivalent to `%identifier`. It is required when valid identifier characters (including underscores) follow the placeholder but are not part of the placeholder, such as `%{noun}ification` or `%{username}_backup.zip`.

Notes:

- Passwords are added to the pavefile vars with the prefix "`pwd_`" e.g., `pwd_appuser` to support variable interpolation and avoid conflicting with existing variable names.
- Variables in the var section will be expanded against themselves just once, in order to create vars that contain fragments of other vars—useful for building paths and such.

The order of variable expansion is as follows:

1. Optional: A templating engine, such as Jinja.
2. string.Template expansion of variables from the vars section performed locally by pave.
3. Bash or other chosen shell will expand and run command strings in the the remote execution context.

Example:

```
vars:    # I put this at the top of the file for easy-access, baby.
    sitename: fooBar
    repo: "http://bitbucket.org/username/%sitename"  # existing var

tasks:     # usage, quotes not needed:
    - "echo %sitename rules"
```

Schema:

dict of *boolean* || *float* || *integer* || *list* || *string*

main Where options to pave are specified. *Required.*

bak-files: Copy files to be modified to NAME.orig beforehand.

env: A place to directly set fabric's environment attributes. Several items, such as hostnames, will be clobbered if you set them here, since pave manages them.

include: Include another file into this pavefile. Multiple entries accepted. Note that included sections will overwrite identical sections in the parent. Useful for separating lengthy orthogonal sections, e.g. `target-groups:`

inspect: Inspect the target to determine platform details. Values:

- `True` - Use the python inspector, if it fails try the shell script next. (Default)
- `False` - Disable inspection, not currently useful.
- `py` - Use the python inspector script only.
- `sh` - Use the sh inspector script only.

jobs: Number of "steamrollers." Each task-list is run in parallel (per host) with the given pool size.

log-to: Specify logging/data folder.

targets: Specify hosts (or groups of hosts) to target. Whitespace-delimited string or list. See the `target-groups:` section below for group definition.

user: Change the login user, defaults to the current one.

version: Version of pave this file was written against, enabling time-travel. Not yet implemented.

warn-only: Continue thru errors with a warning only.

Example:

```
main:
    user: ubuntu
    sudo: True
    targets:
        - 192.168.2.1     # a list or string of hosts, groups to greet.
        - www[1-3]
```

Schema:

dict of

bak-files: *boolean* default: `True`

env: *dict* default: `{}`

include: *list of string*

string default: []

inspect: *boolean* || *string* default: True

jobs: 100 True 1 True *integer* default: 1

log-to: *string* default: ''

sudo: *boolean* default: True

sudo-user: *string*

sys.path: *list of string*

string

targets: *list of string*

string default: ''

user: *string* default: 'CURRENT'

version: *string* default: 'CURRENT'

warn-only: *boolean* default: False

passwords Set or query passwords for later use.

- `ask NAME PROMPT` Query the user for this password, named NAME, given PROMPT, prior to execution. As an "escape hatch," two empty passwords given at a prompt will exit the program.

  Note: do not add a colon character to the end of the PROMPT value. One will be added automatically.

- `set NAME PASSWD` Hard-code something temporarily (not recommended). This can greatly speed up authoring of pavefiles—But don't check these in! ;) See `/main/include` to offload this section to another file instead. Using another file (with a standard name) and adding it to your vcs ignore file will greatly decrease the chances of a password checkin.

Note: 99% of the time when starting out you'll want to give "password" (no quotes) as the value for NAME. This is the name fabric uses for the ssh and/or sudo password.

How to use gathered passwords in other sections:

- `use NAME` Will plug-in the named password into another section's options. See `users:` or `postgres:` for examples.
- `wget --password="%pwd_websvc"` Use variable expansion to render the password into a command-line task. The variable name should be prepended with `pwd_` to avoid collisions.

Example:

```
passwords:
    - ask password  ssh/sudo Password             # fabric default
    - ask pgsu      Postgres SuperUser Password
    - set appuser   "hello world"                 # set not recommended
```

Schema:

list of *string*

target-groups A mapping of group names assigned to lists of targeted hosts.

Example:

```
target-groups:
    dbservers: [host1, host2, host3]     # compact yaml list syntax
    webservers:
        - www[1-3]
```

This is a good place to mention hostname expansion. Leading `0`'s specify padding:

`host[1-3] —> [host1, host2, host3]` `host[01-03] —> [host01, host02, host03]`

Schema:

dict of

string: *list of string*

string

fab-tasks A list of tasks to run directly from an existing `fabfile.py`.

These "fab-tasks" are run before standard task execution and do not interact with standard tasks. However, the fabric `env` (ironment) variables that have been set by pave are passed forward to these fab-tasks, allowing you use pave's target groups and misc settings to drive fabric as if you were using the `fab` command.

`env.fabfile:` Set the path of a fabfile here if it is not in the current or parent folders. Relative paths and tilde expansion are supported.

Arguments and keyword values are passed in using the same command-line syntax fab uses. For example:

```
main:
    env:
        fabfile: ../fabric/fabfile.py   # optional

fab-tasks:
    - task1
    - task2: arg1
    - task2: arg2,kwarg1=kwval1
```

Schema:

list of *string dict*

tasks A listing of tasks to be processed in order.

Each task in the list should be a bare string to be executed by the `run` module, or a mapping of a module name to the data it requires. The second form is called a "task group" at times as it may contain numerous sub-tasks.

See the Tasks Library below and *Conditional Execution* for more details.

Example:

```
tasks:
# tasks could be wide enough that you may want to skip the optional
# yaml list indent.

- banner "Hello!"      # world - strings get passed to the run module

# use the users module explicitly, a "task group"
```

```
- users:
    - name: appuser
```

Schema:

list of *string dict of*

`modname:` `moddata`

Tasks Library Each mapping listed under tasks corresponds to a module in the `site-packages/pave/lib` folder or a third-party module found on the `sys.path`. See the Developer Guide for details.

cleanup Good hygene is important.

`folders:` Deletion of files under `/tmp`, `/var/tmp`, & `/home/` only.

`locales-except:` *"Unless you're working at the UN and administer a central server for all member states, it is difficult to conceive why you would need a system where all of these locales are installed."* Similar to a one-time Debian locale-purge.

`packages:` Run appropriate package cleanup command(s).

Example:

```
- cleanup:
    packages: True
    locales-except: en en_US
    folders:  # use with caution
        - /tmp/stuff/*
        - /tmp/stuff2
```

Schema:

dict of

`do:` *string*

`folders:`

list of

string

default: `[]`

`if`:

`if-exec:` *string*

`if-platform:` *string*

`locales-except:` *string*

`packages:` *boolean* default: `True`

configure Configure the remote system. Also handles templating, see below.

`render:` Renders a template to a local file, which is then uploaded to host.

`engine:` The templating language, such as python (printf|Template|format), or the executable name of an external package, e.g: `jinja2`

`render-remote:` Simple remote "templating" with sed.

Example:

```
- configure:            # str || list accepted on all sub-tasks below:
    create:  # file or folder/
        - foo.conf
        - /tmp/project/

    append:  # text remotefname
        Answer=42 %rc

    comment: # [comment char] from-regex remote-fname
        - ' "insecure on" %rcfile '

    replace: # from-regex to remote-fname
        - ^foo$ bar %rcfile
        # note that because of yaml syntax and splitting, two layers
        # of quoting or blocks may be needed when backslashes used:
        - >
            '^foo\s*$' bar %rcfile

    render: # engine [extra-vars] local-template remote-fname
        - printf swappiness=50 examples/printf_template.txt %rc2
        - format FOO=BAR       examples/format_template.txt %rc3

    render-remote: # [vars] remote-source remote-dest
        - INSTANCE_NUM=1 %sitecfg /etc/init/%svcname.conf

    update: # copy a file into place if newer, SRC DEST
        - ~/%sitename/cfg/celeryd.conf /etc/init/celeryd.conf

    access: # mode user group remote-fname # use "" for empty
        - 0640 %user %user %rc2
```

Schema:

dict of

`access`: *list of string*

string default: `[]`

`append`: *list of string*

string default: `[]`

`comment`: *list of string*

string default: `[]`

`create`: *list of string*

string default: `[]`

`do`: *string*

`if`:

`if-exec`: *string*

`if-platform`: *string*

`render`: *list of string*

string default: `[]`

render-remote: *list of string*

string default: []

replace: *list of string*

string default: []

update: *list of string*

string default: []

django Django helper module. *Captain Obvious recommends Django be installed on the target beforehand, see the* packages *section.*

loaddata: Load fixture files from given path, glob ok.

manage: Execute a management command every time this task group is run.

manage-once: Once and never again.

Example:

```
- django:
    workdir: ~%sitename/%sitename
    loaddata: "fixtures/*.json"
    manage-once: update_index
    syncdb: True
```

Schema:

dict of

do: *string*

if:

if-exec: *string*

if-platform: *string*

loaddata: *string* default: ''

manage: *list of string*

string default: []

manage-once: *list of string*

string default: []

migrate-init: *boolean* default: True

syncdb: *boolean* default: True

user: *string* default: None

workdir: *string* required

groups Create operating system user groups.

Example:

```
- groups:           # str || list || conditional
    - appgroup
    - "dev:1100"    # how to set gid

    - if-exec: test -e /dev/cdrom
      do: cdrom
```

Schema:

list of string

dict of

`do:` *string*

`if:`

`if-exec:` *string*

`if-platform:` *string*

kernel Manage kernel parameters. Saves to memory and disk.

Example:

```
- kernel:           # str || list || conditional
    - vm.swappiness = 10
```

Schema:

list of string

dict of

`do:` *string*

`if:`

`if-exec:` *string*

`if-platform:` *string*

keygen Manage local/remote ssh keys and their generation.

`remote:` Generates a key at the remote host under the given username.

`copy-id:` Copies local keys to the remote host.

Values may be a single or list of Boolean (True copies the public key of the local user to the authorized keys file of the remote user), or string of two arguments containing the local file (.pub appended automatically) and the remote user.

Example:

```
- keygen:
    remote: "%sitename"              # quotes not required but
    copy-id:                         # fixes syntax highlighting
        - True                           # default src dest
        - "%local_path.pub %user"       # specified
```

Schema:

dict of

copy-id:

list of string *boolean*

boolean || *string* default: `True`

do: *string*

`if:`

if-exec: *string*

if-platform: *string*

passphrase: *string* default: `''`

remote: *None* || *string* default: `None`

type: *string* default: `'rsa'`

packages Manage software packages. Packages names for `install`, `remove`, etc. are listed as whitespace-delimited strings.

expiration: The age in days after which the OS package index is considered to have expired, requiring action.

To be a good neighbor, the default, 0 (= 24 hours) means packages won't be updated unless it has been a full day since the last update. A value of -1 forces action.

inventory: Make a list of installed packages to compare against.

update/upgrade: Update the system's package index, upgrade packages to latest release.

upgrade-full: Upgrade packages, even if it causes obsolete packages to be removed, or new packages to be installed, for example new kernel versions. e.g. Debian's "dist-upgrade".

Example:

```
- packages:
    if-platform: Debian
    install:
        sysvbanner python-pip
    # expiration: -1    # force attempt
```

Schema:

dict of

do: *string*

expiration: *integer* default: `0`

`if:`

if-exec: *string*

if-platform: *string*

install: *string* default: `''`

inventory: *boolean* default: `True`

remove: *string* default: `''`

update: *boolean* default: `True`

upgrade: *boolean* default: `True`

upgrade-full: *boolean* default: `True`

postgres Manage postgres. *Captain Obvious here to chime-in that postgres must be installed on remote target beforehand, see* packages.

link current: Create a symbolic link from the last sorted item in `/etc/postgresql/` to `/etc/postgresql/curr`.

replace, access: See the corresponding directive under configure. These are for convenience to group postgres-related configuration together.

user, db, grant, sql: `postgresql-client-common` (or distro equiv.) must be installed on remote target beforehand. Run the appropriate commands to create objects or configure the database. String or list of strings may be given.

Example:

```
- postgres:
    link current: False
    replace:
        - '#\s*password_enc password_enc %pgcfg/postgresql.conf'

    su-password:
        use pgsu                                # see passwords:
        # md5 acbd18db4cc2f85cedef654fccc4a4d8  # how to hard-code

    db: # quotes not required below, but fixes highlighting. :/
        - name: "%sitename"
    user:
        - name: "%sitename"
    grant:
        # privs   on_obj    named         to_user
        all       database  %sitename     %sitename

    sql: # str|list lines of sql, first token should be dbname:
        template1 select datname from pg_database
                  where datname = '%sitename';
```

Schema:

dict of

access: *list of string*

string default: `[]`

db:

list of dict of

description: *None* || *string* default: `[]`

encoding: *string*

locale: *string*

name: *string* required

tablespace: *string*

template: *string* default: `'template0'`

do: *string*

grant: *list of string*

string default: `[]`

`if`:

`if-exec:` *string*

`if-platform:` *string*

`link current:` *boolean* default: `True`

`replace:` *list of string*

string default: `[]`

`sql:` *list of string*

string default: `[]`

`su-password:` *string* default: `''`

`user:`

list of dict of

`createdb:` *None* || *boolean* default: `False`

`createrole:` *boolean* default: `False`

`encrypted:` *boolean* default: `True`

`login:` *boolean* default: `True`

`name:` *string* required

`superuser:` *boolean* default: `False`

run Executes a list of commands using the remote shell. Each command may be:

- **A string** A command that runs under the default user.
- **A mapping with one or more of the following members:**

`do: CMD_STRING`
The command to run.

`user: USERNAME`
Run as another user.
May be `sudo` (root), or a valid remote username.

`title: TITLE`
to document or hide long, ugly commands.

`workdir: PATH` set working directory.

`local: <bool>` runs it locally instead of remotely.

See *Conditional Execution* for more details on `if:` and `if-exec:`.

Note: While *not recommended* to set passwords via command-line tools for various reasons, tasks makes an attempt to hide them (vars starting with "`pwd_`") from standard logs. Passwords will still be displayed in debug logs (and perhaps shell history) upon command-line execution.

