Fabric

Release

June 17, 2014

Contents

This site covers Fabric's usage & API documentation. For basic info on what Fabric is, including its public changelog & how the project is maintained, please see the main project website.

CHAPTER 1

Tutorial

For new users, and/or for an overview of Fabric's basic functionality, please see the *Overview and Tutorial*. The rest of the documentation will assume you're at least passingly familiar with the material contained within.

1.1 Overview and Tutorial

Welcome to Fabric!

This document is a whirlwind tour of Fabric's features and a quick guide to its use. Additional documentation (which is linked to throughout) can be found in the *usage documentation* – please make sure to check it out.

1.1.1 What is Fabric?

As the `README` says:

> Fabric is a Python (2.5-2.7) library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.

More specifically, Fabric is:

- A tool that lets you execute **arbitrary Python functions** via the **command line**;
- A library of subroutines (built on top of a lower-level library) to make executing shell commands over SSH **easy** and **Pythonic**.

Naturally, most users combine these two things, using Fabric to write and execute Python functions, or **tasks**, to automate interactions with remote servers. Let's take a look.

1.1.2 Hello, `fab`

This wouldn't be a proper tutorial without "the usual":

```
def hello():
    print("Hello world!")
```

Placed in a Python module file named `fabfile.py` in your current working directory, that `hello` function can be executed with the `fab` tool (installed as part of Fabric) and does just what you'd expect:

```
$ fab hello
Hello world!

Done.
```

That's all there is to it. This functionality allows Fabric to be used as a (very) basic build tool even without importing any of its API.

Note: The `fab` tool simply imports your fabfile and executes the function or functions you instruct it to. There's nothing magic about it – anything you can do in a normal Python script can be done in a fabfile!

See also:

Execution strategy, *Defining tasks*, *fab options and arguments*

1.1.3 Task arguments

It's often useful to pass runtime parameters into your tasks, just as you might during regular Python programming. Fabric has basic support for this using a shell-compatible notation: `<task name>:<arg>,<kwarg>=<value>,...`. It's contrived, but let's extend the above example to say hello to you personally:

```
def hello(name="world"):
    print("Hello %s!" % name)
```

By default, calling `fab hello` will still behave as it did before; but now we can personalize it:

```
$ fab hello:name=Jeff
Hello Jeff!

Done.
```

Those already used to programming in Python might have guessed that this invocation behaves exactly the same way:

```
$ fab hello:Jeff
Hello Jeff!

Done.
```

For the time being, your argument values will always show up in Python as strings and may require a bit of string manipulation for complex types such as lists. Future versions may add a typecasting system to make this easier.

See also:

Per-task arguments

1.1.4 Local commands

As used above, `fab` only really saves a couple lines of `if __name__ == "__main__"` boilerplate. It's mostly designed for use with Fabric's API, which contains functions (or **operations**) for executing shell commands, transferring files, and so forth.

Let's build a hypothetical Web application fabfile. This example scenario is as follows: The Web application is managed via Git on a remote host `vcshost`. On `localhost`, we have a local clone of said Web application. When we push changes back to `vcshost`, we want to be able to immediately install these changes on a remote host `my_server` in an automated fashion. We will do this by automating the local and remote Git commands.

Fabfiles usually work best at the root of a project:

```
.
|-- __init__.py
|-- app.wsgi
|-- fabfile.py <-- our fabfile!
|-- manage.py
`-- my_app
    |-- __init__.py
    |-- models.py
    |-- templates
    |   `-- index.html
    |-- tests.py
    |-- urls.py
    `-- views.py
```

Note: We're using a Django application here, but only as an example – Fabric is not tied to any external codebase, save for its SSH library.

For starters, perhaps we want to run our tests and commit to our VCS so we're ready for a deploy:

```
from fabric.api import local

def prepare_deploy():
    local("./manage.py test my_app")
    local("git add -p && git commit")
    local("git push")
```

The output of which might look a bit like this:

```
$ fab prepare_deploy
[localhost] run: ./manage.py test my_app
Creating test database...
Creating tables
Creating indexes
.......................................
----------------------------------------------------------------------
Ran 42 tests in 9.138s

OK
Destroying test database...

[localhost] run: git add -p && git commit

<interactive Git add / git commit edit message session>

[localhost] run: git push

<git push session, possibly merging conflicts interactively>

Done.
```

The code itself is straightforward: import a Fabric API function, `local`, and use it to run and interact with local shell commands. The rest of Fabric's API is similar – it's all just Python.

See also:

Operations, *Fabfile discovery*

1.1.5 Organize it your way

Because Fabric is "just Python" you're free to organize your fabfile any way you want. For example, it's often useful to start splitting things up into subtasks:

```
from fabric.api import local

def test():
    local("./manage.py test my_app")

def commit():
    local("git add -p && git commit")

def push():
    local("git push")

def prepare_deploy():
    test()
    commit()
    push()
```

The `prepare_deploy` task can be called just as before, but now you can make a more granular call to one of the sub-tasks, if desired.

1.1.6 Failure

Our base case works fine now, but what happens if our tests fail? Chances are we want to put on the brakes and fix them before deploying.

Fabric checks the return value of programs called via operations and will abort if they didn't exit cleanly. Let's see what happens if one of our tests encounters an error:

```
$ fab prepare_deploy
[localhost] run: ./manage.py test my_app
Creating test database...
Creating tables
Creating indexes
.............E.........................
======================================================================
ERROR: testSomething (my_project.my_app.tests.MainTests)
----------------------------------------------------------------------
Traceback (most recent call last):
[...]

----------------------------------------------------------------------
Ran 42 tests in 9.138s

FAILED (errors=1)
Destroying test database...

Fatal error: local() encountered an error (return code 2) while executing './manage.py test my_app'

Aborting.
```

Great! We didn't have to do anything ourselves: Fabric detected the failure and aborted, never running the `commit` task.

See also:

Failure handling (usage documentation)

Failure handling

But what if we wanted to be flexible and give the user a choice? A setting (or **environment variable**, usually shortened to **env var**) called *warn_only* lets you turn aborts into warnings, allowing flexible error handling to occur.

Let's flip this setting on for our `test` function, and then inspect the result of the `local` call ourselves:

```
from __future__ import with_statement
from fabric.api import local, settings, abort
from fabric.contrib.console import confirm

def test():
    with settings(warn_only=True):
        result = local('./manage.py test my_app', capture=True)
    if result.failed and not confirm("Tests failed. Continue anyway?"):
        abort("Aborting at user request.")

[...]
```

In adding this new feature we've introduced a number of new things:

- The `__future__` import required to use `with:` in Python 2.5;
- Fabric's `contrib.console` submodule, containing the `confirm` function, used for simple yes/no prompts;
- The `settings` context manager, used to apply settings to a specific block of code;
- Command-running operations like `local` can return objects containing info about their result (such as `.failed`, or `.return_code`);
- And the `abort` function, used to manually abort execution.

However, despite the additional complexity, it's still pretty easy to follow, and is now much more flexible.

See also:

Context Managers, *Full list of env vars*

1.1.7 Making connections

Let's start wrapping up our fabfile by putting in the keystone: a `deploy` task that is destined to run on one or more remote server(s), and ensures the code is up to date:

```
def deploy():
    code_dir = '/srv/django/myproject'
    with cd(code_dir):
        run("git pull")
        run("touch app.wsgi")
```

Here again, we introduce a handful of new concepts:

- Fabric is just Python – so we can make liberal use of regular Python code constructs such as variables and string interpolation;
- `cd`, an easy way of prefixing commands with a `cd /to/some/directory` call. This is similar to `lcd` which does the same locally.
- `run`, which is similar to `local` but runs **remotely** instead of locally.

We also need to make sure we import the new functions at the top of our file:

```
from __future__ import with_statement
from fabric.api import local, settings, abort, run, cd
from fabric.contrib.console import confirm
```

With these changes in place, let's deploy:

```
$ fab deploy
No hosts found. Please specify (single) host string for connection: my_server
[my_server] run: git pull
[my_server] out: Already up-to-date.
[my_server] out:
[my_server] run: touch app.wsgi

Done.
```

We never specified any connection info in our fabfile, so Fabric doesn't know on which host(s) the remote command should be executed. When this happens, Fabric prompts us at runtime. Connection definitions use SSH-like "host strings" (e.g. `user@host:port`) and will use your local username as a default – so in this example, we just had to specify the hostname, `my_server`.

Remote interactivity

`git pull` works fine if you've already got a checkout of your source code – but what if this is the first deploy? It'd be nice to handle that case too and do the initial `git clone`:

```
def deploy():
    code_dir = '/srv/django/myproject'
    with settings(warn_only=True):
        if run("test -d %s" % code_dir).failed:
            run("git clone user@vcshost:/path/to/repo/.git %s" % code_dir)
    with cd(code_dir):
        run("git pull")
        run("touch app.wsgi")
```

As with our calls to `local` above, `run` also lets us construct clean Python-level logic based on executed shell commands. However, the interesting part here is the `git clone` call: since we're using Git's SSH method of accessing the repository on our Git server, this means our remote `run` call will need to authenticate itself.

Older versions of Fabric (and similar high level SSH libraries) run remote programs in limbo, unable to be touched from the local end. This is problematic when you have a serious need to enter passwords or otherwise interact with the remote program.

Fabric 1.0 and later breaks down this wall and ensures you can always talk to the other side. Let's see what happens when we run our updated `deploy` task on a new server with no Git checkout:

```
$ fab deploy
No hosts found. Please specify (single) host string for connection: my_server
[my_server] run: test -d /srv/django/myproject

Warning: run() encountered an error (return code 1) while executing 'test -d /srv/django/myproject'

[my_server] run: git clone user@vcshost:/path/to/repo/.git /srv/django/myproject
[my_server] out: Cloning into /srv/django/myproject...
[my_server] out: Password: <enter password>
[my_server] out: remote: Counting objects: 6698, done.
[my_server] out: remote: Compressing objects: 100% (2237/2237), done.
[my_server] out: remote: Total 6698 (delta 4633), reused 6414 (delta 4412)
```

```
[my_server] out: Receiving objects: 100% (6698/6698), 1.28 MiB, done.
[my_server] out: Resolving deltas: 100% (4633/4633), done.
[my_server] out:
[my_server] run: git pull
[my_server] out: Already up-to-date.
[my_server] out:
[my_server] run: touch app.wsgi

Done.
```

Notice the `Password:` prompt – that was our remote `git` call on our Web server, asking for the password to the Git server. We were able to type it in and the clone continued normally.

See also:

Interaction with remote programs

Defining connections beforehand

Specifying connection info at runtime gets old real fast, so Fabric provides a handful of ways to do it in your fabfile or on the command line. We won't cover all of them here, but we will show you the most common one: setting the global host list, *env.hosts*.

env is a global dictionary-like object driving many of Fabric's settings, and can be written to with attributes as well (in fact, `settings`, seen above, is simply a wrapper for this.) Thus, we can modify it at module level near the top of our fabfile like so:

```
from __future__ import with_statement
from fabric.api import *
from fabric.contrib.console import confirm

env.hosts = ['my_server']

def test():
    do_test_stuff()
```

When `fab` loads up our fabfile, our modification of `env` will execute, storing our settings change. The end result is exactly as above: our `deploy` task will run against the `my_server` server.

This is also how you can tell Fabric to run on multiple remote systems at once: because `env.hosts` is a list, `fab` iterates over it, calling the given task once for each connection.

See also:

The environment dictionary, env, *How host lists are constructed*

1.1.8 Conclusion

Our completed fabfile is still pretty short, as such things go. Here it is in its entirety:

```
from __future__ import with_statement
from fabric.api import *
from fabric.contrib.console import confirm

env.hosts = ['my_server']

def test():
    with settings(warn_only=True):
```

```
        result = local('./manage.py test my_app', capture=True)
    if result.failed and not confirm("Tests failed. Continue anyway?"):
        abort("Aborting at user request.")

def commit():
    local("git add -p && git commit")

def push():
    local("git push")

def prepare_deploy():
    test()
    commit()
    push()

def deploy():
    code_dir = '/srv/django/myproject'
    with settings(warn_only=True):
        if run("test -d %s" % code_dir).failed:
            run("git clone user@vcshost:/path/to/repo/.git %s" % code_dir)
    with cd(code_dir):
        run("git pull")
        run("touch app.wsgi")
```

This fabfile makes use of a large portion of Fabric's feature set:

- defining fabfile tasks and running them with *fab*;
- calling local shell commands with `local`;
- modifying env vars with `settings`;
- handling command failures, prompting the user, and manually aborting;
- and defining host lists and `run`-ning remote commands.

However, there's still a lot more we haven't covered here! Please make sure you follow the various "see also" links, and check out the documentation table of contents on *the main index page*.

Thanks for reading!

CHAPTER 2

Usage documentation

The following list contains all major sections of Fabric's prose (non-API) documentation, which expands upon the concepts outlined in the *Overview and Tutorial* and also covers advanced topics.

2.1 The environment dictionary, `env`

A simple but integral aspect of Fabric is what is known as the "environment": a Python dictionary subclass, which is used as a combination settings registry and shared inter-task data namespace.

The environment dict is currently implemented as a global singleton, `fabric.state.env`, and is included in `fabric.api` for convenience. Keys in `env` are sometimes referred to as "env variables".

2.1.1 Environment as configuration

Most of Fabric's behavior is controllable by modifying `env` variables, such as `env.hosts` (as seen in *the tutorial*). Other commonly-modified env vars include:

- `user`: Fabric defaults to your local username when making SSH connections, but you can use `env.user` to override this if necessary. The *Execution model* documentation also has info on how to specify usernames on a per-host basis.
- `password`: Used to explicitly set your default connection or sudo password if desired. Fabric will prompt you when necessary if this isn't set or doesn't appear to be valid.
- `warn_only`: a Boolean setting determining whether Fabric exits when detecting errors on the remote end. See *Execution model* for more on this behavior.

There are a number of other env variables; for the full list, see *Full list of env vars* at the bottom of this document.

The `settings` context manager

In many situations, it's useful to only temporarily modify `env` vars so that a given settings change only applies to a block of code. Fabric provides a `settings` context manager, which takes any number of key/value pairs and will use them to modify `env` within its wrapped block.

For example, there are many situations where setting `warn_only` (see below) is useful. To apply it to a few lines of code, use `settings(warn_only=True)`, as seen in this simplified version of the `contrib` `exists` function:

```
from fabric.api import settings, run

def exists(path):
    with settings(warn_only=True):
        return run('test -e %s' % path)
```

See the *Context Managers* API documentation for details on `settings` and other, similar tools.

2.1.2 Environment as shared state

As mentioned, the `env` object is simply a dictionary subclass, so your own fabfile code may store information in it as well. This is sometimes useful for keeping state between multiple tasks within a single execution run.

Note: This aspect of `env` is largely historical: in the past, fabfiles were not pure Python and thus the environment was the only way to communicate between tasks. Nowadays, you may call other tasks or subroutines directly, and even keep module-level shared state if you wish.

In future versions, Fabric will become threadsafe, at which point `env` may be the only easy/safe way to keep global state.

2.1.3 Other considerations

While it subclasses `dict`, Fabric's `env` has been modified so that its values may be read/written by way of attribute access, as seen in some of the above material. In other words, `env.host_string` and `env['host_string']` are functionally identical. We feel that attribute access can often save a bit of typing and makes the code more readable, so it's the recommended way to interact with `env`.

The fact that it's a dictionary can be useful in other ways, such as with Python's `dict`-based string interpolation, which is especially handy if you need to insert multiple env vars into a single string. Using "normal" string interpolation might look like this:

```
print("Executing on %s as %s" % (env.host, env.user))
```

Using dict-style interpolation is more readable and slightly shorter:

```
print("Executing on %(host)s as %(user)s" % env)
```

2.1.4 Full list of env vars

Below is a list of all predefined (or defined by Fabric itself during execution) environment variables. While many of them may be manipulated directly, it's often best to use `context_managers`, either generally via `settings` or via specific context managers such as `cd`.

Note that many of these may be set via `fab`'s command-line switches – see *fab options and arguments* for details. Cross-references are provided where appropriate.

See also:

`--set`

`abort_exception`

Default: `None`

Fabric normally handles aborting by printing an error message to stderr and calling `sys.exit(1)`. This setting allows you to override that behavior (which is what happens when `env.abort_exception` is `None`.)

Give it a callable which takes a string (the error message that would have been printed) and returns an exception instance. That exception object is then raised instead of `SystemExit` (which is what `sys.exit` does.)

Much of the time you'll want to simply set this to an exception class, as those fit the above description perfectly (callable, take a string, return an exception instance.) E.g. `env.abort_exception = MyExceptionClass`.

abort_on_prompts

Default: `False`

When `True`, Fabric will run in a non-interactive mode, calling `abort` anytime it would normally prompt the user for input (such as password prompts, "What host to connect to?" prompts, fabfile invocation of `prompt`, and so forth.) This allows users to ensure a Fabric session will always terminate cleanly instead of blocking on user input forever when unforeseen circumstances arise.

New in version 1.1.

See also:

--abort-on-prompts

all_hosts

Default: `None`

Set by `fab` to the full host list for the currently executing command. For informational purposes only.

See also:

Execution model

always_use_pty

Default: `True`

When set to `False`, causes `run`/`sudo` to act as if they have been called with `pty=False`.

See also:

--no-pty

New in version 1.0.

colorize_errors

Default `False`

When set to `True`, error output to the terminal is colored red and warnings are colored magenta to make them easier to see.

New in version 1.7.

combine_stderr

Default: `True`

Causes the SSH layer to merge a remote program's stdout and stderr streams to avoid becoming meshed together when printed. See *Combining stdout and stderr* for details on why this is needed and what its effects are.

New in version 1.0.

command

Default: `None`

Set by `fab` to the currently executing command name (e.g., when executed as `$ fab task1 task2`, `env.command` will be set to `"task1"` while `task1` is executing, and then to `"task2"`.) For informational purposes only.

See also:

Execution model

command_prefixes

Default: `[]`

Modified by `prefix`, and prepended to commands executed by `run`/`sudo`.

New in version 1.0.

command_timeout

Default: `None`

Remote command timeout, in seconds.

New in version 1.6.

See also:

`--command-timeout`

connection_attempts

Default: `1`

Number of times Fabric will attempt to connect when connecting to a new server. For backwards compatibility reasons, it defaults to only one connection attempt.

New in version 1.4.

See also:

`--connection-attempts`, *timeout*

cwd

Default: `''`

Current working directory. Used to keep state for the `cd` context manager.

dedupe_hosts

Default: `True`

Deduplicate merged host lists so any given host string is only represented once (e.g. when using combinations of `@hosts` + `@roles`, or `-H` and `-R`.)

When set to `False`, this option relaxes the deduplication, allowing users who explicitly want to run a task multiple times on the same host (say, in parallel, though it works fine serially too) to do so.

