Fabric cheatsheet

Example Code

```
import sys

from fabric import tasks
from fabric.api import run
from fabric.api import env
from fabric.network import disconnect_all

env.hosts = [
    'localhost',
    'examplehost',
    ]

env.user = 'user_name'
env.password = 'password'

def hello(who="world"):
    print "Hello {who}!".format(who=who)

def hostname():
    res = run('hostname')

def uptime():
    res = run('uptime')

def date():
    res = run('date')

def main(who="World"):
    res_uptime   = tasks.execute(uptime)
    res_hostname = tasks.execute(hostname)
    res_date     = tasks.execute(date)
    hello(who)

    disconnect_all() # Call this when you are done, or get an ugly exception!
    exit(0)

if __name__ == '__main__':
    main()
```

Command-line options

```
-a, --no_agent
-A, --forward-agent
--abort-on-prompts
-c RCFILE, --config=RCFILE
-d COMMAND, --display=COMMAND
--connection-attempts=M, -n M
-D, --disable-known-hosts
-f FABFILE, --fabfile=FABFILE
-F LIST_FORMAT, --list-format=LIST_FORMAT
-g HOST, --gateway=HOST
-h, --help
--hide=LEVELS
-H HOSTS, --hosts=HOSTS
-x HOSTS, --exclude-hosts=HOSTS
```

```
-i KEY_FILENAME
-I, --initial-password-prompt
-k
--keepalive=KEEPALIVE
--linewise
-l, --list
-p PASSWORD, --password=PASSWORD
-P, --parallel
--no-pty
-r, --reject-unknown-hosts
-R ROLES, --roles=ROLES
--set KEY=VALUE,...
-s SHELL, --shell=SHELL
--shortlist
--show=LEVELS
--ssh-config-path
--skip-bad-hosts
--skip-unknown-tasks
--timeout=N, -t N
--command-timeout=N, -T N
-V, --version
-w, --warn-only
-z, --pool-size
```

Imports

```
    from fabric.api import *
    # or:
    from fabric.api import abort, cd, env, get, hide, hosts, local, prompt, \
    put, require, roles, run, runs_once, settings, show, sudo, warn
```

CORE API

Context Managers

In `fabric.context_manangers`:

```
with cd('/var/www'):
    run('ls') # cd /var/www && ls
    with cd('website1'):
        run('ls') # cd /var/www/website1 && ls
```

- `cd(path)`
- `char_buffered(*args, **kwds)`
- `hide(*args, **kwds)`
- `lcd(path)`
- `path(path, behavior='append')` 'append': append given path to the current *PATH*, *e.g.PATH*=PATH:. This is the default behavior. 'prepend': prepend given path to the current *PATH*, *e.g.PATH* = <*path* > :PATH. 'replace': ignore previous value of $PATH altogether, e.g. PATH=.
- `prefix(command)`
- `quiet()`
- `remote_tunnel(*args, **kwds):` remote_tunnel accepts up to four arguments: remote_port (mandatory) is the remote port to listen to. local_port (optional) is the local port to connect to; the default is the same port as the remote one. local_host (optional) is the locally-reachable computer (DNS name or IP address) to connect to; the default is localhost (that is, the same computer Fabric is running on). remote_bind_address (optional) is the remote IP address to bind to for listening, on the current target. It should be an IP address assigned to an interface on the target (or a DNS name that resolves to such IP). You can use “0.0.0.0” to bind to all interfaces.
- `settings(*args, **kwargs)`
- `shell_env(**kw)`
- `show(*args, **kwds)`

- `warn_only()`

Decorators

- `fabric.decorators.hosts(*host_list)` Decorator defining which host or hosts to execute the wrapped function on. For example, the following will ensure that, barring an override on the command line, my_func will be run on host1, host2 and host3, and with specific users on host1 and host3:

  ```
  @hosts('user1@host1', 'host2', 'user2@host3')
  def my_func():
      pass
  ```

  Note that this decorator actually just sets the function's .hosts attribute, which is then read prior to executing the function.