Example:

```
- banner "Hello!"     # world

# signal a change event and prevent errors from halting execution:
- touch %filename && echo CHANGED || true
```

```
- rm /etc/nginx/sites-enabled/default && echo CHANGED || echo SKIPPED

# Even poor-er man's conditionals:
- " [ '%mode' == 'qa' ] && echo 'QA install' "

# more robust task definition:
- run:
    - title: Nothing to see here...
      if-exec: test -f /foo
      do: echo "This is running under $USER (appuser)"
      user: appuser
```

Schema:

list of *string dict of*

`do:` *string*

`if:`

`if-exec:` *string*

`if-platform:` *string*

`local:` *boolean*

`title:` *string*

`user:` *string*

`workdir:` *string*

scp Copy files to and fro. Given a source and destination, copy files.

Notes:

- By default a put (upload) is performed, but the direction can be specified as the first parameter.
- Paths with whitespace must be quoted.
- Destinations folders should end in a `/` character for efficiency, as doing so skips the need for several filesystem tests.
- If sudo is enabled, relative paths/tildes will be interpreted (and files owned) by the root user. Therefore it is recommended that absolute remote paths be given.
- Does not currently handle wildcards.
- While this module is similar to using the scp command-line program it may not work identically.

Example:

```
- scp:
    - SRC DEST          # put
    - SRC DEST MODE     # put, w/ Unix octal mode or "same"
    - get SRC DEST

    # example conditional
    - if: inst_ssl == True
      do: put bundle.crt "%sitefldr/ssl/bundle.crt"
```

Additional details:

- fabric.operations.get

- fabric.operations.put

Schema:

list of *string dict of*

> **do:** *string*
>
> if:
>
> **if-exec:** *string*
>
> **if-platform:** *string*
>
> ***string:*** *string*

services Manage system daemons/services. Values are whitespace separated strings.

restart:, restart-fail: restart will start the service if it was stopped, restart-fail will fail with an error if it was not running originally.

Note: Some daemon startup scripts are not written well and quit when the ssh session ends. To prevent this from occurring, prepend the service name with "nohup:" as shown below.

Example:

```
- services:
    # the follwing two are temporary until reboot:
    running: hostname hwclock
    stopped: whoopsie
    # permanently; /packages/remove more reliable on ubuntu
    disabled: whoopsie
    restart: nohup:memcached
```

Schema:

dict of

> **disabled:** *string* default: ''
>
> **do:** *string*
>
> if:
>
> **if-exec:** *string*
>
> **if-platform:** *string*
>
> **restart:** *string* default: ''
>
> **restart-fail:** *string* default: ''
>
> **running:** *string* default: ''
>
> **stopped:** *string* default: ''

users Create operating system users.

Example:

```
- users:
    - name: appuser                # required
      groups: appgroup www-data    # whitespace separated groups
      create-home: True
```

```
    shell: /bin/bash
    password: use appuser         # see passwords: section
```

Schema:

list of

dict of

create-home: *boolean* default: `True`

do: *string*

groups: *None* || *string* default: `None`

`if:`

if-exec: *string*

if-platform: *string*

name: *string* required

password: *string*

shell: *string*

vcs Deploy code with this version control helper. $VCS support must be installed beforehand, see the packages section.

If the destination does not exist, an initial checkout is done. If it does exist, the repository is updated to the latest version.

Formats:

```
- VCS_TYPE REPO_URL DEST EXTRA          # simple or
- do: VCS_TYPE REPO_URL DEST EXTRA      # robust
```

VCS_TYPE: Version Control System, one of `svn`, `hg`, or `git`.

REPO_URL: URL to repository.

DEST: Destination folder.

EXTRA: Extra command line parameters.

do: What to do.

input: Useful to automate interactive prompts.

user:: Execute vcs under this remote username.

Example:

```
- vcs:
    - do: hg %repo_base/appname ~%user/appname
      input: p\nn                          # interactive answer
      user: %user
```

Schema:

list of *string dict of*

do: *string*

`if:`

if-exec: *string*

if-platform: *string*

input: *string*

user: *string*

Command-line Reference

Usage

```
Usage:
    pave v0.68 - (C) 2012-2013 Mike Miller.
    License: GNU GPLv3+
    Yet another config and deployment tool, leveraging fabric.

    pave [options]

Options:
  --version              show program's version number and exit
  -h, --help             show this help message and exit
  -b, --brace-exp        Use string.format-style brace expansion instead of
                         printf.
  -c STR, --command=STR
                         Override pavefile tasks with an ad hoc command.
  -f NAME, --filename=NAME
                         Use an input file besides ./pave.yml.
  -i, --interactive      Stop and ask whether to run each task group.
  -j #, --jobs=#         Run pave tasks in parallel with the given pool size.
  -o N V, --option=N V   Override a pavefile option.
  -s #, --select=#       Select task groups to run, instead of all. (0-based
                         slice syntax, # or #:#).
  -S, --skel             Output a "skeleton" pavefile to get started on.
  -t T, --target=T       Override targets with this host or group.  Multiple
                         accepted.
  -q, --quiet            Silence routine output, allowing warnings and errors.
                         Overrides verbose.
  -v  N V, --var= N V    Override a pavefile variable.
  -V, --verbose          Enable overwhelmingly verbose debugging output.
  --color=CHOICE         Message coloring/icons ('auto', 'on', 'off').
  --crypt                Ask for a password, print its hash and exit. For use
                         with users: module in the pavefile.
  --speak                Talk to me, holmes.
  --test                 Test mode: parse and validate, skip execution.
```

Exit Status Codes Pave returns standard status codes from `sysexits.h`:

```
EX_OK           0       # successful termination
EX_USAGE        64      # command line usage error
EX_DATAERR      65      # data format error
EX_NOINPUT      66      # cannot open input
EX_NOUSER       67      # addressee unknown
EX_NOHOST       68      # host name unknown
EX_UNAVAILABLE  69      # service unavailable
EX_SOFTWARE     70      # internal software error
EX_OSERR        71      # system error (e.g., can't fork)
EX_OSFILE       72      # critical OS file missing
```

```
EX_CANTCREAT    73      # can't create (user) output file
EX_IOERR        74      # input/output error
EX_TEMPFAIL     75      # temp failure; user is invited to retry
EX_PROTOCOL     76      # remote error in protocol
EX_NOPERM       77      # permission denied
EX_CONFIG       78      # configuration error
```

Password "Escape Hatch" In addition to the standard keyboard interrupt `Ctrl+C`, if a password is requested (via the pavefile) at the command line the process can be escaped by giving two empty passwords in a row. The program will exit.

Option Path Syntax To override pavefile options on the fly, "option path" syntax is used, which describes the hierarchy in a compact manner. This syntax is used for command-line (`-o|--option`) overrides. Inside the pavefile, however, path separators (`/`) are *not* supported; indentation is used instead.

```
main:           # pavefile
    jobs: 1
```

equals:

```
/main/jobs 1    # command-line
```

Task Selection Selection of a subset of tasks (or "task groups") can be done with the `-s/--select` option, and uses Python-style "slicing" syntax.

Indexes are zero-based, and use negative numbers to specify items from the end. Start and stop indexes, and/or step values may be given and are separated with the colon : character. If a part is omitted, it defaults to start, end, or a step of 1.

Below is a visualization that may be helpful. Given a list of tasks, the task index marks the beginning (left or top) edge:

```
 +---+---+---+---+---+              # horizontal
 | T | T | T | T | T |
 +---+---+---+---+---+
 0   1   2   3   4   (5=length)
-5  -4  -3  -2  -1

- TASK1, index  0  or  -5           # vertical
- TASK2, index  1  or  -4
- TASK3, index  2  or  -3
- TASK4, index  3  or  -2
- TASK5, index  4  or  -1
```

Example:

```
pave -s 0               # the first task, group only
pave -s -2              # the second to last task, group
pave --select 10:20     # task groups 10-19
pave --select :6:2      # every other task from the first six
```

Further discussion can be found here:

- **http://docs.python.org/2/tutorial/introduction.html#strings** (slicing discussion starts about halfway down the section.)
- http://stackoverflow.com/a/509295/450917

Developer Guide *"Use the SOURCE, Luke."*

Source Tour Pave is GPLv3+ licensed and may be available in commercial-friendly proprietary license in the future as well.

Follow along at bitbucket if you'd like. There is an executable script typically installed to `/usr/local/bin/pave` that does little but hand things off to the main module `pave/main.py`.

This main module gets everything rolling...

- In `load_data()`, the pavefile is parsed and validated via `pave/schema.py` using the voluptuous library.
- The tasks(s) are then handed off to fabric for execution at the end of `main()` — each host gets `steamroll()`'ed.
- Each remote host is first inspected with `pave/inspector.py`. A json blob is cached in the .cache folder and returned with the results.
- With inspector results in hand, a set of platform-dependent shell commands and settings (as specified in `pave/platforms.py`) are chosen.

  This is the place to start on new platform support.

- The `pave/lib/` folder contains the modules that implement items under the `tasks:` section of the pavefile.

  This is the place to add modules to support new functionality.

How to Write a Task Module So, you'd like to create a module to simplify an ugly or tedious task—this is the right place. Task modules are written in Python and run locally, but generally speak in shell script at the remote end. (If you don't know either language, there's still hope, an alternative described in the *FAQ - (Not So) Frequently Asked Questions*).

Let's get started:

1. You'll probably want to copy or at least refer to an existing one from pave/lib that is most similar to one you'd like to write.

   If the module will do a lot of repetetive tiny tasks, you may want the primary data structure to be a list. For a single task with lots of options a mapping is a good choice. Keep this in mind when choosing a module to copy from.

2. The module needs the following items:

   - a handler function of the following signature:

     ```
     def handle(data, cmds, context):
     ```

     – `data` is the task's parsed structure passed in from the pavefile. The convention when using mapping/dict keys is to use dashes instead of spaces, like css, e.g.: `foo-bar:`

     – `cmds` is the platform information object chosen by the inspector, the attributes of which should be used in place of command strings.

     – `context` is a dictionary that keeps track of properties such as the current user, whether to use sudo, and if so, as what user, etc. *Make a copy if you need to modify it per-task*:

     ```
     for task in tasks:
         mycon = context.copy()       # prevents spills
     ```

   - To check against data-entry errors, the passed `data` structure should have a robust schema definition (and validator functions), the building blocks of which can be imported from voluptuous.

- Generally speaking if the data is a list you'll want to loop over it, if a dictionary, you might .get() or .pop() off each member as you go.
- There are a few helpful variables pave attaches to fabric that are accessible by importing its `env` variable:

```
env.pave_platform           # the platform obj also known as cmds
env.pave_platform_details   # a dict holding system parameters
env.pave_vars               # a dict holding pavefile vars:
env.pave_passwords          # a dict holding pavefile passwords:

# configures pave to stop at the first error:
env.pave_raise_errs         # set by /main/warn-only

# makes copies of original files before changing them:
env.pave_bak                # set by /main/bak-files
```

3. Use the helper functions in `pave/utils.py` whenever possible. They handle common situations for modules.

   `runcmd()` is probably the most important, as it wraps fabric operations. See below, `cmds` provides the cmd-line for the current platform, while arguments for printf-style `%s` string expansion follow. Finally `context` provides information for runcmd to make decisions:

```
result = runcmd(cmds.pkg_upgrd_time, expires, **context)
```

   - A note about command strings from `platforms.py`. Occassionally string formatting with positional args is not sufficient, and the need arises to add or subtract whole parameters based on vars, similar to python keyword arguments. This is possible thru what the source calls (for lack of a better term) "chunking"... referring to the chunks in the command line:

```
createdb [--locale] [--encoding] [--tablespace]      # kwargs
createuser [--encrypted|no-encrypted]                # boolean
```

     Chunking can be enabled via `context.chunking = True` and can be seen in the postgres, users, and groups modules, for example.

4. Try to reuse the other modules if possible. For example, the `postgres:` module leverages the `configure:` module for its config file modifications.
5. If a task is about to make a change, log it with `log.change()`. If the task should be skipped, log it with `log.skip()` instead.
6. Keep a running tab of these "events," and return the number of changes and/or errors (in that order) generated during the module handler run.
7. When satisfied with the module, run `pyflakes` and/or `pylint` against it to light up dark corners. There is a `check_files.sh` in the root of the project to help with this. This would also be a good time to write a few tests for it.

- A final note to be careful about importing `pave.main`, `pave.schema`, or handle methods (of other library modules) in new modules. Doing so can create circular dependencies, as the schema module imports all modules at startup looking for schemas. Instead, import these into each needy function to delay import until runtime.

Roadmap and Wishlist Can you help with any of these needs?

- **Platforms - Would appreciate some help getting it to run on:**
  - RHEL6+ and other common distributions.
  - *BSD, OSX client
  - Windows

- Common software support which has special needs or tedium that aren't met with the current library.
- The validation of pavefile sections and options is lacking. For example, data types and number of arguments are generally checked, but files aren't checked for existence prior to execution. This is important for a 1.0 release. Check the voluptuous page linked above on how to do that.
- Optimizations from gurus of various disciplines.
- Necessary features from established systems that don't harm the simple use-cases.
- Tests — The design is starting to solidify, so writing tests would be useful.
- Docs too. Sphinx doc building is currently a bit of a mess.

  Note: parts of the docs are built dynamically, such as schema-related details. Unfortunately, readthedocs doesn't handle this. :/ So, docs will need to be re-built after changes to the schema and the output files checked in. There is a script called `docs/bld_docs` to automate the process.

Contributing

- Please consult with me before making major improvements if you'd like them included here. ;)
- Currently we're asking that you transfer copyright of your additions to the project to avoid issues if a change of gears is warranted. Is this a problem?.
- **Pull requests:**
  - Please fix/improve one thing (or highly-related things) at a time.
  - Please test thoroughly before submission as I don't have much time or resources available. VirtualBox is free and easy to use, (if there isn't something you'd rather use).
- **Code should be in the pep8 style.** I tend to do a few other things like line up columns and put double blank lines between classes and standalone functions that might bug people. Sorry about that. I've got an extra high-res portrait monitor for coding/browsing and vertical space is plentiful.
- Single quotes by default, since they are easier to look at and type on keyboards in common languages.