New in version 1.5.

disable_known_hosts

Default: `False`

If `True`, the SSH layer will skip loading the user's known-hosts file. Useful for avoiding exceptions in situations where a "known host" changing its host key is actually valid (e.g. cloud servers such as EC2.)

See also:

`--disable-known-hosts`, *SSH behavior*

eagerly_disconnect

Default: `False`

If `True`, causes `fab` to close connections after each individual task execution, instead of at the end of the run. This helps prevent a lot of typically-unused network sessions from piling up and causing problems with limits on per-process open files, or network hardware.

Note: When active, this setting will result in the disconnect messages appearing throughout your output, instead of at the end. This may be improved in future releases.

---

exclude_hosts

Default: `[]`

Specifies a list of host strings to be *skipped over* during `fab` execution. Typically set via *`--exclude-hosts/-x`*.

New in version 1.1.

fabfile

Default: `fabfile.py`

Filename pattern which `fab` searches for when loading fabfiles. To indicate a specific file, use the full path to the file. Obviously, it doesn't make sense to set this in a fabfile, but it may be specified in a `.fabricrc` file or on the command line.

See also:

`--fabfile`, *fab options and arguments*

gateway

Default: `None`

Enables SSH-driven gatewaying through the indicated host. The value should be a normal Fabric host string as used in e.g. *env.host_string*. When this is set, newly created connections will be set to route their SSH traffic through the remote SSH daemon to the final destination.

New in version 1.5.

See also:

--gateway

host_string

Default: `None`

Defines the current user/host/port which Fabric will connect to when executing `run`, `put` and so forth. This is set by `fab` when iterating over a previously set host list, and may also be manually set when using Fabric as a library.

See also:

Execution model

forward_agent

Default: `False`

If `True`, enables forwarding of your local SSH agent to the remote end.

New in version 1.4.

See also:

--forward-agent

host

Default: `None`

Set to the hostname part of `env.host_string` by `fab`. For informational purposes only.

hosts

Default: `[]`

The global host list used when composing per-task host lists.

See also:

--hosts, *Execution model*

keepalive

Default: `0` (i.e. no keepalive)

An integer specifying an SSH keepalive interval to use; basically maps to the SSH config option `ClientAliveInterval`. Useful if you find connections are timing out due to meddlesome network hardware or what have you.

See also:

--keepalive

New in version 1.1.

key

Default: `None`

A string, or file-like object, containing an SSH key; used during connection authentication.

Note: The most common method for using SSH keys is to set *key_filename*.

New in version 1.7.

key_filename

Default: `None`

May be a string or list of strings, referencing file paths to SSH key files to try when connecting. Passed through directly to the SSH layer. May be set/appended to with *-i*.

See also:

Paramiko's documentation for SSHClient.connect()

linewise

Default: `False`

Forces buffering by line instead of by character/byte, typically when running in parallel mode. May be activated via *--linewise*. This option is implied by *env.parallel* – even if `linewise` is False, if `parallel` is True then linewise behavior will occur.

See also:

Linewise vs bytewise output

New in version 1.3.

local_user

A read-only value containing the local system username. This is the same value as *user*'s initial value, but whereas *user* may be altered by CLI arguments, Python code or specific host strings, *local_user* will always contain the same value.

no_agent

Default: `False`

If `True`, will tell the SSH layer not to seek out running SSH agents when using key-based authentication.

New in version 0.9.1.

See also:

`--no_agent`

no_keys

Default: `False`

If `True`, will tell the SSH layer not to load any private key files from one's `$HOME/.ssh/` folder. (Key files explicitly loaded via `fab -i` will still be used, of course.)

New in version 0.9.1.

See also:

`-k`

parallel

Default: `False`

When `True`, forces all tasks to run in parallel. Implies *env.linewise*.

New in version 1.3.

See also:

`--parallel`, *Parallel execution*

password

Default: `None`

The default password used by the SSH layer when connecting to remote hosts, **and/or** when answering `sudo` prompts.

See also:

`--initial-password-prompt`, *env.passwords*, *Password management*

passwords

Default: `{}`

This dictionary is largely for internal use, and is filled automatically as a per-host-string password cache. Keys are full *host strings* and values are passwords (strings).

See also:

Password management

New in version 1.0.

path

Default: `''`

Used to set the `$PATH` shell environment variable when executing commands in `run`/`sudo`/`local`. It is recommended to use the `path` context manager for managing this value instead of setting it directly.

New in version 1.0.

pool_size

Default: `0`

Sets the number of concurrent processes to use when executing tasks in parallel.

New in version 1.3.

See also:

`--pool-size`, *Parallel execution*

port

Default: `None`

Set to the port part of `env.host_string` by `fab` when iterating over a host list. May also be used to specify a default port.

real_fabfile

Default: `None`

Set by `fab` with the path to the fabfile it has loaded up, if it got that far. For informational purposes only.

See also:

fab options and arguments

remote_interrupt

Default: `None`

Controls whether Ctrl-C triggers an interrupt remotely or is captured locally, as follows:

- `None` (the default): only `open_shell` will exhibit remote interrupt behavior, and `run`/`sudo` will capture interrupts locally.
- `False`: even `open_shell` captures locally.
- `True`: all functions will send the interrupt to the remote end.

New in version 1.6.

rcfile

Default: `$HOME/.fabricrc`

Path used when loading Fabric's local settings file.

See also:

--config, *fab options and arguments*

reject_unknown_hosts

Default: `False`

If `True`, the SSH layer will raise an exception when connecting to hosts not listed in the user's known-hosts file.

See also:

--reject-unknown-hosts, *SSH behavior*

system_known_hosts

Default: `None`

If set, should be the path to a `known_hosts` file. The SSH layer will read this file before reading the user's known-hosts file.

See also:

SSH behavior

roledefs

Default: `{}`

Dictionary defining role name to host list mappings.

See also:

Execution model

roles

Default: `[]`

The global role list used when composing per-task host lists.

See also:

--roles, *Execution model*

shell

Default: `/bin/bash -l -c`

Value used as shell wrapper when executing commands with e.g. `run`. Must be able to exist in the form `<env.shell> "<command goes here>"` – e.g. the default uses Bash's `-c` option which takes a command string as its value.

See also:

--shell, *FAQ on bash as default shell*, *Execution model*

skip_bad_hosts

Default: `False`

If `True`, causes `fab` (or non-`fab` use of `execute`) to skip over hosts it can't connect to.

New in version 1.4.

See also:

--skip-bad-hosts, *Excluding specific hosts*, *Execution model*

ssh_config_path

Default: `$HOME/.ssh/config`

Allows specification of an alternate SSH configuration file path.

New in version 1.4.

See also:

--ssh-config-path, *Leveraging native SSH config files*

ok_ret_codes

Default: `[0]`

Return codes in this list are used to determine whether calls to `run`/`sudo`/`sudo` are considered successful.

New in version 1.6.

sudo_prefix

Default: `"sudo -S -p '%(sudo_prompt)s' " % env`

The actual `sudo` command prefixed onto `sudo` calls' command strings. Users who do not have `sudo` on their default remote `$PATH`, or who need to make other changes (such as removing the `-p` when passwordless sudo is in effect) may find changing this useful.

See also:

The `sudo` operation; *env.sudo_prompt*

sudo_prompt

Default: `"sudo password:"`

Passed to the `sudo` program on remote systems so that Fabric may correctly identify its password prompt.

See also:

The `sudo` operation; *env.sudo_prefix*

sudo_user

Default: `None`

Used as a fallback value for `sudo`'s `user` argument if none is given. Useful in combination with `settings`.

See also:

`sudo`

tasks

Default: `[]`

Set by `fab` to the full tasks list to be executed for the currently executing command. For informational purposes only.

See also:

Execution model

timeout

Default: `10`

Network connection timeout, in seconds.

New in version 1.4.

See also:

`--timeout`, *connection_attempts*

use_shell

Default: `True`

Global setting which acts like the `use_shell` argument to `run`/`sudo`: if it is set to `False`, operations will not wrap executed commands in `env.shell`.

use_ssh_config

Default: `False`

Set to `True` to cause Fabric to load your local SSH config file.

New in version 1.4.

See also:

Leveraging native SSH config files

user

Default: User's local username

The username used by the SSH layer when connecting to remote hosts. May be set globally, and will be used when not otherwise explicitly set in host strings. However, when explicitly given in such a manner, this variable will be temporarily overwritten with the current value – i.e. it will always display the user currently being connected as.

To illustrate this, a fabfile:

```
from fabric.api import env, run

env.user = 'implicit_user'
env.hosts = ['host1', 'explicit_user@host2', 'host3']

def print_user():
    with hide('running'):
        run('echo "%(user)s"' % env)
```

and its use:

```
$ fab print_user

[host1] out: implicit_user
[explicit_user@host2] out: explicit_user
[host3] out: implicit_user

Done.
Disconnecting from host1... done.
Disconnecting from host2... done.
Disconnecting from host3... done.
```

As you can see, during execution on `host2`, `env.user` was set to `"explicit_user"`, but was restored to its previous value (`"implicit_user"`) afterwards.

Note: `env.user` is currently somewhat confusing (it's used for configuration **and** informational purposes) so expect this to change in the future – the informational aspect will likely be broken out into a separate env variable.

See also:

Execution model, *`--user`*

`version`

Default: current Fabric version string

Mostly for informational purposes. Modification is not recommended, but probably won't break anything either.

See also:

`--version`

`warn_only`

Default: `False`

Specifies whether or not to warn, instead of abort, when `run`/`sudo`/`local` encounter error conditions.

See also:

`--warn-only`, *Execution model*

2.2 Execution model

If you've read the *Overview and Tutorial*, you should already be familiar with how Fabric operates in the base case (a single task on a single host.) However, in many situations you'll find yourself wanting to execute multiple tasks and/or on multiple hosts. Perhaps you want to split a big task into smaller reusable parts, or crawl a collection of servers looking for an old user to remove. Such a scenario requires specific rules for when and how tasks are executed.

This document explores Fabric's execution model, including the main execution loop, how to define host lists, how connections are made, and so forth.

2.2.1 Execution strategy

Fabric defaults to a single, serial execution method, though there is an alternative parallel mode available as of Fabric 1.3 (see *Parallel execution*). This default behavior is as follows:

- A list of tasks is created. Currently this list is simply the arguments given to *fab*, preserving the order given.
- For each task, a task-specific host list is generated from various sources (see *How host lists are constructed* below for details.)
- The task list is walked through in order, and each task is run once per host in its host list.
- Tasks with no hosts in their host list are considered local-only, and will always run once and only once.

Thus, given the following fabfile:

```
from fabric.api import run, env

env.hosts = ['host1', 'host2']

def taskA():
    run('ls')

def taskB():
    run('whoami')
```

and the following invocation:

```
$ fab taskA taskB
```

you will see that Fabric performs the following:

- `taskA` executed on `host1`
- `taskA` executed on `host2`
- `taskB` executed on `host1`
- `taskB` executed on `host2`

While this approach is simplistic, it allows for a straightforward composition of task functions, and (unlike tools which push the multi-host functionality down to the individual function calls) enables shell script-like logic where you may introspect the output or return code of a given command and decide what to do next.

2.2.2 Defining tasks

For details on what constitutes a Fabric task and how to organize them, please see *Defining tasks*.

2.2.3 Defining host lists

Unless you're using Fabric as a simple build system (which is possible, but not the primary use-case) having tasks won't do you any good without the ability to specify remote hosts on which to execute them. There are a number of ways to do so, with scopes varying from global to per-task, and it's possible mix and match as needed.

Hosts

Hosts, in this context, refer to what are also called "host strings": Python strings specifying a username, hostname and port combination, in the form of `username@hostname:port`. User and/or port (and the associated `@` or `:`) may be omitted, and will be filled by the executing user's local username, and/or port 22, respectively. Thus, `admin@foo.com:222`, `deploy@website` and `nameserver1` could all be valid host strings.

IPv6 address notation is also supported, for example `::1`, `[::1]:1222`, `user@2001:db8::1` or `user@[2001:db8::1]:1222`. Square brackets are necessary only to separate the address from the port number. If no port number is used, the brackets are optional. Also if host string is specified via command-line argument, it may be necessary to escape brackets in some shells.

Note: The user/hostname split occurs at the last `@` found, so e.g. email address usernames are valid and will be parsed correctly.

During execution, Fabric normalizes the host strings given and then stores each part (username/hostname/port) in the environment dictionary, for both its use and for tasks to reference if the need arises. See *The environment dictionary, env* for details.

Roles

Host strings map to single hosts, but sometimes it's useful to arrange hosts in groups. Perhaps you have a number of Web servers behind a load balancer and want to update all of them, or want to run a task on "all client servers". Roles provide a way of defining strings which correspond to lists of host strings, and can then be specified instead of writing out the entire list every time.

This mapping is defined as a dictionary, `env.roledefs`, which must be modified by a fabfile in order to be used. A simple example:

```
from fabric.api import env

env.roledefs['webservers'] = ['www1', 'www2', 'www3']
```

Since `env.roledefs` is naturally empty by default, you may also opt to re-assign to it without fear of losing any information (provided you aren't loading other fabfiles which also modify it, of course):

```
from fabric.api import env

env.roledefs = {
    'web': ['www1', 'www2', 'www3'],
    'dns': ['ns1', 'ns2']
}
```

In addition to list/iterable object types, the values in `env.roledefs` may be callables, and will thus be called when looked up when tasks are run instead of at module load time. (For example, you could connect to remote servers to obtain role definitions, and not worry about causing delays at fabfile load time when calling e.g. `fab --list`.)

Use of roles is not required in any way – it's simply a convenience in situations where you have common groupings of servers.

Changed in version 0.9.2: Added ability to use callables as `roledefs` values.

How host lists are constructed

There are a number of ways to specify host lists, either globally or per-task, and generally these methods override one another instead of merging together (though this may change in future releases.) Each such method is typically split into two parts, one for hosts and one for roles.

Globally, via `env`

The most common method of setting hosts or roles is by modifying two key-value pairs in the environment dictionary, *env*: `hosts` and `roles`. The value of these variables is checked at runtime, while constructing each tasks's host list.

Thus, they may be set at module level, which will take effect when the fabfile is imported:

```
from fabric.api import env, run

env.hosts = ['host1', 'host2']

def mytask():
    run('ls /var/www')
```

Such a fabfile, run simply as `fab mytask`, will run `mytask` on `host1` followed by `host2`.

Since the env vars are checked for *each* task, this means that if you have the need, you can actually modify `env` in one task and it will affect all following tasks:

```
from fabric.api import env, run

def set_hosts():
    env.hosts = ['host1', 'host2']

def mytask():
    run('ls /var/www')
```

When run as `fab set_hosts mytask`, `set_hosts` is a "local" task – its own host list is empty – but `mytask` will again run on the two hosts given.

Note: This technique used to be a common way of creating fake "roles", but is less necessary now that roles are fully implemented. It may still be useful in some situations, however.

Alongside `env.hosts` is `env.roles` (not to be confused with `env.roledefs`!) which, if given, will be taken as a list of role names to look up in `env.roledefs`.

Globally, via the command line

In addition to modifying `env.hosts`, `env.roles`, and `env.exclude_hosts` at the module level, you may define them by passing comma-separated string arguments to the command-line switches *--hosts/-H* and *--roles/-R*, e.g.:

```
$ fab -H host1,host2 mytask
```

Such an invocation is directly equivalent to `env.hosts = ['host1', 'host2']` – the argument parser knows to look for these arguments and will modify `env` at parse time.

Note: It's possible, and in fact common, to use these switches to set only a single host or role. Fabric simply calls `string.split(',')` on the given string, so a string with no commas turns into a single-item list.

It is important to know that these command-line switches are interpreted **before** your fabfile is loaded: any reassignment to `env.hosts` or `env.roles` in your fabfile will overwrite them.

If you wish to nondestructively merge the command-line hosts with your fabfile-defined ones, make sure your fabfile uses `env.hosts.extend()` instead:

```
from fabric.api import env, run

env.hosts.extend(['host3', 'host4'])

def mytask():
    run('ls /var/www')
```

When this fabfile is run as `fab -H host1,host2 mytask`, `env.hosts` will then contain `['host1', 'host2', 'host3', 'host4']` at the time that `mytask` is executed.

Note: `env.hosts` is simply a Python list object – so you may use `env.hosts.append()` or any other such method you wish.

Per-task, via the command line

Globally setting host lists only works if you want all your tasks to run on the same host list all the time. This isn't always true, so Fabric provides a few ways to be more granular and specify host lists which apply to a single task only. The first of these uses task arguments.

As outlined in *fab options and arguments*, it's possible to specify per-task arguments via a special command-line syntax. In addition to naming actual arguments to your task function, this may be used to set the `host`, `hosts`, `role` or `roles` "arguments", which are interpreted by Fabric when building host lists (and removed from the arguments passed to the task itself.)

Note: Since commas are already used to separate task arguments from one another, semicolons must be used in the `hosts` or `roles` arguments to delineate individual host strings or role names. Furthermore, the argument must be quoted to prevent your shell from interpreting the semicolons.

Take the below fabfile, which is the same one we've been using, but which doesn't define any host info at all:

```
from fabric.api import run

def mytask():
    run('ls /var/www')
```

To specify per-task hosts for `mytask`, execute it like so:

```
$ fab mytask:hosts="host1;host2"
```

This will override any other host list and ensure `mytask` always runs on just those two hosts.

Per-task, via decorators

If a given task should always run on a predetermined host list, you may wish to specify this in your fabfile itself. This can be done by decorating a task function with the `hosts` or `roles` decorators. These decorators take a variable argument list, like so:

```
from fabric.api import hosts, run

@hosts('host1', 'host2')
def mytask():
    run('ls /var/www')
```

They will also take an single iterable argument, e.g.:

```
my_hosts = ('host1', 'host2')
@hosts(my_hosts)
def mytask():
    # ...
```

When used, these decorators override any checks of `env` for that particular task's host list (though `env` is not modified in any way – it is simply ignored.) Thus, even if the above fabfile had defined `env.hosts` or the call to *fab* uses *`--hosts/-H`*, `mytask` would still run on a host list of `['host1', 'host2']`.

However, decorator host lists do **not** override per-task command-line arguments, as given in the previous section.

Order of precedence

We've been pointing out which methods of setting host lists trump the others, as we've gone along. However, to make things clearer, here's a quick breakdown:

- Per-task, command-line host lists (`fab mytask:host=host1`) override absolutely everything else.
- Per-task, decorator-specified host lists (`@hosts('host1')`) override the `env` variables.
- Globally specified host lists set in the fabfile (`env.hosts = ['host1']`) *can* override such lists set on the command-line, but only if you're not careful (or want them to.)
- Globally specified host lists set on the command-line (`--hosts=host1`) will initialize the `env` variables, but that's it.