- `fabric.decorators.parallel(pool_size=None)`
- `fabric.decorators.roles(*role_list)`
- `fabric.decorators.runs_once(func)`
- `fabric.decorators.serial(func)`
- `fabric.decorators.task(*args, **kwargs)`

- `fabric.decorators.with_settings(*arg_settings, **kw_settings)`

Network

- `fabric.network.disconnect_all()`

Operations

- `fabric.operations.get(remote_path, local_path=None)` local_path is the local file path where the downloaded file or files will be stored.
- `fabric.operations.open_shell(command=None)`
- `fabric.operations.put(local_path, remote_path, use_sudo=False, mirror_local_mode=False, mode=None)`
- `fabric.operations.reboot(wait)`
- `fabric.operations.run(command, shell=True, pty=True, combine_stderr=True)`
- `fabric.operations.sudo(command, shell=True, pty=True, combine_stderr=True, user=None)`
- `fabric.operations.local(command, capture=False) Uses: /bin/sh`
- `fabric.operations.prompt(text, key=None, default='', validate=None)`
- `fabric.operations.require(*keys, **kwargs)`

Tasks

`class fabric.tasks.Task(alias=None, aliases=None, default=False, *args, **kwargs)`

Abstract base class for objects wishing to be picked up as Fabric tasks. Instances of subclasses will be treated as valid tasks when present in fabfiles loaded by the fab tool.

    get_hosts(arg_hosts, arg_roles, arg_exclude_hosts, env=None)

Return the host list the given task should be using.

- `fabric.tasks.execute(task, *args, **kwargs)`

Utils

- `fabric.utils.abort(msg)`
- `fabric.utils.fastprint(text, show_prefix=False, end='', flush=True)`
- `fabric.utils.indent(text, spaces=4, strip=False)`
- `fabric.utils.puts(text, show_prefix=True, end='\n', flush=False)`
- `fabric.utils.warn(msg)`

Contrib API

Console Output Utilities

- `fabric.contrib.console.confirm(question, default=True)`

File and Directory Management

- `fabric.contrib.files.append(filename, text, use_sudo=False, partial=False, escape=True, shell=False)` Append string (or list of strings) text to filename.
- `fabric.contrib.files.comment(filename, regex, use_sudo=False, char='#', backup='.bak', shell=False)` Attempt to comment out all lines in filename matching regex. The default commenting character is # and may be overridden by the char argument.
- `fabric.contrib.files.contains(filename, text, exact=False, use_sudo=False, escape=True, shell=False)` Return True if filename contains text (which may be a regex.) By default, this function will consider a partial line match (i.e. where text only makes up part of the line it's on).

- `fabric.contrib.files.exists(path, use_sudo=False, verbose=False)` Return True if given path exists on the current remote host. If use_sudo is True, will use sudo instead of run. exists will, by default, hide all output (including the run line, stdout, stderr and any warning resulting from the file not existing) in order to avoid cluttering output. You may specify verbose=True to change this behavior.

- `fabric.contrib.files.first(*args, **kwargs)` Given one or more file paths, returns first one found, or None if none exist. May specify use_sudo and verbose which are passed to exists.

- `fabric.contrib.files.is_link(path, use_sudo=False, verbose=False)` Return True if the given path is a symlink on the current remote host. If use_sudo is True, will use sudo instead of run. is_link will, by default, hide all output. Give verbose=True to change this.

- `fabric.contrib.files.sed(filename, before, after, limit='', use_sudo=False, backup='.bak', flags='', shell=False)` Run a search-and-replace on filename with given regex patterns. Equivalent to sed -i -r -e "// s///g" . Setting backup to an empty string will, disable backup file creation. For convenience, before and after will automatically escape forward slashes, single quotes and parentheses for you, so you don't need to specify e.g. http://foo.com, instead just using http://foo.com is fine.

- `fabric.contrib.files.uncomment(filename, regex, use_sudo=False, char='#', backup='.bak', shell=False)` Attempt to uncomment all lines in filename matching regex. The default comment delimiter is # and may be overridden by the char argument. This function uses the sed function, and will accept the same use_sudo, shell and backup keyword arguments that sed does. uncomment will remove a single whitespace character following the comment character, if it exists, but will preserve all preceding whitespace. For example, # foo would become foo (the single space is stripped) but `#  foo` would become `foo` (the single space is still stripped, but the preceding 4 spaces are not.)

- `fabric.contrib.files.upload_template(filename, destination, context=None, use_jinja=False, template_dir=None, use_sudo=False, backup=True, mirror_local_mode=False, mode=None, pty=None)` Render and upload a template text file to a remote host. filename should be the path to a text file, which may contain Python string interpolation formatting and will be rendered with the given context dictionary context (if given.) Alternately, if use_jinja is set to True and you have the Jinja2 templating library available, Jinja will be used to render the template instead. Templates will be loaded from the invoking user's current working directory by default, or from template_dir if given.