  One exception are the command-lines in `platforms.py`. Since fabric doesn't escape single quotes in command-lines, is it easier to visually parse the shell debug output if you quote with single on command-line tasks whenever possible. As single quotes will therefore be common inside command-line strings it is easier to wrap the whole thing in double-quotes or (if mixed, triple-single) in Python.

Module Reference

pave

pave Package

`cmds` Module

`inspector` Module This is a short script which is copied over to remote hosts to gather details. Uses python's platform module to gather the info, which is cached in the ~/.cache folder in json format.

`pave.inspector.`**`query`**`()`

main Module %%prog v%s - (C) 2012-2014 Mike Miller. License: GNU GPLv3+ Yet another config and deployment tool, leveraging fabric.

%%prog [options]

`pave.main.`**`expand_targets`**(*targets*, *groups*)
: Convert group names into targets and add them to the list.

`pave.main.`**`expandstr`**(*self*, *node*)
: Yaml loader/constructor that expands variable strings.

`pave.main.`**`fatal_exit`**(*msg*, *exitcode*, *parser=None*)
: Convenience function to log and exit on fatal error.

`pave.main.`**`get`**(*cfgdata*, *path*, *offset=0*)
: An xpath-like yaml data grabber. Looks at these in order, last one wins:

- Schema default
- Config file
- Command line

`pave.main.`**`get_module`**(*modname*)
: Given a module name, find and load it.

`pave.main.`**`get_platform_cmds`**(*pldata*, *local=False*)
: Given platform data, return the appropriate command class.

`pave.main.`**`load_data`**(*opts*, *args*)
: Load the pavefile and expand any variables found.

`pave.main.`**`main`**(*opts*, *args*)
: Start paving.

`pave.main.`**`prep_environ`**(*data*, *opts*)
: Prepare execution environment according to options given.

`pave.main.`**`setup`**()
: Parse command-line, create temp folders, set up logging.

`pave.main.`**`steamroll`**(*data*, *inspectors*, *context*)
: Let's get rollin...

schema Module (C) 2012-2014 Mike Miller. License: GNU GPLv3+

Strangely enough, schema and schema-related code lives here.

exception `pave.schema.`**`SchemaError`**
: Bases: `exceptions.Exception`

`pave.schema.`**`if_var_three_args`**(*v*)

`pave.schema.`**`validate_varname`**(*value*, *msg=None*)
: Rules for variable names in the pavefile.

targets Module This module is for enhancing the inventory parsing capability such that it can deal with hostnames specified using a simple pattern in the form of [beg:end], example: [1:5] where if begin is not specified, it defaults to 0.

If beg is given and is left-zero-padded, e.g. '001', it is taken as a formatting hint when the range is expanded. e.g. [001:010] is to be expanded into 001, 002 ...009, 010.

Note that when beg is specified with left zero padding, then the length of end must be the same as that of beg, else a exception is raised.

exception `pave.targets.HostSpecError`
Bases: `exceptions.Exception`

`pave.targets.detect_range(hostspec)`
A helper function that checks a given host string to see if it contains a range pattern descibed in the docstring above.

Returns a True match object if the given string contains a pattern, else None.

```
>>> detect_range('hostname4')

>>> detect_range('192.[168].2.1')

>>> detect_range('foo[bar]baz')

>>> detect_range('[@$\n*#:*\t$&@]\]')

>>> detect_range('host_name[1:8]')  # no underscores in hostnames

>>> detect_range('host[1:8]').groups()
('host', '1', '8', '')

>>> detect_range('10.0.0.1[01:08]').groups()
('10.0.0.1', '01', '08', '')

>>> detect_range('host[1:8].with-dash.com').groups()
('host', '1', '8', '.with-dash.com')

>>> detect_range('www-[0001:8000].g.cn').groups()
('www-', '0001', '8000', '.g.cn')
```

`pave.targets.expand_hostname_range(matchobj)`
A helper function that expands a given string that contains a pattern specified in top docstring, and returns a list that consists of the expanded version.

The '[' and ']' characters are used to maintain the pseudo-code appearance. They are replaced in this function with '|' to ease string splitting.

Example:

```
Passing a match obj allows a second execution of the regex to be
skipped:
match = detect_range(hostspec)
if match:
    expand_hostname_range(match)
```

References: http://ansible.github.com/patterns.html#hosts-and-groups

```
>>> expand_hostname_range(detect_range('host[1:8]'))
['host1', 'host2', 'host3', 'host4', 'host5', 'host6', 'host7', 'host8']

>>> expand_hostname_range(detect_range('host[0-2]'))
['host0', 'host1', 'host2']

>>> expand_hostname_range(detect_range('host[0:3]'))
['host0', 'host1', 'host2', 'host3']
```

```
>>> expand_hostname_range(detect_range('host[:2]'))
['host0', 'host1', 'host2']

>>> expand_hostname_range(detect_range('www[01:04].g.cn'))
['www01.g.cn', 'www02.g.cn', 'www03.g.cn', 'www04.g.cn']

>>> expand_hostname_range(detect_range('[0001:0002]node.g.cn'))
['0001node.g.cn', '0002node.g.cn']
```

`utils` Module (C) 2012-2014 Mike Miller. License: GNU GPLv3+

A grab bag of assorted stuff.

class `pave.utils.`**`AttrDict`**

Bases: `dict`

Dict that acts like an object.

`copy`()

class `pave.utils.`**`PaveSafeLoader`**(*stream*)

Bases: `yaml.loader.SafeLoader`

Disables the use of "%" chars as directive characters, so we don't have to quote so much.

`check_directive`()

`check_plain`()

class `pave.utils.`**`VarTemplate`**(*template*)

Bases: `string.Template`

`delimiter` = '%'

`idpattern` = '[_\\w][_\\w0-9]*'

`pattern` = <_sre.SRE_Pattern object at 0x2fcbf40>

`pave.utils.`**`ask_crypt`**(*tries=3*)

Asks for a password and returns crypt'd.

`pave.utils.`**`backup`**(*fname*, *cmds*, *context*)

Make a .orig backup of files we are going to modify.

`pave.utils.`**`bold`**(*text*)

Add a bold attribute to given text.

`pave.utils.`**`check_list_conditionals`**(*item*, *context*, *module=None*, *replace=True*, *force=False*)

Check if item is a dictionary and if so eval conditionals. Helpful For list-based modules. Returns None if item should be skipped. Replace - replace object with its do/then clause.

`pave.utils.`**`contains`**(*filename*, *text*, *exact=False*, *use_sudo=False*, *escape=False*, *fixed=False*)

Copied from fabric–shows output and escape defaults to False. I may remove this and reimplement.

`pave.utils.`**`crypt`**(*passwd*)

Crypt a password. See `man crypt` for details.

`pave.utils.`**`donepath`**(*cmds*, *base*, *itemname*, *item*)

Create and return a path to a "done" file.

`pave.utils.`**`eval_conditionals`**(*sectiondict*, *context*)

If multiple tests, must all be true; boolean AND.

`pave.utils.`**`find_password`**(*directive*, *encrypt=None*)

pave.utils.**fmtcmd**(*cmd*, **args*, ***options*)
format a command line given both arguments and options.

pave.utils.**get_cursor_pos**()
Return the current column number of the terminal cursor. Used to figure out if we need to print an extra newline.

pave.utils.**getpwd**(*prompt*, *tries=3*)
Request a password until two copies match.

pave.utils.**markdone**(*cmds*, *tdir*, *path*, *context*)
Mark a task as "done".

pave.utils.**md5**(*data*)
Given a piece of data, return its md5 hash.

pave.utils.**q**(*fname*)
Quote strings (e.g. filenames) if needed.

pave.utils.**runcmd**(*cmdline*, **args*, ***options*)
Run a command with fabric, using arguments to decide on sudo or not.

pave.utils.**safe_eval**(*text*, ***namespace*)
Disables builtins and provides only given variables. http://lybniz2.sourceforge.net/safeeval.html

pave.utils.**sequence**(*value*, **toappend*)
Given an object, check if it is a sequence. If not, add it to one. Return the sequence.

pave.utils.**trunc**(*msg*, *length*)
Truncate a string with trailing ellipsis.

pave.utils.**wait_key**()
Waits for a keypress at the console.

Subpackages

lib Package

cleanup Module Good hygene is important.

folders: Deletion of files under /tmp, /var/tmp, & /home/ only.

locales-except: *"Unless you're working at the UN and administer a central server for all member states, it is difficult to conceive why you would need a system where all of these locales are installed."* Similar to a one-time Debian locale-purge.

packages: Run appropriate package cleanup command(s).

Example:

```
- cleanup:
    packages: True
    locales-except: en en_US
    folders:  # use with caution
        - /tmp/stuff/*
        - /tmp/stuff2
```

pave.lib.cleanup.**handle**(*section*, *cmds*, *context*)

`configure` Module Configure the remote system. Also handles templating, see below.

`render:` Renders a template to a local file, which is then uploaded to host.

> **`engine:`** The templating language, such as python (printf|Template|format), or the executable name of an external package, e.g: `jinja2`

`render-remote:` Simple remote "templating" with sed.

Example:

```
- configure:            # str || list accepted on all sub-tasks below:
    create:  # file or folder/
        - foo.conf
        - /tmp/project/

    append:  # text remotefname
        Answer=42 %rc

    comment: # [comment char] from-regex remote-fname
        - ' "insecure on" %rcfile '

    replace: # from-regex to remote-fname
        - ^foo$ bar %rcfile
        # note that because of yaml syntax and splitting, two layers
        # of quoting or blocks may be needed when backslashes used:
        - >
            '^foo\s*$' bar %rcfile

    render: # engine [extra-vars] local-template remote-fname
        - printf swappiness=50 examples/printf_template.txt %rc2
        - format FOO=BAR       examples/format_template.txt %rc3

    render-remote: # [vars] remote-source remote-dest
        - INSTANCE_NUM=1 %sitecfg /etc/init/%svcname.conf

    update: # copy a file into place if newer, SRC DEST
        - ~/%sitename/cfg/celeryd.conf /etc/init/celeryd.conf

    access: # mode user group remote-fname # use "" for empty
        - 0640 %user %user %rc2
```

`pave.lib.configure.`**`handle`**(*section*, *cmds*, *context*)

`pave.lib.configure.`**`render`**(*engine*, *text*, *tvars*)

> Render templates.

`pave.lib.configure.`**`req`**(*key*)

`django` Module Django helper module. *Captain Obvious recommends Django be installed on the target beforehand, see the* **'packages'_** *section.*

`loaddata:` Load fixture files from given path, glob ok.

`manage:` Execute a management command every time this task group is run.

`manage-once:` Once and never again.

Example:

```
- django:
    workdir: ~%sitename/%sitename
    loaddata: "fixtures/*.json"
    manage-once: update_index
    syncdb: True
```

pave.lib.django.**handle**(*section*, *cmds*, *context*)
Run the system package manager, and any others given.

groups Module Create operating system user groups.

Example:

```
- groups:           # str || list || conditional
    - appgroup
    - "dev:1100"    # how to set gid

    - if-exec: test -e /dev/cdrom
      do: cdrom
```

pave.lib.groups.**handle**(*section*, *cmds*, *context*)

kernel Module Manage kernel parameters. Saves to memory and disk.

Example:

```
- kernel:           # str || list || conditional
    - vm.swappiness = 10
```

pave.lib.kernel.**handle**(*section*, *cmds*, *context*)

keygen Module Manage local/remote ssh keys and their generation.

remote: Generates a key at the remote host under the given username.

copy-id: Copies local keys to the remote host.

Values may be a single or list of Boolean (True copies the public key of the local user to the authorized keys file of the remote user), or string of two arguments containing the local file (.pub appended automatically) and the remote user.

Example:

```
- keygen:
    remote: "%sitename"             # quotes not required but
    copy-id:                        # fixes syntax highlighting
        - True                          # default src dest
        - "%local_path.pub %user"       # specified
```

pave.lib.keygen.**check_ssh_config**()
Check for config locally.

pave.lib.keygen.**handle**(*section*, *cmds*, *context*)

pave.lib.keygen.**keygen**(*type='rsa'*, *passphrase=''*, ***context*)
Generate ssh keys locally.

packages Module Manage software packages. Packages names for `install`, `remove`, etc. are listed as whitespace-delimited strings.

expiration: The age in days after which the OS package index is considered to have expired, requiring action.

To be a good neighbor, the default, 0 (= 24 hours) means packages won't be updated unless it has been a full day since the last update. A value of -1 forces action.

inventory: Make a list of installed packages to compare against.

update/upgrade: Update the system's package index, upgrade packages to latest release.

upgrade-full: Upgrade packages, even if it causes obsolete packages to be removed, or new packages to be installed, for example new kernel versions. e.g. Debian's "dist-upgrade".

Example:

```
- packages:
    if-platform: Debian
    install:
        sysvbanner python-pip
    # expiration: -1    # force attempt
```

`pave.lib.packages.`**`cleanup`**(*cmds*, *context*)
Clean any mess left over by package installation.

`pave.lib.packages.`**`handle`**(*section*, *cmds*, *context*)
Run the system package manager, and any others given.

postgres Module Manage postgres. *Captain Obvious here to chime-in that postgres must be installed on remote target beforehand, see* **'packages'_**.

link current: Create a symbolic link from the last sorted item in `/etc/postgresql/` to `/etc/postgresql/curr`.

replace, access: See the corresponding directive under configure. These are for convenience to group postgres-related configuration together.

user, db, grant, sql: `postgresql-client-common` (or distro equiv.) must be installed on remote target beforehand. Run the appropriate commands to create objects or configure the database. String or list of strings may be given.