This logic may change slightly in the future to be more consistent (e.g. having *`--hosts`* somehow take precedence over `env.hosts` in the same way that command-line per-task lists trump in-code ones) but only in a backwards-incompatible release.

Combining host lists

There is no "unionizing" of hosts between the various sources mentioned in *How host lists are constructed*. If `env.hosts` is set to `['host1', 'host2', 'host3']`, and a per-function (e.g. via `hosts`) host list is set to just `['host2', 'host3']`, that function will **not** execute on `host1`, because the per-task decorator host list takes precedence.

However, for each given source, if both roles **and** hosts are specified, they will be merged together into a single host list. Take, for example, this fabfile where both of the decorators are used:

```
from fabric.api import env, hosts, roles, run

env.roledefs = {'role1': ['b', 'c']}

@hosts('a', 'b')
@roles('role1')
def mytask():
    run('ls /var/www')
```

Assuming no command-line hosts or roles are given when `mytask` is executed, this fabfile will call `mytask` on a host list of `['a', 'b', 'c']` – the union of `role1` and the contents of the `hosts` call.

Host list deduplication

By default, to support *Combining host lists*, Fabric deduplicates the final host list so any given host string is only present once. However, this prevents explicit/intentional running of a task multiple times on the same target host, which is sometimes useful.

To turn off deduplication, set *env.dedupe_hosts* to `False`.

Excluding specific hosts

At times, it is useful to exclude one or more specific hosts, e.g. to override a few bad or otherwise undesirable hosts which are pulled in from a role or an autogenerated host list.

Note: As of Fabric 1.4, you may wish to use *skip_bad_hosts* instead, which automatically skips over any unreachable hosts.

Host exclusion may be accomplished globally with *--exclude-hosts/-x*:

```
$ fab -R myrole -x host2,host5 mytask
```

If `myrole` was defined as `['host1', 'host2', ..., 'host15']`, the above invocation would run with an effective host list of `['host1', 'host3', 'host4', 'host6', ..., 'host15']`.

> **Note:** Using this option does not modify `env.hosts` – it only causes the main execution loop to skip the requested hosts.

Exclusions may be specified per-task by using an extra `exclude_hosts` kwarg, which is implemented similarly to the abovementioned `hosts` and `roles` per-task kwargs, in that it is stripped from the actual task invocation. This example would have the same result as the global exclude above:

```
$ fab mytask:roles=myrole,exclude_hosts="host2;host5"
```

Note that the host list is semicolon-separated, just as with the `hosts` per-task argument.

Combining exclusions

Host exclusion lists, like host lists themselves, are not merged together across the different "levels" they can be declared in. For example, a global `-x` option will not affect a per-task host list set with a decorator or keyword argument, nor will per-task `exclude_hosts` keyword arguments affect a global `-H` list.

There is one minor exception to this rule, namely that CLI-level keyword arguments (`mytask:exclude_hosts=x,y`) **will** be taken into account when examining host lists set via `@hosts` or `@roles`. Thus a task function decorated with `@hosts('host1', 'host2')` executed as `fab taskname:exclude_hosts=host2` will only run on `host1`.

As with the host list merging, this functionality is currently limited (partly to keep the implementation simple) and may be expanded in future releases.

2.2.4 Intelligently executing tasks with `execute`

New in version 1.3.

Most of the information here involves "top level" tasks executed via *fab*, such as the first example where we called `fab taskA taskB`. However, it's often convenient to wrap up multi-task invocations like this into their own, "meta" tasks.

Prior to Fabric 1.3, this had to be done by hand, as outlined in *Library Use*. Fabric's design eschews magical behavior, so simply *calling* a task function does **not** take into account decorators such as `roles`.

New in Fabric 1.3 is the `execute` helper function, which takes a task object or name as its first argument. Using it is effectively the same as calling the given task from the command line: all the rules given above in *How host lists are constructed* apply. (The `hosts` and `roles` keyword arguments to `execute` are analogous to *CLI per-task arguments*, including how they override all other host/role-setting methods.)

As an example, here's a fabfile defining two stand-alone tasks for deploying a Web application:

```
from fabric.api import run, roles

env.roledefs = {
    'db': ['db1', 'db2'],
    'web': ['web1', 'web2', 'web3'],
}

@roles('db')
def migrate():
    # Database stuff here.
    pass

@roles('web')
def update():
    # Code updates here.
    pass
```

In Fabric <=1.2, the only way to ensure that `migrate` runs on the DB servers and that `update` runs on the Web servers (short of manual `env.host_string` manipulation) was to call both as top level tasks:

```
$ fab migrate update
```

Fabric >=1.3 can use `execute` to set up a meta-task. Update the `import` line like so:

```
from fabric.api import run, roles, execute
```

and append this to the bottom of the file:

```
def deploy():
    execute(migrate)
    execute(update)
```

That's all there is to it; the `roles` decorators will be honored as expected, resulting in the following execution sequence:

- `migrate` on `db1`
- `migrate` on `db2`
- `update` on `web1`
- `update` on `web2`
- `update` on `web3`

Warning: This technique works because tasks that themselves have no host list (this includes the global host list settings) only run one time. If used inside a "regular" task that is going to run on multiple hosts, calls to `execute` will also run multiple times, resulting in multiplicative numbers of subtask calls – be careful!
If you would like your `execute` calls to only be called once, you may use the `runs_once` decorator.

See also:

`execute`, `runs_once`

Leveraging `execute` to access multi-host results

In nontrivial Fabric runs, especially parallel ones, you may want to gather up a bunch of per-host result values at the end - e.g. to present a summary table, perform calculations, etc.

It's not possible to do this in Fabric's default "naive" mode (one where you rely on Fabric looping over host lists on your behalf), but with `execute` it's pretty easy. Simply switch from calling the actual work-bearing task, to calling a "meta" task which takes control of execution with `execute`:

```
from fabric.api import task, execute, run, runs_once

@task
def workhorse():
    return run("get my infos")

@task
@runs_once
def go():
    results = execute(workhorse)
    print results
```

In the above, `workhorse` can do any Fabric stuff at all – it's literally your old "naive" task – except that it needs to return something useful.

`go` is your new entry point (to be invoked as `fab go`, or whatnot) and its job is to take the `results` dictionary from the `execute` call and do whatever you need with it. Check the API docs for details on the structure of that return value.

Using `execute` with dynamically-set host lists

A common intermediate-to-advanced use case for Fabric is to parameterize lookup of one's target host list at runtime (when use of *Roles* does not suffice). `execute` can make this extremely simple, like so:

```
from fabric.api import run, execute, task

# For example, code talking to an HTTP API, or a database, or ...
from mylib import external_datastore

# This is the actual algorithm involved. It does not care about host
# lists at all.
def do_work():
    run("something interesting on a host")

# This is the user-facing task invoked on the command line.
@task
def deploy(lookup_param):
    # This is the magic you don't get with @hosts or @roles.
```

```
    # Even lazy-loading roles require you to declare available roles
    # beforehand. Here, the sky is the limit.
    host_list = external_datastore.query(lookup_param)
    # Put this dynamically generated host list together with the work to be
    # done.
    execute(do_work, hosts=host_list)
```

For example, if `external_datastore` was a simplistic "look up hosts by tag in a database" service, and you wanted to run a task on all hosts tagged as being related to your application stack, you might call the above like this:

```
$ fab deploy:app
```

But wait! A data migration has gone awry on the DB servers. Let's fix up our migration code in our source repo, and deploy just the DB boxes again:

```
$ fab deploy:db
```

This use case looks similar to Fabric's roles, but has much more potential, and is by no means limited to a single argument. Define the task however you wish, query your external data store in whatever way you need – it's just Python.

The alternate approach

Similar to the above, but using `fab`'s ability to call multiple tasks in succession instead of an explicit `execute` call, is to mutate *env.hosts* in a host-list lookup task and then call `do_work` in the same session:

```
from fabric.api import run, task

from mylib import external_datastore

# Marked as a publicly visible task, but otherwise unchanged: still just
# "do the work, let somebody else worry about what hosts to run on".
@task
def do_work():
    run("something interesting on a host")

@task
def set_hosts(lookup_param):
    # Update env.hosts instead of calling execute()
    env.hosts = external_datastore.query(lookup_param)
```

Then invoke like so:

```
$ fab set_hosts:app do_work
```

One benefit of this approach over the previous one is that you can replace `do_work` with any other "workhorse" task:

```
$ fab set_hosts:db snapshot
$ fab set_hosts:cassandra,cluster2 repair_ring
$ fab set_hosts:redis,environ=prod status
```

2.2.5 Failure handling

Once the task list has been constructed, Fabric will start executing them as outlined in *Execution strategy*, until all tasks have been run on the entirety of their host lists. However, Fabric defaults to a "fail-fast" behavior pattern: if anything

goes wrong, such as a remote program returning a nonzero return value or your fabfile's Python code encountering an exception, execution will halt immediately.

This is typically the desired behavior, but there are many exceptions to the rule, so Fabric provides `env.warn_only`, a Boolean setting. It defaults to `False`, meaning an error condition will result in the program aborting immediately. However, if `env.warn_only` is set to `True` at the time of failure – with, say, the `settings` context manager – Fabric will emit a warning message but continue executing.

2.2.6 Connections

`fab` itself doesn't actually make any connections to remote hosts. Instead, it simply ensures that for each distinct run of a task on one of its hosts, the env var `env.host_string` is set to the right value. Users wanting to leverage Fabric as a library may do so manually to achieve similar effects (though as of Fabric 1.3, using `execute` is preferred and more powerful.)

`env.host_string` is (as the name implies) the "current" host string, and is what Fabric uses to determine what connections to make (or re-use) when network-aware functions are run. Operations like `run` or `put` use `env.host_string` as a lookup key in a shared dictionary which maps host strings to SSH connection objects.

Note: The connections dictionary (currently located at `fabric.state.connections`) acts as a cache, opting to return previously created connections if possible in order to save some overhead, and creating new ones otherwise.

Lazy connections

Because connections are driven by the individual operations, Fabric will not actually make connections until they're necessary. Take for example this task which does some local housekeeping prior to interacting with the remote server:

```
from fabric.api import *

@hosts('host1')
def clean_and_upload():
    local('find assets/ -name "*.DS_Store" -exec rm '{}' \;')
    local('tar czf /tmp/assets.tgz assets/')
    put('/tmp/assets.tgz', '/tmp/assets.tgz')
    with cd('/var/www/myapp/'):
        run('tar xzf /tmp/assets.tgz')
```

What happens, connection-wise, is as follows:

1. The two `local` calls will run without making any network connections whatsoever;
2. `put` asks the connection cache for a connection to `host1`;
3. The connection cache fails to find an existing connection for that host string, and so creates a new SSH connection, returning it to `put`;
4. `put` uploads the file through that connection;
5. Finally, the `run` call asks the cache for a connection to that same host string, and is given the existing, cached connection for its own use.

Extrapolating from this, you can also see that tasks which don't use any network-borne operations will never actually initiate any connections (though they will still be run once for each host in their host list, if any.)

Closing connections

Fabric's connection cache never closes connections itself – it leaves this up to whatever is using it. The *fab* tool does this bookkeeping for you: it iterates over all open connections and closes them just before it exits (regardless of whether the tasks failed or not.)

Library users will need to ensure they explicitly close all open connections before their program exits. This can be accomplished by calling `disconnect_all` at the end of your script.

Note: `disconnect_all` may be moved to a more public location in the future; we're still working on making the library aspects of Fabric more solidified and organized.

Multiple connection attempts and skipping bad hosts

As of Fabric 1.4, multiple attempts may be made to connect to remote servers before aborting with an error: Fabric will try connecting *env.connection_attempts* times before giving up, with a timeout of *env.timeout* seconds each time. (These currently default to 1 try and 10 seconds, to match previous behavior, but they may be safely changed to whatever you need.)

Furthermore, even total failure to connect to a server is no longer an absolute hard stop: set *env.skip_bad_hosts* to `True` and in most situations (typically initial connections) Fabric will simply warn and continue, instead of aborting.

New in version 1.4.

2.2.7 Password management

Fabric maintains an in-memory, two-tier password cache to help remember your login and sudo passwords in certain situations; this helps avoid tedious re-entry when multiple systems share the same password [1], or if a remote system's `sudo` configuration doesn't do its own caching.

The first layer is a simple default or fallback password cache, *env.password* (which may also be set at the command line via `--password` or `--initial-password-prompt`). This env var stores a single password which (if non-empty) will be tried in the event that the host-specific cache (see below) has no entry for the current *host string*.

env.passwords (plural!) serves as a per-user/per-host cache, storing the most recently entered password for every unique user/host/port combination. Due to this cache, connections to multiple different users and/or hosts in the same session will only require a single password entry for each. (Previous versions of Fabric used only the single, default password cache and thus required password re-entry every time the previously entered password became invalid.)

Depending on your configuration and the number of hosts your session will connect to, you may find setting either or both of these env vars to be useful. However, Fabric will automatically fill them in as necessary without any additional configuration.

Specifically, each time a password prompt is presented to the user, the value entered is used to update both the single default password cache, and the cache value for the current value of `env.host_string`.

2.2.8 Leveraging native SSH config files

Command-line SSH clients (such as the one provided by OpenSSH) make use of a specific configuration format typically known as `ssh_config`, and will read from a file in the platform-specific location `$HOME/.ssh/config` (or an arbitrary path given to `--ssh-config-path`/*env.ssh_config_path*.) This file allows specification of various SSH options such as default or per-host usernames, hostname aliases, and toggling other settings (such as whether to use *agent forwarding*.)

[1] We highly recommend the use of SSH key-based access instead of relying on homogeneous password setups, as it's significantly more secure.

Fabric's SSH implementation allows loading a subset of these options from one's actual SSH config file, should it exist. This behavior is not enabled by default (in order to be backwards compatible) but may be turned on by setting *env.use_ssh_config* to `True` at the top of your fabfile.

If enabled, the following SSH config directives will be loaded and honored by Fabric:

- `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`.

Note: If all you want to do is bounce SSH traffic off a gateway, you may find *env.gateway* to be a more efficient connection method (which will also honor more Fabric-level settings) than the typical `ssh gatewayhost nc %h %p` method of using `ProxyCommand` as a gateway.

Note: If your SSH config file contains `ProxyCommand` directives *and* you have set *env.gateway* to a non-`None` value, `env.gateway` will take precedence and the `ProxyCommand` will be ignored.

If one has a pre-created SSH config file, rationale states it will be easier for you to modify `env.gateway` (e.g. via `settings`) than to work around your conf file's contents entirely.

2.3 `fab` options and arguments

The most common method for utilizing Fabric is via its command-line tool, `fab`, which should have been placed on your shell's executable path when Fabric was installed. `fab` tries hard to be a good Unix citizen, using a standard style of command-line switches, help output, and so forth.

2.3.1 Basic use

In its most simple form, `fab` may be called with no options at all, and with one or more arguments, which should be task names, e.g.:

```
$ fab task1 task2
```

As detailed in *Overview and Tutorial* and *Execution model*, this will run `task1` followed by `task2`, assuming that Fabric was able to find a fabfile nearby containing Python functions with those names.

However, it's possible to expand this simple usage into something more flexible, by using the provided options and/or passing arguments to individual tasks.

2.3.2 Arbitrary remote shell commands

New in version 0.9.2.

Fabric leverages a lesser-known command line convention and may be called in the following manner:

```
$ fab [options] -- [shell command]
```

where everything after the `--` is turned into a temporary `run` call, and is not parsed for `fab` options. If you've defined a host list at the module level or on the command line, this usage will act like a one-line anonymous task.

For example, let's say you just wanted to get the kernel info for a bunch of systems; you could do this:

```
$ fab -H system1,system2,system3 -- uname -a
```

which would be literally equivalent to the following fabfile:

```
from fabric.api import run

def anonymous():
    run("uname -a")
```

as if it were executed thusly:

```
$ fab -H system1,system2,system3 anonymous
```

Most of the time you will want to just write out the task in your fabfile (anything you use once, you're likely to use again) but this feature provides a handy, fast way to quickly dash off an SSH-borne command while leveraging your fabfile's connection settings.

2.3.3 Command-line options

A quick overview of all possible command line options can be found via `fab --help`. If you're looking for details on a specific option, we go into detail below.

Note: `fab` uses Python's optparse library, meaning that it honors typical Linux or GNU style short and long options, as well as freely mixing options and arguments. E.g. `fab task1 -H hostname task2 -i path/to/keyfile` is just as valid as the more straightforward `fab -H hostname -i path/to/keyfile task1 task2`.

`-a, --no_agent`
Sets *env.no_agent* to `True`, forcing our SSH layer not to talk to the SSH agent when trying to unlock private key files.

New in version 0.9.1.

`-A, --forward-agent`
Sets *env.forward_agent* to `True`, enabling agent forwarding.

New in version 1.4.

`--abort-on-prompts`
Sets *env.abort_on_prompts* to `True`, forcing Fabric to abort whenever it would prompt for input.

New in version 1.1.

`-c` RCFILE, `--config`=RCFILE
Sets *env.rcfile* to the given file path, which Fabric will try to load on startup and use to update environment variables.

-d COMMAND, **--display**=COMMAND
Prints the entire docstring for the given task, if there is one. Does not currently print out the task's function signature, so descriptive docstrings are a good idea. (They're *always* a good idea, of course – just moreso here.)

--connection-attempts=M, **-n** M
Set number of times to attempt connections. Sets *env.connection_attempts*.

See also:

env.connection_attempts, *env.timeout*

New in version 1.4.

-D, --disable-known-hosts
Sets *env.disable_known_hosts* to `True`, preventing Fabric from loading the user's SSH `known_hosts` file.

-f FABFILE, **--fabfile**=FABFILE
The fabfile name pattern to search for (defaults to `fabfile.py`), or alternately an explicit file path to load as the fabfile (e.g. `/path/to/my/fabfile.py`.)

See also:

Fabfile construction and use

-F LIST_FORMAT, **--list-format**=LIST_FORMAT
Allows control over the output format of *--list*. `short` is equivalent to *--shortlist*, `normal` is the same as simply omitting this option entirely (i.e. the default), and `nested` prints out a nested namespace tree.

New in version 1.1.

See also:

--shortlist, *--list*

-g HOST, **--gateway**=HOST
Sets *env.gateway* to `HOST` host string.

New in version 1.5.

-h, --help
Displays a standard help message, with all possible options and a brief overview of what they do, then exits.

--hide=LEVELS
A comma-separated list of *output levels* to hide by default.

-H HOSTS, **--hosts**=HOSTS
Sets *env.hosts* to the given comma-delimited list of host strings.

-x HOSTS, **--exclude-hosts**=HOSTS
Sets *env.exclude_hosts* to the given comma-delimited list of host strings to then keep out of the final host list.