Project Tools

- `fabric.contrib.project.rsync_project(*args, **kwargs)` Synchronize a remote directory with the current project directory via rsync. Where upload_project() makes use of scp to copy one's entire project every time it is invoked, rsync_project() uses the rsync command-line utility, which only transfers files newer than those on the remote end. rsync_project() is thus a simple wrapper around rsync; for details on how rsync works, please see its manpage. rsync must be installed on both your local and remote systems in order for this operation to work correctly.
- `fabric.contrib.project.upload_project(local_dir=None, remote_dir='', use_sudo=False)` local_dir specifies the local project directory to upload, and defaults to the current working directory. remote_dir specifies the target directory to upload into (meaning that a copy of local_dir will appear as a subdirectory of remote_dir) and defaults to the remote user's home directory. use_sudo specifies which method should be used when executing commands remotely. sudo will be used if use_sudo is True, otherwise run will be used.

Output Levels

Standard output levels:

- status: Status messages, i.e. noting when Fabric is done running, if the user used a keyboard interrupt, or when servers are disconnected from. These messages are almost always relevant and rarely verbose.
- aborts: Abort messages. Like status messages, these should really only be turned off when using Fabric as a library, and possibly not even then. Note that even if this output group is turned off, aborts will still occur – there just won't be any output about why Fabric aborted!
- warnings: Warning messages. These are often turned off when one expects a given operation to fail, such as when using grep to test existence of text in a file. If paired with setting env.warn_only to True, this can result in fully silent warnings when remote programs fail. As with aborts, this setting does not control actual warning behavior, only whether warning messages are printed or hidden.
- running: Printouts of commands being executed or files transferred, e.g. [myserver] run: ls /var/www. Also controls printing of tasks being run, e.g. [myserver] Executing task 'foo'.
- stdout: Local, or remote, stdout, i.e. non-error output from commands.
- stderr: Local, or remote, stderr, i.e. error-related output from commands.
- user: User-generated output, i.e. local output printed by fabfile code via use of the fastprint or puts functions.
- debug: Turn on debugging (which is off by default.) Currently, this is largely used to view the "full" commands being run; take for example this run call:

Output level aliases:

- output: Maps to both stdout and stderr. Useful for when you only care to see the 'running' lines and your own print statements (and warnings).
- everything: Includes warnings, running, user and output (see above.) Thus, when turning off everything, you will only see a bare minimum of output (just status and debug if it's on), along with your own print statements.
- commands: Includes stdout and running. Good for hiding non-erroring commands entirely, while still displaying any stderr output.

Parallel Output

```
from fabric.api import *
@parallel(pool_size=5)
def runs_in_parallel():
    pass

@serial
def runs_serially():
    pass
```

Tasks

```
    from fabric.api import task, run
    @task
    def mytask():
        run("a command")
```

When this decorator is used, it signals to Fabric that only functions wrapped in the decorator are to be loaded up as valid tasks. (When not present, classic-style task behavior kicks in.)

- `task_class`: The Task subclass used to wrap the decorated function. Defaults to WrappedCallableTask.
- `aliases`: An iterable of string names which will be used as aliases for the wrapped function. See Aliases for details.
- `alias`: Like aliases but taking a single string argument instead of an iterable. If both alias and aliases are specified, aliases will take precedence.
- `default`: A boolean value determining whether the decorated task also stands in for its containing module as a task name. See Default tasks.
- `name`: A string setting the name this task appears as to the command-line interface. Useful for task names that would otherwise shadow Python builtins (which is technically legal but frowned upon and bug-prone.)

SSH Options

User and Port will be used to fill in the appropriate connection parameters when not otherwise specified, in the following fashion: Globally specified User/Port will be used in place of the current defaults (local username and 22, respectively) if the appropriate env vars are not set. However, if env.user/env.port are set, they override global User/Port values. User/port values in the host string itself (e.g. hostname:222) will override everything, including any ssh_config values.

HostName can be used to replace the given hostname, just like with regular ssh. So a Host foo entry specifying HostName example.com will allow you to give Fabric the hostname 'foo' and have that expanded into 'example.com' at connection time. IdentityFile will extend (not replace) env.key_filename. ForwardAgent will augment env.forward_agent in an "OR" manner: if either is set to a positive value, agent forwarding will be enabled. ProxyCommand will trigger use of a proxy command for host connections, just as with regular ssh.

Color output functions

```
from fabric.colors import green
print(green("This text is green!"))

# Other colours:
# fabric.colors.blue(text, bold=False)
# etc.
# Others: cyan, green, magenta, red, white, yellow
```