Example:

```
- postgres:
    link current: False
    replace:
        - '#\s*password_enc password_enc %pgcfg/postgresql.conf'

    su-password:
        use pgsu                                # see passwords:
        # md5 acbd18db4cc2f85cedef654fccc4a4d8  # how to hard-code

    db: # quotes not required below, but fixes highlighting. :/
        - name: "%sitename"
    user:
        - name: "%sitename"
    grant:
        # privs   on_obj    named         to_user
        all       database  %sitename     %sitename
```

```
sql: # str|list lines of sql, first token should be dbname:
    template1 select datname from pg_database
              where datname = '%sitename';
```

`pave.lib.postgres.`**`handle`**(*section*, *cmds*, *context*)

`run` Module Executes a list of commands using the remote shell. Each command may be:

- **A string** A command that runs under the default user.
- **A mapping with one or more of the following members:**

  `do:` `CMD_STRING`
  The command to run.

  `user:` `USERNAME`
  Run as another user.
  May be `sudo` (root), or a valid remote username.

  `title:` `TITLE`
  to document or hide long, ugly commands.

  `workdir:` `PATH` set working directory.

  `local:` `<bool>` runs it locally instead of remotely.

See *Conditional Execution* for more details on `if:` and `if-exec:`.

Note: While *not recommended* to set passwords via command-line tools for various reasons, tasks makes an attempt to hide them (vars starting with "`pwd_`") from standard logs. Passwords will still be displayed in debug logs (and perhaps shell history) upon command-line execution.

Example:

```
- banner "Hello!"     # world

# signal a change event and prevent errors from halting execution:
- touch %filename && echo CHANGED || true
- rm /etc/nginx/sites-enabled/default && echo CHANGED || echo SKIPPED

# Even poor-er man's conditionals:
- " [ '%mode' == 'qa' ] && echo 'QA install' "

# more robust task definition:
- run:
    - title: Nothing to see here...
      if-exec: test -f /foo
      do: echo "This is running under $USER (appuser)"
      user: appuser
```

`pave.lib.run.`**`handle`**(*section*, *cmds*, *context*, *chatty=True*)

`pave.lib.run.`**`make_title`**(*title*, *command*, *pwds*)

`scp` Module Copy files to and fro. Given a source and destination, copy files.

Notes:

- By default a put (upload) is performed, but the direction can be specified as the first parameter.
- Paths with whitespace must be quoted.

- Destinations folders should end in a / character for efficiency, as doing so skips the need for several filesystem tests.
- If sudo is enabled, relative paths/tildes will be interpreted (and files owned) by the root user. Therefore it is recommended that absolute remote paths be given.
- Does not currently handle wildcards.
- While this module is similar to using the scp command-line program it may not work identically.

Example:

```
- scp:
    - SRC DEST          # put
    - SRC DEST MODE     # put, w/ Unix octal mode or "same"
    - get SRC DEST

    # example conditional
    - if: inst_ssl == True
      do: put bundle.crt "%sitefldr/ssl/bundle.crt"
```

Additional details:

- fabric.operations.get
- fabric.operations.put

`pave.lib.scp.`**`check`**(*v*)
validate command lines. Todo: should check if on disk.

`pave.lib.scp.`**`handle`**(*section*, *cmds*, *context*)

`pave.lib.scp.`**`tilde_helper`**(*path1*, *path2*)

`services` Module Manage system daemons/services. Values are whitespace separated strings.

`restart:`, `restart-fail:` restart will start the service if it was stopped, restart-fail will fail with an error if it was not running originally.

Note: Some daemon startup scripts are not written well and quit when the ssh session ends. To prevent this from occurring, prepend the service name with "`nohup:`" as shown below.

Example:

```
- services:
    # the follwing two are temporary until reboot:
    running: hostname hwclock
    stopped: whoopsie
    # permanently; /packages/remove more reliable on ubuntu
    disabled: whoopsie
    restart: nohup:memcached
```

`pave.lib.services.`**`check_nohup`**(*svcname*)
look for a nohup:SVCNAME construct.

`pave.lib.services.`**`handle`**(*section*, *cmds*, *context*)

`pave.lib.services.`**`req`**(*key*)

users Module Create operating system users.

Example:

```
- users:
    - name: appuser                 # required
      groups: appgroup www-data     # whitespace separated groups
      create-home: True
      shell: /bin/bash
      password: use appuser         # see passwords: section
```

`pave.lib.users.handle(section, cmds, context)`

vcs Module Deploy code with this version control helper. $VCS support must be installed beforehand, see the **'packages'_** section.

If the destination does not exist, an initial checkout is done. If it does exist, the repository is updated to the latest version.

Formats:

```
- VCS_TYPE REPO_URL DEST EXTRA          # simple or
- do: VCS_TYPE REPO_URL DEST EXTRA      # robust
```

VCS_TYPE: Version Control System, one of `svn`, `hg`, or `git`.

REPO_URL: URL to repository.

DEST: Destination folder.

EXTRA: Extra command line parameters.

do: What to do.

input: Useful to automate interactive prompts.

user:: Execute vcs under this remote username.

Example:

```
- vcs:
    - do: hg %repo_base/appname ~%user/appname
      input: p\nn                           # interactive answer
      user: %user
```

`pave.lib.vcs.handle(section, cmds, context)`

`pave.lib.vcs.in_vclist(v)`
validate

`pave.lib.vcs.min_two_tokens(v)`
validate

- Bitbucket Repo

FAQ - (Not So) Frequently Asked Questions

I don't know any bash or Python, can I still use $language with pave? Yes, (although learning some `sh` or `bash` is recommended), it is easy to use another language with pave (like it is done with ansible).

The basic idea is this... that you write a script, upload it to your host, and then run it there:

```
tasks:
    - packages:
        install: other-runtime          # if need-be, or

    - scp: put setup_script.pl /tmp/    # to use perl
    - /tmp/setup_script.pl --kapow
```

How can I make pavefile authoring easier?

- Use a professional text editor which can highlight yaml. If you are just starting out you might try Geany on Linux, Notepad++ on Windows, or TextMate2 on OSX. Make sure to set it to indent with spaces, not tabs.
- Make judicious use of pave's command-line parameters. See the *Command-line Reference* on usage.
  - Turn on the debug logging option `-V` to see how the yaml parser has interpreted the given text and many other details.
  - Likewise, use the test option `--test` to parse and validate the file while skipping execution.
  - `-i/--interactive` is good for stepping through execution of tasks.
  - `-s/--select` is great for testing each task as you write it:

    ```
    pave -s -1 -V
    ```

    Yes, that second parameter is a negative number one, signifying the last task. This is really helpful–you won't have to wait while multiple previous tasks are checked again.
  - `-o/--option` or `-v/--var` options can be used to set values, variables in the pavefile:

    ```
    pave -o /main/inspect sh
    ```

What about troubleshooting?

- See answer above.
- In addition to the verbose command-line parameter, debug text and exception tracebacks, etc, for the last five runs are typically recorded in:

  ```
  ~/.cache/pave/{localhost,targets}
  ```

  This location may be different on your machine, or if you are using the `XDG_CACHE_HOME` environment variable.
- Regexes: `grep -E/egrep` is used extensively on the remote-end when searching. For whatever reason it doesn't support `\d`, so don't waste hours debugging like I have. :/

Why the `%` variable expansion syntax? ————————————————~

Originally pave used Python's newer, niftier string format syntax `{name}` rather than the older printf-style substitution `%(name)s` as it is more readable and sophisticated. Unfortunately though, after writing a few pavefiles it became clear it was too much trouble.

Why? First, it conflicts with yaml mapping syntax. Not the end of the world you might think, it can be quoted... ok. Ooh, it conflicts with bash at times too, braces are used for many substitutions... uggh, ok. Went to add `{{ Jinja }}` support and kaboom!, the last anvil to break the camel's back. To escape braces in string.format syntax you need to double them, just what Jinja uses.

Attempting to use thse features together reminds me of the old driver's-ed film "Red Asphalt". Time to admit defeat on that front.

Printf-syntax on the other hand collides much less often. The `%` character is used for an esoteric feature in YAML that pave disables. I've found but a few scatted shell commands like find and stat that use them. Escaping is easy though, just double it, `%%`, no matching front and back (with different symbols) to worry about.

Finally pave has moved to even simpler shell-style syntax as implemented by Python's string.Template. To avoid collisions with Unix shell `$variables` however, we've kept percent as the expansion character.

There may be other exceptions, but one I can think of is Windows and so I've left a `-b` command line option to use brace format syntax instead.

What licensing is available? pave is licensed as GPLv3+.

A commercial-friendly license may be available for a fee. Support/fixes/features might be offered for a fee as well, regardless of license.

What are your thoughts on the subject?

1.2 Pavefile Reference

Syntax and layout of pave's configuration file is described here. For information on the various sections and task module library, skip ahead to the next chapter, *Pavefile Reference Part II*.

1.2.1 Syntax

At the most basic level, a pavefile is expected to be encoded in UTF-8 format and named `./pave.yml` by default.

Pave utilizes the user-friendly, human-optimized YAML file format with a few extensions. If you haven't encountered it yet, you may wish you had found it earlier. There's not much to learn to get started, less than other formats. No verbose "angle-bracket" tax, less punctuation and synchronization necessary than XML or JSON, and no harsh limitations like .ini files.

YAML Resources

When in doubt I wander over to the sites below:

- http://en.wikipedia.org/wiki/Yaml#Language_elements
- http://pyyaml.org/wiki/PyYAMLDocumentation#YAMLsyntax

General Layout

Comments start with the "#" character, and indentation must be done *with spaces*. The form `name: value` represents a dictionary mapping, often called a hash, or associative array. A value is retrieved by its name. The `- item` form represents a list, an expandable array indexed by number.

A basic file will look like this, with *section* names at the far left:

```
vars:
    name: value

main:                               # starts a map of a map, to
    name1:  value1                  # strings, or a
    name2:  True                    # boolean, or ...
```

```
tasks:                                  # a map to a list of
    - "list item 2"                     # quoted-strings, or
    - 42                                # numbers, or ...
    - name3:  |                         # text block, that preserves newlines
            ls -l /foo
            echo %name
    - name4:  >                         # text block, wrapped. May omit > char.
        This form is good for long command lines.
```

Variables

... are discussed in depth on the next page (at *vars*) but the short story is that they are created in the `vars:` section (see above) and can be expanded in string values with `%varname` style syntax.

Quoting

- If a declaration or command-line string contains positional arguments and an argument needs to contain whitespace, then it should be quoted just like it would in a shell. Also:
- Use double quotes and C/Python-style escape sequences to denote control (or difficult to type) characters in strings:

  ```
  "\n", "\x##" and "\u####"
  ```

- Conversely, use the "stronger" single quotes when a string has backslashes that need to be sent literally, such as definition of regular expressions. If not you'll get errors, such as "`s is not a valid escape sequence.`":

  ```
  - '#\s*password_encryption password_encryption'
  ```

- Quoting a quoted string (with the other kind of quotes) may need to be done if the string starts with a quote but ends early, :/ e.g.:

  ```
  - ' "-s /tmp/memcached.socket" /etc/memcached.conf '
  ```

Blocks, Braces

- If you'd like to avoid one level of the quoting mess above, block styles (see yaml docs above) are the way to go. They are useful for longer text (or command-lines) with special characters or regexes.
- A block is signified by use of an optional block character, > or `|`, and newline followed by the indented body. To enter a block that will be wrapped (newlines converted to spaces) you may omit the character, or add it explicitly. This one is good for long command-lines:

  ```
  important:                      # Add > char here to explicitly wrap.
      sudo foo install
      https://github.com/developer/repository/tarball/master
  ```

  When newlines need to be preserved, use the `|` char, which is helpful when writing haiku I suppose:

  ```
  status: |
      I wakey wakey.
      Production release today.
      No breaky breaky.
  ```

- If using brace variable expansion (python string.format), use quotes for strings when they begin with a brace character:

```
- '{user}'
```

1.2.2 Conditional Execution

Poor Man's Conditionals

If your needs in the area are modest, or you'd prefer to avoid external dependencies, you'll likely be able to squeak by with using one of the alternatives below. Separate pavefiles with includes are an option as well.

There are a few types of simple conditionals supported in the pavefile, executed either locally or remotely. They are supported by all standard modules. Add these to items under the `tasks:` section as needed.

- *Local* conditionals:
  - `if: VARNAME COMPARISON VALUE` This clause will compare a variable from the `vars:` section to a value locally, and if true, execution of the attached item will continue. E.g.:

    ```
    - scp:
        - if: inst_ssl == True
          do: # ...
    ```

    This is the only place where the variable name needn't be dereferenced with the `%` character. If there were whitespace in an unquoted variable for example, an error would occur, however this convenience-form avoids it and is easier to type.

    Comparisons are currently done as strings only, meaning 'True' equals a boolean True, while supported comparisons are == and `!=`.
- *Remote* conditionals:
  - `if-exec: EXECUTABLE_STATEMENT` Execute this statement to do reconnaissance at the *remote* end before primary task execution. If the statement returns with a exit status code of 0 (meaning true), the primary statement is run also. Conversely, a return code of >= 1 (false) will skip the task. Change/Skip events are handled automatically.

    ```
    - run:
        - if-exec: test ! -f media/js/site.js.gz
          do: # ...
    ```
- *Manual* remote conditionals
  - Similar to `if-exec:` above, use `test` (aka `[ ... ]`), `grep` or other command-line program before the primary statement. See the Events section below on manual event dispatch.
- *Remote* conditional handling II:
  - `if-platform: PLATFORM` `if-platform: KEY COMPARISON VALUE` Make decisions based on what we're talking to:

    ```
    - packages:
        if-platform: Ubuntu
        install: sysvbanner

    - packages:
        if-platform: arch[0] == 64bit
        install: ia32-libs
    ```

The available platform keys with example values are given below. The single token PLATFORM format is usually a shortcut to a simplified version of `linux_dist[0]` on Linux:

```
arch:        ['64bit','ELF']
encoding:    'UTF-8'
linux_dist:  ['Ubuntu','13.10','saucy']
machine:     'x86_64'
node:        'darkstar.home.com'
os_release:  '3.11.0-15-generic'
proc:        'x86_64'
py_vers:     '2.7.5+'
py_vert:     ['2','7','5+']
system:      'Linux'
version:     '#23-Ubuntu SMP Mon Dec 9 18:17:04 UTC 2013'
```

Additional example values:

```
mac_vers:    ['10.9.1', ['', '', ''], 'x86_64']  # system: 'Darwin'
win_vers:    ['XP', '5.1.2600', 'SP3', 'Uniprocessor Free']
```

This information may be found under `~/.cache/pave/platform.json` on the client and each target. It is also logged and printed to the console at debug level when pave is run.