New in version 1.1.

-i KEY_FILENAME
When set to a file path, will load the given file as an SSH identity file (usually a private key.) This option may be repeated multiple times. Sets (or appends to) *env.key_filename*.

-I, --initial-password-prompt
Forces a password prompt at the start of the session (after fabfile load and option parsing, but before executing any tasks) in order to pre-fill *env.password*.

This is useful for fire-and-forget runs (especially parallel sessions, in which runtime input is not possible) when setting the password via *--password* or by setting *env.password* in your fabfile, is undesirable.

Note: The value entered into this prompt will *overwrite* anything supplied via *env.password* at module level, or

via *--password*.

See also:

Password management

-k
Sets *env.no_keys* to `True`, forcing the SSH layer to not look for SSH private key files in one's home directory.

New in version 0.9.1.

--keepalive=KEEPALIVE
Sets *env.keepalive* to the given (integer) value, specifying an SSH keepalive interval.

New in version 1.1.

--linewise
Forces output to be buffered line-by-line instead of byte-by-byte. Often useful or required for *parallel execution*.

New in version 1.3.

-l, --list
Imports a fabfile as normal, but then prints a list of all discovered tasks and exits. Will also print the first line of each task's docstring, if it has one, next to it (truncating if necessary.)

Changed in version 0.9.1: Added docstring to output.

See also:

--shortlist, *--list-format*

-p PASSWORD, **--password**=PASSWORD
Sets *env.password* to the given string; it will then be used as the default password when making SSH connections or calling the `sudo` program.

See also:

--initial-password-prompt

-P, --parallel
Sets *env.parallel* to `True`, causing tasks to run in parallel.

New in version 1.3.

See also:

Parallel execution

--no-pty
Sets *env.always_use_pty* to `False`, causing all `run`/`sudo` calls to behave as if one had specified `pty=False`.

New in version 1.0.

-r, --reject-unknown-hosts
Sets *env.reject_unknown_hosts* to `True`, causing Fabric to abort when connecting to hosts not found in the user's SSH `known_hosts` file.

-R ROLES, **--roles**=ROLES
Sets *env.roles* to the given comma-separated list of role names.

--set KEY=VALUE,...
Allows you to set default values for arbitrary Fabric env vars. Values set this way have a low precedence – they will not override more specific env vars which are also specified on the command line. E.g.:

```
fab --set password=foo --password=bar
```

will result in `env.password = 'bar'`, not `'foo'`

Multiple `KEY=VALUE` pairs may be comma-separated, e.g. `fab --set var1=val1,var2=val2`.

Other than basic string values, you may also set env vars to True by omitting the `=VALUE` (e.g. `fab --set KEY`), and you may set values to the empty string (and thus a False-equivalent value) by keeping the equals sign, but omitting `VALUE` (e.g. `fab --set KEY=`.)

New in version 1.4.

-s SHELL, **--shell**=SHELL
Sets *env.shell* to the given string, overriding the default shell wrapper used to execute remote commands.

--shortlist
Similar to *--list*, but without any embellishment, just task names separated by newlines with no indentation or docstrings.

New in version 0.9.2.

See also:

--list

--show=LEVELS
A comma-separated list of *output levels* to be added to those that are shown by default.

See also:

`run`, `sudo`

--ssh-config-path
Sets *env.ssh_config_path*.

New in version 1.4.

See also:

Leveraging native SSH config files

--skip-bad-hosts
Sets *env.skip_bad_hosts*, causing Fabric to skip unavailable hosts.

New in version 1.4.

--timeout=N, **-t** N
Set connection timeout in seconds. Sets *env.timeout*.

See also:

env.timeout, *env.connection_attempts*

New in version 1.4.

--command-timeout=N, **-T** N
Set remote command timeout in seconds. Sets *env.command_timeout*.

See also:

env.command_timeout,

New in version 1.6.

-u USER, **--user**=USER
Sets *env.user* to the given string; it will then be used as the default username when making SSH connections.

-V, --version
Displays Fabric's version number, then exits.

-w, --warn-only
Sets *env.warn_only* to `True`, causing Fabric to continue execution even when commands encounter error conditions.

-z, --pool-size
Sets *env.pool_size*, which specifies how many processes to run concurrently during parallel execution.

New in version 1.3.

See also:

Parallel execution

2.3.4 Per-task arguments

The options given in *Command-line options* apply to the invocation of `fab` as a whole; even if the order is mixed around, options still apply to all given tasks equally. Additionally, since tasks are just Python functions, it's often desirable to pass in arguments to them at runtime.

Answering both these needs is the concept of "per-task arguments", which is a special syntax you can tack onto the end of any task name:

- Use a colon (`:`) to separate the task name from its arguments;
- Use commas (`,`) to separate arguments from one another (may be escaped by using a backslash, i.e. `\,`);
- Use equals signs (=) for keyword arguments, or omit them for positional arguments. May also be escaped with backslashes.

Additionally, since this process involves string parsing, all values will end up as Python strings, so plan accordingly. (We hope to improve upon this in future versions of Fabric, provided an intuitive syntax can be found.)

For example, a "create a new user" task might be defined like so (omitting most of the actual logic for brevity):

```
def new_user(username, admin='no', comment="No comment provided"):
    log_action("New User (%s): %s" % (username, comment))
    pass
```

You can specify just the username:

```
$ fab new_user:myusername
```

Or treat it as an explicit keyword argument:

```
$ fab new_user:username=myusername
```

If both args are given, you can again give them as positional args:

```
$ fab new_user:myusername,yes
```

Or mix and match, just like in Python:

```
$ fab new_user:myusername,admin=yes
```

The `log_action` call above is useful for illustrating escaped commas, like so:

```
$ fab new_user:myusername,admin=no,comment='Gary\, new developer (starts Monday)'
```

Note: Quoting the backslash-escaped comma is required, as not doing so will cause shell syntax errors. Quotes are also needed whenever an argument involves other shell-related characters such as spaces.

All of the above are translated into the expected Python function calls. For example, the last call above would become:

```
>>> new_user('myusername', admin='yes', comment='Gary, new developer (starts Monday)')
```

Roles and hosts

As mentioned in *the section on task execution*, there are a handful of per-task keyword arguments (`host`, `hosts`, `role` and `roles`) which do not actually map to the task functions themselves, but are used for setting per-task host and/or role lists.

These special kwargs are **removed** from the args/kwargs sent to the task function itself; this is so that you don't run into TypeErrors if your task doesn't define the kwargs in question. (It also means that if you **do** define arguments with these names, you won't be able to specify them in this manner – a regrettable but necessary sacrifice.)

Note: If both the plural and singular forms of these kwargs are given, the value of the plural will win out and the singular will be discarded.

When using the plural form of these arguments, one must use semicolons (`;`) since commas are already being used to separate arguments from one another. Furthermore, since your shell is likely to consider semicolons a special character, you'll want to quote the host list string to prevent shell interpretation, e.g.:

```
$ fab new_user:myusername,hosts="host1;host2"
```

Again, since the `hosts` kwarg is removed from the argument list sent to the `new_user` task function, the actual Python invocation would be `new_user('myusername')`, and the function would be executed on a host list of `['host1', 'host2']`.

2.3.5 Settings files

Fabric currently honors a simple user settings file, or `fabricrc` (think `bashrc` but for `fab`) which should contain one or more key-value pairs, one per line. These lines will be subject to `string.split('=')`, and thus can currently only be used to specify string settings. Any such key-value pairs will be used to update *env* when `fab` runs, and is loaded prior to the loading of any fabfile.

By default, Fabric looks for `~/.fabricrc`, and this may be overridden by specifying the *-c* flag to `fab`.

For example, if your typical SSH login username differs from your workstation username, and you don't want to modify `env.user` in a project's fabfile (possibly because you expect others to use it as well) you could write a `fabricrc` file like so:

```
user = ssh_user_name
```

Then, when running `fab`, your fabfile would load up with `env.user` set to `'ssh_user_name'`. Other users of that fabfile could do the same, allowing the fabfile itself to be cleanly agnostic regarding the default username.

2.4 Fabfile construction and use

This document contains miscellaneous sections about fabfiles, both how to best write them, and how to use them once written.

2.4.1 Fabfile discovery

Fabric is capable of loading Python modules (e.g. `fabfile.py`) or packages (e.g. a `fabfile/` directory containing an `__init__.py`). By default, it looks for something named (to Python's import machinery) `fabfile` - so either `fabfile/` or `fabfile.py`.

The fabfile discovery algorithm searches in the invoking user's current working directory or any parent directories. Thus, it is oriented around "project" use, where one keeps e.g. a `fabfile.py` at the root of a source code tree. Such a fabfile will then be discovered no matter where in the tree the user invokes `fab`.

The specific name to be searched for may be overridden on the command-line with the `-f` option, or by adding a *fabricrc* line which sets the value of `fabfile`. For example, if you wanted to name your fabfile `fab_tasks.py`, you could create such a file and then call `fab -f fab_tasks.py <task name>`, or add `fabfile = fab_tasks.py` to `~/.fabricrc`.

If the given fabfile name contains path elements other than a filename (e.g. `../fabfile.py` or `/dir1/dir2/custom_fabfile`) it will be treated as a file path and directly checked for existence without any sort of searching. When in this mode, tilde-expansion will be applied, so one may refer to e.g. `~/personal_fabfile.py`.

Note: Fabric does a normal `import` (actually an `__import__`) of your fabfile in order to access its contents – it does not do any `eval`-ing or similar. In order for this to work, Fabric temporarily adds the found fabfile's containing folder to the Python load path (and removes it immediately afterwards.)

Changed in version 0.9.2: The ability to load package fabfiles.

2.4.2 Importing Fabric

Because Fabric is just Python, you *can* import its components any way you want. However, for the purposes of encapsulation and convenience (and to make life easier for Fabric's packaging script) Fabric's public API is maintained in the `fabric.api` module.

All of Fabric's *Operations*, *Context Managers*, *Decorators* and *Utils* are included in this module as a single, flat namespace. This enables a very simple and consistent interface to Fabric within your fabfiles:

```
from fabric.api import *

# call run(), sudo(), etc etc
```

This is not technically best practices (for a number of reasons) and if you're only using a couple of Fab API calls, it *is* probably a good idea to explicitly `from fabric.api import env, run` or similar. However, in most nontrivial fabfiles, you'll be using all or most of the API, and the star import:

```
from fabric.api import *
```

will be a lot easier to write and read than:

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

so in this case we feel pragmatism overrides best practices.

2.4.3 Defining tasks and importing callables

For important information on what exactly Fabric will consider as a task when it loads your fabfile, as well as notes on how best to import other code, please see *Defining tasks* in the *Execution model* documentation.

2.5 Interaction with remote programs

Fabric's primary operations, `run` and `sudo`, are capable of sending local input to the remote end, in a manner nearly identical to the `ssh` program. For example, programs which display password prompts (e.g. a database dump utility, or changing a user's password) will behave just as if you were interacting with them directly.

However, as with `ssh` itself, Fabric's implementation of this feature is subject to a handful of limitations which are not always intuitive. This document discusses such issues in detail.

Note: Readers unfamiliar with the basics of Unix stdout and stderr pipes, and/or terminal devices, may wish to visit the Wikipedia pages for Unix pipelines and Pseudo terminals respectively.

2.5.1 Combining stdout and stderr

The first issue to be aware of is that of the stdout and stderr streams, and why they are separated or combined as needed.

Buffering

Fabric 0.9.x and earlier, and Python itself, buffer output on a line-by-line basis: text is not printed to the user until a newline character is found. This works fine in most situations but becomes problematic when one needs to deal with partial-line output such as prompts.

Note: Line-buffered output can make programs appear to halt or freeze for no reason, as prompts print out text without a newline, waiting for the user to enter their input and press Return.

Newer Fabric versions buffer both input and output on a character-by-character basis in order to make interaction with prompts possible. This has the convenient side effect of enabling interaction with complex programs utilizing the "curses" libraries or which otherwise redraw the screen (think `top`).

Crossing the streams

Unfortunately, printing to stderr and stdout simultaneously (as many programs do) means that when the two streams are printed independently one byte at a time, they can become garbled or meshed together. While this can sometimes be mitigated by line-buffering one of the streams and not the other, it's still a serious issue.

To solve this problem, Fabric uses a setting in our SSH layer which merges the two streams at a low level and causes output to appear more naturally. This setting is represented in Fabric as the *combine_stderr* env var and keyword argument, and is `True` by default.

Due to this default setting, output will appear correctly, but at the cost of an empty `.stderr` attribute on the return values of `run`/`sudo`, as all output will appear to be stdout.

Conversely, users requiring a distinct stderr stream at the Python level and who aren't bothered by garbled user-facing output (or who are hiding stdout and stderr from the command in question) may opt to set this to `False` as needed.

2.5.2 Pseudo-terminals

The other main issue to consider when presenting interactive prompts to users is that of echoing the user's own input.

Echoes

Typical terminal applications or bona fide text terminals (e.g. when using a Unix system without a running GUI) present programs with a terminal device called a tty or pty (for pseudo-terminal). These automatically echo all text typed into them back out to the user (via stdout), as interaction without seeing what you had just typed would be difficult. Terminal devices are also able to conditionally turn off echoing, allowing secure password prompts.

However, it's possible for programs to be run without a tty or pty present at all (consider cron jobs, for example) and in this situation, any stdin data being fed to the program won't be echoed. This is desirable for programs being run without any humans around, and it's also Fabric's old default mode of operation.

Fabric's approach

Unfortunately, in the context of executing commands via Fabric, when no pty is present to echo a user's stdin, Fabric must echo it for them. This is sufficient for many applications, but it presents problems for password prompts, which become insecure.

In the interests of security and meeting the principle of least surprise (insofar as users are typically expecting things to behave as they would when run in a terminal emulator), Fabric 1.0 and greater force a pty by default. With a pty enabled, Fabric simply allows the remote end to handle echoing or hiding of stdin and does not echo anything itself.

Note: In addition to allowing normal echo behavior, a pty also means programs that behave differently when attached to a terminal device will then do so. For example, programs that colorize output on terminals but not when run in the background will print colored output. Be wary of this if you inspect the return value of `run` or `sudo`!

For situations requiring the pty behavior turned off, the `--no-pty` command-line argument and *always_use_pty* env var may be used.

2.5.3 Combining the two

As a final note, keep in mind that use of pseudo-terminals effectively implies combining stdout and stderr – in much the same way as the *combine_stderr* setting does. This is because a terminal device naturally sends both stdout and stderr to the same place – the user's display – thus making it impossible to differentiate between them.

However, at the Fabric level, the two groups of settings are distinct from one another and may be combined in various ways. The default is for both to be set to `True`; the other combinations are as follows:

- `run("cmd", pty=False, combine_stderr=True)`: will cause Fabric to echo all stdin itself, including passwords, as well as potentially altering `cmd`'s behavior. Useful if `cmd` behaves undesirably when run under a pty and you're not concerned about password prompts.
- `run("cmd", pty=False, combine_stderr=False)`: with both settings `False`, Fabric will echo stdin and won't issue a pty – and this is highly likely to result in undesired behavior for all but the simplest commands. However, it is also the only way to access a distinct stderr stream, which is occasionally useful.
- `run("cmd", pty=True, combine_stderr=False)`: valid, but won't really make much of a difference, as `pty=True` will still result in merged streams. May be useful for avoiding any edge case problems in `combine_stderr` (none are presently known).

2.6 Library Use

Fabric's primary use case is via fabfiles and the *fab* tool, and this is reflected in much of the documentation. However, Fabric's internals are written in such a manner as to be easily used without `fab` or fabfiles at all – this document will show you how.

There's really only a couple of considerations one must keep in mind, when compared to writing a fabfile and using `fab` to run it: how connections are really made, and how disconnections occur.

2.6.1 Connections

We've documented how Fabric really connects to its hosts before, but it's currently somewhat buried in the middle of the overall *execution docs*. Specifically, you'll want to skip over to the *Connections* section and read it real quick. (You should really give that entire document a once-over, but it's not absolutely required.)

As that section mentions, the key is simply that `run`, `sudo` and the other operations only look in one place when connecting: *env.host_string*. All of the other mechanisms for setting hosts are interpreted by the `fab` tool when it runs, and don't matter when running as a library.

That said, most use cases where you want to marry a given task `X` and a given list of hosts `Y` can, as of Fabric 1.3, be handled with the `execute` function via `execute(X, hosts=Y)`. Please see `execute`'s documentation for details – manual host string manipulation should be rarely necessary.

2.6.2 Disconnecting

The other main thing that `fab` does for you is to disconnect from all hosts at the end of a session; otherwise, Python will sit around forever waiting for those network resources to be released.

Fabric 0.9.4 and newer have a function you can use to do this easily: `disconnect_all`. Simply make sure your code calls this when it terminates (typically in the `finally` clause of an outer `try: finally` statement – lest errors in your code prevent disconnections from happening!) and things ought to work pretty well.

If you're on Fabric 0.9.3 or older, you can simply do this (`disconnect_all` just adds a bit of nice output to this logic):

```
from fabric.state import connections

for key in connections.keys():
    connections[key].close()
    del connections[key]
```

2.6.3 Final note

This document is an early draft, and may not cover absolutely every difference between `fab` use and library use. However, the above should highlight the largest stumbling blocks. When in doubt, note that in the Fabric source code, `fabric/main.py` contains the bulk of the extra work done by `fab`, and may serve as a useful reference.

2.7 Managing output

The `fab` tool is very verbose by default and prints out almost everything it can, including the remote end's stderr and stdout streams, the command strings being executed, and so forth. While this is necessary in many cases in order to know just what's going on, any nontrivial Fabric task will quickly become difficult to follow as it runs.

2.7.1 Output levels

To aid in organizing task output, Fabric output is grouped into a number of non-overlapping levels or groups, each of which may be turned on or off independently. This provides flexible control over what is displayed to the user.

Note: All levels, save for `debug`, are on by default.

Standard output levels

The standard, atomic output levels/groups are as follows:

- **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.

Changed in version 0.9.2: Added "Executing task" lines to the `running` output level.

Changed in version 0.9.2: Added the `user` output level.

Debug output

There is a final atomic output level, `debug`, which behaves slightly differently from the rest:

- **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:

  ```
  run('ls "/home/username/Folder Name With Spaces/"')
  ```

  Normally, the `running` line will show exactly what is passed into `run`, like so:

  ```
  [hostname] run: ls "/home/username/Folder Name With Spaces/"
  ```

  With `debug` on, and assuming you've left *shell* set to `True`, you will see the literal, full string as passed to the remote server:

  ```
  [hostname] run: /bin/bash -l -c "ls \"/home/username/Folder Name With Spaces\""
  ```

  Enabling `debug` output will also display full Python tracebacks during aborts.