Templating

For more robust conditional execution, there is the option of using a templating system to build your pavefile on the fly. Remote execution and platform checks are not available with this method, but continue with those previously mentioned. Currently jinja2 is supported.

Usage:

0. Install it if it has not been already, e.g.:

   ```
   sudo apt-get install python-jinja2       # yum, brew, pip, etc.
   ```

1. Create a pavefile with variables (`vars:` section) at the top. Do not include jinja markup in this section. Assign variables (with standard expansions if desired).
2. Add jinja markup to the rest of the document as needed, saving with a .j2 file extension.
3. Run pave with the `-f/--filename` command-line parameter, using a .j2 extension, e.g.:

   pave -f paradise.j2

1.2.3 Events

Events are simple signals that are output during task execution. Modules or commands use them to tell pave what has occurred. Events are generated and handled *automatically* in all modules in the included library, with one exception. When bare strings (commands) are children of `tasks:`, `run:` events must be dispatched manually (see below).

Three types of events are tracked by pave.

Change or Skip

When a change has been made or pre-task test is passed (meaning a change needs to be made) a “change event” is generated.

A skip event is given instead, if a task decides the change is not currently necessary.

Error

Error events are a third kind of event that may occur. These will halt execution on the affected host if the `warn-only:` option is disabled in the `main:` section.

Change and error events are counted and returned to the user at the end of a completed task list.

Manual Dispatch from Command-Strings

- Typically a successful command returns a `0` status code, failure non-zero.
- To harness these follow a command-string with,
  - `" && echo CHANGED"`, which will register a change event,
  - `" || echo SKIPPED"`, a skip event, and/or
  - `" || true "`, to prevent an unimportant error from halting execution.

  Here's an example that combines a manual conditional with manual event dispatch:

  ```
  - "[ '%mode' == 'dev' ] && wget %url && echo CHANGED || echo SKIPPED"
  ```

  A multi-line statement can be sent as well (assuming the remote shell supports it) and you have correctly marked it as a yaml text block (see | or > chars in General Layout)
- CHANGED wins over SKIPPED if the both are output somehow.

1.2.4 Using Custom Modules

A custom or third-party module to use in the `tasks:` section should follow the interface and format of the included library modules, as discussed in the Developer Guide.

To use such a module, you may move it to a location in python's `sys.path` (the current directory "." may already be there). For convenience the folder `~/.config/pave/lib/` is added to the path automatically, and as such is a good place to put custom modules.

Alternatively, you can modify the path to contain additional folders per project. This can be done by adding the path to `/main/sys.path` option in the pavefile, or setting the PYTHONPATH environment variable.

1.2.5 Disabling Sections/Tasks

By convention, all data under `disabled:` is ignored—A place to temporarily bypass tasks without having to futz around commenting them.

Task selection, using the `-s/--select` command-line parameter is also available as an alternative. See the Command-line Reference.

1.2.6 Full Examples

- https://bitbucket.org/mixmastamyk/pave/src/default/examples/

Next, kindly continue on to the next chapter, *Pavefile Reference Part II*, for details on the standard sections and task module library.

1.3 Pavefile Reference Part II

Sections and Task Library available to pave's configuration file are described here. For syntax, conditionals, and general information, see the previous chapter *Pavefile Reference*.

1.3.1 Sections

A section is a top-level portion of a pavefile, which is organized hierarchically by indentation.

Names of sections are given by the labels at the far left in the first column. Names and the data (typically indented) below it are considered the complete section. Therefore, a name in the first column ends the preceding section and starts a new one:

```
# end previous section
foo:            # The "foo" section
    - bar
    - baz
# start next
```

The standard sections described next are handled by pave itself. Tasks implemented by library modules are described afterwards under Tasks Library and are placed in sub-sections under `tasks`.

---

vars

A fine place to define variables, `name:  value` style.

When pave encounters `name` constructs in a string value it will replace them with a value given in this section. This expansion syntax is described in depth here, with the exception that pave uses the **percent** character to avoid clashing with shell scripting.

Pavefile strings support %-based substitutions, using the following rules:

- Expansion can be avoided by doubling the `%` character.
- `%identifier` names a substitution placeholder matching a mapping key of "identifier", which must spell a pave variable. The name must be a letter or underscore, followed by a letter, digit, or underscore. The first non-identifier character after the % character terminates this placeholder specification.
- `%{identifier}` is equivalent to `%identifier`. It is required when valid identifier characters (including underscores) follow the placeholder but are not part of the placeholder, such as `%{noun}ification` or `%{username}_backup.zip`.

Notes:

- Passwords are added to the pavefile vars with the prefix "pwd_" e.g., `pwd_appuser` to support variable interpolation and avoid conflicting with existing variable names.
- Variables in the var section will be expanded against themselves just once, in order to create vars that contain fragments of other vars—useful for building paths and such.

The order of variable expansion is as follows:

1. Optional: A templating engine, such as Jinja.
2. string.Template expansion of variables from the vars section performed locally by pave.
3. Bash or other chosen shell will expand and run command strings in the the remote execution context.

Example:

```
vars:    # I put this at the top of the file for easy-access, baby.
    sitename: fooBar
    repo: "http://bitbucket.org/username/%sitename"  # existing var

tasks:     # usage, quotes not needed:
    - "echo %sitename rules"
```

Schema:

dict of *boolean* || *float* || *integer* || *list* || *string*

main

Where options to pave are specified. *Required.*

bak-files: Copy files to be modified to NAME.orig beforehand.

env: A place to directly set fabric's environment attributes. Several items, such as hostnames, will be clobbered if you set them here, since pave manages them.

include: Include another file into this pavefile. Multiple entries accepted. Note that included sections will overwrite identical sections in the parent. Useful for separating lengthy orthogonal sections, e.g. `target-groups:`

inspect: Inspect the target to determine platform details. Values:

- `True` - Use the python inspector, if it fails try the shell script next. (Default)
- `False` - Disable inspection, not currently useful.
- `py` - Use the python inspector script only.
- `sh` - Use the sh inspector script only.

jobs: Number of "steamrollers." Each task-list is run in parallel (per host) with the given pool size.

log-to: Specify logging/data folder.

targets: Specify hosts (or groups of hosts) to target. Whitespace-delimited string or list. See the `target-groups:` section below for group definition.

user: Change the login user, defaults to the current one.

version: Version of pave this file was written against, enabling time-travel. Not yet implemented.

warn-only: Continue thru errors with a warning only.

Example:

```
main:
    user: ubuntu
    sudo: True
    targets:
        - 192.168.2.1     # a list or string of hosts, groups to greet.
        - www[1-3]
```

Schema:

dict of

bak-files: *boolean* default: `True`

env: *dict* default: `{}`

include: *list of string*

string default: []

inspect: *boolean || string* default: True

jobs: 100 True 1 True *integer* default: 1

log-to: *string* default: ''

sudo: *boolean* default: True

sudo-user: *string*

sys.path: *list of string*

string

targets: *list of string*

string default: ''

user: *string* default: 'CURRENT'

version: *string* default: 'CURRENT'

warn-only: *boolean* default: False

passwords

Set or query passwords for later use.

- `ask NAME PROMPT` Query the user for this password, named NAME, given PROMPT, prior to execution. As an "escape hatch," two empty passwords given at a prompt will exit the program.

  Note: do not add a colon character to the end of the PROMPT value. One will be added automatically.

- `set NAME PASSWD` Hard-code something temporarily (not recommended). This can greatly speed up authoring of pavefiles—But don't check these in! ;) See `/main/include` to offload this section to another file instead. Using another file (with a standard name) and adding it to your vcs ignore file will greatly decrease the chances of a password checkin.

Note: 99% of the time when starting out you'll want to give "password" (no quotes) as the value for NAME. This is the name fabric uses for the ssh and/or sudo password.

How to use gathered passwords in other sections:

- `use NAME` Will plug-in the named password into another section's options. See `users:` or `postgres:` for examples.
- `wget --password="%pwd_websvc"` Use variable expansion to render the password into a command-line task. The variable name should be prepended with `pwd_` to avoid collisions.

Example:

```
passwords:
    - ask password  ssh/sudo Password            # fabric default
    - ask pgsu      Postgres SuperUser Password
    - set appuser   "hello world"                # set not recommended
```

Schema:

list of *string*

target-groups

A mapping of group names assigned to lists of targeted hosts.

Example:

```
target-groups:
    dbservers: [host1, host2, host3]     # compact yaml list syntax
    webservers:
        - www[1-3]
```

This is a good place to mention hostname expansion. Leading `0`'s specify padding:

`host[1-3] --> [host1, host2, host3]` `host[01-03] --> [host01, host02, host03]`

Schema:

dict of

string: *list of string*

string

fab-tasks

A list of tasks to run directly from an existing `fabfile.py`.

These "fab-tasks" are run before standard task execution and do not interact with standard tasks. However, the fabric `env` (ironment) variables that have been set by pave are passed forward to these fab-tasks, allowing you use pave's target groups and misc settings to drive fabric as if you were using the `fab` command.

`env.fabfile:` Set the path of a fabfile here if it is not in the current or parent folders. Relative paths and tilde expansion are supported.

Arguments and keyword values are passed in using the same command-line syntax fab uses. For example:

```
main:
    env:
        fabfile: ../fabric/fabfile.py  # optional

fab-tasks:
    - task1
    - task2: arg1
    - task2: arg2,kwarg1=kwval1
```

Schema:

list of *string dict*

tasks

A listing of tasks to be processed in order.

Each task in the list should be a bare string to be executed by the `run` module, or a mapping of a module name to the data it requires. The second form is called a "task group" at times as it may contain numerous sub-tasks.

See the Tasks Library below and *Conditional Execution* for more details.

Example:

```
tasks:
# tasks could be wide enough that you may want to skip the optional
# yaml list indent.

- banner "Hello!"     # world - strings get passed to the run module

# use the users module explicitly, a "task group"
- users:
    - name: appuser
```

Schema:

list of *string dict of*

> **modname:** moddata

1.3.2 Tasks Library

Each mapping listed under tasks corresponds to a module in the `site-packages/pave/lib` folder or a third-party module found on the `sys.path`. See the Developer Guide for details.

cleanup

Good hygene is important.

folders: Deletion of files under `/tmp`, `/var/tmp`, & `/home/` only.

locales-except: *"Unless you're working at the UN and administer a central server for all member states, it is difficult to conceive why you would need a system where all of these locales are installed."* Similar to a one-time Debian locale-purge.

packages: Run appropriate package cleanup command(s).

Example:

```
- cleanup:
    packages: True
    locales-except: en en_US
    folders:  # use with caution
        - /tmp/stuff/*
        - /tmp/stuff2
```

Schema:

dict of

- **do:** *string*
- **folders:**
  - ***list of***
    - *string*
    - default: `[]`
- `if`:
- **if-exec:** *string*
- **if-platform:** *string*

locales-except: *string*

packages: *boolean* default: `True`

configure

Configure the remote system. Also handles templating, see below.

render: Renders a template to a local file, which is then uploaded to host.

engine: The templating language, such as python (printf|Template|format), or the executable name of an external package, e.g: `jinja2`

render-remote: Simple remote "templating" with sed.

Example:

```
- configure:            # str || list accepted on all sub-tasks below:
    create:   # file or folder/
        - foo.conf
        - /tmp/project/

    append:   # text remotefname
        Answer=42 %rc

    comment: # [comment char] from-regex remote-fname
        - ' "insecure on" %rcfile '

    replace: # from-regex to remote-fname
        - ^foo$ bar %rcfile
        # note that because of yaml syntax and splitting, two layers
        # of quoting or blocks may be needed when backslashes used:
        - >
            '^foo\s*$' bar %rcfile

    render: # engine [extra-vars] local-template remote-fname
        - printf swappiness=50 examples/printf_template.txt %rc2
        - format FOO=BAR       examples/format_template.txt %rc3

    render-remote: # [vars] remote-source remote-dest
        - INSTANCE_NUM=1 %sitecfg /etc/init/%svcname.conf

    update: # copy a file into place if newer, SRC DEST
        - ~/%sitename/cfg/celeryd.conf /etc/init/celeryd.conf

    access: # mode user group remote-fname # use "" for empty
        - 0640 %user %user %rc2
```

Schema:

dict of

access: *list of string*

string default: `[]`

append: *list of string*

string default: `[]`

comment: *list of string*

string default: `[]`

`create:` *list of string*

string default: `[]`

`do:` *string*

`if:`

`if-exec:` *string*

`if-platform:` *string*

`render:` *list of string*

string default: `[]`

`render-remote:` *list of string*

string default: `[]`

`replace:` *list of string*

string default: `[]`

`update:` *list of string*

string default: `[]`

django

Django helper module. *Captain Obvious recommends Django be installed on the target beforehand, see the* packages *section.*

`loaddata:` Load fixture files from given path, glob ok.

`manage:` Execute a management command every time this task group is run.

`manage-once:` Once and never again.

Example:

```
- django:
    workdir: ~%sitename/%sitename
    loaddata: "fixtures/*.json"
    manage-once: update_index
    syncdb: True
```

Schema:

dict of

`do:` *string*

`if:`

`if-exec:` *string*

`if-platform:` *string*

`loaddata:` *string* default: `''`

`manage:` *list of string*

string default: `[]`

manage-once: *list of string*

string default: []

migrate-init: *boolean* default: True

syncdb: *boolean* default: True

user: *string* default: None

workdir: *string* required

groups

Create operating system user groups.

Example:

```
- groups:            # str || list || conditional
    - appgroup
    - "dev:1100"     # how to set gid

    - if-exec: test -e /dev/cdrom
      do: cdrom
```

Schema:

list of string

dict of

do: *string*

if:

if-exec: *string*

if-platform: *string*

kernel

Manage kernel parameters. Saves to memory and disk.

Example:

```
- kernel:            # str || list || conditional
    - vm.swappiness = 10
```

Schema:

list of string

dict of

do: *string*

if:

if-exec: *string*

if-platform: *string*

keygen

Manage local/remote ssh keys and their generation.

remote: Generates a key at the remote host under the given username.

copy-id: Copies local keys to the remote host.

Values may be a single or list of Boolean (True copies the public key of the local user to the authorized keys file of the remote user), or string of two arguments containing the local file (.pub appended automatically) and the remote user.

Example:

```
- keygen:
    remote: "%sitename"              # quotes not required but
    copy-id:                         # fixes syntax highlighting
        - True                           # default src dest
        - "%local_path.pub %user"        # specified
```

Schema:

dict of

copy-id:

list of string *boolean*

boolean || *string* default: `True`

do: *string*

`if`:

if-exec: *string*

if-platform: *string*

passphrase: *string* default: `''`

remote: *None* || *string* default: `None`

type: *string* default: `'rsa'`

packages

Manage software packages. Packages names for `install`, `remove`, etc. are listed as whitespace-delimited strings.

expiration: The age in days after which the OS package index is considered to have expired, requiring action.

To be a good neighbor, the default, 0 (= 24 hours) means packages won't be updated unless it has been a full day since the last update. A value of -1 forces action.

inventory: Make a list of installed packages to compare against.

update/upgrade: Update the system's package index, upgrade packages to latest release.

upgrade-full: Upgrade packages, even if it causes obsolete packages to be removed, or new packages to be installed, for example new kernel versions. e.g. Debian's "dist-upgrade".

Example:

```
- packages:
    if-platform: Debian
    install:
        sysvbanner python-pip
    # expiration: -1    # force attempt
```

Schema:

dict of

> **do:** *string*
>
> **expiration:** *integer* default: `0`
>
> `if`:
>
> **if-exec:** *string*
>
> **if-platform:** *string*
>
> **install:** *string* default: `''`
>
> **inventory:** *boolean* default: `True`
>
> **remove:** *string* default: `''`
>
> **update:** *boolean* default: `True`
>
> **upgrade:** *boolean* default: `True`
>
> **upgrade-full:** *boolean* default: `True`

postgres

Manage postgres. *Captain Obvious here to chime-in that postgres must be installed on remote target beforehand, see* packages.

link current: Create a symbolic link from the last sorted item in `/etc/postgresql/` to `/etc/postgresql/curr`.

replace, access: See the corresponding directive under configure. These are for convenience to group postgres-related configuration together.

user, db, grant, sql: `postgresql-client-common` (or distro equiv.) must be installed on remote target beforehand. Run the appropriate commands to create objects or configure the database. String or list of strings may be given.

Example:

```
- postgres:
    link current: False
    replace:
        - '#\s*password_enc password_enc %pgcfg/postgresql.conf'

    su-password:
        use pgsu                                # see passwords:
        # md5 acbd18db4cc2f85cedef654fccc4a4d8  # how to hard-code

    db: # quotes not required below, but fixes highlighting. :/
        - name: "%sitename"
    user:
        - name: "%sitename"
    grant:
```

```
        # privs   on_obj    named         to_user
        all       database  %sitename     %sitename

    sql: # str|list lines of sql, first token should be dbname:
        template1 select datname from pg_database
                  where datname = '%sitename';
```

Schema:

dict of

access: *list of string*

string default: []

db:

list of dict of

description: *None || string* default: []

encoding: *string*

locale: *string*

name: *string* required

tablespace: *string*

template: *string* default: 'template0'

do: *string*

grant: *list of string*

string default: []

if:

if-exec: *string*

if-platform: *string*

link current: *boolean* default: True

replace: *list of string*

string default: []

sql: *list of string*

string default: []

su-password: *string* default: ''

user:

list of dict of

createdb: *None || boolean* default: False

createrole: *boolean* default: False

encrypted: *boolean* default: True

login: *boolean* default: True

name: *string* required

superuser: *boolean* default: False

run

Executes a list of commands using the remote shell. Each command may be:

- **A string** A command that runs under the default user.
- **A mapping with one or more of the following members:**

  `do: CMD_STRING`
  The command to run.

  `user: USERNAME`
  Run as another user.
  May be `sudo` (root), or a valid remote username.

  `title: TITLE`
  to document or hide long, ugly commands.

  `workdir: PATH` set working directory.

  `local: <bool>` runs it locally instead of remotely.

See *Conditional Execution* for more details on `if:` and `if-exec:`.

Note: While *not recommended* to set passwords via command-line tools for various reasons, tasks makes an attempt to hide them (vars starting with "`pwd_`") from standard logs. Passwords will still be displayed in debug logs (and perhaps shell history) upon command-line execution.

Example:

```
- banner "Hello!"     # world

# signal a change event and prevent errors from halting execution:
- touch %filename && echo CHANGED || true
- rm /etc/nginx/sites-enabled/default && echo CHANGED || echo SKIPPED

# Even poor-er man's conditionals:
- " [ '%mode' == 'qa' ] && echo 'QA install' "

# more robust task definition:
- run:
    - title: Nothing to see here...
      if-exec: test -f /foo
      do: echo "This is running under $USER (appuser)"
      user: appuser
```

Schema:

list of *string dict of*

`do:` *string*

`if:`

`if-exec:` *string*

`if-platform:` *string*

`local:` *boolean*

`title:` *string*

`user:` *string*

`workdir:` *string*

scp

Copy files to and fro. Given a source and destination, copy files.

Notes:

- By default a put (upload) is performed, but the direction can be specified as the first parameter.
- Paths with whitespace must be quoted.
- Destinations folders should end in a / character for efficiency, as doing so skips the need for several filesystem tests.
- If sudo is enabled, relative paths/tildes will be interpreted (and files owned) by the root user. Therefore it is recommended that absolute remote paths be given.
- Does not currently handle wildcards.
- While this module is similar to using the scp command-line program it may not work identically.

Example:

```
- scp:
    - SRC DEST          # put
    - SRC DEST MODE     # put, w/ Unix octal mode or "same"
    - get SRC DEST

    # example conditional
    - if: inst_ssl == True
      do: put bundle.crt "%sitefldr/ssl/bundle.crt"
```

Additional details:

- fabric.operations.get
- fabric.operations.put

Schema:

list of *string dict of*

> **do:** *string*
>
> if:
>
> **if-exec:** *string*
>
> **if-platform:** *string*
>
> ***string:*** *string*

services

Manage system daemons/services. Values are whitespace separated strings.

restart:, restart-fail: restart will start the service if it was stopped, restart-fail will fail with an error if it was not running originally.

Note: Some daemon startup scripts are not written well and quit when the ssh session ends. To prevent this from occurring, prepend the service name with "`nohup:`" as shown below.

Example:

```
- services:
    # the follwing two are temporary until reboot:
    running: hostname hwclock
    stopped: whoopsie
    # permanently; /packages/remove more reliable on ubuntu
    disabled: whoopsie
    restart: nohup:memcached
```

Schema:

dict of

disabled: *string* default: `''`

do: *string*

`if`:

if-exec: *string*

if-platform: *string*

restart: *string* default: `''`

restart-fail: *string* default: `''`

running: *string* default: `''`

stopped: *string* default: `''`

users

Create operating system users.

Example:

```
- users:
    - name: appuser                 # required
      groups: appgroup www-data     # whitespace separated groups
      create-home: True
      shell: /bin/bash
      password: use appuser         # see passwords: section
```

Schema:

list of

dict of

create-home: *boolean* default: `True`

do: *string*

groups: *None* || *string* default: `None`

`if`:

if-exec: *string*

if-platform: *string*

name: *string* required

password: *string*

shell: *string*

vcs

Deploy code with this version control helper. $VCS support must be installed beforehand, see the packages section.

If the destination does not exist, an initial checkout is done. If it does exist, the repository is updated to the latest version.

Formats:

```
- VCS_TYPE REPO_URL DEST EXTRA          # simple or
- do: VCS_TYPE REPO_URL DEST EXTRA      # robust
```

VCS_TYPE: Version Control System, one of `svn`, `hg`, or `git`.

REPO_URL: URL to repository.

DEST: Destination folder.

EXTRA: Extra command line parameters.

do: What to do.

input: Useful to automate interactive prompts.

user:: Execute vcs under this remote username.

Example:

```
- vcs:
    - do: hg %repo_base/appname ~%user/appname
      input: p\nn                           # interactive answer
      user: %user
```

Schema:

list of *string dict of*

do: *string*

if:

if-exec: *string*

if-platform: *string*

input: *string*

user: *string*

1.4 Command-line Reference

1.4.1 Usage

```
Usage:
    pave v0.68 - (C) 2012-2013 Mike Miller.
    License: GNU GPLv3+
    Yet another config and deployment tool, leveraging fabric.

    pave [options]

Options:
  --version             show program's version number and exit
```

```
  -h, --help            show this help message and exit
  -b, --brace-exp       Use string.format-style brace expansion instead of
                        printf.
  -c STR, --command=STR
                        Override pavefile tasks with an ad hoc command.
  -f NAME, --filename=NAME
                        Use an input file besides ./pave.yml.
  -i, --interactive     Stop and ask whether to run each task group.
  -j #, --jobs=#        Run pave tasks in parallel with the given pool size.
  -o N V, --option=N V  Override a pavefile option.
  -s #, --select=#      Select task groups to run, instead of all. (0-based
                        slice syntax, # or #:#).
  -S, --skel            Output a "skeleton" pavefile to get started on.
  -t T, --target=T      Override targets with this host or group.  Multiple
                        accepted.
  -q, --quiet           Silence routine output, allowing warnings and errors.
                        Overrides verbose.
  -v  N V, --var= N V   Override a pavefile variable.
  -V, --verbose         Enable overwhelmingly verbose debugging output.
  --color=CHOICE        Message coloring/icons ('auto', 'on', 'off').
  --crypt               Ask for a password, print its hash and exit. For use
                        with users: module in the pavefile.
  --speak               Talk to me, holmes.
  --test                Test mode: parse and validate, skip execution.
```

Exit Status Codes

Pave returns standard status codes from `sysexits.h`:

```
EX_OK           0       # successful termination
EX_USAGE        64      # command line usage error
EX_DATAERR      65      # data format error
EX_NOINPUT      66      # cannot open input
EX_NOUSER       67      # addressee unknown
EX_NOHOST       68      # host name unknown
EX_UNAVAILABLE  69      # service unavailable
EX_SOFTWARE     70      # internal software error
EX_OSERR        71      # system error (e.g., can't fork)
EX_OSFILE       72      # critical OS file missing
EX_CANTCREAT    73      # can't create (user) output file
EX_IOERR        74      # input/output error
EX_TEMPFAIL     75      # temp failure; user is invited to retry
EX_PROTOCOL     76      # remote error in protocol
EX_NOPERM       77      # permission denied
EX_CONFIG       78      # configuration error
```

Password "Escape Hatch"

In addition to the standard keyboard interrupt `Ctrl+C`, if a password is requested (via the pavefile) at the command line the process can be escaped by giving two empty passwords in a row. The program will exit.

1.4.2 Option Path Syntax

To override pavefile options on the fly, "option path" syntax is used, which describes the hierarchy in a compact manner. This syntax is used for command-line (`-o|--option`) overrides. Inside the pavefile, however, path separators (`/`) are *not* supported; indentation is used instead.

```
main:           # pavefile
    jobs: 1
```

equals:

```
/main/jobs 1    # command-line
```

1.4.3 Task Selection

Selection of a subset of tasks (or "task groups") can be done with the `-s/--select` option, and uses Python-style "slicing" syntax.

Indexes are zero-based, and use negative numbers to specify items from the end. Start and stop indexes, and/or step values may be given and are separated with the colon : character. If a part is omitted, it defaults to start, end, or a step of 1.

Below is a visualization that may be helpful. Given a list of tasks, the task index marks the beginning (left or top) edge:

```
 +---+---+---+---+---+              # horizontal
 | T | T | T | T | T |
 +---+---+---+---+---+
 0   1   2   3   4  (5=length)
-5  -4  -3  -2  -1

- TASK1, index  0  or  -5           # vertical
- TASK2, index  1  or  -4
- TASK3, index  2  or  -3
- TASK4, index  3  or  -2
- TASK5, index  4  or  -1
```

Example:

```
pave -s 0               # the first task, group only
pave -s -2              # the second to last task, group
pave --select 10:20     # task groups 10-19
pave --select :6:2      # every other task from the first six
```

Further discussion can be found here:

- **http://docs.python.org/2/tutorial/introduction.html#strings** (slicing discussion starts about halfway down the section.)
- http://stackoverflow.com/a/509295/450917

1.5 Developer Guide

"Use the SOURCE, Luke."

1.5.1 Source Tour

Pave is GPLv3+ licensed and may be available in commercial-friendly proprietary license in the future as well.

Follow along at bitbucket if you'd like. There is an executable script typically installed to `/usr/local/bin/pave` that does little but hand things off to the main module `pave/main.py`.

This main module gets everything rolling...

- In `load_data()`, the pavefile is parsed and validated via `pave/schema.py` using the voluptuous library.
- The tasks(s) are then handed off to fabric for execution at the end of `main()` — each host gets `steamroll()`'ed.
- Each remote host is first inspected with `pave/inspector.py`. A json blob is cached in the .cache folder and returned with the results.
- With inspector results in hand, a set of platform-dependent shell commands and settings (as specified in `pave/platforms.py`) are chosen.

  This is the place to start on new platform support.