  Note: Where modifying other pieces of output (such as in the above example where it modifies the 'running' line to show the shell and any escape characters), this setting takes precedence over the others; so if `running` is False but `debug` is True, you will still be shown the 'running' line in its debugging form.

Changed in version 1.0: Debug output now includes full Python tracebacks during aborts.

Output level aliases

In addition to the atomic/standalone levels above, Fabric also provides a couple of convenience aliases which map to multiple other levels. These may be referenced anywhere the other levels are referenced, and will effectively toggle all of the levels they are mapped to.

- **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.

Changed in version 1.4: Added the `commands` output alias.

2.7.2 Hiding and/or showing output levels

You may toggle any of Fabric's output levels in a number of ways; for examples, please see the API docs linked in each bullet point:

- **Direct modification of fabric.state.output**: `fabric.state.output` is a dictionary subclass (similar to *env*) whose keys are the output level names, and whose values are either True (show that particular type of output) or False (hide it.)

  `fabric.state.output` is the lowest-level implementation of output levels and is what Fabric's internals reference when deciding whether or not to print their output.

- **Context managers**: `hide` and `show` are twin context managers that take one or more output level names as strings, and either hide or show them within the wrapped block. As with Fabric's other context managers, the prior values are restored when the block exits.

  See also:

  `settings`, which can nest calls to `hide` and/or `show` inside itself.

- **Command-line arguments**: You may use the *`--hide`* and/or *`--show`* arguments to *fab options and arguments*, which behave exactly like the context managers of the same names (but are, naturally, globally applied) and take comma-separated strings as input.

2.8 Parallel execution

New in version 1.3.

By default, Fabric executes all specified tasks **serially** (see *Execution strategy* for details.) This document describes Fabric's options for running tasks on multiple hosts in **parallel**, via per-task decorators and/or global command-line switches.

2.8.1 What it does

Because Fabric 1.x is not fully threadsafe (and because in general use, task functions do not typically interact with one another) this functionality is implemented via the Python multiprocessing module. It creates one new process for each host and task combination, optionally using a (configurable) sliding window to prevent too many processes from running at the same time.

For example, imagine a scenario where you want to update Web application code on a number of Web servers, and then reload the servers once the code has been distributed everywhere (to allow for easier rollback if code updates fail.) One could implement this with the following fabfile:

```
from fabric.api import *

def update():
    with cd("/srv/django/myapp"):
        run("git pull")

def reload():
    sudo("service apache2 reload")
```

and execute it on a set of 3 servers, in serial, like so:

```
$ fab -H web1,web2,web3 update reload
```

Normally, without any parallel execution options activated, Fabric would run in order:

1. `update` on `web1`
2. `update` on `web2`
3. `update` on `web3`
4. `reload` on `web1`
5. `reload` on `web2`
6. `reload` on `web3`

With parallel execution activated (via *-P* – see below for details), this turns into:

1. `update` on `web1`, `web2`, and `web3`
2. `reload` on `web1`, `web2`, and `web3`

Hopefully the benefits of this are obvious – if `update` took 5 seconds to run and `reload` took 2 seconds, serial execution takes (5+2)*3 = 21 seconds to run, while parallel execution takes only a third of the time, (5+2) = 7 seconds on average.

2.8.2 How to use it

Decorators

Since the minimum "unit" that parallel execution affects is a task, the functionality may be enabled or disabled on a task-by-task basis using the `parallel` and `serial` decorators. For example, this fabfile:

```
from fabric.api import *

@parallel
def runs_in_parallel():
    pass

def runs_serially():
    pass
```

when run in this manner:

```
$ fab -H host1,host2,host3 runs_in_parallel runs_serially
```

will result in the following execution sequence:

1. `runs_in_parallel` on `host1`, `host2`, and `host3`
2. `runs_serially` on `host1`
3. `runs_serially` on `host2`
4. `runs_serially` on `host3`

Command-line flags

One may also force all tasks to run in parallel by using the command-line flag `-P` or the env variable *env.parallel*. However, any task specifically wrapped with `serial` will ignore this setting and continue to run serially.

For example, the following fabfile will result in the same execution sequence as the one above:

```
from fabric.api import *

def runs_in_parallel():
    pass

@serial
def runs_serially():
    pass
```

when invoked like so:

```
$ fab -H host1,host2,host3 -P runs_in_parallel runs_serially
```

As before, `runs_in_parallel` will run in parallel, and `runs_serially` in sequence.

2.8.3 Bubble size

With large host lists, a user's local machine can get overwhelmed by running too many concurrent Fabric processes. Because of this, you may opt to use a moving bubble approach that limits Fabric to a specific number of concurrently active processes.

By default, no bubble is used and all hosts are run in one concurrent pool. You can override this on a per-task level by specifying the `pool_size` keyword argument to `parallel`, or globally via `-z`.

For example, to run on 5 hosts at a time:

```
from fabric.api import *

@parallel(pool_size=5)
def heavy_task():
    # lots of heavy local lifting or lots of IO here
```

Or skip the `pool_size` kwarg and instead:

```
$ fab -P -z 5 heavy_task
```

2.8.4 Linewise vs bytewise output

Fabric's default mode of printing to the terminal is byte-by-byte, in order to support *Interaction with remote programs*. This often gives poor results when running in parallel mode, as the multiple processes may write to your terminal's standard out stream simultaneously.

To help offset this problem, Fabric's option for linewise output is automatically enabled whenever parallelism is active. This will cause you to lose most of the benefits outlined in the above link Fabric's remote interactivity features, but as those do not map well to parallel invocations, it's typically a fair trade.

There's no way to avoid the multiple processes mixing up on a line-by-line basis, but you will at least be able to tell them apart by the host-string line prefix.

Note: Future versions will add improved logging support to make troubleshooting parallel runs easier.

2.9 SSH behavior

Fabric currently makes use of a pure-Python SSH re-implementation for managing connections, meaning that there are occasionally spots where it is limited by that library's capabilities. Below are areas of note where Fabric will exhibit behavior that isn't consistent with, or as flexible as, the behavior of the `ssh` command-line program.

2.9.1 Unknown hosts

SSH's host key tracking mechanism keeps tabs on all the hosts you attempt to connect to, and maintains a `~/.ssh/known_hosts` file with mappings between identifiers (IP address, sometimes with a hostname as well) and SSH keys. (For details on how this works, please see the OpenSSH documentation.)

The `paramiko` library is capable of loading up your `known_hosts` file, and will then compare any host it connects to, with that mapping. Settings are available to determine what happens when an unknown host (a host whose username or IP is not found in `known_hosts`) is seen:

- **Reject**: the host key is rejected and the connection is not made. This results in a Python exception, which will terminate your Fabric session with a message that the host is unknown.
- **Add**: the new host key is added to the in-memory list of known hosts, the connection is made, and things continue normally. Note that this does **not** modify your on-disk `known_hosts` file!
- **Ask**: not yet implemented at the Fabric level, this is a `paramiko` library option which would result in the user being prompted about the unknown key and whether to accept it.

Whether to reject or add hosts, as above, is controlled in Fabric via the *env.reject_unknown_hosts* option, which is False by default for convenience's sake. We feel this is a valid tradeoff between convenience and security; anyone who feels otherwise can easily modify their fabfiles at module level to set `env.reject_unknown_hosts = True`.

2.9.2 Known hosts with changed keys

The point of SSH's key/fingerprint tracking is so that man-in-the-middle attacks can be detected: if an attacker redirects your SSH traffic to a computer under his control, and pretends to be your original destination server, the host keys will not match. Thus, the default behavior of SSH (and its Python implementation) is to immediately abort the connection when a host previously recorded in `known_hosts` suddenly starts sending us a different host key.

In some edge cases such as some EC2 deployments, you may want to ignore this potential problem. Our SSH layer, at the time of writing, doesn't give us control over this exact behavior, but we can sidestep it by simply skipping the loading of `known_hosts` – if the host list being compared to is empty, then there's no problem. Set *env.disable_known_hosts* to True when you want this behavior; it is False by default, in order to preserve default SSH behavior.

Warning: Enabling *env.disable_known_hosts* will leave you wide open to man-in-the-middle attacks! Please use with caution.

2.10 Defining tasks

As of Fabric 1.1, there are two distinct methods you may use in order to define which objects in your fabfile show up as tasks:

- The "new" method starting in 1.1 considers instances of `Task` or its subclasses, and also descends into imported modules to allow building nested namespaces.
- The "classic" method from 1.0 and earlier considers all public callable objects (functions, classes etc) and only considers the objects in the fabfile itself with no recursing into imported module.

Note: These two methods are **mutually exclusive**: if Fabric finds *any* new-style task objects in your fabfile or in modules it imports, it will assume you've committed to this method of task declaration and won't consider any non-`Task` callables. If *no* new-style tasks are found, it reverts to the classic behavior.

The rest of this document explores these two methods in detail.

Note: To see exactly what tasks in your fabfile may be executed via `fab`, use *`fab --list`*.

2.10.1 New-style tasks

Fabric 1.1 introduced the `Task` class to facilitate new features and enable some programming best practices, specifically:

- **Object-oriented tasks**. Inheritance and all that comes with it can make for much more sensible code reuse than passing around simple function objects. The classic style of task declaration didn't entirely rule this out, but it also didn't make it terribly easy.
- **Namespaces**. Having an explicit method of declaring tasks makes it easier to set up recursive namespaces without e.g. polluting your task list with the contents of Python's `os` module (which would show up as valid "tasks" under the classic methodology.)

With the introduction of `Task`, there are two ways to set up new tasks:

- Decorate a regular module level function with `@task`, which transparently wraps the function in a `Task` subclass. The function name will be used as the task name when invoking.
- Subclass `Task` (`Task` itself is intended to be abstract), define a `run` method, and instantiate your subclass at module level. Instances' `name` attributes are used as the task name; if omitted the instance's variable name will be used instead.

Use of new-style tasks also allows you to set up *namespaces*.

The `@task` decorator

The quickest way to make use of new-style task features is to wrap basic task functions with `@task`:

```
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.)

Arguments

`@task` may also be called with arguments to customize its behavior. Any arguments not documented below are passed into the constructor of the `task_class` being used, with the function itself as the first argument (see *Using custom subclasses with @task* for details.)

- `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.)

Aliases

Here's a quick example of using the `alias` keyword argument to facilitate use of both a longer human-readable task name, and a shorter name which is quicker to type:

```
from fabric.api import task

@task(alias='dwm')
def deploy_with_migrations():
    pass
```

Calling `--list` on this fabfile would show both the original `deploy_with_migrations` and its alias `dwm`:

```
$ fab --list
Available commands:

    deploy_with_migrations
    dwm
```

When more than one alias for the same function is needed, simply swap in the `aliases` kwarg, which takes an iterable of strings instead of a single string.

Default tasks

In a similar manner to *aliases*, it's sometimes useful to designate a given task within a module as the "default" task, which may be called by referencing *just* the module name. This can save typing and/or allow for neater organization when there's a single "main" task and a number of related tasks or subroutines.

For example, a `deploy` submodule might contain tasks for provisioning new servers, pushing code, migrating databases, and so forth – but it'd be very convenient to highlight a task as the default "just deploy" action. Such a `deploy.py` module might look like this:

```
from fabric.api import task

@task
def migrate():
```

```
    pass

@task
def push():
    pass

@task
def provision():
    pass

@task
def full_deploy():
    if not provisioned:
        provision()
    push()
    migrate()
```

With the following task list (assuming a simple top level `fabfile.py` that just imports `deploy`):

```
$ fab --list
Available commands:

    deploy.full_deploy
    deploy.migrate
    deploy.provision
    deploy.push
```

Calling `deploy.full_deploy` on every deploy could get kind of old, or somebody new to the team might not be sure if that's really the right task to run.

Using the `default` kwarg to `@task`, we can tag e.g. `full_deploy` as the default task:

```
@task(default=True)
def full_deploy():
    pass
```

Doing so updates the task list like so:

```
$ fab --list
Available commands:

    deploy
    deploy.full_deploy
    deploy.migrate
    deploy.provision
    deploy.push
```

Note that `full_deploy` still exists as its own explicit task – but now `deploy` shows up as a sort of top level alias for `full_deploy`.

If multiple tasks within a module have `default=True` set, the last one to be loaded (typically the one lowest down in the file) will take precedence.

Top-level default tasks

Using `@task(default=True)` in the top level fabfile will cause the denoted task to execute when a user invokes `fab` without any task names (similar to e.g. `make`.) When using this shortcut, it is not possible to specify arguments to the task itself – use a regular invocation of the task if this is necessary.

`Task` subclasses

If you're used to *classic-style tasks*, an easy way to think about `Task` subclasses is that their `run` method is directly equivalent to a classic task; its arguments are the task arguments (other than `self`) and its body is what gets executed.

For example, this new-style task:

```
class MyTask(Task):
    name = "deploy"
    def run(self, environment, domain="whatever.com"):
        run("git clone foo")
        sudo("service apache2 restart")

instance = MyTask()
```

is exactly equivalent to this function-based task:

```
@task
def deploy(environment, domain="whatever.com"):
    run("git clone foo")
    sudo("service apache2 restart")
```

Note how we had to instantiate an instance of our class; that's simply normal Python object-oriented programming at work. While it's a small bit of boilerplate right now – for example, Fabric doesn't care about the name you give the instantiation, only the instance's `name` attribute – it's well worth the benefit of having the power of classes available.

We plan to extend the API in the future to make this experience a bit smoother.

Using custom subclasses with `@task`

It's possible to marry custom `Task` subclasses with `@task`. This may be useful in cases where your core execution logic doesn't do anything class/object-specific, but you want to take advantage of class metaprogramming or similar techniques.

Specifically, any `Task` subclass which is designed to take in a callable as its first constructor argument (as the built-in `WrappedCallableTask` does) may be specified as the `task_class` argument to `@task`.

Fabric will automatically instantiate a copy of the given class, passing in the wrapped function as the first argument. All other args/kwargs given to the decorator (besides the "special" arguments documented in *Arguments*) are added afterwards.

Here's a brief and somewhat contrived example to make this obvious:

```
from fabric.api import task
from fabric.tasks import Task

class CustomTask(Task):
    def __init__(self, func, myarg, *args, **kwargs):
        super(CustomTask, self).__init__(*args, **kwargs)
        self.func = func
        self.myarg = myarg

    def run(self, *args, **kwargs):
        return self.func(*args, **kwargs)

@task(task_class=CustomTask, myarg='value', alias='at')
def actual_task():
    pass
```

When this fabfile is loaded, a copy of `CustomTask` is instantiated, effectively calling:

```
task_obj = CustomTask(actual_task, myarg='value')
```

Note how the `alias` kwarg is stripped out by the decorator itself and never reaches the class instantiation; this is identical in function to how *command-line task arguments* work.

Namespaces

With *classic tasks*, fabfiles were limited to a single, flat set of task names with no real way to organize them. In Fabric 1.1 and newer, if you declare tasks the new way (via `@task` or your own `Task` subclass instances) you may take advantage of **namespacing**:

- Any module objects imported into your fabfile will be recursed into, looking for additional task objects.
- Within submodules, you may control which objects are "exported" by using the standard Python `__all__` module-level variable name (thought they should still be valid new-style task objects.)
- These tasks will be given new dotted-notation names based on the modules they came from, similar to Python's own import syntax.

Let's build up a fabfile package from simple to complex and see how this works.

Basic

We start with a single `__init__.py` containing a few tasks (the Fabric API import omitted for brevity):

```
@task
def deploy():
    ...

@task
def compress():
    ...
```

The output of `fab --list` would look something like this:

```
deploy
compress
```

There's just one namespace here: the "root" or global namespace. Looks simple now, but in a real-world fabfile with dozens of tasks, it can get difficult to manage.

Importing a submodule

As mentioned above, Fabric will examine any imported module objects for tasks, regardless of where that module exists on your Python import path. For now we just want to include our own, "nearby" tasks, so we'll make a new submodule in our package for dealing with, say, load balancers – `lb.py`:

```
@task
def add_backend():
    ...
```

And we'll add this to the top of `__init__.py`:

```
import lb
```

Now `fab --list` shows us:

```
deploy
compress
lb.add_backend
```

Again, with only one task in its own submodule, it looks kind of silly, but the benefits should be pretty obvious.

Going deeper

Namespacing isn't limited to just one level. Let's say we had a larger setup and wanted a namespace for database related tasks, with additional differentiation inside that. We make a sub-package named `db/` and inside it, a `migrations.py` module:

```
@task
def list():
    ...