- The `pave/lib/` folder contains the modules that implement items under the `tasks:` section of the pavefile.

  This is the place to add modules to support new functionality.

1.5.2 How to Write a Task Module

So, you'd like to create a module to simplify an ugly or tedious task—this is the right place. Task modules are written in Python and run locally, but generally speak in shell script at the remote end. (If you don't know either language, there's still hope, an alternative described in the *FAQ - (Not So) Frequently Asked Questions*).

Let's get started:

1. You'll probably want to copy or at least refer to an existing one from pave/lib that is most similar to one you'd like to write.

   If the module will do a lot of repetetive tiny tasks, you may want the primary data structure to be a list. For a single task with lots of options a mapping is a good choice. Keep this in mind when choosing a module to copy from.

2. The module needs the following items:

   - a handler function of the following signature:

     ```
     def handle(data, cmds, context):
     ```

     - `data` is the task's parsed structure passed in from the pavefile. The convention when using mapping/dict keys is to use dashes instead of spaces, like css, e.g.: `foo-bar:`
     - `cmds` is the platform information object chosen by the inspector, the attributes of which should be used in place of command strings.
     - `context` is a dictionary that keeps track of properties such as the current user, whether to use sudo, and if so, as what user, etc. *Make a copy if you need to modify it per-task*:

       ```
       for task in tasks:
           mycon = context.copy()       # prevents spills
       ```

   - To check against data-entry errors, the passed `data` structure should have a robust schema definition (and validator functions), the building blocks of which can be imported from voluptuous.

- Generally speaking if the data is a list you'll want to loop over it, if a dictionary, you might .get() or .pop() off each member as you go.
- There are a few helpful variables pave attaches to fabric that are accessible by importing its `env` variable:

```
env.pave_platform           # the platform obj also known as cmds
env.pave_platform_details   # a dict holding system parameters
env.pave_vars               # a dict holding pavefile vars:
env.pave_passwords          # a dict holding pavefile passwords:

# configures pave to stop at the first error:
env.pave_raise_errs         # set by /main/warn-only

# makes copies of original files before changing them:
env.pave_bak                # set by /main/bak-files
```

3. Use the helper functions in `pave/utils.py` whenever possible. They handle common situations for modules.

   `runcmd()` is probably the most important, as it wraps fabric operations. See below, `cmds` provides the cmd-line for the current platform, while arguments for printf-style `%s` string expansion follow. Finally `context` provides information for runcmd to make decisions:

```
result = runcmd(cmds.pkg_upgrd_time, expires, **context)
```

   - A note about command strings from `platforms.py`. Occassionally string formatting with positional args is not sufficient, and the need arises to add or subtract whole parameters based on vars, similar to python keyword arguments. This is possible thru what the source calls (for lack of a better term) "chunking"... referring to the chunks in the command line:

```
createdb [--locale] [--encoding] [--tablespace]      # kwargs
createuser [--encrypted|no-encrypted]                # boolean
```

     Chunking can be enabled via `context.chunking = True` and can be seen in the postgres, users, and groups modules, for example.

4. Try to reuse the other modules if possible. For example, the `postgres:` module leverages the `configure:` module for its config file modifications.
5. If a task is about to make a change, log it with `log.change()`. If the task should be skipped, log it with `log.skip()` instead.
6. Keep a running tab of these "events," and return the number of changes and/or errors (in that order) generated during the module handler run.
7. When satisfied with the module, run `pyflakes` and/or `pylint` against it to light up dark corners. There is a `check_files.sh` in the root of the project to help with this. This would also be a good time to write a few tests for it.

- A final note to be careful about importing `pave.main`, `pave.schema`, or handle methods (of other library modules) in new modules. Doing so can create circular dependencies, as the schema module imports all modules at startup looking for schemas. Instead, import these into each needy function to delay import until runtime.

1.5.3 Roadmap and Wishlist

Can you help with any of these needs?

- **Platforms - Would appreciate some help getting it to run on:**
  - RHEL6+ and other common distributions.
  - *BSD, OSX client

  - Windows

- Common software support which has special needs or tedium that aren't met with the current library.
- The validation of pavefile sections and options is lacking. For example, data types and number of arguments are generally checked, but files aren't checked for existence prior to execution. This is important for a 1.0 release. Check the voluptuous page linked above on how to do that.
- Optimizations from gurus of various disciplines.
- Necessary features from established systems that don't harm the simple use-cases.
- Tests — The design is starting to solidify, so writing tests would be useful.
- Docs too. Sphinx doc building is currently a bit of a mess.

  Note: parts of the docs are built dynamically, such as schema-related details. Unfortunately, readthedocs doesn't handle this. :/ So, docs will need to be re-built after changes to the schema and the output files checked in. There is a script called `docs/bld_docs` to automate the process.

1.5.4 Contributing

- Please consult with me before making major improvements if you'd like them included here. ;)
- Currently we're asking that you transfer copyright of your additions to the project to avoid issues if a change of gears is warranted. Is this a problem?.
- **Pull requests:**
  - Please fix/improve one thing (or highly-related things) at a time.
  - Please test thoroughly before submission as I don't have much time or resources available. VirtualBox is free and easy to use, (if there isn't something you'd rather use).
- **Code should be in the pep8 style.** I tend to do a few other things like line up columns and put double blank lines between classes and standalone functions that might bug people. Sorry about that. I've got an extra high-res portrait monitor for coding/browsing and vertical space is plentiful.
- Single quotes by default, since they are easier to look at and type on keyboards in common languages.

  One exception are the command-lines in `platforms.py`. Since fabric doesn't escape single quotes in command-lines, is it easier to visually parse the shell debug output if you quote with single on command-line tasks whenever possible. As single quotes will therefore be common inside command-line strings it is easier to wrap the whole thing in double-quotes or (if mixed, triple-single) in Python.

1.5.5 Module Reference

pave

pave Package

`cmds` Module

`inspector` Module This is a short script which is copied over to remote hosts to gather details. Uses python's platform module to gather the info, which is cached in the ~/.cache folder in json format.

`pave.inspector.`**`query`**`()`

main Module %%prog v%s - (C) 2012-2014 Mike Miller. License: GNU GPLv3+ Yet another config and deployment tool, leveraging fabric.

%%prog [options]

`pave.main.`**`expand_targets`**(*targets*, *groups*)
Convert group names into targets and add them to the list.

`pave.main.`**`expandstr`**(*self*, *node*)
Yaml loader/constructor that expands variable strings.

`pave.main.`**`fatal_exit`**(*msg*, *exitcode*, *parser=None*)
Convenience function to log and exit on fatal error.

`pave.main.`**`get`**(*cfgdata*, *path*, *offset=0*)
An xpath-like yaml data grabber. Looks at these in order, last one wins:

- Schema default
- Config file
- Command line

`pave.main.`**`get_module`**(*modname*)
Given a module name, find and load it.

`pave.main.`**`get_platform_cmds`**(*pldata*, *local=False*)
Given platform data, return the appropriate command class.

`pave.main.`**`load_data`**(*opts*, *args*)
Load the pavefile and expand any variables found.

`pave.main.`**`main`**(*opts*, *args*)
Start paving.

`pave.main.`**`prep_environ`**(*data*, *opts*)
Prepare execution environment according to options given.

`pave.main.`**`setup`**()
Parse command-line, create temp folders, set up logging.

`pave.main.`**`steamroll`**(*data*, *inspectors*, *context*)
Let's get rollin...

schema Module (C) 2012-2014 Mike Miller. License: GNU GPLv3+

Strangely enough, schema and schema-related code lives here.

exception `pave.schema.`**`SchemaError`**
Bases: `exceptions.Exception`

`pave.schema.`**`if_var_three_args`**(*v*)

`pave.schema.`**`validate_varname`**(*value*, *msg=None*)
Rules for variable names in the pavefile.

targets Module This module is for enhancing the inventory parsing capability such that it can deal with hostnames specified using a simple pattern in the form of [beg:end], example: [1:5] where if begin is not specified, it defaults to 0.

If beg is given and is left-zero-padded, e.g. '001', it is taken as a formatting hint when the range is expanded. e.g. [001:010] is to be expanded into 001, 002 ...009, 010.

Note that when beg is specified with left zero padding, then the length of end must be the same as that of beg, else a exception is raised.

exception `pave.targets.HostSpecError`
Bases: `exceptions.Exception`

`pave.targets.detect_range(hostspec)`
A helper function that checks a given host string to see if it contains a range pattern descibed in the docstring above.

Returns a True match object if the given string contains a pattern, else None.

```
>>> detect_range('hostname4')

>>> detect_range('192.[168].2.1')

>>> detect_range('foo[bar]baz')

>>> detect_range('[@$\n*#:*\t$&@]\]')

>>> detect_range('host_name[1:8]')  # no underscores in hostnames

>>> detect_range('host[1:8]').groups()
('host', '1', '8', '')

>>> detect_range('10.0.0.1[01:08]').groups()
('10.0.0.1', '01', '08', '')

>>> detect_range('host[1:8].with-dash.com').groups()
('host', '1', '8', '.with-dash.com')

>>> detect_range('www-[0001:8000].g.cn').groups()
('www-', '0001', '8000', '.g.cn')
```

`pave.targets.expand_hostname_range(matchobj)`
A helper function that expands a given string that contains a pattern specified in top docstring, and returns a list that consists of the expanded version.

The '[' and ']' characters are used to maintain the pseudo-code appearance. They are replaced in this function with '|' to ease string splitting.

Example:

```
Passing a match obj allows a second execution of the regex to be
skipped:
match = detect_range(hostspec)
if match:
    expand_hostname_range(match)
```

References: http://ansible.github.com/patterns.html#hosts-and-groups

```
>>> expand_hostname_range(detect_range('host[1:8]'))
['host1', 'host2', 'host3', 'host4', 'host5', 'host6', 'host7', 'host8']

>>> expand_hostname_range(detect_range('host[0-2]'))
['host0', 'host1', 'host2']

>>> expand_hostname_range(detect_range('host[0:3]'))
['host0', 'host1', 'host2', 'host3']
```

```
>>> expand_hostname_range(detect_range('host[:2]'))
['host0', 'host1', 'host2']

>>> expand_hostname_range(detect_range('www[01:04].g.cn'))
['www01.g.cn', 'www02.g.cn', 'www03.g.cn', 'www04.g.cn']

>>> expand_hostname_range(detect_range('[0001:0002]node.g.cn'))
['0001node.g.cn', '0002node.g.cn']
```

utils Module (C) 2012-2014 Mike Miller. License: GNU GPLv3+

A grab bag of assorted stuff.

class pave.utils.**AttrDict**

Bases: dict

Dict that acts like an object.

copy()

class pave.utils.**PaveSafeLoader**(*stream*)

Bases: yaml.loader.SafeLoader

Disables the use of "%" chars as directive characters, so we don't have to quote so much.

check_directive()

check_plain()

class pave.utils.**VarTemplate**(*template*)

Bases: string.Template

delimiter = '%'

idpattern = '[_\\w][_\\w0-9]*'

pattern = <_sre.SRE_Pattern object at 0x2fcbf40>

pave.utils.**ask_crypt**(*tries=3*)

Asks for a password and returns crypt'd.

pave.utils.**backup**(*fname*, *cmds*, *context*)

Make a .orig backup of files we are going to modify.

pave.utils.**bold**(*text*)

Add a bold attribute to given text.

pave.utils.**check_list_conditionals**(*item*, *context*, *module=None*, *replace=True*, *force=False*)

Check if item is a dictionary and if so eval conditionals. Helpful For list-based modules. Returns None if item should be skipped. Replace - replace object with its do/then clause.

pave.utils.**contains**(*filename*, *text*, *exact=False*, *use_sudo=False*, *escape=False*, *fixed=False*)

Copied from fabric–shows output and escape defaults to False. I may remove this and reimplement.

pave.utils.**crypt**(*passwd*)

Crypt a password. See man crypt for details.

pave.utils.**donepath**(*cmds*, *base*, *itemname*, *item*)

Create and return a path to a "done" file.

pave.utils.**eval_conditionals**(*sectiondict*, *context*)

If multiple tests, must all be true; boolean AND.

pave.utils.**find_password**(*directive*, *encrypt=None*)

`pave.utils.`**`fmtcmd`**`(`*cmd*, **args*, ***options*`)`
format a command line given both arguments and options.

`pave.utils.`**`get_cursor_pos`**`()`
Return the current column number of the terminal cursor. Used to figure out if we need to print an extra newline.

`pave.utils.`**`getpwd`**`(`*prompt*, *tries=3*`)`
Request a password until two copies match.

`pave.utils.`**`markdone`**`(`*cmds*, *tdir*, *path*, *context*`)`
Mark a task as "done".

`pave.utils.`**`md5`**`(`*data*`)`
Given a piece of data, return its md5 hash.

`pave.utils.`**`q`**`(`*fname*`)`
Quote strings (e.g. filenames) if needed.

`pave.utils.`**`runcmd`**`(`*cmdline*, **args*, ***options*`)`
Run a command with fabric, using arguments to decide on sudo or not.

`pave.utils.`**`safe_eval`**`(`*text*, ***namespace*`)`
Disables builtins and provides only given variables. http://lybniz2.sourceforge.net/safeeval.html

`pave.utils.`**`sequence`**`(`*value*, **toappend*`)`
Given an object, check if it is a sequence. If not, add it to one. Return the sequence.

`pave.utils.`**`trunc`**`(`*msg*, *length*`)`
Truncate a string with trailing ellipsis.

`pave.utils.`**`wait_key`**`()`
Waits for a keypress at the console.

Subpackages

lib Package

`cleanup` Module

Good hygene is important.

`folders:` Deletion of files under `/tmp`, `/var/tmp`, & `/home/` only.

`locales-except:` *"Unless you're working at the UN and administer a central server for all member states, it is difficult to conceive why you would need a system where all of these locales are installed."* Similar to a one-time Debian locale-purge.

`packages:` Run appropriate package cleanup command(s).

Example:

```
- cleanup:
    packages: True
    locales-except: en en_US
    folders:  # use with caution
        - /tmp/stuff/*
        - /tmp/stuff2
```

`pave.lib.cleanup.`**`handle`**`(`*section*, *cmds*, *context*`)`

`configure` Module Configure the remote system. Also handles templating, see below.

`render:` Renders a template to a local file, which is then uploaded to host.

> **`engine:`** The templating language, such as python (printf|Template|format), or the executable name of an external package, e.g: `jinja2`

`render-remote:` Simple remote "templating" with sed.

Example:

```
- configure:            # str || list accepted on all sub-tasks below:
    create:  # file or folder/
        - foo.conf
        - /tmp/project/

    append:  # text remotefname
        Answer=42 %rc

    comment: # [comment char] from-regex remote-fname
        - ' "insecure on" %rcfile '

    replace: # from-regex to remote-fname
        - ^foo$ bar %rcfile
        # note that because of yaml syntax and splitting, two layers
        # of quoting or blocks may be needed when backslashes used:
        - >
            '^foo\s*$' bar %rcfile

    render: # engine [extra-vars] local-template remote-fname
        - printf swappiness=50 examples/printf_template.txt %rc2
        - format FOO=BAR       examples/format_template.txt %rc3

    render-remote: # [vars] remote-source remote-dest
        - INSTANCE_NUM=1 %sitecfg /etc/init/%svcname.conf

    update: # copy a file into place if newer, SRC DEST
        - ~/%sitename/cfg/celeryd.conf /etc/init/celeryd.conf

    access: # mode user group remote-fname # use "" for empty
        - 0640 %user %user %rc2
```

`pave.lib.configure.`**`handle`**(*section*, *cmds*, *context*)

`pave.lib.configure.`**`render`**(*engine*, *text*, *tvars*)
> Render templates.

`pave.lib.configure.`**`req`**(*key*)

`django` Module Django helper module. *Captain Obvious recommends Django be installed on the target beforehand, see the* **'packages'_** *section.*

`loaddata:` Load fixture files from given path, glob ok.

`manage:` Execute a management command every time this task group is run.

`manage-once:` Once and never again.

Example:

```
- django:
    workdir: ~%sitename/%sitename
    loaddata: "fixtures/*.json"
    manage-once: update_index
    syncdb: True
```

`pave.lib.django.`**`handle`**(*section*, *cmds*, *context*)
Run the system package manager, and any others given.

`groups` Module Create operating system user groups.

Example:

```
- groups:           # str || list || conditional
    - appgroup
    - "dev:1100"    # how to set gid

    - if-exec: test -e /dev/cdrom
      do: cdrom
```

`pave.lib.groups.`**`handle`**(*section*, *cmds*, *context*)

`kernel` Module Manage kernel parameters. Saves to memory and disk.

Example:

```
- kernel:           # str || list || conditional
    - vm.swappiness = 10
```

`pave.lib.kernel.`**`handle`**(*section*, *cmds*, *context*)

`keygen` Module Manage local/remote ssh keys and their generation.

`remote:` Generates a key at the remote host under the given username.

`copy-id:` Copies local keys to the remote host.

Values may be a single or list of Boolean (True copies the public key of the local user to the authorized keys file of the remote user), or string of two arguments containing the local file (.pub appended automatically) and the remote user.

Example:

```
- keygen:
    remote: "%sitename"             # quotes not required but
    copy-id:                        # fixes syntax highlighting
        - True                          # default src dest
        - "%local_path.pub %user"       # specified
```

`pave.lib.keygen.`**`check_ssh_config`**()
Check for config locally.

`pave.lib.keygen.`**`handle`**(*section*, *cmds*, *context*)

`pave.lib.keygen.`**`keygen`**(*type='rsa'*, *passphrase=''*, ***context*)
Generate ssh keys locally.

packages Module Manage software packages. Packages names for `install`, `remove`, etc. are listed as whitespace-delimited strings.

expiration: The age in days after which the OS package index is considered to have expired, requiring action.

To be a good neighbor, the default, 0 (= 24 hours) means packages won't be updated unless it has been a full day since the last update. A value of -1 forces action.

inventory: Make a list of installed packages to compare against.

update/upgrade: Update the system's package index, upgrade packages to latest release.

upgrade-full: Upgrade packages, even if it causes obsolete packages to be removed, or new packages to be installed, for example new kernel versions. e.g. Debian's "dist-upgrade".

Example:

```
- packages:
    if-platform: Debian
    install:
        sysvbanner python-pip
    # expiration: -1    # force attempt
```

`pave.lib.packages.`**cleanup**(*cmds*, *context*)
Clean any mess left over by package installation.

`pave.lib.packages.`**handle**(*section*, *cmds*, *context*)
Run the system package manager, and any others given.

postgres Module Manage postgres. *Captain Obvious here to chime-in that postgres must be installed on remote target beforehand, see* **'packages'_**.

link current: Create a symbolic link from the last sorted item in `/etc/postgresql/` to `/etc/postgresql/curr`.

replace, access: See the corresponding directive under configure. These are for convenience to group postgres-related configuration together.

user, db, grant, sql: `postgresql-client-common` (or distro equiv.) must be installed on remote target beforehand. Run the appropriate commands to create objects or configure the database. String or list of strings may be given.

Example:

```
- postgres:
    link current: False
    replace:
        - '#\s*password_enc password_enc %pgcfg/postgresql.conf'

    su-password:
        use pgsu                                # see passwords:
        # md5 acbd18db4cc2f85cedef654fccc4a4d8  # how to hard-code

    db: # quotes not required below, but fixes highlighting. :/
        - name: "%sitename"
    user:
        - name: "%sitename"
    grant:
        # privs   on_obj    named         to_user
        all       database  %sitename     %sitename
```

```
    sql: # str|list lines of sql, first token should be dbname:
        template1 select datname from pg_database
                  where datname = '%sitename';
```

`pave.lib.postgres.handle`(*section*, *cmds*, *context*)

`run` Module Executes a list of commands using the remote shell. Each command may be:

- **A string** A command that runs under the default user.
- **A mapping with one or more of the following members:**

  `do: CMD_STRING`
  The command to run.

  `user: USERNAME`
  Run as another user.
  May be `sudo` (root), or a valid remote username.

  `title: TITLE`
  to document or hide long, ugly commands.

  `workdir: PATH` set working directory.

  `local: <bool>` runs it locally instead of remotely.

See *Conditional Execution* for more details on `if:` and `if-exec:`.

Note: While *not recommended* to set passwords via command-line tools for various reasons, tasks makes an attempt to hide them (vars starting with "`pwd_`") from standard logs. Passwords will still be displayed in debug logs (and perhaps shell history) upon command-line execution.

Example:

```
- banner "Hello!"     # world

# signal a change event and prevent errors from halting execution:
- touch %filename && echo CHANGED || true
- rm /etc/nginx/sites-enabled/default && echo CHANGED || echo SKIPPED

# Even poor-er man's conditionals:
- " [ '%mode' == 'qa' ] && echo 'QA install' "

# more robust task definition:
- run:
    - title: Nothing to see here...
      if-exec: test -f /foo
      do: echo "This is running under $USER (appuser)"
      user: appuser
```

`pave.lib.run.handle`(*section*, *cmds*, *context*, *chatty=True*)

`pave.lib.run.make_title`(*title*, *command*, *pwds*)

`scp` Module Copy files to and fro. Given a source and destination, copy files.

Notes:

- By default a put (upload) is performed, but the direction can be specified as the first parameter.
- Paths with whitespace must be quoted.

- Destinations folders should end in a / character for efficiency, as doing so skips the need for several filesystem tests.
- If sudo is enabled, relative paths/tildes will be interpreted (and files owned) by the root user. Therefore it is recommended that absolute remote paths be given.
- Does not currently handle wildcards.
- While this module is similar to using the scp command-line program it may not work identically.

Example:

```
- scp:
    - SRC DEST          # put
    - SRC DEST MODE     # put, w/ Unix octal mode or "same"
    - get SRC DEST

    # example conditional
    - if: inst_ssl == True
      do: put bundle.crt "%sitefldr/ssl/bundle.crt"
```

Additional details:

- fabric.operations.get
- fabric.operations.put

`pave.lib.scp.`**`check`**(*v*)
validate command lines. Todo: should check if on disk.

`pave.lib.scp.`**`handle`**(*section*, *cmds*, *context*)

`pave.lib.scp.`**`tilde_helper`**(*path1*, *path2*)

`services` **Module** Manage system daemons/services. Values are whitespace separated strings.

`restart:, restart-fail:` restart will start the service if it was stopped, restart-fail will fail with an error if it was not running originally.

Note: Some daemon startup scripts are not written well and quit when the ssh session ends. To prevent this from occurring, prepend the service name with "`nohup:`" as shown below.

Example:

```
- services:
    # the follwing two are temporary until reboot:
    running: hostname hwclock
    stopped: whoopsie
    # permanently; /packages/remove more reliable on ubuntu
    disabled: whoopsie
    restart: nohup:memcached
```

`pave.lib.services.`**`check_nohup`**(*svcname*)
look for a nohup:SVCNAME construct.

`pave.lib.services.`**`handle`**(*section*, *cmds*, *context*)

`pave.lib.services.`**`req`**(*key*)

users Module Create operating system users.

Example:

```
- users:
    - name: appuser                # required
      groups: appgroup www-data    # whitespace separated groups
      create-home: True
      shell: /bin/bash
      password: use appuser        # see passwords: section
```

`pave.lib.users.handle(section, cmds, context)`

vcs Module Deploy code with this version control helper. $VCS support must be installed beforehand, see the **'packages'_** section.

If the destination does not exist, an initial checkout is done. If it does exist, the repository is updated to the latest version.

Formats:

```
- VCS_TYPE REPO_URL DEST EXTRA          # simple or
- do: VCS_TYPE REPO_URL DEST EXTRA      # robust
```

VCS_TYPE: Version Control System, one of `svn`, `hg`, or `git`.

REPO_URL: URL to repository.

DEST: Destination folder.

EXTRA: Extra command line parameters.

do: What to do.

input: Useful to automate interactive prompts.

user:: Execute vcs under this remote username.

Example:

```
- vcs:
    - do: hg %repo_base/appname ~%user/appname
      input: p\nn                          # interactive answer
      user: %user
```

`pave.lib.vcs.handle(section, cmds, context)`

`pave.lib.vcs.in_vclist(v)`
validate

`pave.lib.vcs.min_two_tokens(v)`
validate

- Bitbucket Repo

1.6 FAQ - (Not So) Frequently Asked Questions

1.6.1 I don't know any bash or Python, can I still use `$language` with pave?

Yes, (although learning some `sh` or `bash` is recommended), it is easy to use another language with pave (like it is done with ansible).

The basic idea is this... that you write a script, upload it to your host, and then run it there:

```
tasks:
    - packages:
        install: other-runtime          # if need-be, or

    - scp: put setup_script.pl /tmp/    # to use perl
    - /tmp/setup_script.pl --kapow
```

1.6.2 How can I make pavefile authoring easier?

- Use a professional text editor which can highlight yaml. If you are just starting out you might try Geany on Linux, Notepad++ on Windows, or TextMate2 on OSX. Make sure to set it to indent with spaces, not tabs.
- Make judicious use of pave's command-line parameters. See the *Command-line Reference* on usage.
  - Turn on the debug logging option `-V` to see how the yaml parser has interpreted the given text and many other details.
  - Likewise, use the test option `--test` to parse and validate the file while skipping execution.
  - `-i/--interactive` is good for stepping through execution of tasks.
  - `-s/--select` is great for testing each task as you write it:

    ```
    pave -s -1 -V
    ```

    Yes, that second parameter is a negative number one, signifying the last task. This is really helpful–you won't have to wait while multiple previous tasks are checked again.
  - `-o/--option` or `-v/--var` options can be used to set values, variables in the pavefile:

    ```
    pave -o /main/inspect sh
    ```

1.6.3 What about troubleshooting?

- See answer above.
- In addition to the verbose command-line parameter, debug text and exception tracebacks, etc, for the last five runs are typically recorded in:

  ```
  ~/.cache/pave/{localhost,targets}
  ```

  This location may be different on your machine, or if you are using the `XDG_CACHE_HOME` environment variable.
- Regexes: `grep -E/egrep` is used extensively on the remote-end when searching. For whatever reason it doesn't support `\d`, so don't waste hours debugging like I have. :/

Why the `%` variable expansion syntax? ——————————————~

Originally pave used Python's newer, niftier string format syntax `{name}` rather than the older printf-style substitution `%(name)s` as it is more readable and sophisticated. Unfortunately though, after writing a few pavefiles it became clear it was too much trouble.

Why? First, it conflicts with yaml mapping syntax. Not the end of the world you might think, it can be quoted... ok. Ooh, it conflicts with bash at times too, braces are used for many substitutions... uggh, ok. Went to add `{{ Jinja }}` support and kaboom!, the last anvil to break the camel's back. To escape braces in string.format syntax you need to double them, just what Jinja uses.

Attempting to use thse features together reminds me of the old driver's-ed film "Red Asphalt". Time to admit defeat on that front.

Printf-syntax on the other hand collides much less often. The `%` character is used for an esoteric feature in YAML that pave disables. I've found but a few scatted shell commands like find and stat that use them. Escaping is easy though, just double it, `%%`, no matching front and back (with different symbols) to worry about.

Finally pave has moved to even simpler shell-style syntax as implemented by Python's string.Template. To avoid collisions with Unix shell `$variables` however, we've kept percent as the expansion character.

There may be other exceptions, but one I can think of is Windows and so I've left a `-b` command line option to use brace format syntax instead.

1.6.4 What licensing is available?

pave is licensed as GPLv3+.

A commercial-friendly license may be available for a fee. Support/fixes/features might be offered for a fee as well, regardless of license.

What are your thoughts on the subject?

- *genindex*
- *modindex*
- *search*

Python Module Index

p

pave.inspector, 65
pave.main, 66
pave.schema, 66
pave.targets, 66
pave.utils, 68