@task
def run():
    ...
```

We need to make sure that this module is visible to anybody importing `db`, so we add it to the sub-package's `__init__.py`:

```
import migrations
```

As a final step, we import the sub-package into our root-level `__init__.py`, so now its first few lines look like this:

```
import lb
import db
```

After all that, our file tree looks like this:

```
.
-- __init__.py
-- db
|   -- __init__.py
|   -- migrations.py
-- lb.py
```

and `fab --list` shows:

```
deploy
compress
lb.add_backend
db.migrations.list
db.migrations.run
```

We could also have specified (or imported) tasks directly into `db/__init__.py`, and they would show up as `db.<whatever>` as you might expect.

Limiting with `__all__`

You may limit what Fabric "sees" when it examines imported modules, by using the Python convention of a module level `__all__` variable (a list of variable names.) If we didn't want the `db.migrations.run` task to show up by default for some reason, we could add this to the top of `db/migrations.py`:

```
__all__ = ['list']
```

Note the lack of `'run'` there. You could, if needed, import `run` directly into some other part of the hierarchy, but otherwise it'll remain hidden.

Switching it up

We've been keeping our fabfile package neatly organized and importing it in a straightforward manner, but the filesystem layout doesn't actually matter here. All Fabric's loader cares about is the names the modules are given when they're imported.

For example, if we changed the top of our root `__init__.py` to look like this:

```
import db as database
```

Our task list would change thusly:

```
deploy
compress
lb.add_backend
database.migrations.list
database.migrations.run
```

This applies to any other import – you could import third party modules into your own task hierarchy, or grab a deeply nested module and make it appear near the top level.

Nested list output

As a final note, we've been using the default Fabric *`--list`* output during this section – it makes it more obvious what the actual task names are. However, you can get a more nested or tree-like view by passing `nested` to the *`--list-format`* option:

```
$ fab --list-format=nested --list
Available commands (remember to call as module.[...].task):

    deploy
    compress
    lb:
        add_backend
    database:
        migrations:
            list
            run
```

While it slightly obfuscates the "real" task names, this view provides a handy way of noting the organization of tasks in large namespaces.

2.10.2 Classic tasks

When no new-style `Task`-based tasks are found, Fabric will consider any callable object found in your fabfile, **except** the following:

- Callables whose name starts with an underscore (_). In other words, Python's usual "private" convention holds true here.

- Callables defined within Fabric itself. Fabric's own functions such as `run` and `sudo` will not show up in your task list.

Imports

Python's `import` statement effectively includes the imported objects in your module's namespace. Since Fabric's fabfiles are just Python modules, this means that imports are also considered as possible classic-style tasks, alongside anything defined in the fabfile itself.

> **Note:** This only applies to imported *callable objects* – not modules. Imported modules only come into play if they contain *new-style tasks*, at which point this section no longer applies.

Because of this, we strongly recommend that you use the `import module` form of importing, followed by `module.callable()`, which will result in a cleaner fabfile API than doing `from module import callable`.

For example, here's a sample fabfile which uses `urllib.urlopen` to get some data out of a webservice:

```
from urllib import urlopen

from fabric.api import run

def webservice_read():
    objects = urlopen('http://my/web/service/?foo=bar').read().split()
    print(objects)
```

This looks simple enough, and will run without error. However, look what happens if we run *`fab --list`* on this fabfile:

```
$ fab --list
Available commands:

  webservice_read   List some directories.
  urlopen           urlopen(url [, data]) -> open file-like object
```

Our fabfile of only one task is showing two "tasks", which is bad enough, and an unsuspecting user might accidentally try to call `fab urlopen`, which probably won't work very well. Imagine any real-world fabfile, which is likely to be much more complex, and hopefully you can see how this could get messy fast.

For reference, here's the recommended way to do it:

```
import urllib

from fabric.api import run

def webservice_read():
    objects = urllib.urlopen('http://my/web/service/?foo=bar').read().split()
    print(objects)
```

It's a simple change, but it'll make anyone using your fabfile a bit happier.

CHAPTER 3

API documentation

Fabric maintains two sets of API documentation, autogenerated from the source code's docstrings (which are typically very thorough.)

3.1 Core API

The **core** API is loosely defined as those functions, classes and methods which form the basic building blocks of Fabric (such as `run` and `sudo`) upon which everything else (the below "contrib" section, and user fabfiles) builds.

3.1.1 Color output functions

New in version 0.9.2.

Functions for wrapping strings in ANSI color codes.

Each function within this module returns the input string `text`, wrapped with ANSI color codes for the appropriate color.

For example, to print some text as green on supporting terminals:

```
from fabric.colors import green

print(green("This text is green!"))
```

Because these functions simply return modified strings, you can nest them:

```
from fabric.colors import red, green

print(red("This sentence is red, except for " +           green("these words, which are green") + ".'
```

If `bold` is set to `True`, the ANSI flag for bolding will be flipped on for that particular invocation, which usually shows up as a bold or brighter version of the original color on most terminals.

`fabric.colors.`**`blue`**(*text*, *bold=False*)

`fabric.colors.`**`cyan`**(*text*, *bold=False*)

`fabric.colors.`**`green`**(*text*, *bold=False*)

`fabric.colors.`**`magenta`**(*text*, *bold=False*)

`fabric.colors.`**`red`**(*text*, *bold=False*)

`fabric.colors.`**`white`**(*text*, *bold=False*)

fabric.colors.**yellow**(*text*, *bold=False*)

3.1.2 Context Managers

Context managers for use with the `with` statement.

Note: When using Python 2.5, you will need to start your fabfile with `from __future__ import with_statement` in order to make use of the `with` statement (which is a regular, non `__future__` feature of Python 2.6+.)

Note: If you are using multiple directly nested `with` statements, it can be convenient to use multiple context expressions in one single with statement. Instead of writing:

```
with cd('/path/to/app'):
    with prefix('workon myvenv'):
        run('./manage.py syncdb')
        run('./manage.py loaddata myfixture')
```

you can write:

```
with cd('/path/to/app'), prefix('workon myvenv'):
    run('./manage.py syncdb')
    run('./manage.py loaddata myfixture')
```

Note that you need Python 2.7+ for this to work. On Python 2.5 or 2.6, you can do the following:

```
from contextlib import nested

with nested(cd('/path/to/app'), prefix('workon myvenv')):
    ...
```

Finally, note that `settings` implements `nested` itself – see its API doc for details.

---

fabric.context_managers.**cd**(*path*)

Context manager that keeps directory state when calling remote operations.

Any calls to `run`, `sudo`, `get`, or `put` within the wrapped block will implicitly have a string similar to `"cd <path> && "` prefixed in order to give the sense that there is actually statefulness involved.

Note: `cd` only affects *remote* paths – to modify *local* paths, use `lcd`.

Because use of `cd` affects all such invocations, any code making use of those operations, such as much of the `contrib` section, will also be affected by use of `cd`.

Like the actual 'cd' shell builtin, `cd` may be called with relative paths (keep in mind that your default starting directory is your remote user's `$HOME`) and may be nested as well.

Below is a "normal" attempt at using the shell 'cd', which doesn't work due to how shell-less SSH connections are implemented – state is **not** kept between invocations of `run` or `sudo`:

```
run('cd /var/www')
run('ls')
```

The above snippet will list the contents of the remote user's `$HOME` instead of `/var/www`. With `cd`, however, it will work as expected:

```
with cd('/var/www'):
    run('ls') # Turns into "cd /var/www && ls"
```

Finally, a demonstration (see inline comments) of nesting:

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

Note: This context manager is currently implemented by appending to (and, as always, restoring afterwards) the current value of an environment variable, `env.cwd`. However, this implementation may change in the future, so we do not recommend manually altering `env.cwd` – only the *behavior* of `cd` will have any guarantee of backwards compatibility.

Note: Space characters will be escaped automatically to make dealing with such directory names easier.

Changed in version 1.0: Applies to `get` and `put` in addition to the command-running operations.

See also:

`lcd`

`fabric.context_managers.`**`char_buffered`**(**args*, ***kwds*)

Force local terminal `pipe` be character, not line, buffered.

Only applies on Unix-based systems; on Windows this is a no-op.

`fabric.context_managers.`**`hide`**(**args*, ***kwds*)

Context manager for setting the given output `groups` to False.

`groups` must be one or more strings naming the output groups defined in `output`. The given groups will be set to False for the duration of the enclosed block, and restored to their previous value afterwards.

For example, to hide the "[hostname] run:" status lines, as well as preventing printout of stdout and stderr, one might use `hide` as follows:

```
def my_task():
    with hide('running', 'stdout', 'stderr'):
        run('ls /var/www')
```

`fabric.context_managers.`**`lcd`**(*path*)

Context manager for updating local current working directory.

This context manager is identical to `cd`, except that it changes a different env var (`lcwd`, instead of `cwd`) and thus only affects the invocation of `local` and the local arguments to `get`/`put`.

Relative path arguments are relative to the local user's current working directory, which will vary depending on where Fabric (or Fabric-using code) was invoked. You can check what this is with os.getcwd. It may be useful to pin things relative to the location of the fabfile in use, which may be found in *env.real_fabfile*

New in version 1.0.

`fabric.context_managers.`**`path`**(*path*, *behavior='append'*)

Append the given `path` to the PATH used to execute any wrapped commands.

Any calls to `run` or `sudo` within the wrapped block will implicitly have a string similar to `"PATH=$PATH:<path> "` prepended before the given command.

You may customize the behavior of `path` by specifying the optional `behavior` keyword argument, as follows:

- 'append': append given path to the current $PATH, e.g. PATH=$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=<path>.

Note: This context manager is currently implemented by modifying (and, as always, restoring afterwards) the current value of environment variables, env.path and env.path_behavior. However, this implementation may change in the future, so we do not recommend manually altering them directly.

New in version 1.0.

fabric.context_managers.**prefix**(*command*)

Prefix all wrapped run/sudo commands with given command plus &&.

This is nearly identical to cd, except that nested invocations append to a list of command strings instead of modifying a single string.

Most of the time, you'll want to be using this alongside a shell script which alters shell state, such as ones which export or alter shell environment variables.

For example, one of the most common uses of this tool is with the workon command from virtualenvwrapper:

```
with prefix('workon myvenv'):
    run('./manage.py syncdb')
```

In the above snippet, the actual shell command run would be this:

```
$ workon myvenv && ./manage.py syncdb
```

This context manager is compatible with cd, so if your virtualenv doesn't cd in its postactivate script, you could do the following:

```
with cd('/path/to/app'):
    with prefix('workon myvenv'):
        run('./manage.py syncdb')
        run('./manage.py loaddata myfixture')
```

Which would result in executions like so:

```
$ cd /path/to/app && workon myvenv && ./manage.py syncdb
$ cd /path/to/app && workon myvenv && ./manage.py loaddata myfixture
```

Finally, as alluded to near the beginning, prefix may be nested if desired, e.g.:

```
with prefix('workon myenv'):
    run('ls')
    with prefix('source /some/script'):
        run('touch a_file')
```

The result:

```
$ workon myenv && ls
$ workon myenv && source /some/script && touch a_file
```

Contrived, but hopefully illustrative.

fabric.context_managers.**quiet**()

Alias to settings(hide('everything'), warn_only=True).

Useful for wrapping remote interrogative commands which you expect to fail occasionally, and/or which you want to silence.

Example:

```
with quiet():
    have_build_dir = run("test -e /tmp/build").succeeded
```

When used in a task, the above snippet will not produce any `run: test -e /tmp/build` line, nor will any stdout/stderr display, and command failure is ignored.

See also:

env.warn_only, `settings`, `hide`

New in version 1.5.

`fabric.context_managers.`**`remote_tunnel`**`(*args, **kwds)`

Create a tunnel forwarding a locally-visible port to the remote target.

For example, you can let the remote host access a database that is installed on the client host:

```
# Map localhost:6379 on the server to localhost:6379 on the client,
# so that the remote 'redis-cli' program ends up speaking to the local
# redis-server.
with remote_tunnel(6379):
    run("redis-cli -i")
```

The database might be installed on a client only reachable from the client host (as opposed to *on* the client itself):

```
# Map localhost:6379 on the server to redis.internal:6379 on the client
with remote_tunnel(6379, local_host="redis.internal")
    run("redis-cli -i")
```

`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.

Note: By default, most SSH servers only allow remote tunnels to listen to the localhost interface (127.0.0.1). In these cases, `remote_bind_address` is ignored by the server, and the tunnel will listen only to 127.0.0.1.

`fabric.context_managers.`**`settings`**`(*args, **kwargs)`

Nest context managers and/or override `env` variables.

`settings` serves two purposes:

- Most usefully, it allows temporary overriding/updating of `env` with any provided keyword arguments, e.g. `with settings(user='foo'):`. Original values, if any, will be restored once the `with` block closes.
  - The keyword argument `clean_revert` has special meaning for `settings` itself (see below) and will be stripped out before execution.

- In addition, it will use contextlib.nested to nest any given non-keyword arguments, which should be other context managers, e.g. `with settings(hide('stderr'), show('stdout')):`.

These behaviors may be specified at the same time if desired. An example will hopefully illustrate why this is considered useful:

```
def my_task():
    with settings(
        hide('warnings', 'running', 'stdout', 'stderr'),
        warn_only=True
    ):
        if run('ls /etc/lsb-release'):
            return 'Ubuntu'
        elif run('ls /etc/redhat-release'):
            return 'RedHat'
```

The above task executes a `run` statement, but will warn instead of aborting if the `ls` fails, and all output – including the warning itself – is prevented from printing to the user. The end result, in this scenario, is a completely silent task that allows the caller to figure out what type of system the remote host is, without incurring the handful of output that would normally occur.

Thus, `settings` may be used to set any combination of environment variables in tandem with hiding (or showing) specific levels of output, or in tandem with any other piece of Fabric functionality implemented as a context manager.

If `clean_revert` is set to `True`, `settings` will **not** revert keys which are altered within the nested block, instead only reverting keys whose values remain the same as those given. More examples will make this clear; below is how `settings` operates normally:

```
# Before the block, env.parallel defaults to False, host_string to None
with settings(parallel=True, host_string='myhost'):
    # env.parallel is True
    # env.host_string is 'myhost'
    env.host_string = 'otherhost'
    # env.host_string is now 'otherhost'
# Outside the block:
# * env.parallel is False again
# * env.host_string is None again
```

The internal modification of `env.host_string` is nullified – not always desirable. That's where `clean_revert` comes in:

```
# Before the block, env.parallel defaults to False, host_string to None
with settings(parallel=True, host_string='myhost', clean_revert=True):
    # env.parallel is True
    # env.host_string is 'myhost'
    env.host_string = 'otherhost'
    # env.host_string is now 'otherhost'
# Outside the block:
# * env.parallel is False again
# * env.host_string remains 'otherhost'
```

Brand new keys which did not exist in `env` prior to using `settings` are also preserved if `clean_revert` is active. When `False`, such keys are removed when the block exits.

New in version 1.4.1: The `clean_revert` kwarg.

`fabric.context_managers.`**`shell_env`**`(**kw)`

Set shell environment variables for wrapped commands.

For example, the below shows how you might set a ZeroMQ related environment variable when installing a Python ZMQ library:

```
with shell_env(ZMQ_DIR='/home/user/local'):
    run('pip install pyzmq')
```

As with `prefix`, this effectively turns the `run` command into:

```
$ export ZMQ_DIR='/home/user/local' && pip install pyzmq
```

Multiple key-value pairs may be given simultaneously.

Note: If used to affect the behavior of `local` when running from a Windows localhost, `SET` commands will be used to implement this feature.

`fabric.context_managers.`**`show`**(**args*, ***kwds*)

Context manager for setting the given output `groups` to True.

`groups` must be one or more strings naming the output groups defined in `output`. The given groups will be set to True for the duration of the enclosed block, and restored to their previous value afterwards.

For example, to turn on debug output (which is typically off by default):

```
def my_task():
    with show('debug'):
        run('ls /var/www')
```

As almost all output groups are displayed by default, `show` is most useful for turning on the normally-hidden `debug` group, or when you know or suspect that code calling your own code is trying to hide output with `hide`.

`fabric.context_managers.`**`warn_only`**()

Alias to `settings(warn_only=True)`.

See also:

env.warn_only, `settings`, `quiet`

3.1.3 Decorators

Convenience decorators for use in fabfiles.

`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
```

`hosts` may be invoked with either an argument list (`@hosts('host1')`, `@hosts('host1', 'host2')`) or a single, iterable argument (`@hosts(['host1', 'host2'])`).

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

Changed in version 0.9.2: Allow a single, iterable argument (`@hosts(iterable)`) to be used instead of requiring `@hosts(*iterable)`.

fabric.decorators.**parallel**(*pool_size=None*)
Forces the wrapped function to run in parallel, instead of sequentially.

This decorator takes precedence over the global value of *env.parallel*. It also takes precedence over `serial` if a task is decorated with both.

New in version 1.3.

fabric.decorators.**roles**(**role_list*)
Decorator defining a list of role names, used to look up host lists.

A role is simply defined as a key in `env` whose value is a list of one or more host connection strings. For example, the following will ensure that, barring an override on the command line, `my_func` will be executed against the hosts listed in the `webserver` and `dbserver` roles:

```
env.roledefs.update({
    'webserver': ['www1', 'www2'],
    'dbserver': ['db1']
})

@roles('webserver', 'dbserver')
def my_func():
    pass
```

As with `hosts`, `roles` may be invoked with either an argument list or a single, iterable argument. Similarly, this decorator uses the same mechanism as `hosts` and simply sets `<function>.roles`.

Changed in version 0.9.2: Allow a single, iterable argument to be used (same as `hosts`).

fabric.decorators.**runs_once**(*func*)
Decorator preventing wrapped function from running more than once.

By keeping internal state, this decorator allows you to mark a function such that it will only run once per Python interpreter session, which in typical use means "once per invocation of the `fab` program".

Any function wrapped with this decorator will silently fail to execute the 2nd, 3rd, ..., Nth time it is called, and will return the value of the original run.

Note: `runs_once` does not work with parallel task execution.

---

fabric.decorators.**serial**(*func*)
Forces the wrapped function to always run sequentially, never in parallel.

This decorator takes precedence over the global value of *env.parallel*. However, if a task is decorated with both `serial` *and* `parallel`, `parallel` wins.

New in version 1.3.

fabric.decorators.**task**(**args*, ***kwargs*)
Decorator declaring the wrapped function to be a new-style task.

May be invoked as a simple, argument-less decorator (i.e. `@task`) or with arguments customizing its behavior (e.g. `@task(alias='myalias')`).

Please see the *new-style task* documentation for details on how to use this decorator.

Changed in version 1.2: Added the `alias`, `aliases`, `task_class` and `default` keyword arguments. See *Arguments* for details.

Changed in version 1.5: Added the `name` keyword argument.

See also:

`unwrap_tasks`, `WrappedCallableTask`

fabric.decorators.**with_settings**(**arg_settings*, ***kw_settings*)
Decorator equivalent of fabric.context_managers.settings.

Allows you to wrap an entire function as if it was called inside a block with the settings context manager. This may be useful if you know you want a given setting applied to an entire function body, or wish to retrofit old code without indenting everything.

For example, to turn aborts into warnings for an entire task function:

```
@with_settings(warn_only=True)
def foo():
    ...
```

See also:

settings

New in version 1.1.

3.1.4 Documentation helpers

fabric.docs.**unwrap_tasks**(*module*, *hide_nontasks=False*)
Replace task objects on module with their wrapped functions instead.

Specifically, look for instances of WrappedCallableTask and replace them with their .wrapped attribute (the original decorated function.)

This is intended for use with the Sphinx autodoc tool, to be run near the bottom of a project's conf.py. It ensures that the autodoc extension will have full access to the "real" function, in terms of function signature and so forth. Without use of unwrap_tasks, autodoc is unable to access the function signature (though it is able to see e.g. __doc__.)

For example, at the bottom of your conf.py:

```
from fabric.docs import unwrap_tasks
import my_package.my_fabfile
unwrap_tasks(my_package.my_fabfile)
```

You can go above and beyond, and explicitly **hide** all non-task functions, by saying hide_nontasks=True. This renames all objects failing the "is it a task?" check so they appear to be private, which will then cause autodoc to skip over them.

hide_nontasks is thus useful when you have a fabfile mixing in subroutines with real tasks and want to document *just* the real tasks.

If you run this within an actual Fabric-code-using session (instead of within a Sphinx conf.py), please seek immediate medical attention.

See also:

WrappedCallableTask, task

3.1.5 Network

Classes and subroutines dealing with network connections and related topics.

fabric.network.**disconnect_all**()
Disconnect from all currently connected servers.

Used at the end of fab's main loop, and also intended for use by library users.

class fabric.network.**HostConnectionCache**
Dict subclass allowing for caching of host connections/clients.

This subclass will intelligently create new client connections when keys are requested, or return previously created connections instead.

It also handles creating new socket-like objects when required to implement gateway connections and `ProxyCommand`, and handing them to the inner connection methods.

Key values are the same as host specifiers throughout Fabric: optional username + `@`, mandatory hostname, optional `:` + port number. Examples:

- `example.com` - typical Internet host address.
- `firewall` - atypical, but still legal, local host address.
- `user@example.com` - with specific username attached.
- `bob@smith.org:222` - with specific nonstandard port attached.

When the username is not given, `env.user` is used. `env.user` defaults to the currently running user at startup but may be overwritten by user code or by specifying a command-line flag.

Note that differing explicit usernames for the same hostname will result in multiple client connections being made. For example, specifying `user1@example.com` will create a connection to `example.com`, logged in as `user1`; later specifying `user2@example.com` will create a new, 2nd connection as `user2`.

The same applies to ports: specifying two different ports will result in two different connections to the same host being made. If no port is given, 22 is assumed, so `example.com` is equivalent to `example.com:22`.

__getitem__(*key*)
Autoconnect + return connection object

__weakref__
list of weak references to the object (if defined)

connect(*key*)
Force a new connection to `key` host string.

fabric.network.**connect**(*user*, *host*, *port*, *cache*, *seek_gateway=True*)
Create and return a new SSHClient instance connected to given host.

Parameters

- **user** – Username to connect as.
- **host** – Network hostname.
- **port** – SSH daemon port.
- **cache** – A `HostConnectionCache` instance used to cache/store gateway hosts when gatewaying is enabled.
- **seek_gateway** – Whether to try setting up a gateway socket for this connection. Used so the actual gateway connection can prevent recursion.

fabric.network.**denormalize**(*host_string*)
Strips out default values for the given host string.

If the user part is the default user, it is removed; if the port is port 22, it also is removed.

fabric.network.**disconnect_all**()
Disconnect from all currently connected servers.

Used at the end of `fab`'s main loop, and also intended for use by library users.

fabric.network.**get_gateway**(*host*, *port*, *cache*, *replace=False*)
Create and return a gateway socket, if one is needed.

This function checks `env` for gateway or proxy-command settings and returns the necessary socket-like object for use by a final host connection.

Parameters

- **host** – Hostname of target server.
- **port** – Port to connect to on target server.
- **cache** – A `HostConnectionCache` object, in which gateway `SSHClient` objects are to be retrieved/cached.
- **replace** – Whether to forcibly replace a cached gateway client object.

Returns A `socket.socket`-like object, or `None` if none was created.

fabric.network.**join_host_strings**(*user*, *host*, *port=None*)
Turns user/host/port strings into `user@host:port` combined string.

This function is not responsible for handling missing user/port strings; for that, see the `normalize` function.

If `host` looks like IPv6 address, it will be enclosed in square brackets

If `port` is omitted, the returned string will be of the form `user@host`.

fabric.network.**key_filenames**()
Returns list of SSH key filenames for the current env.host_string.

Takes into account ssh_config and env.key_filename, including normalization to a list. Also performs `os.path.expanduser` expansion on any key filenames.

fabric.network.**key_from_env**(*passphrase=None*)
Returns a paramiko-ready key from a text string of a private key

fabric.network.**needs_host**(*func*)
Prompt user for value of `env.host_string` when `env.host_string` is empty.

This decorator is basically a safety net for silly users who forgot to specify the host/host list in one way or another. It should be used to wrap operations which require a network connection.

Due to how we execute commands per-host in `main()`, it's not possible to specify multiple hosts at this point in time, so only a single host will be prompted for.

Because this decorator sets `env.host_string`, it will prompt once (and only once) per command. As `main()` clears `env.host_string` between commands, this decorator will also end up prompting the user once per command (in the case where multiple commands have no hosts set, of course.)

fabric.network.**normalize**(*host_string*, *omit_port=False*)
Normalizes a given host string, returning explicit host, user, port.

If `omit_port` is given and is True, only the host and user are returned.

This function will process SSH config files if Fabric is configured to do so, and will use them to fill in some default values or swap in hostname aliases.

fabric.network.**normalize_to_string**(*host_string*)
normalize() returns a tuple; this returns another valid host string.

fabric.network.**prompt_for_password**(*prompt=None*, *no_colon=False*, *stream=None*)
Prompts for and returns a new password if required; otherwise, returns None.

A trailing colon is appended unless `no_colon` is True.

If the user supplies an empty password, the user will be re-prompted until they enter a non-empty password.

`prompt_for_password` autogenerates the user prompt based on the current host being connected to. To override this, specify a string value for `prompt`.

`stream` is the stream the prompt will be printed to; if not given, defaults to `sys.stderr`.

`fabric.network.`**`ssh_config`**(*host_string=None*)

Return ssh configuration dict for current env.host_string host value.

Memoizes the loaded SSH config file, but not the specific per-host results.

This function performs the necessary "is SSH config enabled?" checks and will simply return an empty dict if not. If SSH config *is* enabled and the value of env.ssh_config_path is not a valid file, it will abort.

May give an explicit host string as `host_string`.

3.1.6 Operations

Functions to be used in fabfiles and other non-core code, such as run()/sudo().

`fabric.operations.`**`get`**(**args*, ***kwargs*)

Download one or more files from a remote host.

`get` returns an iterable containing the absolute paths to all local files downloaded, which will be empty if `local_path` was a StringIO object (see below for more on using StringIO). This object will also exhibit a `.failed` attribute containing any remote file paths which failed to download, and a `.succeeded` attribute equivalent to `not .failed`.

`remote_path` is the remote file or directory path to download, which may contain shell glob syntax, e.g. `"/var/log/apache2/*.log"`, and will have tildes replaced by the remote home directory. Relative paths will be considered relative to the remote user's home directory, or the current remote working directory as manipulated by `cd`. If the remote path points to a directory, that directory will be downloaded recursively.

`local_path` is the local file path where the downloaded file or files will be stored. If relative, it will honor the local current working directory as manipulated by `lcd`. It may be interpolated, using standard Python dict-based interpolation, with the following variables:

- `host`: The value of `env.host_string`, eg `myhostname` or `user@myhostname-222` (the colon between hostname and port is turned into a dash to maximize filesystem compatibility)
- `dirname`: The directory part of the remote file path, e.g. the `src/projectname` in `src/projectname/utils.py`.
- `basename`: The filename part of the remote file path, e.g. the `utils.py` in `src/projectname/utils.py`
- `path`: The full remote path, e.g. `src/projectname/utils.py`.

Note: When `remote_path` is an absolute directory path, only the inner directories will be recreated locally and passed into the above variables. So for example, `get('/var/log', '%(path)s')` would start writing out files like `apache2/access.log`, `postgresql/8.4/postgresql.log`, etc, in the local working directory. It would **not** write out e.g. `var/log/apache2/access.log`.

Additionally, when downloading a single file, `%(dirname)s` and `%(path)s` do not make as much sense and will be empty and equivalent to `%(basename)s`, respectively. Thus a call like `get('/var/log/apache2/access.log', '%(path)s')` will save a local file named `access.log`, not `var/log/apache2/access.log`.

This behavior is intended to be consistent with the command-line `scp` program.

If left blank, `local_path` defaults to `"%(host)s/%(path)s"` in order to be safe for multi-host invocations.

Warning: If your `local_path` argument does not contain `%(host)s` and your `get` call runs against multiple hosts, your local files will be overwritten on each successive run!

If `local_path` does not make use of the above variables (i.e. if it is a simple, explicit file path) it will act similar to `scp` or `cp`, overwriting pre-existing files if necessary, downloading into a directory if given (e.g. `get('/path/to/remote_file.txt', 'local_directory')` will create `local_directory/remote_file.txt`) and so forth.

`local_path` may alternately be a file-like object, such as the result of `open('path', 'w')` or a `StringIO` instance.

Note: Attempting to `get` a directory into a file-like object is not valid and will result in an error.

Note: This function will use `seek` and `tell` to overwrite the entire contents of the file-like object, in order to be consistent with the behavior of `put` (which also considers the entire file). However, unlike `put`, the file pointer will not be restored to its previous location, as that doesn't make as much sense here and/or may not even be possible.

Note: If a file-like object such as StringIO has a `name` attribute, that will be used in Fabric's printed output instead of the default `<file obj>`

Changed in version 1.0: Now honors the remote working directory as manipulated by `cd`, and the local working directory as manipulated by `lcd`.

Changed in version 1.0: Now allows file-like objects in the `local_path` argument.

Changed in version 1.0: `local_path` may now contain interpolated path- and host-related variables.

Changed in version 1.0: Directories may be specified in the `remote_path` argument and will trigger recursive downloads.

Changed in version 1.0: Return value is now an iterable of downloaded local file paths, which also exhibits the `.failed` and `.succeeded` attributes.

Changed in version 1.5: Allow a `name` attribute on file-like objects for log output

`fabric.operations.`**`local`**(*command*, *capture=False*, *shell=None*)

Run a command on the local system.

`local` is simply a convenience wrapper around the use of the builtin Python `subprocess` module with `shell=True` activated. If you need to do anything special, consider using the `subprocess` module directly.

`shell` is passed directly to subprocess.Popen's `execute` argument (which determines the local shell to use.) As per the linked documentation, on Unix the default behavior is to use `/bin/sh`, so this option is useful for setting that value to e.g. `/bin/bash`.

`local` is not currently capable of simultaneously printing and capturing output, as `run`/`sudo` do. The `capture` kwarg allows you to switch between printing and capturing as necessary, and defaults to `False`.

When `capture=False`, the local subprocess' stdout and stderr streams are hooked up directly to your terminal, though you may use the global *output controls* `output.stdout` and `output.stderr` to hide one or both if desired. In this mode, the return value's stdout/stderr values are always empty.

When `capture=True`, you will not see any output from the subprocess in your terminal, but the return value will contain the captured stdout/stderr.

In either case, as with run and sudo, this return value exhibits the return_code, stderr, failed and succeeded attributes. See run for details.

local will honor the lcd context manager, allowing you to control its current working directory independently of the remote end (which honors cd).

Changed in version 1.0: Added the succeeded and stderr attributes.

Changed in version 1.0: Now honors the lcd context manager.

Changed in version 1.0: Changed the default value of capture from True to False.

fabric.operations.**open_shell**(**args*, ***kwargs*)

Invoke a fully interactive shell on the remote end.

If command is given, it will be sent down the pipe before handing control over to the invoking user.

This function is most useful for when you need to interact with a heavily shell-based command or series of commands, such as when debugging or when fully interactive recovery is required upon remote program failure.

It should be considered an easy way to work an interactive shell session into the middle of a Fabric script and is *not* a drop-in replacement for run, which is also capable of interacting with the remote end (albeit only while its given command is executing) and has much stronger programmatic abilities such as error handling and stdout/stderr capture.

Specifically, open_shell provides a better interactive experience than run, but use of a full remote shell prevents Fabric from determining whether programs run within the shell have failed, and pollutes the stdout/stderr stream with shell output such as login banners, prompts and echoed stdin.

Thus, this function does not have a return value and will not trigger Fabric's failure handling if any remote programs result in errors.

New in version 1.0.

fabric.operations.**prompt**(*text*, *key=None*, *default=''*, *validate=None*)

Prompt user with text and return the input (like raw_input).

A single space character will be appended for convenience, but nothing else. Thus, you may want to end your prompt text with a question mark or a colon, e.g. prompt("What hostname?").

If key is given, the user's input will be stored as env.<key> in addition to being returned by prompt. If the key already existed in env, its value will be overwritten and a warning printed to the user.

If default is given, it is displayed in square brackets and used if the user enters nothing (i.e. presses Enter without entering any text). default defaults to the empty string. If non-empty, a space will be appended, so that a call such as prompt("What hostname?", default="foo") would result in a prompt of What hostname? [foo] (with a trailing space after the [foo].)

The optional keyword argument validate may be a callable or a string:

- If a callable, it is called with the user's input, and should return the value to be stored on success. On failure, it should raise an exception with an exception message, which will be printed to the user.
- If a string, the value passed to validate is used as a regular expression. It is thus recommended to use raw strings in this case. Note that the regular expression, if it is not fully matching (bounded by ^ and $) it will be made so. In other words, the input must fully match the regex.

Either way, prompt will re-prompt until validation passes (or the user hits Ctrl-C).

Note: prompt honors *env.abort_on_prompts* and will call abort instead of prompting if that flag is set to True. If you want to block on user input regardless, try wrapping with settings.

Examples:

```
    # Simplest form:
    environment = prompt('Please specify target environment: ')

    # With default, and storing as env.dish:
    prompt('Specify favorite dish: ', 'dish', default='spam & eggs')

    # With validation, i.e. requiring integer input:
    prompt('Please specify process nice level: ', key='nice', validate=int)

    # With validation against a regular expression:
    release = prompt('Please supply a release name',
            validate=r'^\w+-\d+(\.\d+)?$')

    # Prompt regardless of the global abort-on-prompts setting:
    with settings(abort_on_prompts=False):
        prompt('I seriously need an answer on this! ')
```

fabric.operations.**put**(**args*, ***kwargs*)

Upload one or more files to a remote host.

`put` returns an iterable containing the absolute file paths of all remote files uploaded. This iterable also exhibits a `.failed` attribute containing any local file paths which failed to upload (and may thus be used as a boolean test.) You may also check `.succeeded` which is equivalent to `not .failed`.

`local_path` may be a relative or absolute local file or directory path, and may contain shell-style wildcards, as understood by the Python `glob` module (give `use_glob=False` to disable this behavior). Tilde expansion (as implemented by `os.path.expanduser`) is also performed.

`local_path` may alternately be a file-like object, such as the result of `open('path')` or a `StringIO` instance.

Note: In this case, `put` will attempt to read the entire contents of the file-like object by rewinding it using `seek` (and will use `tell` afterwards to preserve the previous file position).

`remote_path` may also be a relative or absolute location, but applied to the remote host. Relative paths are relative to the remote user's home directory, but tilde expansion (e.g. `~/.ssh/`) will also be performed if necessary.

An empty string, in either path argument, will be replaced by the appropriate end's current working directory.

While the SFTP protocol (which `put` uses) has no direct ability to upload files to locations not owned by the connecting user, you may specify `use_sudo=True` to work around this. When set, this setting causes `put` to upload the local files to a temporary location on the remote end (defaults to remote user's `$HOME`; this may be overridden via `temp_dir`), and then use `sudo` to move them to `remote_path`.

In some use cases, it is desirable to force a newly uploaded file to match the mode of its local counterpart (such as when uploading executable scripts). To do this, specify `mirror_local_mode=True`.

Alternately, you may use the `mode` kwarg to specify an exact mode, in the same vein as `os.chmod` or the Unix `chmod` command.

`put` will honor `cd`, so relative values in `remote_path` will be prepended by the current remote working directory, if applicable. Thus, for example, the below snippet would attempt to upload to `/tmp/files/test.txt` instead of `~/files/test.txt`:

```
with cd('/tmp'):
    put('/path/to/local/test.txt', 'files')
```

Use of `lcd` will affect `local_path` in the same manner.

Examples:

```
put('bin/project.zip', '/tmp/project.zip')
put('*.py', 'cgi-bin/')
put('index.html', 'index.html', mode=0755)
```

Note: If a file-like object such as StringIO has a `name` attribute, that will be used in Fabric's printed output instead of the default `<file obj>`

Changed in version 1.0: Now honors the remote working directory as manipulated by `cd`, and the local working directory as manipulated by `lcd`.

Changed in version 1.0: Now allows file-like objects in the `local_path` argument.

Changed in version 1.0: Directories may be specified in the `local_path` argument and will trigger recursive uploads.

Changed in version 1.0: Return value is now an iterable of uploaded remote file paths which also exhibits the `.failed` and `.succeeded` attributes.

Changed in version 1.5: Allow a `name` attribute on file-like objects for log output

Changed in version 1.7: Added `use_glob` option to allow disabling of globbing.

`fabric.operations.`**`reboot`**(**args*, ***kwargs*)

Reboot the remote system.

Will temporarily tweak Fabric's reconnection settings (*timeout* and *connection_attempts*) to ensure that reconnection does not give up for at least `wait` seconds.

Note: As of Fabric 1.4, the ability to reconnect partway through a session no longer requires use of internal APIs. While we are not officially deprecating this function, adding more features to it will not be a priority.

Users who want greater control are encouraged to check out this function's (6 lines long, well commented) source code and write their own adaptation using different timeout/attempt values or additional logic.

New in version 0.9.2.

Changed in version 1.4: Changed the `wait` kwarg to be optional, and refactored to leverage the new reconnection functionality; it may not actually have to wait for `wait` seconds before reconnecting.

`fabric.operations.`**`require`**(**keys*, ***kwargs*)

Check for given keys in the shared environment dict and abort if not found.

Positional arguments should be strings signifying what env vars should be checked for. If any of the given arguments do not exist, Fabric will abort execution and print the names of the missing keys.

The optional keyword argument `used_for` may be a string, which will be printed in the error output to inform users why this requirement is in place. `used_for` is printed as part of a string similar to:

```
"Th(is|ese) variable(s) (are|is) used for %s"
```

so format it appropriately.

The optional keyword argument `provided_by` may be a list of functions or function names or a single function or function name which the user should be able to execute in order to set the key or keys; it will be included in the error output if requirements are not met.

Note: it is assumed that the keyword arguments apply to all given keys as a group. If you feel the need to specify more than one `used_for`, for example, you should break your logic into multiple calls to `require()`.

Changed in version 1.1: Allow iterable `provided_by` values instead of just single values.

fabric.operations.**run**(**args*, ***kwargs*)

Run a shell command on a remote host.

If `shell` is True (the default), `run` will execute the given command string via a shell interpreter, the value of which may be controlled by setting `env.shell` (defaulting to something similar to `/bin/bash -l -c "<command>"`.) Any double-quote (`"`) or dollar-sign (`$`) characters in `command` will be automatically escaped when `shell` is True.

`run` will return the result of the remote program's stdout as a single (likely multiline) string. This string will exhibit `failed` and `succeeded` boolean attributes specifying whether the command failed or succeeded, and will also include the return code as the `return_code` attribute. Furthermore, it includes a copy of the requested & actual command strings executed, as `.command` and `.real_command`, respectively.

Any text entered in your local terminal will be forwarded to the remote program as it runs, thus allowing you to interact with password or other prompts naturally. For more on how this works, see *Interaction with remote programs*.

You may pass `pty=False` to forego creation of a pseudo-terminal on the remote end in case the presence of one causes problems for the command in question. However, this will force Fabric itself to echo any and all input you type while the command is running, including sensitive passwords. (With `pty=True`, the remote pseudo-terminal will echo for you, and will intelligently handle password-style prompts.) See *Pseudo-terminals* for details.

Similarly, if you need to programmatically examine the stderr stream of the remote program (exhibited as the `stderr` attribute on this function's return value), you may set `combine_stderr=False`. Doing so has a high chance of causing garbled output to appear on your terminal (though the resulting strings returned by `run` will be properly separated). For more info, please read *Combining stdout and stderr*.

To ignore non-zero return codes, specify `warn_only=True`. To both ignore non-zero return codes *and* force a command to run silently, specify `quiet=True`.

To override which local streams are used to display remote stdout and/or stderr, specify `stdout` or `stderr`. (By default, the regular `sys.stdout` and `sys.stderr` Python stream objects are used.)

For example, `run("command", stderr=sys.stdout)` would print the remote standard error to the local standard out, while preserving it as its own distinct attribute on the return value (as per above.) Alternately, you could even provide your own stream objects or loggers, e.g. `myout = StringIO(); run("command", stdout=myout)`.

If you want an exception raised when the remote program takes too long to run, specify `timeout=N` where `N` is an integer number of seconds, after which to time out. This will cause `run` to raise a `CommandTimeout` exception.

If you want to disable Fabric's automatic attempts at escaping quotes, dollar signs etc., specify `shell_escape=False`.

Examples:

```
run("ls /var/www/")
run("ls /home/myuser", shell=False)
output = run('ls /var/www/site1')
run("take_a_long_time", timeout=5)
```

New in version 1.0: The `succeeded` and `stderr` return value attributes, the `combine_stderr` kwarg, and interactive behavior.

Changed in version 1.0: The default value of `pty` is now `True`.

Changed in version 1.0.2: The default value of `combine_stderr` is now `None` instead of `True`. However, the default *behavior* is unchanged, as the global setting is still `True`.

New in version 1.5: The `quiet`, `warn_only`, `stdout` and `stderr` kwargs.

New in version 1.5: The return value attributes `.command` and `.real_command`.

New in version 1.6: The `timeout` argument.

New in version 1.7: The `shell_escape` argument.

`fabric.operations.`**`sudo`**(**args*, ***kwargs*)
Run a shell command on a remote host, with superuser privileges.

`sudo` is identical in every way to `run`, except that it will always wrap the given `command` in a call to the `sudo` program to provide superuser privileges.

`sudo` accepts additional `user` and `group` arguments, which are passed to `sudo` and allow you to run as some user and/or group other than root. On most systems, the `sudo` program can take a string username/group or an integer userid/groupid (uid/gid); `user` and `group` may likewise be strings or integers.

You may set *env.sudo_user* at module level or via `settings` if you want multiple `sudo` calls to have the same `user` value. An explicit `user` argument will, of course, override this global setting.

Examples:

```
sudo("~/install_script.py")
sudo("mkdir /var/www/new_docroot", user="www-data")
sudo("ls /home/jdoe", user=1001)
result = sudo("ls /tmp/")
with settings(sudo_user='mysql'):
    sudo("whoami") # prints 'mysql'
```

Changed in version 1.0: See the changed and added notes for `run`.

Changed in version 1.5: Now honors *env.sudo_user*.

New in version 1.5: The `quiet`, `warn_only`, `stdout` and `stderr` kwargs.

New in version 1.5: The return value attributes `.command` and `.real_command`.

New in version 1.7: The `shell_escape` argument.

3.1.7 Tasks

class `fabric.tasks.`**`Task`**(*alias=None*, *aliases=None*, *default=False*, *name=None*, **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.

For details on how to implement and use `Task` subclasses, please see the usage documentation on *new-style tasks*.

New in version 1.1.

`__weakref__`
list of weak references to the object (if defined)

`get_hosts`(*arg_hosts*, *arg_roles*, *arg_exclude_hosts*, *env=None*)
Return the host list the given task should be using.

See *How host lists are constructed* for detailed documentation on how host lists are set.

class `fabric.tasks.`**`WrappedCallableTask`**(*callable*, **args*, ***kwargs*)
Wraps a given callable transparently, while marking it as a valid Task.

Generally used via `task` and not directly.

New in version 1.1.

See also:

unwrap_tasks, task

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

Execute task (callable or name), honoring host/role decorators, etc.

task may be an actual callable object, or it may be a registered task name, which is used to look up a callable just as if the name had been given on the command line (including *namespaced tasks*, e.g. "deploy.migrate".

The task will then be executed once per host in its host list, which is (again) assembled in the same manner as CLI-specified tasks: drawing from *-H*, *env.hosts*, the hosts or roles decorators, and so forth.

host, hosts, role, roles and exclude_hosts kwargs will be stripped out of the final call, and used to set the task's host list, as if they had been specified on the command line like e.g. fab taskname:host=hostname.

Any other arguments or keyword arguments will be passed verbatim into task (the function itself – not the @task decorator wrapping your function!) when it is called, so execute(mytask, 'arg1', kwarg1='value') will (once per host) invoke mytask('arg1', kwarg1='value').

Returns

a dictionary mapping host strings to the given task's return value for that host's execution run. For example, execute(foo, hosts=['a', 'b']) might return {'a': None, 'b': 'bar'} if foo returned nothing on host a but returned 'bar' on host b.

In situations where a task execution fails for a given host but overall progress does not abort (such as when *env.skip_bad_hosts* is True) the return value for that host will be the error object or message.

See also:

The execute usage docs, for an expanded explanation and some examples.

New in version 1.3.

Changed in version 1.4: Added the return value mapping; previously this function had no defined return value.

fabric.tasks.**requires_parallel**(*task*)

Returns True if given task should be run in parallel mode.

Specifically:

- It's been explicitly marked with @parallel, or:
- It's *not* been explicitly marked with @serial *and* the global parallel option (env.parallel) is set to True.

3.1.8 Utils

Internal subroutines for e.g. aborting execution with an error message, or performing indenting on multiline output.

fabric.utils.**abort**(*msg*)

Abort execution, print msg to stderr and exit with error status (1.)

This function currently makes use of sys.exit, which raises SystemExit. Therefore, it's possible to detect and recover from inner calls to abort by using except SystemExit or similar.

fabric.utils.**error**(*message*, *func=None*, *exception=None*, *stdout=None*, *stderr=None*)

Call func with given error message.

If func is None (the default), the value of env.warn_only determines whether to call abort or warn.

If `exception` is given, it is inspected to get a string message, which is printed alongside the user-generated `message`.

If `stdout` and/or `stderr` are given, they are assumed to be strings to be printed.

`fabric.utils.`**`fastprint`**(*text*, *show_prefix=False*, *end=''*, *flush=True*)
Print `text` immediately, without any prefix or line ending.

This function is simply an alias of `puts` with different default argument values, such that the `text` is printed without any embellishment and immediately flushed.

It is useful for any situation where you wish to print text which might otherwise get buffered by Python's output buffering (such as within a processor intensive `for` loop). Since such use cases typically also require a lack of line endings (such as printing a series of dots to signify progress) it also omits the traditional newline by default.

Note: Since `fastprint` calls `puts`, it is likewise subject to the `user` *output level*.

New in version 0.9.2.

See also:

`puts`

`fabric.utils.`**`indent`**(*text*, *spaces=4*, *strip=False*)
Return `text` indented by the given number of spaces.

If text is not a string, it is assumed to be a list of lines and will be joined by `\n` prior to indenting.

When `strip` is `True`, a minimum amount of whitespace is removed from the left-hand side of the given string (so that relative indents are preserved, but otherwise things are left-stripped). This allows you to effectively "normalize" any previous indentation for some inputs.

`fabric.utils.`**`puts`**(*text*, *show_prefix=None*, *end='\n'*, *flush=False*)
An alias for `print` whose output is managed by Fabric's output controls.

In other words, this function simply prints to `sys.stdout`, but will hide its output if the `user` *output level* is set to `False`.

If `show_prefix=False`, `puts` will omit the leading `[hostname]` which it tacks on by default. (It will also omit this prefix if `env.host_string` is empty.)

Newlines may be disabled by setting `end` to the empty string (`''`). (This intentionally mirrors Python 3's `print` syntax.)

You may force output flushing (e.g. to bypass output buffering) by setting `flush=True`.

New in version 0.9.2.

See also:

`fastprint`

`fabric.utils.`**`warn`**(*msg*)
Print warning message, but do not abort execution.

This function honors Fabric's *output controls* and will print the given `msg` to stderr, provided that the `warnings` output level (which is active by default) is turned on.

3.2 Contrib API

Fabric's **contrib** package contains commonly useful tools (often merged in from user fabfiles) for tasks such as user I/O, modifying remote files, and so forth. While the core API is likely to remain small and relatively unchanged over

time, this contrib section will grow and evolve (while trying to remain backwards-compatible) as more use-cases are solved and added.

3.2.1 Console Output Utilities

Console/terminal user interface functionality.

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

Ask user a yes/no question and return their response as True or False.

`question` should be a simple, grammatically complete question such as "Do you wish to continue?", and will have a string similar to " [Y/n] " appended automatically. This function will *not* append a question mark for you.

By default, when the user presses Enter without typing anything, "yes" is assumed. This can be changed by specifying `default=False`.

3.2.2 Django Integration

New in version 0.9.2.

These functions streamline the process of initializing Django's settings module environment variable. Once this is done, your fabfile may import from your Django project, or Django itself, without requiring the use of `manage.py` plugins or having to set the environment variable yourself every time you use your fabfile.

Currently, these functions only allow Fabric to interact with local-to-your-fabfile Django installations. This is not as limiting as it sounds; for example, you can use Fabric as a remote "build" tool as well as using it locally. Imagine the following fabfile:

```
from fabric.api import run, local, hosts, cd
from fabric.contrib import django

django.project('myproject')
from myproject.myapp.models import MyModel

def print_instances():
    for instance in MyModel.objects.all():
        print(instance)

@hosts('production-server')
def print_production_instances():
    with cd('/path/to/myproject'):
        run('fab print_instances')
```

With Fabric installed on both ends, you could execute `print_production_instances` locally, which would trigger `print_instances` on the production server – which would then be interacting with your production Django database.

As another example, if your local and remote settings are similar, you can use it to obtain e.g. your database settings, and then use those when executing a remote (non-Fabric) command. This would allow you some degree of freedom even if Fabric is only installed locally:

```
from fabric.api import run
from fabric.contrib import django

django.settings_module('myproject.settings')
from django.conf import settings
```

```
def dump_production_database():
    run('mysqldump -u %s -p=%s %s > /tmp/prod-db.sql' % (
        settings.DATABASE_USER,
        settings.DATABASE_PASSWORD,
        settings.DATABASE_NAME
    ))
```

The above snippet will work if run from a local, development environment, again provided your local `settings.py` mirrors your remote one in terms of database connection info.

`fabric.contrib.django.`**`project`**(*name*)
Sets `DJANGO_SETTINGS_MODULE` to `'<name>.settings'`.

This function provides a handy shortcut for the common case where one is using the Django default naming convention for their settings file and location.

Uses `settings_module` – see its documentation for details on why and how to use this functionality.

`fabric.contrib.django.`**`settings_module`**(*module*)
Set `DJANGO_SETTINGS_MODULE` shell environment variable to `module`.

Due to how Django works, imports from Django or a Django project will fail unless the shell environment variable `DJANGO_SETTINGS_MODULE` is correctly set (see the Django settings docs.)

This function provides a shortcut for doing so; call it near the top of your fabfile or Fabric-using code, after which point any Django imports should work correctly.

Note: This function sets a **shell** environment variable (via `os.environ`) and is unrelated to Fabric's own internal "env" variables.

3.2.3 File and Directory Management

Module providing easy API for working with remote files and folders.

`fabric.contrib.files.`**`append`**(*filename*, *text*, *use_sudo=False*, *partial=False*, *escape=True*, *shell=False*)
Append string (or list of strings) `text` to `filename`.

When a list is given, each string inside is handled independently (but in the order given.)

If `text` is already found in `filename`, the append is not run, and None is returned immediately. Otherwise, the given text is appended to the end of the given `filename` via e.g. `echo '$text' >> $filename`.

The test for whether `text` already exists defaults to a full line match, e.g. `^<text>$`, as this seems to be the most sensible approach for the "append lines to a file" use case. You may override this and force partial searching (e.g. `^<text>`) by specifying `partial=True`.

Because `text` is single-quoted, single quotes will be transparently backslash-escaped. This can be disabled with `escape=False`.

If `use_sudo` is True, will use `sudo` instead of `run`.

The `shell` argument will be eventually passed to `run/sudo`. See description of the same argumnet in `~fabric.contrib.sed` for details.

Changed in version 0.9.1: Added the `partial` keyword argument.

Changed in version 1.0: Swapped the order of the `filename` and `text` arguments to be consistent with other functions in this module.

Changed in version 1.0: Changed default value of `partial` kwarg to be `False`.

Changed in version 1.4: Updated the regular expression related escaping to try and solve various corner cases.

New in version 1.6: Added the `shell` keyword argument.

`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.

This function uses the `sed` function, and will accept the same `use_sudo`, `shell` and `backup` keyword arguments that `sed` does.

`comment` will prepend the comment character to the beginning of the line, so that lines end up looking like so:

```
this line is uncommented
#this line is commented
#   this line is indented and commented
```

In other words, comment characters will not "follow" indentation as they sometimes do when inserted by hand. Neither will they have a trailing space unless you specify e.g. `char='# '`.

Note: In order to preserve the line being commented out, this function will wrap your `regex` argument in parentheses, so you don't need to. It will ensure that any preceding/trailing `^` or `$` characters are correctly moved outside the parentheses. For example, calling `comment(filename, r'^foo$')` will result in a `sed` call with the "before" regex of `r'^(foo)$'` (and the "after" regex, naturally, of `r'#\1'`.)

New in version 1.5: Added the `shell` keyword 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). Specify `exact=True` to change this behavior so that only a line containing exactly `text` results in a True return value.

This function leverages `egrep` on the remote end (so it may not follow Python regular expression syntax perfectly), and skips `env.shell` wrapper by default.

If `use_sudo` is True, will use `sudo` instead of `run`.

If `escape` is False, no extra regular expression related escaping is performed (this includes overriding `exact` so that no `^`/`$` is added.)

The `shell` argument will be eventually passed to `run`/`sudo`. See description of the same argumnet in `~fabric.contrib.sed` for details.

Changed in version 1.0: Swapped the order of the `filename` and `text` arguments to be consistent with other functions in this module.

Changed in version 1.4: Updated the regular expression related escaping to try and solve various corner cases.

Changed in version 1.4: Added `escape` keyword argument.

New in version 1.6: Added the `shell` keyword argument.

`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<backup> -r -e "/<limit>/ s/<before>/<after>/<flags>g" <filename>. 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.

If use_sudo is True, will use sudo instead of run.

The shell argument will be eventually passed to run/sudo. It defaults to False in order to avoid problems with many nested levels of quotes and backslashes. However, setting it to True may help when using ~fabric.operations.cd to wrap explicit or implicit sudo calls. (cd by it's nature is a shell built-in, not a standalone command, so it should be called within a shell.)

Other options may be specified with sed-compatible regex flags – for example, to make the search and replace case insensitive, specify flags="i". The g flag is always specified regardless, so you do not need to remember to include it when overriding this parameter.

New in version 1.1: The flags parameter.

New in version 1.6: Added the shell keyword argument.

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.)

Changed in version 1.6: Added the shell keyword argument.

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*)
Render and upload a template text file to a remote host.

Returns the result of the inner call to put – see its documentation for details.

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.

The resulting rendered file will be uploaded to the remote file path `destination`. If the destination file already exists, it will be renamed with a `.bak` extension unless `backup=False` is specified.

By default, the file will be copied to `destination` as the logged-in user; specify `use_sudo=True` to use `sudo` instead.

The `mirror_local_mode` and `mode` kwargs are passed directly to an internal `put` call; please see its documentation for details on these two options.

Changed in version 1.1: Added the `backup`, `mirror_local_mode` and `mode` kwargs.

3.2.4 Project Tools

Useful non-core functionality, e.g. functions composing multiple operations.

`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.

This function makes use of Fabric's `local()` operation, and returns the output of that function call; thus it will return the stdout, if any, of the resultant `rsync` call.

`rsync_project()` takes the following parameters:

- `remote_dir`: the only required parameter, this is the path to the directory on the remote server. Due to how `rsync` is implemented, the exact behavior depends on the value of `local_dir`:
  - If `local_dir` ends with a trailing slash, the files will be dropped inside of `remote_dir`. E.g. `rsync_project("/home/username/project", "foldername/")` will drop the contents of `foldername` inside of `/home/username/project`.
  - If `local_dir` does **not** end with a trailing slash (and this includes the default scenario, when `local_dir` is not specified), `remote_dir` is effectively the "parent" directory, and a new directory named after `local_dir` will be created inside of it. So `rsync_project("/home/username", "foldername")` would create a new directory `/home/username/foldername` (if needed) and place the files there.
- `local_dir`: by default, `rsync_project` uses your current working directory as the source directory. This may be overridden by specifying `local_dir`, which is a string passed verbatim to `rsync`, and thus may be a single directory (`"my_directory"`) or multiple directories (`"dir1 dir2"`). See the `rsync` documentation for details.
- `exclude`: optional, may be a single string, or an iterable of strings, and is used to pass one or more `--exclude` options to `rsync`.
- `delete`: a boolean controlling whether `rsync`'s `--delete` option is used. If True, instructs `rsync` to remove remote files that no longer exist locally. Defaults to False.
- `extra_opts`: an optional, arbitrary string which you may use to pass custom arguments or options to `rsync`.

- `ssh_opts`: Like `extra_opts` but specifically for the SSH options string (rsync's `--rsh` flag.)
- `capture`: Sent directly into an inner `local` call.
- `upload`: a boolean controlling whether file synchronization is performed up or downstream. Upstream by default.
- `default_opts`: the default rsync options `-pthrvz`, override if desired (e.g. to remove verbosity, etc).

Furthermore, this function transparently honors Fabric's port and SSH key settings. Calling this function when the current host string contains a nonstandard port, or when `env.key_filename` is non-empty, will use the specified port and/or SSH key filename(s).

For reference, the approximate `rsync` command-line call that is constructed by this function is the following:

```
rsync [--delete] [--exclude exclude[0][, --exclude[1][, ...]]] \
    [default_opts] [extra_opts] <local_dir> <host_string>:<remote_dir>
```

New in version 1.4.0: The `ssh_opts` keyword argument.

New in version 1.4.1: The `capture` keyword argument.

New in version 1.8.0: The `default_opts` keyword argument.

`fabric.contrib.project.`**`upload_project`**(*local_dir=None*, *remote_dir=''*, *use_sudo=False*)

Upload the current project to a remote system via `tar`/`gzip`.

`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.

This function makes use of the `tar` and `gzip` programs/libraries, thus it will not work too well on Win32 systems unless one is using Cygwin or something similar. It will attempt to clean up the local and remote tarfiles when it finishes executing, even in the event of a failure.

Changed in version 1.1: Added the `local_dir` and `remote_dir` kwargs.

Changed in version 1.7: Added the `use_sudo` kwarg.

f